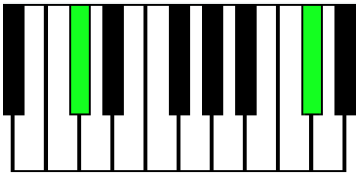


Think Sharply with C#

or

How to Think Like a Computer Scientist

(last revised December 2015)

Peter Wentworth

p.wentworth@ru.ac.za

The downloadable versions of this book are at http://www.ict.ru.ac.za/resources/ThinkSharply

Foreword

- Chapter 1 *Introduction*
- Chapter 2 *Visual Studio Survival Guide*
- Chapter 3 *Start with a Window*
- Chapter 4 *Code-behind: Events and Handlers*
- Chapter 5 *Diving into Code*
- Chapter 6 *Nesting Things in Other Things*
- Chapter 7 *Hello, Little Turtles!*
- Chapter 8 *Void Methods*
- Chapter 9 *Working with Booleans and Conditional Statements*
- Chapter 10 *Value-returning Methods*
- Chapter 11 *Iteration*
- Chapter 12 *Strings*
- Chapter 13 *Classes and Objects — an Overview*
- Chapter 14 *Arrays*
- Chapter 15 *Lists*
- Chapter 16 *More Event Handling*
- Chapter 17 *Odds and Ends*
- Chapter 18 *I/O, Files and Networks*
- Chapter 19 *List Algorithms*
- Chapter 20 *The N-Queens Puzzle — a Case Study*
- Chapter 21 *Recursion*
- Chapter 22 *Exceptions*
- Chapter 23 *The .NET Framework*
- Chapter 24 *Scope and Lifetime*
- Chapter 25 *GUIs for our Queens*
- Chapter 26 *Writing our own Classes*
- Chapter 27 *In the Caves — a Case Study*
- Chapter 28 *Inheritance*

- Chapter 29 *Dictionaries*
- Chapter 30 *Interfaces*

Appendices

- A Few Tips
- ThinkLib.Turtle Documentation
- ThinkLib.Tester Documentation
- Getting Started with ThinkLib

Search Page

Foreword

Thanks

This is an Open Source textbook

That means that anyone can use it non-commercially, or modify it, extend it, translate it into other languages, or even make a movie of its exciting plot. However, any use must stick to the legal fine-print at http://creativecommons.org/licenses/by-nc-sa/3.0/

The first chapter is an extract from a tutorial by Yusuf Motara. Find the full tutorial at http://www.ict.ru.ac.za/resources/way_of_the_program/. He based some of it on an earlier version of a Python book at http://openbookproject.net/thinkcs/python/english3e/, and that was based on yet an earlier version by different authors. And even though Yusuf wrote most of the text in this chapter, it was subsequently modified by the current author, Peter Wentworth, who bears ultimate responsibility for any errors, omissions, or mistakes...

Why?

As an educator, the book tries to do a few things that are different from many of the usual approaches.

I'm trying to move gently towards the idea of Computation-as-Interaction, rather than Computation-as-Calculation. These days we interact with computational artefacts through GUIs. Notions of events and responses are more prevalent than console-based top-to-bottom calculation. Many of our students have never seen a command-line program, and perhaps some never will. Lynn Andrea Stein's *Rethinking CS101* project has been influential, see http://www.cs101.org/

In recent times the theory of *threshold concepts* and *troublesome concepts* has become a fashionable and useful lens for understanding why students have difficulty transitioning to newer ways of thinking about a subject. Although there probably won't be widespread agreement soon about what exactly the threshold concepts are for our students, the book does isolate and emphasize some concepts like state, flow of control, recursion, representation, data transformation, mental chunking, and so on. If you've not encountered this idea yet, see http://crpit.com/confpapers/CRPITV95Rountree.pdf or looking for Threshold Concepts on Google will do the trick.

I like visual examples. Some of our students, sadly, often seem very weak in the cognitive areas of what Piaget theory refers to as "Formal Operations" and the ability to abstract, manipulate, reason about, and control the internal, non-visible state of the program. This means they're more comfortable with more concrete approaches.

So where possible, I prefer the state of the computation to be explicitly visible. To this end I use a Logo-like turtle, and I use the graphical controls that are available in Windows Presentation Foundation quite heavily. And when we first tackle recursion, I go for turtle-based fractal curves because of this explicit visibility of the internal unfolding of the algorithm.

Our literature abounds too with studies that show that students who are left to infer and construct their own internal cognitive models of computation get it wrong much of the time. So they end up with non-viable models of even the most fundamental ideas, like how assignment works. (Non-viable means their conceptualization may work in some situations, but it gives wrong answers in other cases.) So I put a

strong emphasis on debugging, single-stepping, inspecting the internal state of the computation as it runs, etc. in the hope that we can build accurate internal mental models.

Ask yourself or your teaching peers to take their best guess as to what percentage of students typically come out of a year-long programming course with a viable understanding of value assignment, or with a viable understanding of reference assignment. See how that compares with the study at http://dl.acm.org/citation.cfm?id=1227481

And then, of course, we must face the question: "Do we do objects right up front, early, or do we delay them while we try to build the algorithmic and coding skills in a procedural way?" I advocate early *exposure* to objects, terminology, and use-cases, but late *synthesis* of objects. A GUI-builder and some turtle examples seem ideal for this. We can work with objects from a library, and build familiarity with the OOP concepts like instances, properties, internal state, lifetimes, events, inheritance and so on way before we get into "Now synthesize your own objects". On the same theme, I have a distaste for introductory approaches that use very trivial (or minimalist) classes. Nor do I like OOP examples that give rise to just a single instance. Here I find students struggle to separate the idea of the class as a factory, and the instances that it produces.

So from the first time students see the Button type, this book will have at least two instances, or at least two turtles. When we do inheritance, I prefer inheriting from something that already has significant complexity and functionality. (In this book we inherit the Turtle type and give it some extra and some overriding behaviour.)

There is also a tendency in the book to often introduce concepts "informally" the first time, just because we need it for a compelling current example. Then later we'll hit the chapters where we revisit and cover the concept more fully. So you might bump your head against a *while* loop in Chapter 7, and some arrays in Chapter 8, even though the chapters on loops and arrays only come later in the book. I hope this provides an early motivating context for the construct or for the feature. It also seems useful to be able to tell students "Sure, we haven't done that. So go see what you can find out about it on your own."

The Tips appendix in this book also explains some of my thinking around teaching Computer Science: I like to cover the appendix material explicitly with my students.

Enjoy the book.

Peter Wentworth, January 2015.

1. Introduction

The goal of these notes is to help you to think like a computer scientist. This way of thinking combines some of the best features of mathematics, engineering, natural science, philosophy and art. If you like any of those, you might find that you like Computer Science, too. Most of all, Computer Science is challenging and fun, and a good computer scientist is always surprised that people are willing to pay him or her to work on interesting problems and have fun all day!

The single most important skill for a new computer scientist is **problem-solving**. Problem-solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills. Programming is the process of translating problem-solving into a language that the computer can understand.

There is a bigger picture ...

We're at university to toughen up our thinking skills. We understand things not only by remembering facts, but by mentally organizing the ideas, and relating them to things we already know.

New university students often have trouble. Here is a quote from page 9 of a document [1] written by those who've done a careful study of where the weaknesses are:

> *"Entering university students therefore have limited ability to solve unseen problems, to apply their knowledge flexibly and appropriately in varied contexts, to demonstrate a range of cognitive skills, to extract and integrate salient information in order to create their own, meaningful knowledge and to monitor their own learning."*

This here are some of the things we'll really need if we're going to be successful in our studies:

- to be able to extract and pick the salient (relevant, or important) facts and ideas,
- to relate these new ideas and facts to our existing knowledge,
- to spend time thinking about how we learn best and where our strengths are,
- to understand the ideas in such a way that we can apply them to unseen problems and use them flexibly!

So let's get into some of the ideas from Computer Science, while we do the tough mental work of making sense of them and relating them all together!

[1] http://www.che.ac.za/sites/default/files/publications/QEP%20Framework%20Feb%202014.pdf

1.1. What is a program?

A **program** contains instructions that tell the computer what to do. These instructions are given according to the rules of your **programming language**, and are sometimes called **code**. Programming languages are designed for humans, but the computer can only follow a different set of instructions that are less convenient for humans — it has its own internal language. So after writing a program, you can tell the computer to **compile** your program; it checks whether your instructions make sense and translates each of your programming language instructions into its internal language that it can follow. If

Programming styles

There are many different programming languages and many different ways of programming, just as there are many different human languages and many different ways of speaking. C# is a "high-level garbage-collected statically-typed object-oriented C-family" language. You'll discover what this means as you progress through Computer Science and

everything you wrote makes sense to the computer, you can ask the computer to follow the instructions (this is called **executing** or **running** the program). We also say that instructions are **executed**.

The programming language you will be learning is called C# (pronounced "C sharp"). Some other popular computer languages you might have heard about are Java, Python, C, or Pascal.

learn languages which aren't like C#. For now, be aware that what you're learning is only the smallest fraction of the vast field of Computer Science – but everyone has to start somewhere, and here is as good a place as anywhere else!

Programming languages such as C# are extremely *unforgiving*. If a single letter or punctuation-mark of the program is out-of-place or missing, the program will not compile. Many programming languages are so strict that they don't even allow you to alter the case of a letter – so if you type `If` instead of `if`, your program will probably fail to compile! Beginning programmers make many simple mistakes because they aren't precise enough. With practice and caution, you will make fewer and fewer of these simple mistakes, until you eventually make no simple mistakes whatsoever.

1.2. What kind of programs will we start with?

Most modern programs have three main parts to them. Let us think about a program that sends SMS messages. It could look like this:

- Part 1: The **GUI** — the Graphical User Interface is the part that determines what it looks like on our screen or on our device (phone, tablet, XBox, TV). The interface will contain buttons, sliders, text boxes, check boxes and so on. These parts that make up our GUI are called *controls*, because they let the user control what work the program does. (We'll learn more about controls soon.) Part of the GUI designer's job is to choose nice colours, text sizes, choose the fonts, and to lay out the different controls so that they are nicely organized, and the user of your program can see what needs to be done.

  The GUI or interface is sometimes called the **front-end** of the application.

- Part 2: The **code-behind** is the programming logic that carries out the tasks in response to what the user does. So, for example, when the "Send" button is clicked, there has to be some hand-written code (that we'll soon learn to write in C#) to make the useful work happen. In this SMS

example, the code-behind would probably check that the cellphone number had the correct number of digits, it might remove unnecessary spaces from whatever the user has typed in, it could check that the user did not type in any invalid characters (like xyz) in the number, it could check the length of the message, make sure the message was not empty. Then, if everything seemed okay, the program might use the third part to actually send the message.

- Part 3: The **back-end** part of an application (`application` is another name for a program) is responsible for long-term storage and other operations, like sending the SMS messages to the mobile network. So if we're busy with on-line banking or we're active on facebook, or we're checking our available airtime, there will be some part of the application (often running remotely on another machine) that does this back-end work for us.

GUIs are often designed by well-paid graphic designers with good artistic taste and great colour sense. Our SMS example is not very pretty, which says something about the textbook author.

The user will interact with our program via its GUI: they will enter text, click buttons and drag things around. Actions that the user takes cause some **events** to happen. Those events will cause some code in the code-behind part of our application to be executed. And that is where we'll do our magic that will make our program useful.

We'll start off by learning to design and build a few GUIs. Once we can build simple GUIs, we'll move along to learning about the events that happen, and we'll write some code-behind. In this book none of our programs will be complex enough to have a third back-end part.

1.3. How do we get started?

We need a tool that allows us to design our GUIs, write some program code, and run and test our programs. The tool we're going to use in this book is one designed exactly for this purpose: Visual Studio, a tool provided by Microsoft Corporation. So finding out a little about how the tool works is our next major objective, starting in the next chapter.

1.4. Glossary

In any new subject, the special terminology and words are important. We've summarized the main terms here. If one doesn't know what the words mean, one can't follow the ideas. So it is worth spending some time getting a really good understanding of what each word means.

application

Another name for a program.

back-end

One of the three parts of a modern application. The back-end often runs remotely (at our bank, or on facebook). See also *front-end* and *code-behind*.

code-behind

The part of an application that responds to what the user does and carries out the useful work of the program. See also *back-end* and *front-end*.

event

See the definition in the glossary of Chapter 3.

front-end

Another name for the GUI or interface to an application. See also *back-end* and *code-behind*.

GUI

A Graphical User Interface. This is the part of your application that the user can interact with, by entering text, clicking buttons, setting sliders, etc.

1.5. Exercises

1. Become familiar with a couple of computer or smartphone applications like a web browser, Google search, a chat application, email, a facebook application, or a game like Sudoku.

   Identify each of the different controls (buttons, check boxes, places to type text, etc.) that the application offers the user.

   When the application does something (like make the next move in a game, or display another web page, or check that we've entered some information correctly), can you figure out which things are happening on your own computer or phone, and which things are being done remotely on some back-end computer?

2. In your notebook, write down the most important ideas from this chapter, in just one line each. (Trying to get an idea down into one sentence or one short line is an important part of organizing it well in our minds.)

2. The Visual Studio Survival Guide

We'll jump straight in.

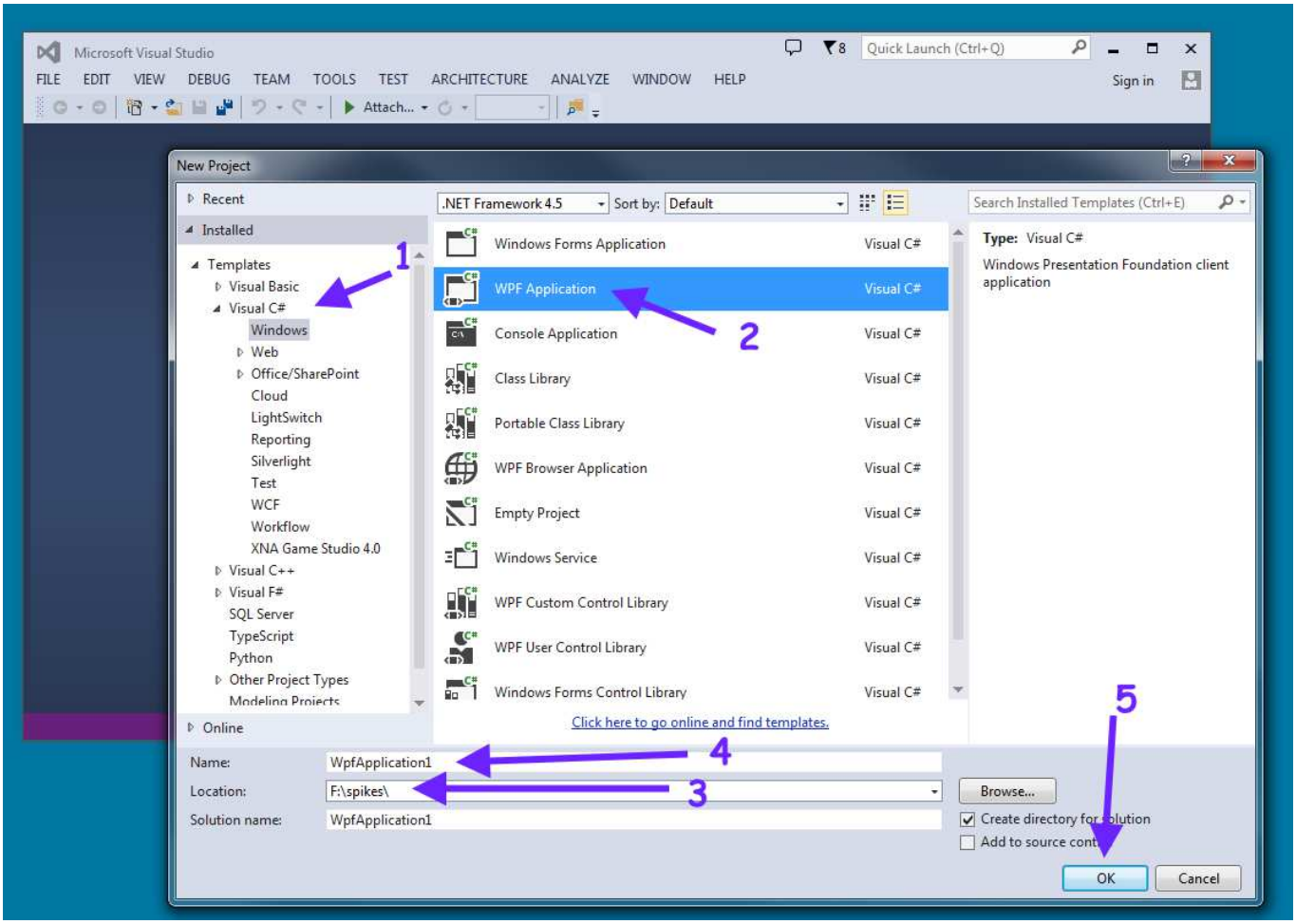
We'll assume we have Visual Studio 2013 (VS) installed and working. There are a few different versions of VS: the one we used is called Visual Studio 2013 Ultimate. But if we don't want to buy one, we should use the free VS Community 2013 Edition from https://www.dreamspark.com/Product/Product.aspx?productid=89.

If we use a different version of VS a few things may look slightly different, but we'll still be able to do all the samples in this textbook. The high-end commercial versions have extra features, options, and languages that we don't need here.

2.1. Our first "Hello, World!" program:

We start Visual Studio, select the menu items **File** | **New** | **Project** (we use this notation to mean "click on the **File** menu (top left corner), then open the **New** menu, then click on the **Project** item").

The **New Project** window will open, and our screen should look similar to this:

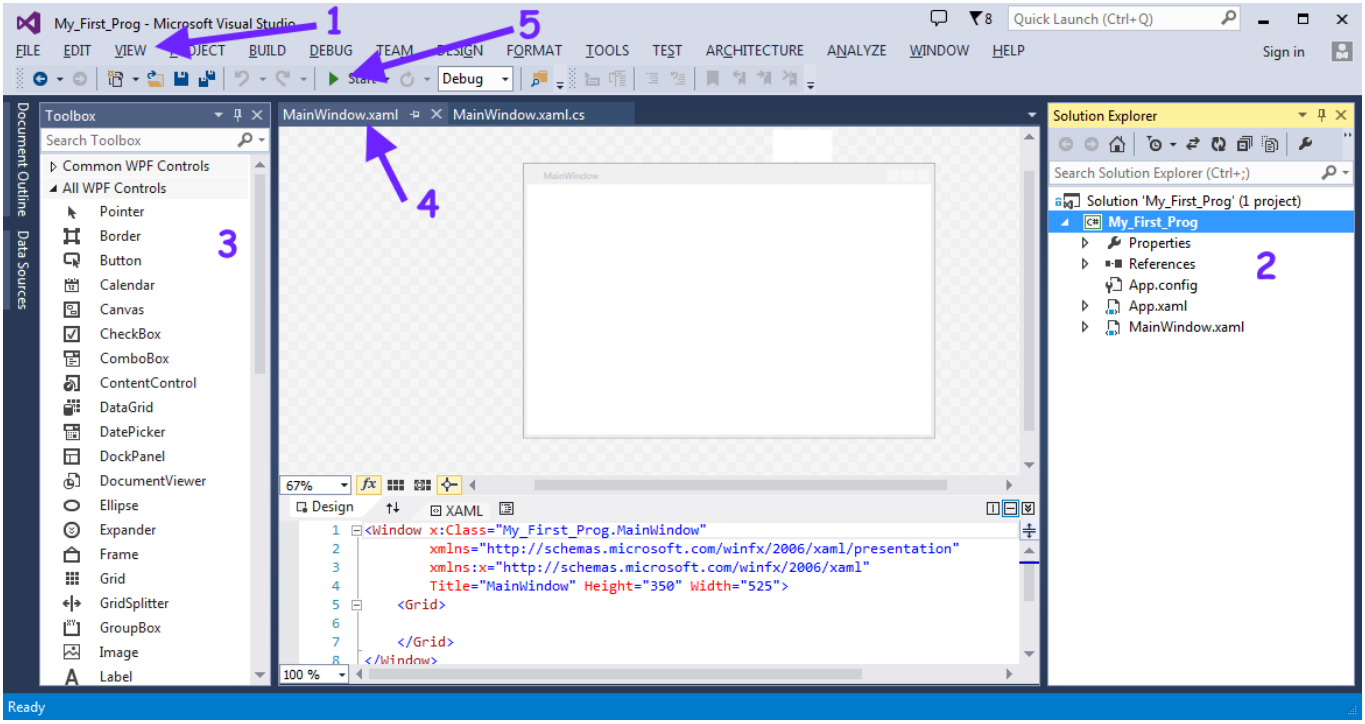


Now we need 5 steps to create our new project:

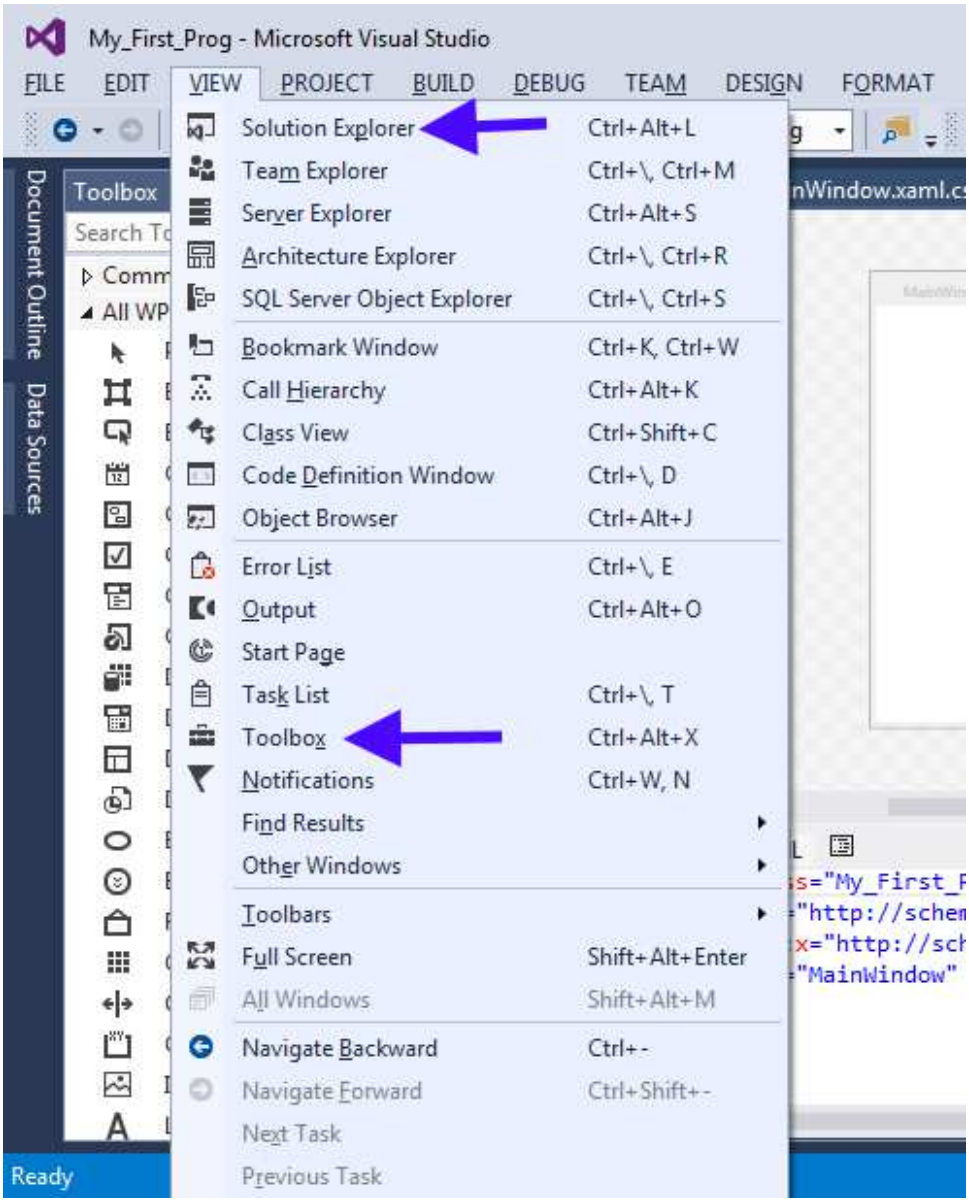
1. Make sure we are creating a C# program for Windows.
2. We choose to create a WPF (Windows Presentation Foundation) flavour of program.
3. We select a folder or directory where we want our program to be created and saved.

4. We choose a nice name for our program. Always start the name with a letter! Don't leave spaces in the name, or use any special characters other than the digits 0,1,2,3,4,5,6,7,8,9, or an underscore (_). So we'll call our first program something like **My_First_Prog**. We type this into the **Name** field.
5. We're ready for some magic. We click the OK button, and Visual Studio will get to work and will give us a skeleton of our first program.

We should now see this:



Perhaps the most important menu for our initial survival is the View menu:



If we don't have our *Solution Explorer* window showing, (the window labelled 2) we can make it show up from the View menu. The Solution Explorer is the "gateway" that allows us to get to our work.

If we don't have our **Toolbox** showing, (labelled 3), the View menu has an option again. (It looks like a toolbox, to remind us that these are our tools!)

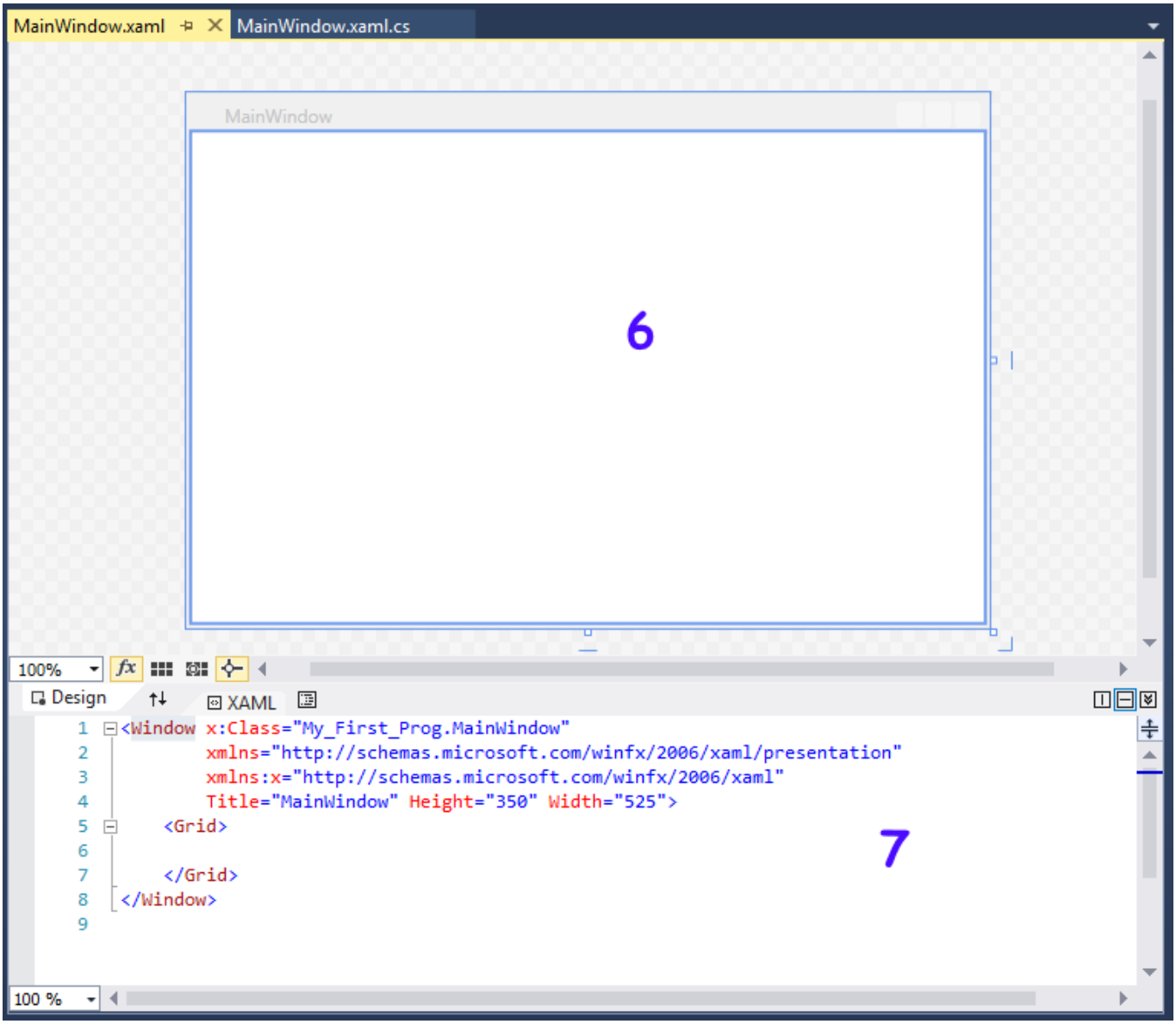
Practice opening and closing the Solution Explorer and the Toolbox. If we are feeling adventurous, we can also drag any window to another part of the screen or even onto a second screen, or get it completely detached from Visual Studio.

There are also some interesting little "pins" next to the close button of each window: they "pin" the window so that it remains open. If we unpin a window we'll get an even more powerful way to organize everything on our desktop! Try it!

There are two other important labels on the image above: label 4 shows what we call a "tab control" — a way to have multiple windows stacked behind each other. We have two windows open in our tab control, the one called "MainWindow.xaml" is at the front, and "MainWindow.xaml.cs" is behind it. We can click on the tabs in the tab control to bring its window to the front.

And finally, label 5 shows where to find the "Start" button. It starts running the program we're about to write.

Let's take a closer look at MainWindow.xaml:

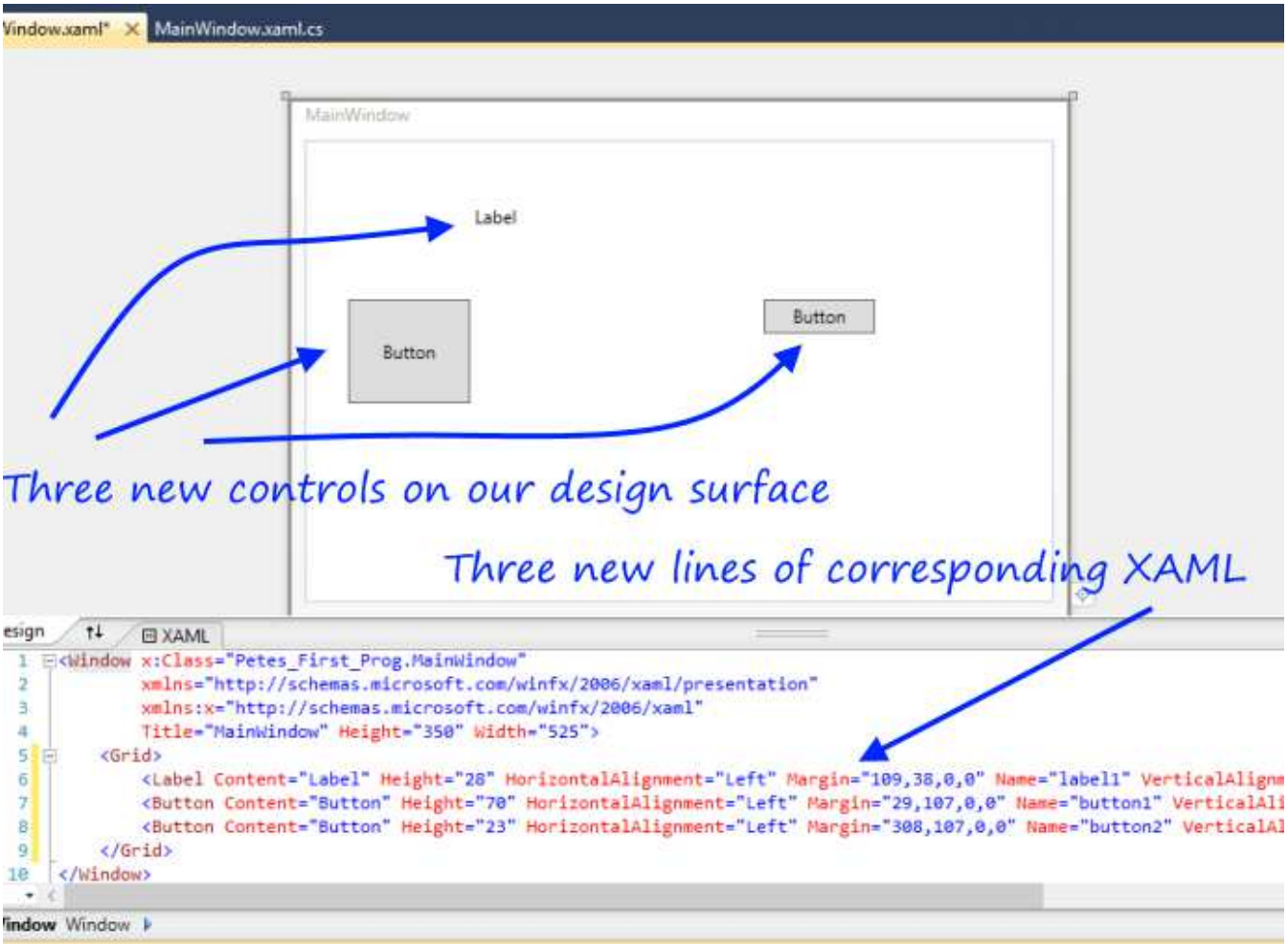


This is all one window with two sub-windows. It provides a "design surface", like a blank sheet of paper, that allows us to design what we want our program's GUI to look like.

It has two "views" of the same information: the Design view, labelled 6, lets us see our design. The window labelled 7 is an ugly, not-really-intended-for-humans description of our design, called the XAML (say it as "zammel", like "camel").

2.2. Designing our GUI

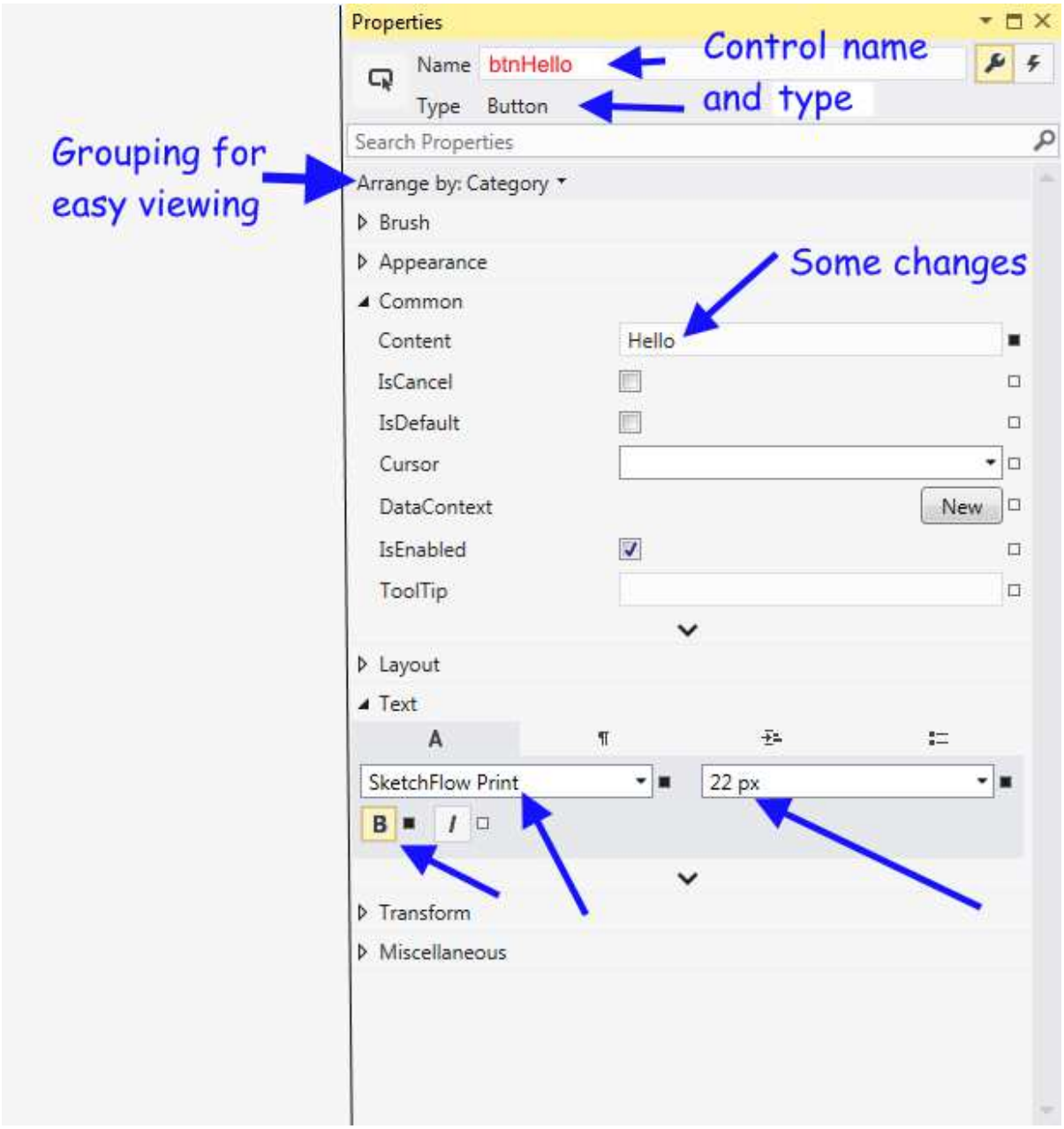
The Toolbox provides a collection of **controls** (see the definition in the glossary at the end of this chapter) that can be added to our main window. If we drag and drop any control, say the **Button**, from the Toolbox to our main window, it will create a new button in our main window. Then we can grab the newly created button, and move it (or resize it). Similarly, we can create a new label as part of our design by dragging a **Label** control from the tollbox. Here is what we'll have after putting one label and two buttons on our window, and moving them around and resizing them a bit:



We notice that as Design view changes when we add new things, the XAML also changes. They are two views, or ways of looking at the same thing. We can experiment by dragging or resizing the big button to a new location, and seeing how the XAML changes in response. Or, we can go into the XAML and change the string that gets displayed in the control. For example, if we change the XAML corresponding to the big button (the one with `Height="70"`) to `Content="Hello"`, we'll see the new text displayed on the face of the button in the Design view too.

So one way of changing controls is by dragging and moving them in the designer. A second way is by hand-editing the ugly XAML. But there is a third way too, one that we prefer to use. The View menu allows us to open a **Properties** window to change properties. (Pressing the key F4 is a short-cut method of opening the Properties window.) Whichever control is currently selected (click on it, or click on its XAML) will have its own properties displayed in the Properties window.

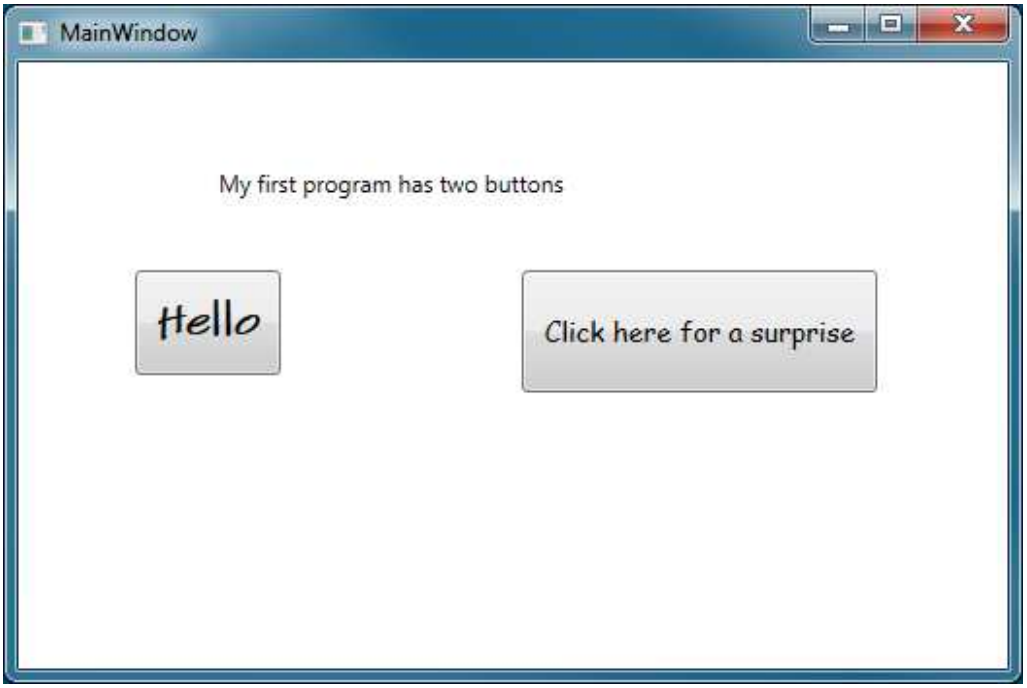
Some users prefer to work with the Properties window "detached from the others" so that as the properties are changed they can also see the Design view (and the XAML) changing at the same time. The Property window gives us an easier way to change the **properties** of any control. So after a few changes, it could look something like this:



There are more than 60 properties that can be set on a button. Here they are arranged by category, so everything to do with text properties is in one place, and everything to do with brushes for the colours is grouped together.

Here we've changed five properties of our button: the Control's *name*, at the top, in red. (We're going to use this name later when we write some code-behind.) The Content is what gets displayed on the button. And we've changed the Font, the Font Size, and we've clicked the Bold button to make the font bold. Each change here also shows up as a change in the XAML, and we can also see the changes in the Design view.

Once we've changed what we want to for this button, we can close its property window, go back to our Main Window. Now we can repeat what we did for the second button: open its property window, change some of its properties, and close the property window. Finally, we can also change some properties on the label. We'll now have our Design window looking like this:



2.3. Adding some code to our program

When our program is running, we expect that the user will click one of the buttons. This causes an **event** from the button. We now need to write our first bit of C# *handler* code that will *respond* to the event.

In the Design view, double-click on the "Hello" button. Visual Studio now writes some skeleton handler code for us. It also changes the XAML for the button so that the button click event becomes "tied to" the new handler code.

We'll add a line to the skeleton handler that VS has written for us. So when we've typed that line in, we'll have this fragment of code, and some other code that we'll ignore for the moment, in the file too.

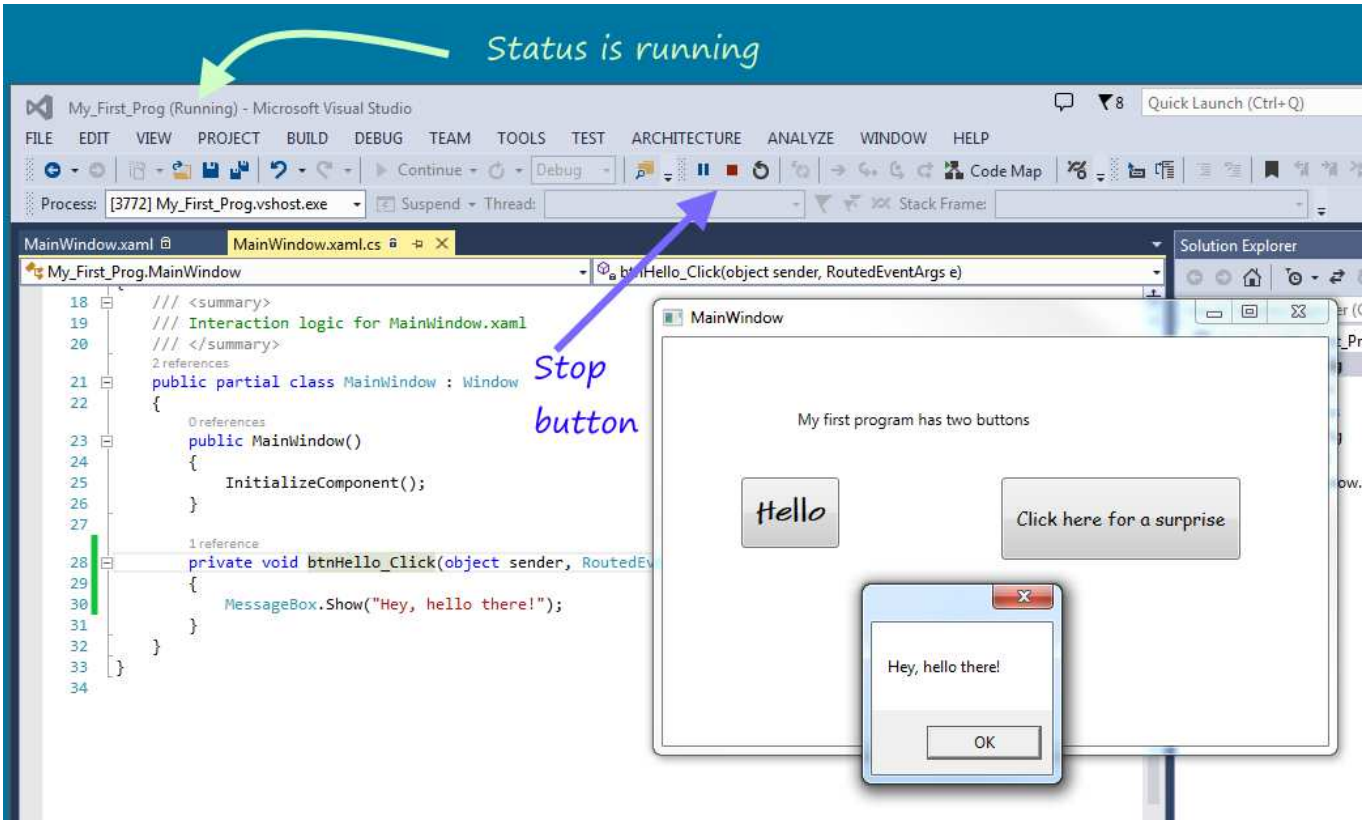
```
1  private void btnHello_Click(object sender, RoutedEventArgs e)
2  {
3      MessageBox.Show("Hey, hello there!");
4  }
```

Lines 1,2 and 4 are the skeleton of the handler (provided for free by Visual Studio). But line 3 we'll have to type by hand. Be careful with all the syntax and the capitalization — it must be precisely as shown.

2.4. Now we can run our first program!

In Visual Studio, click the run button. Notice a few things.

- Our new program starts running and opens our newly designed GUI with a title bar that says MainWindow.
- Our program's GUI window opens up in front of Visual Studio.
- If we click our button labelled *Hello*, a message box will pop up with our message.
- We can close the message box and try clicking the button again. (What do we think will happen?)
- While our program is running, the Visual Studio title bar (at the top of the main VS window) displays the state as "(Running)".
- And Visual Studio also gives us an extra button to Stop our program.



We can stop our program by closing its MainWindow. (So the program stops itself.) Or we can use the Visual Studio stop button to force it to stop. Forcing a program to stop (by using Visual Studio) is useful when we write an incorrect program that cannot correctly stop itself.

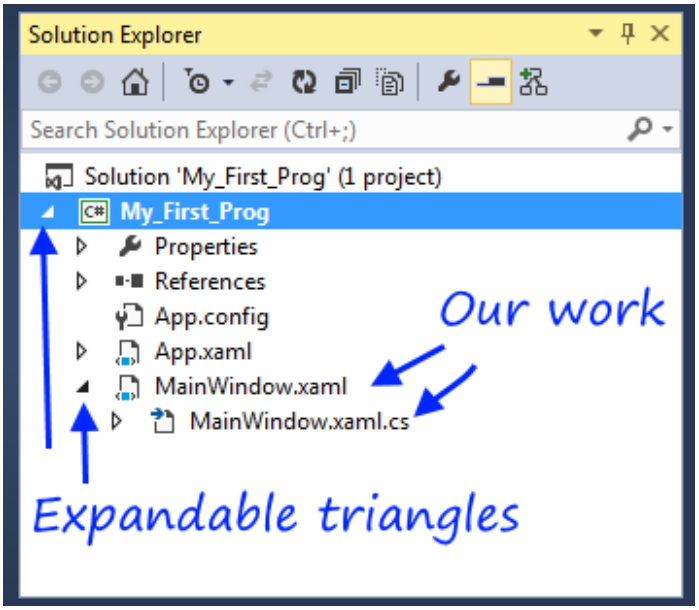
2.5. Your turn

Stop the program, go back and add a second handler for the other button, do something similar, and make it work so that the user can click either button and get a different response from our program.

2.6. Hey, my work has gone missing!

Close the two windows (MainWindow.xaml) and (MainWindow.xaml.cs). Now we're stuck. How can we get them back?

That's where the Solution Explorer window is so useful. Make sure it is open (Section 2.1 tells you what to do if it is not open). Notice the little expandable triangle icons occur all over the place: they always mean that there is some sub-detail that can be shown (or hidden) if we click on them. So let us get our Solution Explorer looking like this:



From there, we can double-click on our working files to open them again.

2.7. Mental workout: Connecting the dots

It is important to understand that the toolbox offers you a number of different *types* of controls for building your GUI — Button, Label, etc. When we drag a button onto our design window, we're making a new *instance* of that type of thing. So we had two button instances and one label instance in our first program.

One button instance can display some text, another might be a different size, at a different position, and might show something different on the button face. And we've already seen that each button can have its own "handler" code, and can do something different when clicked.

We'll see this idea often. We have a *type* of building block, but we can make as many different instances as we need to. This is an idea we see all around us everyday. For example, the Samsung S5 Mini is a type of phone. There are many instances, each with their own address book, their own ring tones, and so on.

A good hint for becoming a good Computer Scientist (any kind of scientist, really) is to always look for opportunities to join the dots. Can you relate some seemingly new idea to other ideas that you already know and understand. How is this new situation similar, how is it different?

2.8. Key survival skills and concepts that we'll need

- How to start up Visual Studio, choose the right language and flavour of program, and build a simple program.
- Understand *design time* versus *run time*. How to know which state Visual Studio is in. (We'll watch the VS title bar if we are unsure).
- How to run our program.
- How to stop a running program, even if it doesn't want to stop itself!
- How to find and open the different VS sub-windows after they have been closed or hidden. (Solution Explorer, Toolbox, our MainWindow, the code associated with the main window)
- We'll need some familiarity with the Toolbox of available *control* types, like Button, and Label.
- How to create different instances of any controls on our MainWindow GUI.
- The idea that each instance has its own properties.

- How to use the Properties window to edit the properties of any instance.
- How to name each instance uniquely.
- The idea that we've seen three different ways to change properties: we can move or resize things in the designer view; or we can use the Properties window to make changes; or we can change the XAML. It doesn't matter which method we use: so pick the one that works best for you.

2.9. Glossary

container control

A control that can have children controls. It allows us to group, organize, and lay out the children controls. The Grid is a container control. Visual Studio starts off a new WPF application by giving us a blank Grid to begin with. (See line 5 of the XAML code in the screen shot above: there is a Grid instance that contains everything else.)

control

Any of the visual components or building blocks that we can use to design our window. Some examples are buttons, sliders, menus, panels, labels, images and text boxes.

design time

The time when we are designing our GUI window, or writing code-behind for our application. See *run time*.

properties window

One of the Visual Studio windows that lets us edit (change) properties of our control instances.

property

Each instance of a control (say a button instance) has properties such as the content that it displays; its size; its position; its background colour; the font it uses for text; the font size, etc.

run time

The time when our program is running. See *design time*.

toolbox

One of the Visual Studio sub-windows that shows the control types that we can use when designing our own WPF flavour of GUI applications.

Visual Studio

A software product from Microsoft that helps us design applications that start with a main GUI window which can contain other visual components like buttons, sliders and images.

WPF

Windows Presentation Foundation (WPF) is a set of tools (many of them found in the toolbox) that allows us to build applications with a GUI window for interaction with our users.

WPF application

A WPF application (which is the kind we're building here) is an application built using the WPF tools. It will always open a main GUI window when it starts running.

2.10. Exercises

1. Connect the dots ... Find your favourite music or video player application, and check out what icons (little shapes or symbols) it uses for Play, Pause and Stop. Then check out the Visual Studio buttons for Running, Pausing and Stopping a program, and see if they are the same.

3. Start with a Window

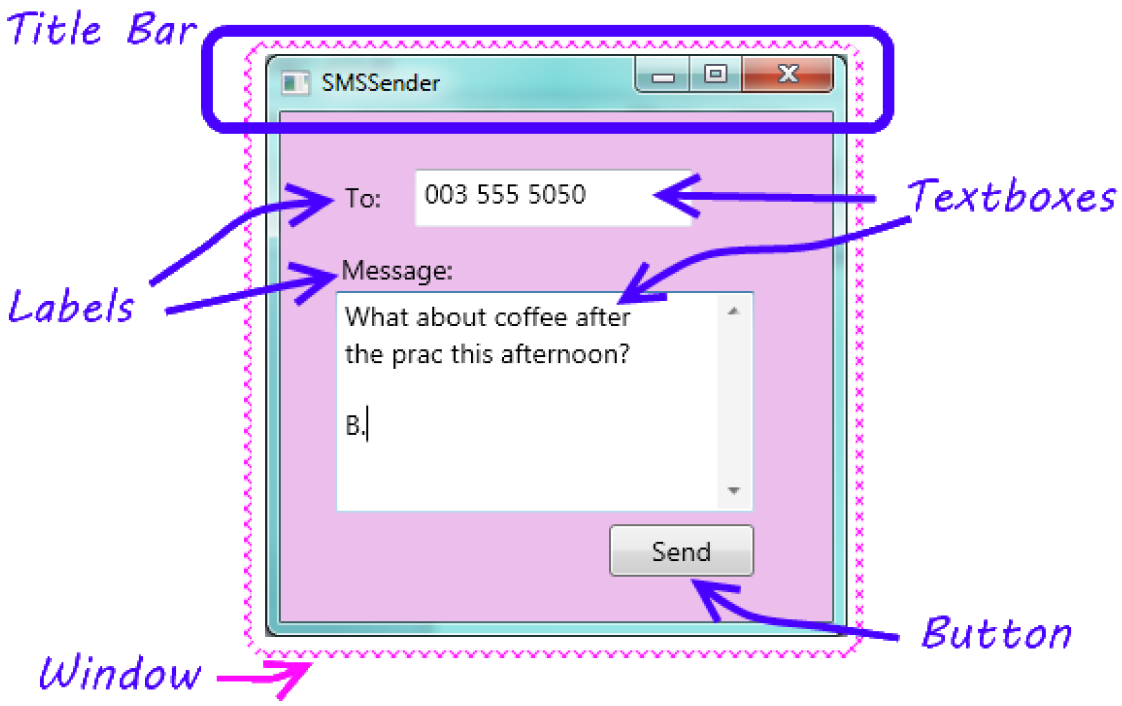
3.1. Do this chapter by hands-on practice

Most of the the ideas in this chapter are best conveyed by hands-on practical demonstrations. So rather than write long wordy text, you should watch out for some short videos that we'll release to cover this content.

3.2. A little more about the GUI

In Chapter 1 we introduced the idea that programs have a GUI, some code-behind, and sometimes a back-end. Let us revisit our SMS application we introduced there.

Our programs will usually have a single window. We'll want to refer to some of the parts in the window, so here is the diagram again, with some terminology added ...



3.3. Separating our two roles in our head

When we write programs, we write them for others to use. So there are two very distinct roles: we design and write our programs, and our users use them.

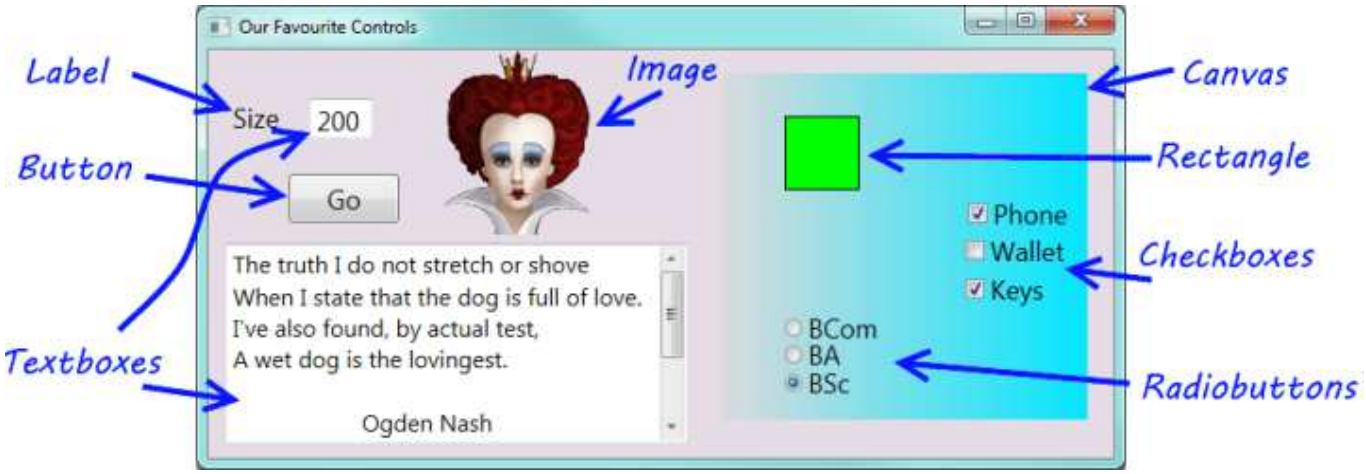
When we start learning to program we will sometimes be programming, and sometimes we'll also be the user. We need to separate these two roles in our heads, and imagine what we want our users to experience, then we can design our interfaces and write our code to give them the desired experience.

We saw in the previous chapter that Visual Studio behaves differently at *design time* and *run time*. At design time, or when we are debugging, we are in the role of a programmer. When we're running our programs, we're in the role of a user.

3.4. Our favourite controls

We saw in the Visual Studio Toolbox that there are a large number of controls that can do very impressive and powerful things: progress bars, web browsers, calendars, and so on. So we could have a whole textbook devoted to describing all the controls and their detail. But we won't do that. Instead, we'll introduce very few controls, just enough to get us comfortable with how the front-end GUI and the code-behind work together.

Here is a GUI showing a few of our favourite controls used in this textbook.



Most of these controls have properties called `Background` and `Foreground` that can set the colours.

For controls that can display text, we can set properties like the `Font` and `FontSize`.

- **Label**: We'll use a label whenever we want some fixed text on the GUI. The most important property is `Text` — the text to be displayed.
- **Button**: A button has a `Content` property. It can display many kinds of things, including text. Buttons are great because they allow the user to click and control our program. In the top right corner of the window you'll find more buttons that belong to the window. These are displaying images rather than text, and the one has a red background colour.
- **TextBox**: A text box is used for displaying text, or for allowing the user to type text. In our example we've used two text boxes: the user has typed the value 200 into one of them, and we've made our program put a poem by Ogden Nash into the second text box.

  The second text box is also interesting because it has multiple lines for displaying its text, and because we've set one of its properties to turn on a vertical scroll bar so that we'll be able to bring the extra text into view.

  The most important property of a text box is `Text` — the text to be displayed.
- **Image**: An image control lets us display a picture, with some options to stretch or crop (cut off the bits that do not fit) the picture to fit the available space. The most important property is the `Source` which tells the control where to find the picture to be displayed.
- **Canvas**: We will use the canvas control extensively starting from our Turtle chapter. A canvas is really cool because it is a *container* that can hold and manage other controls. So we've put a canvas here on our GUI, given it a nice light-to-darker blue gradient colour, and then we've put seven more controls onto the canvas. These are called the *children* of the canvas.

- **Rectangle**: We use these quite a bit in the later chapters of the book when we draw some chess boards.
- **CheckBox**: A check box allows the user to check (or tick) an option. The two important properties are `Content` (same as the button above), and a property called `IsChecked` that allows the program to find out whether the user has ticked the check box or not.
- **RadioButton**: A group of radio buttons act together: only one can be selected at any time. So they provide a nice way to get the user to make a choice, or select one of a small set of options. They too have `Content` and `IsChecked` properties.

3.5. Glossary

container control

A control that can have children controls. It allows us to group, organize, and lay out the children controls. The Canvas is a container. In the last chapter we also saw that a Grid is a container, and can have children.

event

Something external that happens. An SMS arriving at your phone is an event. For GUI applications that we write, events occur when a user clicks a button, or moves a slider, or resizes our window, or moves the mouse, or presses a key on the keyboard. An event can occur on a control — if the `send` button is clicked, the button gets the click event and passes it to our code behind. Events can also occur on our window — e.g. if the window is resized, so the resize event happens to the window.

interface

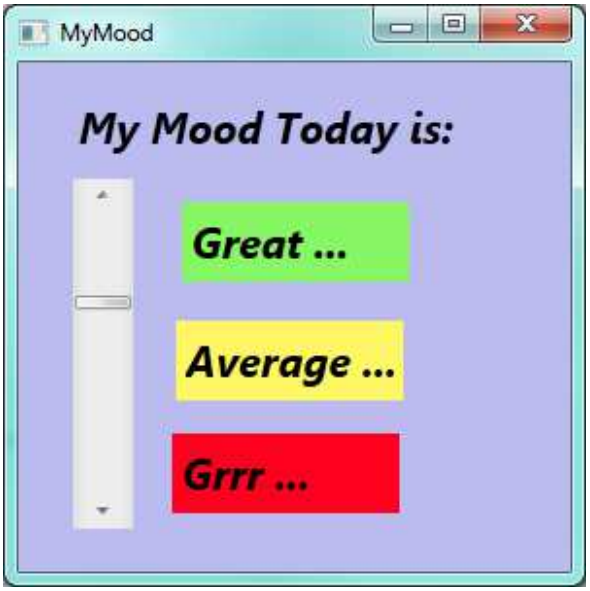
How a system (like an application or a cellphone) presents its features and capabilities to the outside world. More generally, how one system presents its capabilities to another. So a printer presents an interface to a computer.

window

A rectangular area on the screen. A window usually has a title bar at the top giving some description.

3.6. Exercises

1. Design a GUI that lets you show your mood. It won't need any code-behind. So make you GUI look (approximately) like this:



2. Design a GUI that allows you to set an option telling others where you are. It should look something like this:

(Hint: the Radio Buttons are all placed on a Canvas to group them.)

3. Design a GUI that allows the user to choose a meal and a drink. The meal can be one of Hamburger, Pizza, Fish and Chips or Curry. The drinks can be Coke, Fanta, or Tea. The default options when the program runs should be a Hamburger and a Coke.

   (Hint: each independent group of Radio Buttons should be on its own Canvas.)

4. Design an interface that allows the user to select the kinds of movies they like watching. It should look something like this:

(Hint: the Canvas background uses *gradient* colours.)

5. Explain why we chose to use Radio Buttons for Question 2, but we used Check Boxes for Question 4. If we used Check Boxes for Question 2, what would it mean for the user of our program? If we used Radio Buttons for Question 4, what would it mean for the user of our program?
6. Design an interface with two combo box controls that allows a user to enter the month and the year of their birth from the drop-down items. (Hint: the property editor has a very tedious process for achieving this. Add one or two months of the year in the tedious way with the property editor, then inspect your XAML. You should be able to copy, paste and change the XAML much faster.) Here is what the fragment of XAML is going to look like...

```
<ComboBox ...  >
    <ComboBoxItem Content="January" />
    <ComboBoxItem Content="February" />
    <ComboBoxItem Content="March" />
    ...
</ComboBox>
```

7. Design a GUI for sending an SMS. See whether you're able to produce a more tasteful design than the one in our textbook.
8. Experiment and learn how to use one or two controls that we have not covered in this chapter.

4. Code-Behind: Events and Handlers

Once we have designed our GUI, we need to write the code-behind: the part of the program that will respond to events caused by the user.

You really need a bit of theory about program code: types, variables, assignment statements, etc. before we can write code. But we're going to delay talking about that until the next chapter. We want to emphasize first the links between the front-end GUI and the code-behind, and the idea that the program responds to the user's actions. So for this chapter, you might have to copy the code verbatim into your program, and focus more on how the big parts fit together. We'll dive deeper into the concepts of coding quite soon!

There are two tasks we'll need to get a GUI and a its code-behind to cooperate with one another:

- Determine which events the GUI should respond to. Use the GUI designer to "wire them up" so that they trigger the running of some **handler** code-behind,
- Write the guts of the handler code.

We'll sometimes call a handler an **event handler**. We mean the same thing.

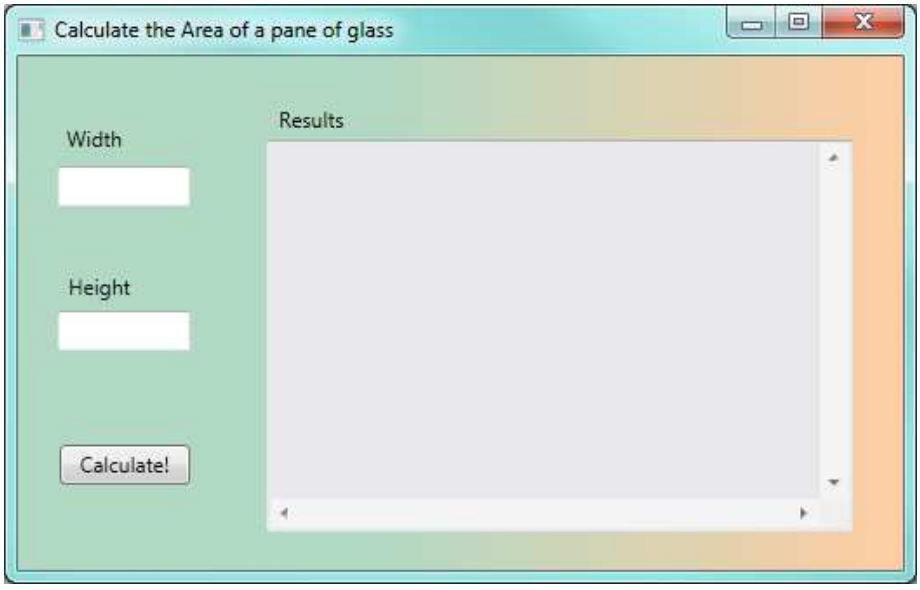
4.1. How to wire up an event so that it triggers execution of our handler code:

The easiest event to start with is a `Click` event on a button. What code do we want to run when the user clicks the button?

Let's assume we need an application that calculate the area of a rectangular pane of glass from its width and height.

The *inputs* to the program are going to be the width and height. The *calculation* will compute width times height. The *output*, or feedback to the user, will be the result of our calculation.

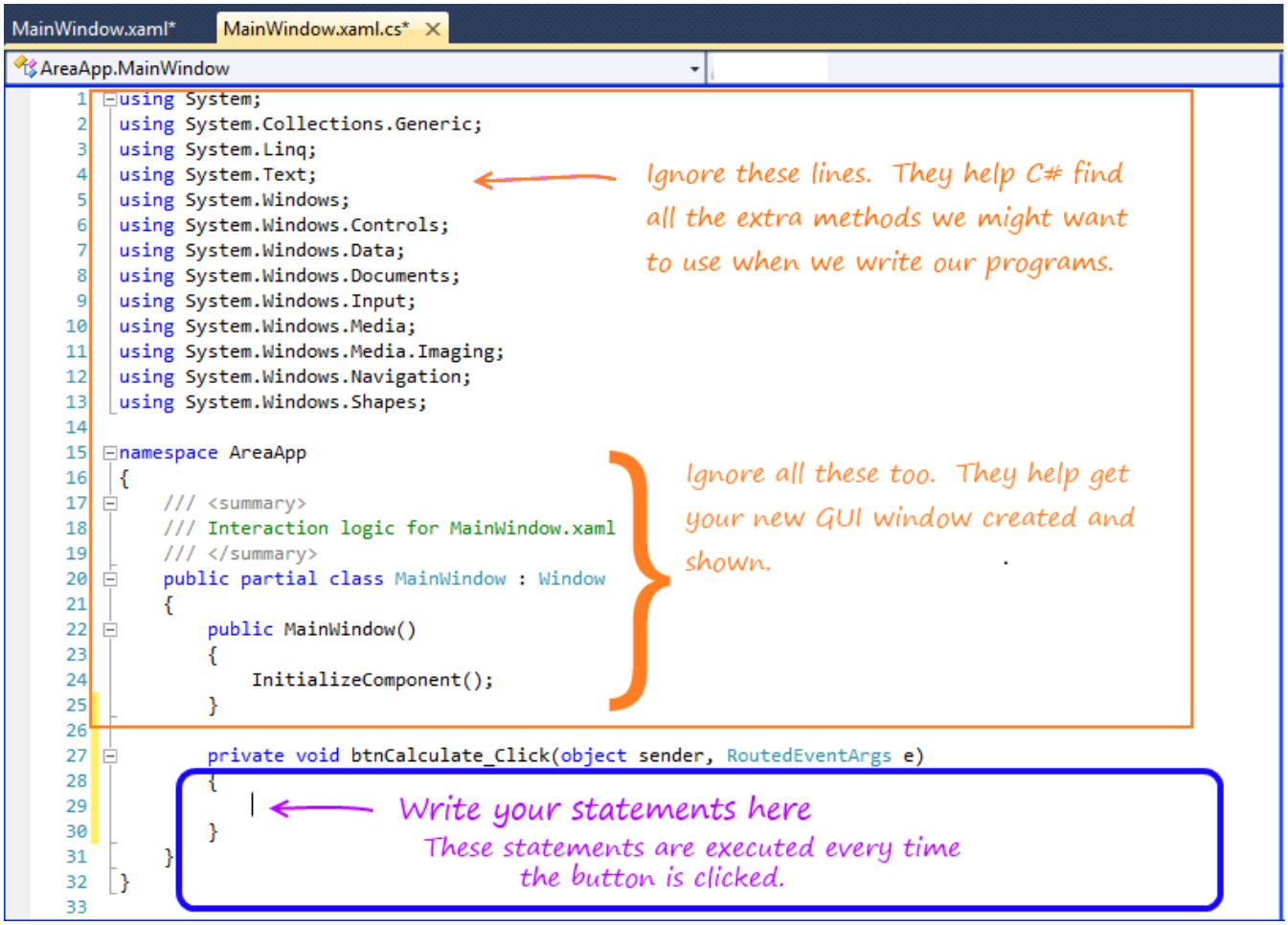
We design our Window to look like this: it has three text box controls, a few labels, and a button to perform the calculations.



When we run the program the user will be able to enter the width and height. But what we want to do next is to respond when the user clicks the `Calculate!` button. In Visual Studio (VS), we can get into the Designer View, and simply double-click on the button. VS then creates a skeleton handler **method** in the code-behind, and changes our view so that we're now viewing the code behind, ready to write our new code. If our button was called `btnCalculate` [1], after double-clicking the button in the designer, VS will show us this code-behind:

[1] If you can't remember how to use the Property Editor to change the name of a control, peek back to Section

2.2.



VS did a lot of (necessary) work, but at this stage the best advice is to ignore most of it. Lines 27–30 is the bit that interests us: — this is the handler method that will be executed each time the button is clicked by the user. We are going to write our code at line 29 (where VS has conveniently positioned our cursor).

Notice that there are two tabs now at the top: one called `MainWindow.xaml`, which is our GUI component of the program, and the active one (shown highlighted), which is the code-behind. Its name (`MainWindow.xaml.cs`) hints that it is the C# code (the .cs part of the name tells us that) behind MainWindow.xaml. You can switch between the two tabs freely. If you close a tab, you can always open it again from the Solution Explorer window in VS.

4.2. How to write some interesting code in the handler:

Before we tackle the real problem, let's play around a bit. At line 29, we can insert a fragment of code like this:

```
1   MessageBox.Show("Pete's first handler!", "It says ...");
```

Now we run our program and click its *Calculate!* button, and a message box pops up like this:

So we've got the program to *do* something in response to our event! This is significant progress! You can close the pop-up message box and next time you click the button you'll get a new event, the handler code will be called again, and a new message box will pop up.

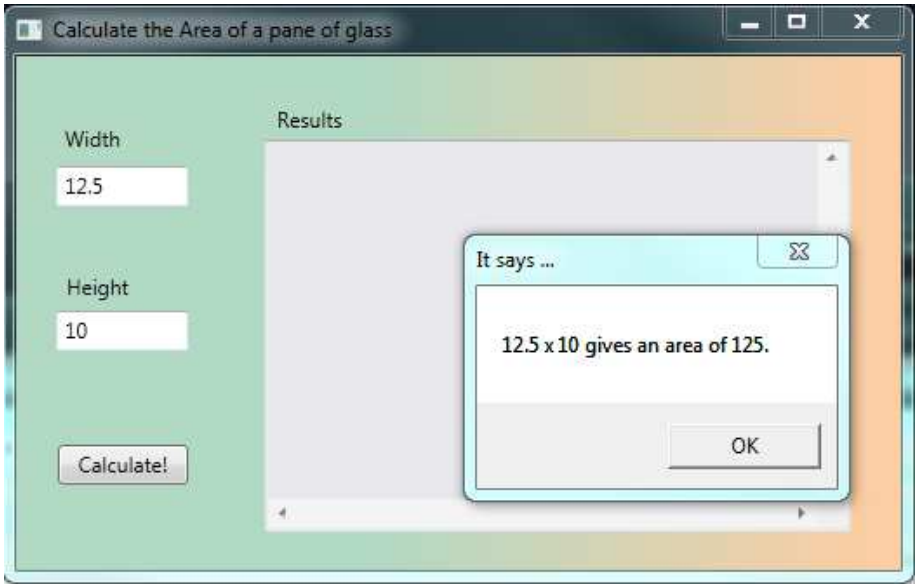
With confidence now that we can get from our GUI to our code-behind, we'll delete the line we've just written and get back to solving the problem of calculating the area.

We expect the user to type some numbers into the two text boxes on the GUI. We named them `txtWidth` and `txtHeight` while we were setting properties in the designer, and we named the big text box `txtResult`. The names are not used while we are designing the GUI, but they are used in the code-behind that we write, where need have to refer to the controls by their names.

So in our handler we have to take the text (a string) in each of the two input text boxes, and convert it into a double number so that we can do arithmetic. Then we'll compute the area (a double), and we'll have to convert that back to a string. Then we can output it. Here is our code for our handler (you may want to just copy and paste this from the on-line textbook into your own code):

```
1  private void btnCalculate_Click(object sender, RoutedEventArgs e)
2  {
3      double w = Convert.ToDouble(txtWidth.Text);  // Convert string in textbox to double.
4      double h = Convert.ToDouble(txtHeight.Text); // Same again.
5      double area = w * h;
6      string result = string.Format(" {0} x {1} gives an area of {2}\n", w, h, area);
7      MessageBox.Show(result, "It says ...");
8  }
```

And here is what gets displayed when we run the program, enter data, and click the button:



There are a few different options for displaying our results once we've calculated them. We've chosen to use a message box here, but in future we'll probably prefer to put the answers into the third big text box that we've got on the screen. We can do this quite easily by changing line 7 in the above code, to this:

```
1      txtResults.AppendText(result);
```

The nice thing about doing it like this is that new results simply get appended to the bottom of previous results: so you can click the `Calculate!` button repeatedly or change the width and height and see all the results appear in your text box. (We even made the vertical and horizontal scroll bars visible for this text box, so if you get enough results you'll be able to scroll through them).

4.3. Console-based programs: Computation as Calculation

In this book we're stressing the idea that modern computing is about *computation as interaction*. Our touch screens, smart phones, tablet devices have controls like scrollbars, buttons, and menus, so *interaction* is currently the big deal.

We've used a GUI-based program to program in this style, and this chapter has shown how we wire up our programs for this way of thinking. In the GUI style, events occur, and these are wired up to trigger execution of the

code in the handlers.

There is another style of programming called *computation as calculation*. Here a program doesn't have a GUI, or events, or handlers.

It often does three steps:

- input some data,
- process the data,
- output the results.

It does the input and output in a **console**, and the flow of logic is very strictly top-to-bottom, pre-determined by the program logic, rather than dictated by when and how the user clicks buttons. We'll call this style of program a **console-based program**.

Fortunately, C# can work well either with GUIs, or using the console (although the console is just represented as a window on the screen these days). Sometimes we can even use a mix of the two styles in the same program.

Let us re-code the above problem about computing the area of glass. This time we'll do it as a Console program. So in Visual Studio, create a new project. But instead of choosing a WPF Application, create a Console Application. VS will write a skeleton like this:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  namespace AreaOfGlass
7  {
8      class Program
9      {
10         static void Main(string[] args)
11         {
12         }
13     }
14 }
```

A teletype console from 1963

Once upon a time all input and output with computers was like this... The image comes from the museum at http://www.fcet.staffs.ac.uk/jdw1/sucfm/sucfmoutpu

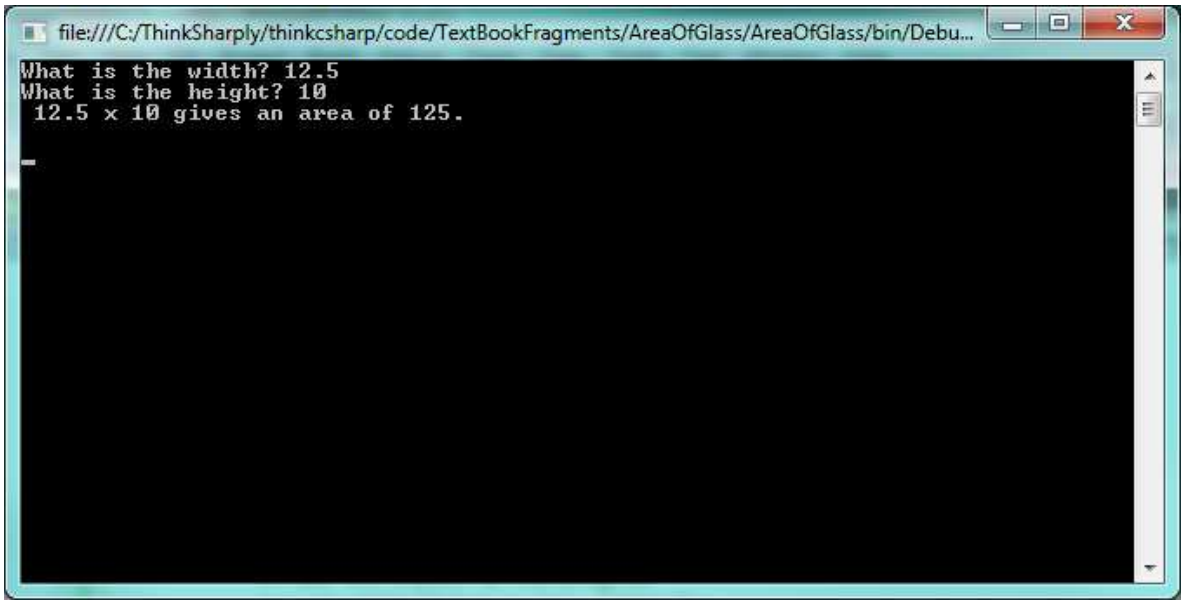
Once again, there is some necessary scaffolding code (much less than the scaffolding for a GUI, though), that we can mainly ignore. What is very different from the WPF flavour of programs is that there is no GUI in a Console-based program.

The program will start executing our code in the body of the Main method, i.e. at line 11. So here is a completed method that you can copy and paste into the scaffolding that VS gave you:

```
1  static void Main(string[] args)
2  {
3      // Get the inputs to the program
4      Console.Write("What is the width? ");
5      string s1 = Console.ReadLine();
6      Console.Write("What is the height? ");
7      string s2 = Console.ReadLine();
8
9      // Do the calculations
10     double w = double.Parse(s1);
11     double h = double.Parse(s2);
12     double area = w * h;
13
14     // Output the results
15     string result = string.Format(" {0} x {1} gives an area of {2}\n", w, h, area);
16     Console.WriteLine(result);
```

```
17
18          // This extra ReadLine keeps the console window open while we admire our results.
19          string dummy = Console.ReadLine();
20      }
```

When we run the program, the console-based interaction looks like this:



You'll find many Console-based examples in the Help system, in textbooks, and in on-line tutorials. Authors sometimes prefer them, because they can demonstrate one small technique without the overhead of having to design a window. The top-to-bottom execution is also a bit easier because we don't need to understand events and handlers. And everything is in one file, rather than having to have a GUI window, and some code-behind in another window.

Of course, one downside to Console programs is that they can only work with text, not graphics, and they don't match our modern experience with our computers, our iPads, and our cellphones.

In this book we'll tend to focus more on GUI-based programs that promote the idea of "computation as interaction".

4.4. Glossary

code-behind

Code that is "attached" to a GUI and provides the logic for the application. See the slightly different definition in the first chapter's glossary too.

console-based program

Alternative to a GUI-based program. A console-based program does its input from and its output to a text-only console window.

GUI-based program

An application that uses a GUI as its interface to the user.

handler

Code that we write to respond to an event.

response

We get an event, and give a response. The handler is the code that determines what our response is.

4.5. Exercises

1. a. Design an application that contains a button with a suitable caption, and a text box (with pretty colours and a nice font!). When the button is clicked, the text in the text box should show the current time of day. Here is the code similar to that needed in the handler for your button click event:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
   textbox1.AppendText(DateTime.Now.ToShortTimeString());
}
```

   b. Now experiment by changing the code. There is another method called `ToLongTimeString`, Use it instead, and make a note of what is different.

   c. Add another button to your interface. When it is clicked, show the date.

2. Add another button to your program above. When this button's click event occurs, make the program respond by randomly choosing an inspirational saying, or a joke, and displaying it in a message box. Here is some skeleton code that randomly picks one of your sayings: (You are not expected to understand this all at this stage ... you can just copy and paste the code into your program. All you still have to do to complete this is to show the chosen saying in a message box.)

```
private void button1_Click(object sender, RoutedEventArgs e)
{
      // Make a list of possible sayings ...
      string[] possibleSayings = {
          "Out of my mind... Back in five minutes.",
          "Your gene pool needs a little chlorine.",
          "Ever stop to think, and forget to start again?",
          "I used to have a handle on life, but it broke."
      };

      // Pick a random number between 0 and 3
      // (because that is how we number these possible sayings)
      Random rng = new Random();
      int choice = rng.Next(0, 4); // sets choice to either 0,1,2, or 3

      // Now put the chosen saying into a string variable ...
      string whatToSay = possibleSayings[choice];

      // Complete this now by using a message box to
      // show whatever is in 'whatToSay' to the user.
 }
```

3. Add three more buttons to your program above, with content "Red", "Green" and "Blue". When one of the buttons is clicked the background brush of the window should be changed. Here is some sample code:

```
private void buttonRed_Click(object sender, RoutedEventArgs e)
{
     this.Background = Brushes.Red;     // "this" refers to our GUI window
}
```

5. Diving into Code

Some of this chapter was adapted or taken from Yusuf's tutorial at http://www.ict.ru.ac.za/resources/way_of_the_program/.

There are a few *threshold concepts* in this chapter. Briefly, a threshold concept is one that is often troublesome for some beginners. It changes the way we think about something. (The education theorists use the word *transformational*.) And wrapping our heads around a threshold concept usually requires us to let go of the "old way" of thinking about or doing things before we can see things in the new or different way. (The fact that the new way of seeing things might conflict with other knowledge you already have is what sometimes makes threshold ideas so troublesome!) If you'd like to know more about threshold concepts a Google search will turn up some amazing stuff.

When writing code, there are three essential ideas that are absolutely fundamental:

- What a **variable** is, and how it is created, given a value, perhaps changed, used, and later destroyed. (This is different from maths, so don't get confused!)
- Computer code and statements are executed one-at-a-time, and one statement can change what a previous statement has done. (This is different from maths too, so it can easily be another stumbling point if we're not careful.)
- At any instance in time, every "live" variable has some or other value. All our live variables and their current values are referred to as the **state** of our program. A key idea in programming is that each step of the code can update or change the program state. (Most of the Maths you have seen before doesn't have the idea of a state that changes over time. But this is a central idea in programming. So once again, it can be a troublesome threshold idea that we have to become very comfortable with if we're going to learn to think like programmers.)

Hint to instructors

Getting across the idea of what Computer Scientists mean by a **variable** sometimes gives trouble, because it is not the same as the maths meaning of the word. So one useful analogy might be to compare it to the memory in an older calculator [1].

- We create the variable (with an initial value of zero) when we turn on our calculator.
- We can assign (or *store*) a new value into the variable/memory at any time.
- We can retrieve and use the value that is currently stored in the calculator's memory. Importantly, reading the value from a variable does not remove it or "use it up". So we can use the value as often as we like.
- If we assign a different value to the variable / calculator memory at some later time, the old value is lost, gone forever!
- When we turn off the calculator we destroy the variable and its contents.

A simple game, getting one student in the class to use their calculator memory to look after one variable (say `x`) and other students to look after `y` or `z` might drive home an accurate mental model of what a variable is, and how it works.

5.1. Basic C# programming theory

5.1.1. Types

Before you can write your first code, there is some theory that you must know. The first thing to know is that according to C#, everything has a **type**. For example, in C#, *2* is an *int*. (In maths you'd call it an integer, but we're more lazy that that, and always use use this short name for the type.) We can say that the **type** of *2* is *int*.

We are going to use five of the C# **primitive** types. ("Primitive" here means already built-in to C#, and

understood in any C# program.). They are:

[1] Some newer calculators remember the value in memory even after the calculator is turned off. That is not how variables work in programming languages.

int
: This is any whole number between approximately -2000000000 and 2000000000.

bool
: This is either true or false.

double
: This is any number with a decimal point in it.

char
: This is any single character, digit, or punctuation mark. In C#, a char is always enclosed in single-quotation (') marks.

string
: This is zero or more characters, digits, and/or punctuation marks. In C#, a string is always enclosed in (double-)quotation (") marks.

Important

We will be referring to these five types very often, and using one or more of them in *every single program* for which we write code! Therefore, become very familiar with them. When you hear the word *string* or *bool*, for example, you should immediately understand the Computer Science meaning of that term; and whenever you see any data, you should immediately understand what its type is.

Type	Examples
int	• 576 • 20140214 • -873 • 0
bool	• true • false
double	• 7.51954 • 985.0 • -33.614
char	• '8' • 'z' • ',' • ' '
string	• "Are you excited?" • "z" • "" • " "

Exercises

What is the type of:

1. `89.9` _____
2. `false` _____
3. `'p'` _____
4. `-0.0` _____
5. `11` _____
6. `'0'` _____
7. `0` _____
8. `"0"` _____
9. `true` _____
10. `"z"` _____
11. `"775.21"` _____

When we program, we have to take real-world ideas like kilometers or money or names and student numbers, and figure out how best to work with these things in C#. Which C# primitive type would be best suited to work with:

1. Your name? _____
2. Your age in years? _____
3. Whether you are alive? _____
4. Your grade (A, B, C, D, or E) for an assignment? _____
5. The length of a line in centimetres? _____
6. The number of pages in a book? _____
7. The exact price of a loaf of bread? _____
8. Whether you have eaten anything in the last 6 hours? _____
9. The second letter of your first name? _____
10. The number of grains of sand in a jar? _____
11. The square root of 2? _____
12. Your reasons for reading this book? _____
13. The colour of your eyes? _____

5.1.2. Working with time

The second (and last) bit of theory to know before we begin writing programs is that programming instructions that we're going to deal with in these notes are executed **sequentially**, one after the other. We've already seen that our code-behind is organized into handler methods that are called when some event happens. This means that the order in which you write programming statements within your handler matters. Each statement might change the state of the program (by, say, creating a new variable, or changing an existing variable.) If you get the order incorrect, the program will do the incorrect thing.

There Is No "Right" Program

There are an infinite number of ways to write any particular program, just as there are an infinite number of ways to paint a picture of a person. The solutions that your instructors give you are just the way that **they** would write the program. Study them, and learn from them, but understand that their solutions are not the only way (or even the best way) to write a program.

The order in which statements are executed is called the **control flow** of the program. Control flow can sometimes depend on data: if your age is less than 18 then we want this bit of code to execute, otherwise we want some different code to execute. So the control flow could consist of multiple **paths**, each of which could be taken during program execution. For this kind of programming, one of the most important skills of a programmer is the ability to "see" the control flow of a program. The best way to get better at "seeing" control flow is to read other people's

programs, try to work out what they do *without executing them*, and then execute them to see if your reasoning was correct (or, if your reasoning was incorrect, where and why it was incorrect).

The way we're programming in this book, it is the *event* — a click of a button (and later, other events like moving the mouse) — that causes the application to spring into action and to begin executing the code in the handler method.

5.2. A first program

With that, you are ready to begin your first program!

```
1  int a;
2  int best;
3  int c;
```

> **Syntax**
>
> **Syntax** is the collection of rules that define what a programming language looks like. Each programming language has its own syntax, just as each human language has its own grammar and spelling rules.

This simply says that we should define (create) a variable called `a`, and that `a` should be capable of holding an `int` value. (It will get an initial value of 0, just like your calculator memory is set to zero when you first put your batteries into and turn on the calculator.) Then we create a second variable called `best`, and a third one called `c`. Notice that each statement ends with a `;` (**semicolon**) character. A shorthand, equivalent way of saying exactly the same thing is

```
1  int a, best, c;
```

`a`, `best`, and `c` are **variables**. A variable has a name (which is an **identifier**) and a type. It holds a value. The value of a variable can be changed, but its name and type cannot. Identifiers should start with a letter and contain no spaces or punctuation.

> **Important**
>
> A **variable** in Computer Science is not the same as a **variable** in Statistics, or a **variable** in Mathematics. Understand the specific Computer Science meaning, and do not confuse it with your other subjects!

Merely creating a variable doesn't make for an exciting program. Let's try this program instead:

```
1  int a, b, c;
2  a = 5;
3  b = 10;
4  c = a + b;
```

Let's understand what it does. First, we define (create) the `a`, `b`, and `c` variables. Then we assign the value `5` to `a`. Then we give the value `10` to `b`. Lastly, we put the value `15` into `c`.

These statements, of the form `X = Y;`, are called **assignment statements**. Assignment statements are used to store a new value into a variable. All assignment statements have a right-hand-side (on the right of the `=`) and a left-hand-side (on the left of the `=`). There is always a variable name on the left-hand-side. An assignment

> **Why not one way to do things?**
>
> Programming languages are designed for humans as well as computers, and language designers have tried to make languages that let you express yourself in the way that you find to be most natural. Some people like writing very short, very concise code. Others like to write many lines, believing that this leads to clearer, more readable code. Most people find a

statement executes the right-hand-side first, and then puts the answer it got into the variable that is named on the left-hand-side.

While the computer is working out the value on the right-hand-side, whenever it encounters a variable name it will substitute the value that is currently stored in that variable. So say we're executing line 4 above. The computer fetches the value in the *a* variable, and substitutes a 5. In the *b* variable it finds a 10. So it adds the 5 and the 10, and gets the value 15. This is why the variable *c* is given the value *15*, rather than the value "a + b".

balance between very-concise and very-verbose. It doesn't matter which style you prefer to write in. What matters is that you are able to *read* programs written in *any* style.

Remember that control flow matters ...

Programs do their work one statement at a time. So we would be wrong if we looked at the program above and tried to think about all four statements at the same time. Each statement in a program changes something from how it was before. (We said that the program has a *state*, and statements change the state of the program.) So the first key skill we'll need is to realize that statements are executed in a specific order, top-to-bottom, and that each one makes a change to the state. It is a bit like baking a cake from a recipe — you need to do things one step at a time, in the correct order. If you bake your cake before you've mixed the ingredients, you're asking for trouble!

A key programming skill is *planning* what steps are needed to solve our problem, and getting the steps into the correct order.

We can tighten this code up a bit, by combining definitions (where we define, or create, the variable) and assignment statements, and reduce it from 4 lines to 3 lines in length.

```
1  int a = 5;
2  int b = 10;
3  int c = a + b;
```

And if we'd like to, we can use the shorthand form of definitions to reduce it to a single line:

```
1  int a = 5, b = 10, c = a + b;
```

Now lets make it a bit more interesting...

```
1  int a, b, c;
2  a = 5;
3  b = 10;
4  c = a + b;
5  b = a + c;
```

Now our variables are given initial values in lines 2-4, but when line 5 is executed can you say what value is assigned to *b*? And when this code completes its execution, what value is in the variable *c*?

If you said the value in *b* was 20, and the value in *c* was still the 15 that was left there in line 4, you'd be right.

Putting things together

If you're not yet convinced that Computer Science isn't Mathematics, think about this: in Mathematics, there are common equivalents for *int*, *double*, and *bool* types. There are no common equivalents for *char* and *string* types. That might be because Computer Science is a more practical and pragmatic discipline, and one of the common things that people want to be able to do is read and write text. So it makes sense for Computer Science to have *char* and *string*, and it makes sense for Mathematics to not use those types. It also makes sense for programming languages, such as C#, to use those types in a non-mathematical way. For example, if you wrote this:

```
string a = "super";
string b = "man";
string hero = a + b;
```

That would be perfectly OK in C# (and *hero* would end up with the value *superman*). In Computer Science, *+* commonly means *put these things together*. If you put two *int*s together, you get the sum of the *int*s. If you put two *string*s together, you get the combined value of both *string*s. This makes absolutely no sense, mathematically. But this is Computer Science – not Mathematics!

Exercises

There is something wrong with each of these lines of code. For each line, write a sentence describing what is wrong.

1. in a;
2. string a b;
3. bool a
4. int a, string b;
5. int 3d;
6. int a = 3.0;
7. double a, b = 5.0, c = “6.0”;
8. char zoology = “”;
9. char biology = “E”;
10. bool botany = True;

11. INT CHEMISTRY = 100;

12. bool able, able;

13. int articulate = 8; bool ampersand = articulate;

14. string o'brien = "Connor";

What type would we need to fill in to define *x* in each line below?

1. _____ x = "Yes";
2. _____ x = 213;
3. _____ x = 0.0;
4. _____ x = 'y';
5. _____ x = "y";
6. _____ x = true;
7. _____ x = 5 + 3;
8. _____ x = "false";
9. _____ x = 3.2/5.1;
10. _____ x = false;
11. _____ x = "20/5";

5.3. Basic Arithmetic

There are five arithmetic **operators** that are commonly used in C#. The first four should be familiar to you from school:

- *X* `+` *Y* adds X and Y
- *X* `-` *Y* subtracts Y from X
- *X* `*` *Y* multiplies X and Y
- *X* `/` *Y* divides X by Y

The last operator is the **modulus** operator:

- *X* `%` *Y* gives the remainder that would result from dividing X by Y

Negative numbers and spacing

You usually don't need to put spaces between the parts of an arithmetic expression: `4-5+3` is the same thing as `4 - 5 + 3`. The only exception to this is when you have a negative term in your expression. C# doesn't understand the arithmetic expression `7--4`, but it does understand `7 - -4`, so be sure to put a space before negative numbers.

The first time you did division at school, you probably expressed your answer as "A remainder B". For example, you would say that 7 divided by 4 was "1 remainder 3", or that 24 divided by 6 was "4 remainder 0". The modulus operation gives you the "remainder" part of the answer; so `7 % 4` gives 3, and `24 % 6` gives 0. The sign of the answer is the same as the sign of the dividend. (Dividend is the one on the left: dividend/divisor.) So, for example, `-7 % 4` gives `-3`.

5.4. Order of Operations

You can (and should) group arithmetic terms using parentheses (round brackets), just as you would in ordinary mathematics.

The usual precedence of operations (parentheses, division & multiplication, addition & subtraction) applies to arithmetic expressions in C#. When operators have the same precedence (division and multiplication do, and addition and subtraction do), arithmetic expressions are evaluated from left to right.

5.5. Can we mix different types in an expression?

Arithmetic that is done exclusively with *`int`*s *always* results in integer answers. This means that 7 / 4 = 1, and not 1.75 or 2. Arithmetic that is done exclusively with *`double`*s results in *`double`* answers. This means that 7.0 / 4.0 = 1.75.

Arithmetic that mixes *`double`*s and *`int`*s results in *`double`* answers. This means that 7.0 / 4 = 1.75, and 7 / 4.0 = 1.75.

Exercises

What is the answer to these arithmetic expressions? When the answer is a *`double`*, be sure to show a decimal point in your answer.

1. 6 * 4 - 1 _____
2. 6 * (4 - 1) _____
3. 3.2 * 2.0 _____
4. 3.2 * 2 _____
5. (3 * (5 - 2) / 4) + 1 _____
6. ((5 - 2) / 4 * 3) + 1 _____
7. 1 + 1.0 _____
8. (-(-2) + (-2 * -2 - 4 * 3 * 2)) / 2 * 3 _____
9. (-(-2) + (-2 * -2 - 4.0 * 3 * 2)) / 2 * 3 _____
10. 8 - -8 _____
11. (10 * 7) % 3 _____
12. (5 % 2) + (71553 % 2) _____
13. (19 % 4) + (-72 % 10) _____

5.6. Tokens and Keywords

A **token** in a programming language is a group of one or more characters that are treated as a single chunk. So a number like `1234` would be a token. An identifier like `x` or `average` is a single token. The semicolon (`;`), the parentheses, all the arithmetic operators, the assignment symbol (`=`) are also tokens.

Some tokens consist of more than one character: we use the `==` token to test for equality. We pronounce it *equals*. Other multi-character tokens that we'll see in the next section are `//`, `/*` and `*/`. You may not leave a space between these characters: the sequence `*/` is one token, but `* /` is two tokens.

Some identifiers (multi_character tokens that start with a letter) are **keywords** in the language: they have a special meaning that the compiler understands. Type names like `int`, `bool`, and `string` are keywords, `true` and `false` are keywords, and we'll soon meet other keywords like `if`, `else`, `while`, `new`, and `for`. You can find a list of all the keywords at http://msdn.microsoft.com/en-us/library/x53a06bb.aspx.

You can't use keywords for your own identifiers. So don't choose a name like `double` for your variable (or for a control in your GUI).

In Visual Studio, and in the on-line version of this textbook, keywords are often shown in different colours. This colour-coding helps us "chunk" the program mentally.

5.7. Comments

A comment is some part of your program code that exists only for humans. C# understands that when it encounters a comment, it should not try to make sense of the comment, nor should it try to execute it.

There are two mechanisms for introducing comments. A *single-line comment* starts with a double-forward-slash token `//`. C# will ignore everything else to the right of it on the same line. Look at the code below at lines 10 and 11 where we use single-line comments.

Block-comments are comments that can extend over many lines. A comment starts whenever the compiler finds the start-of-comment token (`/*`) and everything is treated as a comment until the compiler finds a matching end-of-comment token (`*/`). We have a block-comment in the program below that starts on line 1 and ends at line 6.

5.8. Type conversions

Let's revisit the code from the previous chapter for calculating the area of a pane of glass. (We've added a new block-comment, so it not completely identical.)

```
1   /* This is an example of a block-comment that can span
2      many lines. The C# compiler will ignore comments like this.
3
4      This fragment of code was written by Pete.
5      Date of last change:  13 Jan 2014.
6   */
7
8   private void btnCalculate_Click(object sender, RoutedEventArgs e)
9   {
10      double w = Convert.ToDouble(txtWidth.Text);  // Convert string in texbox to double.
11      double h = Convert.ToDouble(txtHeight.Text); // Same again.
12      double area = w * h;
13      string result = string.Format(" {0} x {1} gives an area of {2}\n", w, h, area);
14      MessageBox.Show(result, "It says ...");
15  }
```

Because C# has this idea of every value has a type, a string like "123" is not the same as the integer 123, or the double number 123.0

Humans would have no problem in automatically converting a string into a number, or vice-versa. But our programs have to do it the hard way, explicitly.

Normally input and output from a program will use strings, because that is what we humans read and write. So C# needs a mechanism to convert from the text (or strings) that you type into numbers, and vice-versa.

So lines 10 and 11 illustrate the way we will take some text that the user has entered on our GUI, and convert it to a different type: a double in this case.

C# provides a built-in library of `Convert` methods. In this book we'll only use two of them: the one we've just seen, and a conversion that can convert a string to an `int`.

```
1   int n = Convert.ToInt32(txtAge.Text);  // Convert string to an int.
```

5.9. String formatting

New let's tackle the opposite problem. Our program has calculated some answers, but we need to convert them into strings so that our human users can read them.

In line 13 of the program above we used the `string.Format` method to create a new string. Let's dig a bit deeper into how that works, as we'll be using it a lot. String formatting allows us to provide a text *template* that contains some *place-holders*. The three place-holders in the template in line 13 above are `{0}`, `{1}` and `{2}`.

When the `string.Format` method is executed, it creates a new string from its template, and it replaces all the place-holders with values. The place-holders are numbered, and their number determines which argument will get plugged in at that position.

This results in a new string, built from the template and the values that were substituted. We can assign this to a string variable, or use it in some other way.

So in the example above, when line 13 is executed, the current value in variable `w` gets plugged into the template in place of the place-holder `{0}`, the current value in `h` gets plugged into the template in place of `{1}`, and the area that we previously calculated on line 12 and then saved into a variable gets plugged into the template in place of `{2}`.

String formatting is very powerful and can do much more than we shown here.

5.10. Debugging

Programming is a complex process, and because programs are written by human beings, they often contain errors. Some of these errors are simple, like messing up the grammar of the programming language, and the compiler can tell you there's a problem and (usually) also tell you where it is, so you can fix it. These kinds of simple errors are called **syntax** errors. **Syntax** refers to the structure of a program and the rules about that structure. Mis-spelling a word in a programming language, or putting it in the wrong order, or not including necessary punctuation, or including too much punctuation, or not capitalising a word that should be capitalised (or vice versa!) are all examples of syntax errors.

During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer syntax errors and find them faster. After a few weeks, you won't make any syntax errors at all! This is just like the process of learning a new spoken language. While you are learning, you will make silly errors that a native speaker wouldn't make. But after a while, you don't make any errors. And, just like learning a new spoken language, the more you practise, the better you'll get ... and if you don't practice enough, you'll keep making mistakes. So practice: it really does make a big difference.

The more difficult errors are **semantic** errors, where you've given the computer a valid sequence of instructions that the compiler can understand, but the instructions don't lead to the result that you think they lead to. You'll run into plenty of these, and when you do you'll need to understand exactly what you're asking the computer to do, so that you can figure out where the flaw in your reasoning is.

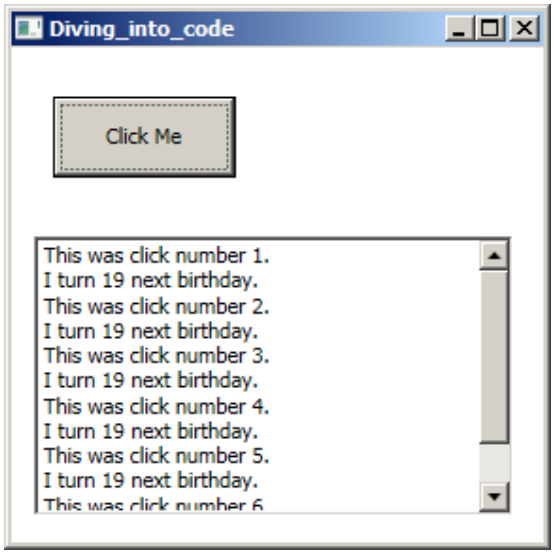
Runtime errors, which cause a program to stop running, occur because of **semantic** errors. Usually, you've made the error of assuming something that you shouldn't have assumed! These errors are also called **exceptions** because they indicate that something exceptional (and bad) has happened. For

example, not even the computer can divide a number by zero. So if you've written some code that tries to divide by zero, you'll get a runtime error.

Programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**. It is useful to distinguish between types of bugs in order to find the cause of the error and fix it more quickly. Here is how you can distinguish between types of bugs:

1. Do you get errors when you *compile* the program? If so, you have a **syntax** error.
2. Does the program fail to do what you expect it to do? If so, you have a **semantic** error.
3. Does the program stop running before you expect it to? If so, you have a **runtime** error which is caused by a **semantic** error. You need to fix the semantic error so that the runtime error will go away.

5.11. Variables in your code-behind methods



An important idea about variables is that at some point in time they are created, and later they die. They have a *lifetime*.

When we define a variable *inside* a code-behind method, it only lives *while our program is busy executing the method*. When the method completes its work, any variables that were created by it are destroyed, and their values are lost.

This means that we can't use a variable defined inside a method to "remember" values over longer periods of time.

We can also define variables at the *class-level* (you'll learn a lot more about classes soon). These are called *class-level variables*. In our programs, these variables are created when our window first opens as we run our program. They will stay alive as long as our window is alive, i.e. while our program is running.

So let's do a very simple example. We'll have a GUI with a single button. Each time the button is clicked, we'll show a message that tells us how many times the button has been clicked since the program started running.

To keep a counter, we'll need a variable, but one that stays alive between clicks of the button. So it will need to be defined in the class, rather than inside the method of the handler code. We'll assume we can create the GUI, and attach a handler:

```
1  public partial class Diving_into_code : Window
2  {
3      // Because this variable definition is outside any method,
4      // it is a class-level variable and it lives on, even after a
5      // method finishes executing.
6      int clickCounter = 0;
7
8      public Diving_into_code()
9      {
10         InitializeComponent();
11     }
12
13     private void button1_Click(object sender, RoutedEventArgs e)
14     {
15         // A variable defined here only has a short lifetime
16         // while this method is active.
```

```
17          int ageNow = 18;
18
19          // Add one to the class-level counter variable
20          clickCounter = clickCounter + 1;
21
22          // prepare the message we want shown.
23          string msg = string.Format("This was click number {0}.\n", clickCounter);
24
25          // Show the results in our GUI
26          txtResults.AppendText(msg);
27
28          // Now even if we change the local variable, it is about to die!
29          ageNow = ageNow + 1;
30          string msg2 = string.Format("I turn {0} next birthday.\n", ageNow);
31          txtResults.AppendText(msg2);
32      }
33  }
```

Notice the variable is defined at line 6, not inside the handler method. Each time the handler method is executed, line 20 retrieves the current value in the variable, adds one to it, and assigns that value back to become the new value of the variable. Then in lines 23–26 we organize to put the answer back into the GUI front-end. But notice how that works differently from the variable `ageNow` which is defined inside the method. Study the results carefully. Why doesn't `ageNow` also count up 20, 21, 22, 23, 24 ... ?

5.12. Glossary

assignment statement

An assignment statement first evaluates the expression on its right-hand-side, and then stores the value into the variable on its left hand side.

comment

A part of the program that the complier ignores. It exists to be read by other humans.

control flow

Statements in a program are executed in a specific order, and each statement can change the state of the program (via an assignment statement). Control flow refers to the order in which execution happens in your code.

expression

A combination of operators and operands. This can also include a call to a method, so for example `2.0 * 3.14159 * Math.Sqrt(x)` is an expression that can be evaluated.

keyword

An identifier that is reserved in the language, and has special meaning.

identifier

Used to name something, like the name we choose for a variable or a control. There are some rules that identifiers must start with a character and may not contain symbols or spaces. (There is one exception: the underscore character _ is legal anywhere within an identifier, even as the first character.)

operator

The main arithmetic operators that combine operands while they evaluate the expression. For example, in the expression `5 + y`, the operator is `+`.

operand

The things that an operator works on. In in the expression `x + 3`, the `x` and the `3` are the operands.

sequentially

One after the other.

state of the program

Refers to what variables exist in a program at any point in time, and what values they hold.

token

A “chunk” of one or more characters that the compiler treats as a single “word”. When we read text we have to group characters into syllables and words. In the same way, when we read code we have to group the characters into tokens that carry meaning.

type

Every expression and value has a specific type. Types we’ll use most are `int`, `double`, `string`, `bool` and `char`.

type conversion

An expression that converts from one type of value into another, typically from a string like `"42"` to an `int` value 42.

variable

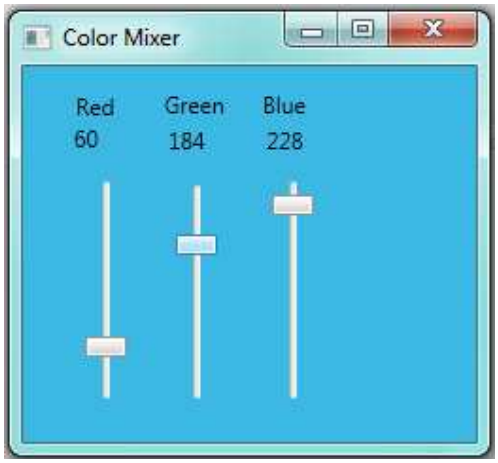
A variable is a place in the computer’s memory where it can hold a value. The value can be read by your code, and can be overwritten using an assignment statement.

variable definition

A statement that defines a variable causes it be created in the computer’s memory. A variable definition always gives the type and the name of the variable.

5.13. Exercises

1. Write a program that inputs a number of hours, a number of minutes, and a number of seconds (all integers). Output the total number of seconds represented by the time. For example, 2 hours 42 minutes and 17 seconds would convert to 9737 seconds in total.
2. Write a program that does the opposite of the above: input an integer number of seconds, and display the number of hours, minutes and seconds. For example, 3665 seconds would display as 1 hour, 1 minute and 5 seconds.
3. Write a program that inputs the lengths of the two shorter sides of a right-angled triangle. It should output the length of the hypotenuse. (Refresh your knowledge of Pythagoras’ theorem if you’re not comfortable with it.)
4. Design an application with 3 buttons showing A, B, and C. Or perhaps instead, (“crouch”, “touch”, “set”), or (“Three”, “Blind”, “Mice”). The buttons should be clicked in strict cyclic sequence A,B,C,A,B,C,A,B...
    a. Arrange program logic that pops up an error message if the user gets the sequence wrong.
    b. Use the `IsEnabled` property on the buttons so that it becomes impossible for the user to make a mistake.
5. Design a colour-mixing application containing three sliders with a label above each. Each should allow the user to slide between 0 and 255. When any slider is changed, update its label to show its value.



b. Add this method to your code-behind (copy and paste is useful, and we haven't covered these ideas yet. The code should make more sense as we progress in the book. At this stage we're focusing on how the GUI interacts via events with the code-behind.)

```
private void setBackgroundColorFromSliders()
{
    // Some say that WPF has a bug, others call it a feature ...
    // Do not handle events if the window is still initializing itself.
    if (!this.IsInitialized) return;

    byte r = Convert.ToByte(slider1.Value);
    byte g = Convert.ToByte(slider2.Value);
    byte b = Convert.ToByte(slider3.Value);

    label1.Content = r;
    label2.Content = g;
    label3.Content = b;

    this.Background =  // Create a brush from the slider values
                new SolidColorBrush(Color.FromRgb(r, g, b));
}
```

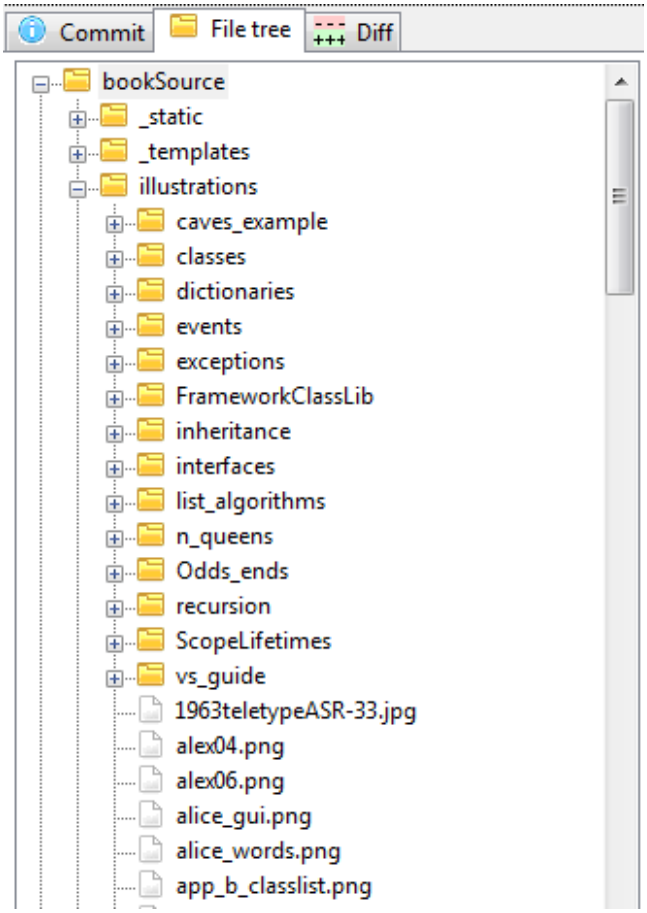
A `byte` type is like a small integer with values between 0 and 255. We use this because the `Color.FromRgb` method wants byte arguments.

c. Now add handlers so that each time any of the sliders is changed it responds by calling our new `setBackgroundColor` method. Run the program and play. To call the method above, use a line like this in your handler:

```
  setBackgroundColorFromSliders();
```

Video demonstration of building this colour-mixing application
http://www.ict.ru.ac.za/resources/thinksharply/videos/colormixer.avi

6. Nesting Things in Other Things



In Computer Science we often organize things into **trees**.

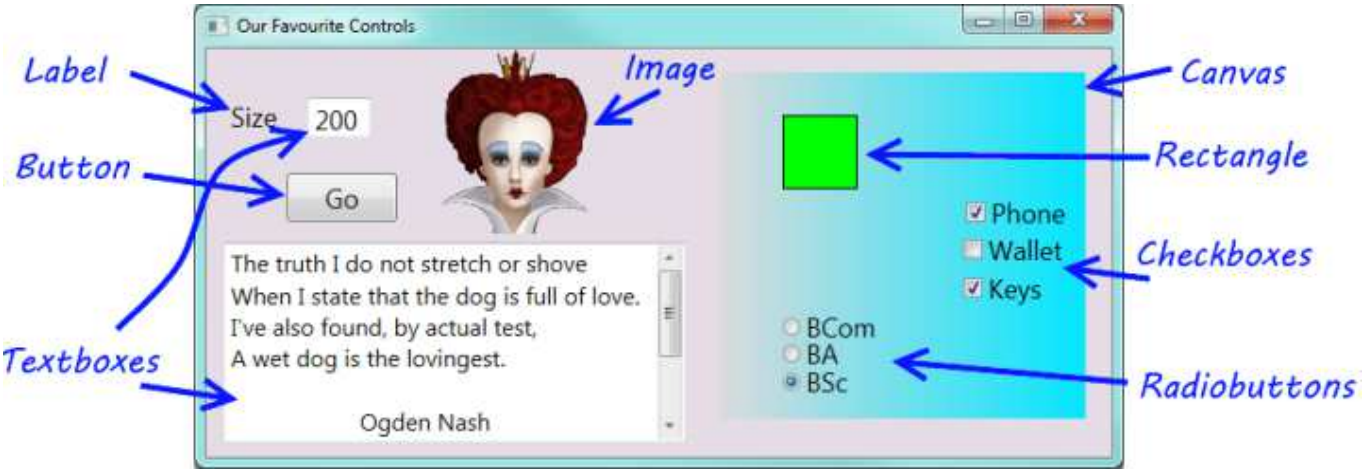
One really useful example is the directory structures on our disk drives. In the example shown here (the source code for this book), we have a top-level directory called `bookSource` which has subdirectories `_static`, `_templates` and `illustrations`. There are many sub-directories under `illustrations`, (none of their sub-detail is presently shown). In addition, within the `illustrations` directory we have a number of files like `alex04.png`.

Now go back to Section 2.6 and look again at the Solution Explorer in Visual Studio. It uses the same ideas: our solution contains some programs which contain a number of other sub-trees. The syntax is a little different, but the idea is similar.

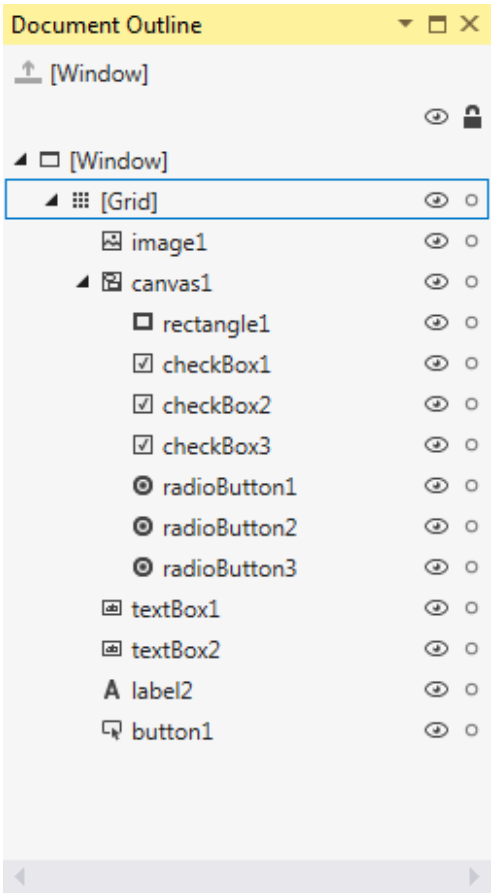
What other trees have we already seen? Well, this textbook (like most) is arranged into chapters with sections and subsections.

6.1. Nesting in our GUIs

Here again are some of our favourite controls, as presented in Chapter 3. For a bit more insight into how the GUI works, we're going to look at how some controls are nested, or contained, in others.



In Visual Studio, we can navigate to the menu item `View | Other Windows | Document Outline` and we can get a tree-like view that shows the nesting of our GUI controls:



We need to be able to relate what we see in our document outline to what we see in our GUI. The outline shows us that our top-level structure is a Window that contains just one control, a grid. The grid in turn contains a canvas (named `canvas1`), two text boxes, a label, an image, and a button.

But the canvas is itself is a container with more children: it contains a description for its background, a rectangle, three check boxes, and three radio buttons.

6.1.1. How does this help us?

Play a bit with the GUI design in Visual Studio, for example, by moving the canvas to a different position. What we immediately notice is that all the children controls that are nested within the canvas move with the parent canvas. In other words, their positions are defined relative to the canvas, not relative to the window.

Similarly, when we run our programs, if we move our window to a different part of our screen, everything owned by the window moves too. (It would be weird if we moved a window and the window left its buttons behind in the old positions!) So here too, everything "owned" by the window is positioned and defined relative to where the window is at the moment.

So by using a canvas (or a panel, or any other control that can act as a parent for other controls), we bring a very convenient organization to our programs.

6.1.2. But there is even more

Every control has a property called `Opacity` (from the word *opaque* which is the opposite of *transparent*). Normally, the opacity of a control has the value 1, (i.e. 100%) which means "fully opaque". Setting the value smaller make the control partially transparent, so you can see through it. If we set the value to 0 it becomes fully transparent.

Let us do the following now: we'll move the canvas over the top of the image of the queen. So the queen won't be visible when we run our program. Then we'll create a handler for the button, and each time we click the button we'll execute this code which makes the canvas a little less opaque:

```
1  private void button1_Click(object sender, RoutedEventArgs e)
2  {
3      canvas1.Opacity = canvas1.Opacity * 0.90;
4  }
```

Line 3, an assignment statement, works by first evaluating the right hand side to compute the new opacity value. Then the value is stored into the `Opacity` property of `canvas1`.

When we run our program the queen is initially hidden behind the canvas. As we repeatedly click the button the canvas becomes less opaque: we see less of the canvas and more of the queen.

But an important point is that because the canvas controls its children controls (the rectangle, the check boxes and the radio buttons), they also automatically become more transparent, along with the parent. So the opacity property applies to the parent and any children that it contains.

If we look back at the Document Outline window above, we'll see that there are are visibility radio buttons next to each control too. So if we hide `canvas1`, all its children become hidden too. Under the padlock icon we can also check the button to lock the canvas. As we do this, all its children also become locked. A locked control cannot be moved on the GUI in the GUI designer: it is a safety feature that allows us to get our layout looking good, and then to prevent us from accidentally messing it up again.

Having our controls nested and grouped in this way allows us to work with the whole nested structure (move it about, change its opacity, kide it, etc.) by simply changing a property in the parent container.

6.1.3. The XAML also reflects this things-within-other-things nesting structure

If you are interested in reading the XAML (and recall that the XAML is just another way to describe the structure of the GUI), we'll need to understand a few rules about the XAML. XAML is a special usage of a more general notation called XML.

In XML, every element is described by enclosing it in opening and closing tags. An opening tag can also describe some properties. Tags are a bit like brackets and parentheses in familiar arithmetic expressions: they can nest inside one another, and every open bracket (or parenthesis) must have a matching closing bracket (or parenthesis). We must be able to spot where the tag opens, and where it ends.

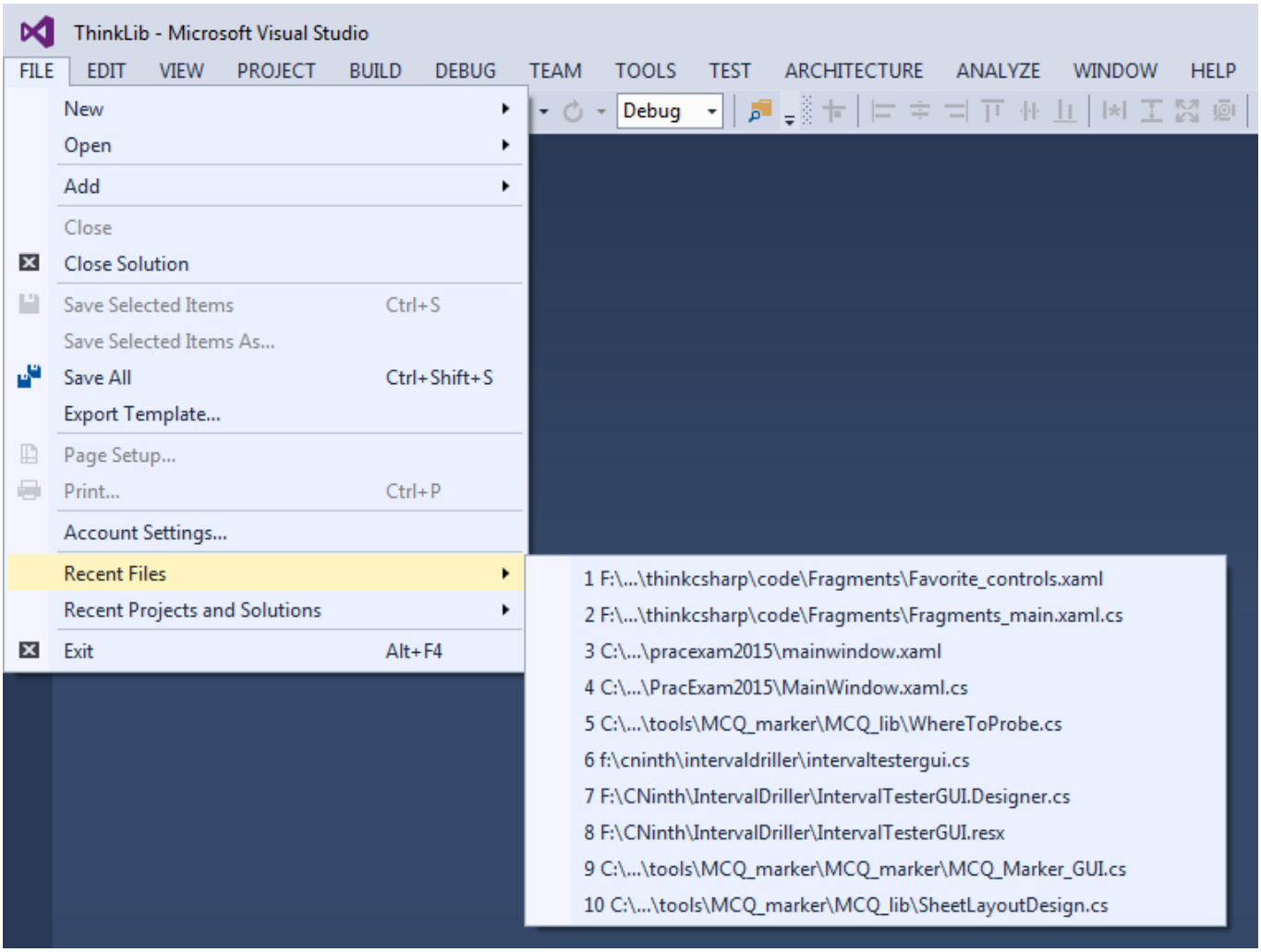
Here is the XAML for this GUI:

On line 6 we see the "opening" canvas tag with some properties. The corresponding "closing" tag is on line 20. The small boxes at the left on lines 1, 5, 6 and 7 are great for exploring and understanding the nested tree-like structure: we can expand or fold down the detail of any element, and hide its children.

XML also uses a shorthand for closing a tag if it has no nested sub-components. In XML, the long-winded `<SomeTag> </SomeTag>` can be abbreviated to `<SomeTag />`. This shorthand form is used on lines 13-19.

6.2. We want Menus, with Sub-Menus, and more Menus!

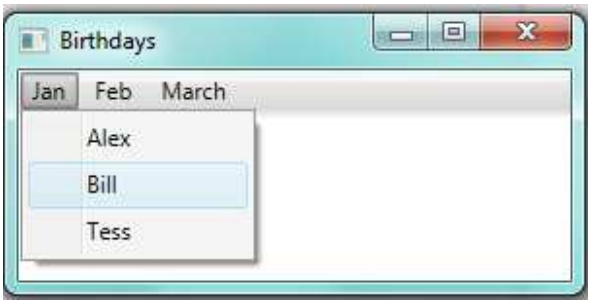
Menus are really powerful tools for organizing a GUI. Program designers can pack a lot of functionality into a very convenient (and compact) menu structure, as shown in this fragment of the Visual Studio menus:



Looking at Visual Studio (yours might be a bit different from a different set of preferences), all the functionality is arranged into a menu with 13 major groupings across the top. In the illustration, we've gone down into the `File` menu, which in turn contains another menu with a number of items, some of them with sub-menus. (This should sound similar to having directories that contain files and more sub-directories!) In the illustration we've opened yet another sub-menu, and we can now make a choice.

In our programs, each menu item that has no further substructure works a lot like a button. When a menu item is selected it generates a `Click` event. We can attach a handler to respond to the event so that when our user selects the menu item we can respond by executing some code.

Let us build a simple menu for our GUI: we'll have month names across the top, and under each month we'll keep the names of friends with birthdays in the month. Here is what it will look like when we start:



The Visual Studio properties editor is not very helpful when editing menus like this. So in this case, it might be easier for us just to edit the XAML instead.

If we copy and paste this code into our XAML, (make this a child of the `Grid` control of your window), you'll get a menu just like the one shown above.

```
1  <Menu Height="25" HorizontalAlignment="Stretch" Name="menu1" VerticalAlignment="Top" >
2      <MenuItem Header="Jan">
3          <MenuItem Header="Alex" />
4          <MenuItem Header="Bill" />
5          <MenuItem Header="Tess" />
6      </MenuItem>
7      <MenuItem Header="Feb" >
8          <MenuItem Header="Zola" />
9      </MenuItem>
10     <MenuItem Header="March" />
11 </Menu>
```

Now we can just edit the XAML to extend the menu for all twelve months of the year, and to add more friends.

None of these menu items has been given its own name yet (and we can omit this step if we want to), nor have they had an event handler attached. Let us attach one event handler to the `Tess` menu item by changing line 5 of the XAML like this:

```
1  <MenuItem Header="Tess" Click="Tess_Click" />
```

As we've seen before, what this does is "attaches" an event handler to a click event on this menu item. All we have to do now is jump into the code-behind, and provide the event handler, which might look like this:

```
1  private void Tess_Click(object sender, RoutedEventArgs e)
2  {
3      MessageBox.Show("It is Tess' birthday on 15 January!", "Birthday reminder");
4  }
```

We now have a program with a rich menu structure. When we select the `Tess` menu item our handler responds by popping up a reminder.

Nesting

This idea of organizing complex information or creating complex structures by nesting things in others — e.g. menus that contain other menus, directories that contain other directories, or GUI controls like the canvas that can contain other controls (even other canvas controls), and fragments of XAML that contain other fragments of XAML — is a big deal in Computer Science. It is important enough that we've devoted a separate chapter to this idea.

Try to find other examples of nesting too, and work on thinking about all these different uses as coming from the same essential underlying idea.

6.3. Exercises

1. In an earlier chapter we had some exercises in which we built GUIs. Revisit those GUIs, and observe the nesting structure using both the Document Outline feature, and by examining the XAML.
2. In the previous chapter we introduced block-comments. Devise an experiment to determine whether one block-comment can nest inside another. Record your findings.
3. Suppose you're going to write an application of your own invention. Perhaps you'll use it to manage your music collection, or to chat in a chat room, or to keep track of famous quotes. Design a GUI for your application with a menu structure that contains at least 10 menu items, one of which must be nested at least 3 levels deep.
4. Do this experiment: place a button on a GUI and make it pop up a message box when it is clicked. Now change the opacity of the button so that it becomes fully transparent, and run your program. Can you still click the button even when it is hidden behind its invisibility cloak? Can you imagine how this might be used to trick a user into doing something that they might not otherwise do?

7. Hello, Little Turtles!

There are many *modules* written for C# that provide very powerful features that we can use in our own programs. Some of these are can send email, or fetch web pages. Some come with C#, others are add-ons or modules that we write ourselves, or perhaps download from the Internet.

We've written our own module called **ThinkLib** to go with this book. One of the things it does is it allows us to create turtles and we can get them to draw shapes and patterns.

See the appendix Getting Started with ThinkLib for details (and a video) about how to set it up for your C#.

The turtles are fun, but the real purpose of the chapter is to teach ourselves a little more C#, and to develop our theme of *computational thinking*, or *thinking like a computer scientist*. Most of the C# covered here will be explored in more depth later.

7.1. Our first turtle program

Our turtles will live and play in a *playground*.

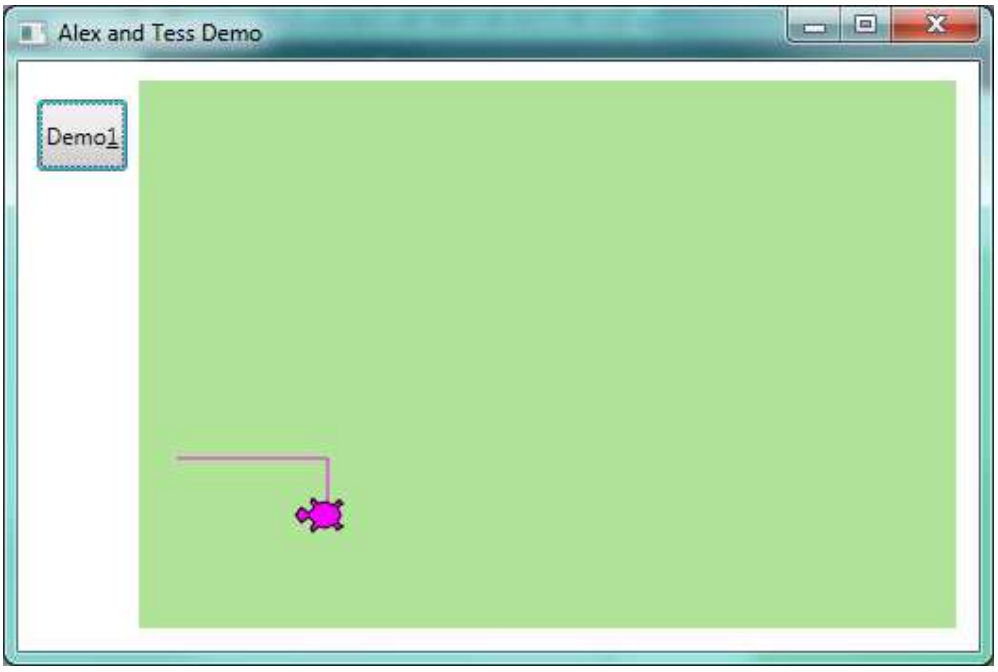
We'll create a WPF application. In WPF, the playground will be a Canvas control. So we'll drag a Canvas onto our application. I like to name my canvas "playground". I also like to set the Width and Height properties to Auto, and to set the HorizontalAlignment and VerticalAlignment properties to Stretch. This means the playground will also resize when you change the size of the application's main window.

We can also set some canvas properties to give it a nice background colour, (or perhaps even a background image). So let's assume we have our application with a canvas and a button. Here is our first turtle example which draws a square:

```
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using ThinkLib;

namespace Part1
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        Turtle tess;              // Define a variable to refer to our turtle

        public MainWindow()
        {
            InitializeComponent();
            tess = new Turtle(playground); // Give birth to a turtle in the playground.
        }                                  // Let variable tess refer to the new turtle.

        private void btnDemo1_Click(object sender, RoutedEventArgs e)
        {
            tess.Forward(80.0);
            tess.Right(90.0);
            tess.Forward(30.0);
            tess.Right(90.0);
        }
```

```
31          }
32      }
```

When we run this program, after clicking the button, our window looks like this:



There are a number of new things in this code:

Line 7 is new: it tells the compiler to use our library ThinkLib. Line 16 defines a variable for referencing a turtle.

When the MainWindow is first instantiated (created), it has a special **constructor** method at lines 18–22 which is automatically executed. (C# knows this method is the constructor because it has the same name as the class name on line 14.) The skeleton of the constructor is usually written for us by Visual Studio, and it is responsible for any "factory settings" that we want our new window to have.

We added line 21 as part of our factory settings. (When we add our own statements into the constructor, they should always come *after* the one that Visual Studio already put there — i.e. after the one at line 20).

Line 21 creates our first turtle in the playground. The turtle always starts its life at a fixed home position. If you don't like where your turtle is "born" [1] into the playground, you can provide some extra arguments when you create the turtle to say exactly where it must be placed). It always starts off facing East (to the right of the screen). The newly created turtle is assigned to the variable `tess` defined in line 16.

[1] Turtles come out of eggs. So it might be more fun (and more accurate) to say "they're *hatched* into the playground".

Notice that `tess` is defined as a class-level variable. What that means is that the turtle will stay alive as long as our GUI window stays alive.

When the user clicks the button, the *click* event occurs, and the handler code at lines 25–28 is executed. It instructs the **object** referred to by tess to move, and to turn. We do this by **invoking**, or calling, **methods** — these are the instructions that all turtles know how to respond to. The four statements are

executed in lines 26,27,28,29, and the picture above shows what you'll see after the button has been clicked.

As it stands now, the handler code only draws only half a rectangle. Another click of the button will complete the rectangle. (Or better yet, we could modify the code so that it draws a complete rectangle on a single click — we'll need a total of 8 statements in the handler.)

The coordinate system for the playground is the same as it is for all the controls we've seen so far: A position is referred to as (x,y). (0,0) is at the top left corner of the canvas, and increasing x moves to the right, increasing y moves down. (The y axis is opposite to what most school geometry teaches.)

An object can have various methods — things it can do — and it can also have **properties** (like we've already seen with the window and the controls). For example, each turtle draws with a *brush*. The brush can be a solid colour or it can be textured. We can set the width of the brush, and we can pick it up or put it down. (If a turtle moves while its brush is up, it doesn't draw a line in the playground.)

The brush associated with the turtle, the width of its brush, the position of the turtle within the playground, which way it is facing, and so on are all part of its current **state**.

7.2. Instances — a herd of turtles

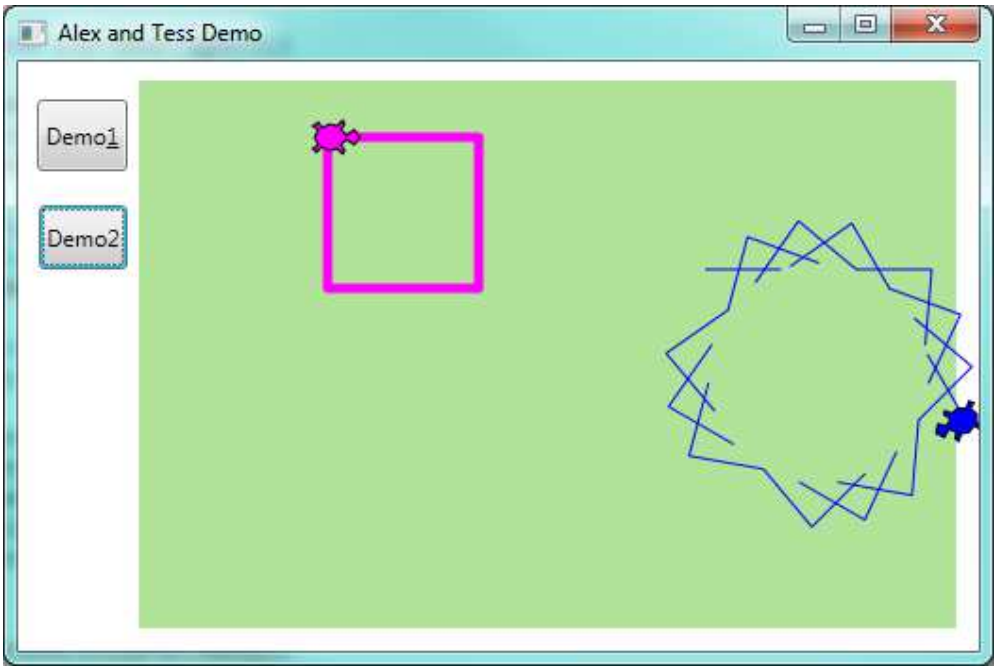
Just as we can have many different buttons or menu items in a program, we can have many playgrounds and many turtles. Each of them is called an **instance**.

Each instance has its own properties and methods — so Alex might paint with a thin black brush and be at some position, while Tess might be going in her own direction with a fatter brush. So let's add a few lines to our constructor method to create a second turtle in the same playground, and we'll set some properties for each of them:

```
1  public partial class MainWindow : Window
2  {
3      Turtle tess, alex;
4
5      public MainWindow()
6      {
7          InitializeComponent();
8          tess = new Turtle(playground, 100.0, 30.0);  // Create a turtle in the playground.
9          tess.BrushWidth = 5.1;                       // Set some properties.
10
11         alex = new Turtle(playground, 300.0, 100.0); // Create another turtle
12         alex.LineBrush = Brushes.Blue;               // Make its lines blue
13         alex.BodyBrush = Brushes.Blue;               // and make its body blue too
14         alex.BrushWidth = 1;
15     }
16
17     private void btnDemo1_Click(object sender, RoutedEventArgs e)
18     {
19         tess.Forward(80.0);   // Make Tess draw a square
20         tess.Right(90.0);
21         tess.Forward(80.0);
22         tess.Right(90.0);
23         tess.Forward(80.0);
24         tess.Right(90.0);
25         tess.Forward(80.0);
26         tess.Right(90.0);
27     }
28
29     private void btnDemo2_Click(object sender, RoutedEventArgs e)
30     {
31         // Draw a broken line by picking up the brush in the middle
```

```
32          alex.Forward(40.0);
33          alex.BrushDown = false;
34          alex.Forward(40.0);
35          alex.BrushDown = true;
36          alex.Forward(40.0);
37          alex.Right(95);
38      }
39  }
```

Here is what we get after clicking the buttons a few times:



Find a video demonstration of building this application at

http://www.ict.ru.ac.za/resources/thinksharply/videos/TessAndAlex.avi

Here are some *Think Sharply* observations:

- There are 360 degrees in a full circle. If we add up all the turns that a turtle makes, *no matter what steps occurred between the turns*, we can easily figure out if they add up to some multiple of 360. This should convince us that Tess is facing in exactly the same direction as she was when she was first created. (Geometry conventions have 0 degrees facing East, and that is the case here too!)
- We could have left out the last turn for Tess, but that would not have been as satisfying. If we're asked to draw a closed shape like a square or a rectangle, it is a good idea to complete all the turns and to leave the turtle back where it started, facing the same direction as it started in. This makes reasoning about the program and composing chunks of code into bigger programs easier for us humans!
- Alex turns 95 degrees on each click of the button. *How many clicks are required before he is back facing East, and we have a pleasant complete pattern on the screen? How many corner points will the complete pattern have?*
- Comments (at lines 8,9,11,18,30) help record our mental chunking, and big ideas. These ideas are not always clear when all we have is the bare code.
- Notice that Alex drew some lines and is partially off the playground. If we don't like that, we can set the ClipToBounds property of our canvas. Turtles will still be able to go outside the playground, but their lines and the turtle itself won't be rendered. (*Render* is our fancy computer-speak for drawing something.)

- And, uh-huh, two turtles may not be enough for a herd. But the important idea we want to emphasize is that the ThinkLib module gives us a kind of *factory* that lets us create as many turtles as we need. Each instance has its own state and behaviour.

7.3. The while loop

It should have taken you 72 clicks to complete Alex's pattern. That is a tedious amount of clicking.

A basic building block of all programs is to be able to repeat some code, over and over again — in programming we call this *looping*.

C#'s **while** loop solves this for us. Let's say we want to repeat some code 72 times.

```
1  int i = 0;
2  while (i < 72) {
3      // the code you want repeated goes here ...
4      i = i + 1;
5  }
```

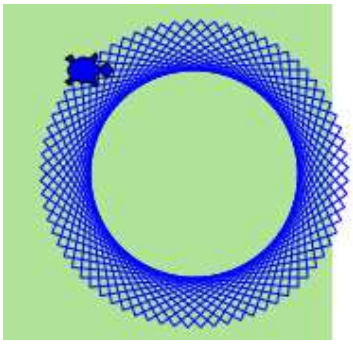
- In this loop, the variable `i` is used to control what happens. We call it the **loop control variable**. We could have chosen any other variable name instead.
- Between the curly braces is the **loop body**. This is the section of code that will be repeatedly executed.
- Before each *iteration* or *pass* of the loop, a check is done to see whether the **controlling expression** `(i < 72)` is true or false. If it is true, the loop body is executed again. At the end of the loop body the assignment `i = i + 1` is executed. This adds one onto the variable `i`.
- So the full sequence of actions here is
    - Define a new variable i and give it an initial value of 0.
    - Test the loop controlling expression.
    - If it true, execute all the statements in the body of the loop. If it is false leave the loop.
    - Repeat everything from the second step in this sequence.

  Once the controlling expression is tested and found to be false, program execution continues at the next statement after the loop body (line 6).

Let's modify our handler for Demo2: on a button click we'll do the same thing 72 times:

```
1  private void btnDemo2_Click(object sender, RoutedEventArgs e)
2  {
3      int i = 0;
4      while (i < 72)
5      {
6          alex.Forward(120.0);
7          alex.Right(95);
8          i = i + 1;
9      }
10 }
```

A single click on the button now produces this pattern:



Some observations:

- What is important here is that we've found a "repeating pattern" of statements, and used a loop to repeat the pattern. Finding the chunks and getting our programs arranged to repeat useful chunks is a vital skill in computational thinking.
- Computer scientists like to count from 0! Observe carefully that the body of the loop executes exactly 72 times: the first time `i` has the value 0, then 1, then 2, and the last iteration of the loop occurs when `i` has the value 71! When the program is about to execute line 10, the value of `i` will be 72.
- Notice that after 72 turns of 95 degrees each, you end up with an exact multiple of 360, so Alex is facing the same way as he started — East.

7.4. A few more turtle methods and tricks

Turtle methods can use negative angles or distances. So `tess.Forward(-100.0)` will move Tess backwards, and `tess.Left(-30.0)` turns her to the right. Additionally, because there are 360 degrees in a circle, turning 30 to the left will get Tess facing in the same heading as turning 330 to the right!

This suggests that we don't need both a left and a right turn method — we could be minimalists, and just have one turn method.

One aspect of *thinking like a scientist* is to understand more of the structure and rich relationships in whatever field we are working in. So revising a few basic facts about geometry and number lines, and spotting the relationships between left, right, backward, forward, negative and positive distances or angles values is a good start if we're going to play with turtles.

Every turtle has a Boolean property `Visible`. If you set this false, the turtle becomes invisible. It's brush can still draw in the playground, though.

Turtle methods are executed as fast as possible. There is a property that can force the turtle to delay between each operation. This appears to slow the turtle down, and might be more fun if you prefer watching the pattern build up rather than just want to see the finished product.

```
1  alex.DelayMillisecs = 100;
```

A turtle can also "stamp" its footprint onto the playground (it can also stamp some other things, like text, into the playground). This will remain after the turtle has moved somewhere else. Stamping works, even when the brush is up or the turtle is invisible.

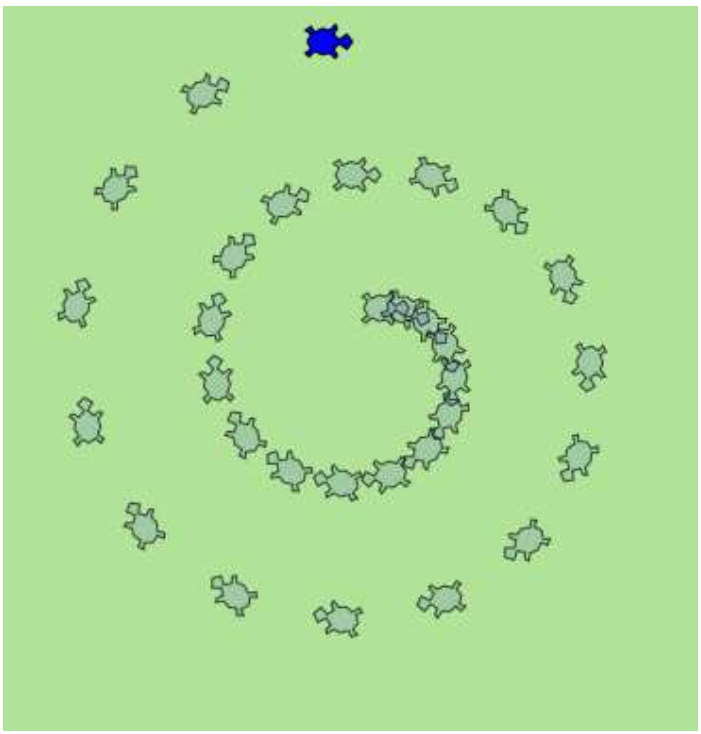
Methods like `Forward` and `Left` use *relative* geometry: they move forward from the current position, or turn relative to the current heading. But there are also some "absolute" methods.

- `alex.Goto(x,y)` will go to absolute position (x,y). If the brush is down, the turtle will leave a trail.
- `alex.WarpTo(x,y)` moves the turtle to position (x,y) without drawing. The brush remains in the state it was in.
- `alex.Heading` is a property that gets or sets the turtle's heading.
- `alex.Position` is a property that gets or sets the turtle's position. Setting this is like calling `WarpTo`.

Let's put a new button on our GUI and get Alex to show off some of these new features:

```
1  private void btnDemo3_Click(object sender, RoutedEventArgs e)
2  {
3      alex.WarpTo(200.0, 200.0);     // Warp without drawing
4      alex.BrushDown = false;        // Pick up the brush
5      double size = 10.0;
6      int i = 0;
7      while (i < 30)
8      {
9          alex.Stamp();              // Stamp a footprint
10         size = size + 2.0;
11         alex.Forward(size);
12         alex.Right(24.0);
13         i = i + 1;
14     }
15 }
```

Here's what this gives:



How many times was the body of the loop executed? How many turtle images do we see on the playground? All except one of the shapes are footprints (and footprints look a bit different from the turtle itself). But the program still only has *one* turtle instance.

Read the Code to Extract Meaning

Beginners sometimes make the mistake of "skip-reading" or glossing over fragments of code. If you can't explain *why* the turtle drew that spiral, you need to go back and study every line of code. Did the angle of turn keep on changing as the loop ran? Why would changing the step size that the turtle takes give us a spiral?

Reading code requires that you make sense of it in your mind, and can imagine how the statements work together to achieve the outcome. Practice "deep-reading" all code: what would happen if this statement was left out, or changed? If you're not sure, you can always just type the code in and check whether you have grasped the ideas well.

7.5. Readings

C# loops come in four different flavours. We've only seen the `while` loop here: you'll meet `for`, `foreach` and `do while` shortly.

Learn more about C# loops:

- http://en.wikibooks.org/wiki/C_Sharp_Programming/Control
- http://msdn.microsoft.com/en-us/library/2aeyhxcd(v=vs.100).aspx

7.6. Glossary

constructor

A special method that has the same name as the class that contains it. If it exists, it is automatically executed when we create any new instance of the class.

controlling expression

A test to control if the loop should be executed again or not.

instance

An object of a certain type, or class. Tess and Alex are different instances of the class `Turtle`.

loop body

Any number of statements nested inside the curly braces of a loop.

method

A method is a code block containing a series of statements.

Methods are defined within a class.

Invoking or activating the method causes the object to respond in some way, e.g. `Forward` is the method when we say `tess.Forward(100.0)`.

invoke

An object has methods. We use the verb invoke to mean *activate the method*. Invoking a method is done by putting parentheses after the method name, with some possible arguments. So `tess.Forward(20.0)` is an invocation of the `Forward` method.

loop

A construct in the programming language that lets us repeatedly execute statements.

object

A "thing" to which a variable can refer. This could be a canvas, a button, or one of the turtles we have created.

property

Some state or value that belongs to an object. For example, Tess has a brush that she uses for painting her trail.

`while` loop

A statement in C# for convenient repetition of statements in the *body* of the loop.

7.7. Exercises

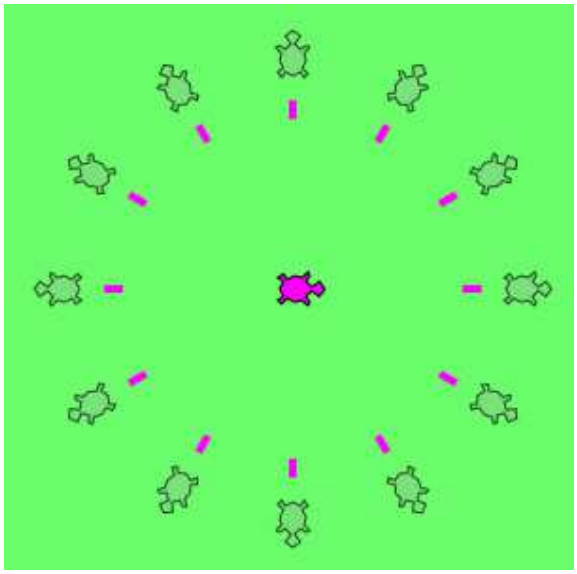
1. Give three properties of your cellphone object. Give three methods of your cellphone.
2. In the last spiral drawing it looks like Alex ended up facing the same direction as he started in. Prove that this is indeed the case.
3. Suppose our turtle Tess is at heading 0 — facing East. We execute the statement `tess.Left(3645);` What does Tess do, and what is her final heading?
4. Provide four buttons on you GUI to allow the user to get Tess to draw any any of these regular polygons (regular means all sides the same length, all angles the same.)
    - An equilateral triangle
    - A square
    - A hexagon (six sides)
    - An octagon (eight sides)
5. If you were going to draw a regular polygon with 18 sides, what angle would you need to turn the turtle at each corner?
6. Add buttons to your GUI to allow allow the user to clear the drawings of all turtles.
7. Add a slider control to your GUI that enables the user to set Tess' brush width to any value between 1 and 20.
8. Add a sub-menu with items that will allow the user to choose one of three brushes for Tess.
9. Use a loop to make a turtle draw a shape like this:

Hints:

- Try this on a piece of paper, moving and turning your cellphone as if it was a turtle. Watch how many complete rotations your cellphone makes before you complete the star. Since each full rotation is 360 degrees, you can figure out the total number of degrees that your phone was rotated through. If you divide that by 5, because there are five points to the star, you'll know how many degrees to turn the turtle at each point.

10. In this chapter we introduced the `while` loop, which repeated statements in the loop body. Do you think one of the statements in the loop body could be another loop? Or, asked another way, can we nest our loops? Experiment by adapting the code from the previous exercise to draw this:

11. Make a turtle draw a face of a clock that looks something like this:



12. Do you think it should be possible to have more than one playground in a program? Write an application that puts Alex and Tess each in their own playgrounds.
13. What is the collective noun for turtles? (Hint: they don't come in *herds*, like this chapter suggested.)

8. Void Methods

8.1. Methods

In C#, a **method** is a named sequence of statements that belong together. Their primary purpose is to help us organize programs into chunks that match how we think about the problem.

There are two main kinds of methods that we work with:

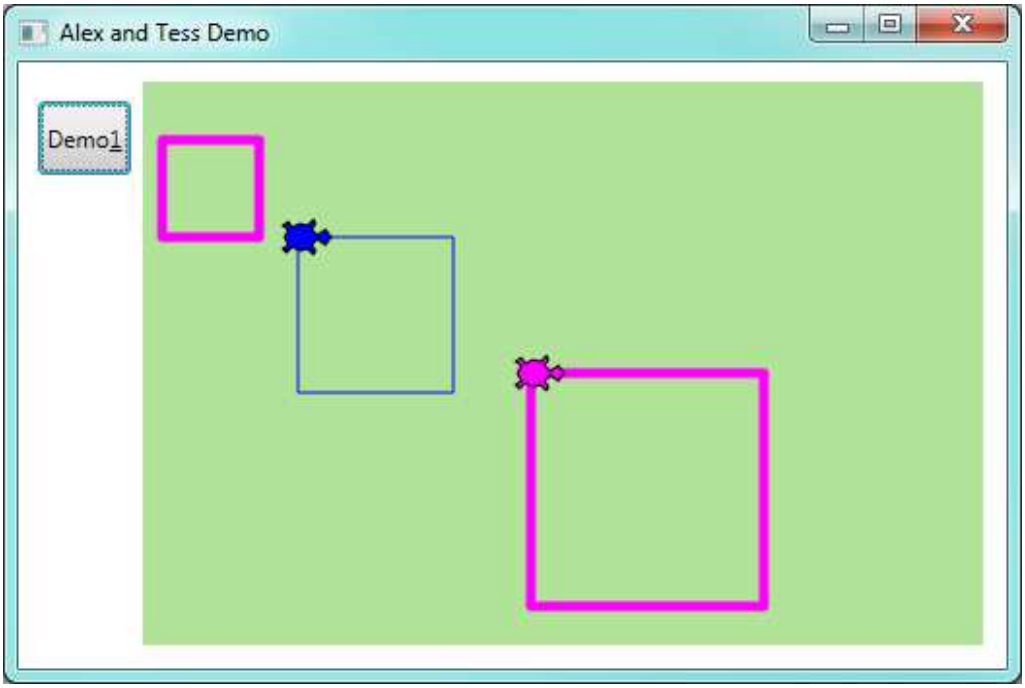
- **value-returning methods** which are executed because we want them to do some calculation and return a value,
- **void methods** which do not return a value. They are executed because they *do* something interesting.

We cover the void methods in this chapter, and talk about value-returning methods a little further into the course.

Suppose we're working with turtles, and we find that we're having to draw many squares, all of different sizes. "Draw a square" is an *abstraction*, or a mental chunk, of a number of smaller steps. So let's write a method to capture the pattern of this "building block": We'll make these changes to our Alex and Tess program from the previous chapter. This will be a void method, because our goal is not to calculate a result: our goal is to get the turtle to *do* something.

```
1  private void drawSquare(Turtle t, double sz)  // t and sz are parameters
2  {
3      int side = 0;
4      while (side < 4)
5      {
6          t.Forward(sz);
7          t.Right(90);
8          side = side + 1;
9      }
10 }
11
12 private void btnDemo1_Click(object sender, RoutedEventArgs e)
13 {
14     tess.WarpTo(10.5, 30.8);
15     drawSquare(tess, 50.2);      // this is a call-site where we use our method
16
17     alex.WarpTo(80.0, 80.0);
18     drawSquare(alex, 80.0);
19
20     tess.WarpTo(200.0, 150.0);
21     drawSquare(tess, 120.0);     // tess and 120.0 are arguments
22 }
```

Clicking the button now produces this:



Every method has a **method signature** followed by a **method body**.

Here the method signatures are at lines 1 and 12.

The body of the `DrawSquare` method contains everything from line 2 to line 10, while the body of the `btnDemo1_Click` method is at lines 13–22.

The signature contains a name for our method — in our example we called it `drawSquare`. The signature also has two **parameters** that make it general: one to tell the method which turtle to use, and the other to tell it the size of the square to draw.

Defining a new method does not make the method run. To **invoke** the method we need a **method call**. We've already seen how to call some of the turtle methods like `Right` and `Forward`. A method call requires the name of the method followed by some values, called **arguments**. The arguments are **passed** to the method by assigning them to the parameters in the method.

At this stage, if your method has two parameters, your **call sites** — the places in our code where we call the method — should have two matching arguments. (This rule is relaxed in more advanced C#.) And any argument value you provide must be of an appropriate type that can be assigned to its corresponding parameter.

We use the terms *call a method* and *invoke a method* interchangeably — they mean the same thing.

So lines 15, 18, and 21 are our call sites. We call our new method three times in this code, passing different arguments each time. So, for example, when the method is invoked from line 15, the variable `sz` is assigned the value 50, and the variable `t` refers to Tess.

Once we've defined a method, we can call it as often as we like, and its statements will be executed each time we call it. And, as we see above, we can use it to get any of our turtles to draw a square of any size.

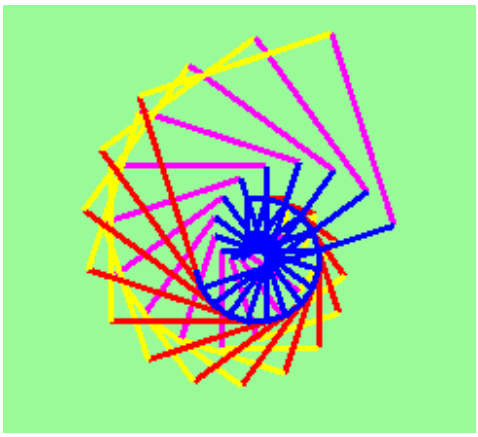
Parameters play a very interesting role in methods. They allow us to generalize the methods. Look again at `drawSquare`. We could have written it without the `sz` parameter, and simply hard-coded a value like 100 into line 6. But then our method would be less general: it would only be able to draw squares of

size 100. Similarly, we could have omitted the Turtle parameter from the method and hard-coded to use some specific turtle, say Tess, at lines 6 and 7. Once again, that would make the method less general.

An important thinking skill when we create our own methods is to decide what parts of the task (or mental chunk) that we're coding for must be fixed for all uses of the methods, and what parts should be *pluggable* to make the method general. Parameters provide the mechanism to plug different values into the method at each different call site.

In the next example, we've created a variation, and we get Alex to draw 15 squares.

```
1  private void drawMulticolorSquare(Turtle t, double sz)
2  {
3      Brush[] bs = { Brushes.Red, Brushes.Yellow, Brushes.Magenta, Brushes.Blue };
4      foreach (Brush b in bs)
5      {
6          t.LineBrush = b;
7          t.Forward(sz);
8          t.Right(90.0);
9      }
10 }
11
12 private void btnDemo1_Click(object sender, RoutedEventArgs e)
13 {
14     alex.BrushWidth = 3.14;
15     alex.Visible = false;
16     double size = 20.0;
17     for (int i = 0; i < 15; i = i + 1)
18     {
19         drawMulticolorSquare(alex, size);
20         size = size + 6.1;        // Increase the size for next time
21         alex.Forward(10.0);       // Move alex along a little
22         alex.Right(18.0);
23     }
24 }
```



Here we've sneaked in arrays, and two more of C#'s looping mechanisms.

Line 3 creates an *array* of values — four in this case. Each value is a different colour brush. Line 4 *iterates* through all the elements in the array using C#'s `foreach` loop. So on the first pass of the loop, the variable `b` is assigned (given) the value `Brushes.Red`. Then the body of the loop is executed. Next time, `b` is assigned the value `Brushes.Yellow`. Again the body of the loop is executed (and it sets the turtle brush to `b`, draws a side, turns the corner). So in a `foreach` loop, the number of times the loop is executed depends on how many elements are in the array. In our example, we have four different brushes in the array, so we draw four sides.

Line 17 introduces the third flavour of C# loops: the `for` loop. Look up the `for` loop in Microsoft's C# Reference manual.

8.2. Methods can call other Methods

Let's assume now we want a method to draw a rectangle. (A rectangle is a generalization of a square.) We need to be able to call the method with different arguments for width and height. And, unlike the case of the square, we cannot repeat the same thing 4 times, because the four sides are not equal.

So we eventually come up with this rather nice code that can draw a rectangle.

```
1  private void drawRectangle(Turtle t, double wSz, double hSz)
2  {
3      for (int i = 0; i < 2; i = i + 1)
4      {
5          t.Forward(wSz);
6          t.Right(90.0);
7          t.Forward(hSz);
8          t.Right(90.0);
9      }
10 }
```

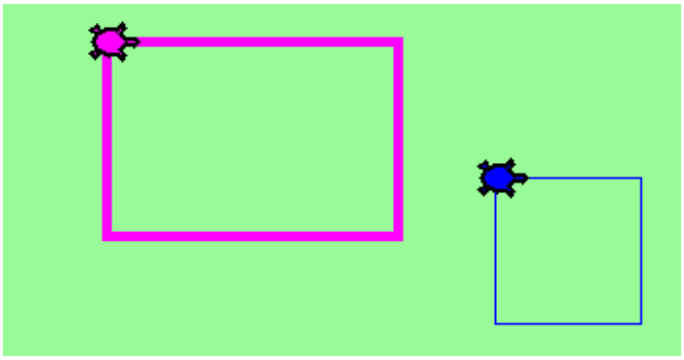
The parameter names are deliberately chosen to be short to ensure they're not misunderstood. In real programs, once we've had more experience, we will insist on better variable names than this. But the point is that the program doesn't "understand" that we're drawing a rectangle, or that the parameters represent the width and the height. Concepts like rectangle, width, and height are the meaning we humans have, not concepts that the program or the computer understands.

Thinking like a scientist involves looking for patterns and relationships. In the code above, we've done that to some extent. We did not just draw four sides. Instead, we spotted that we could draw the rectangle as two halves, and used a loop to repeat that pattern twice.

But now we connect some more dots in our head: we realize that a square is a special kind of rectangle. We already have a method that draws a rectangle, so we can use that to draw our square too.

```
1  private void drawSquare(Turtle tx, double sz)   // a new version
2  {
3      drawRectangle(tx, sz, sz);
4  }
5
6  private void btnDemo1_Click(object sender, RoutedEventArgs e)
7  {
8      drawRectangle(tess, 150.0, 100.0);
9      drawSquare(alex, 75.0);
10 }
```

After clicking the button we'll find this in our playground:



There are some points worth noting here:

- Methods can call other methods.
- Rewriting `drawSquare` like this captures the relationship that we've spotted between squares and rectangles.
- A caller of this method might say `drawSquare(alex, 75.0)`. The parameters `tx` and `sz`, are given the values of alex and 75.0 respectively.
- In the body of `drawSquare`, the parameters are just like any other variable.
- When the call is about to be made to `drawRectangle`, the values in variables `tx` and `sz` are fetched first, then the call happens. So as we enter the top of method `drawRectangle`, its parameter `t` is assigned the Tess object. `wSz` and `hSz` are both given the value 75.
- In this example we've chosen different names for the parameters in `drawSquare` and in `drawRectangle`. But the names are what we call **local** names: You can choose any name you like for the parameters of a method. Even if you re-use the same name as is used in some other method, they are different variables. (Different families could have a dog called Rex: the same name doesn't mean they're the same dog!) So make sure you still understand the argument passing if the parameter name in `drawSquare` is changed to `t` instead of `tx`.

So far, it may not be clear why it is worth the trouble to create all of these new methods. There are many good reasons, but this example demonstrates two:

1. Creating a new method gives us an opportunity to name a group of statements. Methods can simplify a program by hiding a complex computation behind a single call. The method (including its name) can capture our mental chunking, or *abstraction*, of the problem.
2. Creating a new method can make a program smaller by eliminating repetitive code.

8.3. Flow of Control

We can imagine the computer has a moving finger which always points to the next statement to be executed. The way the finger moves from one statement to the next is called the **flow of control**, sometimes also called **control flow**.

We've see event handlers attached to events: when a button is clicked, the flow of control executes the statements in the handler one after the other, until it leaves the handler and goes back to wait for another event.

Method calls are a detour in the control flow. Instead of going to the next statement, the control jumps to the first line of the called method, executes all the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until we remember that one method can call another. While in the middle of one method, the program might have to execute the statements in another method. But while executing that new method, the program might have to execute yet another method!

C# keeps track of where it is, so each time a method completes, control returns to where it was called from.

Watch the control flow in action

See a video demonstration at http://www.ict.ru.ac.za/resources/thinksharply/videos/debugging1.avi

In Visual Studio (VS), we can debug our programs and "single-step" through any program. VS will highlight each line of code just before it is about to be executed.

To get into *debugging mode*, we set **breakpoints** in the code. This fragment shows a breakpoint set on line 50. (Set or remove the breakpoints by clicking where the red icon is shown to the left of line 50.) The red highlighted section shows that execution will be interrupted before the first step of the `for` loop (defining the variable `i` and initializing it to zero.)

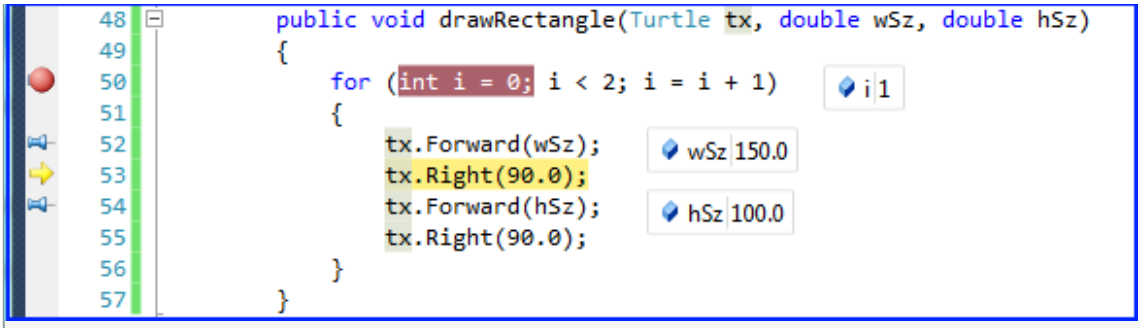
```
47
48          public void drawRectangle(Turtle tx, double wSz, double hSz)
49          {
50              for (int i = 0; i < 2; i = i + 1)
51              {
52                  tx.Forward(wSz);
53                  tx.Right(90.0);
54                  tx.Forward(hSz);
55                  tx.Right(90.0);
56              }
57          }
```

Once breakpoints are set we can execute the program, click its buttons, etc. and when we reach a breakpoint VS enters **debugging mode**. (VS has at least three modes: editing, running and debugging. We know which mode we're in by watching the title bar of VS: when editing, only the name of your solution is shown. Otherwise VS shows the name of the solution followed by "(Debugging)" or by "(Running)".)

In debugging mode we can single-step (key F11 executes the next instruction), or we can hover the cursor over any variable to get VS to pop up the current value of that variable. So this makes it easy to inspect the state of the program — the current values that are assigned to the program's variables. Below we see that we've stepped through a few statements and we're about to execute line 55. We've hovered the cursor over the variable `hSz` and we see that it currently has the value 100.0.

```
46
47
48          public void drawRectangle(Turtle tx, double wSz, double hSz)
49          {
50              for (int i = 0; i < 2; i = i + 1)
51              {
52                  tx.Forward(wSz);
53                  tx.Right(90.0);
54                  tx.Forward(hSz);
55                  tx.Right(90.0);   hSz 100.0
56              }
57          }
```

Notice also that there is a little pin on the pop-up. If you pin the pop-up, it remains there. So you can pin values for many variables and see how they change as you execute your code step by step. So here is the same code with three variables pinned for inspection:



Single-stepping is a powerful mechanism for building a deep and thorough understanding of what is happening as each statement is executed. Learn to use breakpoints and single-stepping feature well, and be mentally proactive: as you work through the code, challenge yourself before each step: *"What changes will this line make to any variables (or *state*) in my program?"** and *"Where will flow of execution go next?"*

After a few steps of debugging you can go back to running mode by removing the breakpoint and clicking the Run icon again.

8.4. A Turtle Bar Chart

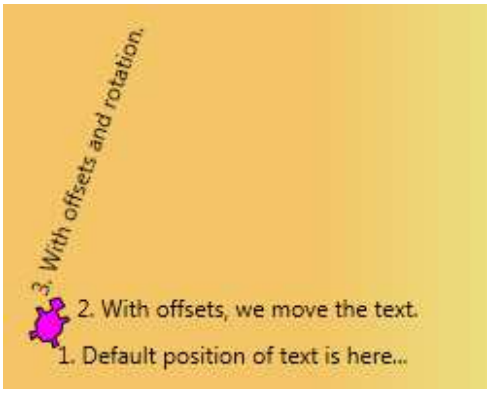
Let's do a slightly bigger example now.

The turtle has a lot more power than we've seen so far. Details are in the documentation for the turtle, in the appendix of this book.

Here are a couple of new tricks for our turtles:

- We can get a turtle to display text at its current position. The method to do that is `alex.Stamp("Hello")`. The argument is the text to be displayed — it must be a string. But the Stamp method takes some extra *optional* arguments which can change the position and rotation of the text. If you use the method with only the one string argument, it creates a rectangular label containing the text, and places the top left edge of the label at the turtle position. So the text lines up below the turtle, and is written horizontally across the screen.
- If you supply the next two arguments e.g. `alex.Stamp("Hello",5, -20)` the label will be offset an additional 5 units in the X direction, and -20 in the Y direction. So the text will now appear further to the right of the turtle, and above the turtle, 20 units nearer the top edge of the screen.
- And there is still another optional argument. Once we know where the text should go relative to a turtle who is facing east (its Heading is 0.0 degrees), we can provide a Boolean argument to tell the system to rotate the text to the same heading as the turtle. So here is a small sample that shows the options we have:

```
1    tess.Reset();
2    tess.Left(70);
3    tess.Stamp("1. Default position of text is here...");
4    tess.Stamp("2. With offsets, we move the text. ", 10.0, -24.1);
5    tess.Stamp("3. With offsets and rotation.", 10.0, -24.1, true);
```



- Every turtle has some properties called TextBrush, TextFontFamily, TextFontSize, TextFontStyle, TextFontWeight, which we can set: so we can get our turtle to create huge text in hot pink colours with our favourite font. Experiment a bit.
- Turtles usually draw shapes with lines: but there is a property called `Filling` that tells the turtle to fill the interior of its shapes rather than draw an outline. Along with this we can set the FillBrush property (If we don't set it, the turtle's LineBrush will be used to fill too.)

Ok, so can we get Tess to draw a bar chart? Let us start with some data to be charted,

```
int[] xs = { 24, 57, 100, 120, 80, 130, 110 };
```

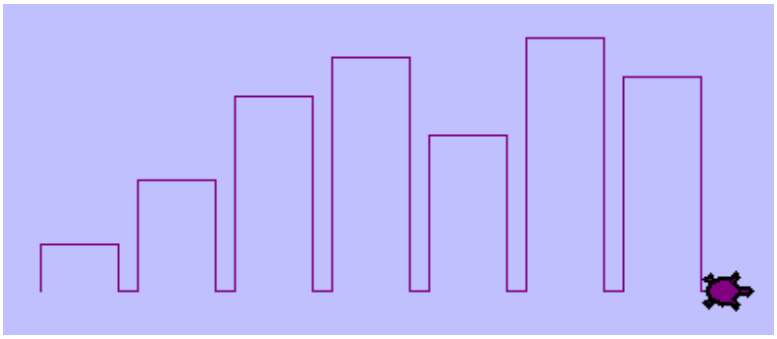
To tackle the problem, we think about how to break it into decent sub-problems or chunks. A good building block is to be able to draw a single bar of a given height, with a fixed width. So let's make a void method to capture this mental chunking:

```
1  private void draw_bar(Turtle t, int height)
2  {
3      // Get turtle t to draw one bar, of height.
4      t.Left(90);
5      t.Forward(height);      // Draw up the left side
6      t.Right(90);
7      t.Forward(40);          // Width of bar, along the top
8      t.Right(90);
9      t.Forward(height);      // And down again!
10     t.Left(90);             // Put the turtle facing the way we found it.
11     t.Forward(10);          // Leave small gap after each bar
12 }
```

Now to draw the whole chart we need to repeatedly call our method. So code like this could appear in a handler:

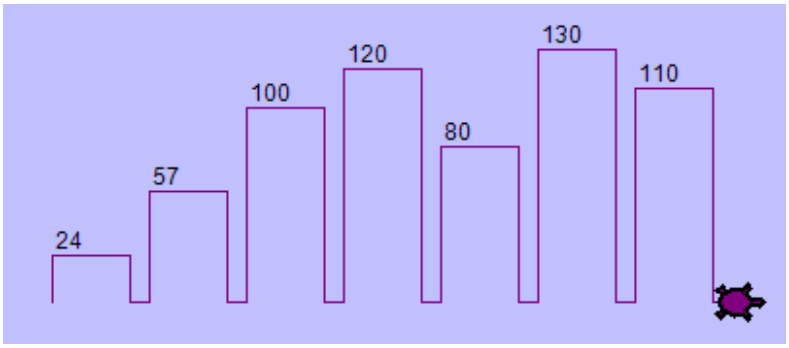
```
1  private void drawChart_Click(object sender, RoutedEventArgs e)
2  {
3      tess.Reset();
4      int[] xs = { 24, 57, 100, 120, 80, 130, 110 };
5      tess.LineBrush = Brushes.Purple;
6
7      foreach (int v in xs)
8      {
9          draw_bar(tess, v);
10     }
11 }
```

What we get is not fantastically impressive, but it is a good start!



Notice that we used an array of integers and integer angles for turning the turtle. But, of course, C# can see that the turtle's methods take double parameters. C# will do the type conversions automatically for us, because it is always safe to convert an int to a double. But C# won't try to do the automatic conversion in the opposite direction, because converting a double to an int can lose information. If we want to do a conversion that potentially could discard some precision, we have to do it ourselves.

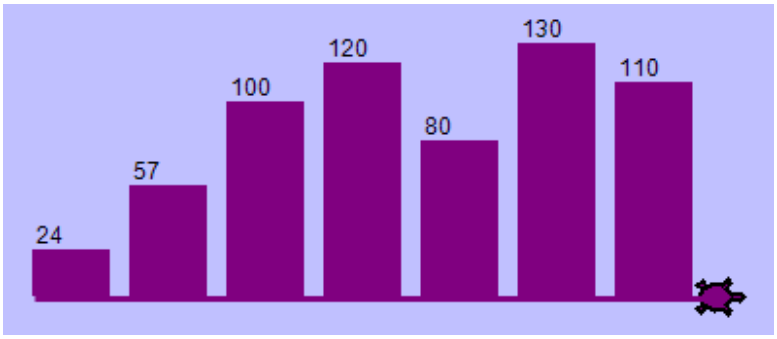
Next, at the top of each bar, we'll print the value of the data. We'll do this in the body of `draw_bar`, by adding `t.Stamp(string.Format("{0}", height), 0, -24);` just before line 7. The result looks much better now:



Now let's turn on the filling capability for a nice effect. Our final handler now looks like this:

```
1   private void drawChart_Click(object sender, RoutedEventArgs e)
2   {
3       int[] xs = { 24, 57, 100, 120, 80, 130, 110 };
4       tess.Reset();
5       tess.LineBrush = Brushes.Purple;
6
7       tess.Filling = true;
8       foreach (int v in xs)
9       {
10          draw_bar(tess, v);
11      }
12      tess.Filling = false;
13  }
```

It produces the following very satisfying result:



A small challenge

Brushes are very flexible things. A LinearGradientBrush gives a blend of colours along a certain angle. A RadialGradientBrush is also quite interesting. Turtles have different brushes for their line operations and their filling operations. So after line 5, you could try one of these.

```
1  tess.FillBrush = new LinearGradientBrush(Colors.Cyan, Colors.Red, 45);
2  tess.FillBrush = new RadialGradientBrush(Colors.Yellow, Colors.Magenta);
```

8.5. Glossary

argument

A value provided to a method when the method is called. This value is assigned to the corresponding parameter in the method. The argument can be the result of an expression which may involve operators, operands and calls to other value-returning methods.

call site

A place in our code where we call a method.

flow of execution

The order in which statements are executed.

method

A named sequence of statements that performs some useful operation. Methods may or may not take parameters. In this chapter we've seen void methods (they don't return any result). In a future chapter we'll see value-returning methods.

method body

The block of statements that come after the method signature. The body always starts and ends with matching braces { }.

method call

A statement that causes a method to execute. It consists of the name of the method followed by arguments enclosed in parentheses.

method signature, signature

The initial part of a method which contains details about its name, it's return type (or the special keyword `void`), and its parameters. (Some headers may use more advanced features — for example, the keyword `private` will be covered later.) See *method body*.

lifetime

Variables and objects have lifetimes — they are created at some point during program execution, and are destroyed at some later time.

local variable

A variable defined inside a method. A local variable can only be used inside its method. Parameters of a method are also special kinds of local variables.

parameter

A name used inside a method to refer to the argument which was passed to it from the call site.

refactor

A fancy word to describe reorganizing our program code, usually to make it more understandable. Typically, we have a program that is already working, then we go back to "tidy it up". This often involves writing good comments, choosing better variable names, simplifying the code where we can, or spotting repeated patterns and moving that code into a method.

signature

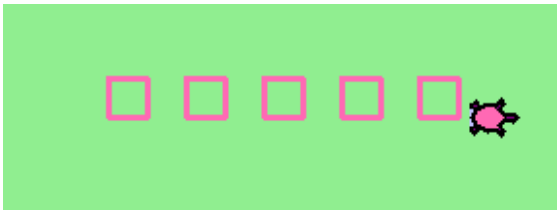
See *method signature*.

void method

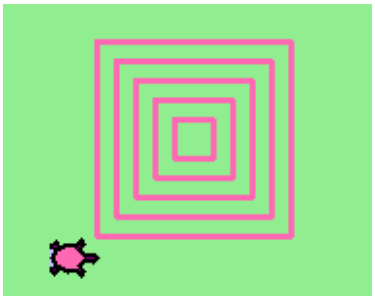
The opposite of a value-returning method: one that does not return a value. It is executed for the work it does, rather than for the value it returns.

8.6. Exercises

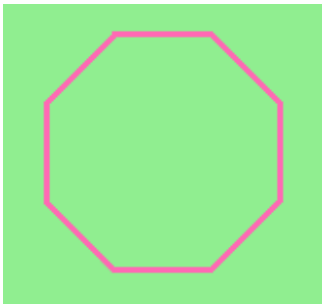
1. Write a void method to draw a square. Use it in a program to draw the image shown below. Assume each side is 20 units. (Hint: notice that the turtle has already moved away from the ending point of the last square when the program ends.)



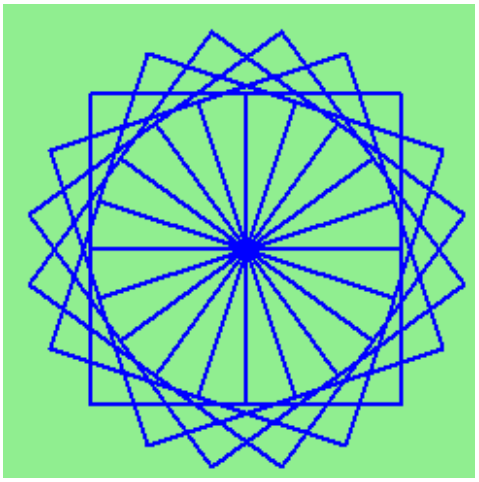
2. Write a void method to draw this. Assume the innermost square is 20 units per side, and each successive square is 20 units bigger, per side, than the one inside it.



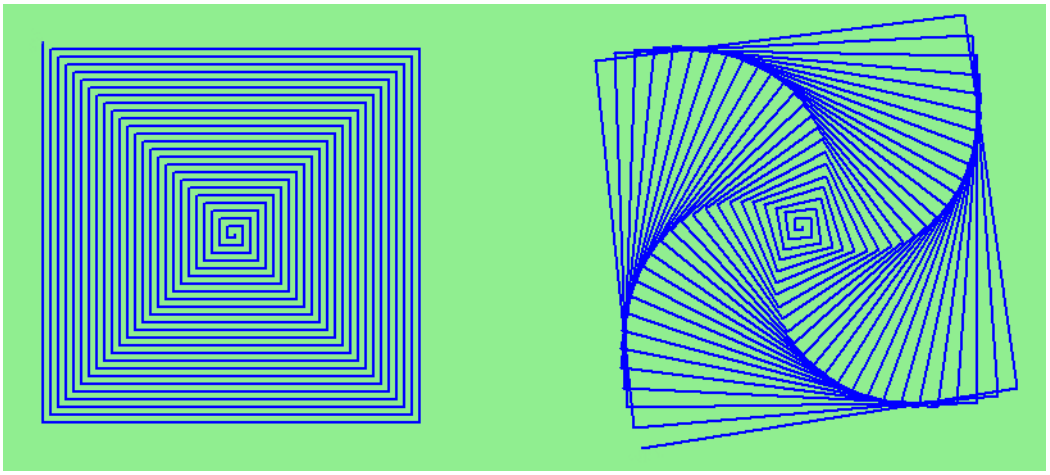
3. Write a void method `draw_poly(t, n, sz)` which makes a turtle draw a regular polygon. When called with `draw_poly(tess, 8, 50)`, it will draw a shape like this:



4. Draw this pretty pattern.

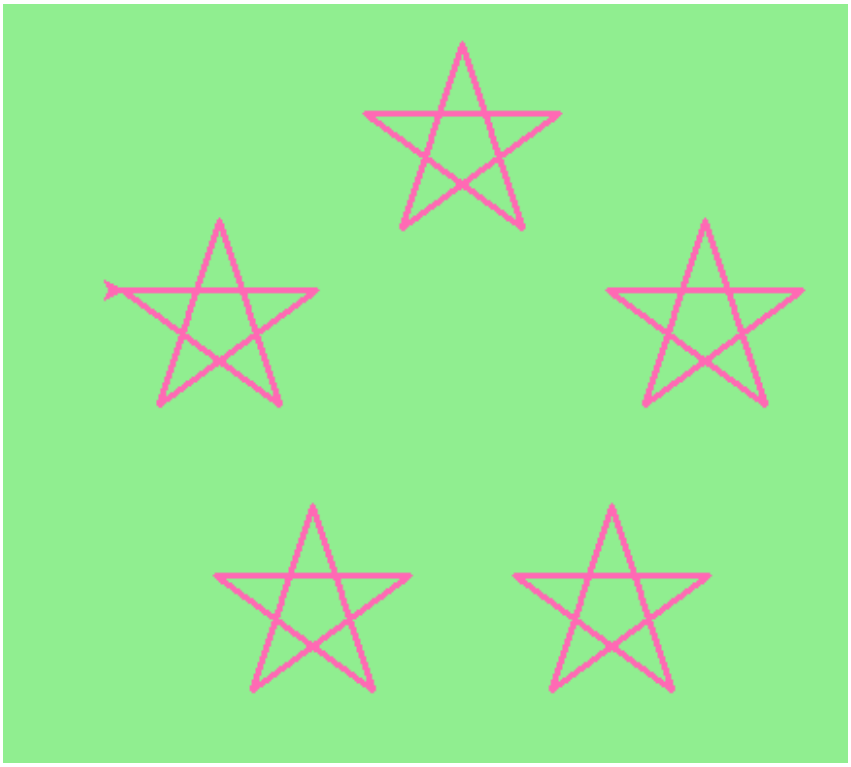


5. The two spirals in this picture differ only by the turn angle. Draw both.



6. Write a void method `draw_equitriangle(t, sz)` which calls `draw_poly` from the earlier question to have its turtle draw a equilateral triangle.
7. Write a void method to draw a star, where the length of each side is 50 units. (Hint: You should turn the turtle by 144 degrees at each point.)

8. Extend your program above. Draw five stars, but between each, pick up the pen, move forward by 175 units, turn right by 144, put the pen down, and draw the next star. You'll get something like this:



What would it look like if you didn't pick up the pen?

9. Study the turtle bar chart program, and determine what you think will happen if we provide some negative values in the data to be charted. Then try it out and confirm that you understand it well! Then peek ahead at the next chapter, see how the `if` statement works, and change the turtle bar chart program so that positive bars are filled with a blue brush, and negative bars are filled with a red brush. Also arrange that the value that labels negative bars displays below the bar, rather than inside it.

9. Working with Booleans and Conditional Statements

Programs get really interesting when we can test conditions and change the program behaviour depending on the outcome of the tests. That's what this chapter is about.

9.1. Boolean values and expressions

A *Boolean* value is either true or false. It is named after the British mathematician, George Boole, who first formulated *Boolean algebra* — some rules for reasoning about and combining these values. This is the basis of all modern computer logic.

In C#, the two Boolean values are `false` and `true` (remember that this language is case sensitive, so they must be exactly as shown), and the C# type is `bool`. We can define and initialize variables of type `bool`:

```
1  bool passwordIsValid = false;
2
3  ...
4  if (enteredPassword == "magic")
5  {
6      passwordIsValid = true;
7  }
8  else
9  {
10     passwordIsValid = false;
11 }
```

A **Boolean expression** is an expression that evaluates to produce a result which is a Boolean value. For example, the operator `==` tests if two values are equal. It produces (or *yields*) a Boolean value. So another much more compact way to write lines 4 - 11 in the above code would be like this:

```
1  passwordIsValid = enteredPassword == "magic";
```

Remember the rule for assignment: evaluate the right side of the assignment first. Then assign the resulting value to the variable on the left. Convince yourself that the short version is equivalent to the longer version, and ensure that you understand the *generalization* here: variables, expressions and assignment are not just restricted to simple number types: some expressions can produce Boolean results, some can produce string results, some integer results, etc. So assignment is very general, and will work as long as we don't violate the C# rules about types.

9.2. Comparison operators

The `==` operator is one of six common **comparison operators** which all produce a `bool` result; here are C# expressions showing all six:

```
                    //  Evaluates to true if _________________
x == y              //    x is equal to y
x != y              //    x is not equal to y
x > y               //    x is greater than y
x < y               //    x is less than y
x >= y              //    x is greater than or equal to y
x <= y              //    x is less than or equal to y
```

Although these operations are probably familiar, the C# symbols are different from the mathematical symbols. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a comparison operator. Also, there is no such thing as =< or =>.

These comparison operators will work for most simple types like `int`, `double` and `char`. The operators that test for equality (== and !=) also work for strings, but we unfortunately cannot use the ordering operators (< , <=, >, or >=) for comparing strings. (There is another way to compare strings which we'll get to in the chapter about strings.)

Like any other types we've seen so far, Boolean values can be assigned to variables (of type `bool`), or used in a `String.Format` method in much the same way as we previously used `int` or `double` values.

```
int age = 23;
...
bool old_enough_to_get_driving_licence = age >= 18;
string s = string.Format("It says {0}.", old_enough_to_get_driving_licence);
```

9.3. Logical operators

There are three **logical operators** associated with Booleans: *and*, *or*, and *not*. C# uses the syntax &&, ||, and ! for these. So when we see these C# tokens we'll pronounce && as *and*, we'll say *or* when we read the token ||, and we'll pronounce ! as *not*.

Logical operators allow us to build more complex Boolean expressions from simpler Boolean expressions. For example, `(x > 0) && (x < 10)` evaluates to `true` only if `x` is greater than 0 *and* at the same time, x is less than 10.

This shorthand is *not* allowed

In maths we often see shorthand such as $0 < x < 10$. There are really two separate sub-conditions here. In C# we have to write those two parts separately as

```
...      (0 < x) && (x < 10)
```

The parentheses could have been left out, but it looks nicer grouped like this, and we don't have to remember whether the && or the < operator takes precedence (gets done first).

`e1 || e2` is `true` if *either* of the conditions is `true`. For example, `(n % 2 == 0) || (n % 3 == 0)` determines if the number `n` is divisible by 2 *or* if it is divisible by 3. (What do we think happens if `n` is divisible by both 2 and by 3 at the same time? What is the smallest positive integer that can be divided exactly by both 2 and 3? Will the expression yield `false` or `true`? Try it.)

Finally, the ! logical operator negates a Boolean value, so `!(x > y)` is `true` if `(x > y)` is `false`, that is, if `x` is less than or equal to `y`.

The expression on the left of the || operator is evaluated first: if the result is `true`, C# does not (and need not) evaluate the expression on the right — this is called *short-circuit evaluation*. Similarly, for the *and* operator, if the expression on the left of && yields `false`, C# does not need to, nor attempt to, evaluate the expression on the right.

So there are no unnecessary evaluations.

Short circuit evaluation means that the order in which one writes the tests can make a difference. And programmers take advantage of that fact. For example, dividing by zero will cause a run-time error. Consider these two fragments of code:

```
...      (k != 0) && (x / k > 10)     // this works even when k == 0
...      (x / k > 10) && (k != 0)     // this crashes when k == 0
```

If `k` is zero, the first one works, and returns `false` without even attempting to do the problematic division. But the second one will crash. So we have to be careful to order our expressions correctly.

9.4. Truth Tables

A truth table is a small table that allows us to show all the possible inputs, and to give the results for the logical operators. Because the `&&` and `||` operators each have two operands, and each operand can have one of two values, there are only four rows in a truth table that describes the semantics (meaning) of these operators:

e1	e2	(e1 && e2)
false	false	false
false	true	false
true	false	false
true	true	true

In a Truth Table, we sometimes use T and F as shorthand for the two Boolean operands: here is the truth table describing `||`:

e1	e2	(e1 || e2)
F	F	F
F	T	T
T	F	T
T	T	T

The third logical operator, `!`, only takes a one operand, so describing its operation using a truth table only needs two rows:

e	! e
F	T
T	F

9.5. Simplifying Boolean Expressions

Any set of rules for simplifying and rearranging expressions is called an *algebra*. For example, we are all familiar with school algebra rules, such as *n times 0 is 0*.

Here we see a different algebra — the *Boolean* algebra — which provides a slightly different set of rules for working with Boolean values.

First, some simplification rules involving the `&&` operator:

x && false	==	false

false && x	==	false
y && x	==	x && y [1]
x && true	==	x
true && x	==	x
x && x	==	x

Here are some corresponding rules for the || operator:

x || false	==	x
false || x	==	x
y || x	==	x || y [1]
x || true	==	true
true || x	==	true
x || x	==	x

Two ! operators cancel each other:

! (! x)	==	x

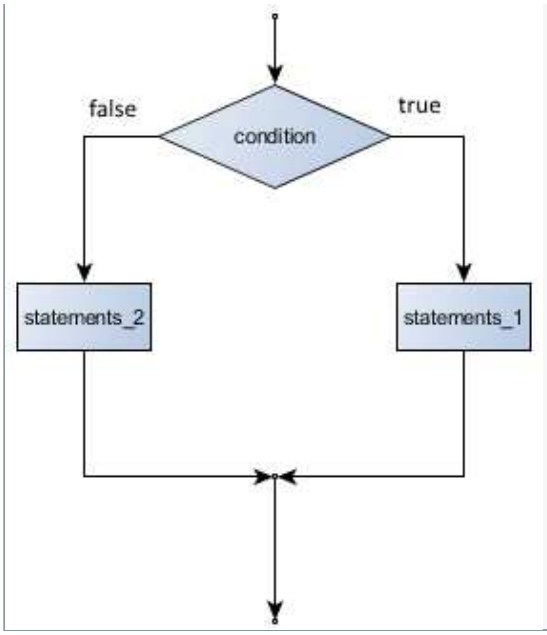
[1] *(1, 2)* In the algebra, these rules are valid. But because of short-circuit evaluation (described in the previous section), changing the order of evaluation of x and y might sidestep (or cause) some crashing expression. So in a program we need extra care when using && and ||.

9.6. Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behaviour of the program. **Conditional statements** give us this ability. The simplest kind of conditional statement is the **if** statement with an `else` part:

```
1  if (x % 2 == 0)
2  {
3      Console.WriteLine(string.Format("{0} is even.", x));
4      Console.WriteLine("Did you know that 2 is the only even prime number?");
5  }
6  else
7  {
8      Console.WriteLine(string.Format("{0} is odd.", x));
9      Console.WriteLine(
10        "Multiplying two odd numbers always gives an odd result!");
11 }
```

Flowchart of an if statement with an else part



The flow of control enters at the top and the condition is evaluated. If it evaluates to `true`, the first block of statements is executed. If the condition (which is a Boolean expression) evaluates to `false` the entire first block of statements is skipped, and instead the block of statements under the `else` clause is executed.

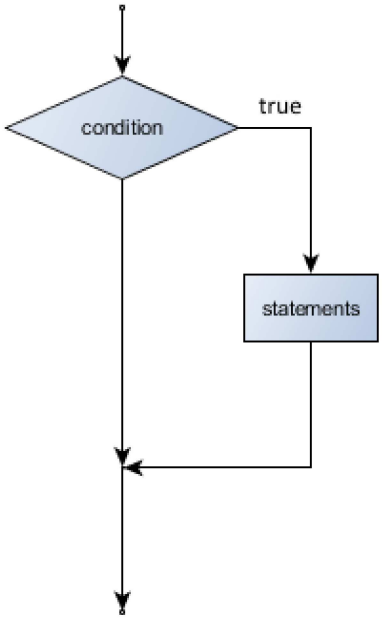
There is no limit on the number of statements that can appear in a block of statements — blocks can even be empty – like this — `{ }` for those occasions when we might want to "do nothing". (This often is the case when we create some scaffolding code and intend to return later with the exact details.) So we can encounter code like this:

```
1  bool computerTurnToPlay = true;
2
3  if (computerTurnToPlay)
4  {
5      // TODO: arrange logic for computer to make next move ...
6  }
7  else
8  {
9     // TODO: arrange logic for human to make the next move ...
10 }
11 computerTurnToPlay = ! computerTurnToPlay;
```

Can you see what we're doing at line 11 here?

9.7. Omitting the `else` part

Flowchart of an if statement with no else



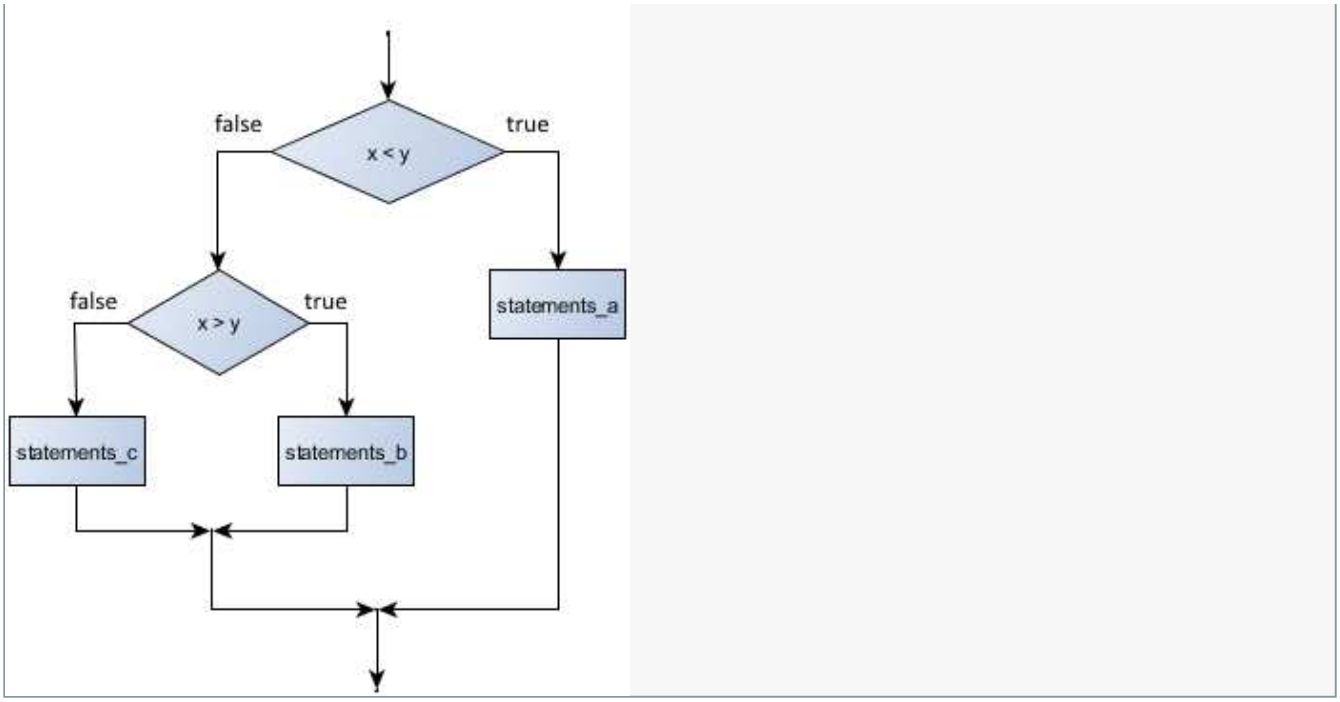
A simplified form of the `if` statement is one in which the `else` clause is omitted entirely. In this case, when the condition evaluates to `true`, the statements are executed, otherwise the flow of execution continues to the statement after the `if`.

```
if (x < 0) {
    msg = string.Format("Invalid negative number {0}. I'll use 42.", x);
    Console.WriteLine(msg);
    x = 42;
}
...
```

9.8. Nested conditionals

One conditional can also be **nested** within another. (We'll keep repeating the theme of composability — that each feature in the language can be re-used in other features to build bigger compositions.) Consider this flowchart:

Flowchart of a nested conditional



We could code it in C# like this:

```
1  if (x < y)
2  {
3      ... statements_a
4  }
5  else
6  {
7       if (x > y)
8       {
9          ... statements_b
10      }
11      else
12      {
13         ... statements_c  // executed when x == y
14      }
15 }
```

Nested conditionals very quickly become difficult to read and understand. It is probably a good idea to avoid them when we can.

Logical operators sometimes provide a way to simplify nested conditional statements. Say we start with this code:

```
1  if (x > 0)
2  {
3      if (x < 10)
4      {
5          Console.WriteLine("x is a positive single digit.");
6      }
7  }
```

Line 5 is only executed if we satisfy both the conditionals, so instead of the above we could make a more complex condition using the `&&` operator. Now we'll only need a single `if` statement, and it now

suggests to the human reader that both conditions must be mentally chunked, and are not separate unrelated things:

```
if ((x > 0) && (x < 10))
{
        Console.WriteLine("x is a positive single digit.");
}
```

9.9. Conditional Expressions

C# also has a conditional *expression* (not to be confused with the conditional *statement* – the `if` statement), that can be used in the middle of any other expression. The syntax is `condition ? e1 : e2`. Depending on `condition`, either `e1` or `e2` will become the result of evaluating the expression.

```
int bigger = x > y ? x : y;   // assign the larger of x and y to bigger
```

To emphasize that we can use this in the middle of any other expression, and the return type doesn't need to be an integer, or even a number, we could say

```
Console.WriteLine(string.Format("x is {0} 10.",
              x < 10 ? "less than" : "greater or equal to");
```

Conditional expressions seem more readable if we add unnecessary parentheses, especially around the conditional part:

```
Console.WriteLine(string.Format("x is {0} 10.",
            (x < 10) ? "less than" : "greater or equal to");
```

This again emphasizes *composability* — we can put expressions inside other expressions, and now we can also put conditional expressions inside other expressions. And of course, conditional expressions can nest inside other conditional expressions too!

9.10. Logical opposites

Each of the six relational operators has a logical opposite: for example, suppose we can get a driving licence when our age is greater or equal to 18, we can *not* get the driving licence when we are less than 18.

Notice that the opposite of >= is <.

operator	logical opposite
==	!=
!=	==
<	>=
<=	>
>	<=
>=	<

Understanding these logical opposites allows us to sometimes get rid of ! operators. Not operators are often quite difficult to read in computer code, and our intentions will sometimes be clearer if we can eliminate them.

For example, if we wrote this C#:

```
if (! (age >= 18)) {
    Console.WriteLine("Hey, you're too young to get a driving licence!");
}
```

it would probably be clearer to use the simplification laws, and to write instead:

```
if (age < 18) {
    Console.WriteLine("Hey, you're too young to get a driving licence!");
}
```

9.11. de Morgan's Laws

Two powerful simplification laws (called de Morgan's laws) are often helpful when simplifying Boolean expressions:

```
!(e1 && e2)  ==  (! e1) || (! e2)
!(e1 || e2)  ==  (! e1) && (! e2)
```

This is a bit like school algebra where we learned that we could take a factor into brackets, or take a factor out. Here we're taking negation into or out of the brackets.

For example, suppose we can slay the dragon only if our magic light-sabre sword is charged to 90% or higher, and we have 100 or more energy units in our protective shield. We find this fragment of C# code in the game [2]:

```
if (!((sword_charge >= 0.90) && (shield_energy >= 100)))
{
    Console.WriteLine("Your attack fails, the dragon fries you to a crisp!");
}
else
{
    Console.WriteLine("The dragon dies. You rescue the gorgeous princess!");
}
```

[2] Of course, in the real game we would have a profile that would indicate our preference for rescuing a gorgeous princess, a handsome prince, or both.

de Morgan's laws together with the logical opposites let us rework the condition into a (perhaps) easier to understand fragment:

```
if ((sword_charge < 0.90) || (shield_energy < 100))
{
    Console.WriteLine("Your attack fails, the dragon fries you to a crisp!");
}
else
{
    Console.WriteLine("The dragon dies. You rescue the gorgeous princess!");
}
```

Another way to eliminate the negation from the original example would be to swap around the `then` and `else` parts of the conditional. So here is a third version, also equivalent:

```
if ((sword_charge >= 0.90) && (shield_energy >= 100))
{
    Console.WriteLine("The dragon dies. You rescue the gorgeous princess!");
}
else
{
    Console.WriteLine("Your attack fails, the dragon fries you to a crisp!");
}
```

This last version is probably the best of the three, because it very closely matches the initial English statement. Clarity of our code (for other humans), and making it easy to see that the code does what was expected should always be a top priority.

As our programming skills develop we'll find we have more than one way to solve any problem. So good programs are *designed*. We make choices that favour clarity, simplicity, and elegance. The job title *software architect* says a lot about what — we are *architects* who make trade-off decisions in our products to balance beauty, functionality, simplicity and clarity in our creations.

Tip: Spend some time in playful mode

With programming there is often anxiety about whether we'll get it working by the deadline.

But once our program works, we can relax into a more playful mode where really deep learning and fun can happen. So we should play around a bit trying to polish it up. Write good comments. Would the code be clearer with different variable names? Could we have done it more elegantly? Should we rather use a method to chunk these statements that seem to do something sensible? Can we simplify the conditionals and the boolean expressions?

We think of our code as our creative expression, our work of art! We make it great, something we're proud of!

9.12. The `switch` statement

The `if` statement is useful for for choosing between two alternatives. By contrast, the `switch` statement handles multiple selections by passing control to one of the `case` statements within its body. It takes the following form:

```
switch (expression)
{
   case constant-expression-1:
      statements
      break
   case constant-expression-2:
      statements
      break
   ...
   default:
      statements
      break
}
```

The *switch expression* on line 1 must be a bool, char, string, int, or an enum (which we will cover later). Each constant expression in the different cases must be of this type too. Consider this sample method:

```
1  void funFact(string planetName)
2  {  // assign to fact something interesting about each planet
3    string fact = "";
4    switch (planetName.ToLower())  // converts the string all to lowercase letters
5    {
6      case "mercury":
7        fact = "Closest planet to the Sun.";
8        break;
9      case "venus":
10       fact = "Brightest object visible from Earth, apart from the Sun and the Moon.";
11       break;
12     case "earth":
13       fact = "We live here. More than 7 billion of us!";
14       break;
15     case "mars":
16       fact = "Curiosity landed here in 2012. It sent back great pictures.";
17       break;
18     case "jupiter":
19       fact = "One of two planets which are called Gas Giants.";
20       break;
21     case "saturn":
22       fact = "The second Gas Giant. It has spectacular rings.";
23       break;
24     case "uranus":
25       fact = "The first Ice Giant, with winds up to 900 km/h.";
26       break;
27     case "neptune":
28       fact = "The second Ice Giant, about 30 times further from the Sun than the Earth.";
29       break;
30     case "pluto":
31       fact = "After redefining the criteria for a planet, Pluto is no longer a planet.";
32       break;
33      default:
34       fact = string.Format("{0} Not a planet in this universe!", planetName);
35       break;
36     }
37     Console.WriteLine("{0}: {1}", planetName, fact);
38 }
```

At line 4 the expression is evaluated, and control is transferred to the matching case statement. The switch statement can include any number of cases, but no two case constants can be the same.

Notice that a `break` statement (or a `return` or some not-yet-covered ways of transferring control) is required after each case block.

If the switch expression does not match any case, the `default` block will be executed (if we provide one). But if we don't provide one, then nothing is executed and control jumps to the statement after the switch.

It is even possible to have multiple case labels apply to a single block of statements. For example,

```
1  switch (dayNum)   // assume days are numbered from 0 to 6, with 0 being Sunday
2  {
3      case 0: case 6:
4          Console.WriteLine("Weekend");
5          break;
6      case 1: case 2: case 3: case 4: case 5:
7          Console.WriteLine("Weekday");
```

```
8           break;
9   }
```

Cute, but is it necessary?

We can program without a `switch` statement: we can write equivalent code just using a bunch of `if` statements. Computer Scientists (the theoreticians, especially), like to identify which statements are "essential", and which are just "nice-to-have".

It turns out that one needs very few essential features in languages like this: some way of looping, (`while`), some way of testing (`if`) and some way of moving from one statement to the next. And we need some way to store values, and a way to get back the stored values (i.e. variables).

So `switch` is nice-to-have, not essential. The same is true for `for` and `foreach`: they can be written using `while`.

All the other features exist to make programming more convenient, and to capture our mental chunking and help us organize our thoughts better. But we could still get the computation done without all the nice-to-have extras like methods, `switch` and `foreach`.

9.13. Glossary

Boolean algebra

Some rules for rearranging and reasoning about Boolean expressions.

Boolean expression

An expression that, when evaluated, will result in either true or false.

Boolean value

There are only two possible Boolean values: `false` and `true`. Boolean values result when a Boolean expression is evaluated by C#. They have type `bool`.

branch

One of the possible paths of the flow of execution determined by conditional execution.

comparison operator

One of the six operators that compares two values: `==`, `!=`, `>`, `<`, `>=`, and `<=`.

condition

Another name for a Boolean expression.

conditional expression

An expression where the resulting value is a choice of one of two alternatives: a condition is evaluated to determine which choice gets made. The syntax is `(condition) ? e1 : e2`

conditional statement

A statement that controls the flow of execution depending on some condition. The `if` statement and the `switch` statements were covered in this chapter.

logical operator

One of the operators that combines or manipulates Boolean expressions: `&&`, `||`, and `!`.

nesting

Putting one program structure within another, such as a conditional statement inside a branch of another conditional statement.

switch statement

A statement that allows selection of one of many cases. Each case can contain a block of statements for execution.

truth table

A concise table of Boolean values that can describe the semantics (meaning) of an operator.

9.14. Exercises

1. Give the logical opposites of these conditions

    1. `a > b`
    2. `a >= b`
    3. `a >= 18  &&  day == 3`
    4. `a >= 18  &&  day != 3`

2. What do these expressions evaluate to? (Assume `x` has the value 15 and `y` has the value 10)

    1. `3 == 3`
    2. `3 != 3`
    3. `x >= y`
    4. `! (x < y)`
    5. `x % 2 == 0 ? "even" : "odd"`
    6. `x > y ? x : y`
    7. `x > y ? x-y ? y-x`

3. In the Conditional Expressions section we wrote an expression to find the bigger of two integers, and we assigned it to `bigger`. We also noted that conditional expressions can nest inside other conditional expressions. Write a single expression that returns the biggest of three integers, `x`, `y`, `z`.

4. Complete this truth table:

p	q	r	(! (p && q)) || r
F	F	F	?
F	F	T	?
F	T	F	?
F	T	T	?
T	F	F	?
T	F	T	?
T	T	F	?
T	T	T	?

5. Simplify these expressions or fragments of code:

    1. `!(subjectsPassed >= 4)`
    2. `! ((likeClickCount < 8) && (phoneMaker != "Samsung"))`
    3. `if (! (! oldEnoughToDrive)) { ... }`

6. Modify the turtle bar chart exercise from the earlier chapter so that the bar for any value of 100 or more is filled with red, values between [50 and 100) are filled with yellow, and bars representing

values less than 50 are filled with green. If our program can also deal with negative numbers, can we make this colour scheme reflect on the X axis?

7. Floating point arithmetic is inaccurate. To understand why, on a piece of paper, divide 10 by 3 and write down the decimal result. We'll find it does not terminate, so we'll need an infinitely long sheet of paper. The *representation* of numbers in computer memory or on our calculator has similar problems: memory is finite, and some digits eventually have to be discarded or rounded. So small inaccuracies creep in. Try this code:

```
1  double a = Math.Sqrt(2.0);
2       // `a` is irrational, i.e. has an infinite number of non-repeating digits
3  double b = a * a;
4  bool is_B_two = (b == 2.0);    // After squaring a, do we arrive back at 2.0?
5  double error = b - 2.0;
6  string msg = string.Format(" a is {0}\n b is {1}\n is_B_two is {2}\n error is {3}",
7                                    a,         b,         is_B_two,         error);
8  MessageBox.Show(msg);
```

The take-away message is that we have to be especially careful with floating-point arithmetic. Floating point values are *approximations* to the actual values, so it is never safe to test floating point numbers for exact equality against each another. So even a number like `Math.PI` in C# is not exactly accurate: it is just "very close"!

If we have run this code we'll also notice an interesting thing: although the numbers do not *compare* as being equal to each other on line 5, the output shows `b is 2.0`. So the `string.Format` has kindly rounded our *very close* number to display it as human-friendly 2.0, but when we subtract it from 2.0 we still get a non-zero result! Beware!

10. Value-returning methods

10.1. Methods that return values

In an earlier chapter we wrote a void method, `drawSquare`. It was called when we wanted it to execute a sequence of steps that caused the turtle to draw.

Now we write the other kind of method: a value-returning method. In an earlier exercise we saw the standard formula for compound interest:

$$A = P\left(1 + \frac{r}{n}\right)^{nt}$$

Where,

- P = principal amount (initial investment)
- r = annual nominal interest rate (as a decimal)
- n = number of times the interest is compounded per year
- t = number of years

So let's code a method that can calculate according to this formula:

```
1  private double finalAmt(double p, double r, int n, double t)
2  {
3      // Apply the compound interest formula to p to produce the final amount.
4      double a = p * Math.Pow((1 + r/n), (n*t));
5      return a;
6  }
```

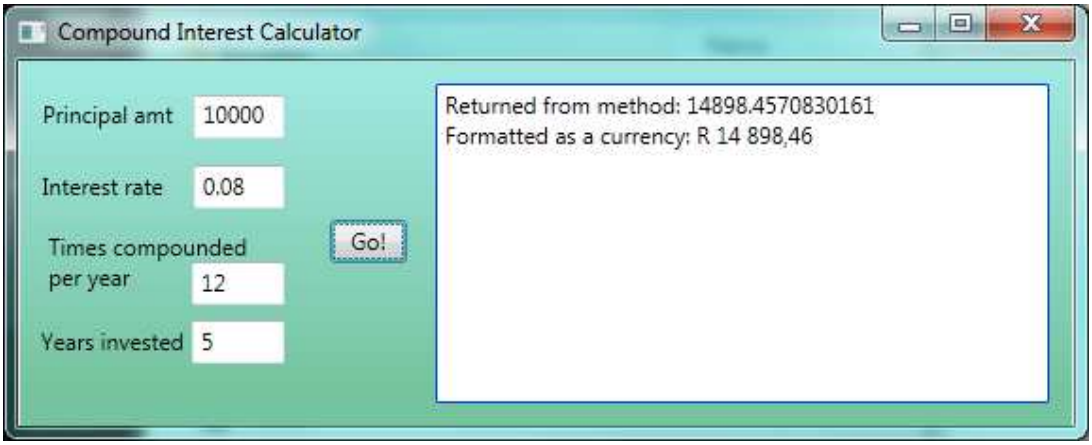
- On line 1 the **return type** of the method — the type of value it will return — is `double`. (In void methods the keyword `void` was used instead of a return type).
- Methods can return strings, booleans, doubles, integers, turtles, etc. — in fact, any type! But we must explicitly say what type the method will return.
- The **return** statement on line 5 is followed an expression. This expression will be evaluated and returned to the caller. If our method signature promised to return a double, we can't return a string or a boolean. [1]
- We've used the `Pow` method from the built-in `Math` module to raise one value to the power of another.
- Because we're working with money that has decimal points, we've used double types for the amount, the rate of interest, and the period (so 5.5 years is possible).

[1] In C#, a value-returning method must say what type it intends to return, and it must stick to its promise from the signature. Some other computer languages (e.g. Python) have a more relaxed attitude towards types. Sometimes a single method can return a string, at other times the same method might return an int or a double, or a turtle.

Once we have a method that captures our desired calculations, (and our mental chunking that allows us to think of a compound interest as a single calculation) it becomes a reusable chunk that we can call from elsewhere. We could use the method in a console-based program, a WPF program, or in some code behind a web page. Here we've created a simple WPF application and we show the handler which we've attached to the button.

```
private void btnGo_Click(object sender, RoutedEventArgs e)
{
    // The double.Parse(string s) method does the same
    // as Convert.ToDouble

    double toInvest = double.Parse(txtPrincipal.Text);
    double rate = double.Parse(txtInterestRate.Text);
    int n = int.Parse(txtNumTimesCompounded.Text);
    double t = double.Parse(txtYearsInvested.Text);

    double final = finalAmt(toInvest, rate, n, t);
    txtResult.AppendText(string.Format("Returned from method: {0}\n", final));
    txtResult.AppendText(string.Format("Formatted as a currency: {0:C}\n", final));
}
```

Running the program we get results similar to this:



- The first line of output is a bit messy with all these decimal places, but remember that C# doesn't understand that we're working with money: C# just does the calculation to the best of its ability, perhaps with some inaccuracy, but without rounding to two decimals.
- In the second line of output we've used a fancier feature of the `Format` method for strings: this allows us to convert the number according to local currency rules for the computer. This example shows what is displayed if the local currency is South African Rands. Elsewhere in the world the output might show a `$14,898.46`. So your computer knows what region of the world it is set up for, and what the currency rules are for that region.

Notice something else very important here. The name of the variable we pass as an argument — `toInvest` — can be different from the name of the parameter — `p`. (But it doesn't have to be different!)

These short variable names are getting quite tricky, so here are two more versions of our method.

```
private double finalAmtV2(double principalAmount, double nominalRate,
                          int numTimesPerYear, double years)
{
    double a = principalAmount *
               Math.Pow((1 + nominalRate / numTimesPerYear), (numTimesPerYear * years));
    return a;
}

private double finalAmtV3(double amt, double rate, int compounded, double years)
{
    double a = amt * Math.Pow((1 + rate / compounded), (compounded * years));
    return a;
}
```

They all perform exactly the same computation. Use your judgement to write code that can be best understood by other humans! Short variable names are more economical and sometimes make code easier to read: $E = mc^2$ would not be nearly so memorable or elegant if Einstein had used longer variable names! If you do prefer short names, make sure you also have some comments to enlighten the reader about what the variables are used for.

10.2. Scopes and Lifetimes

Any object that is created, (e.g. your main window, turtle Tess, a variable) has a **scope** — that portion of your program in which it can be accessed and used.

When we create a variable inside a method, it only exists inside the method, it belongs to the method, and we cannot use it outside the method. We say it has **local scope**, and we sometimes call it a **local variable**. For example, consider the `finalAmtV3` method above. If we try to use variable `a` outside the method, we'll get an error. The variable `a` is local to `finalAmtV3`.

Additionally, variable `a` only exists while the method is being executed — we call this its **lifetime**. When the execution of the method ends, all local variables are destroyed.

Parameters are also local, and have local scope and short lifetimes too. For example, the variables `amt`, `rate`, `compounded`, and `years` are created when `finalAmtV3` is called, and their lifetime ends when the method completes its execution.

Why is this idea of a lifetime of a variable (or any object) important? Because it is not possible for a method to save some value to a local variable and get it back next time the method is called. Each call of the method creates new local variables, and their lifetimes expire (along with any values they were holding) when the method returns to the caller.

The object that is instantiated from the class that contains our methods also has its own lifetime. Typically, for our applications, our main window object is instantiated (created) when the program starts running. So the main window lifetime is usually starts when the program starts running, and it dies when the window is closed or the application ends.

Any variables defined directly in a class (rather than in a method of the class) are called **class-level variables**. Their lifetime is not the same as that of a local variable in a method.

Our main window object dies when the application is closed.

So we've always defined our turtle variables as class-level variables. And we create the turtle in the constructor (i.e. when the window is born, we create the turtle object). Turtle Tess contains some of its own variables and properties — the brush width, the heading, the position, and so on. When Tess dies, her variables and properties die too. But she will usually only die when the window for the application is closed. Of course we'll also instantiate Alex, and the variables and properties associated with Alex will persist for his lifetime.

So class-level variables — those defined in the class — have a longer lifetime that local variables defined in a method.

Now the nice thing about class-level variables is that they are in scope — they can be seen and used — by any other method in the class.

So if a method needs to remember a value between calls to itself, (e.g. suppose the method wants to count how many times it has been called so far), it must use a class-level variable to keep this counter.

Class-level variables — We've seen this movie before!

Let's take another look at the very first turtle program we saw. The variable `tess` at line 3 is class-level: it is defined in the class. So it can be used in both the constructor method (lines 5–9) and the handler method (lines 11–17).

```
1  public partial class MainWindow : Window
2  {
3      Turtle tess;              // Define a variable to refer to our turtle
4
5      public MainWindow()
6      {
7          InitializeComponent();
8          tess = new Turtle(playground);  // Create a turtle in the playground
9      }
10
11     private void btnDemo1_Click(object sender, RoutedEventArgs e)
12     {
13         tess.Forward(80.0);
14         tess.Right(90.0);
15         tess.Forward(30.0);
16         tess.Right(90.0);
17     }
18 }
```

If we mistakenly deleted line 3 and defined `Turtle tess = new Turtle(playground)` at line 8, it would be a local variable inside the constructor method `MainWindow()`. So

- the variable `tess` would not be visible and we'd get errors on lines 13–16, and
- even if we commented out the lines in error, Tess' lifetime would start at line 8, and would end at line 9 when the constructor method completed execution.

10.3. More value-returning methods

The next example is `area`, which returns the area of a circle with the given radius.

```
1  private double area(double radius)
2  {
3      double b = 3.14159 * radius * radius;
4      return b;
5  }
```

The expression to be returned can be arbitrarily complicated, so we could have written this method body in just one line. On the other hand, **temporary variables** like `b` above often make debugging easier, because we can set breakpoints and inspect them.

```
1  private double area(double radius)
2  {
3      return Math.PI * radius * radius;
4  }
```

Here we've also used the constant `PI` from the `Math` library. It is more accurate than our `3.14159`, and it makes the programmer's intentions clearer.

Sometimes it is useful to have multiple return statements, one in each branch of a conditional. This method takes an integer mark, and returns "Whoops!" if the mark is below 50, and "Good news!" if it is 50 or above:

```
1  private string classifyMark(int percent)
2  {
3      if (percent < 50)
4      {
5          return "Whoops!";
6      }
7      else
8      {
9          return "Good news!";
10     }
11 }
```

Another way to write the above method is to leave out the `else` and just follow the `if` condition by the second `return` statement.

```
1  private string classifyMark(int percent)
2  {
3      if (percent < 50)
4      {
5          return "Whoops!";
6      }
7      return "Good news!";
8  }
```

The moment a `return` statement is executed, anywhere in the method, the flow of execution returns immediately to the call site and the method call is completed. (There are some more advanced features, like `try` and `catch` where this can change — we'll get to them a bit later.)

Code that appears after a `return` statement, or at any other place the flow of execution can never reach is called **unreachable code**. Your C# compiler should warn you about unreachable code, but your program can still run.

In a value-returning method, every possible path of the flow of execution must return a value. If, for example, you left out line 7 in the method above, your program would give a compile error, saying "not all code paths return a value".

It is also possible to use a return statement in the middle of a loop, in which case control immediately returns from the method. Let us assume that we want a method which looks through an array of words. It should return the first two-letter word. If there is not one, it should return the empty string:

```
1  private string find_first_2_letter_word(string[] xs)
2  {
3      foreach (string wd in xs)
4      {
5          if (wd.Length == 2) {
6             return wd;
7          }
8      }
9      return "";
10 }
```

Executing this fragment of code would pop up a message box with the result "is":

```
string[] wds = { "The", "world", "is", "not", "enough", "go", "in" };
MessageBox.Show(find_first_2_letter_word(wds));
```

Single-step through this code and convince yourself that for the data we've provided, the method returns while processing the third element in the array: it does not have to traverse the whole array.

10.4. Program development

At this point, you should be able to look at complete methods and tell what they do. Also, if you have been doing the exercises, you have written some small methods. As you write larger methods, you might start to have more difficulty, especially with runtime and semantic errors.

To deal with increasingly complex programs, we are going to suggest a technique called **incremental development**. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose we want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2). By the Pythagorean theorem, the distance is:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a `distance` method should look like in C#. In other words, what are the inputs (parameters) and what is the output (return value)? And what is the best type for each of these?

In this case, the two points are the inputs, which we can represent using four double parameters. The return value is the distance, also a double.

Already we can write an outline of the method that captures our thinking so far:

```
1  private double distance(double x1, double y1, double x2, double y2)
2  {
3      return 0.0;
4  }
```

Obviously, this version of the method doesn't compute distances correctly; it always returns zero. (It has to return some double, or we'll get an error!) But it is syntactically correct, we've got its signature as we want it, and it will compile and run, which means that we can test it before we make it more complicated. We'll call a method like this (one for which we have not yet written the logic of the innards) a *stub*.

Often stubs are a useful thinking mechanism to help with *abstraction*: we are able to say "Aha, we know we can write a method for distance or compound interest. Let's ignore the inner detail for the moment (abstraction), and focus instead on the bigger picture problem".

To test the new method, we could call it with sample values and confirm that it returns 0:

```
MessageBox.Show(distance(1.0, 2.0, 4.0, 6.0).ToString());
```

We chose these values so that the horizontal distance equals 3 and the vertical distance equals 4; that way, the result is 5 (the hypotenuse of a 3-4-5 triangle). When testing a method, it is useful (perhaps even essential) to know the right answer.

At this point we have confirmed that the method is syntactically correct, and we can start adding lines of code. After each incremental change, we test the method again. If an error occurs at any point, we know where it must be — in the last line we added.

A logical first step in the computation is to find the differences $x_2 - x_1$ and $y_2 - y_1$. We will refer to those values using temporary variables named `dx` and `dy`.

```
private double distance(double x1, double y1, double x2, double y2)
{   double dx = x2 - x1;
    double dy = y2 - y1;
    return 0.0;
}
```

If we call the method with the arguments shown above, when the flow of execution gets to the return statement, `dx` should be 3 and `dy` should be 4. We can check that this is the case by setting a breakpoint and inspecting the variables to confirm that the method is getting the right parameters and performing the first computation correctly. If not, there are only a few lines to check.

Next we compute the sum of squares of `dx` and `dy`:

```
private double distance(double x1, double y1, double x2, double y2)
{   double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx * dx + dy * dy;
    return 0.0;
}
```

Again, we could run the program at this stage and check the value of `dsquared` (which should be 25).

Finally, we'll use the `Sqrt` method to compute and return the result:

```
private double distance(double x1, double y1, double x2, double y2)
{   double dx = x2 - x1;
    double dy = y2 - y1;
    double dsquared = dx * dx + dy * dy;
    double d = Math.Sqrt(dsquared);
    return d;
}
```

If that works correctly, you are done. Otherwise, you might want to inspect the value of `d` before the return statement.

When you start out, you might add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger conceptual chunks. Either way, stepping through your code one line at a time and verifying that each step matches your expectations can save you a lot of debugging time. As you improve your programming skills you should find yourself managing bigger and bigger chunks: this is very similar to the way we learned to read letters, syllables, words, phrases, sentences, paragraphs, etc., or the way we learn to chunk music — from individual notes to chords, bars, phrases, movements, and so on.

The key aspects of the process are:

1. Start with a working skeleton program and make small incremental changes. At any point, if there is an error, you will know exactly where it is.
2. Use temporary variables to refer to intermediate values so that you can easily inspect and check them.

3. Once the program is working, relax, sit back, and play around with your options. (There is interesting research that links "playfulness" to better understanding, better learning, more enjoyment, and a more positive mindset about what you can achieve — so spend some time fiddling around!) You might want to consolidate multiple statements into one bigger compound expression, or rename the variables you've used, or see if you can make the method shorter. A good guideline is to aim for making code as easy as possible for others to read.

A tip about debugging

You must have a clear solution to the problem, and must know what should happen before you can debug a program. Work on *solving* the problem on a piece of paper *before* you concern yourself with writing code.

Writing a program doesn't solve the problem — it simply *automates* the manual steps you would take.

So first make sure you have a pen-and-paper manual solution that works. Programming then is about making those manual steps happen automatically.

10.5. Composition

As you should expect by now, you can call one method from within another. This ability is called **composition**.

As an example, we'll write a method that takes two points, the centre of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the centre point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. Fortunately, we've just written a method, `distance`, that does just that, so now all we have to do is use it:

```
1  double radius = distance(xc, yc, xp, yp);
```

The second step is to find the area of a circle with that radius and return it. Again we will use one of our earlier methods:

```
1  double result = area(radius);
2  return result;
```

Wrapping that up in a method, we get:

```
1  private double area2(double xc, double yc, double xp, double yp)
2  {
3      double radius = distance(xc, yc, xp, yp);
4      double result = area(radius);
5      return result;
6  }
```

We called this method `area2` to distinguish it from the `area` method we wrote earlier.

The temporary variables `radius` and `result` are useful for development, debugging, and single-stepping through the code to inspect what is happening, but once the program is working, we can make it more concise by composing the method calls:

```
private double area3(double xc, double yc, double xp, double yp)
{
    return area(distance(xc, yc, xp, yp));
}
```

Try single stepping through both.

10.6. Boolean methods

Methods can return Boolean values, which is often convenient for hiding complicated tests inside methods. For example:

```
private bool isDivisible(int x, int y)
{
    // Test if x is exactly divisible by y.
    if (x % y == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

It is common to give **Boolean methods** names that sound like yes/no questions. `isDivisible` returns either `false` or `true` to indicate whether the `x` is or is not divisible by `y`.

We can make the method more concise by taking advantage of the fact that the condition of the `if` statement is itself a Boolean expression. We can return it directly, avoiding the `if` statement altogether:

```
private bool isDivisible(int x, int y)
{
    // Test if x is exactly divisible by y.
    return (x % y == 0);
 }
```

Boolean methods are often used in conditional statements:

```
if (isDivisible(x, y)) {  ... // Do something ...  }
else {  ... // Do something else ...  }
```

It might be tempting to write something like:

```
if (isDivisible(x, y) == true) { ... }
```

but the extra comparison is unnecessary.

10.7. Unit testing

It is a common best practice in software development to include automatic **unit testing** of source code. Unit testing provides a way to automatically verify that individual pieces of code, such as methods, are working properly. This makes it easier to change the implementation of a method at a later time and quickly check that it still does what it was intended to do.

Some years back organizations had the view that their valuable asset was the program code and documentation. Organizations will now spend a large portion of their software budgets on crafting (and preserving) their tests.

Unit testing also forces the programmer to think about the different cases that the method needs to handle. Another benefit is that you only have to code up the tests once, rather than having to keep entering the same test data over and over as you develop your code.

Extra code in your program which is there because it makes debugging, development, or testing easier is called **scaffolding**.

A collection of tests for some code is called its **test suite**.

There are a few different ways to do unit testing in C#, but they carry quite a bit of overhead. So for now we're going to ignore what the C# community usually does, and we're going to start with our own library that provides an easier way to get started with writing unit tests.

You'll need to add ThinkLib to your program

As we did earlier with the Turtles, you'll need the ThinkLib module. See the appendix for details.

Let's start with the `classifyMark` method that we wrote earlier in this chapter. First we plan our tests. We'd like to know if the method returns the correct value when its argument is below 50, exactly 50, or above 50. So three tests should do the trick. When planning tests, you'll always want to think carefully about the "edge" cases — here, an argument of exactly 50 is on the edge of where the method behaviour changes. It might be an easy spot for the programmer to make a mistake! So it is a good case to include in our test suite.

We add this code to our program ...

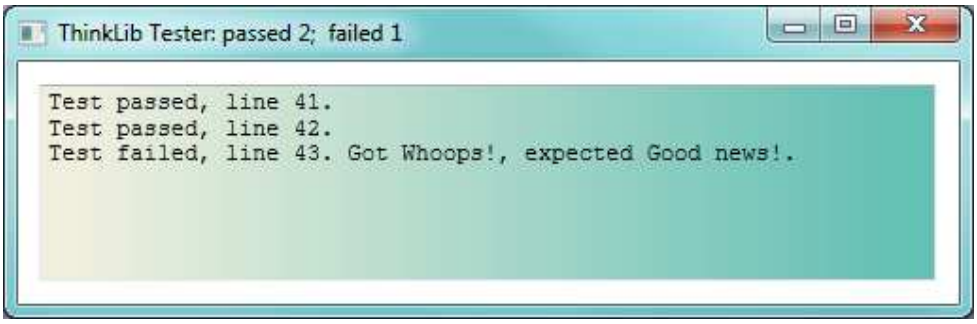
```
using ThinkLib;   // As we did before for the turtles...

private void button1_Click(object sender, RoutedEventArgs e)
{
    Tester.TestEq(classifyMark(25), "Whoops!");
    Tester.TestEq(classifyMark(65), "Good news!");
    Tester.TestEq(classifyMark(50), "Good news!");  // Ha! We thought about the ea
}
```

When the click event occurs, the tests run, and the results pop up in a window:

If we fiddle with the code, let's say we change `classifyMark` like this:

```
private string classifyMark(int percent)
{
    if (percent > 50)
    {
        return "Good news!";
    }
    return "Whoops!";
}
```

Now we re-run our tests:



Aha! The opposite of < was incorrectly coded as >. That's a common mistake! Changing our code to improve its style or readability caused our code to *regress*, or go backwards.

10.8. Glossary

Boolean method

A method that returns a Boolean value. The only possible values of the `bool` type are `false` and `true`.

composition (of methods)

Calling one method from within the body of another, or using the return value of one method as an argument to the call of another.

value-returning method

A method that yields a return value. The other kind of method is a void method, discussed in an earlier chapter.

incremental development

A program development plan intended to simplify debugging by adding and testing only a small amount of code at a time.

return type

A value-returning method returns a value of some specific type. When the method is defined, this type comes before the method name. In the body of the method there must be a return statement that returns some value matching the return type of the method.

return value

The value provided as the result of a value-returning method call.

scaffolding

Code that is used during program development to assist with development and debugging. The unit test code that we added in this chapter are examples of scaffolding.

stub

A method with dummy innards, but the signature is correct, and having the method allows us to write other code or unit tests. A stub is often part of our scaffolding.

temporary variable

A variable used to store an intermediate value in a complex calculation.

test suite

A collection of tests for some code you have written.

unit testing

An automatic mechanism used to validate that individual chunks of code are working properly. Having a test suite is extremely useful when somebody modifies or extends the code: it provides a safety net against going backwards by putting new bugs into previously working code. The term *regression testing* is often used to capture this idea that we don't want to go backwards!

unreachable code

Statements in a program that can never be executed, often because they appear after a `return` statement.

10.9. Exercises

1. Assume the days of the week are numbered 0,1,2,3,4,5,6 from Sunday to Saturday. Write a method which is given the day number, and it returns the day name (a string).
2. We go on a wonderful holiday (perhaps to jail, if we don't like happy exercises) leaving on day number 3 (a Wednesday). We return home after 137 sleeps. Write a general version of the program which takes the starting day number, and the length of our stay, and it will output the name of day of the week we will return on.
3. Write a method which is given an exam mark, and it returns a string — the grade for that mark — according to this scheme:

Mark	Grade
>= 75	First
[70-75)	Upper Second
[60-70)	Second
[50-60)	Third
[45-50)	F1 Supp
[35-45)	F2
< 35	F3

   The square and round brackets denote closed and open intervals. A closed interval includes the number, and open interval excludes it. So 34.99999 gets grade F3, but 35 gets grade F2. Assume

```
double[] xs = {83, 75, 74.9, 70, 69.9, 65, 60, 59.9, 55, 50,
                      49.9, 45, 44.9, 35, 34.9, 2, 0};
```

   Test your method by showing the mark and the grade for all the elements in this array.

4. Write a method `find_hypot` which, given the length of two sides of a right-angled triangle, returns the length of the hypotenuse. (Hint: `Math.Sqrt` will compute the square root.)

5. Write a method `is_rightangled` which, given the length of three sides of a triangle, will determine whether the triangle is right-angled. Assume that the third argument to the method is always the longest side. It will return `true` if the triangle is right-angled, or `false` otherwise.

   Hint: Floating point arithmetic is not always exactly accurate, so it is not safe to test floating point numbers for equality. If a good programmer wants to know whether `x` is equal or close enough to `y`, they would probably code it up as:

   ```
   if  (Math.Abs(x-y) < 0.000001)    // If x is approximately equal to y
       ...
   ```

6. Extend the above program so that the sides can be given to the method in any order.

   All of the exercises below can be done in a single program. The button handler can run all the tests in your test suite. As you work through the exercises, add the new tests to your test suite. (If you open the on-line version of the textbook, you can easily copy and paste the tests and the fragments of code into your C# editor.) After completing each exercise, confirm that all the tests pass.

7. Write a method `sum_to(n)` that returns the sum of all integer numbers up to and including `n`. So `sum_to(10)` would be *`1+2+3...+10`* which would return the value 55.
8. Write a method `area_of_circle(r)` which returns the area of a circle of radius `r`.
9. The four compass points can be abbreviated by characters as 'N', 'E', 'S', and 'W'. Write a method `turn_clockwise` that takes one of these four compass points as its parameter, and returns the next compass point in the clockwise direction. Here are some tests that should pass:

   ```
   Tester.TestEq(turn_clockwise('N'), 'E');
   Tester.TestEq(turn_clockwise('W'), 'N');
   ```

   You might ask *"What if the argument to the method is some other value?"* For all other cases, the method should return the character '?':

   ```
   Tester.TestEq(turn_clockwise('Q'), '?');
   Tester.TestEq(turn_clockwise('X'), '?');
   ```

10. Write a method `day_name` that converts an integer number 0 to 6 into the name of a day. Assume day 0 is "Sunday". Once again, return "Oops" if the arguments to the method are not valid. Here are some tests that should pass:

    ```
    Tester.TestEq(day_name(3), "Wednesday");
    Tester.TestEq(day_name(6), "Saturday");
    Tester.TestEq(day_name(42), "Oops");
    Tester.TestEq(day_name(-1), "Oops");
    ```

11. Write the inverse method `day_num` which is given a day name, and returns its number:

    ```
    Tester.TestEq(day_num("Friday"), 5);
    Tester.TestEq(day_num("Sunday"), 0);
    Tester.TestEq(day_num(day_name(3)), 3);    // Make sure you get this!
    Tester.TestEq(day_name(day_num("Thursday")), "Thursday");
    ```

12. (This is similar to question 2) Write a method that helps answer questions like "Today is Wednesday. I leave on holiday in 19 days time. What day will that be?" So the method must take a

day name and a `delta` argument — the number of days to add — and should return the resulting day name:

```
Tester.TestEq(day_add("Monday", 4),  "Friday");
Tester.TestEq(day_add("Tuesday", 0), "Tuesday");
Tester.TestEq(day_add("Tuesday", 14), "Tuesday");
Tester.TestEq(day_add("Sunday", 100), "Tuesday");
```

Hint: use the first two methods written above to help you write this one.

13. Can your `day_add` method already work with negative deltas? For example, -1 would be yesterday, or -7 would be a week ago:

```
Tester.TestEq(day_add("Sunday", -1), "Saturday");
Tester.TestEq(day_add("Sunday", -7), "Sunday");
Tester.TestEq(day_add("Tuesday", -100), "Sunday");
```

If your method already works, explain why. If it does not work, make it work.

Hint: Play with some cases of using the remainder operator `%` (introduced at the beginning of the previous chapter). Specifically, explore what happens to `x % 7` when x is negative.

14. Write a method `days_in_month` which takes the name of a month, and returns the number of days in the month. Ignore leap years:

```
Tester.TestEq(days_in_month("February"), 28);
Tester.TestEq(days_in_month("December"), 31);
```

If the method is given invalid arguments, it should return -1.

15. Write a method `to_secs` that converts hours, minutes and seconds to a total number of seconds. Here are some tests that should pass:

```
Tester.TestEq(to_secs(2, 30, 10), 9010);
Tester.TestEq(to_secs(2, 0, 0), 7200);
Tester.TestEq(to_secs(0, 2, 0), 120);
Tester.TestEq(to_secs(0, 0, 42), 42);
Tester.TestEq(to_secs(0, -10, 10), -590);
```

16. Extend `to_secs` so that it can cope with real values as inputs. It should always return an integer number of seconds (truncated towards zero):

```
Tester.TestEq(to_secs(2.5, 0, 10.71), 9010);
Tester.TestEq(to_secs(2.433,0,0), 8758);
```

17. Write three methods that are the "inverses" of `to_secs`:

    `hours_in` returns the whole integer number of hours represented by a total number of seconds.

    `minutes_in` returns the whole integer number of left over minutes in a total number of seconds, once the hours have been taken out.

    `seconds_in` returns the left over seconds represented by a total number of seconds.

You may assume that the total number of seconds passed to these methods is an integer. Here are some test cases:

```
Tester.TestEq(hours_in(9010), 2);
Tester.TestEq(minutes_in(9010), 30);
Tester.TestEq(seconds_in(9010), 10);
```

It won't always be obvious what is wanted ...

In the third case above, the requirement seems quite ambiguous and fuzzy. But the test clarifies what we actually need to do.

Unit tests often have this secondary benefit of clarifying the specifications. If you write your own test suites, consider it part of the problem-solving process as you ask questions about what you really expect to happen, and whether you've considered all the possible cases.

Interestingly, some of the literature on Computational Thinking suggests that having a high tolerance for ambiguity is necessary for success in this field!

18. Which of these tests fail? Explain why.

```
Tester.TestEq(3 % 4, 0);
Tester.TestEq(3 % 4, 3);
Tester.TestEq(3 / 4, 0);
Tester.TestEq(3+4  *  2, 14);
Tester.TestEq(4-2+2, 0);
Tester.TestEq("hello, world!".Length, 13);
```

19. Write a `compare` method that returns `1` if `a > b`, `0` if `a == b`, and `-1` if `a < b`

```
Tester.TestEq(compare(5, 4), 1);
Tester.TestEq(compare(7, 7), 0);
Tester.TestEq(compare(2, 3), -1);
Tester.TestEq(compare(42, 1), 1);
```

20. Write a method called `hypotenuse` that returns the length of the hypotenuse of a right triangle given the lengths of the two shorter edges as parameters:

```
Tester.TestEq(hypotenuse(3, 4), 5.0);
Tester.TestEq(hypotenuse(12, 5), 13.0);
Tester.TestEq(hypotenuse(24, 7), 25.0);
Tester.TestEq(hypotenuse(9, 12), 15.0);
```

21. Write a method `slope(x1, y1, x2, y2)` that returns the slope of the line through the points (x1, y1) and (x2, y2). Be sure your implementation of `slope` can pass the following tests:

```
Tester.TestEq(slope(5, 3, 4, 2), 1.0);
Tester.TestEq(slope(1, 2, 3, 2), 0.0);
Tester.TestEq(slope(1, 2, 3, 3), 0.5);
Tester.TestEq(slope(2, 4, 1, 2), 2.0);
```

Then use a call to `slope` in a new method named `intercept(x1, y1, x2, y2)` that returns the y-intercept of the line through the points `(x1, y1)` and `(x2, y2)`

```
Tester.TestEq(intercept(1, 6, 3, 12), 3.0);
Tester.TestEq(intercept(6, 1, 1, 6), 7.0);
Tester.TestEq(intercept(4, 6, 12, 8), 5.0);
```

22. Write a method called `isEven(n)` that takes an integer as an argument and returns `true` if the argument is an even number and `false` if it is odd.

Add your own tests to the test suite.

23. Now write the method `is_odd(n)` that returns `true` when `n` is odd and `false` otherwise. Include unit tests for this method too.

    Finally, modify it so that it uses a call to `is_even` to determine if its argument is an odd integer, and ensure that its test still pass.

24. Write a method `is_factor(f, n)` that passes these tests:

```
Tester.TestEq(is_factor(3, 12), true);
Tester.TestEq(is_factor(5, 12), false);
Tester.TestEq(is_factor(7, 14), true);
Tester.TestEq(is_factor(7, 15), false);
Tester.TestEq(is_factor(1, 15), true);
Tester.TestEq(is_factor(15, 15), true);
Tester.TestEq(is_factor(25, 15), false);
```

    An important role of unit tests is that they can also act as unambiguous "specifications" of what is expected. These test cases answer the question *Do we treat 1 and 15 as factors of 15?*

25. Look up what a *proper factor* is. Write a method `is_proper_factor(f, n)` that passes these tests:

```
Tester.TestEq(is_proper_factor(3, 12), true);
Tester.TestEq(is_proper_factor(5, 12), false);
Tester.TestEq(is_proper_factor(7, 14), true);
Tester.TestEq(is_proper_factor(7, 15), false);
Tester.TestEq(is_proper_factor(1, 15), false);
Tester.TestEq(is_proper_factor(15, 15), false);
Tester.TestEq(is_proper_factor(25, 15), false);
```

26. Write `is_multiple` to satisfy these unit tests:

```
Tester.TestEq(is_multiple(12, 3), true);
Tester.TestEq(is_multiple(12, 4), true);
Tester.TestEq(is_multiple(12, 5), false);
Tester.TestEq(is_multiple(12, 6), true);
Tester.TestEq(is_multiple(12, 7), false);
```

    Can you find a way to use `is_factor` in your definition of `is_multiple`?

27. Write the method `f2c(t)` designed to return the integer value of the nearest degree Celsius for given temperature in Fahrenheit.:

```
    Tester.TestEq(f2c(212), 100);     // Boiling point of water
    Tester.TestEq(f2c(32), 0);        // Freezing point of water
    Tester.TestEq(f2c(-40), -40);     // Wow, what an interesting case!
    Tester.TestEq(f2c(36), 2);
    Tester.TestEq(f2c(37), 3);
    Tester.TestEq(f2c(38), 3);
    Tester.TestEq(f2c(39), 4);

If someone says "Wow, it was cold at the North Pole, -40 degrees" you don't need to
ask whether they measured in Fahrenheit or Celsius.  Both measurement scales are straig
line graphs, and -40 is the value at which the lines intersect each other.
```

28. Now do the opposite: write the method `c2f` which converts Celsius to Fahrenheit:

```
Tester.TestEq(c2f(0), 32);
Tester.TestEq(c2f(100), 212);
Tester.TestEq(c2f(-40), -40);
```

```
Tester.TestEq(c2f(12), 54);
Tester.TestEq(c2f(18), 64);
Tester.TestEq(c2f(-48), -54);
```

29. Since our book title mentions *thinking*, read at least one reference about thinking, and about fun ideas like *fluid intelligence*, a key ingredient in problem solving. See, for example, http://psychology.about.com/od/cognitivepsychology/a/fluid-crystal.htm. Being good at Computer Science requires a good mix of both fluid and crystallized kinds of intelligence.

11. Iteration

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

Repeated execution of a set of statements is called **iteration**. Because iteration is so common, C# provides several language features to make it easier. We've already seen the `while` statement. But in this chapter we've going to look at the `for` and `foreach` statements again — another way to have your program do iteration, useful in slightly different circumstances.

Before we do that, let's review a few ideas...

11.1. Assignment

As we have mentioned previously, as the program runs over time, it is legal to make more than one assignment to the same variable. A new assignment gives an existing variable a new value (and it loses its old value). Of course, the variable must already have been defined before we can assign to it. C# also allows us to give a newly defined variable a value right at the time we define it — we call this an **initializer**. Using initializers is considered good practice.

```
1  int airtime_remaining = 1500;
2  Console.WriteLine(airtime_remaining);
3  airtime_remaining = 700;
4  Console.WriteLine(airtime_remaining);
```

The output of this program on the Console is:

```
1500
700
```

because the first time `airtime_remaining` is printed, its value is 1500, and the second time, its value is 700.

It is especially important to distinguish between an assignment statement and a Boolean expression that tests for equality. Because C# uses the equal token (=) for assignment, it is tempting to interpret a statement like `a = b` as a Boolean test. Unlike mathematics, it is not! Remember that the C# token for the equality operator is `==`.

Note too that an equality test is symmetric, but assignment is not. For example, if `a == 7` then `7 == a`. But in C#, the assignment `a = 7` is legal but `7 = a` is not.

In C#, an assignment statement can make two variables equal, but because further assignments can change either of them, they don't have to stay that way:

```
1  int a = 5;
2  int b = a;      // After executing this line, a and b are now equal
3  a = 3;          // After executing this line, a and b are no longer equal
```

The third line changes the value of `a` but does not change the value of `b`, so they are no longer equal. (In some programming languages, a different symbol is used for assignment, such as `<-` or `:=`, to avoid confusion. Some people also think that *variable* was an unfortunate word to choose, and instead we should have called them *assignables*. C# chooses to follow common terminology and token usage, also found in languages like C, C++, Java, and Python, so we use the tokens `=` for assignment, `==` for equality, and we talk of *variables*.

11.2. Updating variables

When an assignment statement is executed, the right-hand side expression (i.e. the expression that comes after the assignment token) is evaluated first. This produces a value. Then the assignment is made, so that the variable on the left-hand side now holds the new value.

One of the most common forms of assignment is an *update*, where the new value of the variable depends on its old value.

```
1  //Deduct 40 cents from my airtime balance.
2  airtime_remaining = airtime_remaining - 0.40;
3
4  int n = 5;
5  // Change n, based on it's current value.
6  n = (n % 2 == 0) ? n / 2 : 3 * n + 1;
```

When reading an assignment statement, develop the habit of evaluating the right hand side first, because this is how computer languages do it. So line 6 means *first get the current value of n and evaluate the conditional expression to test if it is even. Since it is not even, multiply it by three and then add one. (This results in the value 16 in this case.) Then assign that 16 to the variable n*.

If we try to get the value of a variable that has never been assigned to, (and if the compiler is not clever enough to understand the situation) we'll get its *default initialization value*. This is the value that C# gives the variable if we do not provide an initializing expression. For numbers the initial value is zero, for strings it is the value `null`.

Some programming languages don't provide initializing values automatically, and our variables could start off containing random nonsense. For this reason it is a good habit to make sure that we explicitly give every variable an initial value.

The C# compiler can sometimes spot that we have never initialized a variable, and it will give an error if we try to use it. But it doesn't always catch this. And if we inspect the variable values in memory using the debugger, we'll see that they have been initialized to their default initialization values.

The automatic initialization feature will become more useful when we cover arrays in the subsequent chapters. We can define an array of a thousand integers, and we can safely assume that they will all have their initial value of zero.

11.3. Abbreviated assignment

Adding something to a variable is so common that C# provides an abbreviated syntax for it: `count += 4;` is an abbreviation for `count = count + 4;` We pronounce the operator as *"plus-equals"*.

```
runs_scored += 4;
```

There are similar abbreviations for `-=`, `*=`, `/=` and `%=`.

What is the final value of `n` after executing this sequence of statements?

```
int n = 123;
n -= 10;
n *= 2;
n /= 3;
n %= 7;
```

Incrementing (adding one to) a variable (we say we *bump* the variable) or decrementing (subtracting one from) a variable is also so common that C# provides more special shorthand:

```
n++;      // Adds one to n
k--;      // Subtracts one from k
```

C is an older language that uses this shorthand. C evolved into a new language called C++ (a clever name, suggesting that C++ is "one better than C"). C# is a great name for a language too. It says "We're also related to C, but we're sharper!". (In music, C# (C sharp) is one semitone above C.)

11.4. The `foreach` loop revisited

Recall that the `foreach` loop processes each item in an array. Each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed. Running through all the items in an array is called **traversing** the items.

Let us write a method to sum up all the elements in an array of numbers. *We should do this by hand first*, and try to isolate exactly what steps we take. We'll find we need to keep track of some "running total" of the sum so far, either on a piece of paper, in our head, or in our calculator. Remembering things from one step to the next is precisely why we have variables in a program, and why we have a memory in a calculator: so our program will need a variable to remember the "running total". It should be initialized to the value zero. As we traverse the items in the array, we'll update the running total by adding the current item to the running total.

```
1  private int mySum(int[] xs)
2  {   // Sum all the numbers in the array xs, and return the total.
3      int running_total = 0;
4      foreach (int x in xs) {
5          running_total += x;
6      }
7      return running_total;
8  }
9
10     // Add tests like these to your test suite ...
11     int[] ws = {1, 2, 3, 4};
12     Tester.TestEq(mySum(ws), 10);
13
14     int[] ys = {1, -2, 3};
15     Tester.TestEq(mySum(ys),  2);
16
17     int[] zs = { }; // notice this case. foreach works fine with an empty array.
18     Tester.TestEq(mySum(zs), 0);
```

11.5. The `for` loop

C# comes with extensive documentation for all its built-in methods, and its libraries. Different systems have different ways of accessing this help. In Visual Studio, you can put your mouse cursor above a keyword like `for` and hit the `F1` key to get help. Learn to read and make frequent use of the documentation — it takes a little getting used to. We'll use Microsoft's documentation here instead of writing our own:

for (C# Reference)

Visual Studio 2013 | Other Versions

By using a **for** loop, you can run a statement or a block of statements repeatedly until a specified expression evaluates to **false**. This kind of loop is useful for iterating over arrays and for other applications in which you know in advance how many times you want the loop to iterate.

Example

In the following example, the value of `i` is written to the console and incremented by 1 during each iteration of the loop.

C#

```
class ForLoopTest
{
    static void Main()
    {
        for (int i = 1; i <= 5; i++)
        {
            Console.WriteLine(i);
        }
    }
}
/*
Output:
1
2
3
4
5
*/
```

The **for** statement in the previous example performs the following actions.

1. First, the initial value of variable `i` is established. This step happens only once, regardless of how many times the loop repeats. You can think of this initialization as happening outside the looping process.
2. To evaluate the condition (`i <= 5`), the value of `i` is compared to 5.
    - If `i` is less than or equal to 5, the condition evaluates to **true**, and the following actions occur.
        a. The `Console.WriteLine` statement in the body of the loop displays the value of `i`.
        b. The value of `i` is incremented by 1.
        c. The loop returns to the start of step 2 to evaluate the condition again.
    - If `i` is greater than 5, the condition evaluates to **false**, and you exit the loop.

Note that, if the initial value of `i` is greater than 5, the body of the loop doesn't run even once.

Every **for** statement defines initializer, condition, and iterator sections. These sections usually determine how many times the loop iterates.

11.6. The `while` statement

Let us now write a method to calculate the sum of all the numbers between 1 and N. There are at least two ways to do this: perhaps we know the formula for the answer, and can just plug N into the formula. Or, on the other hand, perhaps we'll just solve it the long way: write a loop and total them all up one at a time. Let's start with a method for doing this that demonstrates the use of the `while` statement:

```
1  private int sumTo(int n)
2  {  // Return the sum of 1+2+3 ... n
3      int ss = 0;
4      int v = 1;
5      while (v <= n)
6      {
7          ss += v;
8          v++;
9      }
10     return ss;
11 }
12
13     // For our test suite
14     Tester.TestEq(sumTo(4), 10);
15     Tester.TestEq(sumTo(1000), 500500);
```

We can almost read the `while` statement as if it were English. It means, while `v` is less than or equal to `n`, continue executing the body of the loop. Within the body, each time, increment `v`. When `v` passes `n`, return our accumulated sum.

More formally, here is precise flow of execution for a `while` statement:

- Evaluate the condition at line 5, yielding a value which is either `false` or `true`.
- If the value is `false`, exit the `while` statement and continue execution at the next statement (line 10 in this case).
- If the value is `true`, execute each of the statements in the body (lines 7 and 8) and then go back to the `while` statement at line 5.

The body of the loop consists of all of the statements within the braces after the `while` keyword.

Notice that if the loop condition is `false` the first time we get loop, the statements in the body of the loop are never executed. (What test case could you write if you didn't want the body of the loop to execute at all?)

Usually the body of the loop will change the value of one or more variables so that eventually the controlling condition becomes false and the loop terminates. Otherwise the loop will repeat forever. This is called an **infinite loop**.

In the case here, we can easily reason that the loop terminates because we know that the value of `n` is finite, and we can see that the value of `v` increments each time through the loop, so eventually it will have to exceed `n`. In other cases, it is not so easy, sometimes even impossible, to prove, that a loop will or will not terminate.

What we notice in the previous example is that the `while` loop is more work for us — the programmers — than the equivalent `for` loop. When using a `while` loop one has to manage the loop variable yourself: define it, give it an initial value, test for completion, and then make sure you change something in the body so that the loop terminates. By comparison, here is a method that uses `for` instead:

```
1
2  private int sumTo(int n)
3  {  // Return the sum of 1+2+3 ... n
4      int ss = 0;
5      for (int v = 1; v <= n; v++)
6      {
7          ss += v;
8      }
9      return ss;
}
```

There is also a simple formula for doing this really fast, without the need for N iterations. Check out http://en.wikipedia.org/wiki/Summation and make sure you can also write a C# method to compute `sumTo` without using any looping at all!

So why have a `while` loop if a `for` loop seems easier? This next example shows a case where we need the extra power that we get from the `while` loop.

11.7. The Collatz 3n + 1 sequence (AKA the *Hailstone* sequence)

Let's look at a simple sequence that has fascinated and foxed mathematicians since 1937. They still cannot answer even quite simple questions about this.

The "computational rule" for creating the sequence is to start from some given `n`, and to generate the next term of the sequence from `n`, either by halving `n`, (whenever `n` is even), or else by multiplying it by three and adding 1. The sequence terminates when `n` reaches 1. Start by doing some sequences by hand or by using your calculator.

This C# method captures that algorithm:

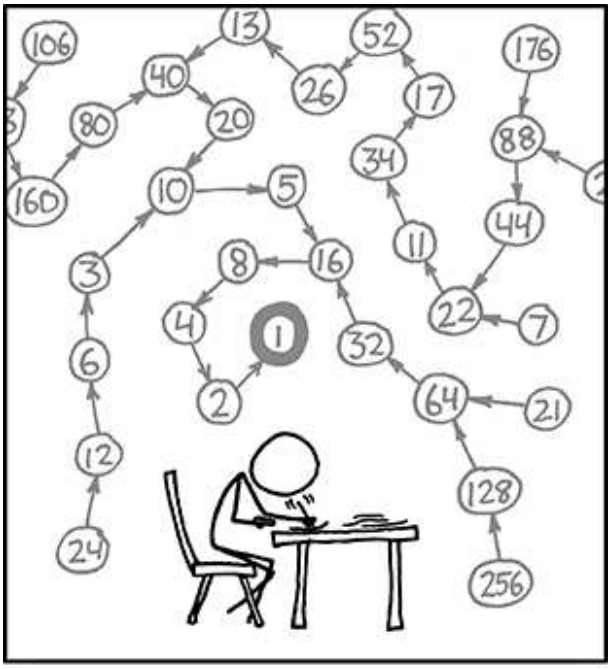
```
1  private void collatz(int n)
2  {  // Print the 3n+1 sequence starting from n,
3     //    terminating when it reaches 1.
4
5      while (n != 1)
6      {
7          Console.Write("{0}, ", n);
8          if (n % 2 == 0)        // n is even
9          {
10             n /= 2;
11         }
12         else                   // n is odd
13         {
14             n = n * 3 + 1;
15         }
16     }
17     Console.WriteLine("{0}. Yes, it got to 1!", n);
18 }
```

Notice first that lines 7 and 17 perform output, this time to the console. If you're working in Visual Studio, use View | Output to see your answers. If we pass the value 24 to the method, your Console output should look like this: (Visual Studio outputs a lot of other stuff into the same Console window, so you may need to hunt for your answers in the middle of other noise!)

```
24, 12, 6, 3, 10, 5, 16, 8, 4, 2, 1. Yes, it got to 1!
```

Of course you might prefer to send your output to your GUI instead of having to search for the Console window, and search for the text within that window. You could replace lines 7 and 17 above with these two lines:

```
1     txtResult.AppendText(n.ToString() + ", ");
2     ...
3     txtResult.AppendText(n.ToString() + ". Yes, it got to 1!\n");
```



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

The condition for continuing with this loop is `n != 1`, so the loop will continue running until it reaches its termination condition, (i.e. `n == 1`).

Each time through the loop, the program outputs the value of `n` and then checks whether it is even or odd. If it is even, the value of `n` is divided by 2 using integer division. If it is odd, the value is replaced by `n * 3 + 1`.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program terminates. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, then the value of `n` will be even each time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

See if you can find a small starting number that needs more than a hundred steps before it terminates.

Particular values aside, the interesting question was first posed by a German mathematician called Lothar Collatz: the *Collatz conjecture* (also known as the *3n + 1 conjecture*), is that this sequence terminates for *all* positive values of `n`. So far, no one has been able to prove it *or* disprove it! (A conjecture is a statement that might be true, but nobody knows for sure.)

Think carefully about what would be needed for a proof or disproof of the conjecture *"All positive integers will eventually converge to 1 using the Collatz rules"*. With fast computers we have been able to test every integer up to very large values, and so far, they have all eventually ended up at 1. But who knows? Perhaps there is some as-yet untested number which does not reduce to 1.

You'll notice that if you don't stop when you reach 1, the sequence gets into its own cyclic loop: 1, 4, 2, 1, 4, 2, 1, 4 ... So one possibility is that there might be other cycles that we just haven't found yet.

Wikipedia has an informative article about the Collatz conjecture. The sequence also goes under other names (Hailstone sequence, Wonderous numbers, etc.), and you'll find out just how many integers have already been

There is a subtle shift of emphasis here ...

Up to this point in the book we've mainly thought about programs and code, and how to do things.

In Computer Science our attention turns quite quickly instead to "the nature of the problem" that we're dealing with. So questions like *"Do we need definite or*

tested by computer, and so far, every one has been found to converge to 1!

indefinite iteration?", or *"What would we need for a proof?"*, or *"How does this sequence climb and fall as we iterate it?"* are focussed on the problem rather than on the code.

As you progress in your Computer Science studies you'll see more emphasis shifting in this direction.

Choosing between `for` and `while`

Use a `for` loop if you know, before you start looping, the maximum number of times that you'll need to execute the body. For example, if you're traversing an array of elements, you know that the maximum number of loop iterations you can possibly need is "all the elements in the array". Or if you need to print the 12 times table, we know right away how many times the loop will need to run.

So any problem like "iterate this weather model for 1000 cycles", or "search this array of words", "find all prime numbers up to 10000" suggests that a `for` loop is better.

By contrast, if you are required to repeat some computation until some condition is met, and you cannot calculate in advance when (or if) this will happen, (as was the case in this 3n + 1 problem), you'll need a `while` loop.

We call the first case **definite iteration** — we know ahead of time some definite bounds for what is needed. The latter case is called **indefinite iteration** — we're not sure how many iterations we'll need — we cannot even establish an upper bound!

11.8. Counting

We often want to count something: perhaps the number of times we had to iterate a Collatz sequence number before we reached one. The pattern for counting has three parts:

- define and initialize a counter to 0 before your iteration or before the region of code that needs the counting operation,
- in the loop body or repeated code, increment the counter each time you want to count another item,
- when the loop terminates, your counter contains the count you want.

Sometimes you'll be asked to only count items that satisfy some condition: how many students passed the test, or how many numbers are odd. So you'll put the increment statement under a conditional, so that it only gets counted sometimes.

We illustrate by changing our Collatz program to also count and show how many elements of the sequence were strictly greater than 10. The new bits of the program are highlighted. We've also got rid of the verbose `if` statement and used the more compact conditional expression in its place at line 11.

```
1  private void collatz(int n)
2  {  // Print the 3n+1 sequence starting from n,
3     //    terminating when it reaches 1.
4
5      int bigNumCount = 0
6
7      while (n != 1)
8      {
9          Console.Write("{0}, ", n);
10         if (n > 10) bigNumCount++;
11         n = (n % 2 == 0) ? n / 2 : n * 3 + 1;
```

```
12      }
13      Console.WriteLine("{0}. Yes, it got to 1!", n);
14      Console.WriteLine("{0} numbers were bigger than 10.", bigNumCount);
15  }
```

11.9. Tracing a program

To write effective computer programs, and to build a good conceptual model of program execution, a programmer needs to develop the ability to **trace** the execution of a computer program. Tracing involves becoming the computer and following the flow of execution through a sample program run, recording the state of all variables and any output the program generates after each instruction is executed.

To understand this process, let's trace the call to `collatz(3)` from the previous section. At the start of the trace, we have a variable, `n` (the parameter), with an initial value of 3. Since 3 is not equal to 1, the `while` loop body is executed. 3 is printed. At line 10, we see that `n` is not bigger than `n`, so nothing happens. Now the conditional expression at line 11 is executed: first its condition `(3 % 2 == 0)` is evaluated. Since it evaluates to `false`, the value of `3 * 3 + 1` is evaluated and assigned to `n`.

To keep track of all this as you hand trace a program, make a column heading on a piece of paper for each variable, and another column for output. On each line, record the values of the variables at the end of each iteration of the loop: Our trace so far would look something like this:

```
n       bigNumCount        output so far
--      -----------        -------------
3                 0        3,
10
```

Since `10 != 1` evaluates to `true`, the loop body is again executed, and 10 is printed. `(10 % 2 == 0)` is true, so the first part of the conditional expression is evaluated and `n` becomes 5. By the end of the trace we have:

```
n       bigNumCount       output so far
--      -----------       -------------
3                 0       3,
10                0       3, 10,
5                 0       3, 10, 5,
16                1       3, 10, 5, 16,
8                 1       3, 10, 5, 16, 8,
4                 1       3, 10, 5, 16, 8, 4,
2                 1       3, 10, 5, 16, 8, 4, 2,
1                 1       3, 10, 5, 16, 8, 4, 2, 1. Yes, it got to 1!
                          1 numbers were bigger than 10.
```

Tracing can be a bit tedious and error prone (that's why we get computers to do this stuff in the first place!), but it is an essential skill for a programmer to have. From this trace we can learn a lot about the way our code works. We can observe that as soon as `n` becomes a power of 2, for example, the program will require $\log_2(n)$ executions of the loop body to complete. We can also see that the final 1 will not be printed as output within the body of the loop, which is why we put the special statement at line 13.

Tracing a program is, of course, related to single-stepping through your code and being able to inspect the variables. Using the computer to **single-step** is less error prone and more convenient. Also, as your programs get more complex, they might execute many millions of steps before they get to the code that you're really interested in, so manual tracing becomes impossible. Being able to set a **breakpoint** where

you need one is far more powerful. So we strongly encourage you to invest time in learning using to use your programming environment (Visual Studio, in these notes) to full effect.

11.10. Counting digits

The following method counts the number of decimal digits in a positive integer:

```
1  private int num_digits(int n)
2  {
3      int count = 0;
4      while (n != 0)
5      {
6          count++;
7          n /= 10;
8      }
9      return count;
10 }
```

Calling `num_digits(710)` will return the value `3`. Trace the execution of this method call (perhaps using the single step method in Visual Studio, or on some paper) to convince yourself that it works.

If we wanted to only count digits that are either 0 or 5, adding an `if` statement before incrementing the counter will do the trick:

```
1  private int num_zero_and_five_digits(int n)
2  {
3      int count = 0;
4      while (n > 0)
5      {
6          int digit = n % 10;
7          if ((digit == 0) || (digit == 5))
8          {
9              count++;
10         }
11         n /= 10;
12     }
13     return count;
14 }
```

Notice a clever technique in line 6 here. By finding the remainder after division by 10 we have isolated the last digit of the current number.

Confirm that `Tester.TestEq(num_zero_and_five_digits(1055030250), 7);` passes. Notice, however, that `Tester.TestEq(num_digits(0), 1);` fails. Explain why. Do you think this is a bug in the code, or a bug in the specifications, or our expectations, or the tests?

11.11. Tables

One of the things loops are good for is generating tables. Before computers were readily available, people had to calculate logarithms, sines and cosines, etc. by hand. To make that easier, mathematics books often contained long tables to let the reader look up the answer rather than calculate it from scratch. Creating the tables was slow and boring, and they tended to have occasional errors.

When computers appeared on the scene, one of the initial reactions was, *"This is great! We can use the computers to generate the tables, so there will be no errors."* That turned out to be true (mostly) but short sighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete, and we found more interesting uses for computers — YouTube, cellphones, games, etc.

For some operations, computers still use tables of values to get an approximate answer, and then they perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the tables that one generation of the Intel Pentium processor chip used to perform floating-point division.

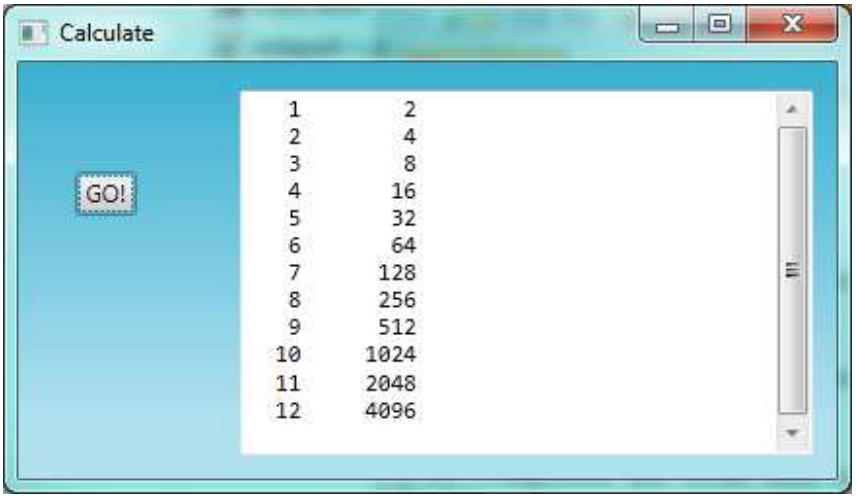
Although a log or power table is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and 2 raised to the power of that value in the right column:

```
1  private void button1_Click(object sender, EventArgs e)
2  {
3      for (int i=1; i <= 12; i++)
4      {
5          string txt = string.Format("{0,4} {1,8}\n", i, Math.Pow(2, i));
6          txtResult.AppendText(txt);
7      }
8  }
```

Line 3 uses a `for` loop to run the body of the loop 12 times, while varying `i` between 1 and 12 (both inclusive). Notice that we've now used the abbreviated assignment `i++` — this is the typical way you're going to see the loop written in other code. Line 5 does some magic with string formatting. We'll be covering more about this in the next chapter, but here is a quick preview: The string.Format method in this case has 3 arguments: the first is a template string with "place-holders" `{0}` and `{1}`. The value of `i` is converted to a string and takes the place of the first place-holder. Then `Math.Pow(2, i)` computes 3 to the power of i, and it too gets converted to a string and takes the place of the second place-holder. The resulting string is assigned to `txt`, and we then find some way to display or output the text. (Of course, if you prefer working with the console you could have said `Console.Write(txt)` instead.)

In the example here, each place-holder has an extra bit of information about how wide the field should be to hold the number. This lets us make the columns in table line up nicely.

We'll see output like this:



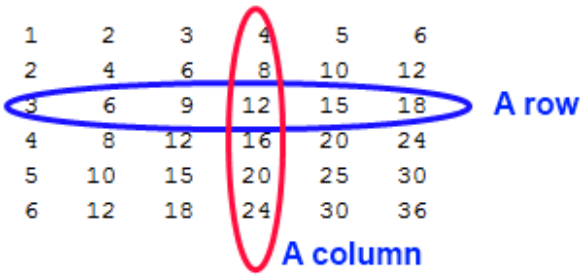
Proportional vs Monospace fonts

If you try this, your output may initially look different. To lay out data in fixed columns, we need to make sure that every character that is output takes exactly the same number of pixels, or width. This kind of font is called a **monospace** font (or *fixed-width* font). Normal fonts, like this text you are reading, are called **proportional** fonts — thin characters like 'i' and 'l' take less width that wide characters like 'w' and 'b'.

So in the example image above, we've set a property for `txtResult` to use the font *Consolas*. Other popular monospace fonts are the *Courier* family. See which you prefer. The Console output window in C# will always automatically use a monospace font, and program source code is usually shown in a monospace font.

11.12. Two-dimensional tables

A two-dimensional table is a table where you can look up the value at the intersection of a row and a column. (A row in a table is laid out horizontally, on a single line. The columns go vertically down the page.) A multiplication table is a good example. Let's say you want to create a multiplication table for the values from 1 to 6. (So it has 6 rows, and 6 columns.) Like this:



A good way to start is to write a loop that generates one of the rows, say all the multiples of 2, all on one line:

```
1  for (int col=1; col <= 6; col++)
2  {
3      txtResult.AppendText(string.Format("{0,5}", 2 * col));
4  }
5  txtResult.AppendText("\n");
```

As the loop executes, the value of `col` changes from 1 to 6. When `col` gets larger than 6, the loop terminates, and we end off the line. Each value of `2 * col` uses a field width of 5 spaces.

Line 5 finishes the current line, and starts a new line. Once again, if you prefer working with the Console, you could have said `Console.Write(...)` on line 3, and `Console.WriteLine()` on line 5.

The output of the program is now the second row of the table we want to create:

```
2    4    6    8   10   12
```

So far, so good. The next step is to **encapsulate** and **generalize**.

11.13. Encapsulation and generalization

Encapsulation is the process of wrapping a piece of code in a method, allowing you to take advantage of all the things methods are good for. You have already seen some examples of encapsulation, including `isDivisible` in a previous chapter.

Generalization means taking something specific, such as generating the multiples of 2, and making it more general, such as generating the multiples of any integer.

This method encapsulates the previous loop and generalizes it to create multiples of `n`:

```
1  private void generateOneRow(int n)
2  {
3      for (int col=1; col <= 6; col++)
4      {
5          txtResult.AppendText(string.Format("{0,5}", n * col));
6      }
7      txtResult.AppendText("\n");
8  }
```

To encapsulate, all we had to do was add the signature, which defines the name of the method with it's parameter. To generalize, all we had to do was replace the value 2 with the parameter `n`.

If we call this method with the argument 2, we get the same output as before. With the argument 3, the output is:

```
3    6    9   12   15   18
```

With the argument 4, the output is:

```
4    8   12   16   20   24
```

By now you can probably guess how to create our multiplication table — by calling `generateOneRow` repeatedly with different arguments. In fact, we can use another loop:

```
1  private void generateTable()
2  {
3      for (int row = 1; row <= 6; row++)
4      {
5          generateOneRow(row);
6      }
7  }
```

By calling this method we get the table shown at the beginning of this section. We called our loop variables `row` and `col`, but of course any legal identifier would have worked. But when you work with tables you'll find programmers often choose these names (or if they're a bit lazier, they might use `r` and `c`).

Exercise: can we generalize further?

The method `generateTable` can only generate a 6x6 table. What if we wanted a 10x10 table, or 12x12, or 50x50?

Add a new parameter to `generateTable` that controls how many rows to generate. When that works you'll find each row still only has 6 columns. So generalize `generateOneRow` by adding another parameter to control how many columns to generate.

11.14. Local variables, revisited

Do the following experiment: change the variable called `row` and call it `i` instead. (Visual Studio understands when you rename a variable, and will offer to rename all other places where it is used. So it can help you make small cosmetic changes like this very easily.) Make sure your program still works as expected.

Now do the same for the `col` variable: change it to `i`. Does it still work?

You might be wondering how we can use the same variable, `i`, in both `generateTable` and `generateOneRow`. Doesn't it cause problems when one of the methods changes the value of the variable `i`?

The answer is no, because the `i` in `generateTable` and the `i` in `generateTable` are *not* the same variable.

In C#, a number of statements grouped together (usually by curly braces) is called a **block**, or equivalently, a **compound statement**. Each block in a program can have its own local variable definitions, and these don't clash with variables defined in other blocks.

So variables created inside a block (or a method) are local to the block (or method); you can't access a local variable from outside its block where it is defined. That means you are free to have multiple variables with the same name as long as they are not in the same block. It also means that updating the variable `i` in `generateOneRow` has absolutely no effect on the variable `i` in `generateTable` — they are different variables, even though they have the same name! (You probably also know quite a few friends called John. Of course they're different!)

In C# the `for` and `foreach` statements also let you define your loop control variable, and that is a local variable that only exists for the loop. So you'll often find C# code that does something like this:

```
1  for (int i = 1; i <= 6; i++)
2  {
3      int p = 123 * i;
4      // ... do stuff
5  }
6
7  for (int i = 1; i <= 15; i++)   // Use i again, it is a different variable.
8  {
9      double p = 3.1415 * i;     // Use p again, it is in a different block.
10     // ... do more stuff
11 }
```

The variables `i` in lines 1 and 7 are different variables. Similarly, the variable `p` at line 3 is not the same variable as the `p` at line 9.

It is common and perfectly legal to have different local variables with the same name. In particular, names like `i` and `j` are used frequently as loop variables. If you avoid using them in one method just

because you used them somewhere else, you will probably make the program harder to read.

11.15. The break statement

The **break** statement is used to immediately leave the body of its loop. The next statement to be executed is the first one after the body:

```
1  int[] nums = {12, 16, 17, 24, 29, 30};
2  foreach (int n in nums)
3  {
4      if (n % 2 == 1)
5      {
6          break;   // immediately exit the loop.
7      }
8      txtResult.AppendText(string.Format("{0} ", n));
9  }
10 txtResult.AppendText("done\n");
```

This will put the following text into `txtResult`:

```
12 16 done
```

Notice that on the third iteration of the loop, `n` is assigned the value 17. So line 6 executes, and immediately the flow of control jumps to line 10: the 17 is not shown in the output. How would this be different if we moved the test at line 4 below the statement on line 8?

11.16. The continue statement

This is a control flow statement that causes the program to immediately skip the processing of the rest of the body of the loop, *for the current iteration*. But the loop still carries on running for its remaining iterations:

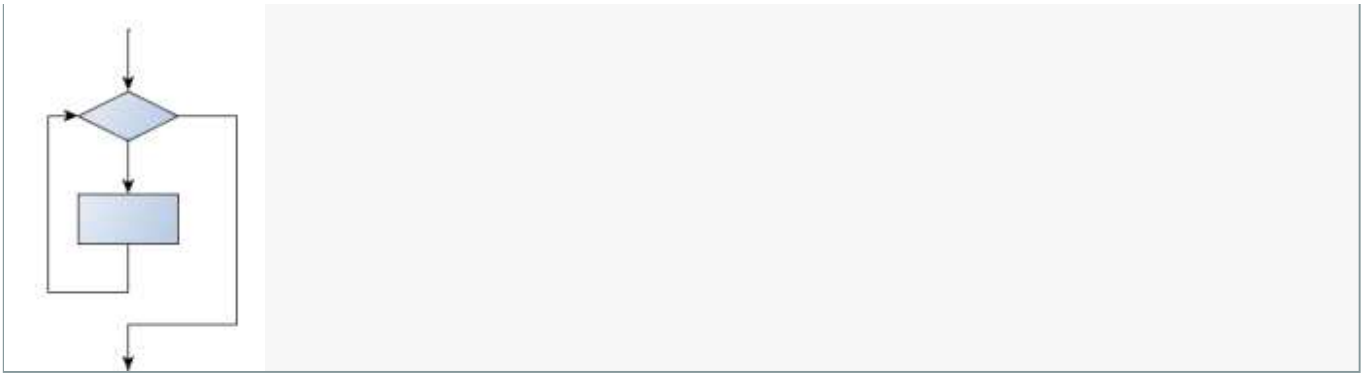
```
1
2  int[] nums = {12, 16, 17, 24, 29, 30};
3  foreach (int n in nums)
4  {
5      if (n % 2 == 1)
6      {
7          continue;   // skip the rest of the loop for odd numbers.
8      }
9      txtResult.AppendText(string.Format("{0} ",n));
10 }
   txtResult.AppendText("done\n");
```

This generates the output

```
12 16 24 30 done
```

The pre-test loop — standard loop behaviour

`for`, `foreach` and `while` loops do their tests at the start, before executing any part of the body. They're called **pre-test** loops, because the test happens before (*pre*) the body. `break`, `return` and `continue` are our tools for adapting this standard behaviour.



11.17. More generalization

As another example of generalization, imagine you wanted a program that would print a multiplication table of any size, not just the six-by-six table. You could add a parameter to `generateMultiplicationTable`:

```
1  private void generateMultiplicationTable(int sz)
2  {
3      for (int i = 1; i <= sz; i++)
4      {
5          generateMultiples(i);
6      }
7  }
```

We replaced the value 6 with the expression `sz`. If we call `generateMultiplicationTable` with the argument 3, it displays:

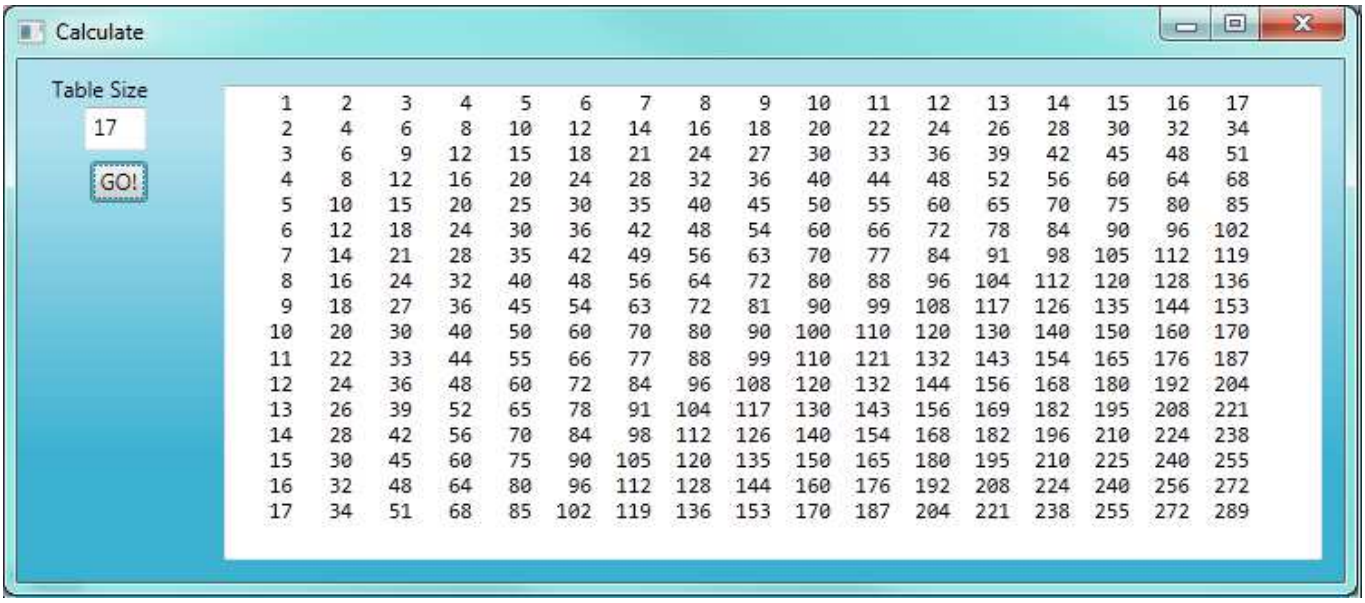
```
1    2    3    4    5    6
2    4    6    8   10   12
3    6    9   12   15   18
```

This is fine, except that we probably want the table to be square — with the same number of rows and columns. To do that, we add another parameter to `generateOneRow` to specify how many columns the table should have.

We also call this parameter `sz`, demonstrating that different methods can have parameters with the same name (just like local variables). Here's the modified section of code.

```
1  private void generateMultiples(int n, int sz)
2  {
3      for (int i=1; i <= sz; i++)
4      {
5          txtResult.AppendText(string.Format("{0,5}",i*n));
6      }
7      txtResult.AppendText("\n");
8  }
9
10 private void generateMultiplicationTable(int sz)
11 {
12     for (int i = 1; i <= sz; i++)
13     {
14         generateMultiples(i, sz);
15     }
16 }
```

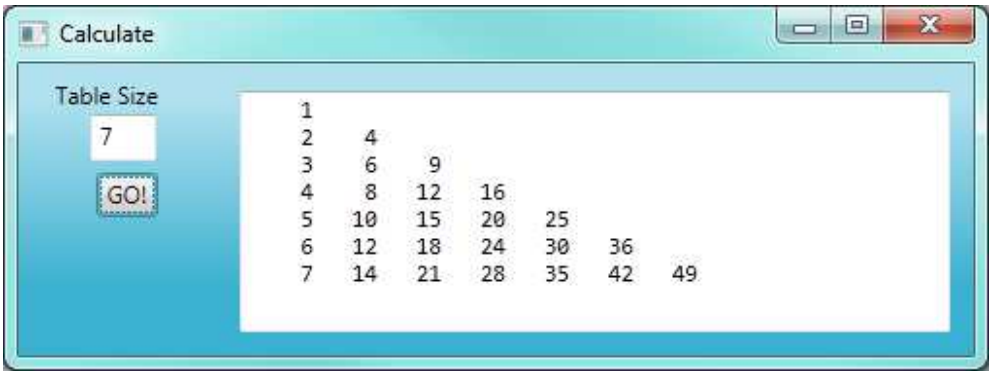
```
17
18  private void button1_Click(object sender, RoutedEventArgs e)
19  {
20      int tsz = int.Parse(textBox1.Text);
21      generateMultiplicationTable(tsz);
22  }
```



We notice that, because `a*b == b*a`, the non-diagonal entries in the table appear twice. (Our mathematicians would call this a symmetric matrix.) We can highlight this important insight by only generating half the table. To do that, we'd only have to change one line of code — line 14:

```
1  generateMultiples(i, i);
```

and now, for a 7 times table, we'll get:



11.18. Nested loops

C# has a very general approach to composibility: if it is legal to write a statement at a given point, then it is legal to write any kind of statement, or even a whole block of statements. What this means is that it is legal to put an `if` statement inside a `for` loop, or a `for` loop inside another `for` loop. When we put a loop inside another loop we say the loops are *nested*. Nested loops are often required to process two-dimensional data — like our table that we've just created, or some image processing that needs to look at all the pixels in an image.

We generated our multiplication table using two separate methods: that mental decomposition helped keep things simple at each step. But separate methods also introduce some overhead - we had to pass arguments between them, etc.

Here we generate the same triangular multiplication table again, but this time we only use a single method with a nested loop.

```
1  private void generateMultiplicationTable(int sz)
2  {
3      for (int r = 1; r <= sz; r++)
4      {
5          for (int c = 1; c <= r; c++)
6          {
7              txtResult.AppendText(string.Format("{0,5}", r * c));
8          }
9          txtResult.AppendText("\n");
10     }
11 }
```

How many times is body of the inner loop (at line 7) executed for a given `sz`? (Or, asked another way, how many numbers were output?) It turns out to be a very popular formula for Computer Scientists: 1 on the first line, + 2 + 3 + 4 ... sz. So exactly the number we computed earlier using our `sumTo` method. And if you followed the link there to Wikipedia, you'll know that the sum of this sequence can be instantly calculated as `sz*(sz+1)/2`.

11.19. Methods

A few times now, we have mentioned all the things methods are good for. By now, you might be wondering what exactly those things are. Here are some of them:

1. Capturing your mental chunking. Breaking your complex tasks into sub-tasks, and giving the sub-tasks a meaningful name is a powerful mental technique. Looking at how we generated the multiplication table, our initial mental chunk was to generate one row of the table. This proved a useful way to break up the problem.
2. Dividing a long program into methods allows you to separate parts of the program, debug them in isolation, and then compose them into a whole.
3. Methods facilitate the use of iteration.
4. Well-designed methods are often useful for many programs. Once you write and debug one, you can reuse it.

11.20. Newton's algorithm for finding square roots

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, before we had calculators or computers, people needed to calculate square roots manually. Sir Isaac Newton, (according to Wikipedia, considered by many to be the greatest and most influential scientist who ever lived) used a particularly good algorithm (there is some evidence that this algorithm was known many years before). Suppose that you want to know the square root of `n`. If you start with almost any approximation, you can compute a better approximation (closer to the actual answer) with the following formula:

```
better = (approx + n/approx)/2
```

Repeat this calculation a few times using your calculator. Can you see why each iteration brings your estimate a little closer? One of the amazing properties of this particular algorithm is how quickly it converges to an accurate answer — a great advantage for doing it manually.

By using a loop and repeating this formula until the better approximation gets close enough to the previous one, we can write a method for computing the square root. (In fact, this is how your calculator finds square roots — it may have a slightly different formula and method, but it is also based on repeatedly improving its guesses.)

This is a good example of an *indefinite* iteration problem: we cannot predict in advance how many times we'll want to improve our guess — we just want to keep getting closer and closer. Our stopping condition for the loop will be when our old guess and our improved guess are "close enough" to each other.

Ideally, we'd like the old and new guess to be exactly equal to each other when we stop. But exact equality is a tricky notion in computer arithmetic when real numbers are involved. Because real numbers are not represented absolutely accurately (after all, a number like pi or the square root of two has an infinite number of decimal places because it is irrational), we need to formulate the stopping test for the loop by asking "is *a* close enough to *b*"? This test can be coded like this:

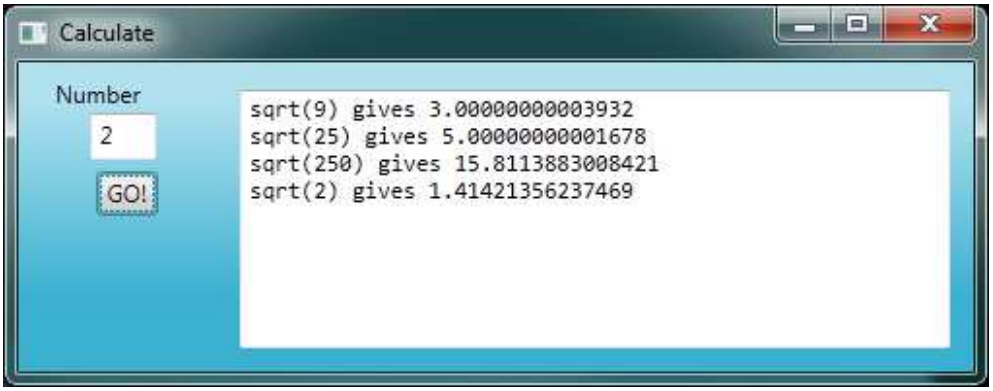
```
if (Math.Abs(a-b) < 0.001)  // Make this smaller for better accuracy
{
    break;
}
```

Notice that we take the absolute value of the difference between `a` and `b`! Make sure you understand why this is necessary.

Here is the code now for Newton's square root finder:

```
private double sqrt(double n)
{
    double approx = n/2.0;     // Start with some or other guess at the answer
    while (true)
    {
        double better = (approx + n / approx) / 2.0;
        if (Math.Abs(approx - better) < 0.001)
        {
            return better;
        }
        approx = better;
    }
}

private void button1_Click(object sender, EventArgs e)
{
    double x = double.Parse(textBox1.Text);
    double y = sqrt(x);
    string output = string.Format("sqrt({0}) gives {1}\n", x, y);
    txtResult.AppendText(output);
}
```

The output is:

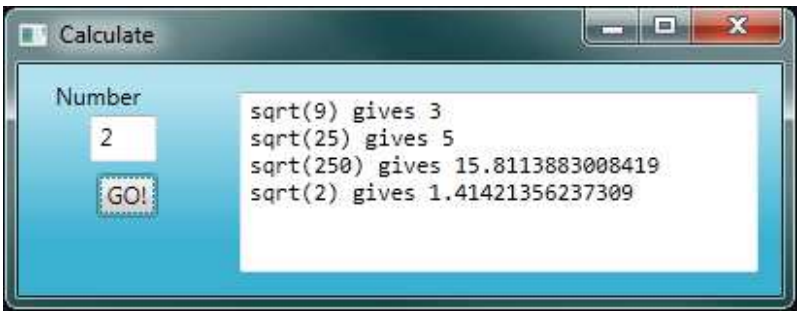


There are a number of interesting things about this code.

- The `while` loop at line 4 has a strange stopping condition — always true! This means it never stops, unless we use a `break` or a `return` to leave the loop. This is what we've done at line 9. Returning from the method, even if we're in the middle of some iteration, happens immediately. The loop does no more iterations.
- The `while (true) { ... }` form of the loop is widely used. This is a language *idiom* — a convention that most programmers will recognize and understand immediately.
- For handling the GUI, we used a text box to allow the user to enter a number. The code at line 17 shows how we extract the text from the text box, and convert it into a double number before calling our new method.

See if you can improve the approximations by changing the stopping condition. Also, step through the algorithm (perhaps by hand, using your calculator) to see how many iterations are needed before it achieves this level of accuracy for `sqrt(25)`.

Here we've made some changes to the loop termination condition: we only stop when the two values are less than 0.00000001 from each other. These results are more accurate now:



11.21. Algorithms

Newton's way of getting the answer is an example of an **algorithm**: it is a mechanical process for solving a category of problems (in this case, computing square roots).

Some kinds of knowledge are not algorithmic. For example, learning dates from history or your multiplication tables involves memorization of specific solutions.

But the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. Or if you are an avid Sudoku puzzle solver, you might have some specific set of steps that you always follow.

One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules. And they're designed to solve a general class or category of problems, not just a single problem.

Understanding that hard problems can be solved by step-by-step algorithmic processes (and having technology to execute these algorithms for us) is one of the major breakthroughs that has had enormous benefits. So while the execution of the algorithm may be boring and may require no intelligence, algorithmic or computational thinking — i.e. using algorithms and automation as the basis for approaching problems — is rapidly transforming our society. Some claim that this shift towards algorithmic thinking and processes is going to have even more impact on our society than the invention of the printing press. And the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming.

Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of a step-by-step mechanical algorithm.

11.22. Glossary

algorithm

A step-by-step process for solving a category of problems.

block of statements

A block is a group of statements, usually enclosed in braces.

body

The statements inside a loop. Usually a loop body is a block.

breakpoint

A place in your program code where program execution will pause (or break), allowing you to inspect the state of the program's variables, or single-step through individual statements, executing them one at a time.

bump

Programmer slang that means increment.

compound statement

see block of statements.

continue statement

A statement that causes the remainder of the current iteration of a loop to be skipped. The flow of execution goes back to the top of the loop, evaluates the condition, and if this is true the next iteration of the loop will begin.

counter

A variable used to count something, usually initialized to zero and incremented in the body of a loop.

decrement

Decrease by 1.

definite iteration

A loop where we have an upper bound on the number of times the body will be executed. Definite iteration is usually best coded using a `for` loop.

encapsulate

To divide a large complex program into components (like methods) and isolate the components from each other (by using local variables, for example).

generalize

To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

increment

Used both as a noun and as a verb. The verb means to add to something. The noun is used like this: *"Your study fees increment this year will be 8%"*.

indefinite iteration

A loop where we just need to keep going until some condition is met. A `while` statement is good to use in this situation.

infinite loop

A loop in which the terminating condition is never satisfied.

initializer

An expression that gives an initial value to a variable where the variable is first defined.

initialization (of a variable)

To initialize a variable is to give it an initial value. In C#, if you don't do this yourself, your variables will be given default initial values.

iteration

Repeated execution of a set of programming statements.

loop

The construct that allows allows us to repeatedly execute a statement or a group of statements (a block or a compound statement) until a terminating condition is satisfied.

loop variable

A variable used as part of the terminating condition of a loop.

nested loop

A loop inside the body of another loop.

pre-test loop

A loop that tests before deciding whether the execute its body. `for` and `while` are both pre-test loops.

single-step

A mode of execution where you are able to execute your program one step at a time, and inspect the consequences of that step. Useful for debugging and building your internal mental model of what is going on.

trace

To follow the flow of execution of a program by hand, recording the change of state of the variables and any output produced.

11.23. Exercises

This chapter showed us how to sum an array of items, and how to count items. The counting example also had an `if` statement that let us only count some selected items. In the previous chapter we also showed a method `find_first_2_letter_word` that allowed us an "early exit" from inside a loop by using `return` when some condition occurred. We now also have `break` to exit a loop (but not the enclosing method, and `continue` to abandon the current iteration of the loop without ending the loop.

Composition of array traversal, summing, counting, testing conditions and early exit is a rich collection of building blocks that can be combined in powerful ways to create many methods that are all slightly different.

The first six questions are typical methods you should be able to write using only these building blocks.

1. Write a method to count how many odd numbers are in an array.
2. Sum up all the even numbers in an array.
3. Sum up all the negative numbers in an array.
4. Count how many words in an array have length 5. (Use help to find out how to determine the length of a string.)
5. Sum all the elements in an array up to but not including the first even number. (Write your unit tests. What about the case when there is no even number?)
6. Count how many words occur in an array up to and including the first occurrence of the word "sam". (Write your unit tests for this case too. Do something sensible if "sam" does not occur.)
7. Add a print statement to Newton's `sqrt` method to show `better` each time it is calculated. Call your modified method with 25 as an argument and record the results.
8. Trace the execution of the last version of `generateTable` and make yourself more comfortable with single stepping, and debugging.
9. Write a method `print_triangular_numbers(int n)` that prints out the first n+1 triangular numbers. A call to `print_triangular_numbers(5)` would produce the following output:

```
0         0
1         1
2         3
3         6
4        10
5        15
```

   (*Hint*: use a web search to find out what a triangular number is.)

10. a. What happens if we call our Collatz sequence generator with a negative integer?
    b. What happens if we call it with zero?
    c. Change the method so that it outputs an error message in either of these cases, and doesn't get into an infinite loop.
11. Write a method, `isPrime`, which takes a single integer argument and returns `true` when the argument is a *prime number* and `false` otherwise. Add tests for cases like this:

```
// 1 is not regarded as prime. It does not have two distinct factors. Ask Google.
Tester.TestEq(isPrime(1), false);
Tester.TestEq(isPrime(2), true);
Tester.TestEq(isPrime(11), true);
Tester.TestEq(isPrime(25), false);
Tester.TestEq(isPrime(35), false);
Tester.TestEq(isPrime(101), true);
Tester.TestEq(isPrime(19951123), true);
Tester.TestEq(isPrime(19961107), true);
Tester.TestEq(isPrime(19951121), false);
Tester.TestEq(isPrime(19961007), false);
```

The last cases could represent your birth date. Were you born on a prime day? In a class of 100 students, how many do you think would have prime birth dates?

12. A drunk pirate makes a turn, and then takes some steps forward, and repeats this. A social science student records *pairs* of data: the angle of each turn, and the number of steps taken after the turn. Her experimental data is turned into C# as

```
int[] turns = {160, -43, 270, -43};
int[] steps = {20, 10, 8, 12 };
```

Use a turtle to draw the path taken by our drunk friend.

13. Many interesting shapes can be drawn by the turtle by giving a pair of arrays of like we did above, where the first array holds the angles to turn, and the second array holds the distances to move forward. Set up a void method that takes a turtle and two arrays, and make the turtle draw the shape defined by the arrays. Then call it so that the turtle draws a house with a cross through the centre, as show here. This should be done without going over any of the lines / edges more than once, and without lifting your pen.

14. Not all shapes like the one above can be drawn without lifting your pen, or going over an edge more than once. Which of these can be drawn?

Now read Wikipedia's article(http://en.wikipedia.org/wiki/Eulerian_path) about Eulerian paths. Learn how to tell immediately by inspection whether it is possible to find a solution or not. If the path is possible, you'll also know where to put your pen to start drawing, and where you should end up!

15. What will `num_digits(0)` return? Modify it to return `1` for this case. Does a call to `num_digits(-12345)` work? Trace through the execution and see what happens if you start with a

negative number. Modify `num_digits` so that it works correctly with any integer value. Add these tests:

```
Tester.TestEq(num_digits(0), 1);
Tester.TestEq(num_digits(-12345), 5);
```

16. Without making use of strings, write a method `num_even_digits(n)` that counts the number of even digits in `n`. Here are some tests that should pass:

```
Tester.TestEq(num_even_digits(123456), 3);
Tester.TestEq(num_even_digits(2468), 4);
Tester.TestEq(num_even_digits(1357), 0);
// Normally we never put leading zeros on numbers, but our number
// system has a special case that probably needs special case handling in the code.
Tester.TestEq(num_even_digits(0), 1);
Tester.TestEq(num_even_digits(0002468), 4)
Tester.TestEq(num_even_digits(-12345), 2);
Tester.TestEq(num_even_digits(-2468), 4);
```

17. Write a method `sum_of_squares(xs)` that computes the sum of the squares of the numbers in the array `xs`. For example, `sum_of_squares(new double[] {2, 3, 4})` should return 4+9+16 which is 29:

```
Tester.TestEq(sum_of_squares(new double[] {2, 3, 4}), 29);
Tester.TestEq(sum_of_squares(new double[] {}), 0);
Tester.TestEq(sum_of_squares(new double[] {2, -3, 4}), 29);
```

12. Strings

So far we have seen built-in types like `int`, `double`, `char`, `bool`, `string` and we've also had some brief exposure to arrays.

Strings and arrays are different from the others because they are made up of smaller pieces. In the case of strings, they're made up of *characters*.

Depending on what we are doing, we may want to work with the string as a whole, or we may want to access its characters. This ambiguity is useful.

12.1. Working with strings as single things

We previously saw that each button or turtle instance (object) has its own properties, and some methods that can be applied to the instance. For example, we could set the button's colour or size, and we wrote `tess.Right(90)` to get Tess to turn.

Just like a turtle, or a button, a string is also an object. So each string instance has its own properties and methods. For example:

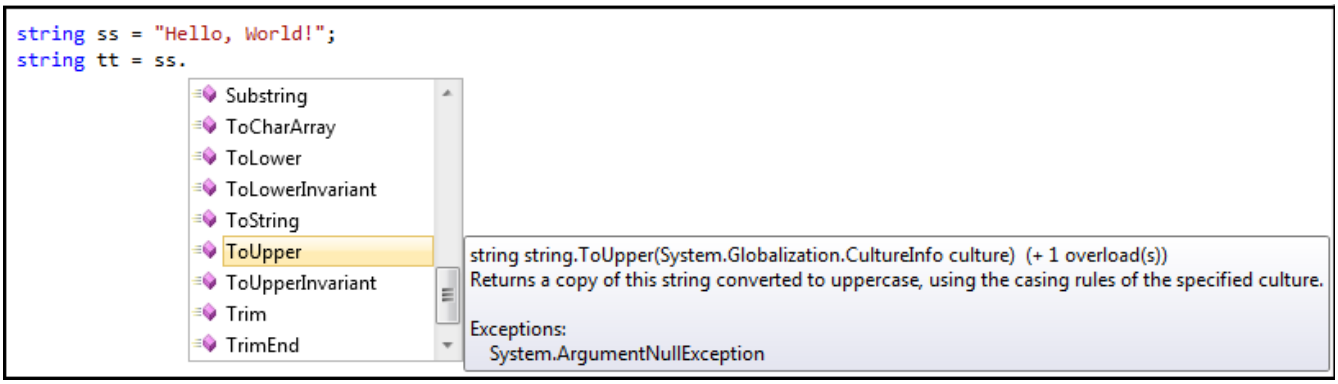
```
1  string ss = "Hello, World!";
2  string tt = ss.ToUpper();
3  MessageBox.Show(tt);
```

This code pops up this message box:

`ToUpper` is a method that can be invoked on any string object. It creates a new string, in which all the characters are in upper-case. (The original string `ss` remains unchanged.)

There are also methods such as `ToLower`, `Trim`, and `Split` that do other interesting things.

To learn what methods are available, you can consult the Help documentation, look for string methods, and read the documentation. Or, if you're a bit lazier, simply type your string name followed by a period. Visual Studio will pop up a selection box (Microsoft call this feature *IntelliSense*) showing all the methods (there are around 40 of them — thank goodness we'll only use a few of those!) that could be used on your string.



12.2. Working with the individual characters of a string

The **indexing operator** (C# uses square brackets to enclose the index) selects a single character from a string:

```
1  string fruit = "banana";
2  char m = fruit[1];
3  MessageBox.Show(string.Format("The character is {0}", m));
```

The expression `fruit[1]` selects character number 1 from whatever string the variable `fruit` refers to. This is assigned to variable `m`. When we display `m`, we could get a surprise: It displays an 'a'!

Computer scientists usually start counting from zero! The character at subscript position zero of `"banana"` is `b`. At position `[1]` we have the character `a`.

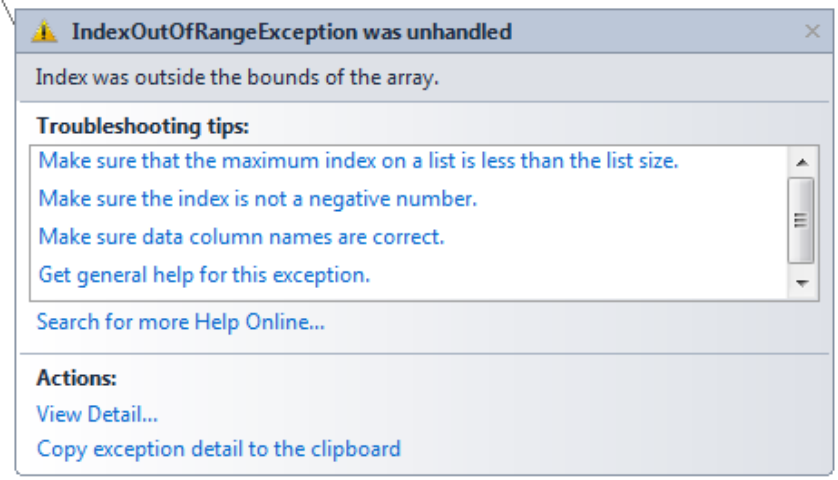
If we want to access the zero'th character of a string, we just place 0, or any expression that evaluates to 0, in the indexing brackets:

```
1  string fruit = "banana";
2  char m = fruit[25 / 5 - 5];
```

The expression in brackets is called an **index**, or a **subscript**. We'll use either word to mean the same thing.

If we attempt to access a character from an index position that does not exist, we'll get a runtime error...

```
string fruit = "banana";
char m = fruit[15];
```



Indexing works on strings, and can also be used on arrays, lists and other ordered collections of data. (Er ... we haven't done lists and collections yet, so our current take-away message is that indexing is general, if we understand it here and we'll be able to use it in other situations too.)

```
1  string[] dayNames = {"Sunday", "Monday", "Tuesday", "Wednesday",
2                       "Thursday", "Friday", "Saturday"};
3  string x = dayNames[3];
4  char c = dayNames[3][2];
```

This makes variable `x` point to the string "Wednesday", and it puts a 'd' into the variable `c`.

12.3. Substrings

The `Substring` method let's us create a new string from a section of the another string:

```
1  string s = "Pirates of the Caribbean";
2  MessageBox.Show(s.Substring(0,7));
3  MessageBox.Show(s.Substring(11,3));
4  MessageBox.Show(s.Substring(15));
```

This will show "Pirates", "the", and "Caribbean" in three successive message boxes.

The `Substring` method is used here in two different ways. (A method that can take various combination of arguments is called **overloaded**.) On line 2 we build a new string from the original, by taking the characters starting at index position 0, and taking exactly 7 characters. Line 3 above takes exactly 3 characters, starting from index position 11. But if you leave out the second argument to `Substring`, it takes all the rest of the string. So line 4 here takes a substring starting at index position 15, all the way up to the end of the string.

12.4. The `Length` property

The `Length` property of a string gives the number of characters in a string:

```
1  string fruit = "banana";
2  int n = fruit.Length;      // The value 6 is assigned to n
```

To get the last character of a string, you might be tempted to try something like this:

```
1  char last = fruit[fruit.Length];       // Oops!
```

That won't work. You'll get an `IndexOutOfRange` exception. The reason is that there is no character at index position 6 in `"banana"`. Because indexing is *zero-based* (i.e. we start counting at zero), the six indexes are numbered 0 to 5. To get the last character, we have to subtract 1 from the length of `fruit`:

```
1  char last = fruit[fruit.Length-1];     // Yay!
```

12.5. Strings are *read only*

It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
1  string myName = "Pexer";
2  myName[2] = 't';            // ERROR!
```

Strings are **read only**, which means you can't change an existing string once it has been created. The best you can do is create a new string that is a variation of the original:

```
1  string bookName = "Thank Sharply!";
2  string correctedName = bookName.Substring(0, 2) + "i" + bookName.Substring(3);
3  MessageBox.Show(correctedName);
```

Remember that the + operator means *concatenation* when we apply it to strings: so the strings are joined together to create a bigger string.

Understand this subtle point

The variable `bookName` in line 1 above 'points to' the string "Thank Sharply!". The *pointee* (the thing being pointed to) is read-only, and cannot be modified after it is created. But we can assign some other string to `bookName` (so it will then point to a new pointee). So the variable *can* be changed, but the thing it points to *cannot* be changed.

This code is therefore perfectly legal:

```
1  string bookName = "Thank Sharply!";
2  bookName = bookName.Substring(0, 2) + "i" + bookName.Substring(3);
3  MessageBox.Show(bookName);
```

Remember how assignment works: we first evaluate the expression on the right hand side, (in this case that evaluation creates a new string "Think Sharply!". Then we update the variable on the left to point to the new expression.

12.6. Traversal with `while`, `for` and `foreach` loops

Computations often involve processing a string one character at a time. Usually they start at the beginning, select each character in turn, do something with it, and continue until the end. This pattern of processing is called a **traversal**. One way to encode a traversal is with a `while` loop:

```
1  int ix = 0;
2  while (ix < fruit.Length)
3  {
4      char c = fruit[ix];
5      Console.WriteLine(c);
6      ix++;
7  }
```

This loop traverses the string and displays each character on a line by itself. The loop condition is `ix < fruit.Length`, so when `ix` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `fruit.Length-1`, which is the last character in the string.

Our advice in the previous chapter is that `for` loops ire easier when definite iteration is involved. So here is a better example that does the same thing:

```
1  for (int ix = 0; ix < fruit.Length; ix++)
2  {
3      Console.WriteLine(fruit[ix]);
4  }
```

Both examples above force us to think about the indexes, and we then use the index to extract the character we want. But we've previously seen how the `foreach` loop can easily iterate over the elements in a array without us needing to think about the indexes. It works for the characters in a string too (and it works for elements in other kinds of collections that we will cover later):

```
1  foreach (char c in fruit)
2  {
3      Console.WriteLine(c);
4  }
```

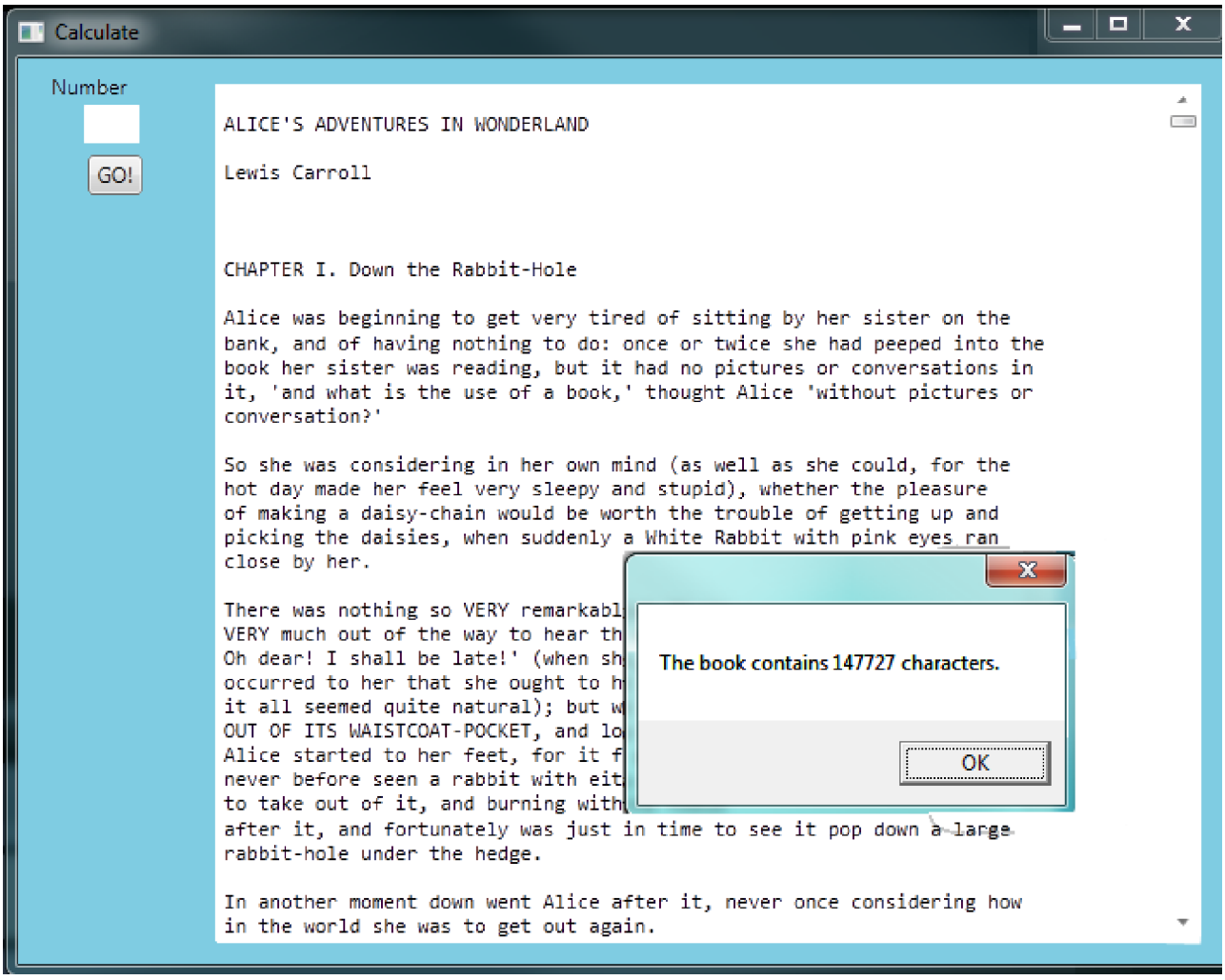
Each time through the loop, the next character in the string is assigned to the variable `c`. The loop continues until no characters are left. Here we can see the expressive power the `foreach` loop gives us compared to the

`while` or `for` loops when traversing a string.

Let's look at a bigger example now: we're going to read a book from a file on our disk into a string in the program. That will give us a longer string to work with.

```
1  private void button1_Click(object sender, RoutedEventArgs e)
2  {
3      string book = System.IO.File.ReadAllText("C:\\temp\\alice_in_wonderland.txt");
4      txtResult.Text = book;
5      MessageBox.Show(string.Format("The book contains {0} characters.", book.Length));
6  }
```

Line 3 we'll cover in more detail later in the course. But for now, it tells our program to find `File.ReadAllText` in a specific library called `System.IO`. The argument to the method is a fully qualified pathname to the file. Line 4 makes our GUI text box show the contents of the book (and in the screen snapshot below, the vertical scroll bar has also been turned on for the text box). Line 5 pops up a message box telling us how many characters are in the string.



Now let's do a string traversal. Let's say we want to convert all the text to lower-case, and then count how many times 'e' occurs in the book:

```
1  private int countEs(string s)
2  {
3      string slc = s.ToLower();
4      int count = 0;
5      foreach (char c in slc)
6      {
```

```
        if (c == 'e')
        {
            count++;
        }
    }
    return count;
}

private void button1_Click(object sender, RoutedEventArgs e)
{
    string book = System.IO.File.ReadAllText("c:\\temp\\alice_in_wonderland.txt");
    txtResult.Text = book;
    int c = countEs(book);
    MessageBox.Show(string.Format("The book contains {0} 'e's.", c));
}
```

A message box pops up to let us know that there are 13572 occurrences of the letter 'e' in the book.

Notice something important and interesting here: when we executed line 1 above, to create a lower-case version of the string, the original string remains unchanged. So the string displayed in the text box on our GUI does not change to lower-case. Remember that once they've been initially created, strings are read only.

We didn't need to make a separate method, but it seems like the kind of logic that we might want to adapt and reuse later. Also, making it a separate method allows us to add test cases with small examples:

```
Tester.TestEq(countEs("only one"), 1);
Tester.TestEq(countEs("morE than one"), 2);
Tester.TestEq(countEs("good day"), 0);
Tester.TestEq(countEs(""), 0);
```

12.7. String comparisons

Not all the comparison operators work on strings. We can directly check strings for equal or not equal, but the less than and greater than comparisons work a little differently:

```
if (fruit == "banana") ...  // this works
if (word != "hello") ...    // this works
if (fruit <= "banana") ...  // OOPS, ERROR!
```

Comparison are useful for ordering our strings, or putting words in *lexicographical* order. (This is similar to the alphabetical order we would use with a dictionary, except that in our computer representations, all the upper-case letters come before all the lower-case letters.)

In C#, every string has a very general `CompareTo` method that compares itself to some other string. (`CompareTo` works like this for other types of objects too, so we'll see it later in our course.)

The `CompareTo` method returns one of 3 outcomes — an integer which can be less than zero, exactly zero, or greater than zero. So `"apple".CompareTo("plum")` will return a negative integer, because "apple" comes before "plum" in the ordering. `"apple".CompareTo("apple")` will return zero, because the strings are identical. `"apple".CompareTo("a")` returns a positive value because "apple" comes after "a".

Of course we'll seldom see literal strings compared to each other: it is much more common to use string variables, and we'll often see the tests written like this:

```
string word1 = ...;
string word2 = ...;

string smaller = word2;
```

```
5  if (word1.CompareTo(word2) < 0)
6  {
7     smaller = word1;
8  }
9
10 Console.WriteLine("The word that comes first is " + smaller);
```

Line 5 asks *"Is word1 less than word2?"* If it is, the variable smaller is changed to reference `word1`. The way this code is written shows a typical programming *idiom* for setting a variable to the smaller of two values. At line 4 we make a tentative assignment into `smaller` — we guess that `word2` could be the smaller. Then we do the test, and if we find we guessed wrong, we fix it up.

Why do it this way? Because it is shorter and more convenient. Here is an alternative to lines 4–8 above that also works, but it feels rather more clunky!

```
1  string smaller;
2  if (word1.CompareTo(word2) < 0)
3  {
4      smaller = word1;
5  }
6  else
7  {
8      smaller = word2;
9  }
```

If we are fluent with the conditional expressions we saw earlier, it gets even easier:

```
1  string smaller = (word1.CompareTo(word2) < 0) ? word1 : word2;
```

Of course, it is important that we understand what we mean by "smaller than" for strings: we're not testing if one string is shorter than the other, we're asking if it comes before. So we need to be sure what we expect in a case like this:

```
1  bool b  = "kite".CompareTo("donkey") < 0;
```

12.8. Finding the index of a character in a string

What does the following method do?

```
1  private int IndexOf(string strng, char ch)
2  {
3     // Find the first ch in strng, and return its index position.
4     // Return -1 if ch does not occur in strng.
5
6    int ix = 0;
7    while (ix < strng.Length)
8    {
9        if (strng[ix] == ch)
10       {
11           return ix;
12       }
13       ix++;
14   }
15   return -1;
16 }
```

In a sense, `IndexOf` is the opposite of the indexing operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character first appears. If the character we seek is not found, the method returns `-1`.

This is another example where we see a `return` statement inside a loop. At line 11, if `strng[ix] == ch`, the method returns immediately, breaking out of the loop prematurely, and leaving the method.

If the character we seek doesn't appear in the string, then the program exits the loop normally, gets to line 15, and returns `-1`.

This pattern is common in searching algorithms: as soon we find what we are looking for, we can immediately stop looking.

Here is a version of the same logic, done with a `for` loop instead.

```
1  private int IndexOf(string strng, char ch)
2  {
3    for (int ix = 0; ix < strng.Length; ix++)
4        if (strng[ix] == ch) return ix;
5
6    return -1;
7  }
```

For conditional and looping statements, we may omit the braces that make a block (or compound statement) if we only have one statement in the block. We've done that here, making the code a bit shorter.

12.9. Overloaded methods

To find the locations of the second or third occurrence of a character in a string, we 'd like to be able to tell our `IndexOf` method where to start looking — so instead of always starting at position 0, it could start searching from some other position. This requires a new parameter, and one small change to the code.

```
1  private int IndexOf(string strng, char ch, int startPos)
2  {
3      for (int ix = startPos; ix < strng.Length; ix++)
4          if (strng[ix] == ch) return ix;
5
6      return -1;
7  }
8
9  ...
10 Tester.TestEq(IndexOf("banana", 'a', 2),  3);
```

The call `IndexOf("banana", 'a', 2)` now returns `3`, the index of the first occurrence of "a" in "banana" starting the search at index 2. Similarly, `IndexOf("banana", 'n', 3)` returns 4.

But the interesting idea here is that we can have have more than one method with the same name, provided the different methods have different numbers or types of parameters. In this case we say the method is **overloaded**. C# will analyse the arguments used at each call site, and will arrange to call the method with parameters that match the arguments.

> *Hey, shouldn't this be in one of the chapters about methods?*
>
> *"Perhaps. But if felt nicer to postpone it until we needed it."*

Let's provide a third overloading for `IndexOf`: in addition to providing a starting position for the search, we want to provide a count that determines the maximum number of characters that can be inspected. This code could look like this:

```
private int IndexOf(string strng, char ch, int startPos, int count)
{
    for (int ix = startPos; ix < startPos+count; ix++)
        if (strng[ix] == ch) return ix;

    return -1;
}

Tester.TestEq(IndexOf("C# is powerful", 'w', 2, 3),  -1);
Tester.TestEq(IndexOf("C# is powerful", 'w', 2, 10),  8);
```

We now have three overloadings of `IndexOf` — one that takes 2 arguments, one that takes 3, and one that takes 4. Here is another observation, and we'll redo the code. The one written last here is the most general — we can control both the count and the starting position. The less general one is the one with three arguments, and the least general one, (or the most specific) is the original one we wrote. We could rewrite all three like this:

```
private int IndexOf(string strng, char ch, int startPos, int count)
{
    for (int ix = startPos; ix < startPos+count; ix++)
        if (strng[ix] == ch) return ix;

    return -1;
}

private int IndexOf(string strng, char ch, int startPos)
{
    return IndexOf(strng, ch, startPos, strng.Length-startPos);
}

private int IndexOf(string strng, char ch)
{
    return IndexOf(strng, ch, 0, strng.Length);
}
```

So now the loop traversal only happens in one place rather than three separate places. We code the more specific versions of `IndexOf` by calling the general one that already works.

Here are some test cases that should pass:

```
string ss = "C# strings have some interesting methods.";
Tester.TestEq(IndexOf(ss, 's'), 3);
Tester.TestEq(IndexOf(ss, 's', 7), 9);
Tester.TestEq(IndexOf(ss, 's', 10, 5), -1);
Tester.TestEq(IndexOf(ss, 's', 10, 10), 16);
Tester.TestEq(IndexOf(ss, '.'), ss.Length - 1);
```

This code could crash!

There are a number of situations in which our `IndexOf` method could fail. As one example, if we provide a negative starting index for the search it will crash. There are some other problems too.

Write some unit tests to "stress-test" all the edge cases or weird situations you can imagine.

Do you think the method should handle all those cases and make itself bullet-proof, or is it acceptable just to say "This method only works for well-behaved inputs, and is designed to crash in other situations"?

12.10. The built-in `IndexOf` method on strings

Now that we've done all this work to write our own overloadings of `IndexOf`, we can reveal that strings already have their own built-in `IndexOf` method (with overloadings). They can do everything that our code can do, and more!

```
1  string ss = "C# strings have some interesting methods.";
2  Tester.TestEq(ss.IndexOf('s'), 3);
3  Tester.TestEq(ss.IndexOf('s', 7), 9);
4  Tester.TestEq(ss.IndexOf('s', 10, 5), -1);
5  Tester.TestEq(ss.IndexOf('s', 10, 10), 16);
6  Tester.TestEq(ss.IndexOf('.'), ss.Length - 1);
7  Tester.TestEq(ss.IndexOf("some"), 16);
8  Tester.TestEq(ss.IndexOf("C#"), 0);
```

The built-in `IndexOf` method has more overloaded options than our version (9 overloadings, in fact). Most usefully, as shown on lines 7 and 8 above, it can find substrings, not just single characters.

Usually we'd prefer to use the methods that C# provides rather than reinvent our own equivalents. But many of the built-in methods are good teaching exercises, and the underlying techniques you learn are your building blocks to becoming a proficient programmer.

You might want to also try some edge cases for the built-in version, and see how Microsoft handles these tricky cases.

12.11. The `Split` method

One of the most useful methods on strings is the `Split` method: it splits a single string into an array of individual substrings, removing all the white space between them. (White space means any tabs, newlines, or spaces.) This allows us to read input as a single string, and split it into words.

```
1  string ss = "Well I never did said Alice";
2  string[] words = ss.Split();
3  foreach (string wd in words)
4  {
5      Console.WriteLine(wd);
6  }
```

Line 2 initializes a new array of 6 strings, each one a word. These are then output underneath each other on the Console.

`Split` can also take arguments to make it break the original string on any set of characters. A popular usage would be to break the string at every newline character: you could break the Alice in Wonderland book into lines, for example, like this:

```
1  string[] lines = book.Split('\n');
2  MessageBox.Show(string.Format("There are {0} lines in the book.", lines.Length));
```

It is pleasing to know that this program says there are 3337 lines in the file, and that exactly matches what my text editor tells me!

12.12. Cleaning up your strings

We'll often work with strings that contain punctuation, or tab and newline characters, especially, as we'll see in a future chapter, when we read our text from a file (like the Alice in Wonderland book) or from the Internet. But if we're writing a program, say, to count word frequencies or check the spelling of each word, we'd prefer to strip off these unwanted characters.

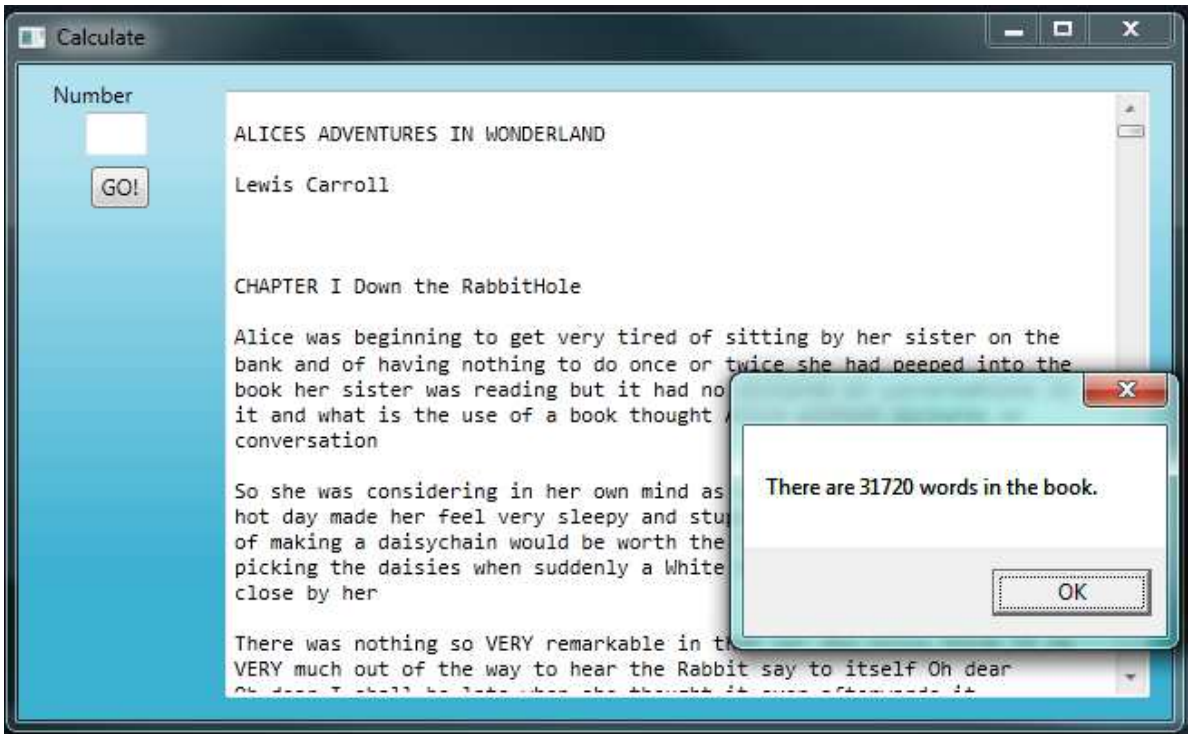
We'll show just one example of how to strip punctuation from a string.

```
1  private string remove_punctuation(string s)
2  {
3      string result = "";
4      foreach (char c in s)
5      {
6          if (! char.IsPunctuation(c)) {
7              result += c;                // This step is inefficient!
8          }
9      }
10     return result;
11 }
12 ...
13     Tester.TestEq(remove_punctuation("'Well, I never did!', said Alice."),
14                                                 "Well I never did said Alice");
15     Tester.TestEq(remove_punctuation("Are you very, very, sure?"),
16                                                 "Are you very very sure");
```

Composing together this method and the `Split` method from the previous section makes a useful combination — we'll clean out the punctuation, and `Split` will break the string into an array of words:

```
1  private void button1_Click(object sender, RoutedEventArgs e)
2  {
3      string book = System.IO.File.ReadAllText("c:\\temp\\alice_in_wonderland.txt");
4      string cleanedString = remove_punctuation(book);
5      txtResult.Text = cleanedString;
6      string[] words = cleanedString.Split();
7      MessageBox.Show(string.Format("There are {0} words in the book.", words.Length));
8  }
```

The output:



This runs quite slowly. The reason is at line 7 of `remove_punctuation`. Because strings are read only, we cannot add a new character onto the end of an existing string: we have to build a new copy of the string from scratch. Since there were more than 147 000 characters in the book, our program is building a new string for

each one! C# provides a mechanism called a `StringBuilder` to cater for this exact situation: so if you're looking for a challenge, see if you can speed up this code by using a more advanced feature.

12.13. Glossary

index

A variable or value used to select a member of an ordered collection, such as a character from a string, or an element from an array.

overloaded method

There can be more than one method with the same name, in which case we say the method is overloaded. Each overloading will have a different number of parameters, or different types of parameters. The C# compiler is clever enough to inspect the types and number of arguments at each call site, and work out which of the identically-named methods should be invoked.

read only objects

A value or object which cannot be modified. Assignments to a part of a string elements causes a runtime error.

traverse

To iterate through the elements of a collection (so far, a string or an array), considering each element in turn.

12.14. Exercises

1. What is the value of each of the following expressions:

```
"C#"[1]
"C#"[2]
"Strings are sequences of characters."[5]
"wonderful".Length
"app" + "le"
"appl" + 'e'
"Mystery".Substring(4)
"Mystery".Substring(4, 2)
"Mystery".IndexOf('y')
"Mystery".IndexOf('z')
"Mystery".IndexOf('y',3)
"Mystery".IndexOf('y',3, 2)
"apple".CompareTo("pineapple") > 0
"pineapple".CompareTo("Peach") == 0
```

2. Encapsulate

```
1  string fruit = "banana";
2  int count = 0;
3  foreach (char c in fruit)
4  {
5      if (c  == 'a')
6          count += 1;
7  }
8  Console.WriteLine(count);
```

   in a method named `count_letters`, and generalize it so that it accepts the string and the letter as arguments. Make the method return the number of characters, rather than show the answer.

3. Now rewrite the `count_letters` method so that instead of traversing the string, it repeatedly calls the `IndexOf` method, with the optional third parameter to locate new occurrences of the letter being

counted.

4. How many times does the word “queen” occur in the Alice in Wonderland book? Write some code to count them.
5. Use string formatting to produce a neat looking multiplication table like this:

```
      1   2   3   4   5   6   7   8   9  10  11  12
  :-----------------------------------------------
 1:   1   2   3   4   5   6   7   8   9  10  11  12
 2:   2   4   6   8  10  12  14  16  18  20  22  24
 3:   3   6   9  12  15  18  21  24  27  30  33  36
 4:   4   8  12  16  20  24  28  32  36  40  44  48
 5:   5  10  15  20  25  30  35  40  45  50  55  60
 6:   6  12  18  24  30  36  42  48  54  60  66  72
 7:   7  14  21  28  35  42  49  56  63  70  77  84
 8:   8  16  24  32  40  48  56  64  72  80  88  96
 9:   9  18  27  36  45  54  63  72  81  90  99 108
10:  10  20  30  40  50  60  70  80  90 100 110 120
11:  11  22  33  44  55  66  77  88  99 110 121 132
12:  12  24  36  48  60  72  84  96 108 120 132 144
```

6. Write a method that reverses its string argument, and satisfies these tests:

```
Tester.TestEq(reverse("happy"), "yppah");
Tester.TestEq(reverse("C#"), "#C");
Tester.TestEq(reverse(""), "");
Tester.TestEq(reverse("a"), "a");
```

7. Write a method that mirrors its argument:

```
Tester.TestEq(mirror("good"), "gooddoog");
Tester.TestEq(mirror("C#"), "C##C");
Tester.TestEq(mirror(""), "");
Tester.TestEq(mirror("a"), "aa");
```

8. Write a method that removes all occurrences of a given letter from a string:

```
Tester.TestEq(remove_letter('a', "apple"), "pple");
Tester.TestEq(remove_letter('a', "banana"), "bnn");
Tester.TestEq(remove_letter('z', "banana"), "banana");
Tester.TestEq(remove_letter('i', "Mississippi"), "Mssssppp");
Tester.TestEq(remove_letter('b', ""), "");
Tester.TestEq(remove_letter('b', "c"), "c");
```

9. Write a method that recognizes palindromes. (Hint: use your `reverse` method to make this easy!):

```
Tester.TestEq(is_palindrome("abba"), true);
Tester.TestEq(is_palindrome("abab"), false);
Tester.TestEq(is_palindrome("tenet"), true);
Tester.TestEq(is_palindrome("banana"), false);
Tester.TestEq(is_palindrome("straw warts"), true);
Tester.TestEq(is_palindrome("a"), true);
Tester.TestEq(is_palindrome(""),  ??);    // Is an empty string a palindrome?  You decide.
```

10. Write a method that counts how many times a substring occurs in a string:

```
Tester.TestEq(count("is", "Mississippi"), 2);
Tester.TestEq(count("an", "banana"), 2);
Tester.TestEq(count("ana", "banana"), 2);
Tester.TestEq(count("nana", "banana"), 1);
```

```
Tester.TestEq(count("nanan", "banana"), 0);
Tester.TestEq(count("aaa", "aaaaaa"), 4);
```

11. Write a method that removes the first occurrence of a string from another string:

```
Tester.TestEq(remove("an", "banana"), "bana");
Tester.TestEq(remove("cyc", "bicycle"), "bile");
Tester.TestEq(remove("iss", "Mississippi"), "Missippi");
Tester.TestEq(remove("eggs", "bicycle"), "bicycle");
```

12. Write a method that removes all occurrences of a string from another string:

```
Tester.TestEq(remove_all("an", "banana"), "ba");
Tester.TestEq(remove_all("cyc", "bicycle"), "bile");
Tester.TestEq(remove_all("iss", "Mississippi"), "Mippi");
Tester.TestEq(remove_all("eggs", "bicycle"), "bicycle");
```

13. Classes and Objects — an Overview

We've been working with Turtles, Windows, Buttons, and so on. We look a bit deeper under the covers now...

13.1. Creating our objects

Remember this first program where we used two turtles? Let's take another look.

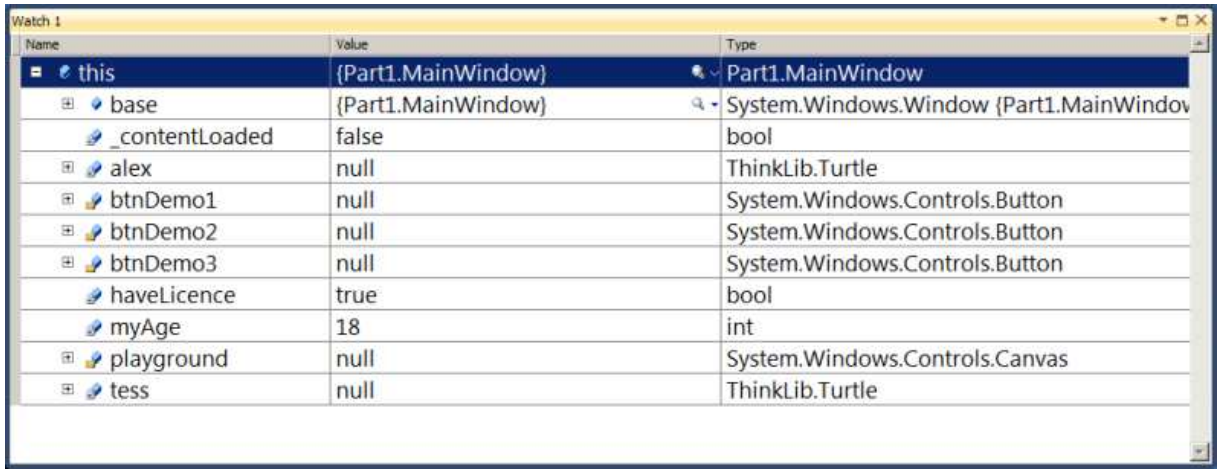
```
1  public partial class MainWindow : Window
2  {
3      Turtle tess, alex;    // Define class-level variables to hold references to turtles
4      int myAge = 18;       // Define and initialize some other class-level variables.
5      bool haveLicence = true;
6
7      public MainWindow()
8      {
9          InitializeComponent();
10         tess = new Turtle(playground, 100.0, 30.0);  // Create a turtle in the playground.
11         tess.BrushWidth = 5.1;                       // Set some properties.
12
13         alex = new Turtle(playground, 300.0, 100.0); // Create another turtle.
14         alex.LineBrush = Brushes.Blue;
15         alex.BodyBrush = Brushes.Blue;
16         alex.BrushWidth = 1;
17     }
18
19     ...
20 }
```

The class defines two class-level variables for the turtles we want to use. We've also added two other class-level variables with initializers here.

We don't have to create our `MainWindow` object ourselves — that is done automatically for us by WPF when our application starts running.

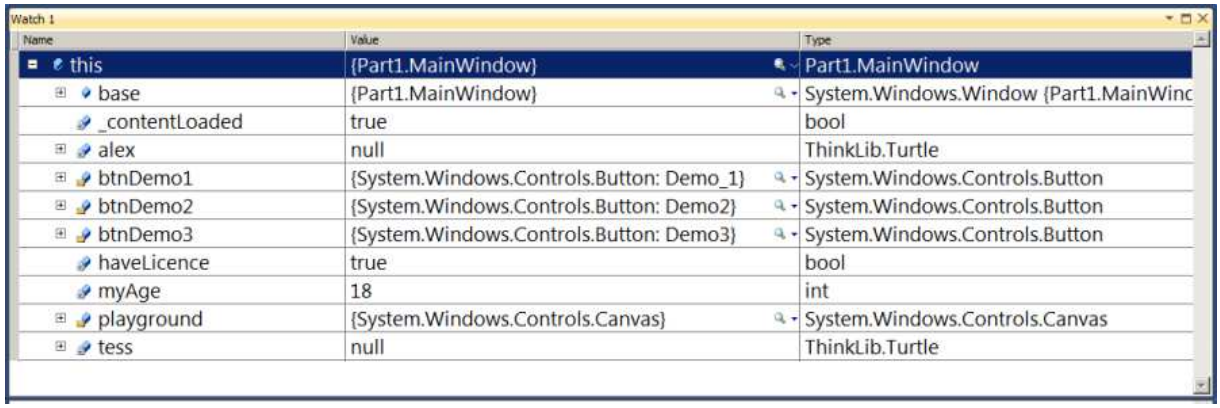
When our window object is first created it sets aside space within the new object for the variables we have defined. The window also has other built-in class-level variables, some automatically part of every window object, and some from the XAML controls we defined in our GUI front-end.

Part of creating any object involves "setting up its factory defaults". That is the job of the **constructor method** of any class (lines 7-17 in this example). But before that method executes, (e.g. if we set a breakpoint before line 9 executes), we can peek into memory and see what is inside our new window object:



What is significant is that some variables (our turtles and the playground) hold **references** to objects. A reference is sometimes called a **pointer**. Variables that can point to other objects start their existence holding the `null` value — a special value that says "I don't point to (or reference) anything yet". But the other kind of variables, value variables like `myage` and `haveLicence` already have their values.

Then at line 9 we call a method that creates and initializes all the window's controls (the `playground`, the buttons, any textboxes which we put there in the GUI designer, etc). Now memory looks like this ...



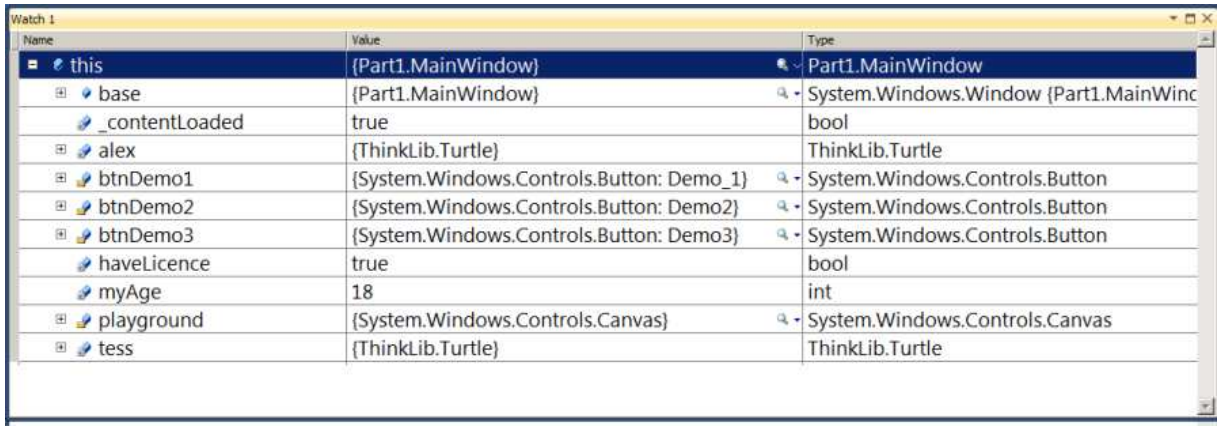
It tells us that the canvas and button objects have now been created (they are not `null` any longer), so we can start using them.

At line 10 we *instantiate*, or create, our first turtle. The keyword **new** says *"get the turtle factory to make us a new turtle"*.

Once the turtle has been created (and its constructor has set all its properties to their factory defaults), we're ready to save its reference into variable `tess`. Only after this can we start using Tess.

The same is done for Alex.

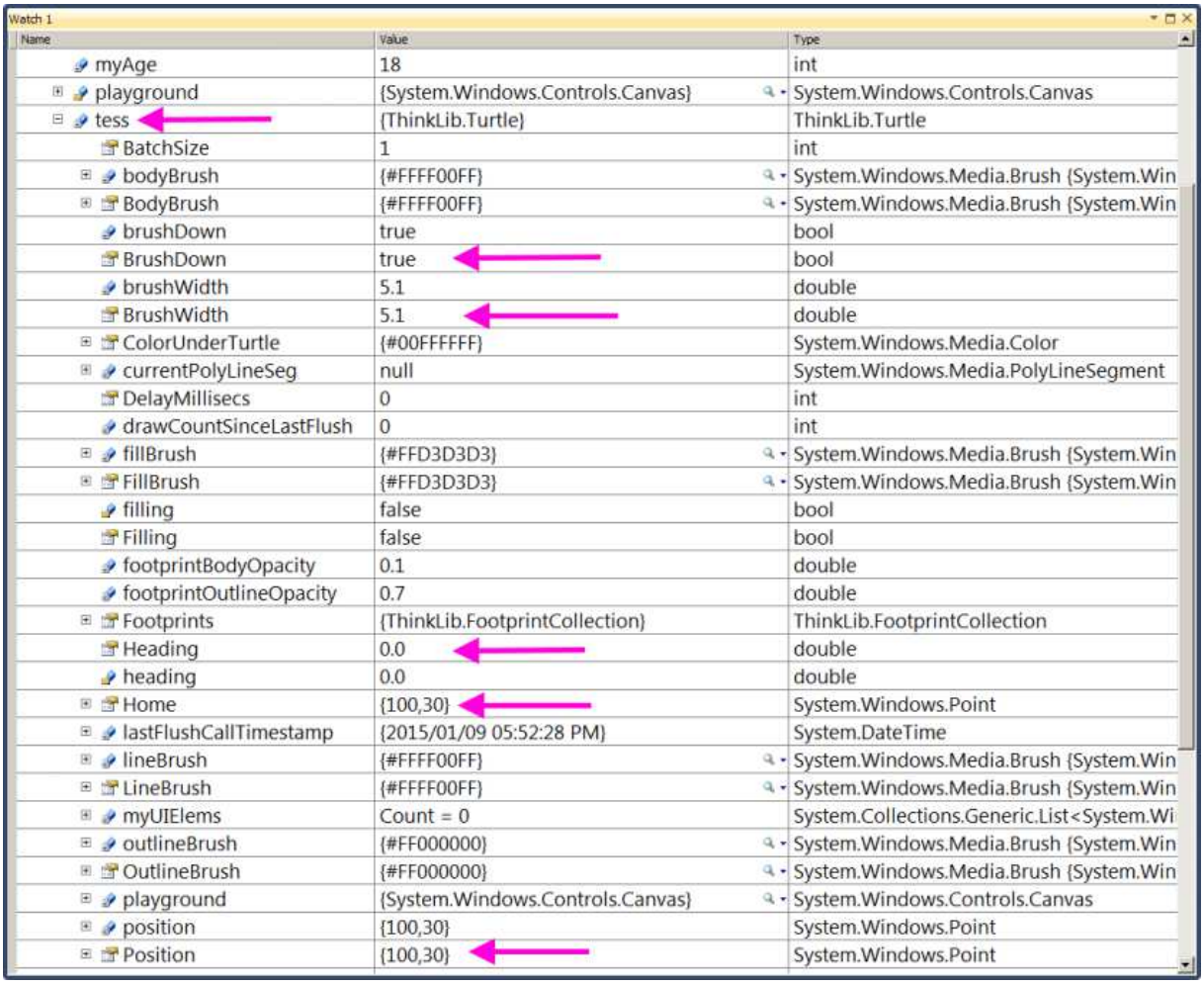
By time we reach line 19, we have this:



What does all this really mean?

Defining a Turtle variable doesn't make a turtle: it only gives us a variable that can reference a turtle. Creating (or instantiating) a turtle is a separate step.

Let's look at the previous debugging information: if we expand the tree of the object referenced by `tess` (by clicking on the '+'), we can also inspect the variables and properties in this turtle instance. We've highlighted a few of them that make up the turtle's state: its current heading and position, its home position, its brushwidth, and the fact that the brush is currently down.



13.2. What's the difference between a class and an object?

In our example, Turtle is a class. Window, Button, Canvas, etc. are all classes too.

A class is a *type* in C#. It is a blueprint that determines how to create objects, or instances, and how they can be used.

When we create an object, that object has the type of the class.

We like thinking about a class as a *factory* that can produce instances of a particular type.

So the Turtle class is a factory for turtle instances. Every time we execute `new Turtle(...)` we'll produce a new one. We could have a loop and create 25 turtles, or 25 buttons if we wished to. Each one is an **instance**, or an **object**.

13.3. Value Types vs Reference Types

Every type in C# is either a **value type** or a **reference type**.

Value types are built-in types like int, double, char, bool, byte, float, and some other types (from the C# libraries) like Color, DateTime, Point, Vector4, Matrix etc. When we define a variable of one of these types, we set aside enough memory to hold its value. So on most computers, an int needs 4 bytes of memory, while a double needs 8 bytes, and a bool only needs one byte.

By contrast, a variable for a reference type does not hold the object itself: it holds a reference (also called a pointer, or an address) to the object, while the object itself resides in an area of memory called the **heap**.

All object types (turtles, forms, buttons, arrays, lists, and a host of others that we are yet to cover) are reference types — the actual thing we're creating resides in the heap, and all we have to work with is a reference, or pointer, to where the object is.

NB! *Value type assignment and reference type assignment behave differently!*

So we need to be aware of this difference on every assignment statement, And let us not forget that we also make an assignment when we pass an argument to a parameter, so this also applies there.

It is important that we understand whether we are copying a value into a variable, or whether we are copying a reference. Here is why...

```
1  int x = 42;
2  int y = x;
3  x = x + 100;
4
5  Turtle tess = new Turtle(playground);
6  Turtle t = tess;
7  tess.Right(90);
```

In line 2, the value in x is copied into y. So at that time, x holds 42. And y holds 42. But they are different 42s. Now on line 3 we change the value in x. But y is a different variable with its own value, so y still keeps its value of 42.

By contrast, in line 5 we create a new turtle and make the variable `tess` reference it. Then we copy the same reference (address, or pointer) into another variable `t`. We now have two variables. `tess` and `t`, *but there is still only one turtle*. Both variables refer to the *same* turtle in the heap.

So at line 7, we turn `tess` to the right by 90 degrees. Then what is turtle `t`'s heading? If we said 90 we'd be correct.

So we have two variables in the program referencing the same underlying object.

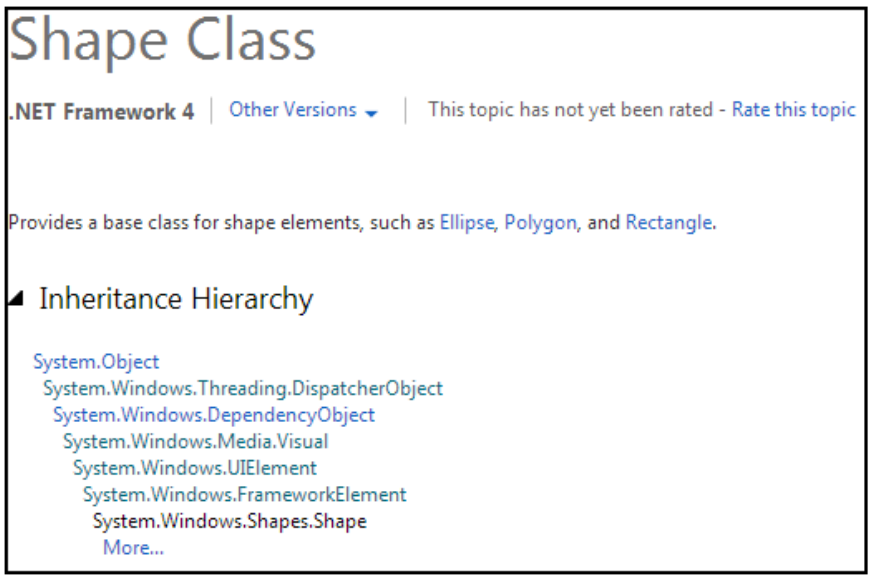
This isn't new. If we look back at the first example in the chapter "Void Methods", we passed the turtle `tess` to a method called `drawSquare`. Within that method the parameter name `t` refers to the same turtle as the variable `tess` does: there is still only one turtle, but it now has two names referring to it.

When more than one reference points to the same object, we say the object has an **alias** — there is more than one name that it is known by in different parts of the program.

13.4. Inheritance

We know that a dog is a kind of mammal. Humans, cows and pigs are also mammals (mammals have mammary glands, so they have milk). A mammal is a kind of vertebrate (has a backbone). Reptiles like snakes are also vertebrates. And of course, vertebrates are one kind of animal, but so are invertebrates like jellyfish, flies and worms.

Just as the types of the animal kingdom are organized into a hierarchy (a tree), so too are our C# types organized into a hierarchy. If we define a variable of type `Shape` in our code, put the cursor on the type Shape, and hit the `F1` key to get help, we'll get to a page with a fragment that looks like this:



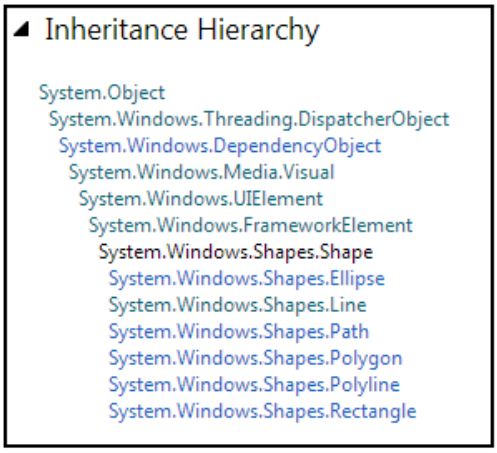
What that inheritance tells us is that our objects in WPF are even more complicated than the animal kingdom! But we can read from this that any Shape object **is-a** [1] kind of FrameworkElement, which in turn is-a kind of UIElement, which in

turn ... So as we move up the inheritance we get more general.

[1] *is-a* is used in computer science to tell us about a relationship: that X is a type of Y. By hyphenating the words so that they look a bit different from the normal English "is a" we can make our intentions and meaning clearer.

For using WPF, the UIElement is important at this time. Of course the UI stands for User Interface - our window and everything in it. So any type of object that falls anywhere below UIElement can be put into our window somehow.

Clicking on `More...` in the help above lets us go further down the hierarchy. We now get this:



This tells us that there are six specific built-in types of Shape that we could use in WPF: Ellipse, Line, Path, Polygon, Polyline, and Rectangle. And, of course, an Ellipse is a kind of Shape which is a kind of UIElement...

13.5. Summary

At this stage it is probably not too important in our C# programming development that we know about the huge number of classes that are available. And we're probably not going to need too much detail about this specific inheritance hierarchy unless we become WPF wizard programmers.

But what is important are three big ideas:

- Some types are value types: their values are stored directly. Some are reference types. We store pointers to reference types, and have to be careful when copying them, because we can create aliases.
- Objects have internal structure and organization (we saw this by peeking inside `MainWindow` and inside `tess`).
- Types are related to each other via inheritance. And even though your own type is Human [2], at the same time you are also an instance of Mammal, and an instance of Vertebrate, and an instance of Animal. The message for programming is that although an object might be a Button, it can also be used as a FrameworkElement, or a UIElement, or an Object. One thing can be all of these types simultaneously.

[2] *If there are any aliens or fish enjoying this book, please drop me an email.*

Later in the course we'll start to create our own classes and we'll learn more about inheritance.

13.6. Glossary

aliases

Multiple references to the same object.

class

A class is type. It is a blueprint that determines how to create objects, or instances, and how they can be used.

constructor method

A special method in a class that is automatically called whenever a new instance is created. Its job is to set up the internal state of the new object to its initial factory defaults.

heap

A region of memory that holds objects. Objects are created with the `new` keyword. They live in the heap until the system figures out that there are no references left that can possibly reach them, at which time the available memory will be freed up for other use.

is-a

An abbreviation of "is a" as in "A human *is a* mammal". It describes a relationship that shows that an object with a specific type, e.g. Button, can also be treated as an object with a more general type, e.g. a UIElement.

null

A special value that can be assigned to any reference variable. It means "this variable doesn't point to any object".

object

A thing created in the heap. Its type is the class (or blueprint) that it was created from. It can be referenced, or pointed to. Variables can hold references or pointers to an object.

pointer

Another name for a reference.

reference

A reference is an address of some memory location in the heap. It allows us to locate an object in the heap.

reference type

Opposite of a value type. A type (or class) from which instances are created in the heap. The objects must always be accessed via a reference. When an object is passed into a method or assigned to another variable, only the reference is duplicated. This means the object gets another alias.

value type

Opposite of a reference type. Anything that is stored directly. When value types are assigned or passed as arguments to methods, the value is copied and the two copies become completely independent of each other.

13.7. Exercises

1. Consider this fragment of code:

```
Turtle Tess = new Turtle(playground);
Turtle Alex = Tess;
Alex.BrushWidth = 6;
```

   Does this create one or two turtle instances? Does setting the BrushWidth of Alex also change the BrushWidth of Tess? Explain in detail.

2. Create and instantiate two turtles, Tess and Alex. Make Tess pink, make Alex blue. Now predict what will happen after executing this code, and then check if you were correct.

```
Turtle Bert = Tess;
Tess = Alex;
Alex = Bert;
Alex.Forward(100);
Bert.Right(50);
Tess.Stamp();
Tess.Forward(10);
```

3. A turtle's appearance can be changed. So this line of code can make our turtle look like an amoeba (well, perhaps more like an ellipse).

```
tess.SetAppearance(new EllipseGeometry(new Point(0, 0), 20, 10), Brushes.Red, Brushes.Blue);
```

   Explore the WPF inheritance hierarchy for `Geometry` — specifically, what specialized kind of Geometries exist. Instantiate one, and get Tess to have a different shape.

4. A turtle can stamp any `UIElement` onto its playground at its current position and orientation. Here is an example that stamps 5 buttons:

```
string[] daynames = { "Mon", "Tue", "Wed", "Thur", "Fri" };
tess.Reset();
tess.WarpTo(200, 200);
for (int i = 0; i < 5; i++)   // A five-sided regular polygon is called a pentagon.
{
    Button b = new Button();  // Make a new button for the current corner.
    b.Content = daynames[i];  // Set the caption on the button.
    tess.Stamp(b);            // Stamp the button into the playground, where tess is at the moment.
    tess.Forward(100);
    tess.Left(360 / 5);
}
```

a. Use Help, and write down the full inheritance hierarchy from `Button` is-a ?? is-a ?? ... is-a `System.Object`.

b. True or false? `Image` is-a `UIElement`?

c. Change the code so that Tess stamps an `Image` control showing a ladybug at every corner of a pentagon.

Here's a hint... Right-click and save this picture, then ...

```
Image img = new Image();
img.Source = new BitmapImage(new Uri("c:\\temp\\ladybug.png"));
```

14. Arrays

An array is a structure that can hold a number of individual values at the same time.

The individual values that make up an array are called its **elements**, or its **items**. (We will use either term to mean the same thing.) All the elements in an array must be of the same type: so we can have an array of students, an array of integers, an array of strings, or an array of turtles.

Like strings, individual elements in the array can be accessed by *indexing*. The first element is always at position 0, the next at position 1, and so on. Unlike strings, though, each element of the array is a variable, so an element can by changed by assigning another value to it as the program runs. (Recall that strings are read-only — we cannot assign to individual elements.)

14.1. Defining and initializing arrays

We've already seen the easiest way to define and initialize small arrays:

```
1  string[] daysOfWeek = { "Sun", "Mon", "Tue", "Wed",
2                                "Thur", "Fri", "Sat" };
3  int[] daysInMonth = { 31, 28, 31, 30, 31, 30,
4                        31, 31, 30, 31, 30, 31 };
```

The first example defines and initializes an array of seven strings. (The initializer is the bit in braces.) The second defines and initializes an array of 12 integers.

We read `string[] daysOfWeek` as *string array* called *daysOfWeek*. Do learn to mentally group the array definition brackets with the type — the new type is "int array" or "string array".

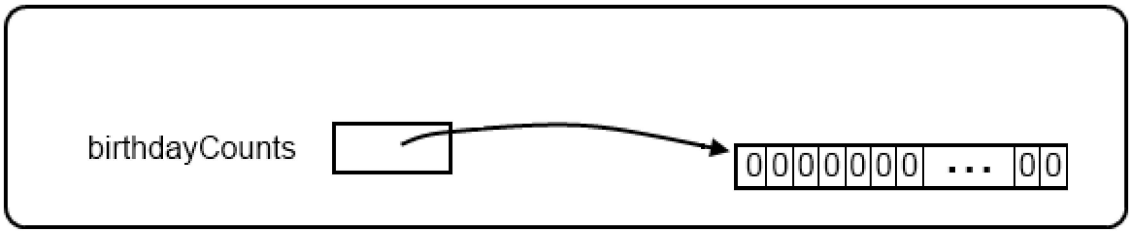
An array is an object. One needs to define a variable that can hold a reference to the object. And, as a separate step, we need to create the object and make the variable reference it.

So let's imagine that we want to count how many students will have a birthday on each day this year. (We will assume it is not a leap year). We'd need 365 counters to keep track of how many birthdays we've seen so far on each day of the year. Each student calculates their "day index number" corresponding to their birthday (1 Jan would be index 0, 31 Jan would be index 30, 1 Feb would be index 31, 31 December would be index 364, etc.) If the student's birthday is on day 42, then we'd increment element 42 in the array.

We start like this:

```
1  int[] birthdayCounts;
2  ...
3  birthdayCounts = new int[365];
```

Line 1 defines a variable that can reference an array of int. It gives it an initial value `null`. Line 3 (always evaluate the right-hand side of an assignment first) creates a new object — an array of 365 integers — in the heap. It then assigns the reference to that object to the variable. Recall that if we don't give an integer an initial value, C# will set it to 0 for us. So all 365 ints in the array are now zero. So memory looks like this now:



So defining an array doesn't give us an array. It gives us a variable that can reference an array. We strongly encourage to using the single-stepping debugger and inspector to watch the variable `birthdayCounts` as we execute the lines above. The idea that these variables do not hold their values directly, but they hold a reference to the value that is created separately was covered in the previous chapter.

C# tries to make our lives easier, so they allow little short cuts. For example, we can combine lines 1 and 3 above into a single line:

```
1  int[] birthdayCounts = new int[365];
```

Even though there is only one line of code, there are still three distinct things going on here. The right hand side creates the array, and initializes its elements. The left hand side creates a variable that can hold a reference to an array of int. Then the assignment makes the variable point to the array.

So the easy syntax we've used to create small arrays with an initializer is also just shorthand:

```
1  string[] daysOfWeek = { "Sun", "Mon", "Tue", "Wed",
2                          "Thur", "Fri", "Sat" };
3  int[] daysInMonth = { 31, 28, 31, 30, 31, 30,
4                        31, 31, 30, 31, 30, 31 };
```

This short syntax is entirely equivalent to the more verbose:

```
1  string[] daysOfWeek = new string[7] { "Sun", "Mon", "Tue", "Wed",
2                                        "Thur", "Fri", "Sat" };
3  int[] daysInMonth =  new int[12] { 31, 28, 31, 30, 31, 30,
4                                     31, 31, 30, 31, 30, 31 };
```

Here the explicit `new` step reminds us that we have to instantiate the array. If we use this syntax, the C# compiler will double-check that the length of the initializer matches the number of elements we allocated for the array. So the information is actually redundant — it can tell how big the array needs to be by looking at the initializer first. So if we provide an initializer we can leave out the explicit size too:

```
1  string[] daysOfWeek = new string[] { "Sun", "Mon", "Tue", "Wed",
2                                       "Thur", "Fri", "Sat" };
3  int[] daysInMonth =  new int[] { 31, 28, 31, 30, 31, 30,
4                                   31, 31, 30, 31, 30, 31 };
```

Sometimes having the programmer supply the number of elements in the array explicitly can be a good thing. For example, if we left out one of the initializer values when typing in the `daysInMonth` initializer,

we'd either end up with only 11 months in our year, or we'd get a compilation error if we explicitly said "expect 12 elements".

There are other ways to create arrays also. In the chapter on strings we also saw the `Split` method which takes a string and splits it into pieces, returning an array of strings. We were able to break *Alice in Wonderland* into an array of strings. We did this twice. The first time, each string was just one word from the book. In the other case, each string was a whole line from the book. So there are some methods like this that we'll encounter that can create and initialize an array of values for us.

Once an array is created its number of elements stays fixed. (This is not entirely true, but trying to resize arrays is beyond our scope, and is probably a bad idea anyway.)

14.2. Accessing elements

The syntax for accessing the elements of an array is the same as the syntax for accessing the characters of a string — the index operator: `[]`. The expression inside the brackets specifies the index. Remember that the indices start at 0. So `daysOfWeek[2]` has the value `"Tue"`.

Any expression that evaluates to an integer can be used as an index:

If we try to access or assign to an element at a position that is outside the bounds of the array, we'll get an exception (error) saying "Index was outside the bounds of the array".

It is common to use a loop variable as an index for traversing all the elements of an array. Like strings, every array has a `Length` property that determines the number of elements in the array. So let's consider this method that sums up an array of ints:

```
1  private int sumElems(int[] xs)
2  {
3      int total = 0;
4      for (int i=0; i < xs.Length; i++)
5      {
6         total += xs[i];
7      }
8      return total;
9  }
10
11  ...
12    int[] daysInMonth = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
13    Tester.TestEq(sumElems(daysInMonth), 365);
```

Each time through the loop, the variable `i` is used as an index into the array, accessing the `i`'th element. This pattern of computation is called an **array traversal**.

The example shows how to define a parameter that is an array (line 1), and shows that arrays can be passed as arguments (line 13)

The above sample doesn't need or use the index `i` for anything besides getting the items from the array, so this more direct version — where the `foreach` loop gets the items for us — might be preferred:

```
1  private int sumElems(int[] xs)
2  {
3      int total = 0;
4      foreach (int val in xs)
5      {
```

```
6           total += val;
7       }
8       return total;
9   }
```

The test on line 13 still passes.

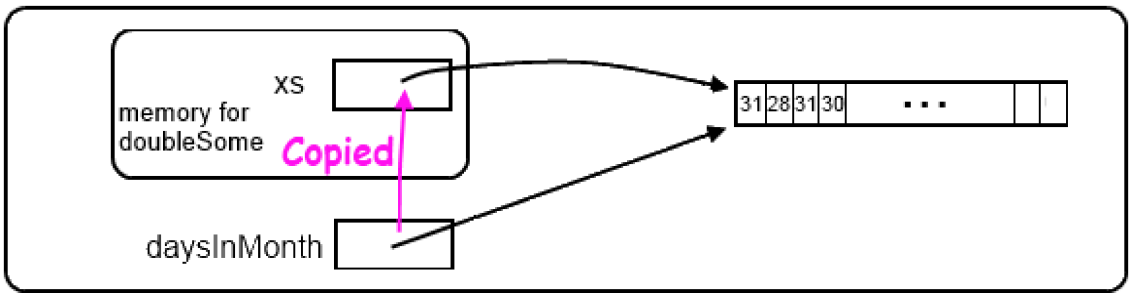
14.3. Modifying elements of an array

Unlike read-only strings, we can assign new values to elements of an array. So let's start with our `daysInMonth` array above, and we'll traverse all the elements, doubling any element that is odd.

```
1   private void doubleSome(int[] xs)
2   {
3       for (int i=0; i < xs.Length; i++)
4       {
5          if (xs[i] % 2 == 1)              // is it odd?
6          {
7             xs[i] *= 2;
8          }
9       }
10  }
11
12   ...
13     int[] daysInMonth = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
14     doubleSome(daysInMonth);
15     ...
```

After this operation (say at line 15) we could inspect the array (using the debugger), or output the elements (one at a time), and we'd confirm that the `daysInMonth` array now contains the values { 62, 28, 62, 30, 62, 30, 62, 62, 30, 62, 30, 62 }.

We previously saw that when we pass a reference-type argument (`daysInMonth`, in this case) to a parameter (`xs` in this case) we make a copy of what is in `daysInMonth`. But what gets copied is the *reference* to the array, not the array itself. So if we inspect the state of memory when we get to line 3 in the code above, we'd have this situation:



This shows that while we are executing `doubleSome`, the two variables `xs` and `daysInMonth` are aliases. So if we make changes to elements of `xs`, we're actually changing the same array as is referenced by `daysInMonth`.

14.4. Methods that return arrays

Methods can return objects. So they can return arrays. A method that needs to return an array should a) find out how many elements are required, b) create the array of the correct size c) fill the array with the

values to be returned, d) return the array to the caller.

Let us generate and return an array containing the first 100 prime numbers. (We'll assume we did the exercise to write `isPrime` in the chapter about iteration.)

```
private int[] generateArrayOf100Primes()
{
    int[] ps = new int[100];  // Create an array of 100 items

    int numFound = 0;
    int candidate = 2;

    while (numFound < 100)
    {
        if (isPrime(candidate))
        {
            ps[numFound] = candidate;
            numFound++;
        }
        candidate++;
    }
    return ps;
}
```

We should be able to inspect (or print) our data, and confirm that the 100th prime number is indeed 541.

14.5. Arrays of Arrays

We often want arrays of arrays, to represent two-dimensional data, or sometimes even more dimensions. For example, suppose we have rainfall data from three different sites, for each of the four quarters of the year. We might end up with a table like this:

Quarter	site1	site2	site3
Q1	165.3	170.5	172.1
Q2	149.6	140.3	152.3
Q3	44.3	42.4	45.0
Q4	95.3	89.8	92.4

How could we represent this data in a C# program? By creating an array where each element was another array representing one of the quarters. (Arrays of arrays are often called *jagged arrays*.)

```
double[][] rainfall =
    { new double[] { 165.3,  170.5,  172.1 },  // Q1
      new double[] { 149.6,  140.3,  152.3 },  // Q2
      new double[] {  44.3,   42.4,   45.0 },  // Q3
      new double[] {  95.3,   89.8,   92.4 }   // Q4
    };
```

With an array of arrays, a traversal of all the elements will need two loops, one nested in the other. So let us find the average rainfall for each quarter. This means we'll need to sum up across each row, divide by 3, and we'll get four averages — one for each quarter.

```
// Create an average reading for each quarter
double[] avgs = new double[4];
for (int q = 0; q < 4; q++)
{
    int[] currQuarter = rainfall[q];
    double sum = 0;
    for (int col = 0; col < 3; col++)
    {
        sum += currQuarter[col];
    }
    avgs[row] = sum / 3.0;
}

Tester.TestEq(avgs,  new double[] {169.3, 147.4,  43.9, 92.5}, 0.1);
```

At this point we can inspect the `avgs` array, or display the values and confirm that they match. We've used a unit test here, with an extra third argument that provides for some tolerance for possible errors because we're working with floating point values. So this test says "call the numbers equal if they're within 0.1 of each other".

Here is a jagged example: Each student in a university is registered for some subjects:

```
string[][] registrations =
  { new string[] { "Maths",  "CompSci",  "Music", "Physics" },
    new string[] { "Maths",  "CompSci", "Economics" },
    new string[] { "CompSci", "Economics", "Accounting", "Law" },
    new string[] { "Philosophy" }
  };
```

14.6. Glossary

element

One of the values in a array (or other sequence). The bracket operator selects elements of an array. An element is also called an *item*.

index

An integer value that indicates the position of an item in an array. Indexes start from 0.

item

See *element*.

jagged array

An array of arrays. Not all "rows" of data need to have the same number of elements (so it could look jagged!)

14.7. Further Reading

Microsoft's Arrays Tutorial, at http://msdn.microsoft.com/en-us/library/aa288453(v=vs.71).aspx

14.8. Exercises

1. Write a method to return the biggest item from a non-empty array of int. Provide some test cases to test your method.

2. Write a method to return the sum of all the odd numbers in an array of int. Provide some test cases to test your method.
3. Write a method to search for a given target string in an array of strings. The method should return the index at which the target is found. If the target is not found, it should return -1. Provide test cases to test all the important cases: the target could match the first element or the last element in the array, or some element in the middle, or it may not exist in the array at all.
4. Use the method above to write a method that turns a month name into a corresponding month number, so that these tests pass:

```
Tester.TestEq(monthName("January"), 1);
Tester.TestEq(monthName("June", 6);
Tester.TestEq(monthName("November"), 11);
```

5. Assume we have this definition in our code:

```
int[] daysInMonth = new int[] { 31, 28, 31, 30, 31, 30,
                                31, 31, 30, 31, 30, 31 };
```

   Write a method that takes a day number and a month name, and returns the day number within the (non-leap) year. Assume days are numbered starting from 0. For example, `dayMonthToDay("March", 12)` should give the result 70.

6. Arrays can be used to represent mathematical *vectors*. In the next few exercises we will write methods to perform standard operations on vectors. Write C# code to pass the tests in each case.

   Write a method `dotProduct(u, v)` that takes two arrays of numbers of the same length, and returns the sum of the products of the corresponding elements of each (the dot product).

```
Tester.TestEq(dotProduct(new double[] {1, 1}, new double[] {1, 1]},  2);
Tester.TestEq(dotProduct(new double[] {1, 2}, new double[] {1, 4.5]}, 10);
Tester.TestEq(dotProduct(new double[] {1, 2, 1}, new double[] {1, 4, 3}), 12);
```

7. Write a method `addVectors(u, v)` that takes two arrays of doubles of the same length, and returns a new array containing the sums of the corresponding elements of each:

```
double[] v1 = {1, 1};
double[] v2 = {2, 2};
double[] v3 = {1, 2};
double[] v4 = {1, 4};
double[] v5 = {2, 6};
double[] v6 = {1, 2, 1};
double[] v7 = {1, 4, 3};
double[] v8 = {2, 6, 4};

Tester.TestEq(addVectors(v1, v1), v2);
Tester.TestEq(addVectors(v3, v4), v5);
Tester.TestEq(addVectors(v6, v7), v8);
```

8. Write a method `scalarMult(s, v)` that takes a number, `s`, and a array, `v` and returns the scalar multiple of `v` by `s`:

```
Tester.TestEq(scalarMult(5.5, new double[] {1, 2}), new double[] {5.5, 11.0});
```

```
Tester.TestEq(scalarMult(3, new double[] {1, 0, -1}), new double[] {3, 0, -3});
Tester.TestEq(scalarMult(7, new double[] { 3, 0, 5, 11, 2 }),
                            new double[] { 21, 0, 35, 77, 14 });
```

9. *Extra challenge for the mathematically inclined*: Write a method `crossProduct(u, v)` that takes two arrays of doubles of length 3 and returns their cross product. Write your own tests.
10. In the chapter on Iteration we had two problems (the Drunk Pirate, and Tess draws a House) that used "parallel arrays": we had an array of `turns` for the angle to turn the turtle, and an array `steps` for the distance to move the turtle. The i'th element of the one array was (implicitly) associated with the i'th element of the other array.

    Re-solve those problems using a jagged array instead. Each row in the outer array contains an inner array with two elements — a step and a turn.

15. Lists

Lists and arrays are similar to each other in many ways; for the moment we can think of a list as a supercharged kind of array with some nice extra features. We highlight the essential differences as we go along. Let's just review what we already know about arrays, because all this information applies to lists too.

Like an array, a list is an ordered **collection** of values. We'll see other kinds of collections later, e.g. a *dictionary*.

The individual values that make up a list (or an array) are called its **elements**, or its **items**. (We will use the term `element` or `item` to mean the same thing.) All the elements in a list (or array) must be of the same type: so we can have a list (or array) of students, or a list of integers, or a list of strings, or a list of turtles.

Like strings and arrays, individual elements in the list can be accessed by *indexing*. The first element is always at position 0, the next at position 1, and so on. This means the whole collection (the list or the array) is *ordered* in the sense that one string comes before another in the structure. (Take note: an *ordered* collection means "they're in a definite known sequence". This is not the same as a *sorted* collection, which means they're in ascending (or perhaps descending) order of values.)

15.1. What's the key difference from arrays?

Once an array has been created, its length remains unchanged.

A list is a *dynamic* structure: it can grow or shrink as our program runs. We can easily add new elements to the existing list, or remove elements. To do this, (and some other things that arrays cannot do), a list has a number of convenient methods available.

15.2. Defining and initializing lists

```
1  List<string> weekdayNames = new List<string>() { "Sun", "Mon", "Tue", "Wed",
2                                                    "Thur", "Fri", "Sat" };
3  List<int> daysInMonths = new List<int>() { 31, 28, 31, 30, 31, 30,
4                                             31, 31, 30, 31, 30, 31 };
5  List<double> readings = new List<double>();
6  List<Button> btns = null;
```

The first example defines and initializes an array of seven strings. The second defines and initializes an array of 12 integers. The elements of a list must all be of the same type.

The form `List<T>` is called a **generic** list. The T between the angle brackets is a **type parameter**. It allows us to substitute any type for T. So in line 1 in the definition above, we've substituted the type `string`, and have a list of strings. Similarly, in line 3, we define a `List<int>`. Both these definitions also have initializers, so those elements are added to the list. In line 5, we have an example where we instantiate a list, but it will not contain any elements at this point. In line 6 we define a variable that can reference a list of Button elements, but at this stage the list has not yet been instantiated, so we initialize it with the value `null`;

Lists, like arrays, are reference types: a variable refers to the actual object, which must be instantiated in the heap.

Notice also a very important difference between lines 5 and 6: `readings` references a list that has no elements in it. But `btns` has the value `null` — it doesn't reference anything.

15.3. Accessing elements

The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string or the elements of an array — the index operator: `[]`. The expression inside the brackets specifies the index. Remember that the indices start at 0. So `daysInMonths[2]` has the value 31.

With arrays we have a `Length` property that tells us how many elements are in the array. With lists, (and all other collections that we'll work with) the property is `Count`.

So let's provide two methods to show how we would sum the elements in a list of int:

```
1  private int sumList1(List<int> xs)
2  {
3      int sum = 0;
4      for (int i = 0; i < xs.Count; i++)
5      {
6          sum += xs[i];
7      }
8      return sum;
9  }
10
11 private int sumList2(List<int> xs)
12 {
13     int sum = 0;
14     foreach (int x in xs)
15     {
16         sum += x;
17     }
18     return sum;
19 }
20
21 private void button1_Click(object sender, RoutedEventArgs e)
22 {
23     List<int> daysInMonths = new List<int>() { 31, 28, 31, 30, 31, 30,
24                                                31, 31, 30, 31, 30, 31 };
25     int v1 = sumList1(daysInMonths);
26     int v2 = sumList2(daysInMonths);
27     Tester.TestEq(v1, v2);
28 }
```

Line 1 demonstrates how we define a list parameter. Line 6 shows how we index the list.

The `foreach` in line 14 works because a list is an *enumerable* structure. We say that the `foreach` **enumerates** the items in the collection.

And lines 25 and 26 show that passing a `List<int>` argument to a method is just like passing a string, an array, or a turtle.

15.4. When are lists really special?

Because lists can grow and shrink in size, they're ideal for problems that need that kind of flexibility. Let's start with a very simple problem: Write a method to return all the prime numbers less than some number N.

We need to return a list or an array, because there are potentially many such primes. But because we don't know how many primes there are going to be when N=1000, we can't allocate a fixed-size array to hold the answer. So we need a structure that can grow as we discover each new prime.

```
1  private List<int> findPrimesLessThan(int N)
2  {
3      List<int> results = new List<int>();
4      for (int i=2; i<N; i++)
5      {
6           if (isPrime(i))
7           {
8               results.Add(i);
9           }
10     }
11     return results;
12 }
```

Line 8 is the interesting one. The `Add` method of a list puts the new item at the end of the current items. So the list grows in size. The example also demonstrates a value-returning method that returns a list.

15.5. Converting arrays to lists, and lists to arrays

If you have an array but need the additional flexibility of a list, you can pass the array into the list Constructor, or you can add an array of items to the end of an existing list:

```
1  string[] semester1 = {"Jan", "Feb", "Mar", "Apr", "May", "June"};
2  List<string> xs = new List<string>(semester1);
3  string[] semester2 = {"July", "Aug", "Sep", "Oct", "Nov", "Dec"};
4  xs.AddRange(semester2);
5
6  MessageBox.Show(string.Format("There are {0} strings in xs", xs.Count));
```

Lines 1 and 3 define and initialize two string arrays. In line 2 we construct a new list initialized with all the strings in `semester1`. In line 4, we dynamically add all the second semester's lines onto the end of `xs`.

Going the other way — from a list to an array, is just as easy:

```
1  string[] allTheMonths =  xs.ToArray();
```

15.6. Other list methods that we'll find useful

```
1  xs.Sort();
2  int i = xs.IndexOf("rotten");
3  xs.RemoveAt(i);
4  xs.Insert(3, "potato");
5  xs.Clear();
```

`Sort` will put the list into a sorted order.

`IndexOf` works on lists like it does on strings: we can find the index of the first occurrence of an element in the list. If it is not found, we'll get back -1. So this is a good way to test whether something is in a list.

And `RemoveAt` allows us to remove the item at a given position. The list shrinks, and all the items after i get shifted to the left.

At line 4, a new element is squeezed into the list at index position 3. All the other items have to move up, so the item originally at 3 now goes to position 4, and so on.

At line 5, the `Clear` method empties the list.

15.7. Cloning lists

Because a list is a reference type, the warnings we had earlier apply, When we assign a list to another list variable, or pass a list as an argument, we create an alias — another reference to the same underlying object. Aliases are tricky when the referenced object is modified.

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just an alias of its reference. This process is sometimes called **cloning**, to avoid the ambiguity of the word copy.

The easiest way to clone a list is to use the constructor pattern we used for arrays:

```
1  List<string> one =  ....;
2  ...
3  List<string> two = new List<string>(one);
4  two.Sort();
```

Now the sort method on list `two` won't rearrange the order of the elements in `one`.

15.8. Don't fiddle with a collection (list) that is being enumerated

Enumeration of a list (e.g. using `foreach`) "locks" the list against changes to its size or contents. So we cannot add or delete elements to a collection, nor can we modify any of the elements in the collection while the `foreach` construct is busy working its way through the collection:

```
1  foreach (int d in daysInMonths)
2  {
3      if (...)
4      {    // Add a 13th month to our year, with 25 days.
5           daysInMonths.Add(25); // Will give an error
6      }
7  }
```

We can, however, use a `for` or `while` loop with our own loop indexes, and we can delete or add elements at specific positions in the list.

But you need to be careful: we saw above that if you delete or insert a new element at, say, index position 5, all the other elements previously at positions 6, 7, 8 etc., move up or move down and so they change their index positions. This can make your loops tricky.

For this reason it is often easier to make the loop run backwards so we don't have to work out how the indexes to the left of where we are working will change as we add or delete items.

15.9. Is the original list (or array) mutated, or do we want a new list?

Sometimes we'll need to create a new list, while leaving the original list (or array) unchanged. Sometimes we'll be asked to mutate, or make the changes to the original. If the changes are going to be made to the original list, we'll say that the change is an **in-place update**.

So we should always be clear on which situation we're dealing with.

In this book we sometimes express our requirement as a test case. let's consider an example, starting from some unit tests. We want two methods that work with `List<int>`. The one method should take a list and return a new list in which all the elements have been doubled:

```
1  List<int> xs = new List<int>() { 3, 5, 4, 7, 2 };
2  Tester.TestEq(doubleAll_1(xs), new List<int>() { 6, 10, 8, 14, 4 });
3  Tester.TestEq(xs, new List<int>() { 3, 5, 4, 7, 2 });
```

Looking at the test case on line 2 we can see that `doubleAll_1` should be a value-returning method. The test at line 3 checks that the `xs` that we passed into `doubleAll_1` still has the original values that we gave it.

So a method like this passes the tests:

```
1  private List<int> doubleAll_1(List<int> xs)
2  {
3      List<int> result = new List<int>();
4      foreach (int x in xs)
5      {
6          result.Add(2*x);
7      }
8      return result;
9  }
```

Now let's solve an in-place version of the problem that passes this test:

```
1  List<int> ys = new List<int>() { 3, 5, 4, 7, 2 };
2  doubleAll_2(ys);
3  Tester.TestEq(ys, new List<int>() { 6, 10, 8, 14, 4 });
```

The method now should be a void method. And the test at line 3 checks that the list that was passed as an argument to our method has indeed been mutated.

```
1  private void doubleAll_2(List<int> xs)
2  {
3      for (int i=0; i < xs.Count; i++)
4      {
5          xs[i] *= 2;
6      }
7  }
```

Notice that we did not create a new list — we changed the elements that were in `xs`.

15.10. When should we use a list, when should we use an array?

Some say that the era of the fixed-size array is dead: we should always prefer a list.

But the historical role of arrays keeps them alive. For example, `string.Split()` and `File.ReadAllLines(...)` (which we'll cover soon) return arrays of strings, not a lists. This is because methods like this were part of the earliest versions of C#, before lists were introduced. And by time generic lists were introduced, it was too much trouble to change things. (Programmers get resentful when their code breaks because older features are changed or removed in newer versions of the language, so *backward compatibility* — the older code must run on the newer versions — is important.)

Definitely use a list if you need a structure in which the number of elements can change while your program runs. Or if you need the more powerful methods that a list provides, then the list should be your choice.

In other situations either a list or an array should work equally well.

15.11. Glossary

alias

Multiple variables that contain references to the same object.

clone

To create a new object that has the same value as an existing object. Copying a reference creates an alias but doesn't clone the object.

collection

A collection of elements is stored in a single structure. So far we've seen arrays and lists. Both are collections.

dynamic data structure

A way to organize data that changes it shape or size over time. A List in C# is a dynamic data structure that can expand or contract as we add or delete items. Arrays, by contrast, are fixed size.

enumerate

To visit each element of a collection in turn. The `foreach` loop allows us to enumerate a collection.

element

One of the values in a list (or other sequence, like an array). The bracket operator selects elements of a list. Also called an *item*.

generic

General, or able to work with many different types. The type `List<T>` means we can have a list of any type T — we can substitute any specific type for T.

index

An integer value that indicates the position of an item in a list. Indexes start from 0.

in-place

(or in-place update) A change that is done to the original list (or collection). The opposite idea is that we create a new collection while leaving the original unchanged.

item

See *element*.

list

A collection of values, each in a specific position within the list.

list traversal

The sequential accessing of each element in a list.

sequence

Any of the data types that consist of an ordered collection of elements, with each element identified by an index.

type parameter

A type parameter is a place-holder for an actual type in a *generic* type definition. In C#, the type parameters occur in angle brackets, e.g. `List<T>`. When we specialize a generic type we have to provide an actual type (e.g. `int`, `string`, `Turtle`), in our definition, e.g. `List<string> myFriends;`

15.12. Exercises

1. Consider this fragment of code:

```
List<int> xs = new List<int>();
List<int> ys = xs;
ys.Add(42);
```

   Does this create one or two list instances? Would would the value of `xs.Count` be after executing this code?

2. What will be the output of the following program?

```
string[] us = { "I", "am", "not", "a", "crook" };
string[] vs = { "I", "am", "not", "a", "crook" };
Console.WriteLine("Test 1: {0}",  us == vs);
us = vs;
Console.WriteLine("Test 2: {0}", us == vs);
```

   Provide a *detailed* explanation of the results.

3. The `us == vs` expression doesn't look "into" the list or an array when it makes its comparison: it simply asks "are these two references referring to the same object in memory?". We call this kind of test a *shallow equality* test. By contrast, a *deep equality* test asks "do the lists contain the same items in the same order?".

   a. Write a deep equality test for two arrays of string so that these unit tests pass:

```
string[] us = { "I", "am", "not", "a", "crook" };
string[] vs = { "I", "am", "not", "a", "crook" };
string[] ws = { "I", "am", "a", "crook" };
string[] xs = { "I", "am", "not", "a", "cowboy" };
Tester.TestEq(myEquals(us, vs),  true);
Tester.TestEq(myEquals(us, ws),  false);
Tester.TestEq(myEquals(us, xs),  false);
Tester.TestEq(myEquals(xs, xs),  true);
```

   b. Now do the same for a deep equality test for List<string>.

4. Write two methods that remove all the odd numbers from a list. The first method should build a new list containing only the even elements. The second method should do an in-place change to the original list.

```
List<int> xs = new List<int>() { 3,5,4,7,2 };
Tester.TestEq(removeOdds_1(xs), new List<int>() {4, 2});
Tester.TestEq(removeOdds_1(new List<int>() {}), new List<int>() {});
Tester.TestEq(xs, new List<int>() { 3,5,4,7,2 });

removeOdds_2(xs);
Tester.TestEq(xs, new List<int>() {4, 2});

List<int> ys = new List<int>() { 3, 5, 7, 9, 11, 13, 15};
removeOdds_2(ys);
Tester.TestEq(ys, new List<int>() { });
```

5. Write a method `moveToBack(xs, p)`. The p'th element of the list should "lose its place" and go to the back of the list. If p is out of bounds, no changes are made. This should be an in-place update.

```
List<int> xs = new List<int>() { 30,50,40,70,20 };
moveToBack(xs, 2);         // move element at position 2 to the back
Tester.TestEq(xs, new List<int>() { 30,50,70,20,40 });
moveToBack(xs, 0);
moveToBack(xs, -1);
moveToBack(xs, 4);
moveToBack(xs, 5);
moveToBack(xs, 2);
Tester.TestEq(xs, new List<int>(){ 50,70,40,30,20});
```

6. Re-do the above exercise, this time with fixed-size arrays. You may not use a list for the logic, nor are you allowed to attempt to resize the array.
7. Write a method that deletes any items in a `List<int>` that are smaller than their immediate predecessor in the original list. The list should be mutated: do not build a new list of items. Study the tests carefully to make sure you understand the requirements.

   You should try this problem in two ways and compare the code you get. In the first case, work backwards, starting at the last element of the list, and deciding if it needs to be deleted. Then work towards the front of the list.

   In the second variation, use a while loop to start at the left and work to the end. This is more difficult!

```
List<int> xs = new List<int>() { 12, 16, 14, 14, 16, 18, 11, 9, 12, 4, 2 };
deleteSmallerSuccessors(xs);
Tester.TestEq(xs, new List<int>() { 12, 16, 14, 16, 18, 12 });
deleteSmallerSuccessors(xs);
Tester.TestEq(xs, new List<int>() { 12, 16, 16, 18 });
```

16. More Event Handling

So far, events are generated from controls like buttons and sliders when we interact with them. We have also seen how we can attach a handler to an event to cause our program to respond in some way. Here we'll investigate three other really useful sources of events: the keyboard, the mouse, and timers. These will allow us to do more interactive things like we often find in games.

16.1. Timers

One or more *timers* can be added to any application. A timer generates regular *tick* events to which we can attach handler code. So perhaps we can check for new email arriving once every five minutes. Every timer has an `Interval` property that can be set to control the time between successive tick events.

These timers are not brilliantly accurate or fine-grained

Our computers multi-task: they can run more than one program simultaneously and are always doing some background work. So when we ask for 20 ticks per second we'll get something close to that, but they won't necessarily all be exactly evenly spaced. It will depend on what else is going on in the background. So these timers are not the best mechanism for building a metronome, or a heart pacemaker, or for high-speed games that require very smooth graphics. They also cannot go very fast: the maximum rate is typically limited to about 60 Hertz (ticks per second). There are other ways of doing more accurate and faster timing in C#, but they're beyond the scope of these notes.

We'll use these timers anyway for game-like situations, even if our on-screen graphics sometimes looks a little jerky. There are other C# gaming frameworks (MonoGame, XNA) which are more appropriate for for high-speed, smooth graphics.

See also the tip at the end of this chapter.

Let's begin with a simple game called Pong: our goal is to get a ball to move around, and to bounce off the edges of a container.

- We create a new project of type WPF Application.
- We put a green Canvas into the Grid that already exists in our main window (the canvas is our container for the ball).
- We set its properties so that when the window resizes the canvas will also change size.
- We add pictures to our project. In this case we just want a ball [1].
- On the canvas we place an `Image` Control (a control which can display a picture) which we name `ball`.
- We edit its Source property and choose to display the picture of the ball .
- We set the image control width and height to fit the ball snugly. (If you use this beach ball by right-clicking and saving the image, it's size is 34x34 pixels.)

Visual Studio Tip: To add images to your project ...

[1] The easiest way to add pictures to our project is to locate them on our hard drive, and drag and drop the pictures onto our project name in the Solution Explorer. They'll get copied into the project, (so we can delete the original files but they'd still be in our project), and we'll see them appear as part of our project in the Solution Explorer.

Now we need a timer object. Like our turtles, we'll need to define a variable for the timer, create (instantiate) an instance, and set its properties (and its handler) by writing your own code. Dive into the code behind

your window, and add a definition for a timer, create the timer, set some properties, and bind it to a handler by copying this code:

```
1  public partial class MainWindow : Window
2  {
3      System.Windows.Threading.DispatcherTimer theTimer;
4
5      public MainWindow()
6      {
7          // Initializes the window and all the components defined in the XAML.
8          InitializeComponent();
9
10         // If you initialize your own objects like a turtle or a timer, always
11         // make sure you do it after the call to InitializeComponent();   That way
12         // you'll be sure that the window exists, the playground exists, etc.
13
14         theTimer = new System.Windows.Threading.DispatcherTimer();
15         theTimer.Interval = TimeSpan.FromMilliseconds(100);
16         theTimer.IsEnabled = true;
17         theTimer.Tick += dispatcherTimer_Tick;
18      }
19
20     private void dispatcherTimer_Tick(object sender, EventArgs e)
21     {
22         updateBall();
23     }
24
25     private void updateBall()
26     {
27         // To be written...
28     }
29 }
```

Line 3 defines a class-level variable that can reference our timer. (Because it is class-level, its lifetime will only end when the window closes.) Line 14 creates the timer, and its properties are set in line 15 and 16. In this case, we enable the timer, so it can generate its tick events, and the Interval property controls their frequency. The TimeSpan specifies 100 milliseconds between each tick, so we should (in theory) get 10 events per second, and each will call `updateBall`.

Finally, at line 17 we connect an event handler to the timer's Tick event, and lines 20–23 implement the handler for that event.

That's quite a lot of manual work, but the timer gives us a really fun source of events...

The position of the ball on the canvas is controlled via the canvas rather than by properties of the ball. The method `Canvas.GetLeft(ball)` tells us where the top left-hand corner of the ball is, relative to the left edge of the canvas. `Canvas.SetLeft(ball, x)` will change the ball's position on the canvas [2]. Similarly, there are methods to control the distance of the ball from the top of the canvas. So lets change `updateBall` and put the ball in motion:

```
1      double velX = 4, velY = 2;
2
3      private void updateBall()
4      {
5          double nextX = Canvas.GetLeft(ball) + velX;
6          Canvas.SetLeft(ball, nextX);
7
8          double nextY = Canvas.GetTop(ball) + velY;
9          Canvas.SetTop(ball, nextY);
10     }
```

[2] For the super-observant: `GetLeft` and `SetLeft` methods are what we call *class methods* — they belong

to the type `Canvas`, rather than to `canvas1`, which is our specific canvas instance.

We've made two class-level variables for the velocity in both the x and y directions. Each time the timer ticks we handle the event by computing and setting a new position for the ball, based on its old position.

When we run this the ball drifts towards the lower right of the window, and eventually disappears from sight. We can experiment with making the timer tick more rapidly, or changing the values for the velocities.

How does one bounce the ball off the edges? Let's just think about the X movement initially. (A nice simplification is to comment out lines 8-9 while we think about movement in the X direction.)

If we negate `velX`, the ball will move to the left. If we negate it again, it will become positive and move to the right again. So bouncing the ball in the X direction is as simple as changing the sign of `velX`. But we have to make the changes at the right time.

When should we do this? On the left side it is easy: if the `nextX` that we've calculated in line 5 becomes negative, it is time to change direction.

On the right side it is a bit more tricky: remember the position of a control is determined by its top left corner. Let's look at modified code now...

```
1  private void updateBall()
2  {
3      double nextX = Canvas.GetLeft(ball) + velX;
4      Canvas.SetLeft(ball, nextX);
5      if (nextX < 0 || nextX + ball.ActualWidth > canvas1.ActualWidth && velX > 0)
6      {
7          velX = -velX;         // Change direction
8      }
9      ...
10 }
```

- The ActualWidth property of the canvas or the ball is the one we're interested in working with. As you play, you can change the size of the window and the canvas size changes. ActualWidth tells what the width is at the moment.
- We want the ball to bounce off the right side when the ball's right edge overshoots the edge of the canvas. So in line 5 we had to add the ball's width onto the ball's position.
- Because the ball can be right near the right edge of a big canvas, and we can suddenly resize the window to be much narrower, it is possible that the ball may stay off the right edge for a while. So it is not good enough to simply code up "If the ball is off the right edge change direction" — the test has to be more sophisticated. So if we're off the right edge we also want to check if `velX > 0`, i.e. is the ball still moving further away.

The timer ticks drive the animation ...

The program organization here is often found in games with sprites (a sprite is something that moves, or animates, during a game — a bullet, a ball, a player, an explosion...).

Here the timer ticks periodically: as part of handling the tick, we update all the (potentially changing) elements of our game (we only have a ball in our game right now, but we'll add a paddle shortly ...)

We'll leave it as an exercise for the reader to get the ball bouncing nicely off the top and bottom edges of the canvas.

The `IsEnabled` property of a timer can be set true or false, to start or stop the timer. This will allow us to have some keyboard key or button on the GUI to let us pause or resume our game.

We can also change `theTimer.Interval` while the code is running. So we can speed up a game as it runs, or provide a slider control to determine how fast we want the game play to happen.

16.2. Keyboard events

When a keyboard key is pressed and released, we get two separate events — `KeyDown` and `KeyUp`. You can ignore both of course, or you can bind handlers to either, or to both, of these.

It is important to understand which program, and which control within the program receives events from the keyboard. Suppose we're working and we have our email open, and a browser, and a music player. Only one of our desktop applications has **input focus** (i.e. it's window is uppermost above the other windows and is the focus of the user's attention). It's window title bar is highlighted differently from other applications. It is the application to which key press events will be delivered. Open a few applications and switch between them to make sure you understand which one has focus at all times. (On Windows 7, one way to tell is that the *close* button at the top right of the window is highlighted (shown in red if you have a default colour scheme) for the application that has focus, while other title bars of other applications are greyed out.)

Not only does an application have input focus, but if the window contains a number of sub-controls like text boxes, buttons, and scrollbars, only one of these can have focus within the application: it gets the keyboard event first. There are subtle visual changes to help us understand which control has focus, as shown here:

In this case the button is the focussed control on the form — and the little visual cue we get is the extra active highlighting around the button. Keys pressed on the keyboard now are sent first to the button. (Buttons tend to ignore most keys, but one notable exception is the `Enter` key, which causes a click event to occur, as if the button had been clicked with the mouse!)

Fortunately Windows Presentation Foundation (WPF) has a nice mechanism: after the button gets the event, its container (the button is probably on a canvas) will also get the same event, then if the canvas is on a grid, the grid will get the event, and finally the main window will get the event. We say that the event "bubbles" up the hierarchy of container controls. This makes things easier for us: rather than work with events on individual controls and concern ourselves with which control has focus, we'll just attach our

handlers to the main window. (This works nicely for gaming type situations, it may not be best in other situations.)

Let's use a simple turtle example now: we want a turtle that will turn left or right when the user pushes the left or right arrow keys. This will allow us to use the keyboard to turn the turtle. Additionally, we'll provide a timer that ticks regularly and moves the turtle forward. So we'll create a handler for the main window's KeyDown event, and well also create a timer (as we did in the last section) to move our turtle automatically.

```
1   private void updateTess()
2   {
3        tess.Forward(10);    // On each timer tick, move tess forward automatically.
4   }
5
6   private void Window_KeyDown(object sender, KeyEventArgs e)
7   {
8       switch (e.Key)
9       {
10          case Key.Escape:
11              this.Close();   // Close the window, exit the game.
12              break;
13
14          case Key.Left:
15              tess.Left(15);
16              break;
17
18          case Key.Right:
19              tess.Right(15);
20              break;
21      }
22  }
```

Here are some points to note:

- We must attach the code-behind handler at line 6 to our main window's KeyDown event. (This step is usually done in the GUI: either by tweaking the XAML by hand, or by using the properties editor. C# won't automatically make the association simply because you've called the handler method `Window_KeyDown`.)
- The `switch` is great for deciding what to do with each of the different keys. We've handled three specific keys here, but it is very easy to add more cases to the switch statement.
- The `Key` type is known as an *enumerated type* — it simplifies our code and lets us match any of the keys on the keyboard, including function keys, arrow keys, backspace, etc.
- At lines 10–12, pressing the Escape key on the keyboard quits the program by closing the window (`this` refers to the window itself).
- When the event is delivered to our handler, the parameter `e` contains additional information that we can use. For example, we can find out whether any modifier keys (e.g. Alt, Shift or Ctrl) are pressed at the same time.

With this code in place (and the timer enabled, so that it ticks regularly, and its event handler calls `updateTess`), Tess will move forward on every tick, but we'll also be able to steer her from the keyboard.

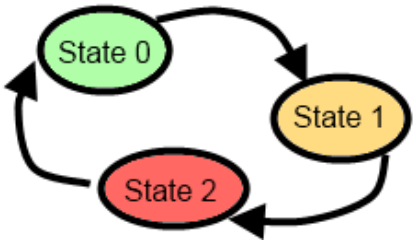
16.3. An example: finite state machines

A finite state machine is a system that can be in one of a few different *states*. (The finite bit of the definition says it has a finite number of states.) We draw a state diagram to represent the machine, where each state is drawn as a circle or an ellipse. Certain events occur which cause the system to leave one state and *transition* into a different state. These *state transitions* are usually drawn as an arrow on the diagram.

The controller is the part of a program that sends the events to the the rest of the program, or for this example, to our state machine. A keyboard, mouse, or timers could all act as a controller to get the state machine to switch to another state.

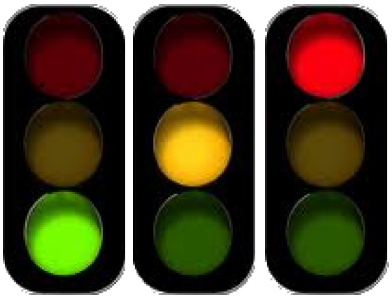
Some controllers depend on user-generated events: for example, when first turning on a cellphone, it goes into a state which we could call “Awaiting PIN”. When the correct PIN is entered, it transitions into a different state — say “Ready”. Then we could lock the phone, and it would enter a “Locked” state, and so on.

A common case where we can use a timer as our controller (to generate events and state changes) is for a traffic light. Here is a state diagram which shows that a traffic light state machine continually cycles through three different states, which we’ve numbered 0, 1 and 2.



The viewer parts of our program will determine how the current state of the state machine appears on our GUI: we’ll display the traffic lights by using an Image control in our window.

We prepare three different pictures for the three states of the state machine. (If you’re reading the book on-line you can right click on these pictures and save them from the web page, and get them into your application using the same technique as we did with the ball in the first part of this chapter — just drag and drop them onto the project name in your Solution Explorer window.)



```
    int currentState = 0;

    private void dispatcherTimer_Tick(object sender, EventArgs e)
    {
        stateControllerAdvance();
        updateView();
    }

    private void stateControllerAdvance()
    {
        switch (currentState)
        {
            case 0:
                currentState = 1;
                break;

            case 1:
                currentState = 2;
                break;

            case 2:
                currentState = 0;
```

```
23                  break;
24              }
25          }
26
27          private void updateView()
28          {
29              image1.Source = thePics[currentState];
30          }
```

We've organized the code for the state machine in its own method: when the timer ticks we call the method to advance the state machine to its next state. Once that has happened we'll call `updateView` to change the user's view of the state machine.

The pattern of code is typical for a state machine: the `switch` statement selects the appropriate case on the basis of the machine's current state, and the code within the selected block might do some actions, and then it sets the machine into its next state. For this simple example, the states are cyclic 0,1,2,0,1,2... so we could probably use simpler code (shown below), but the more general pattern of using a `switch` statement to select the current state will serve us well in the long run.

```
1  private void stateControllerAdvance()     // alternative version
2  {
3     currentState = (currentState + 1) % 3;
4  }
```

Line 29 changes what is displayed in the image. We still have to define and initialize the array of pictures, but assuming we can get that step done, we'll have completed our traffic lights exercise: they'll change state by themselves.

16.3.1. Working with resources

When we put together a program that uses pictures, sound files, fancy fonts, video clips, the text file containing the Alice in Wonderland book, etc. these additional bits and pieces are called the program's **resources**.

We can arrange to collect all our resources into a single folder somewhere on our disk. This is not too messy to program, but it has the disadvantage that if we give our program to a friend, we'll have to remember to copy the resources folder too, and our friend will have to put her resources at the same path on her disk so that our code can find them.

A better way to do this is to build the resources into the program — we call these **embedded resources**. It makes it a lot easier to deploy programs (put them on other machines) because all we have to copy is the program as it already contains all the resources it needs. Visual Studio gives us mechanisms for embedding resources — we mentioned earlier in this chapter that you can just drop the pictures onto the project name in the Solution Explorer.

```
private BitmapImage[] thePics;  // Define a reference to an array of images

    // In the constructor for the window, create the array of pictures
    // We also need a magic spell that tells C# to find the images "in this application" ...
    string inThisProject = "pack://application:,,,/";
    thePics = new BitmapImage[] {
                    new BitmapImage(new Uri(inThisProject + "TrafficLightGreen.png")),
                    new BitmapImage(new Uri(inThisProject + "TrafficLightAmber.png")),
                    new BitmapImage(new Uri(inThisProject + "TrafficLightRed.png")) };
```

This array definition has an initializer that loads three pictures from our embedded resources. The magic spell part of the URI (pack://application:,,,/) means "you'll find these images inside the current project".

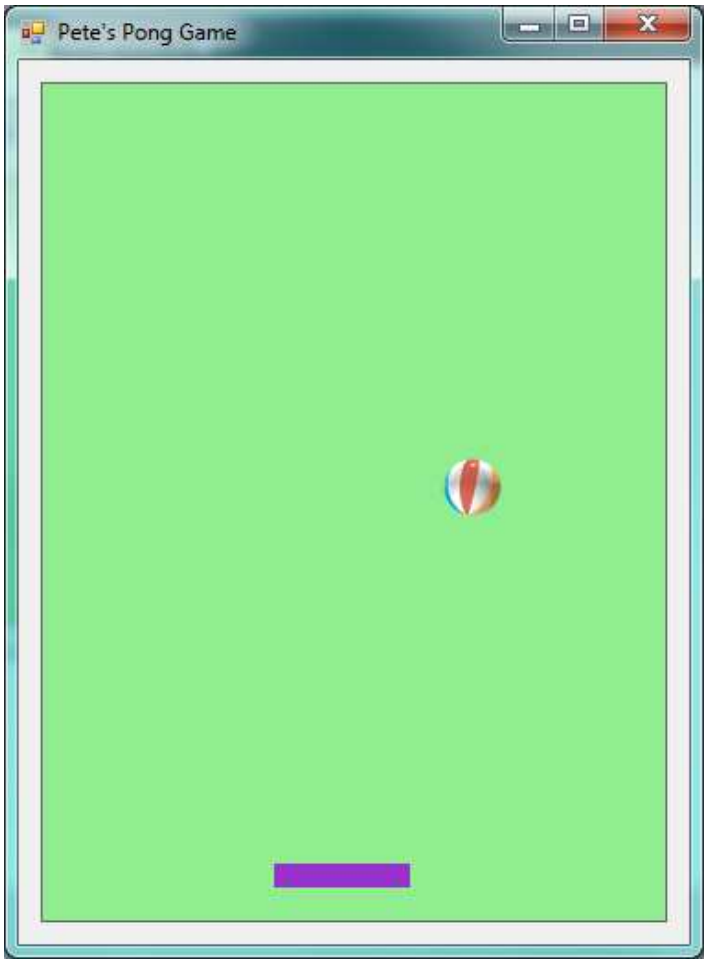
It is also possible to load pictures without embedding them first: this is useful if we want to open our own photographs. We obviously don't want to build our personal photo albums into our programs. So here we show how we can load a picture from a hard disk, or from the web.

```
BitmapImage greenPic = new BitmapImage(new Uri("c:\\temp\\trafficLightGreen.png"));
BitmapImage logo = new BitmapImage(new Uri("http://www.ict.ru.ac.za/Resources/thinkSharply/Think
```

So perhaps you can get three different pictures of yourself: jumping, laughing, hugging. Stick them somewhere on your hard disk, or drag them into your project. Instead of silly traffic lights you can be much more creative!

16.4. Adding a Paddle to Pong

Our Pong program from earlier in the chapter used a timer to drive some updates to the ball position. Now we'll add a paddle that can move left or right (while we hold down the left or right arrow keys). To represent the paddle in the game we'll just use a button (named `paddle`). If the paddle can only move horizontally, we can move it by using the `Canvas.SetLeft` method that we saw earlier. (If you want to be able to resize your window while playing, it would be nice to make the paddle stay near the bottom of the canvas. So `Canvas.SetTop` will be useful too!)



The game will be to prevent the ball from getting past the paddle. We'll move the paddle from left to right by holding down one of the left or right arrow keys.

The keyboard events we can get (`KeyDown` and `KeyUp`) don't allow us to easily respond to "key held down". So instead, we're going to build our own small finite state machine to track the desired state of the paddle.

Keyboard presses won't directly move the paddle: instead, they'll just trigger state changes in the finite state machine that controls our paddle.

The paddle's controller logic (it's state machine) can be in one of three states — moving left, moving right, or stopped.

Now, when our timer ticks, we'll update the position of the paddle. Depending on the state of the controlling state machine, we'll move it left, move it right, or leave it where it is:

```
1   double paddleSpeed = 4;
2   string paddleState = "stopped";
3
4   private void dispatcherTimer_Tick(object sender, EventArgs e)
5   {
6       updatePaddle();
7       updateBall();
8   }
9
10  private void updatePaddle()
11  {
12      switch (paddleState)    // See what state the paddle is in, and respond appropriately
13      {
14          case "stopped":
15              break;
16          case "movingLeft":
17              double nextX1 = Canvas.GetLeft(paddle) - paddleSpeed;
18              Canvas.SetLeft(paddle, nextX1);
19              break;
20          case "movingRight":
21              double nextX2 = Canvas.GetLeft(paddle) + paddleSpeed;
22              Canvas.SetLeft(paddle, nextX2);
23              break;
24      }
25  }
```

In line 2 we've chosen to represent the state of our paddle as a string. (We could have used an integer with values 0,1,2, etc.) On the timer tick we add code to also update the paddle, and either do nothing, or we move the paddle left or we move it right.

Now all we need to do is get the events from the keyboard, and update the state machine.

```
1   private void Window_KeyDown(object sender, KeyEventArgs e)
2   {
3       switch (e.Key)
4       {
5           case Key.Left:
6               paddleState = "movingLeft";
7               break;
8           case Key.Right:
9               paddleState = "movingRight";
10              break;
11          default:
12              paddleState = "stopped";
13              break;
14      }
15  }
16
17  private void Window_KeyUp(object sender, KeyEventArgs e)
18  {
19      paddleState = "stopped";
20  }
```

Note the design here.

Now the paddle can be moved by holding down the arrow keys.

There are a number of enhancements you can make now. The paddle should not be able to move off the edges of its container. And, of course, you need some *collision detection*. The ball should no longer bounce off the bottom: the game should end if the player misses the ball. But it should bounce off the paddle. And it would be nice to add some text for a score: perhaps you score 10 points each time you hit the ball with the paddle. And, if you're into this kind of thing, you can speed up the timer ticks or the velocity of the ball and the paddle as the score gets higher, to make the game more challenging. So there is a lot of experimentation, plenty of learning opportunity, and some fun to be had with a simple game like this!

Our state machine is acting as a kind of middle-man between the keyboard events, and the timer events which move the paddle, and update its view.

So key presses do not directly move the paddle: they set a finite state machine into some state which indicates what needs to be done. The handler code for the timer tick then looks to see what state the machine is in, and uses that information to decide how to move the paddle.

This is a really useful separation, easy to understand, and easy to code up.

Hey, what if the timer tick and the keypresses events occur at exactly the same time?

This will not happen. All events arriving at your window are put into a special queue, and (unless you do some amazing things) your window will not start handling a new event until it has finished handling the current one. So one of the two will win the race to get to the queue first, and only when it has finished executing, can the other event handler begin.

16.5. Mouse events

When the mouse moves over any control, that control receives events generated by the mouse. The most popular events are clicks, double-clicks, and mouse movement. Like the keypress events, they will also bubble up to their container control, and its container, until the finally reach the topmost window for the application. So we'll go to our main window and attach a handler for whenever the mouse moves:

```
1  private void Window_MouseMove(object sender, MouseEventArgs e)
2  {
3      Point pos = e.GetPosition(canvas1);
4      this.Title = string.Format("Mouse at {0}", pos);
5  }
```

- If you move the mouse over your window, you'll find the window's title bar changes very rapidly as the events fire.
- Line 3 shows how to retrieve the position of the mouse relative to canvas1.
- Line 4 just turns this position into a string and puts it into the title bar.
- Experiment with what happens when the mouse goes outside the canvas, and outside the window.

Now we can just use the mouse as a controller for the paddle, like this:

```
1  private void Window_MouseMove(object sender, MouseEventArgs e)
2  {
3      Point mousePos = e.GetPosition(canvas1);
4      Canvas.SetLeft(paddle, mousePos.X);
5  }
```

You may also like to try setting the `Cursor` property of the main window to `None`. Then the Cursor will disappear when you're over the window. But the MouseMove events will still fire, so you'll be able to control the paddle by using the mouse.

16.6. Using the mouse to draw

As our turtles move, they draw lines behind them. So assuming we have a turtle set up, here is a simple two-line handler:

```
1  private void Window_MouseMove(object sender, MouseEventArgs e)
2  {
3      Point mousePos = e.GetPosition(canvas1);
4      tess.Position = mousePos;
5  }
```

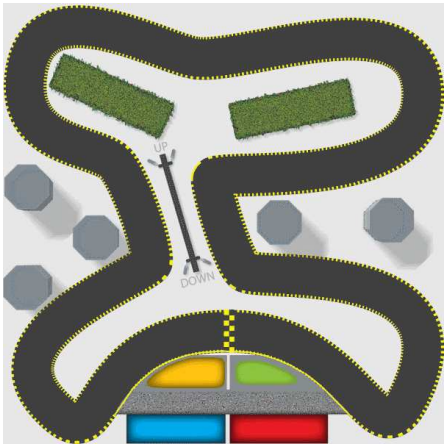
Now moving the mouse moves the turtle to the mouse position. If the turtle brush is down, it will draw!

Now a clever idea is to use the state of the mouse button (up or down) to set whether the turtle brush is down. Then we could draw only while we hold down the button. One extra line of code before line 4 can set the brush up or down for us:

```
1      tess.BrushDown = e.LeftButton == MouseButtonState.Pressed;
```

16.7. Racing turtles

Okay, we can control Tess using key presses. And we can update Tess automatically when a timer ticks. The update can move her forward. Can we make the turtle playground into a race-track, and race Tess around our race-track?



It is easy to get the playground to look like a race-track: the playground is a `Canvas` control, so we can set any background, including a picture. So we'll set the canvas background to this picture (right-click and save it from the web page for your own version of the program...)

If we get the picture into our project as one of its resources (like we did earlier in this chapter), and open the picture with Visual Studio, we'll find its size is 600x600, and we'll also find that (287, 444) are great starting coordinates for positioning our turtle at the start/finish line on the track.

Now we're ready to race. We can start (and perhaps stop) the timer using the space bar. And we can steer the turtle. So we have a racing game just using the bits of code that we've already covered in this chapter.

Now for some fun improvement: if we go off-road, Tess should move really slowly, but when she's on the highway, we'd like to keep her moving fast. So our next abstraction is to assume that we have a way to test if she is on or off the road. We'll code the update logic like this:

```
1  private void updateTess()
2  {
3      if (onTheRoad(tess))
4
```

```
    {
        tess.Forward(tessSpeed);
    }
    else   // off-road, go 5 times slower
    {
        tess.Forward(tessSpeed / 5.0);
    }
    // Or we can write it all in one line like this:
    //    tess.Forward(onTheRoad(tess) ? tessSpeed : tessSpeed / 5.0);
}
```

Of course we still need to write the test about whether a turtle is on the road. But the approach of *"let's assume we can solve the sub-problem"* is a really powerful way to make progress. It allows us to break the whole problem into simpler sub-problems and focus on just one thing at a time.

If you look again at the image we're using you'll see that it has been quite carefully doctored so that the road is darker than all other regions of the image. So testing if we're on the road can be coded like this:

```
private bool onTheRoad(Turtle t)
{
    Color c = t.ColorUnderTurtle;
    return (c.G < 66);    // Inspect the green channel to tell if Tess is on the road.
}
```

Notice that we didn't work with turtle Tess. We used a parameter `t` instead. Perhaps this will allow us to also put Bert and Alex on the race track too, and turn this into a proper race.

How did I come up with that magic constant 66 in line 4 of `onTheRoad`?

By experiment and clever guessing. I used the trick from the previous section of making Tess follow the mouse. Each time I got a MouseMove event on the main window, I moved Tess to the new position, and then got the colour under the turtle. By displaying this (I used the Title bar of the main window for a quick and dirty experiment), I could scan the colour values rapidly by just moving the mouse around. From there it was easy to guess that 66 would be a good threshold for the test.

Putting these bits of code together gives a fun game: a turtle that moves forward at different speeds depending on whether it is on the road or off the road. And we can steer it with keyboard keys. We'll leave it as an extension now to add some logic for a second turtle and a second player to race against you.

16.8. Sounds

We can get our games to play sounds. Download some suitable "wav" files from the Internet, or use bounce.wav. (We can't play mp3s, but free tools like Audacity can convert mp3 files into wav files. The wav files can get very big, so exercise some caution!)

In the same way that we embed images in our project, we can embed sound files (or any other kind of file) by dragging and dropping the files onto our project name in the Solution explorer. Then right-click on the new files in your Solution Explorer, choose the Properties, and ensure that the **Build Action** property is set to **Resource** so that Visual Studio embeds (copies) the file as a resource into the executable file.

We'll need some `using` directives at the top,

```
using System.Media;
using System.Windows.Resources;
```

and new the code is straightforward:

```
private void makeBounceSound()
{
    Uri pathToFile = new Uri("pack://application:,,,/bounce.wav");
    StreamResourceInfo strm = Application.GetResourceStream(pathToFile);
    SoundPlayer sp = new SoundPlayer(strm.Stream);
    sp.Play();
}
```

We can now put a call to this method into our Pong game whenever we hit the wall or paddle and change the direction of the ball.

Alternatively, we could also put some nice music to loop in the background while our game plays. Here we don't embed the resource, we simply play the wav file directly from our hard drive.

```
SoundPlayer sp = new SoundPlayer("C:\\temp\\music.wav");
sp.PlayLooping();
```

The sound player will only play one sound file at any time, so we can't have background music and bounce sounds at the same time: the bounce sound will kill our background music.

16.9. Glossary

controller

Some device or mechanism like a keyboard, mouse, touch screen, timer, or game-pad that lets us interact with objects in our program.

embedded resource

A self-contained resource that is part of the executable program that you ship to your customer.

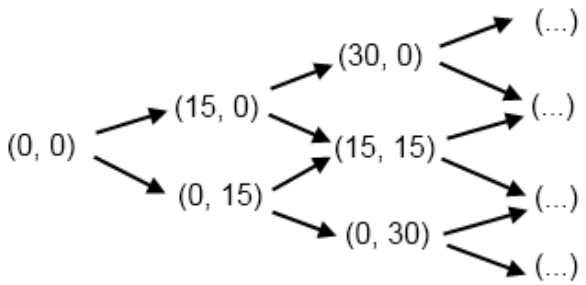
resource

Something that your program uses when it runs — sounds, pictures, video clips, background scenery, text passages, fonts, etc.

16.10. Exercises

1. In the bouncing ball example we reasoned that it was not good enough to change the ball direction simply because the ball was beyond the right edge of the canvas. Remove the extra test `velX > 0` from that logic, and show that you can make the program go wrong.
2. Your traffic light program has been patented, and you're about to become seriously rich. But your new client needs a change. They want different times spent in each state. The machine should spend 3 seconds in the Green state, followed by one second in the Orange state, and then 2 seconds in the Red state. Change the logic.
3. Write one of those irritatingly stupid programs that has a button labelled "Click Me!". When the cursor moves over the button, make the button immediately jump to some other random location so that it can't be clicked by the user. (Hint: look at what events you have available on the button.)
4. If you don't know how tennis scoring works, ask a friend or consult Wikipedia. A single game in tennis between player A and player B always has a score. We want to think about the "state of the score" as a state machine. The game starts in state (0, 0), meaning neither player has any score yet. We'll assume the first element in this pair is the score for player A. If player A wins the first point, the score

becomes (15, 0). If B wins the first point, the state becomes (0, 15). Below are the first few states and transitions for a state diagram. In this diagram, each state has two possible transitions (A wins the next point, or B does), and the uppermost arrow is always the transition that happens when A wins the point. Complete the diagram, showing all transitions and all states. (Hint: there are twenty states, if you include the deuce state, the advantage states, and the "A wins" and "B wins" states in your diagram.)



5. Change the Pong program into a two-player game where you can play against a friend. Make the paddles move vertically, one player on the left and one with a paddle on the right. Each player uses their own keyboard keys to control their paddle. Whoever misses the ball loses the point.

   A few seconds after a player loses a point the ball is "served" from the paddle of the winner of the previous point. The game should play for a total of 11 points, and should show the players' scores as they play.

6. If you've done the racing turtle game you'll find your turtles slow down when they hit the yellow grid on the start/finish line, or the yellow paint on the borders of the racetrack. But that just seems like a bug. Fix it.

7. Turn the turtle racing program into a two-player game. Also provide keyboard keys to increase or decrease the turtle speed, so that players can accelerate on the straight, but apply brakes for the corners.

8. Make an autonomous racing turtle (a bot) that steers itself around the track.

9. Design a better race track.

(Advanced) Next Steps for Better Timing in our Games

The slight irregularity of the timer tick events in these games is sometimes noticeable when we're moving sprites like the ball and the paddle. This is because we move the sprites a fixed distance on each tick, but the ticks are not precisely evenly spaced.

We can compensate for that if we can get an accurate measure of the time since we got our last tick event. The `Systems.Diagnostics.Stopwatch` gives us a high-resolution way to accurately measure intervals. Once we know the time since we last moved the ball, we can scale the distance we move the ball on this tick. This should give our users a much smoother experience, and our game will appear to run at the same speed whether we are running on very fast or quite slow hardware.

Also, if we don't set any interval at all for a Timer, the timer ticks occur "as fast as the system can manage them" - much faster than the typical 60 events per second. This may be useful in some situations.

17. Odds and Ends

In this chapter we cover a couple of useful things about C# that we omitted for simplicity in earlier sections of the book.

17.1. Escape sequences in strings

Some characters (like a newline) are not printable (or directly typeable on the keyboard), but we'd like a way to make them part of our strings. So here is the workaround.

Executing this fragment of code

```
1  string s = "Hello, world.\nThis is fun.\nLots of fun!";
2  MessageBox.Show(s);
```

gives us this:

When we code up an "ordinary" literal string,

- The string may not extend over more than one line of code in our program.
- We can put escape sequences with special meaning into the strings. In this case the `\n` escape sequence stands for just one character, called "newline". The backslash character is called the escape character, and means "I'm not really a backslash, but here comes a something special". There are a few other escape sequences used sometimes. But the important thing is that when we see a backslash in a string, it doesn't mean backslash!

So what do we do if we need a real backslash in a string, as in `C:\temp\myfile.txt`? We have to escape our backslash with its own backslash — so we code up two backslashes to end up with just one in our string. So the usual way to code a file path like the above in C# is `"C:\\temp\\myfile.txt"`. Note that the length of this string is 18, not 20! Each escaped sequence `\n` or `\\` is just a one character in the string. For a list of common character escapes see https://msdn.microsoft.com/en-us/library/4edbef7e(v=vs.110).aspx

17.2. Verbatim literal strings

The escape mechanism described above was introduced in C, the grandparent language of C#, so we still use it. But over time we thought "Wouldn't it be better to sometimes be able to turn off the string escape mechanism?"

So there is another newer way to code up literal strings: if the string starts with @" it is called a **verbatim string**. Two things happen: none of the characters are treated as escapes, and the string can flow over more than a single line of source code. Any newline characters in the source string become newline characters in the string, and any single backslash characters are backslashes!

So here is a simple fragment of code:

```
80              string poem =
81  @"    The truth I do not stretch or shove
82  When I state that the dog is full of love.
83        I've also found, by actual test,
84           A wet dog is the lovingest.
85
86                                      Ogden Nash";
87
88              MessageBox.Show(poem, "The Dog");
```

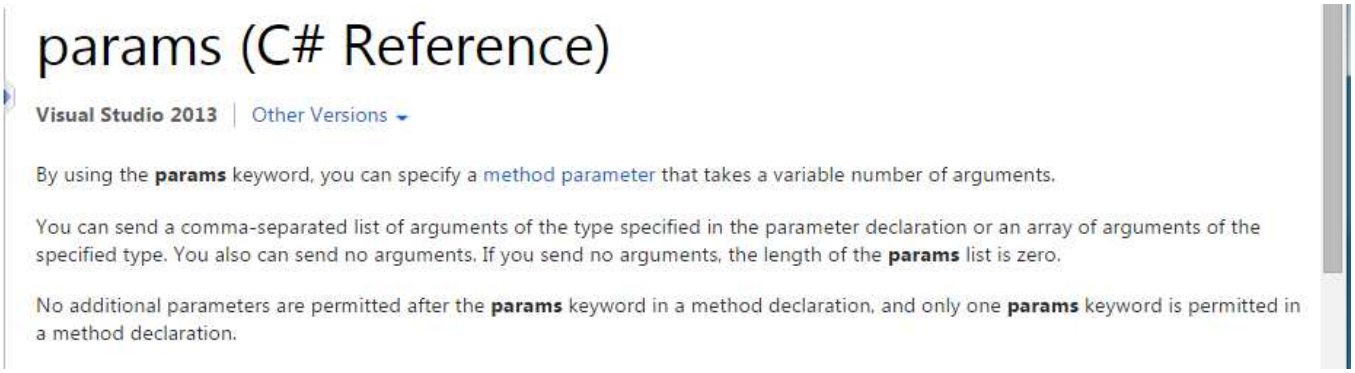
which pops up this message box:

You'll quite often find verbatim strings used with file paths: `@"C:\temp\myfile.txt"`.

With hindsight, it now seems a bit weird to have a special-case escape mechanism and then a second layer of special syntax to turn off the first lot of special cases, but that is what history has left us with! And even as C# keeps evolving, in C# 6.0 they are introducing even more mechanisms into strings! http://www.codeproject.com/Articles/846566/What-s-new-in-Csharp-String-Interpolation

17.3. The `params` keyword

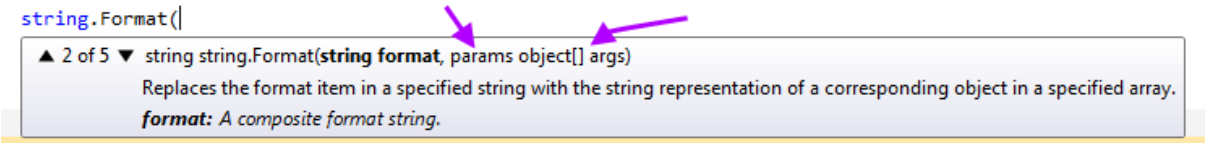
Let's use Microsoft's explanation for this one:



When is this used? One example we've seen is the `string.Format` method, where we can pass in a formatting string and a variable number of arguments that should be plugged into the string, as shown in this fragment:

```
1  int yearOfBirth = 1995;
2  int yearNow = DateTime.Now.Year;
3  int age = yearNow - yearOfBirth;
4  string s1 = string.Format("{0}, born in {1}, turns {2} years old in {3}.",
5        "Joe", yearOfBirth, age, yearNow);
6  ...
```

As we're typing this in, Visual Studio pops up some *intellisense* help:



But wait! The big deal here, different from what we've seen before, is that this `string.Format` signature only defines *two* parameters, but our call site in line 4 above supplies *five* arguments. How do we pass 5 arguments to just 2 parameters? The first argument is assigned to the first parameter, and because of the `params` keyword, all the remaining arguments are bundled up into an array, and (a reference to) the array is assigned to the second parameter. The IntelliSense above shows the signature of the `Format` method, with some highlighted cues that we should learn to notice:

We can write our own methods that use the `params` keyword. See the Microsoft reference at http://msdn.microsoft.com/en-us/library/w5zay9db.aspx if you want an example of how to do that.

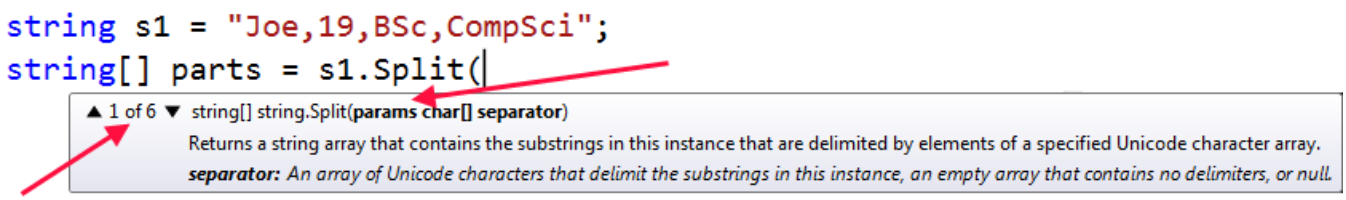
17.4. `string.Split` revisited

The `Split` method on strings cuts (a copy of) a string into pieces. The cuts are made on the delimiters we pass to the method. For example,

```
1  string s1 = "Joe,19,BSc,CompSci";
2  string[] parts = s1.Split(',');
```

will chop `s1` into parts by removing each comma. So we'll end up with an array of four strings in `parts`. (Notice that `s1` still has its original value: recall that strings are read-only and can never by modified.)

When we're typing the above code into Visual Studio we get some interesting IntelliSense:



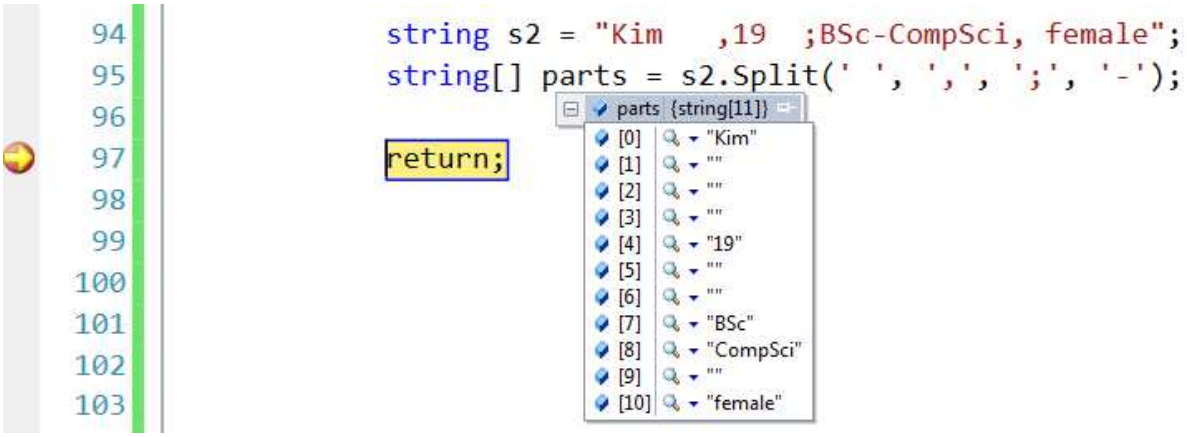
Notice two important things:

- There are six different overloadings of this method, and we're currently looking at the hint for the first one. Clicking the triangles allows us to see all the other overloadings. (Or pressing F1 for help while your cursor is over the word `Split` should invoke help, where you can read about the different overloadings.)
- The first overloading has a `params` keyword. We can pass in any number of delimiters (even zero).

So the semantics (meaning) of `Split` is quite tricky:

- If you pass no arguments at all to `Split`, e.g. `string[] xs = poem.Split();` the delimiters are taken to be any white space characters (spaces, newlines, tabs, etc.)
- If you pass just a space, it becomes the only delimiter. Other white space characters like tab or newline will not be used to separate parts of the string.
- If you pass in multiple delimiters, any one of them causes a cut whenever it is found.

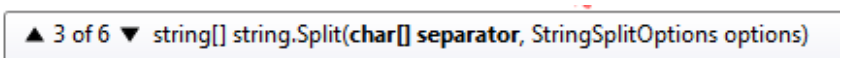
Let's consider this example:



At line 95 we split the string on any of four delimiters — spaces; commas; semicolons; or hyphens. We've put a breakpoint at line 97, and are inspecting the `parts` array to see what we got. Notice that none of the delimiters occur in any of the parts, and the string has been split where we asked for splits.

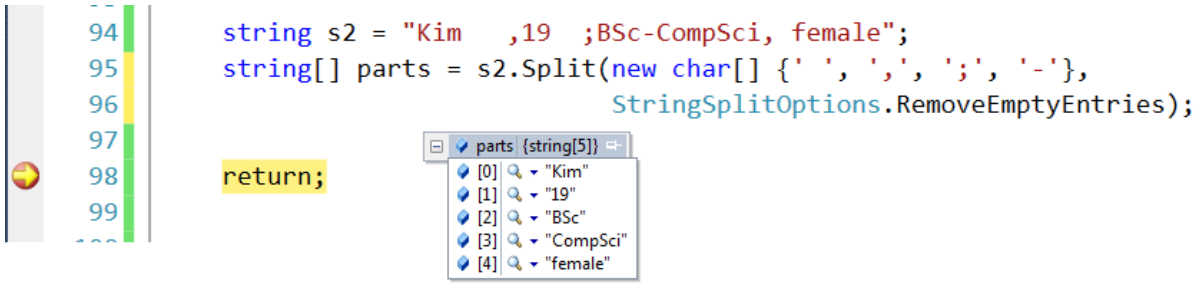
But we also notice that there are some empty strings in the `parts` array: whenever two of our delimiters occur next to one another or at the end/start of the string, the section between them becomes an empty string. So the multiple spaces, and the space next to a comma or a semicolon all generate extra (perhaps unwanted) empty strings.

Let's fix this common problem. Overloading 3 of the `Split` method allows us to provide some extra options to control how `Split` works — in particular, it allows us to tell `Split` to remove all empty entries from the result. The signature is



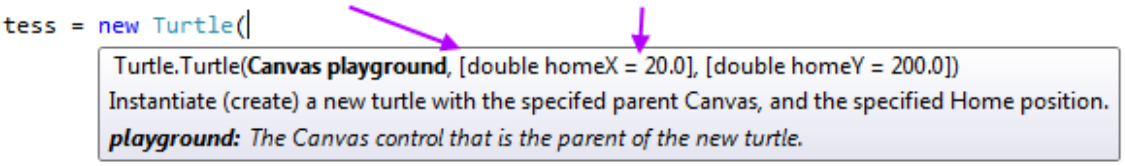
It is a bit trickier to use this version of `Split` because it wants us to pass in an array of delimiters (separators) now. That array can even be the value `null` — in which case `Split` assumes we want to split on white space.

So here is the new code, with more satisfying results — no empty strings in the resulting array:



17.5. Optional Parameters

We've created quite a few turtles so far: have you ever noticed this?



What the square bracket notation means in this IntelliSense pop-up is that `homeX` and `homeY` are **optional parameters** — at the call site we may optionally provide arguments to pass into the parameters, but if we don't tell the turtle what its home position is, it will use some **default values** instead. In this case the default `homeX` value is 20.0, and the default `homeY` value is 200.0.

Note that the square brackets used here are not part of the C# syntax — they are what we call *meta-notation*. They they tell us that supplying these arguments is optional.

So in our first chapter about turtles we saw both ways of using this:

```
...
// Use the default home position for tess
tess = new Turtle(playground);

// Start alex at a specific position in the playground
alex = new Turtle(playground, 300.0, 100.0);
```

17.6. Glossary

default value

A value that is used if the programmer does not explicitly choose another value.

escape sequence

In a literal string, the backslash `\` is treated as an escape character. This mechanism allows us to write special escape sequences that can put non-printable or special characters into our strings.

optional parameters

Parameters in a method for which the call site *may* provide arguments. If the call site chooses not to provide an argument, the parameter will be given a default value.

`params`

A keyword in C# that is used before an array of parameters. It allows us to write methods that can pass a variable number of arguments (perhaps zero) which are all assigned to a single parameter.

verbatim literal strings

A string that starts with @". It can span multiple lines, and does not process escape characters. What we see is what we get!

17.7. Exercises

1. This string contains some escape characters (of a form that we never covered in this chapter).

```
string s = "\u2659  \u265A \u2669  \u266A  \u266B  \u266C  \u263a";
textBox1.Text = s;
```

Assign the string to the text of a text box, where we can make the font big. Or output the string as a MessageBox, set your Window title to this string. What do we get? See http://en.wikipedia.org/wiki/Miscellaneous_Symbols.

18. I/O, Files, and Networks

18.1. About I/O, Files and Networks

While a program is running, its data is stored in *random access memory* (RAM). RAM is fast, but it is also **volatile**, which means that when the program ends, or the computer shuts down, data in RAM disappears. To make data available the next time the computer is turned on and the program is started, it has to be written to a **file** on a **non-volatile** device such as a disk or USB memory stick, or the data needs to saved somewhere on the Internet (in the "cloud").

I/O is a shorthand for Input/Output. So whenever a program reads or writes data to something (or somewhere) outside itself (including files, memory, or the Internet), we'll call this I/O.

By reading and writing files, programs can save information between program runs.

Working with I/O is often like working with a notebook. To use a notebook, it has to be opened. When done, it has to be closed. While the notebook is open, it can either be read from or written to. In either case, the notebook holder knows where they are in the notebook. They can read the whole notebook in its natural order or they can skip around.

All of this applies to files as well. To open a file, we specify its name and indicate whether we want to read or write.

Most of the file-related classes are in the library called `System.IO`, so all the programs in this chapter will need a `using System.IO;` directive at the top of the file.

18.2. Writing our first file ¶

Let's begin with a simple program that writes a few lines of text to a file:

```
1  private void writeMyFirstTextFile()
2  {
3      TextWriter myfile = File.CreateText("C:\\temp\\test1.txt");
4      myfile.WriteLine("My first file written from C#");
5      myfile.WriteLine("----------------------------");
6      myfile.WriteLine("Hello, world!");
7      myfile.WriteLine("The time now is {0}", DateTime.Now.ToLongTimeString());
8      myfile.Close();
9  }
```

If there is no file with the given name on our disk, one will be created. If there already is one, it will be replaced by the file we are writing now.

To put a line of data into the file we invoke the `Write` or `WriteLine` method on the object, shown in lines 4–7 above. In bigger programs, these lines will usually be replaced by a loop that writes many lines into the file.

Closing the file (line 8) tells the system that we are done writing and makes the file available for reading by other programs (or by our own program).

The `WriteLine` method can do everything that `string.Format` can do, so it provides a convenient way to format our strings as we write them to the file.

18.3. Reading a file line-by-line (style 1)

Let us now read all the lines in some file, one line at a time.

```
1  private void readMyFirstFileLineByLine()
2  {
3      TextReader myfile = File.OpenText("C:\\temp\\poem.txt");
4      txtResult.Clear();
5      while (true)                              // Keep reading forever.
6      {
7          string theline = myfile.ReadLine();   // Try to read another line.
8          if (theline == null) break;           // If no more lines, leave the loop.
9
10         txtResult.AppendText(theline + "\n"); // process the line we've just read.
11     }
12     myfile.Close();                           // Close the file.
13 }
```

This is a handy pattern for our toolbox. In bigger programs, we'd normally squeeze more extensive logic into the body of the loop at line 10 — perhaps even call another method and pass the line to that method for handing. For example, if each line of the file contained the name and email address of one of our friends, perhaps we'd split the line into some pieces and send the friend a party invitation.

The end-of-file detection logic is important: when there are no more lines to be read from the file, `ReadLine` returns `null` — it does not refer to any string, not even an empty string.

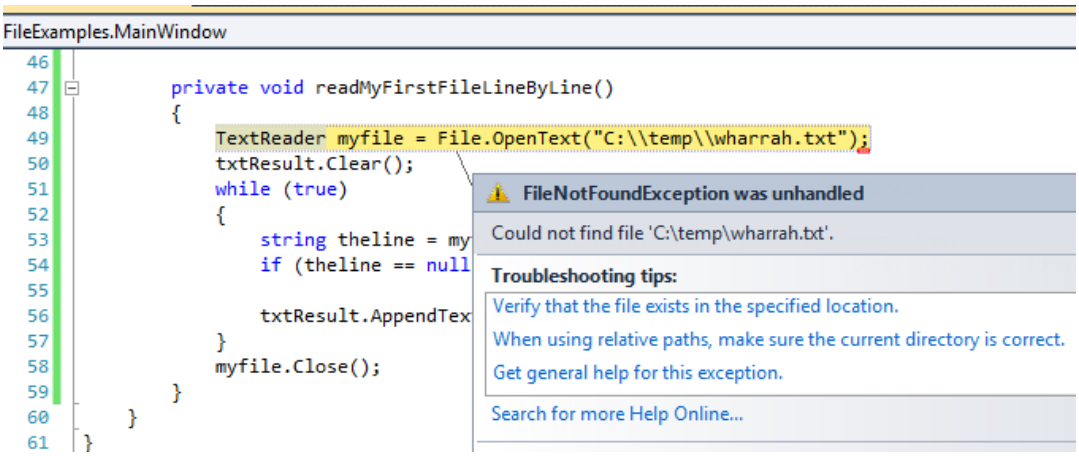
Fail first ...

In our sample case here, if we have three lines in the file, we'll execute line 7 and 8 *four* times. In C#, we only learn that the file has no more lines when we try to read another line, and fail. In some other programming languages (e.g. Pascal), things are different: we have what is called *look ahead*. Before reading each line we must call a method to test if there is a line to be read. In Pascal, we're not allowed to try to read a line that does not exist.

So the styles and templates for working line-at-a-time in Pascal and C# are subtly different!

When we transfer our C# skills to our next computer language, we'll need to clarify how we'll know when the file has ended: is the style in the language "try, and after we've failed we'll know", or is the style "look ahead"?

If we try to open a file for reading, but it doesn't exist, we'll get an error:



Relative File Paths

A file path like `C:\\temp\\poem.txt` is an **absolute file path**. Navigation begins at `C:` and goes into the `temp` folder.

Another way of expressing file locations — **relative paths** - can be used instead. In that case, navigation starts in the folder where the executable *.exe file is. The designation `..\\` means "go up one level to my parent". So consider these statements:

```
1  TextWriter a = File.CreateText("test.txt");
2  TextWriter b = File.CreateText("..\\..\\test.txt");
3  TextWriter c = File.CreateText("..\\..\\..\\test.txt");
```

- Line 1 creates the file in the same folder as the executable.
- Line 2 creates the file two levels above where the executable is. With a standard configuration of Visual Studio, that means "in our project directory".
- Line 3 creates the file three directory levels above the executable. In VS that usually means "in our solution directory".

18.4. Reading a file into an array of lines (style 2)

It is often useful to fetch data from a disk file and turn it into a array of strings, one string per line in the file. Suppose we have a file containing our friends and their email addresses, one per line in the file. But we'd like the lines sorted into alphabetical order. A good plan is to read everything into an array of lines, then sort the array, and then process the lines:

```
1  private void readToArrayOfLines()
2  {
3      string[] lines = File.ReadAllLines("..\\..\\poem.txt");
4      Array.Sort(lines);
5
6      foreach (string line in lines)
7      {
8          txtResult.AppendText(line + "\n");
```

```
    }
}
```

The `ReadAllLines` method in line 3 reads all the lines and returns an array of the strings. (Notice it even opens and closes the file for us, so it just needs one line our our code!) Line 4 uses a method in the `Array` class to sort the array, and lines 6–9 processes each line in some way.

We could have used the template from the previous section to read each line one-at-a-time, and to build up a list ourselves, but it is a lot easier to use the method that the C# implementors gave us!

What if we want to read a file into a list instead of array?

We cannot do so directly. But, recall that it is easy to convert arrays to lists, or vice-versa. Here we show how to read the contents of two files and put all their lines into the same list:

```
string[] textLines1 = File.ReadAllLines("..\\..\\vocab.txt");
List<string> xs = new List<string>(textLines1);

string[] textLines2 = File.ReadAllLines("..\\..\\alice_in_wonderland.txt");
xs.AddRange(textLines2);

MessageBox.Show(string.Format("There are {0} strings in list xs", xs.Count));
```

Line 1 reads one text file into one array of strings, line 4 does the same for reading another file to a second array. In line 2 we construct a new list containing all the strings in the first array. In line 5, we add all the second file's lines onto the end of `xs`.

Of course, you could even combine the logic of lines 1 and 2 into a single line, like this:

```
List<string> xs = new List<string>(File.ReadAllLines("..\\..\\vocab.txt"));
```

And some Computer Scientists consider it fun to see how much logic they can cram into a single line of code. So this one-liner is possible too ...

```
List<string> xs = new List<string>(File.ReadAllLines(...)).AddRange(File.ReadAllLines(...));
```

18.5. Reading the whole file into one string (style 3)

Another way of working with text files is to read the complete contents of the file into one big string, and then to use our string-processing skills to work with the contents.

We'd normally choose this style of working if we were not interested in the line-by-line structure of the file. For example, we've seen the `Split` method on strings which can break a string into words. So here is how we might count the number of words in a file:

```
private void readToString()
{
    string content = File.ReadAllText("..\\..\\poem.txt");

    string[] delimiters = null;
    string[] words = content.Split(delimiters, StringSplitOptions.RemoveEmptyEntries);

    MessageBox.Show(string.Format("There are {0} words in the file.", words.Length));
}
```

18.6. Processing data from text files

We're often asked to process data from scientific instruments, or from surveys, or data that has been exported from a spreadsheet. We'll assume here that we want to process "one line at a time", because data collection files are often very large. We'll call each line a **record**, and each record will have a couple of **fields**.

There are two cases we'll consider: some data files are "fixed-format": the fields occur in the same columns in every line of the file. In the other kind of file, we have fields that will be delimited by some special character that doesn't otherwise occur.

In the fixed-format case our strategy will be to read the line, (so we'll have a string), and then to extract our fields from the file using the string's `Substring` method. We might also need to convert the field into a number. Here are a few lines from a fixed-format text file

```
Tiger Woods             67   67   71   70
Kevin Streelman         69   70   71   67
Jeff Maggert            70   71   66   70
David Lingmerth         68   68   69   72
Martin Laird            71   67   73   67
Henrik Stenson          68   67   71   72
Ryan Palmer             67   69   70   72
...
```

In a delimited file, we'll use the string `Split` method to turn the fields into an array of strings, and we may also need to convert some of the fields to numbers. Here is the same data in a comma-delimited file:

```
Tiger Woods,67,67,71,70
Kevin Streelman,69,70,71,67
Jeff Maggert,70,71,66,70
David Lingmerth,68,68,69,72
Martin Laird,71,67,73,67
Henrik Stenson,68,67,71,72
Ryan Palmer,67,69,70,72
...
```

There are some exercises at the end of this chapter where we can practice processing both fixed-format or delimited files ...

18.7. An example: programming a filter

Many useful line-processing programs will read a text file line-at-a-time and do some minor processing as they write the (modified) lines to an output file. They might number the lines in the output file, or insert extra blank lines after every 60 lines to make it convenient for printing on sheets of paper, or only extract some specific columns from each line in the source file, or only print lines that contain a specific substring. This kind of program is called a **filter**.

Here is a filter that copies one file to another, omitting any lines that are too short:

```
1  TextReader myfile = File.OpenText("..\\..\\poem.txt");
2  TextWriter newfile = File.CreateText("..\\..\\poem_filtered.txt");
3  while (true)
4  {
5      string theline = myfile.ReadLine();
6      if (theline == null) break;
7      if (theline.Length > 30)
8      {
9          newfile.WriteLine(theline);
10     }
11 }
12 myfile.Close();
13 newfile.Close();
```

18.8. Working with binary files

Files that hold photographs, videos, zip files, executable programs, etc. are called **binary** files: they're not organized into lines, and cannot be opened with a normal text editor like NotePad++. C# works just as easily with binary files, but when we read from the file we're going to get bytes back rather than a string. Here we'll copy one binary file to another the easy way:

```
1  byte[] buffer = File.ReadAllBytes("C:\\temp\\somewhere.mp4");
2  File.WriteAllBytes("C:\\temp\\thecopy.mp4", buffer);
```

All the bytes are read into a buffer (which has type `byte[]`), and then the content of the buffer is written to a new file.

This method only works well if the file size is small enough so that the buffer can comfortably be held in memory. But if the buffer gets too big your system will become really slow or it will crash.

A more reliable but long-winded way of copying a big binary file is to read a chunk of the file, write it out, read the next chunk, etc. This means our computer memory only has to be large enough for the chunk that we're working with. Here is some code that shows how we'd organize that:

```
1  Stream inpf = File.OpenRead("C:\\temp\\somewhere.mp4");
2  Stream outf = File.Create("C:\\temp\\thecopy2.mp4");
3
```

```
4  const int chunkSz = 1024;
5  byte[] buffer = new byte[chunkSz];
6  while (true)
7  {
8      int n = inpf.Read(buffer, 0, chunkSz);
9      if (n == 0) break;
10     outf.Write(buffer, 0, n);
11 }
12 inpf.Close();
13 outf.Close();
```

There are a few new things here. In line 4 the const keyword prevents the value of chunkSz from being changed or reassigned at a later stage. In line 5 we pre-allocate an array (our buffer) to hold the data. At line 8 we read data into the buffer. The third argument gives the maximum number of bytes to be read. The `Read` method will transfer bytes into the buffer, and return the number of bytes successfully transferred. This gives us a mechanism for detecting when we're at the end of the input file, at line 9, to break out of the loop.

Note too that in line 10 we only write `n` bytes to the new file. On the last iteration of the loop, we're almost certain to get back fewer bytes than we requested. So it is important to use the count of how many we *actually* got when we write the data to our new file.

We don't do any more detailed work with the `byte` type in these notes!

18.9. What about fetching something from a network?

18.9.1. The general, harder way

In the above example our file was exposed to our program as a `Stream` object that we could read from.

But there are other kinds of specialized streams too, some very sophisticated. A stream can get its data from memory, from a file, from a connection to another program, from a database, or from the web, to name just a few sources. Additionally, some streams can unzip or decrypt data while reading, or encrypt or compress while writing.

So a `Stream` is a very central type of object allowing for different kinds of input and output in your program. (In the early sections of this chapter we used other object types like `TextReader` and `TextWriter`. These are "convenience" wrappers around streams, so there is still a `Stream` object at work in the heart of the system.)

To fetch a resource from the web we'll create a stream to read from. Then we'll just copy and reuse lines 4-13 from the code above.

```
1  using System.Net;  // Include this directive to access the Internet
2
3  ...                // Find an image to download from the web.
4  string myUri = "http://www.ict.ru.ac.za/resources/thinkSharply/thinksharply/_images/csharp_lessons.png";
5
6  // Create a web request
7  HttpWebRequest myReq = (HttpWebRequest) HttpWebRequest.Create(myUri);
8  // Get the response from the remote server
9  WebResponse myResp = myReq.GetResponse();
10
11 // Part of the response is a stream with our data ...
12 Stream inpf = myResp.GetResponseStream();
13 Stream outf = File.Create("C:\\temp\\theLogo.png");
14
15 // Repeat / adapt the code at lines 4-13 from the previous example.
```

We'll need to get a few things right before this works:

- The resource we're trying to fetch must exist! Check this using a browser.
- We'll need permission to write to the destination file, and the file will be created in the "current directory" — i.e. the same folder that the C# program is saved in.
- If we are behind a proxy server that requires authentication, (as some students are), this may require some more special handling to work around our proxy. Use a local resource for the purpose of this demo!

With the powerful notion of streams we can start to generalize how we think about data. Whether we're reading or writing data from or to files, the web, or another program, there are one consistent set of underlying I/O ideas.

There are some useful samples of how to do common I/O tasks at http://msdn.microsoft.com/en-us/library/ms404278(v=vs.100).aspx.

18.9.2. An easier way

```
using System.Net;  // Include this directive to access the Internet
...

// Now let a web client do the messy work for us ...
WebClient wc = new WebClient();

string addr1 = "http://www.ict.ru.ac.za/resources/ThinkSharply/thinkSharply/_downloads/golf_fixed_format.txt";
string content1 = wc.DownloadString(addr1);  // brings down the whole text file.

// Now fetch an image into a byte array, and save it to disk
string addr2 = "http://www.ict.ru.ac.za/resources/ThinkSharply/thinkSharply/_images/csharp_lessons.png";
byte[] buf = wc.DownloadData(addr2);
File.WriteAllBytes("C:\\temp\\pic.png",  buf);
```

The WebClient is a powerful class that makes fetching text or binary data from the web easy. We show two of its methods here (there are more): in the first case we fetch a text file. In the second we fetch an image. Instead of "http" as the protocol, others are possible too. So if the server understands "ftp", you could download a file using the ftp protocol instead. (ftp was an older way of fetching network files before the Web and http were invented.)

The important thing here is how our libraries can encapsulate and hide detail, and provide us with really powerful abstractions.

18.9.3. A cool thing about network programming

Historically, we had files but no networks. Modern network tools can not only access data over the network, but they can also treat your local disks as part of the network:

```
// One way to get all the text from a file
string content1 = File.ReadAllText("C:\\temp\\poem.txt");

// And a newer way to do the same thing!
WebClient wc = new WebClient();
string content2 = wc.DownloadString("file:///C:/temp/poem.txt");
```

Note `File.ReadAllText` uses backslashes whereas `wc.DownloadString` uses forwardslashes.

18.10. Glossary

delimited file

Often comma-delimited file or semicolon-delimited file: the fields in each record (line) are separate from one another by a delimiter. (Sometimes we also call this a file with *comma separated values*, or a CSV file). Our technique for processing a file like this will be to use the string `Split` method to split the record into its fields. (Compare fixed-format file.)

delimiter

Something that shows boundaries. In files, fields in a record are often delimited or separated by commas or semicolons.

field

A single unit of information, usually as part of a bigger record. In a student record, the student name would be one field, the birth date another field, etc.

file

A named entity, usually stored on a disk drive or network. It contains data. (See text file.)

file system

A method for naming, accessing, and organizing files and the data they contain.

filter

A program that reads an input file and produces an output file, generally with some transformation or selection of the data. It might only output lines that are not empty, or it might number the lines in the output file.

fixed-format file

A file in which every record, or line, has a fixed layout in specific columns. Our strategy for processing a file like this will be to extract the fields we need by using the string's `Substring` method. (Compare delimited file.)

I/O

An acronym for Input/Output. The mechanisms by which your program reads and writes data to other devices or other programs.

non-volatile memory

Memory that can maintain its state without power. Hard drives, flash drives, and compact disks (CDs) are examples of non-volatile memory.

path

A sequence of directory names that specifies the exact location of a file.

record

A sequence of fields that make up a single entity. Each student would have a record in a school's mark system. If a file is used to hold records, a record would usually be a single line in the file.

`Stream`

A type in the C# libraries that provides a very flexible source or destination for doing I/O. Streams can be used to read or write to files, or to or from memory, or to or from the web.

text file

A file that contains characters organized into lines separated by newline characters.

volatile memory

Memory which requires an electrical current to maintain state. The *main memory* or RAM of a computer is volatile. Information stored in RAM is lost when the computer is turned off.

18.11. Exercises

1. Write a program that reads a file and writes out a new file with the lines in reversed order (i.e. the first line in the old file becomes the last one in the new file.)
2. Write a program that reads a file and prints only those lines that contain the substring `void`.
3. Write a program that reads a file and produces an output file which is a copy of the file, except the first five columns of each line contain a four digit line number, followed by a space. Start numbering the first line in the output file at 1. Ensure that every line number is formatted to the same width in the output file. Use one of your C# programs as test data for this exercise: your output should be a printed and numbered listing of the C# program.
4. Write a program that undoes the numbering of the previous exercise: it should read a file with numbered lines and produce another file without line numbers.
5. Building on the example data in section Processing data from text files , download and save the files `a fixed format data file` and `a comma-delimited data file`.

   a. Write a program that reads the fixed format file and writes a new fixed-format file that shows each golfer, their scores for the four rounds, and their total score for all four rounds of the tournament:

```
Tiger Woods             67   67   71   70    275
Kevin Streelman         69   70   71   67    277
Jeff Maggert            70   71   66   70    277
David Lingmerth         68   68   69   72    277
Martin Laird            71   67   73   67    278
Henrik Stenson          68   67   71   72    278
Ryan Palmer             67   69   70   72    278
...
John Senden             73   70   71   73    287
...
```

   b. Now also output the golfer's position in the tournament. Woods was first with the lowest score, then the next three players all came (tied) second, and the next three all came (tied) 5th, etc. So output a file as shown below, where the field-width of the position is four columns, and the position is right-justified:

```
Position   1  Tiger Woods             67   67   71   70    275
Position   2  Kevin Streelman         69   70   71   67    277
Position   2  Jeff Maggert            70   71   66   70    277
Position   2  David Lingmerth         68   68   69   72    277
Position   5  Martin Laird            71   67   73   67    278
Position   5  Henrik Stenson          68   67   71   72    278
Position   5  Ryan Palmer             67   69   70   72    278
```

```
...
Position  45  John Senden            73   70   71   73    287
...
```

c. Now repeat the exercise: output exactly the same as the above, but take the input data from the comma-delimited file instead of the fixed format file. You should attempt to organize the code so that, as far as possible, you have as few changes between this version of the solution and the previous one.

19. List and Array Algorithms

This chapter is a bit different from what we've done so far: rather than introduce more new C# syntax and features, we're going to focus on the program development process, and some algorithms that work with lists and arrays.

As we've already seen, arrays and lists share many common features. The key difference is that arrays are fixed-size: once you create an array, its number of elements remains fixed. Lists, by contrast, can grow and shrink as our program runs. So in some examples here we use arrays, but we'll expect that we could use the same algorithm for lists. And vice-versa. And we sometimes use the word *list* to mean either an array or a list.

As in all parts of this book, our expectation is that you, the reader, will copy our code into your C# environment, play and experiment, and work along with us.

Part of this chapter works with the book `Alice in Wonderland` and a `vocabulary file`. Your browser should be able to download and save these files from these links.

19.1. Test-driven development

Early in our *`Value-returning methods`* chapter we introduced the idea of *incremental development*, where we added small fragments of code to slowly build up the whole, so that we could easily find problems early. Later in that same chapter we introduced *`unit testing`* and gave code for our testing framework so that we could capture, in code, appropriate tests for the methods we were writing.

Test-driven development (TDD) is a software development practice which goes one step further. The key idea is that automated tests should be written *first*. This technique is called *test-driven* because — if we are to believe the extremists — non-testing code should only be written after writing the tests, and in response to the fact that some test is failing.

We can still work in small incremental steps, but now we'll define and express those steps in terms of increasingly sophisticated unit tests that demand more from our code at each stage.

We'll turn our attention to some standard algorithms that process lists now, but as we proceed through this chapter we'll attempt to do so in the spirit envisaged by TDD.

19.2. The linear search algorithm

We'd like to know the index where a specific item occurs within in an array or list of items. We'll return the index of the item if it is found, or we'll return `-1` if the item doesn't occur in the list / array. Let us start with some tests in an array of strings:

```
string[] friends = { "Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris" };
Tester.TestEq(searchLinear(friends, "Zoe"), 1);
Tester.TestEq(searchLinear(friends, "Joe"), 0);
Tester.TestEq(searchLinear(friends, "Paris"), 6);
Tester.TestEq(searchLinear(friends, "Bill"), -1);
```

Motivated by the fact that our tests don't even run, let alone pass, we now write the method:

```
/// <summary>
/// Find and return the index of target in xs
/// </summary>
private int searchLinear(string[] xs, string target)
{
    for (int i = 0; i < xs.Length; i++)
    {
        if (xs[i] == target)
            return i;
    }
    return -1;
}
```

There are a some points to learn here: We've seen a similar algorithm before in the chapter on strings: there we searched for the index of a character in a string. There we also used a `while` loop, here we've used a `for` loop. There are other variations — perhaps we could use a `List<string>` instead of an array of strings, but the essential similarity in all these variations is that we test every item in turn. But we also ensure that as soon as we find the item we immediately return, without needed to examine the rest of the items.

Searching all items of a sequence from first to last is called a **linear search**. Each time we check an item, we'll call it a **probe**. We like to count probes as a measure of how efficient our algorithm is, and this will be a good indication of how long our algorithm will take to execute.

Let N be the length of the list to be searched. Linear searching is characterized by the fact that the number of probes needed to find some target depends directly on N. So if the list becomes ten times bigger, we can expect to wait ten times longer when searching for things. Notice too, that if we're searching for a target that is not present in the list, we'll have to go all the way to the end before we can return the negative value. So this case needs N probes. However, if we're searching for a target that does exist in the list, we could be lucky and find it immediately in position 0, or we might have to look further, perhaps halfway, perhaps even all the way to the last item. On average, when the target is present, we're going to need to go about halfway through the list, or N/2 probes.

We say that this search has **linear performance** (linear meaning *straight line*) because, if we were to measure the average search times for different sizes of lists (N) all containing random values, and then plot a graph of probes against N, we'd get an approximately straight line graph — as N gets bigger, so probes will increase proportionally.

Analysis like this is pretty meaningless for small collections — the computer is quick enough not to bother if the list only has a handful of items. So generally, we're interested in whether our algorithms are **scalable** — do they perform adequately if we throw bigger problems at them? Would this search be a sensible one to use if we had a million or ten million items (perhaps the catalogue of books in your local library)? What happens for really large datasets, e.g. how does Google search so brilliantly well?

19.3. A more realistic problem

As children learn to read, there are expectations that their vocabulary will grow. So a child of age 14 is expected to know more words than a child of age 8. When choosing reading books, an important question might be *"which words in this book are not in the expected vocabulary?"* Let's write a program to find out!

Let us assume we can already load a vocabulary of words into our program, and we can input the text of a book, and split it into an array of words. Let us write some tests for what we need to do next. Test data can usually be very small, even if we intend to finally use our methods for much larger cases:

```
string[] vocab = {"apple", "boy", "dog", "down",
                  "fell", "girl", "grass", "the", "tree"};
string[] book_words = "the apple fell from the tree to the grass".Split();
string[] empty = {};
Tester.TestEq(findUnknownWords(vocab, book_words), new string[] {"from", "to"});
Tester.TestEq(findUnknownWords(empty, book_words),  book_words);
Tester.TestEq(findUnknownWords(vocab, new string[] { "the", "boy", "fell" }), empty);
```

Notice we used `Split` to create our array of words — it is easier than typing in the array, and very convenient if you want to input a sentence into the program and turn it into an array of words.

We now need to implement the method for which we've written tests, and we'll make use of our linear search. The basic strategy is to run through each word in the book, look it up in the vocabulary, and if it is not in the vocabulary, save it into a new resulting array which we return from the method:

```
/// <summary>
/// Return an array of words from wds that do not occur in knownVocab
/// </summary>
private string[] findUnknownWords(string[] knownVocab, string[] wds)
{
    List<string> results = new List<string>();
    foreach (string w in wds)
    {
```

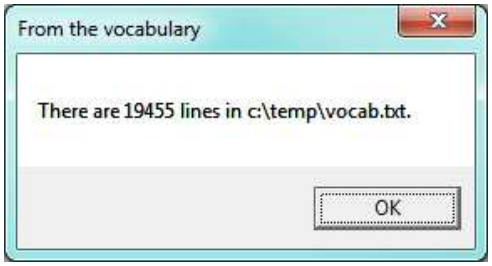
```
        if (searchLinear(knownVocab, w) < 0)
        {
            results.Add(w);
        }
    }
    return results.ToArray();
}
```

Now our tests all pass. In this example it makes sense to use a list rather than an array to collect the unknown words, because we can't tell how many there are going to be until we've found them. So we need the dynamic expandability that the list offers. Perhaps the method should have returned a list rather than an array.

Now let us look at scalability. We have more realistic vocabulary in the download provided at the beginning of this chapter, so let us read in that file as an array of lines. Here is a fragment of code:

```
string vocabPath = "..\\..\\vocab.txt";  // in the project folder
string[] vocab = File.ReadAllLines(vocabPath);
MessageBox.Show(string.Format("There are {0} lines in {1}.", vocab.Length, vocabPath),
                "From the vocabulary");
```

C# responds with:



So we've got a more sensible size vocabulary. If we open the file in a text editor we can confirm that the number of words matches what our program reports.

Now we tackle the problem of getting the book loaded and split into words. We're going to need a little black magic. Books have punctuation, and have mixtures of lower-case and upper-case letters. We need to clean up the contents of the book. This will involve converting everything to the same case (we'll choose lower-case, because our vocabulary happens to be lower-case), removing all the characters we don't want, and breaking what remains into words. But, in the spirit of Test Driven Development, we begin by writing some tests:

```
Tester.TestEq(convertToCleanedWords("My name ?? is Earl!"),
              new string[] {"my", "name", "is", "earl"});
Tester.TestEq(convertToCleanedWords("\"Well, I never!\", said Alice."),
              new string[] {"well", "i", "never", "said", "alice"});
```

We recall that the `Split` method has a convenient overloading that can do what we need: if we supply an array of `char`, it will use any one of the chars in the array as a delimiter. So our strategy will be to make a char array of all the punctuation, white space and characters that we don't want, and to split the string wherever we find one of those delimiters.

```
private string[] convertToCleanedWords(string theText)
{
    string t = theText.ToLower();
    string unwanted = " 0123456789!\"#$%&()*+,-./:;<=>?@[]^_`{|}~'\\;\r\n";
    char[] delims = unwanted.ToCharArray();
    string[] results = t.Split(delims, StringSplitOptions.RemoveEmptyEntries);
    return results;
}
```

Line 3 turns the whole string into lower-case. In line 4 we list the characters we want to get rid of, and line 5 turns this into a array of char, ready for use on line 6. Line 6 splits the text into a new word whenever it finds any one of the delimiters, and the delimiter is discarded in this process. The special option `StringSplitOptions.RemoveEmptyEntries` ensures that we don't return any empty words in the result array. Our tests pass now. (This is not a perfect word-splitter — for example, it will split "Alice's" into two words — "alice" and "s". But it is adequate for our textbook purpose of teaching some algorithms!)

It would be possible to combine all 5 lines in the body of the method into just a single line of code. But the step-by-step approach used here somehow feels more readable and more easily understandable (this should always be the top priority). It is certainly easier to step through with the debugger if we do more smaller steps rather than one giant one.

So now we're ready to read in our book with this fragment of code:

```
1  private string[] getWordsInBook(string bookPath)
2  {
3     return convertToCleanedWords(File.ReadAllText(bookPath));
4  }
5
6  ... //   ..\\..\\ goes up two levels, i.e. in the project folder
7  string bookPath = "..\\..\\alice_in_wonderland.txt";
8  string[] bookWords = getWordsInBook(bookPath);
9  MessageBox.Show(string.Format("There are {0} words in the book {1}.",
10                 bookWords.Length, bookPath), "From the book");
```

The MessageBox informs us that there are 27336 words in our book. If we set a breakpoint on line 9, and inspect `bookwords`, we find words like "alice", "s", "adventures", "in", "wonderland", "lewis", "carroll" ...

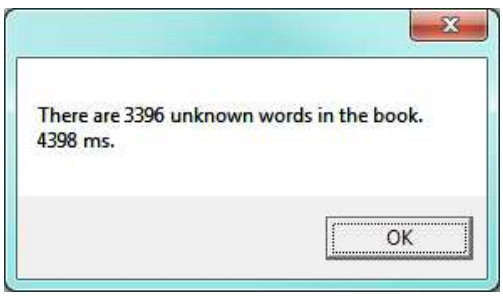
Now we have all our pieces ready. Let us see what words in this book are not in the vocabulary:

```
string[] missingWords = findUnknownWords(vocab, bookWords);
```

We wait some time while C# works its way through this, and finds the 3396 words in the book that are not in the vocabulary. Mmm... This is not particularly scalable. For a vocabulary that is twenty times larger (you'll often find school dictionaries with 300 000 words, for example), and for longer books, this is going to be quite slow. So let us make some timing measurements while we think about how we can improve this in the next section.

```
1  using System.Diagnostics;  // for Stopwatch
2
3  Stopwatch sw = new Stopwatch();
4  sw.Start();
5  string[] missing_words = findUnknownWords(vocab, bookWords);
6  double elapsedMilliSecs = (sw.Elapsed).TotalMilliseconds;
7  MessageBox.Show(string.Format("There are {0} unknown words in the book.\n{1:F0} ms.",
8                                missingWords.Length, elapsedMilliSecs));
```

We get the results and some timing that we can use for comparisons later:



Did it really give us a "correct" answer?

If you inspect the content of the `missingWords` array, it hasn't really answered our original question well. In fact, 398 of those missing words are repetitions of the word "alice", because "alice" isn't in our vocabulary. If an unknown word occurs multiple times, should we just count it once? Perhaps we should have asked our original question better.

19.4. Binary Search

If you think about what we've just done, it is not how we'd work in real life. If you were given a vocabulary and asked to tell if some word was present, you'd probably start in the middle. You can do this because the vocabulary is ordered — so you can probe some word in the middle, and immediately realize that your target was before (or perhaps after) the one you had probed. Applying this principle repeatedly leads us to a very much better algorithm for searching in a collection of items that are already ordered. (Note that if the items are not ordered, you have little choice other than to look through all of them. But, if we know the items are in order, we can improve our searching technique).

Lets start with some tests. Remember, the items need to be sorted if we're going to use this method:

```
string[] friends = { "Angelina", "Brad", "Joe", "Paris", "Thandi", "Zoe", "Zuki" };
Tester.TestEq(searchBinary(friends, "Bill"), -1);
Tester.TestEq(searchBinary(friends, "Abbey"), -1);
Tester.TestEq(searchBinary(friends, "Zummy"), -1);
for (int i = 0; i < friends.Length; i++)
{
    Tester.TestEq(searchBinary(friends, friends[i]), i);
}
```

Even our test cases are interesting this time: notice that we start with items not in the array and look at boundary conditions — in the middle of the array, less than all items in the array, bigger than the biggest. Then our loop uses every element as a target, and confirms that our binary search returns the corresponding index of that item.

It is useful to think about having a *region-of-interest* (ROI) within the array or list that is being searched. This ROI will be the portion of the list in which it is still possible that our target might be found. Our algorithm will start with the ROI initially set to all the items in the array. We'll always probe in the middle of the current ROI. On every probe there are three possible outcomes: either we find the target, or we learn that we can discard the top half of the ROI, or we learn that we can discard the bottom half of the ROI. We keep probing repeatedly, narrowing down the ROI, until we find our target or until we end up with no more items in our ROI. We can code this as follows:

```
/// <summary>
/// Find and return the index of target in array xs.
/// Return -1 if target does not exist.
/// </summary>
public int searchBinary(string[] xs, string target)
{
    int lb = 0;
    int ub = xs.Length;
    while (lb < ub) // exit if region of interest (ROI) becomes empty
    {
        // Next probe should be in the middle of the ROI
        int mid_index = (lb + ub) / 2;

        // Do the probe by fetching the item at that position
        string item_at_mid = xs[mid_index];

        // Console.WriteLine("Target=\"{0}\"  ROI=[{1} - {2}) (size={3}), probed xs[{4}]=\"{5}\"",
        //                   target, lb, ub, ub - lb, mid_index, item_at_mid);

        // How does the probed item compare to the target?
        int c = item_at_mid.CompareTo(target);
        if (c == 0)
        {
            return mid_index;      // Found it!
        }
        if (c < 0)
        {
            lb = mid_index + 1;    // Use upper half of ROI next time
        }
        else
        {
            ub = mid_index;        // Use lower half of ROI next time
        }
    }
    return -1; // we did not find it, and ROI is empty.
}
```

The region of interest is represented by two variables, a lower bound `lb` and an upper bound `ub`. It is important to be precise about what values these indexes have. We'll make `lb` hold the index of the first item in the ROI, (it is an *inclusive*, or *closed* bound), and we'll make `ub` hold the index just *beyond* the last item of interest (it is an *exclusive*, or *open* bound).

With this code in place, our tests pass. Great. Now if we substitute a call to this search algorithm instead of calling the `searchLinear` in `findUnknownWords`, can we improve our performance? When we do that, we get our results much faster: using the binary search reported a speed of 64ms whereas the linear search reported a speed of 4398ms — almost 70 times faster. (Your own measurements may be a bit different, of course.)

Why is this binary search so much faster than the linear search? If we uncomment the statement on lines 17 and 18, we'll be able to see the probes done during a search. Let's go ahead, and try that:

```
Target="magical"  ROI=[0 - 19455) (size=19455), probed xs[9727]="knowing"
Target="magical"  ROI=[9728 - 19455) (size=9727), probed xs[14591]="resurgence"
Target="magical"  ROI=[9728 - 14591) (size=4863), probed xs[12159]="overslept"
Target="magical"  ROI=[9728 - 12159) (size=2431), probed xs[10943]="misreading"
Target="magical"  ROI=[9728 - 10943) (size=1215), probed xs[10335]="magnet"
Target="magical"  ROI=[9728 - 10335) (size=607), probed xs[10031]="lightning"
Target="magical"  ROI=[10032 - 10335) (size=303), probed xs[10183]="longitudinal"
Target="magical"  ROI=[10184 - 10335) (size=151), probed xs[10259]="lumber"
Target="magical"  ROI=[10260 - 10335) (size=75), probed xs[10297]="lyrical"
Target="magical"  ROI=[10298 - 10335) (size=37), probed xs[10316]="made"
Target="magical"  ROI=[10317 - 10335) (size=18), probed xs[10326]="magic"
Target="magical"  ROI=[10327 - 10335) (size=8), probed xs[10331]="magnanimity"
Target="magical"  ROI=[10327 - 10331) (size=4), probed xs[10329]="magician"
Target="magical"  ROI=[10327 - 10329) (size=2), probed xs[10328]="magically"
Target="magical"  ROI=[10327 - 10328) (size=1), probed xs[10327]="magical"
Binary search returned 10327
```

Here we see that finding the target word "magical" needed just 15 probes before it was found at index 10327. The important thing is that each probe halves (with some truncation) the remaining region of interest. By contrast, the linear search would have needed 10328 probes to find the same target word.

The word *binary* means *two*. Binary search gets its name from the fact that each probe splits the list into two pieces and discards the one half from the region of interest.

The beauty of the algorithm is that we could double the size of the vocabulary, and it would only need one extra probe! And after another doubling, just another one probe. So as the vocabulary gets bigger, this algorithm's performance becomes even more impressive.

Can we put a formula to this? If our list size is N, what is the biggest number of probes k we could need? The maths is a bit easier if we turn the question around: how big a list N could we deal with, given that we were only allowed to make k probes?

With 1 probe, we can only search a list of size 1. With two probes we could cope with lists up to size 3 — (test the middle item with the first probe, then test either the left or right sub-list with the second probe). With one more probe, we could cope with 7 items (the middle item, and two sub-lists of size 3). With four probes, we can search 15 items, and 5 probes lets us search up to 31 items. So the general relationship is given by the formula

$$N = 2^k - 1$$

where k is the number of probes we're allowed to make, and N is the maximum size of the list that can be searched in k probes. This method is *exponential* in k (because k occurs in the exponent part). If we wanted to turn the formula around and solve for k in terms of N, we need to move the constant 1 to the other side, and take a log (base 2) on each side. (The log is the inverse of an exponent.) So the formula for k in terms of N is now:

$$k = \lceil log_2(N + 1) \rceil$$

The square-only-on-top brackets are called *ceiling brackets*: this means that you must round the number up to the next whole integer (because we can't make 7.3 probes — it will need to be 8).

Let us try this on a calculator, or in C#, which is the mother of all calculators: Here is a fragment of code that calculates and outputs the formula for various values of N:

```
foreach (int N in new int[] {1, 2, 3, 4, 5, 10, 100, 1000, 1000000, 1000000000})
{
   int maxProbes = Convert.ToInt32(Math.Ceiling(Math.Log(N + 1, 2)));
   Console.WriteLine("{0} items would need at most {1} probes.", N, maxProbes);
}
```

And here are the magical results:

```
1 items would need at most 1 probes.
2 items would need at most 2 probes.
3 items would need at most 2 probes.
4 items would need at most 3 probes.
5 items would need at most 3 probes.
10 items would need at most 4 probes.
100 items would need at most 7 probes.
1000 items would need at most 10 probes.
1000000 items would need at most 20 probes.
1000000000 items would need at most 30 probes.
```

This tells us that searching 1000 items needs 10 or fewer probes. (Well technically, with 10 probes we can search exactly 1023 items, but the easy and useful stuff to remember here is that “1000 items needs 10 probes, a million needs 20 probes, and a billion items only needs 30 probes.”)

You will rarely encounter algorithms that scale to large datasets as beautifully as binary search does!

19.5. Getting the unique elements in an array

We sometimes want to get the unique elements in an array or a list. We could use a list and delete the elements we don’t want. Or we could build a new list that contains only those elements we do want, while leaving the original array or list unchanged.

Consider our case of looking for words in Alice in Wonderland that are not in our vocabulary. We had a report that there are 3398 such words, but there are duplicates in that list. How can we remove these duplicates?

A good approach is to first sort the array — this means any duplicates will be positioned next to each other. Then we can build a new list, without duplicates. Let us start with some test cases for removing adjacent duplicates from an array that is already sorted:

```
1  Tester.TestEq(removeAdjacentDups(new string[]{"a","b","b","b","c","c"}),
2                                   new string[]{"a","b","c"});
3  Tester.TestEq(removeAdjacentDups(new string[]{}), new string[]{});
4  Tester.TestEq(removeAdjacentDups(new string[]{"a", "big", "big", "bite", "dog"}),
5                              new string[]{"a", "big", "bite", "dog"});
```

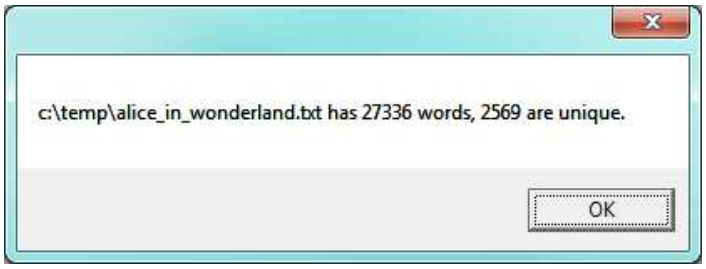
The algorithm is easy and efficient. We simply have to remember the most recent item that was inserted into the result, and avoid inserting it again:

```
1  private string[] removeAdjacentDups(string[] xs)
2  {
3      if (xs.Length == 0) return new string[] {};
4      List<string> results = new List<string>();
5      string elemLastAdded = xs[0];
6      results.Add(elemLastAdded);
7      foreach (string x in xs)
8      {
9          if (x != elemLastAdded) {
10             results.Add(x);
11             elemLastAdded = x;
12         }
13     }
14     return results.ToArray();
15 }
```

The amount of work done in this algorithm is linear — each item in `xs` causes the loop to execute exactly once, and there are no nested loops. So doubling the number of elements in `xs` should cause this method to run twice as long: the relationship between the size of the list and the time to run will be graphed as a straight (linear) line.

Let us go back now to our analysis of *`ALice in Wonderland`*. Before checking the words in the book against the vocabulary, we'll sort those words into order, and eliminate duplicates. So our new code and the resulting output looks like this:

```
string[] wds = getWordsInBook(bookPath);
Array.Sort(wds);
string[] uniqueWds = removeAdjacentDups(wds);
MessageBox.Show(string.Format("{0} has {1} words, {2} are unique.",
                bookPath, wds.Length, uniqueWds.Length));
```



If we now look up our unique words the vocabulary, we'll realize two advantages:

- We do about 10 times less work because the number of unique words is about 10 times smaller than the original. So it runs about 10 times faster.
- The answer is more useful: it gives us an accurate idea of what new words the children will encounter if we prescribe this book as a set work. And, of course, the unknown words will already be in alphabetic order.

It is pretty amazing that Lewis Carroll was able to write a classic piece of literature using only 2569 different words!

19.6. Merging sorted arrays

Suppose we have two sorted arrays. Devise an algorithm to merge them together into a single sorted array.

A simple but inefficient algorithm could be to define a new array that is big enough to hold all the elements, copy all the elements to the new array, and then sort it.

But this doesn't take advantage of the fact that the input arrays are already sorted.

Lets get some tests together first:

```
int[] xs = { 1, 3, 5, 7, 8, 8, 13, 15, 17, 19 };
int[] ys = { 4, 8, 12, 16, 20, 24 };
int[] zs = { 1, 3, 4, 5, 7, 8, 8, 8, 12, 13, 15, 16, 17, 19, 20, 24 };
int[] empty = { };

Tester.TestEq(merge(xs, empty), xs);
Tester.TestEq(merge(empty, ys), ys);
Tester.TestEq(merge(empty, empty), empty);
Tester.TestEq(merge(xs, ys), zs);
```

Here is our merge algorithm:

```
public int[] merge(int[] xs, int[] ys)
{
    int[] results = new int[xs.Length + ys.Length];
    int xi = 0, yi = 0, zi = 0;

    while (xi < xs.Length && yi < ys.Length)
```

```
7       {
8           // Both arrays still have items, copy smaller item to result.
9           results[zi++] =  (xs[xi] <= ys[yi] ? xs[xi++] : ys[yi++]);
10      }
11
12      // Copy items from whichever array has some remaining.
13      while (yi < ys.Length) // Add remaining items from ys
14      {
15          results[zi++] = ys[yi++];
16      }
17      while (xi < xs.Length) // Add remaining items from xs
18      {
19          results[zi++] = xs[xi++];
20      }
21
22      return results;
23  }
```

The algorithm works as follows: we create a result array that is the correct size. We keep three indexes, one into each input array, and one into the result array. On each iteration of the loop, whichever array item is smaller gets copied to the result, and that array's index is advanced. (The index of the result array is also advanced.) As soon as either index for the input arrays reaches the end of its array, we exit the loop and copy any remaining items to the result.

Line 15 is a compact way of writing three separate statements. `result[zi++]` is common idiom in the C-like family of languages. It means "use the value of `zi` to index the array, and then (for next time), increment the value of `zi`". So line 15 is equivalent to the more verbose version

```
1   result[zi] = ys[yi];    // First use the old values of xi and yi, do the work,
2   zi++;                   // then increment the two indexes.
3   yi++;
```

Line 9 uses this shorthand, and also uses a *conditional operator* (also found in C, C++, Java, etc.). The expression before the `?` is a boolean condition that is evaluated. If it is true, the result of the expression become the expression after the question mark, otherwise the result of the whole expression becomes the expression after the colon. So line 9 could be written more verbosely like this:

```
1   if  (xs[xi] <= ys[yi])
2   {
3        result[zi] = xs[xi];
4        zi++;
5        xi++;
6   }
7   else
8   {
9        result[zi] = ys[yi];
10       zi++;
11       yi++;
12  }
```

What if I wanted to use this algorithm to merge arrays of strings? I can create an overloaded method (another method with the same name, but a different signature) that uses string arrays. Three lines would need to change:

```
1   public string[] merge(string[] xs, string[] ys)
2   {
3       string[] results = new string[xs.Length + ys.Length];
4       int xi = 0, yi = 0, zi = 0;
5
6       while (xi < xs.Length && yi < ys.Length)
7       {
8           // Both lists still have items, copy smaller item to result.
9           results[zi++] = (xs[xi].CompareTo(ys[yi]) <= 0 ? xs[xi++] : ys[yi++]);
10      }
11
12      // Copy items from whichever list has some remaining.
13      while (yi < ys.Length) // Add remaining items from ys
```

```
14      {
15          results[zi++] = ys[yi++];
16      }
17      while (xi < xs.Length) // Add remaining items from xs
18      {
19          results[zi++] = xs[xi++];
20      }
21
22      return results;
23  }
```

Notice that in line 9 we had to use the `CompareTo` method for the comparison (because strings cannot be compared with `<=`). Interestingly, though, integers, doubles, chars, etc. can also be compared using `CompareTo` — so we could use line 9 from this version in the version for integers, and we'd reduce the number of differences to just two lines: the method signature on line 1, and the definition of the new array on line 3.

19.7. Generic methods — a first look

If I wanted to merge arrays of `double`, I'd need another method overloading with two lines different. And then another overloading for merging arrays of `char`, or arrays of `student`, and so on. Each new method would have just two lines of code different from the others.

C# has a powerful mechanism that allows us to parametrize a method so that it works for any type T. (We've already seen the notation with lists.)

```
1   public T[] merge<T>(T[] xs, T[] ys)
2   {
3       T[] results = new T[xs.Length + ys.Length];
4       int xi = 0, yi = 0, zi = 0;
5
6       while (xi < xs.Length && yi < ys.Length)
7       {
8           // Both lists still have items, copy smaller item to result.
9           results[zi++] = (xs[xi].CompareTo(ys[yi]) <= 0 ? xs[xi++] : ys[yi++]);
10      }
11
12      // Copy items from whichever list has some remaining.
13      while (yi < ys.Length) // Add remaining items from ys
14      {
15          results[zi++] = ys[yi++];
16      }
17      while (xi < xs.Length) // Add remaining items from xs
18      {
19          results[zi++] = xs[xi++];
20      }
21
22      return results;
23  }
```

This still needs another tweak before it works, but let's understand what we've got so far:

The `<T>` in the signature in line 1 is called a *type parameter*. It says "this works for any type, call the type T". A method that works for many types is called a **generic method**. Then the T can also be used in the signature and in the body of the method as a substitution. So reading line 1, it says "The method expects two arrays with elements of type `T` as inputs, and it returns an array of T". Each time we call this method from a different place in our code, the `T` could be substituted by a different *concrete type* (e.g. `int`, `string`, `Turtle`).

So, if we attempt to merge an array of strings with another array of strings to produce an array of strings, it is going to work just fine. But trying to merge an array of ints with an array of strings will produce an error: `T` cannot be an `int` and a `string` at the same time.

Line 3 says "Define a new array of type T, and instantiate it."

If we copy this code into our program, we get an error on line 9. The compiler complains that *"T does not contain a definition for 'CompareTo'."*

This is because not all element types T can be compared to each other. So, for example, if we tried to merge two arrays of `Turtle` or `Button`, there is no `CompareTo` method that lets us compare one turtle to another. At the heart of the problem is that our "generic" mechanism is *too* general. We need a little more magic on line 1, like this:

```
1  public T[] merge<T>(T[] xs, T[] ys) where T:IComparable
```

The `where` keyword introduces an extra constraint on what concrete types can be used at any call site. It says "Only types that satisfy the `IComparable` interface are acceptable". We'll learn more about interfaces shortly. But for now, it is enough to know that this means that the compiler now has a guarantee that there will be a `CompareTo` method for type T, and our error on line 9 disappears.

With this change in place we now have a generic `merge` method that works for any arrays of element types, provided they can be compared to each other. If we did attempt to pass arrays of `Turtle` or `Button`, or any other types that don't support comparison, we'd get a compilation error at at the call site.

Most of the methods we work with can be generalized in this way: we can have a binary search, for example, that works for arrays of any comparable type. For most of the methods we cover in this chapter we won't do this generalization — simply because it allows us to focus more on the logic of the algorithms, rather than the C# machinery for making the algorithm more general.

19.8. Alice in Wonderland, again!

Underlying the algorithm for merging sorted lists is a deep pattern of computation that is widely reusable. The pattern essence is *"Run through the lists always processing the smallest remaining items from each, with these cases to consider:"*

- What should we do when either list has no more items?
- What should we do if the smallest items from each list are equal to each other?
- What should we do if the smallest item in the first list is smaller than the smallest one the second list?
- What should we do in the remaining case?

Lets assume we have two sorted lists. Exercise your algorithmic skills by adapting the merging algorithm pattern for each of these cases:

- Return only those items that are present in both lists.
- Return only those items that are present in the first list, but not in the second.
- Return only those items that are present in the second list, but not in the first.
- Return items that are present in either the first or the second list.
- Return items from the first list that are not eliminated by a matching element in the second list. In this case, an item in the second list "knocks out" just one matching item in the first list. This operation is sometimes called *bagdiff*. For example `bagdiff(new int[] {5,7,11,11,11,12,13}, new int[] {7,8,11});` would return `new int[] {5,11,11,12,13}`

In the previous section we sorted the words from the book, and eliminated duplicates. Our vocabulary is also sorted. So third case above — find all items in the second list that are not in the first list, would be another way to implement `findUnknownWords`. Instead of searching for every word in the dictionary (either by linear or binary search), why not use a variant of the merge to return the words that occur in the book, but not in the vocabulary.

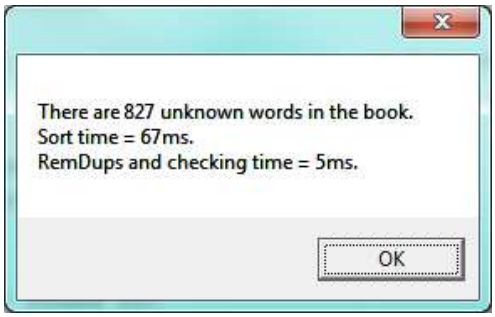
```
1  private string[] findUnknownsMergePattern(string[] vocab, string[] wds)
2  {
3      List<string> results = new List<string>();
4
5      int xi = 0, yi = 0;
6
7      while (xi < vocab.Length && yi < wds.Length)
8      {
9          int v = vocab[xi].CompareTo(wds[yi]);
10         if (v < 0)
11         {
12             xi++;         // move past this vocab word
```

```
        }
        else if (v == 0)
        {
            yi++;         // this word is recognized
        }
        else
        {
            results.Add(wds[yi++]);   // this word not in vocab
        }
    }

    // Copy any words that have not yet been checked.
    // If the vocab is at the end, they are all unrecognised.
    while (yi < wds.Length)
    {
        results.Add(wds[yi++]);
    }

    return results.ToArray();
}
```

Now we put it all together:

```
private void btnMergeBook_Click(object sender, RoutedEventArgs e)
{
    string[] vocab = File.ReadAllLines(vocabPath);
    string[] bookWords = getWordsInBook(bookPath);

    Stopwatch sw = new Stopwatch();
    sw.Start();
    Array.Sort(bookWords);
    sw.Stop();
    int sortTime = Convert.ToInt32((sw.Elapsed).TotalMilliseconds);

    sw.Restart();
    string[] uniqueWds = removeAdjacentDups(bookWords);
    string[] missingWords = findUnknownsMergePattern(vocab, uniqueWds);

    int checkingTime = Convert.ToInt32(sw.Elapsed).TotalMilliseconds);
    MessageBox.Show(
         string.Format("There are {0} unknown words in the book.\n" +
               "Sort time = {1}ms.\nRemDups and checking time = {2}ms.",
           missingWords.Length, sortTime, checkingTime));
}
```

Even more stunning performance here:



Let's review what we've done. We started with a word-by-word linear lookup in the vocabulary that ran in about 5 seconds. We implemented a clever binary search, about 70 times faster. But then we did something even better: we sorted the words from the book, eliminated duplicates, and used a merging pattern to find words from the book that were not in the dictionary. Not only did we now have a more useful result: the unique unknown words in alphabetical order, but the algorithm is a lot faster than our first attempt!

That is what we can call a good day at the office!

19.9. Glossary

linear

Relating to a straight line. Here, we talk about graphing how the time taken by an algorithm depends on the size of the data it is processing. Linear algorithms have straight-line graphs that can describe this relationship.

merge algorithm

An efficient algorithm that merges two already sorted lists, to produce a sorted list result. The merge algorithm is really a pattern of computation that can be adapted and reused for various other scenarios, such as finding words that are in one array but not in another.

probe

While searching for an item, each time we take a look we call it a probe.

scalable

An algorithm or technique which remains viable or practical when applied to large problems.

search; binary

A famous algorithm that searches for a target in a sorted list. Each probe in the list allows us to discard half the remaining items, so the algorithm is very efficient.

search; linear

A search that probes each item in a list or sequence, from first, until it finds what it is looking for. It is used for searching for a target in unordered lists of items.

test-driven development

A software development practice which arrives at a desired feature through a series of small, iterative steps motivated by automated tests which are *written first* that express increasing refinements of the desired feature. (see the Wikipedia article on Test-driven development for more information.)

19.10. Exercises

1. The section in this chapter called Alice in Wonderland, again! started with the observation that the merge algorithm uses a pattern that can be reused in other situations. Adapt the merge algorithm to write each of these methods, as was suggested there:

    a. Return only those items that are present in both arrays.
    b. Return only those items that are present in the first array, but not in the second.
    c. Return only those items that are present in the second array, but not in the first.
    d. Return items that are present in either the first or the second array.
    e. Return items from the first array that are not eliminated by a matching element in the second array. In this case, an item in the second array "knocks out" just one matching item in the first array. This operation is sometimes called *bagdiff*. For example `bagdiff([5,7,11,11,11,12,13], [7,8,11])` would return `[5,11,11,12,13]`

2. Every week a computer scientist buys four lotto tickets. On each lotto ticket she must pick 6 numbers between 1 and 49. She always chooses prime numbers, with the hope that if she ever hits the jackpot, she will be able to go onto TV and facebook and tell everyone her secret. This will suddenly create widespread public interest in prime numbers, and will be the trigger event that ushers in a new age of enlightenment and world peace. She represents her weekly tickets as an array of arrays:

```
int[] [] my_tickets =
          {   new int[] { 7, 17, 37, 19, 23, 43},
              new int[] { 7,  2, 13, 41, 31, 43},
              new int[] { 2,  5,  7, 11, 13, 17},
              new int[] {13, 17, 37, 19, 23, 43} };
```

    Complete these exercises.

    a. Each lotto draw picks six random balls, numbered from 1 to 49. Write a method to return a random lotto draw. Note that it should not be possible to pick the same ball more than once. Statisticians call this "picking balls without replacement".

b. Write a method that compares a single ticket and a draw, and returns the number of correct picks on that ticket:

```
Tester.TestEq(lotto_match(new int[]{42,4,7,11,1,13}, new int[]{2,5,7,11,13,17}), 3);
```

c. Write a method that takes an array of tickets and a draw, and returns an array telling how many picks were correct on each ticket:

```
Tester.TestEq(lotto_matches(new int[]{42,4,7,11,1,13}, my_tickets), new int[]{1,2,3,1});
```

d. Write a method that takes an array of integers, and returns the number of primes in the array:

```
Tester.TestEq(primes_in(new int[]{42, 4, 7, 11, 1, 13}), 3);
```

e. Write a method to discover whether the computer scientist has missed any prime numbers in her selection of her tickets. Return an array of all primes that she has missed:

```
Tester.TestEq(prime_misses(my_tickets), new int[]{3, 29, 47});
```

f. Write a method that repeatedly makes a new draw, and compares the draw to the four tickets.

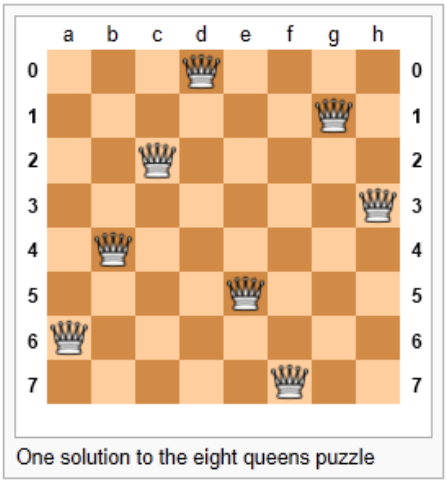
   i. Count how many draws are needed until one of the computer scientist's tickets has at least 3 correct picks. Try the experiment twenty times, and average out the number of draws needed.
   ii. How many draws are needed, on average, before she gets at least 4 picks correct?
   iii. How many draws are needed, on average, before she gets at least 5 correct? (This might take a while.)

Notice that we have difficulty constructing test cases here, because our random numbers are not deterministic. Automated testing only really works if you already know what the answer should be!

3. Read *Alice in Wonderland*. You can read the plain text version we have with this textbook, or if you have e-book reader software on your PC, or a Kindle, iPhone, Android, etc. you'll be able to find a suitable version for your device at http://www.gutenberg.org/. They also have html and pdf versions, with pictures, and thousands of other classic books!

20. The N-Queens Puzzle — a Case Study

As told by Wikipedia, *"The eight queens puzzle is the problem of placing eight chess queens on an 8x8 chessboard so that no two queens attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal."*



One solution to the eight queens puzzle

Please try this yourself, and find a few more solutions by hand.

We'd like to write a program to find solutions to this puzzle. In fact, the puzzle generalizes to placing N queens on an NxN board, so we're going to think about the general case, not just the 8x8 case. Perhaps we can find solutions for 12 queens on a 12x12 board, or 20 queens on a 20x20 board.

20.1. Getting Started

How do we approach a complex problem like this? A good starting point is to think about our *data structures* — how exactly do we plan to represent the state of the chessboard and its queens in our program? Once we have an initial idea about how our puzzle is going to be represented in memory, we can begin to think about the methods and logic we'll need to solve it, i.e. perhaps we'll need a way to put another queen onto the board somewhere, or to check whether a specific queen clashes with any of the other queens on the board.

The steps of finding a good representation, and then finding a good algorithm to operate on the data cannot always be done independently of each other. As you think about the operations you require, you may want to change or reorganize the data somewhat to make it easier to do the operations you need.

This relationship between algorithms and data was elegantly expressed in the title of a book *Algorithms + Data Structures = Programs*, written by one of the pioneers in Computer Science, Niklaus Wirth, the inventor of Pascal.

Let's brainstorm some ideas about how a chessboard and queens could be represented in memory.

- A two dimensional 8x8 array or a list of lists is one possibility. At each square of the board we would like to know whether it contains a queen or not — just two possible states for each square — so perhaps each element could be a boolean, or, more simply, 0 or 1.

  Our state for the solution pictured above could then have this data representation: [1].

  [1] Notice that we're not using C# syntax here — it is a bit easier and cleaner without C#.

```
bd1 = [[0,0,0,1,0,0,0,0],
       [0,0,0,0,0,0,1,0],
       [0,0,1,0,0,0,0,0],
       [0,0,0,0,0,0,0,1],
       [0,1,0,0,0,0,0,0],
       [0,0,0,0,1,0,0,0],
       [1,0,0,0,0,0,0,0],
       [0,0,0,0,0,1,0,0]]
```

You should also be able to see how the empty board would be represented, and you should start to imagine what operations or changes you'd need to make to the data to place another queen somewhere on the board.

- Another idea might be to keep a list of coordinate positions of where the queens are currently placed. Using the notation in Wikipedia's illustration at the start of this chapter, for example, we could represent the state of that solution as:

```
bd2 = [ "a6", "b4", "c2", "d0", "e5", "f7", "g1", "h3" ]
```

- We could make other tweaks to this — perhaps each element in this array should rather be an x,y pair, with integer coordinates for both axes. And being good computer scientists, we'd probably start numbering each axis from 0 instead of at 1. Now our representation could be:

```
bd3 = [(0,6), (1,4), (2,2), (3,0), (4,5), (5,7), (6,1), (7,3)]
```

- Looking at this representation, we can't help but notice that the first coordinates are `0,1,2,3,4,5,6,7` and they correspond exactly to the index position of the pairs in the array. So we could discard them, and come up with this really compact alternative representation of the solution:

```
bd4 = [6, 4, 2, 0, 5, 7, 1, 3]
```

This is what we'll use: a simple 1-dimensional array (or list) of int. Let's see where that takes us.

This representation is not general

We've come up with a great representation for the N-Queens problem. But will it work for other puzzles?

Our simple 1-dimensional array has the limitation that we can only put one value in each position, which means we'll never be able to represent a board with two queens in the same column. But that is a puzzle constraint anyway — no two queens are allowed to share the same column. So puzzle and data representation are well matched.

But if we were trying to solve a different puzzle on a chessboard, perhaps play a game of checkers, where many pieces could be in the same column, our one-dimensional array representation would not be general enough.

Let us now take some grand "Aha!" insight into the problem. Is it just a coincidence that there are no repeated numbers in the solution? The solution `[6,4,2,0,5,7,1,3]` contains the numbers `0,1,2,3,4,5,6,7`, but none are duplicated! Could other solutions ever contain duplicate numbers, or not?

A little thinking should convince you that there can never be duplicate numbers in a solution: the numbers represent the row on which the queen is placed, and because we are never permitted to put two queens in the same row, no solution can have duplicate row numbers in it.

Our key "Aha!" insight

In our chosen representation, every solution to the N queens problem must therefore be a permutation of the numbers [0 .. N-1].

Not all permutations are solutions. For example, `[0,1,2,3,4,5,6,7]` has all queens on the same diagonal, so it is not a solution. But all solutions are permutations.

Wow, we seem to be making progress on this problem merely by thinking, rather than coding!

Our algorithm can start taking shape now. We can start with an array containing values [0..N-1], generate various permutations of the array, and check each permutation to see if it has any clashes (queens that are on the same diagonal). If it has no clashes, we have found a solution, and we can output it.

Let us be precise and clear on this issue: if we only use permutations of the rows, and we're using our compact representation, no queens can clash on either rows or columns, and we don't even have to concern ourselves with testing those cases. So the only clashes we need to test for are clashes on the diagonals.

It sounds like a useful method will be one that can test if two queens share a diagonal.

Each queen is at some (x,y) position. So does the queen at (5,2) share a diagonal with the one at (2,0)? Does (5,2) clash with (3,0)?

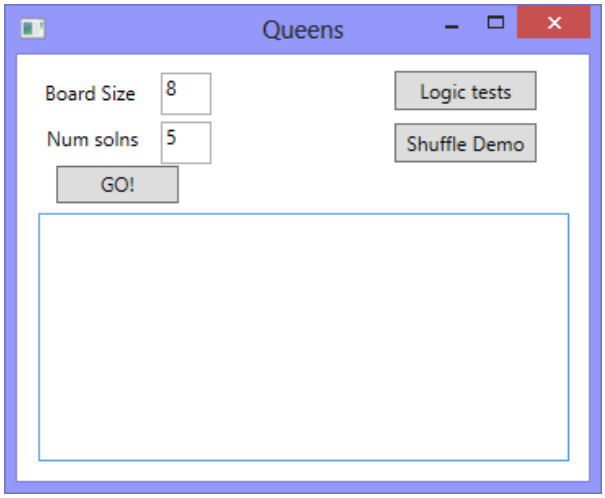
```
Tester.TestEq(shareDiagonal(5, 2, 2, 0), false);
Tester.TestEq(shareDiagonal(5, 2, 3, 0), true);
Tester.TestEq(shareDiagonal(5, 2, 4, 3), true);
Tester.TestEq(shareDiagonal(5, 2, 4, 1), true);
```

A little geometry will help us here. A diagonal has a slope of either 1 or -1. The question we really want to ask is *is their distance between them the same in the x and the y direction?* If it is, they share a diagonal. Because diagonals can be to the left or right, it will make sense for this program to use the absolute distance in each direction:

```
/// <summary>
/// Is (x0, y0) on a shared diagonal with (x1, y1)?
/// </summary>
private bool shareDiagonal(int x0, int y0, int x1, int y1)
{
    int dy = Math.Abs(y1 - y0);       // Calc the absolute y distance
    int dx = Math.Abs(x1 - x0);       // Calc the absolute x distance
    return dx == dy;                  // They clash if dx == dy
}
```

If you copy the code and run it, you'll be happy to learn that the tests pass!

Let's put together a little GUI now so that we have some scaffolding from which to run our tests, or to show our solutions. This GUI is mostly self explanatory: some labels and text boxes for input, a couple of buttons to run some tests, and a text box for showing the results that we're interested in. We'll set up the results text box to stretch when the window resizes, and we can set its property to give it a vertical scroll bar.



One way to check all queens for any clashes is to work through the board, left-to-right, and checking each queen against all those to its left. For example, when we consider the queen on column 6, it is enough to check for clashes against those in all the columns to its left, i.e. in columns 0,1,2,3,4,5. (Convince yourself that no queen needs to check any queens to its right.)

So the next building block is a method that can check whether the queen at column `c` clashes with any of the queens to its left, at columns 0,1,2,..c-1: The tests can go behind the button called "Logic tests" in our GUI.

```
Tester.TestEq(colHasDiagonalClashes(new int[] { 6, 4, 2, 0, 5 }, 4), false);
Tester.TestEq(colHasDiagonalClashes(new int[] { 6, 4, 2, 0, 5, 7, 1, 3 }, 7), false);
Tester.TestEq(colHasDiagonalClashes(new int[] { 0, 1 }, 1), true);
Tester.TestEq(colHasDiagonalClashes(new int[] { 5, 6 }, 1), true);
Tester.TestEq(colHasDiagonalClashes(new int[] { 6, 5 }, 1), true);
Tester.TestEq(colHasDiagonalClashes(new int[] { 0, 6, 4, 3 }, 3), true);
Tester.TestEq(colHasDiagonalClashes(new int[] { 5, 0, 7 }, 2), true);
```

```
Tester.TestEq(colHasDiagonalClashes(new int[] { 2, 0, 1, 3 }, 1), false);
Tester.TestEq(colHasDiagonalClashes(new int[] { 2, 0, 1, 3 }, 2), true);
```

Here is our method that makes them all pass:

```
/// <summary>
/// Return true if the queen at column col clashes
/// on the diagonal with any queen to its left.
/// </summary>
private bool colHasDiagonalClashes(int[] bd, int col)
{
    for (int i = 0; i < col; i++)   // Look at all columns to the left of col
    {
        if (shareDiagonal(i, bd[i], col, bd[col]))
            return true;
    }
    return false;           // No clashes
}
```

Finally, we're going to give our program one of our permutations — i.e. all queens placed somewhere, one on each row, one on each column. But does the permutation have any diagonal clashes?

```
// Solution from picture above
Tester.TestEq(boardHasDiagonalClashes(new int[] { 6, 4, 2, 0, 5, 7, 1, 3 }), false);
Tester.TestEq(boardHasDiagonalClashes(new int[] { 4, 6, 2, 0, 5, 7, 1, 3 }), true);
Tester.TestEq(boardHasDiagonalClashes(new int[] { 0, 1, 2, 3 }), true);  // Try small 4x4 board
Tester.TestEq(boardHasDiagonalClashes(new int[] { 2, 0, 3, 1 }), false); // Solution to 4x4 case
```

And the code to make the tests pass:

```
/// <summary>
/// Determine whether we have any queens clashing on the diagonals.
/// </summary>
private bool boardHasDiagonalClashes(int[] bd)
{
    for (int col = 1; col < bd.Length; col++)
    {
        if (colHasDiagonalClashes(bd, col)) return true;
    }
    return false;
}
```

Summary of what we've done so far: we now have a powerful method called `boardHasDiagonalClashes` that can tell if a permutation is a solution to the queens puzzle. Let's get on now with generating lots of permutations and finding which of them happen to be solutions!

20.2. Shuffling an array

Before we return to the Queens problem, we'll need a method that shuffles an array of integers into a random order. C# provides a built-in method that can sort a pair of two "parallel" arrays. The first array contains keys (to be used for the comparisons), and the second array contains associated data. So, if we had keys [30, 10, 20] and data ["Joe", "Zim", "Abe"] we can do the paired-array sort and the arrays would be changed to [10, 20, 30] and ["Zim", "Abe", "Joe"]. So the relationship between the i'th elements (between the 10 and "Zim") in the paired arrays is preserved.

Our shuffling strategy is to create a new key array of the same length as the items, and to fill it with random numbers. Then we can use the built-in C# method to sort the pair of arrays on the keys. This will randomly shuffle the item array.

```
Random rng = new Random();

private void shuffle(int[] xs)
{
    // Create an array of random numbers to use as the sorting key.
    int[] keys = new int[xs.Length];
```

```
7       for (int i = 0; i < xs.Length; i++)  keys[i] = rng.Next();
8
9       Array.Sort(keys, xs);
10  }
```

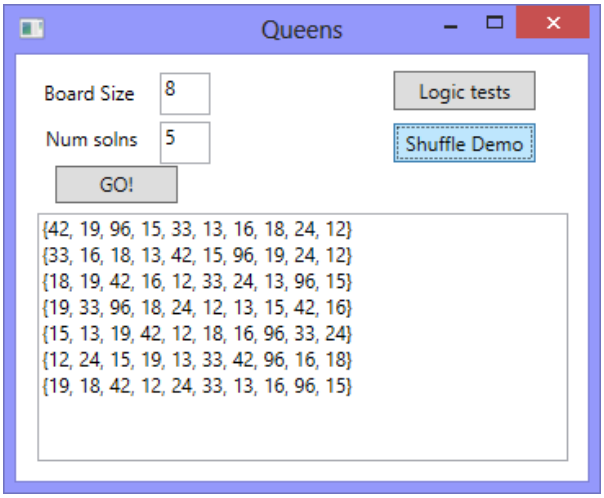
Effectively each value in the array xs is paired to a randomly generated int (the key). When the randomly generated keys are sorted into an ascending order in line 9 it results in the array, xs, being sorted into a random order (ie. randomly shuffled). We've chosen to put the random number generator outside of the method, as a class-level variable, so it only gets instantiated once.

Well, here our unit testing strategy breaks down. We don't have an easy way to automatically check that the shuffle is working because its output cannot be predicted from its inputs. It turns out that testing and verifying random processes is a complex task — people expend a great deal of effort attempting to show that the random numbers are not predictable.

For our purposes, a handler like the following will probably be enough to convince us that each time we click the button we get a different shuffle of the array.

```
1   private string stringify(int[] xs)
2   {
3       string result = "{";
4       string separator = "";
5       foreach (int x in xs)
6       {
7           result += separator + x.ToString();
8           separator = ", ";
9       }
10      return result + "}";
11  }
12
13  private void btnShuffleTest_Click(object sender, RoutedEventArgs e)
14  {
15      int[] xs = { 12, 15, 13, 18, 16, 24, 96, 42, 33, 19 };
16      shuffle(xs);
17      txtResults.AppendText(string.Format("{0}\n",stringify(xs)));
18      txtResults.ScrollToEnd();
19  }
```

Repeated events gave us output like this, but of course each time we run it we'd expect different results:



20.3. Putting it all together

This is the fun part where we get to see some results of all our hard work. We could try to find all permutations of `[0,1,2,3,4,5,6,7]` — that might be algorithmically challenging, and would be a *brute force* way of tackling the problem. We just try everything, and find all possible solutions.

Of course we know there are N! permutations of N things, so we can get an early idea of how long it would take to search all of them for all solutions. Not too long at all, actually - 8! is only 40320 different cases to check out. This is vastly better than starting with 64 places to put eight queens. If you do the sums for how many ways can you choose 8 of the 64

squares for your queens, the formula (called *N choose k* where you're choosing k=8 squares of the available N=64) yields a whopping 4426165368, obtained from (64! / (8! x 56!)).

What is the most significant thing in this chapter?

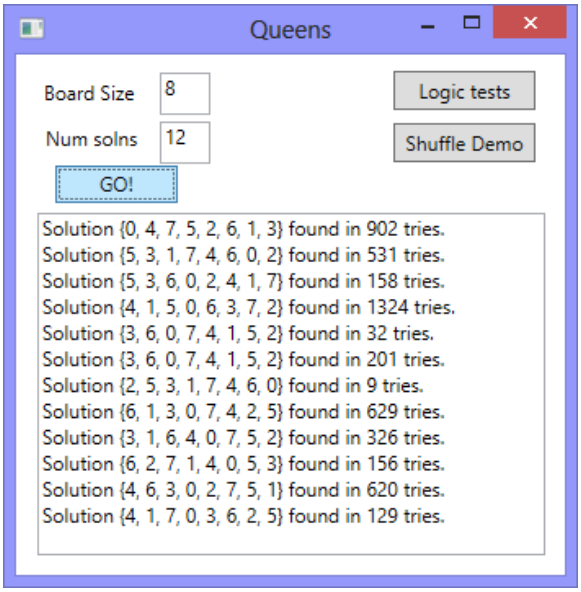
Our earlier important insight — that we only need to consider permutations — has reduced what we call the *problem space* for the 8x8 Queens problem from about 4.4 billion cases to just 40320!

Good algorithms and good programs are not just about neat lines of code. Here we took advantage of a very simple representation for our boards, and had a pretty deep insight into the properties that any solution needed to possess.

We're not even going to explore all 40320 permutations, however. Instead we're going to write a "random" algorithm to find solutions to the N queens problem. We'll begin with the permutation [0,1,2,3,4,5,6,7] and we'll repeatedly shuffle and test the array, looking for a solution! Along the way we'll also count how many shuffle/test attempts we need before we stumble across the next solution. (We could find the same solution more than once, because our shuffling will be random!).

```
/// <summary>
/// Find and return one random solution to the NxN queens problem.
/// </summary>
/// <param name="N">The size of the board</param>
/// <returns>A solution to the N-queens problem</returns>
private int[] findQueensSolution(int N)
{
    // set up the initial board of the correct size
    int[] bd = new int[N];
    // Fill the initial array with numbers 0,1,2,...N-1
    for (int i = 0; i < N; i++) bd[i] = i;

    int tries = 1;
    // Just keep checking and shuffling until we get lucky.
    while (boardHasDiagonalClashes(bd))
    {
        shuffle(bd);
        tries++;
    }

    // output the results ...
    txtResults.AppendText(string.Format("Solution {0} found in {1} tries.\n", stringify(bd), tries));
    txtResults.ScrollToEnd();

    return bd;
}

private void btnGo_Click(object sender, RoutedEventArgs e)
{
    int sz = Convert.ToInt32(txtSz.Text);
    int numSolutionsWanted = Convert.ToInt32(txtNumSolutions.Text);

    int numFound = 0;
    while (numFound < numSolutionsWanted)
    {
        int[] solution = findQueensSolution(sz);
        numFound++;
    }
}
```

And now we get ...



Here is an interesting observation. On an 8x8 board, there are known to be 92 different solutions to this puzzle. By shuffling, we are randomly picking one of 40320 possible permutations of our representation. So our chances of picking a solution on each try are 92/40320. Put another way, on average we'll need 40320/92 tries — about 438.26 random shuffles — before we stumble across a solution. The number of tries we showed here looks like our experimental data average should be quite close to that, so our experimental data agrees quite nicely with our theory!

We'll leave it as an exercise to you, the reader, to repeatedly find solutions and work out the average number of tries needed for different size boards. Perhaps leave your computer running overnight, and see if you can find some solutions to a 24x24 board.

Save this code for later.

Later in the course we'll want to draw the boards.

20.4. Exercises

1. Modify the queens program to solve some boards of size 4, 12, and 16. What is the maximum size puzzle you can usually solve in under a minute?
2. What is the average number of shuffles you'd expect to have to get for a 10x10 puzzle before you stumbled across a solution? What about bigger puzzles? Wikipedia can tell you how many solutions exist for different size boards, so you can compute the expected number of tries quite easily.
3. Add some logic to find and display the average number of tries needed to find N solutions. So when the GO! button is clicked asking for 20 solutions, say, compute the average number of tries for finding those 20. Compare this experimental data to the theoretical prediction.
4. Adapt the queens program so that we keep a list of solutions that have already been found, so that we don't find the same solution more than once. (This may be trickier than it first appears: we'll need to be able to compare two boards to tell if their array elements are identical, but there is no built-in comparison in C# that can test an array for element-by-element equality.)
5. Chess boards are symmetric: if we have a solution to the queens problem, its mirror image — either flipping the board on the X or on the Y axis, or on a diagonal, is also a solution. And rotating a solution by 90 degrees, 180 degrees, or 270 degrees gives more solutions. In some sense, solutions that are just mirror images or rotations of other solutions — in the same family — are less interesting than the unique "core cases". Of the 92 solutions for the 8 queens problem, there are only 12 unique families if you take rotations and mirror images into account. Wikipedia has some fascinating stuff about this.
    a. Write a method to mirror a solution in the Y axis (i.e. given `[0,4,7,5,2,6,1,3]`, produce `[3,1,6,2,5,7,4,0]`).
    b. Write a method to mirror a solution in the X axis.
    c. Write a method to rotate a solution by 90 degrees anti-clockwise, and use this to provide 180 and 270 degree rotations too.

d. Write a method which is given a solution, and it generates the family of symmetries for that solution. For example, the symmetries of `[0,4,7,5,2,6,1,3]` (not C# syntax, again) are

```
[[0,4,7,5,2,6,1,3],[7,1,3,0,6,4,2,5],
 [4,6,1,5,2,0,3,7],[2,5,3,1,7,4,6,0],
 [3,1,6,2,5,7,4,0],[0,6,4,7,1,3,5,2],
 [7,3,0,2,5,1,6,4],[5,2,4,6,0,3,1,7]]
```

e. Now adapt the queens program so it won't output solutions that are in the same family. Only show solutions from unique families.

21. Recursion

Recursion means “defining something in terms of itself” usually at some smaller scale, perhaps multiple times, to achieve your objective. For example, we might say “A human being is someone whose mother is a human being”, or “a directory is a structure that holds files and (smaller) directories”, or “a family tree starts with a couple who have children, each with their own family sub-trees”.

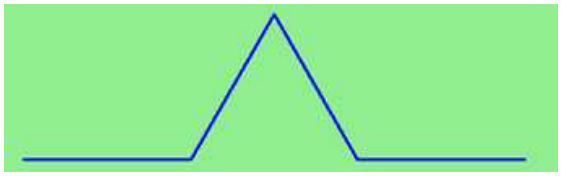
So methods can *call themselves* to solve smaller sub-problems. This idea that the method can usefully call itself is the essence of recursion.

21.1. Drawing Fractals

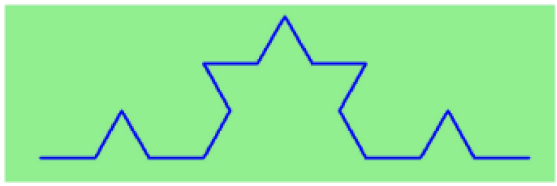
For our purposes, a **fractal** is a recursive drawing which has *self-similar* structure. It can be defined in terms of itself.

Let us start by looking at the famous Koch fractal. An order 0 Koch fractal is simply a straight line of a given size.

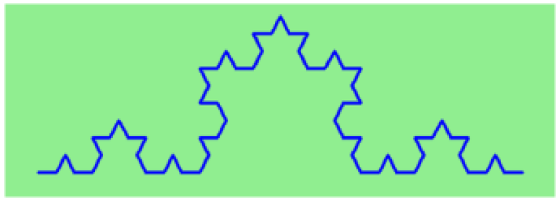
An order 1 Koch fractal is obtained like this: instead of drawing just one line, draw instead four smaller segments, in the pattern shown here:



Now what would happen if we repeated this Koch pattern again on each of the order 1 segments? We’d get this order 2 Koch fractal:



Repeating our pattern again gives us an order 3 Koch fractal:



Now let us think about it the other way around. To draw a Koch fractal of order 3, we can simply draw four order 2 Koch fractals. But each of these in turn needs four order 1 Koch fractals, and each of those in turn needs four order 0 fractals. Ultimately, the only drawing that will take place is at order 0. This is very simple to code up in C#:

```
    void koch(Turtle t, int order, double size)
    {
        // Make turtle t draw a Koch fractal of 'order' and 'size'.
        // Leave the turtle facing the same direction as it started in.

        if (order == 0)
        {   // The base case is just a straight line
            t.Forward(size);   // This is the only place we draw!
        }
        else
        {
            koch(t, order - 1, size / 3);   // Make a smaller koch drawing, 1/3 size
            t.Left(60);
            koch(t, order - 1, size / 3);
            t.Right(120);
            koch(t, order - 1, size / 3);
```

```
            t.Left(60);
            koch(t, order - 1, size / 3);
        }
    }
```

The key thing that is new here is that if order is not zero, `koch` calls itself recursively to get its job done.

Whenever `koch` calls itself, it asks for a koch line of one order less than itself. Take some time to convince yourself that every recursive call is a "simpler" sub-problem, and that the process must terminate.

Recursion, the high-level view

One way to think about this is to convince yourself that the method works correctly when you call it for an order 0 fractal. Then do a mental *leap of faith*, saying *"the fairy godmother* (or C#, if you can think of C# as your fairy godmother) *knows how to handle the recursive order 0 calls for me on lines 12, 14, 16, and 18, so I don't need to think about that detail!"* All I need to focus on is how to draw an order 1 fractal *if I can assume the order 0 one is already working.*

You're practising *mental abstraction* — ignoring the sub-problem while you solve the big problem.

If this mode of thinking works for you (and you should practice it!), then take it to the next level. Aha! Now can I see that it will work when called for order 2 *under the assumption that it is already working for order 1*.

And, in general, if I can use mental abstraction to gloss over how the order n-1 case works, can I just focus on solving the order n problem?

Students of mathematics who have played with proofs of induction should see some very strong connect-the-dots similarities here.

Recursion, the low-level operational view

Another way to understand recursion is to get rid of it! If we use separate methods to draw an order 3 fractal, an order 2 fractal, an order 1 fractal and an order 0 fractal, we could simplify the above code, quite mechanically, to a situation where there was no longer any recursion, like this:

```
void koch0(Turtle t, double size)
{
    t.Forward(size);
}

void koch1(Turtle t, double size)
{
    koch0(t, size / 3);
    t.Left(60);
    koch0(t, size / 3);
    t.Right(120);
    koch0(t, size / 3);
    t.Left(60);
    koch0(t, size / 3);
}

void koch2(Turtle t, double size)
{
    koch1(t, size / 3);
    t.Left(60);
    koch1(t, size / 3);
    t.Right(120);
    koch1(t, size / 3);
    t.Left(60);
    koch1(t, size / 3);
}

void koch3(Turtle t, double size)
{
    koch2(t, size / 3);
    t.Left(60);
    koch2(t, size / 3);
    t.Right(120);
    koch2(t, size / 3);
    t.Left(60);
    koch2(t, size / 3);
}
```

This trick of "unrolling" the recursion gives us an operational view of what happens. You can trace or single-step the program into `koch3`, and from there, into `koch2`, and then into `koch1`, etc., all the way down the different layers of method calls.

If `koch3` is called once, `koch2` will be called four times. How many times will `koch0` eventually be called?

Single-stepping and taking an operational view of recursion can help build your insight. The mental goal is, however, to be able to do the abstraction for the general case!

21.2. Case study: Factorials

Six factorial (written elsewhere as 6!) can easily be calculated in a loop, as 6x5x4x3x2x1. But we can also look at this with our recursive spectacles: N factorial is N x (N-1 factorial). Of course, if we define it in terms of itself we'll need a base case too. So we complete our recursive definition by defining 0 factorial to be 1. Here is a recursive value-returning method that computes factorials:

```
1  private int fact(int n)
2  {
3      if (n <= 1) return 1;
4      else return n * fact(n - 1);
5  }
6
7  private void btnFact_Click(object sender, RoutedEventArgs e)
8  {
9      int n = Convert.ToInt32(txtFact.Text);
10     MessageBox.Show(string.Format("{0} factorial is {1}", n, fact(n)));
11 }
```

When we run this a message box pops up to (correctly) tell us that 5 factorial is 120.

A deck of playing cards has 52 cards (without Jokers). If we shuffle a deck, how many possible shufflings are there? (Shuffling a deck puts the cards into a different permutation, so the question becomes "how many permutations can we have of 52 distinct objects?", and the answer, of course, is 52 factorial.)

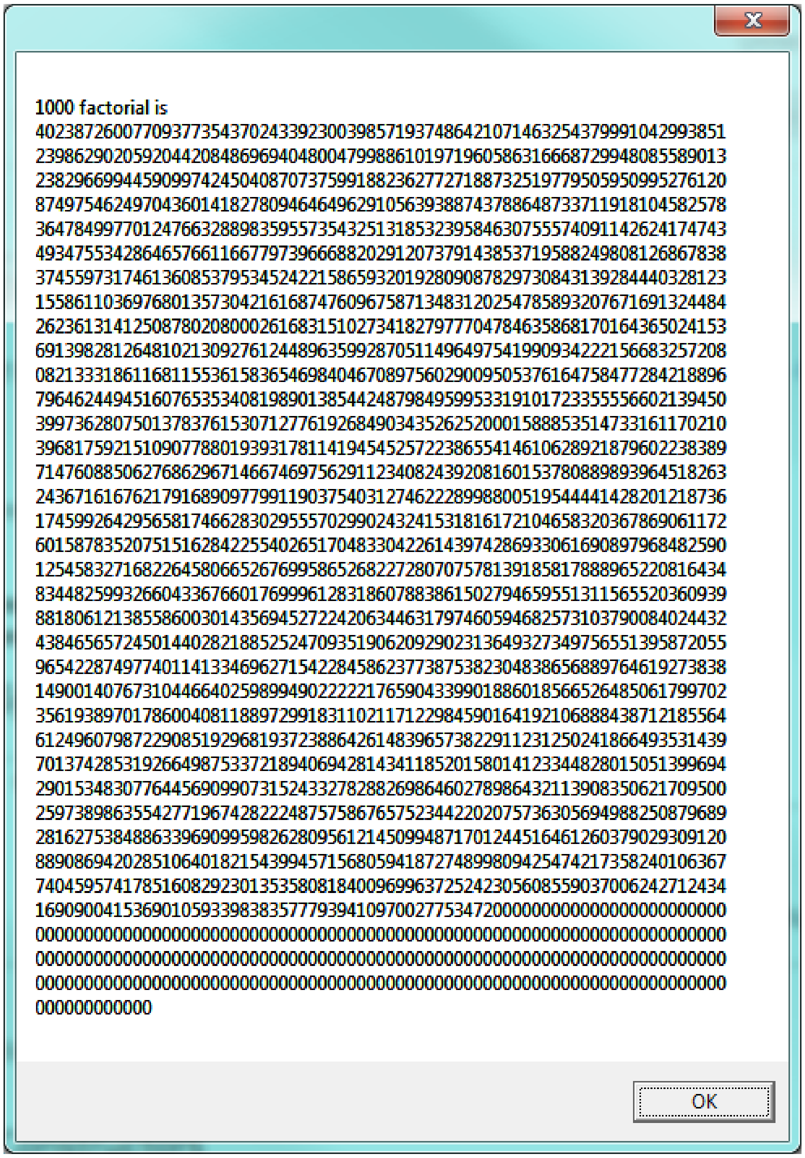
But when we run our program to compute 52 factorial, our program gets it wrong, with an incorrect result of zero.

52! is a large number (relative to our 32-bit integers in C#) — too big to store in a `int`. So the computation fails with what is called an *overflow error*, and fails to detect the problem.

We can set some options in Visual Studio for *overflow checking*, and make sure that the run-time error is brought to our attention rather than silently ignored. (Search the Internet for "C# enable Overflow Checking" if you want to try this approach, but it won't help you to compute 52! or 1000!)

Often we do need to compute with big integers, especially in fields like cryptography and problems that need us to compute 1000! exactly. So there is a library called `System.Numerics` that provides a structure called `BigInteger` for representing an arbitrary large signed integer.

We need to add a reference to our project to include the `System.Numerics` module, and we'll add the directive `using System.Numerics;` at the top of the file we're working with. But after that, it is really simple: all we change is the return type of the `fact` method in line 1 above, to make the return type `BigInteger` rather than `int`. Now we're good for this example:



21.3. Case study: Fibonacci numbers

The famous **Fibonacci sequence** 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 134, ... was devised by Fibonacci (1170-1250), who used this to model the breeding of pairs of rabbits. If, in generation 7 you had 21 pairs in total, of which 13 were adults, then next generation the adults will all have bred new children, and the previous children will have grown up to become adults. So in generation 8 you'll have 13+21=34, of which 21 are adults.

This *model* to explain rabbit breeding made the simplifying assumption that rabbits never died. Scientists often make (unrealistic) simplifying assumptions and restrictions to make some headway with the problem.

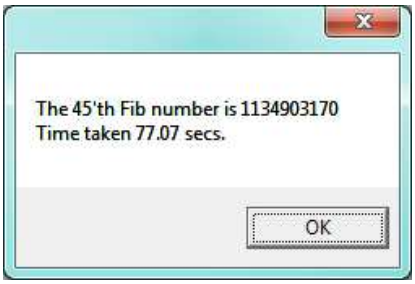
If we number the terms of the sequence from 0, we can describe each term recursively as the sum of the previous two terms:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)  for n >= 2
```

This translates very directly into some C#:

```
using System.Diagnostics;   // for Stopwatch

private int fib(int n)
{
    if (n <= 1) return n;
    int t = fib(n - 1) + fib(n - 2);
    return t;
```

```
}

private void btnFib_Click(object sender, RoutedEventArgs e)
{
    int n = Convert.ToInt32(txtFib.Text);
    Stopwatch sw = new Stopwatch();
    sw.Start();
    int result = fib(n);
    double elapsedSecs = (sw.Elapsed).TotalSeconds;
    MessageBox.Show(string.Format("The {0}'th Fib number is {1}\nTime taken {2:F2} secs.",
                                                              n, result, elapsedSecs));
}
```

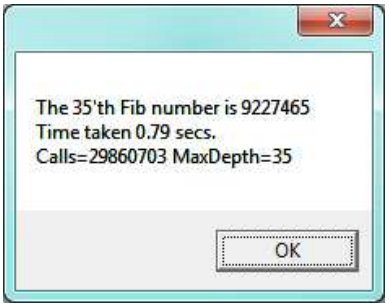


We get the correct result, but as we ask for more terms in the sequence we get an exploding amount of work! This is a particularly inefficient algorithm. There are much better ways to compute this.

21.4. Debugging and instrumenting recursion

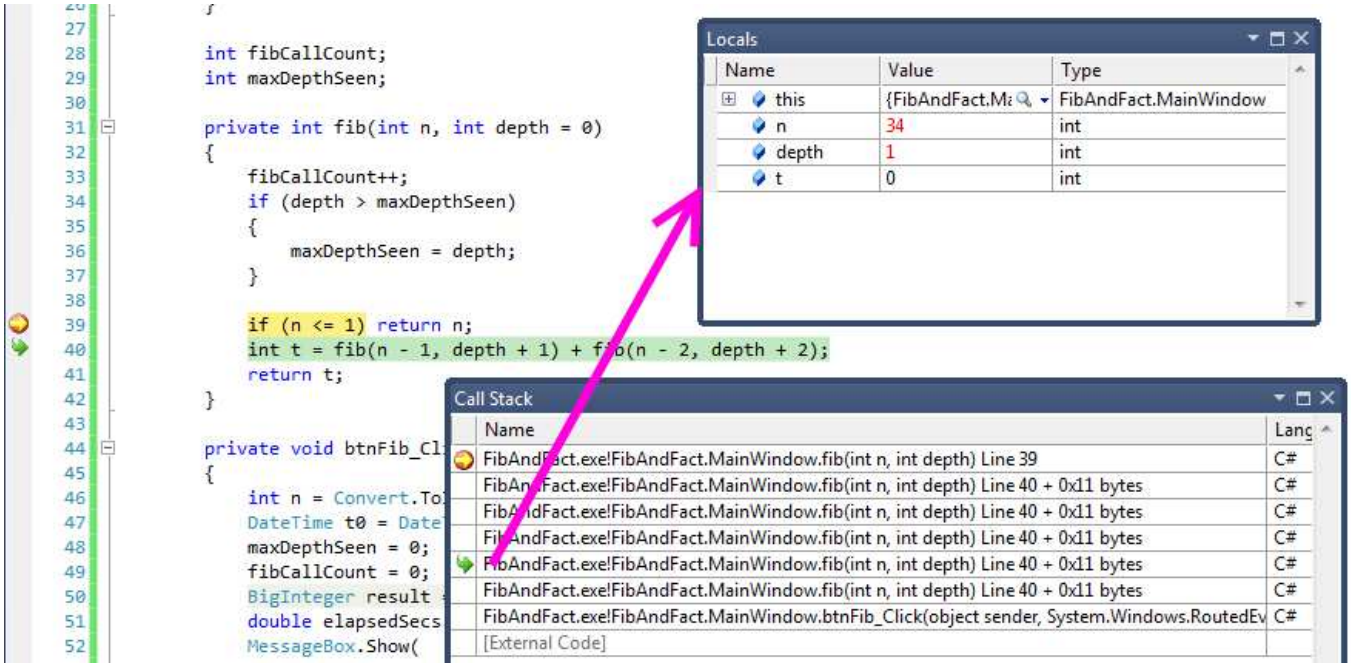
Suppose we wanted to know how many times `fib` had been called above, and what the maximum depth of recursion was that had been reached. Using class-level variables and some extra parameter information is a handy way of arranging this:

```
using System.Diagnostics; // for Stopwatch

int fibCallCount, maxDepthSeen;

private int fib(int n, int depth = 0)
{
    fibCallCount++;
    if (depth > maxDepthSeen)
    {
        maxDepthSeen = depth;
    }

    if (n <= 1) return n;
    int t = fib(n - 1, depth + 1) + fib(n - 2, depth + 2);
    return t;
}

private void btnFib_Click(object sender, RoutedEventArgs e)
{
    int n = Convert.ToInt32(txtFib.Text);
    Stopwatch sw = new Stopwatch();
    sw.Start();
    maxDepthSeen = 0;
    fibCallCount = 0;
    BigInteger result = fib(n);
    double elapsedSecs = (sw.Elapsed).TotalSeconds;
    MessageBox.Show(
        string.Format("The {0}'th Fib number is {1}\nTime taken {2:F2} secs.\nCalls={3} MaxDepth={4}",
                      n, result, elapsedSecs, fibCallCount, maxDepthSeen));
}
```



Notice that the caller had to set the two class-level variables to zero at lines 20-21 before calling `fib`. Notice too on line 3 that depth was made an optional parameter with a default value of zero. So the first call to `fib` (line 22) did not supply an argument, but the recursive calls at line 11 made sure than the value passed to the "next" recursive invocation was one more than the depth at the current invocation.

Another useful mechanism is single-stepping and debugging. In Visual Studio one can set a *condition* on a breakpoint by right-clicking on the breakpoint, and adding a condition. So let's set a conditional breakpoint at line 11 in the program above: the program should only break when `n == 30`.



The picture shows that the program has entered debugging mode at line 39. The Call Stack window shows the stack frames of all the recursive calls, with the most recent one at the top. We've chosen one of the stack frames (with the curly green arrow at the left). The local variable window shows the local variables for whichever stack frame we select: in this case we're inspecting the values in the frame that is at depth 1 (counting from the bottom, the first `fib` frame is at depth zero).

Using the debugger cleverly like this will not only help us find problems, but inspecting the stack and the locals in each stack frame will build a solid and accurate understanding of how our programs are executing.

21.5. Processing recursive directories and files

While the recursive Fibonacci and Factorial examples are interesting teaching examples (especially if you know the typically recursive definitions from a mathematics course), both are easier to implement without using recursion. This section introduces a much more compelling example — one that is much more difficult to implement if we don't use recursion.

The problem is to traverse a recursive data structure: in our case, a directory on our disk. The directory may recursively contain other directories and files.

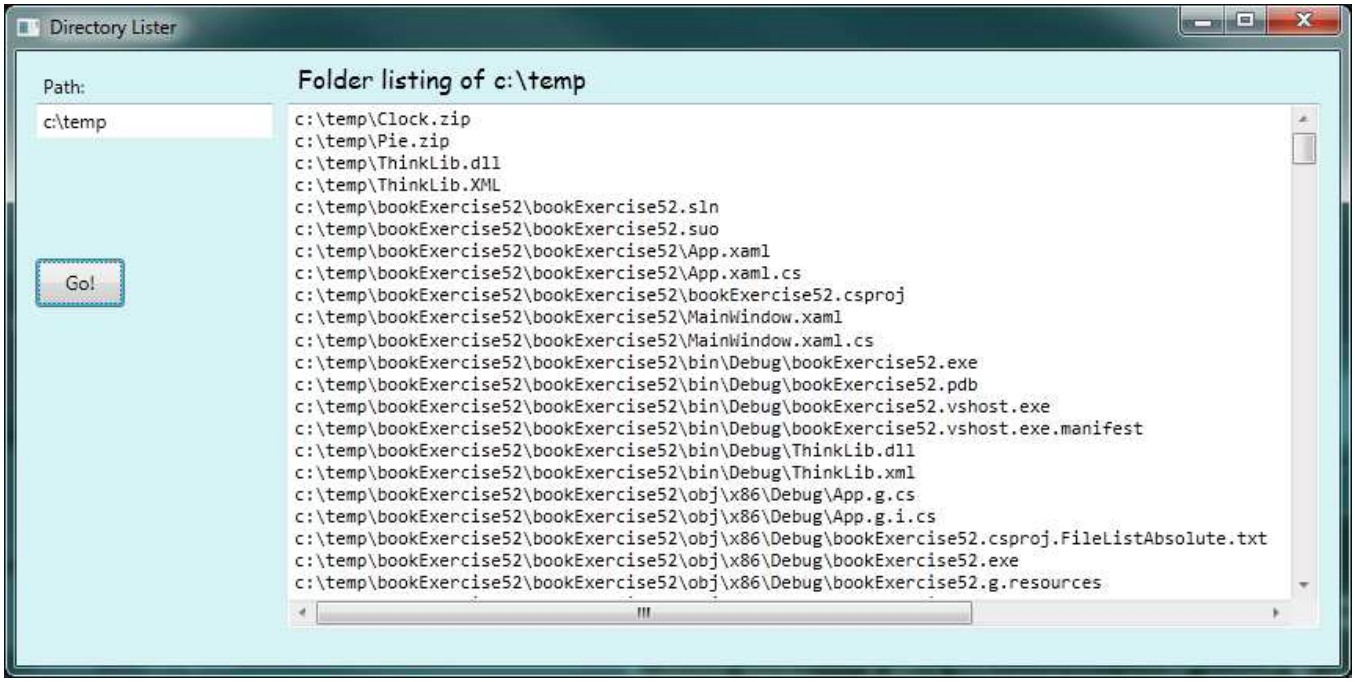
The following program lists all the files in the directory and its sub-directories (and of course, the sub-sub-sub directories ...)

```
1 ...
2 using System.IO;
```

```
3  ...
4
5  private void btnGo_Click(object sender, RoutedEventArgs e)
6  {
7      lblHeading.Content = string.Format("Folder listing of {0}\n", txtPath.Text);
8      txtResults.Clear();
9      showFilesIn(txtPath.Text);
10 }
11
12 private void showFilesIn(string path)
13 {
14     string [] filenames = Directory.GetFiles(path);
15     foreach (string fn in filenames)
16     {
17         txtResult.AppendText(String.Format("{0}\n", fn));
18     }
19
20     string[] subfolders = Directory.GetDirectories(path);
21     foreach (string fldr in subfolders)
22     {
23         showFilesIn(fldr);  // recurse for each subfolder!
24     }
25 }
```

- Line 14 shows how we can get the full pathnames of all the files in a given folder. Line 20 is very similar: it gets the full pathnames of all the sub-directories.
- The interesting recursive call occurs in line 23.

With a suitable font-end GUI we could get this:



Tip:

Recursive structures, such as directories and menu trees, are *much* easier to process if you use a recursive algorithm.

Let's take this example one or two steps further. A nice extension is to only list files matching a certain pattern — let's say only the C# code-behind our GUI files. Those all have names that end in `*.xaml.cs`. One way to do this would be to wrap line 16 in an `if` test. But even easier is that the `GetFiles` method in line 14 has another overloading: one that accepts a pattern, and does the filtering for us. When searching for a string (eg. the file's name) the asterix (*) is a special character which allows us to return all strings starting or ending in a specific pattern as it returns all results which contain any number and variation of characters in the place of the asterix.

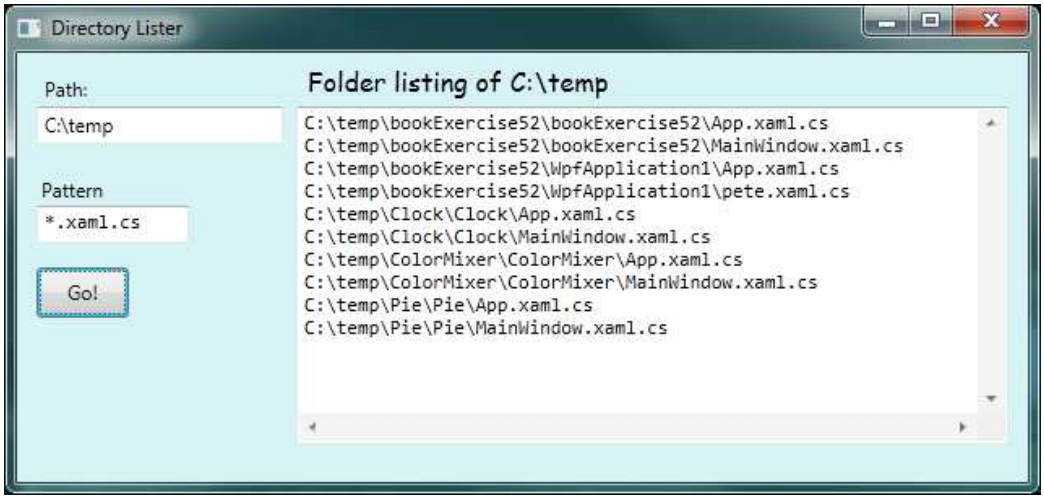
We'll provide a new text box on the GUI to allow the user to enter a pattern, and use this slightly revised code:

```
1  private void btnGo_Click(object sender, RoutedEventArgs e)
2  {
```

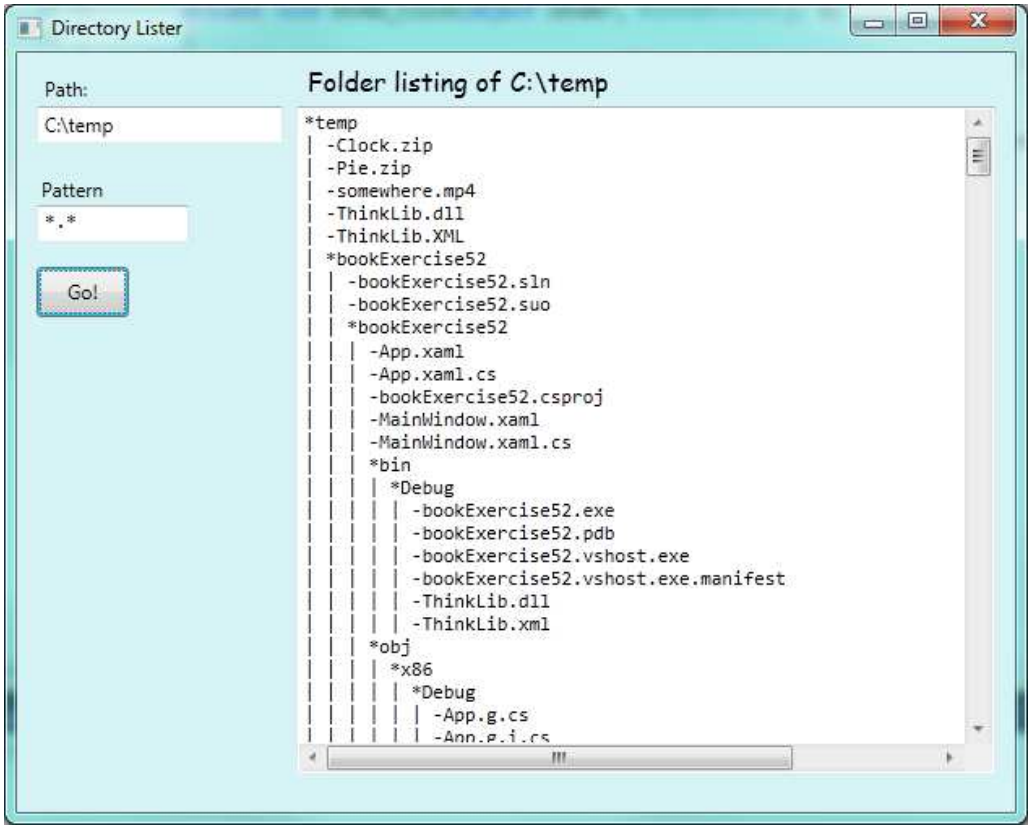
```
3       lblHeading.Content = string.Format("Folder listing of {0}\n", txtPath.Text);
4       txtResults.Clear();
5       showFilesIn(txtPath.Text, txtPattern.Text);
6   }
7
8   private void showFilesIn(string path, string pattern)
9   {
10      foreach (string filename in Directory.GetFiles(path, pattern)) //utilising overload of GetFiles
11      {
12          txtResults.AppendText(string.Format("{0}\n", filename));
13      }
14
15      foreach (string foldername in Directory.GetDirectories(path))
16      {
17          showFilesIn(foldername, pattern);
18      }
19  }
```

Notice the changes to pass the extra filtering pattern into our method, to use the extra pattern in the call to *GetFiles*, and to ensure that when we call `ShowFilesIn` recursively at line 17, we supply the extra pattern argument to the method that must solve our subproblem.

Now our listing looks like this:



In the listings above, each file name is shown with its full path. A more interesting idea is to just show the directory names and file names (without the full path from C:), but to use some simple layout or extra characters to help the user visualize the recursive folder structure. So for the next version of this program, we want to produce output like this:



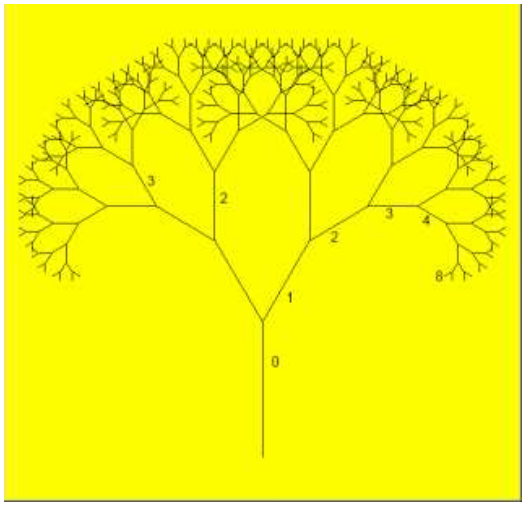
To do this, we will output each line with a prefix string: something like "| | |" that will indicate that we're three levels deep into the structure. In the code below, we start on line 6 with an initial prefix `"| "`. Then, on line 19, each time we recurse down to a deeper level of the directory tree, we extend the existing prefix string.

We also use some existing magic from the built-in libraries on lines 5, 13, and 18 (the `GetFilename` method): this call returns just the file name part of a full file path:

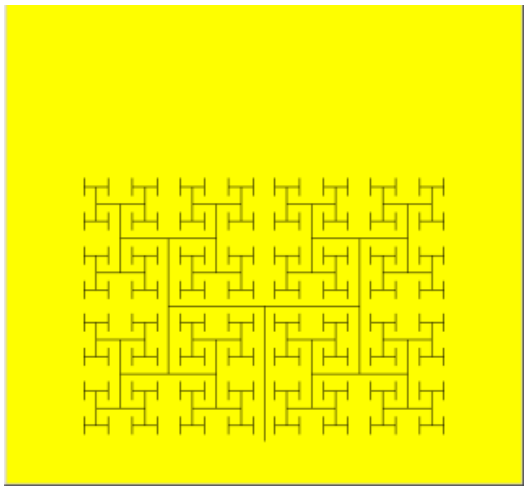
```
1  private void btnGo_Click(object sender, RoutedEventArgs e)
2  {
3      lblHeading.Content = string.Format("Folder listing of {0}\n", txtPath.Text);
4      txtResult.Clear();
5      txtResult.AppendText(string.Format("*{0}\n", Path.GetFileName(txtPath.Text)));
6      showFilesAsTree(txtPath.Text, txtPattern.Text, "| ");
7  }
8
9  private void showFilesAsTree(string path, string pattern, string prefix)
10 {
11     foreach (string filename in Directory.GetFiles(path, pattern))
12     {
13         txtResult.AppendText(string.Format("{0}-{1}\n", prefix, Path.GetFileName(filename)));
14     }
15
16     foreach (string foldername in Directory.GetDirectories(path))
17     {
18         txtResult.AppendText(string.Format("{0}*{1}\n", prefix, Path.GetFileName(foldername)));
19         showFilesAsTree(foldername, pattern, prefix + "| ");
20     }
21 }
```

21.6. An animated fractal using our turtle

Here we have a tree fractal pattern of order 8. We've hand-labelled some of the edges, showing the depth of the recursion at which each edge was drawn.



In the tree above, the angle of deviation from the trunk is 30 degrees. Varying that angle gives other interesting shapes, for example, with the angle at 90 degrees we get this:



An interesting animation occurs if we generate and draw trees very rapidly, each time varying the angle a little. We'll use a timer to generate regular tick events. On each event, we'll clear the previous tree and draw the new tree from scratch. Each tick will also change the angle slightly, to give the animation we desire. Let's begin with the definition for the turtle, and setting up of the timer. (You can cut and paste this code into your programming environment if you're reading the book online.)

```
...
public partial class MainWindow : Window
{
    Turtle tess;
    double theta = 90;

    public MainWindow()
    {
        InitializeComponent();
        tess = new Turtle(playground);
        tess.Visible = false;

        System.Windows.Threading.DispatcherTimer theTimer = new System.Windows.Threading.DispatcherTimer();
        theTimer.Interval = TimeSpan.FromMilliseconds(50);
        theTimer.IsEnabled = true;
        theTimer.Tick += dispatcherTimer_Tick;
    }

    private void dispatcherTimer_Tick(object sender, EventArgs e)
    {
        drawNextTree();
    }
```

The most notable thing here is on line 14: the timer interval times the period between the end of one tick event, and the start of the next. So if we have processor-and-drawing intensive work (which is the case here), this does not translate into a regular tick. We can set the interval to zero, (or leave out line 14 completely), which means "tick as fast as possible". Experiment with some different values for the timer interval.

Let's consider the recursive pattern now for the tree: To draw a tree of order 0, just draw a straight line trunk part, with no further branches or leaves. And when we exit the method, we'll always ensure that the turtle is back where it started, in the same orientation as it was in when the method was called.

For a tree of order greater than 0, we draw the trunk, turn for the new sub-tree, compute the smaller sub-tree size, and recursively draw the sub-tree. We do this for both our left and right sub-tree. So the code comes out like this:

```
1  const double trunk_ratio = 0.29;    // The trunk length is 29% of the tree size.
2
3  public void drawTree(int order, double theta, double treeSize)
4  {
5      double trunkSize = treeSize * trunk_ratio;  // compute length of trunk
6      tess.Forward(trunkSize);                    // always draw the trunk
7
8      if (order > 0)                              // must we also draw subtrees?
9      {
10         double branchSize = treeSize - trunkSize;
11         tess.Left(theta);
12         drawTree(order - 1, theta, branchSize);
13         tess.Right(2 * theta);
14         drawTree(order - 1, theta, branchSize);
15         tess.Left(theta);
16     }
17
18     tess.Forward(-trunkSize);    // make sure we end up back where we started.
19 }
```

- Lines 11,13 and 15 ensure that whatever heading the turtle started at will be its final heading too.
- Line 18 puts the turtle back where it began, by "undoing" the movement from line 6.
- The all-important recursive calls on line 12 and 14 decrease the order at the next level, and decrease the overall remaining size to be drawn.

So what is left: we need to handle the timer tick, clear the old drawing, draw the new tree, and change our class-level variable `theta`. So the final piece of the puzzle is this:

```
1  private void drawNextTree()
2  {
3      tess.Clear();
4      tess.BatchSize = 0;
5      tess.WarpTo(playground.ActualWidth / 2, playground.ActualHeight - 10);
6      tess.Heading = -90;
7      drawTree(7, theta, playground.ActualHeight - 10);
8
9      theta += 5;
10 }
```

- Lines 5 and 7 show that we can make the tree size and the starting position for the turtle depend on the playground size. If your playground stretches as your window resizes, the tree will grow bigger or smaller.
- Line 4 needs some explanation. When the turtle "draws" it constructs new WPF controls - mainly line segments. In it's default setting, the turtle forces our screen to refresh after each new line segment is created. This one-at-a-time refreshing can be very slow if there are many line segments in our drawing. Setting the batch size of the turtle to zero means "do not refresh the screen until all the current computation finishes". It means we'll show the whole tree at once, and things will be much faster. Try commenting out line 4 and see the difference.

21.7. Glossary

base case

A branch of the conditional statement in a recursive method that does not give rise to further recursive calls.

infinite recursion

A method that calls itself recursively without ever reaching any base case. Eventually, infinite recursion causes a runtime error.

recursion

The process of calling a method that is already executing.

recursive call

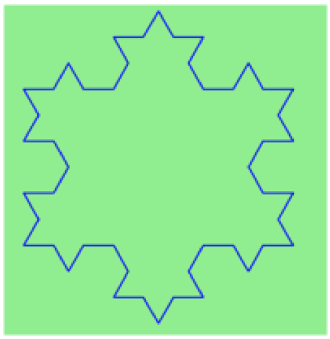
The statement that calls an already executing method. Recursion can also be indirect — method *f* can call *g* which calls *h*, and *h* could make a call back to *f*.

recursive definition

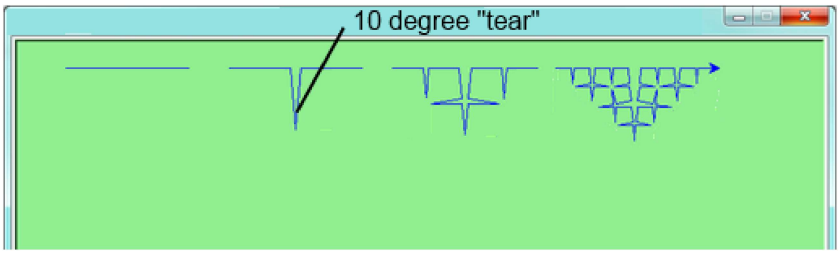
A definition which defines something in terms of itself. To be useful it must include *base cases* which are not recursive. In this way it differs from a *circular definition*. Recursive definitions often provide an elegant way to express complex data structures, like a directory that can contain other directories, or a menu that can contain other menus.

21.8. Exercises

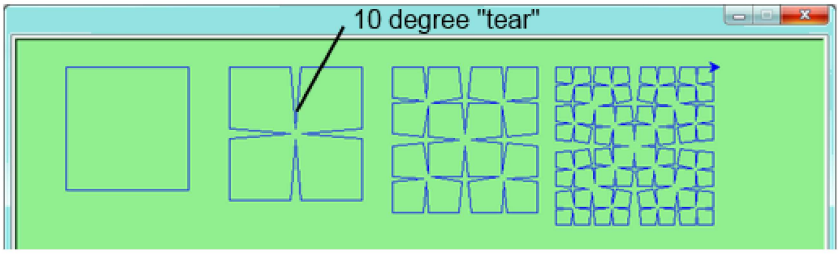
1. Change a small recursive program from this chapter so that the base case test is never satisfied. Run it and see what happens. Can you make sense of the error message, and recognize it in future?
2. Modify the Koch fractal program so that it draws a Koch snowflake, like this:



3. a. Draw a Cesaro torn line fractal, of the order given by the user. We show four different lines of orders 0,1,2,3. In this example, the angle of the tear is 10 degrees.



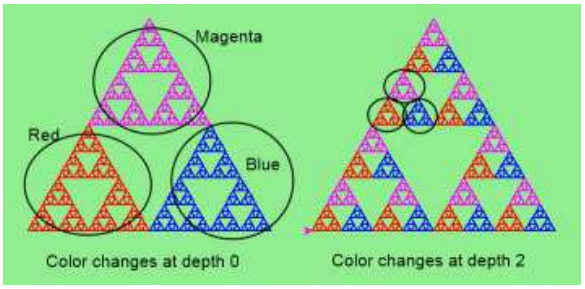
b. Four lines make a square. Use the code in part a) to draw Cesaro squares. Varying the angle gives interesting effects — experiment a bit, or perhaps let the user input the angle of the tear.



c. (For the mathematically inclined). In the squares shown here, the higher-order drawings become a little larger. (Look at the bottom lines of each square - they're not aligned.) This is because we just halved the drawn part of the line for each recursive sub-problem. So we've "grown" the overall square by the width of the tear(s). Can you solve the geometry problem so that the total size of the sub-problem case (including the tear) remains exactly the same size as the original?

4. A Sierpinski triangle of order 0 is an equilateral triangle. An order 1 triangle can be drawn by drawing 3 smaller triangles (shown slightly disconnected here, just to help our understanding). Higher order 2 and 3 triangles are also shown. Draw Sierpinski triangles of any order input by the user.

5. Adapt the above program to change the brush colour of its three sub-triangles at some depth of recursion. The illustration below shows two cases: on the left, the colour is changed at depth 0 (the outermost level of recursion), on the right, at depth 2. If the user supplies a negative depth, the colour never changes. (Hint: add a new optional parameter `colourChangeDepth` (which defaults to -1), and make this one smaller on each recursive sub-call. Then, in the section of code before you recurse, test whether the parameter is zero, and change color.)



6. Rewrite the Fibonacci algorithm without using recursion. Make use of the BigInteger type that was introduced in the `factorial` section. Find `fib(200)`.
7. 10! has two trailing zeros. 20! has four trailing zeros. But 30! has 7 trailing zeros, which is not what you might have guessed! Explain where these trailing zeros are coming from, and what is special about the range between 20! and 30! (Hint: Count and display the trailing zeros to allow you to easily investigate the problem.)
8. Modify our program that traverses a directory structure. Instead of outputting file names, it returns a list of all the full paths of files in the directory or the subdirectories. (Don't include directories in this list — just the files.) For example, the output list might have elements like this:

```
["C:\temp\Clock.zip", "C:\temp\Pie.zip", "C:\temp\ThinkLib.dll",  ... ]
```

9. Write a method named `litter` that creates an empty file named `trash.txt` in each subdirectory of a given directory tree. *Hint:* Use `File.Create` to create a file.
10. Now write a method named `cleanup` that removes all the litter files.

    Hint 1: Use the directory lister program from this chapter as a basis for these recursive programs. Because you're going to destroy files on your disks, you had better get this right, or you risk losing files you care about. So excellent advice is that initially you should fake the deletion of the files — just output the full path names of each file that you intend to delete. Once you're happy that your logic is correct, and you can see that you're not deleting the wrong things, you can put the real code in place.

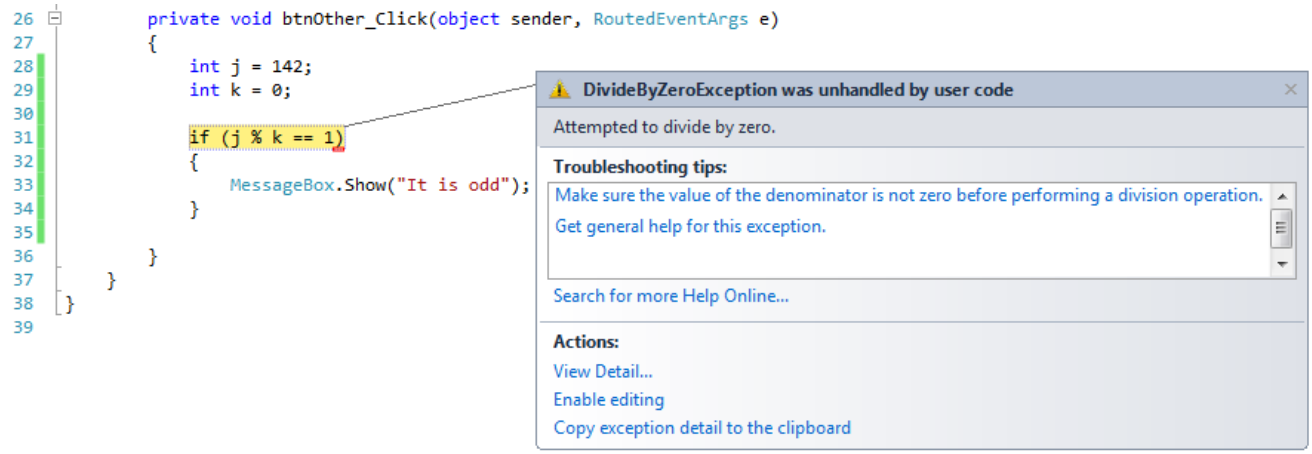
    Hint 2: Use `File.Delete` to remove a file.

22. Exceptions

22.1. Catching exceptions

Whenever a runtime error occurs, the system **throws** an **exception**. When this happens, an `Exception` object is created. It holds details about what went wrong, and where. We use the word "throw" to describe what happens, as one might say *"I'm throwing this problem at my boss"*, or *"I'm throwing this problem over the fence"*.

Unless we take some special action to **catch** and handle the problem, the program will *crash* at this point. For example, division by zero will throw an exception. So will an array or list index out of bounds, or an attempt to use a file that does not exist.



The key to what we're going to do here is given by the title bar of the exception window in this image: the exception *"was unhandled by user code"*. We're about to see how we can handle it.

In the picture above, also on the title line, we're also told what type of exception was thrown, in this case a `DivideByZeroException`.

Sometimes we want to execute an operation that might throw an exception, but we don't want the program to crash. We can handle the exception using the `try ... catch` statement to "wrap" a region of code.

For example, we might ask our user to enter a name of a file and then we'll try to read from it. If the file doesn't exist, we want to catch and handle the exception:

```
1  string filename = textBox1.Text;
2  try
3  {
4      string content = File.ReadAllText(filename);
5      /// do other useful things with the content
6      /// ...
7  }
8  catch (Exception ex)
9  {
10     string msg = string.Format("Reading file {0} threw an exception: \n{1}",
11           filename, ex);
12     MessageBox.Show(msg);
13 }
14 ...
```

The `try ... catch` statement consists of a try block followed by one or more catch clauses, which specify handlers for different types of exceptions. Each catch clause has an associated block of code.

`try` executes and monitors the statements in the first block (lines 3–7). If no exception is thrown, it skips any `catch` clauses and continues execution (at line 14 in this example). If any exception occurs, information about the error is packed into a new exception object, and the exception is thrown. In this example, the exception is caught at line 8,

and is assigned to variable `ex`. Control immediately jumps into the `catch` block, where we execute lines 10–12. When the block completes, execution continues at line 14 in this example.

Within the catch block, object `ex` is like any other parameter we've encountered: it has a type, it has some properties, and it has some methods, and you can choose your own variable name for it. If you're inquisitive about what else is inside an exception object, set a breakpoint at line 10 and use the debugging tools to inspect it.

22.2. We can catch exceptions that occur in methods we've called

As our software becomes more complex we attempt to manage the complexity by breaking it into methods (and coming soon) separate objects, and perhaps even separate libraries.

Suppose your method `P` has a `try ... catch` statement to catch exceptions. `P` calls method `Q` which calls `W`. But `W` throws a `FileNotFoundException`. What happens next?

If `W` does not catch the exception, it exits immediately, and the exception is thrown up to its caller `Q`. If `Q` can't catch it, the same happens again, with the exception thrown up to `P`. Now because `P` has a `catch`, it can recover.

We call the process of exiting all the called methods until we find one that is willing to catch our exception *unwinding the call stack*.

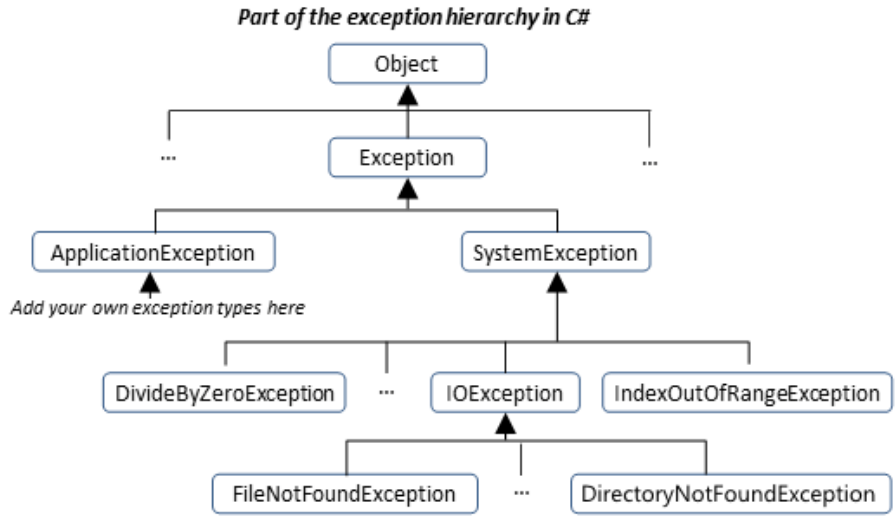
So the important idea is that a `try ... catch` statement can catch exceptions that occur, even if they occur indirectly in other methods that are called from the wrapped statements.

We'll need some care, though. In the example above we caught the exception at line 8, with an assumption that it was line 4 that caused the fault. But in fact lines 5 or 6, or any methods called by them could be our culprit.

22.3. The Exception Hierarchy

In Section 13.4 we introduced the idea that classes, or types, were organized hierarchically via a system of inheritance. We saw that an `Ellipse` object is-a kind of `Shape` object which is-a kind of `UIElement`.

Exception types are also organized hierarchically. Here is a small fragment diagram showing just a few possible types of exception.



A `catch` can only catch certain kinds of exception. In line 8 in our example, we offered to catch anything of type `Exception`. That is the most general type of exception, and because `FileNotFoundException` is-a `Exception`, our `catch` will catch it.

But you can (and are generally encouraged) to catch specific exception types. Consider this fragment of code:

```
1 try
2 {
```

```
3       ...
4   }
5   catch (FileNotFoundException ex1)
6   {  ... }
7   catch (DirectoryNotFoundException ex2)
8   {  ... }
9   catch (IOException ex3)
10  {  ... }
```

Here we're offering to catch three different kinds of exceptions (so our `try` can have multiple `catch` clauses), but we wouldn't catch a `DivideByZeroException` if one occurred, either on line 3, or in some method called indirectly by line 3.

Notice too that line 9 catches `IOException`. Although `FileNotFoundException` is-a `IOException`, it won't be caught at line 9 — it will get caught instead at line 5. The `catch` clauses are examined in the order that we write them, top-to-bottom.

The very readable Microsoft documentation at http://msdn.microsoft.com/en-us/library/vstudio/0yd65esw.aspx says

> It is possible to use more than one specific catch clause in the same `try ... catch` statement. In this case, the order of the catch clauses is important because the catch clauses are examined in order. Catch the more specific exceptions before the less specific ones. The compiler produces an error if you order your catch blocks so that a later block can never be reached.

A good guideline is to only catch the exceptions that you can recover from. For example, if your user is entering the name of the file to open, they're going to get it wrong sometimes. Perhaps catch `FileNotFoundException`, or the more general `IOException`.

There is a trade-off, though. If you catch the general exception you might be able to sidestep a wider range of problems. For example we might try to open a file that is already in use by another program, or perhaps the file is on a network drive and we have a timeout because the network fails temporarily. If you catch the general exception, you'll be able to recover from all these cases, however, you won't be able to react uniquely to each cause of the error.

22.4. Throwing our own exceptions

So far, run time errors throw exceptions. But there doesn't always have to be an error. Our program can deliberately use a `throw` statement to throw an exceptions. Here is a method that expects a non-empty `List<int>` from the caller, and it finds the maximum:

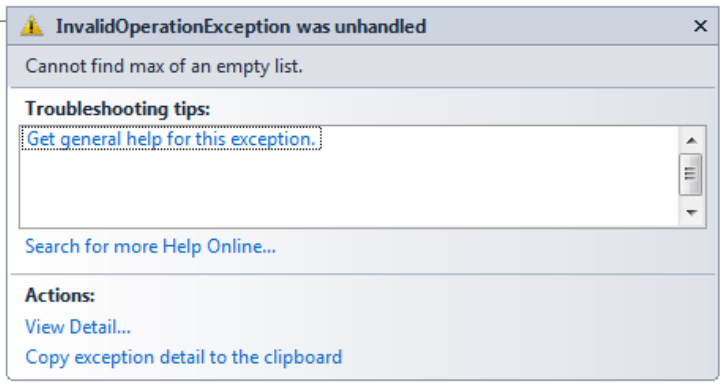
```
1   private int findMax(List<int> xs)
2   {
3       if (xs.Count == 0)
4       {
5           Exception myBad = new InvalidOperationException("Cannot find max of an empty list.");
6           throw myBad;
7       }
8       int result = xs[0];
9       for (int i = 0; i < xs.Count; i++)
10      {
11          if (xs[i] > result)
12          {
13              result = xs[i];
14          }
15      }
16      return result;
17  }
```

Line 5 creates an exception object which encapsulates specific information about the error, and line 6 throws the exception.

`InvalidOperationException` is one of the built-in exception types that seem to most closely match the kind of error we want to raise.

If the method that called `findMax` (or its caller, or their caller, ...) handles the error, then the program can carry on running; otherwise the program crashes.

```
189         private int findMax(List<int> xs)
190         {
191             if (xs.Count < 1)
192             {
193                 Exception myBad = new InvalidOperationException("Cannot find max of an empty list.");
194                 throw myBad;
195             }
196             int result = xs[0];
197             for (int i = 0; i < xs.Count; i++)
198             {
199                 if (xs[i] > result)
200                 {
201                     result = xs[i];
202                 }
203             }
204             return result;
205         }
206
207
208
209
```



The error message includes the exception type and the additional information that was provided when the exception object was first created.

It is often the case that lines 193 and 194 (creating the exception object, then raising the exception) are combined into a single statement, but there are really two different and independent things happening, so perhaps it makes sense to keep the two steps separate when we first learn to work with exceptions. Here we show it all in a single statement:

```
1  throw new InvalidOperationException("Cannot find max of an empty list.");
```

22.5. Glossary

catch an exception

Instead of the default behaviour of having our program crash, we can catch and recover from exceptions by wrapping regions of code in a `try` ... `catch` statement.

exception

An error that occurs at runtime. We say the exception is *thrown*.

exception hierarchy

Different types of exceptions are already defined in C#. The types of exceptions are organized into a hierarchical (tree-like) type structure, so, for example, an `FileNotFoundException` is-a kind of `IOException` which is-a kind of `SystemException`, which is-a kind of `Exception`.

is-a

A made-up word that Computer Scientists use to describe a relationship between a more specific type and a more general type. A Toyota is-a car.

`throw` statement

A C# statement that allows us to throw exceptions from our code.

unwinding the call stack

The process of exiting all the called methods until we find one that is willing to catch our exception. If the method which throws an error is unable to handle it we will exit that method and return to the call-site to see

if the caller is able to catch the exception. If it cannot, we will continue to exit called methods until one is found which can handle the exception or the pogram crashes.

22.6. Exercises

1. Create a GUI with a textbox for entering a month number. The number is valid if it is an integer between 1 and 12. Convert what the user enters to an integer, and validate that the number is between 1 and 12. If not, pop up a MessageBox that describes the problem, and ask the user to try again. Catch any cases where the user enters invalid characters, floating point numbers, leaves the text box blank, etc.
2. Your car can break down in various ways. Make a list of at least 15 different ways (e.g. flat tyre, no fuel, door fell off, battery flat, won't start, license expired, crashed into lamp pole, repossessed by bank, etc.) Google may be able to help if you ask the right question.

   Now organize these problems into a hierarchy of `BreakDown` exceptions. Draw a tree diagram like the one in this chapter to show these relationships.

23. The .NET Framework

The .NET framework is a software framework developed by Microsoft. It has two main parts:

- A large collection of pre-written classes called the *.NET Framework Class Library* (see http://msdn.microsoft.com/en-us/library/gg145045.aspx) that we can use when we write our programs.
- A component called the Common Language Runtime (CLR). This is a software environment (as opposed to a hardware environment) that runs our programs for us. It provides services like debugging, exception handling, security, and management of memory for our programs.

  As the "common" part of the name suggests, the CLR can run many different languages, not just C#. So we could write our programs in languages like Visual Basic, C#, F#, C++, Haskell, Ruby (or about 50 others). We can even write programs that mix components from different languages.

  They all share the same Framework Class Library, so learning a bit about this library here will be useful when we move to another .NET programming language in future.

Just how big is this library?

If we believe some of the claims on the Internet, more than 10000 classes which provide more than 300000 methods that we can call!

Other mainstream languages like Java and Python also have huge libraries. These libraries help for all kinds of scenarios — accessing the network, working with files, doing mathematical calculations, organizing collections, encryption, writing web servers, and so on.

When we see adverts that want to hire "experienced C# programmers" or "experienced Java programmers", it really means "knows the language, and also knows their way around some of the important libraries".

We've used many of the libraries already in this book: every time we have a `using` directive we're telling the compiler that we want to use some or other functionality from a library.

So for the rest of this chapter we'll just pick a few small samples, we'll show how to use them, and we'll discuss a couple of new C# language features that are related.

23.1. Random numbers

Although we've seen them before in the book, we'll cover the Random class again. Here are a few typical uses of random numbers:

- To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin,
- To shuffle a deck of playing cards randomly,
- To allow/make an enemy spaceship appear at a random location and start shooting at the player,
- To simulate possible rainfall when we make a computerized model for estimating the environmental impact of building a dam,
- For encrypting banking sessions on the Internet.

C# provides a class `Random` that helps with tasks like this. We can look it up using help, but here are the key things we'll do with it:

```
1  using System;
2  using System.Windows;
3
4  namespace Fragments
5  {
6      public partial class RandomDemoGUI : Window
7      {
8          Random myRandomSource;
9
10         public RandomDemoGUI()
11         {
12             InitializeComponent();
13             myRandomSource = new Random();
```

```
14          }
15
16          private void btnRandom_Click(object sender, RoutedEventArgs e)
17          {
18              // Pick two (different) random cards from a deck of cards numbered 0 to 51.
19
20              int card1 = myRandomSource.Next(52);
21              int card2 = myRandomSource.Next(52);
22              while (card2 == card1)  // oops. try again till we get a different second card.
23              {
24                  card2 = myRandomSource.Next(52);
25              }
26              txtResult.AppendText(string.Format("The two cards are {0} and {1}\n", card1, card2));
27              txtResult.ScrollToEnd();
28          }
29      }
30  }
```

Line 8 defines a variable that can reference a Random number generator object. Line 13 instantiates the object, and makes our variable reference it. Then in lines 20, 21 and 24 we call a method of the object.

The `Next` call randomly picks an integer less than the upper bound argument. All the values have an equal probability of occurring (i.e. the results are *uniformly* distributed).

The `Next` method has another overloading that allows us to specify both a lower bound and an upper bound. The lower bound is inclusive (the generator might pick it), while the upper bound is exclusive. Of we don't supply a lower bound, 0 is used. So if we want to throw two dice, and the outcome for each needs to be an integer between 1 and 6 (inclusive), we could do it in either of two ways:

```
1      int d1 = myRandomSource.Next(1, 7);
2      int d2 = myRandomSource.Next(6) + 1;
```

The `NextDouble` method returns a floating point number in the interval [0.0, 1.0) — the square bracket means "closed interval on the left" and the round parenthesis means "open interval on the right". In other words, 0.0 is possible, but all returned numbers will be strictly less than 1.0. It is usual to *scale* the results after calling this method, to get them into an interval suitable for our application. In the case shown below, we generate a uniformly distributed double random number in the range [20, 27.5): (Hey, that's quite a mouthful of dense terminology — *"uniformly distributed double random number"*. So it says a lot!)

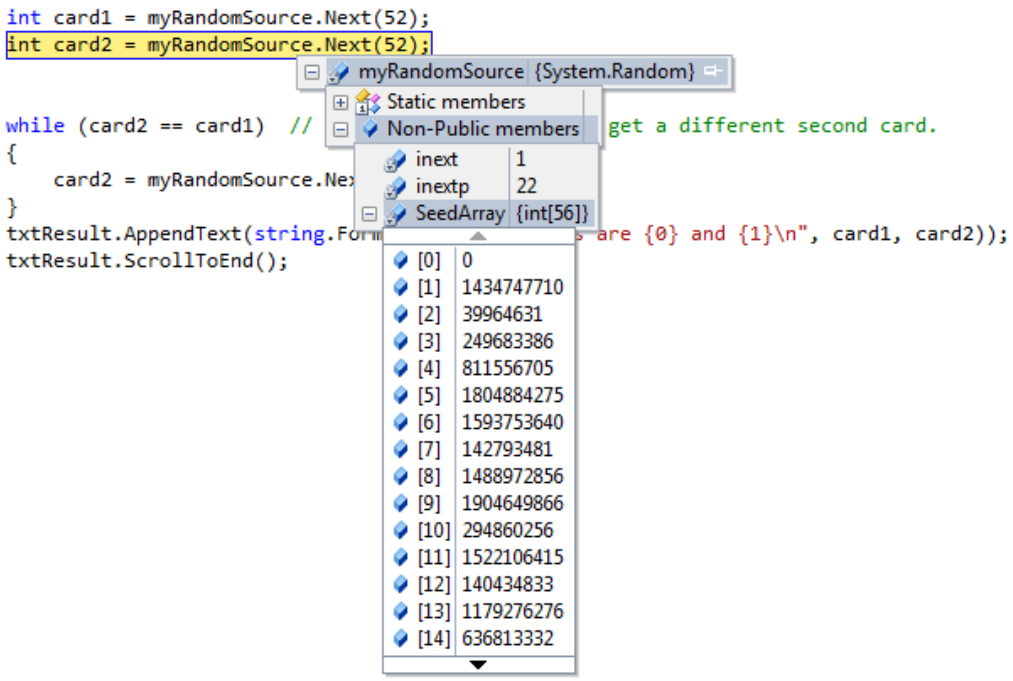
```
1  double num = myRandomSource.NextDouble() * 7.5 + 20;
```

Where did the 7.5 and the 20 come from? Well the size of the range is 27.5–20, or 7.5. So once we've generated the random number in the range [0.0, 1.0) we scale it by multiplication. (We might like to think of scaling as stretching the number line like a rubber band.) That will leave us with a random number in the range [0.0, 7.5). Then we'll add 20 to give us a number in the range we want.

23.2. Repeatability and Testing

Random number generators are based on a **deterministic** algorithm — repeatable and predictable. So they're called **pseudo-random** generators — they are not genuinely random, like flipping a coin or throwing some dice. They start with a *seed* value. Each time we ask for another random number, we'll get one based on the current seed, and the seed itself will be updated for next time.

If we're interested, we can put a breakpoint in our code and inspect the internal state of the generator. It uses an array of 56 integers — or 1792 bits — as its seed.



For debugging and for writing unit tests, it is convenient to have repeatability — programs that do the same thing every time they are run. We can arrange this by forcing the random number generator to be initialized with a known seed every time. (Often this is only wanted during testing — playing a game of cards where the shuffled deck was always in the same order as last time we played would quickly become boring!)

```
1  myRandomSource = new Random(42);  // Create a generator with known starting seed
```

This alternative way of instantiating a random number generator provides an explicit starting seed. Without this argument, the system probably uses something based on the time. So with a fixed starting seed, grabbing some random numbers from `myRandomSource` today will give us precisely the same random sequence as it will tomorrow! Try it!

23.3. Access Modifiers — `public`, `private`, `internal` and `protected`

As we start combining libraries and building bigger programs we require some control over "which parts of the program can use which methods or properties". For example, look above at where we inspected the state of `myRandomSource`. It holds some state in an array called `SeedArray`.

Even though this array exists in the object, there is no way for our program to either read or alter the elements of the array. This is because the array is defined with an **access modifier** `private`.

Any **member** of a class (a member could be a variable, a method, a type, or some other things) in a class that is tagged as `private` means "cannot be used from outside this class". Methods within the class can use other private methods, or private variables and types.

The opposite of `private` access is `public` access. A member that is tagged as public can be called and used from other code in other classes.

When our programs get bigger we find that the "all-or-nothing" distinction that `public` and `private` gives us sometimes needs to be more refined. `internal` and `protected` access modifiers give us these in-between capabilities.

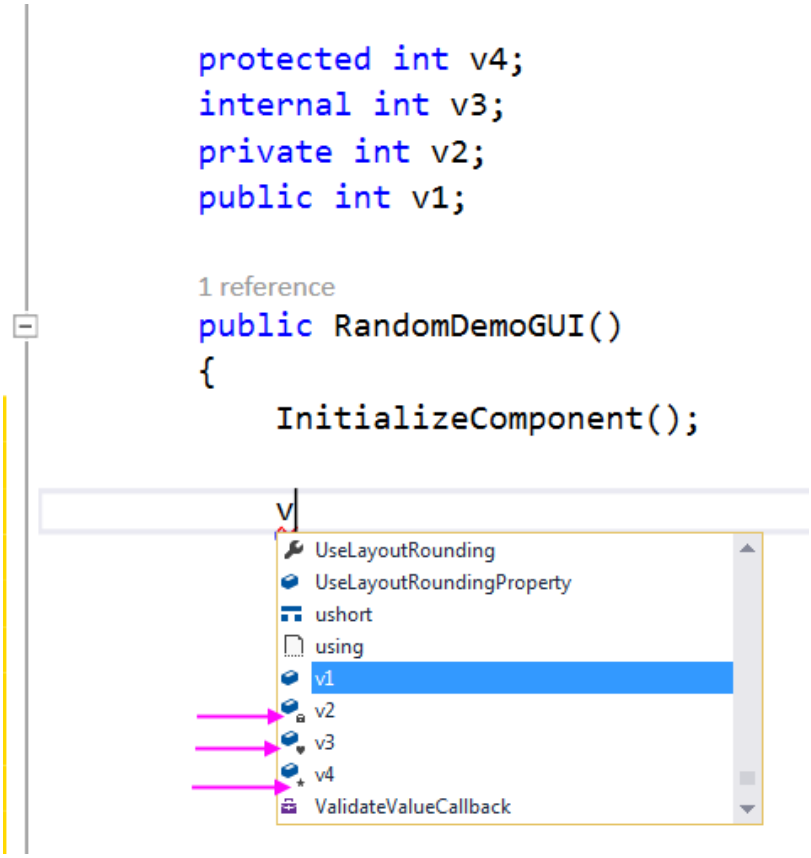
`internal` access means accessible to the class and any other class that is part of the same project, but not accessible to code in other projects.

`protected` access means accessible to the class and any children of the class, but otherwise private. We've seen some hierarchical organization of GUI types and Exception types, and we've talked about the `is-a` relationship that denotes a

child-parent relationship between types. So protected access means “available to my descendants”.

What happens if we are lazy and don’t put any access modifier on our methods or variable definitions within a class? The default assumption in that case is the “safest” one — they’ll be private. (The rule is modified when something is declared in a namespace: if we don’t put an access modifier on our class definition, it becomes internal rather than public.)

Keep an eye on IntelliSense when it pops up: if a member is not public, it puts an extra little icon into the IntelliSense: a padlock for private, a heart (why a heart?, I hear you ask) and a star for protected. So here is a simple experiment that defines four variables with four different access modifiers, and seeing what we get from IntelliSense:



Let us now revisit the sample program at the top of this chapter. Lines 6 and 10 are the only public modifiers. Line 6 defines a new kind of Window called `RandomDemoGUI`. It is the name I chose for my GUI. Our GUI window is-a Window, and it is public. So any other piece of code can see this new class. Line 10 has the code for the *constructor* — the code that gets executed whenever we instantiate a new instance of this type of window. It is also public, so what that means is “outsiders can create new instances of RandomDemoGUI windows.”

But everything else in the class is private (including the variable `myRandomSource`, because we never gave it any access modifier).

23.4. Namespaces

To help organize the many classes and names in the *.NET Framework Class Library*, and in our own software, they are grouped into **namespaces**. For example, the `System` namespace contains a number of classes: we’ve seen the `System.Math` class, the `System.Random` class and there are also others, like `System.GC`. When we use a dot like this it means “the `Random` class in the `System` namespace. We say the name `Random` is *qualified*.

Namespaces can probably be thought of somewhat like a folder system on a disk. If files are in different directories, we can have many different files called `ReadMe.txt`. Similarly, in C#, we can have many different classes with the same name, as long as we keep them in separate namespaces.

When we put `using` directives in our code (line 1 of the sample program at the top of this chapter), it tells the compiler to automatically search those namespaces. So when we use a “shorthand” type name like `Random` or `Turtle`, or `File`, the compiler will search in the namespaces and know that we mean `System.Random`, `ThinkLib.Turtle`, or `System.IO.File`.

Line 4 of our program (the one at the start of this chapter) says that our new class, `RandomDemoGUI`, is defined in a namespace called `Fragments`. The namespace is often set to be the same as the name of the project we choose. So in my case, I asked Visual Studio to create a new WPF project called `Fragments`. It then chose the namespace `Fragments` for me.

We can comment out the `using` directives in our program, one at a time, and then fix any errors we get by fully qualifying the places where we use names from the namespace. It should convince us that the `using` directive is not an essential feature of C# — it is just a convenient mechanism. We sometimes call features that provide convenience (but no new functionality) **syntactic sugar**.

Namespaces can contain other namespaces, (just as directories can contain other directories). So on line 2 of our program we use the `System.Windows` namespace. This means that `Windows` is a namespace within the `System` namespace. There are also more deeply nested namespaces, like `System.Net.Security`, `System.Windows.Media.Imaging` and `System.Windows.Shapes`.

23.5. The `partial` keyword

There is one other special keyword in our program that deserves an explanation: On line 6 we have the word `partial` attached to our class. It means that there can be other code elsewhere (that we don't necessarily get to see) that is also part of this same class. In WPF programs, some of the code is generated from our window's XAML which describes our GUI. It is ugly, but if we want to take a peek and see the "helper code" that the compiler wrote for us behind our back, we can start at our project folder and look for any `.cs` files in the sub-folder `obj\x86\Debug`.

23.6. You call it "Int32", but we call it "int"

Because the .Net Framework Class Library serves many languages, not only C#, its classes and values have names that may not always be the same as the names we prefer to use in C#, or in Visual Basic, or F#. We've seen a few examples already:

```
1  int x = Convert.ToInt32(TxtSize.Text);
2  string y = string.Format("{0}", isPrime(17));
3  MessageBox.Show(y);
4  Int32 k = 42;
```

- The framework has a type called `System.Int32`. In C#, we call that same type `int`. If we have a `using System;` directive at the top of our code, we can use `Int32` without having to fully qualify it, as shown on line 4 above.
- The framework type `System.String` with a capital letter corresponds to the C# type `string`. So `String` works in our C# programs too.
- The framework type `System.Boolean` is an alias for the C# type `bool`. In line 2, the framework also formats its Boolean values as "False" and "True" with capital letters, whereas in C# we prefer the values `false` and `true`.
- `System.Object` and the C# `object` are aliases.

So be aware that occasionally we might see the framework spelling of a type name, or the framework representation of a value, rather than the preferred C# spelling. In line 4 above, if we set a breakpoint and inspect `k`, we'll notice that we're told that it is an ordinary `int`, the same as `x`. This is further confirmation that the framework names and the C# names are often aliases of each other.

23.7. Glossary

access modifier

In C#, one of four keywords: `public`, `private`, `internal` and `protected` that determine how other parts of the software may access (or may be restricted from accessing) a member.

deterministic

Something that is repeatable, and will always produce the same results for the same inputs. See also non-deterministic.

member

A member of a class is something that belongs to the class: e.g. variables, methods, properties, constructors, constants, and more. (See http://msdn.microsoft.com/en-us/library/vstudio/ms173113.aspx if you're inquisitive about other kinds of members.)

namespace

A grouping for names that helps organize software. Namespaces mean that that the same common names can be used in different namespaces without ambiguity. (We probably have plenty of different classes called `MainWindow` if we're doing all these textbook exercises.)

non-deterministic

Something that has an element of chance built into in. One is not sure when we flip a coin whether it will land on heads or tails. We use random number generators in programs to provide (fake) non-determinism.

pseudo-random

Something that appears to be random, but isn't really random. Our random number generators are based on a deterministic algorithm and some seed value. If you know the seed, you know what the next random number will be. Since random numbers are used for setting up secure banking sessions and for encryption, one popular attack to try to break security is to see if one can guess what values the random number generator is going to produce next.

Can real randomness ever exist?

For those who enjoy physics, our universe is really random — no matter what we know about the state of things, we cannot predict with certainty what happens next.

About 120 years ago (with Newton's laws) we thought that if we knew what every atom was doing, we could (in theory) figure out exactly what the final state of everything would be.

But Quantum Theory messed up that nice fairy-tale idea. Quantum theory is all about our "genuinely non-deterministic" universe.

qualified name

A name that is prefixed by some contextual information, e.g. `System.Random`, `Math.Sqrt`, or `System.IO.File`.

seed

A value which a pseudo-random number generator uses to initialize its algorithm.

syntactic sugar

Something that makes it sweeter or easier to express our intentions, but it doesn't allow us to do anything fundamentally new. For example, a `switch` statement in C# is really just syntactic sugar, because we could do the same thing with `if` statements.

`using` directive

A line at the top of a file that makes all the names in that namespace available without the need to qualify them.

23.8. Exercises

1. The `Random` class has a nasty surprise if you instantiate two instances in quick succession. Read the "Remarks" section of the help documentation for `Random`, understand why there is a problem, and see if you can replicate their evidence.

24. Scope and Lifetime

24.1. Scope

The **scope** of a name is the region of program code in which a name can be used without the need for extra qualification.

There are two important scopes in C#:

- **Local scope** refers to names defined within a method or a within a block within a method.
- **Class-level scope** refers to all the names defined within the current class (but this excludes any local names that are defined inside methods of the class.)

Let's look at the `RandomDemoGUI` program of the previous chapter again. The class `RandomDemoGUI` extends from line 6 to line 29.

```
1  using System;
2  using System.Windows;
3
4  namespace Fragments
5  {
6      public partial class RandomDemoGUI : Window
7      {
8          Random myRandomSource;
9
10         public RandomDemoGUI()
11         {
12             InitializeComponent();
13             myRandomSource = new Random();
14         }
15
16         private void btnRandom_Click(object sender, RoutedEventArgs e)
17         {
18             // Pick two (different) random cards from a deck of cards numbered 0 to 51.
19
20             int card1 = myRandomSource.Next(52);
21             int card2 = myRandomSource.Next(52);
22             while (card2 == card1)  // oops. try again till we get a different second card.
23             {
24                 card2 = myRandomSource.Next(52);
25             }
26             txtResult.AppendText(string.Format("The two cards are {0} and {1}\n", card1, card2));
27             txtResult.ScrollToEnd();
28         }
29     }
30 }
```

Within the class there are 3 class-level definitions: one variable called `myRandomSource` is defined on line 8, and two methods are defined on lines 10 and 16 (one of these methods is the constructor). These are the class-level scope names.

In addition, we have two local variables defined on lines 20 and 21. They are local to the `btnRandom_Click` method. On line 16, that method also takes two parameters with names `sender` and `e`. Parameters of a method are in the local scope of the method in which they are defined.

What this means is that the four names defined in the `btnRandom_Click` method (i.e. `sender`, `e`, `card1`, `card2`) can only be used within the method where they're defined. They're local. You would get an error if you attempted to assign something to `card1` at line 13, for example.

But class-level names can be accessed from anywhere in the class, including from inside methods of the class. So at lines 13, 20, 21 and 24 we can use `myRandomSource`.

Now what would happen if we moved the definition at line 8 into the constructor method at line 10? It would become a local definition within the constructor, rather than a class-level definition. So within the constructor we'd still be able

to access it, but lines 20, 21 and 24 would give errors: *The name 'myRandomSource' does not exist in the current context.*

24.2. Scopes can nest inside each other

When we said "local scope" or "class-level scope" we made a bit of a simplification. It is slightly more complicated that that. Although we've not seen examples yet, one can define one class inside another class, so we could nest classes very deeply. (We do not have any use for so-called *inner-classes*, or nested classes, in this book, but we should know that such things can exist!)

Additionally, C# allows us to define variables inside a block of code (recall that a block is a group of statements enclosed in braces). The scope rule is then that the name is only visible within the block where it has been defined.

In another twist of convenience, C# also allows us to define variables as part of a `for` or `foreach` statement. In this case the scope of the defined variable is the statement and its loop body.

With classes or scopes that are nested within each other, there is a potential ambiguity: if we have a class-level variable called `num`, and we also call one of our parameters to a method `num`, then we need to be clear about which `num` we're referring to when we use it!

The scope lookup rules are that the most closely nested scope where `num` is defined is the one that will be used. Let's look at this very contrived example:

```
1  double v1 = 10.5;
2  int v2 = 2;
3
4  private int f1(int v1)
5  {
6      int result = 0;
7
8      for (int i = 0; i < v1; i++)
9      {
10         result += v2 * i;
11     }
12
13     {
14         Random rng = new Random(2013);
15         int d1 = rng.Next(1, 7);
16         int d2 = rng.Next(1, 7);
17         result += d1 + d2;
18     }
19
20     {
21         int i = 1;
22         int d1 = 15;
23         result += d1 + i;
24     }
25
26     return result;
27 }
28
29 private void btnScopes_Click(object sender, RoutedEventArgs e)
30 {
31     int v2 = 5;
32     int n = f1(v2);
33     MessageBox.Show(string.Format("The result is {0}", n));
34 }
```

Lines 1 and 2 define two class-level variables with initializers. But method `f1` names its parameter `v1`. This means that within the method body, any reference to `v1` uses the local name, not the class-level `v1`. So on line 8 (where we use `v1`), the number of times the loop body is executed depends on the argument passed to it from the call site. (At line 32 we have a call site that passes the value 5 to the method.)

At line 31 we define a new local variable `v2`. This definition *hides* the class-level variable defined at line 2, and when `v2` is used on line 32, it uses the local `v2`.

There are a number of local variables defined inside method `f1`. We have the definition of `result` on line 6 (and its uses on lines 10, 17, 23 and 26). We also have a definition of variable `i` as part of the for loop at line 8. It's scope is lines 8 - 11, and cannot be used outside that scope. There are two blocks at lines 13-18, and 20-24. Each block has some definitions of new variables that are local to the blocks. We've deliberately defined variables `i` and `d1` at lines 21 and 22 to make the point that these are different variables (in different scopes) from the earlier `i` defined in the loop, and the `d1` variable defined at line 15.

On line 10 we use variable `v2`. It is not locally defined, so the scope lookup rule says "use the variable at the closest enclosing scope". In this case, the class-level variable at line 2 is used. Note that although line 31 also defines a variable called `v2`, that variable is not the one referred to at line 10!

You should ensure that you can trace through the code and determine what value the method will return. (Check yourself by copying the code and running it.)

If you want some more advanced information about scopes, take a look at http://msdn.microsoft.com/en-us/library/aa691132(v=vs.10).aspx

24.3. Lifetimes of objects and variables

Each time you click the button in the sample above, you'll get a message box showing the value 42. But wait! We're using a random number generator at lines 15 and 16 to throw two dice which we add to the result. Why are we not getting random results?

Every object and variable in a program has a **lifetime**. It gets created (or instantiated) at some particular moment in time, and then at some later time it dies.

Variables that are defined within a method or a block only live *while the method or block is being executed*. So calling method `f1` creates local parameter `v1` and local variable `result`. When `f1` returns, those variables are destroyed.

Similarly, in the loop, the variable `i` is created when the loop starts running, and it is destroyed when the loop exits. And the same is true for the blocks of code at lines 14-18, and 20-24.

What this means is that we're not creating a single random number generator here: we're creating a brand new one each time we execute line 14. And because we've given it a seed value as an argument in line 14, we will always get the same throws for the two dice.

If we moved line 14 out of the method `f1` and put its definition and instantiation at line 3, say, it would become a class-level variable instead of a local variable. Now the behaviour of our program would change. The random number generator would be created when the class was created (i.e. just before our window first appears, if this code is part of a WPF window), and its lifetime would only end when the window's lifetime ended.

Class-level instance variables (i.e. not variables that are static) have a lifetime that is the same as the lifetime of the object they belong to.

For writing the kinds of event-based GUI programs that we work with in this book, what this means is that if we want something to persist across multiple events, we should define it as a class-level variable. So if we need to count how many times a particular button is clicked, we need that counter to keep its value between clicks: it needs to be a class-level variable.

24.4. Objects can take some time to die

We've mentioned earlier that we have two kinds of types in C# — value types and reference types. A value type (like `int`, `double` or `Point`) is stored directly in a variable. Variables are instantly destroyed when control exits the scope in which they are defined.

But a reference type creates a reference, or a pointer, to the object it refers to. Our `Turtle` is a reference type, `Window` is a reference type, and `Random` is a reference type.

With reference types, the object is stored elsewhere in a region of memory called the *heap*. We've also seen that when we pass a reference type as an argument to a method, or when we assign one Turtle to another, we can create *aliases* — we can have more than one reference referring to the same object.

Objects become *inaccessible* when there are no longer any references pointing to them. So looking at our program above, on line 14, the variable `rng` is created, a new Random object is created in the heap, and `rng` is made to reference it. After passing line 18 in the program, the `rng` variable is destroyed. So at this point the object in the heap becomes inaccessible — no references point to it. We call inaccessible objects **garbage**. The object is still "alive", however.

Periodically, C# runs a **garbage collector**. It's job is to kill any garbage objects in the heap, and to recycle the memory so that the memory can be used again for new objects.

This is a little like some file or email systems

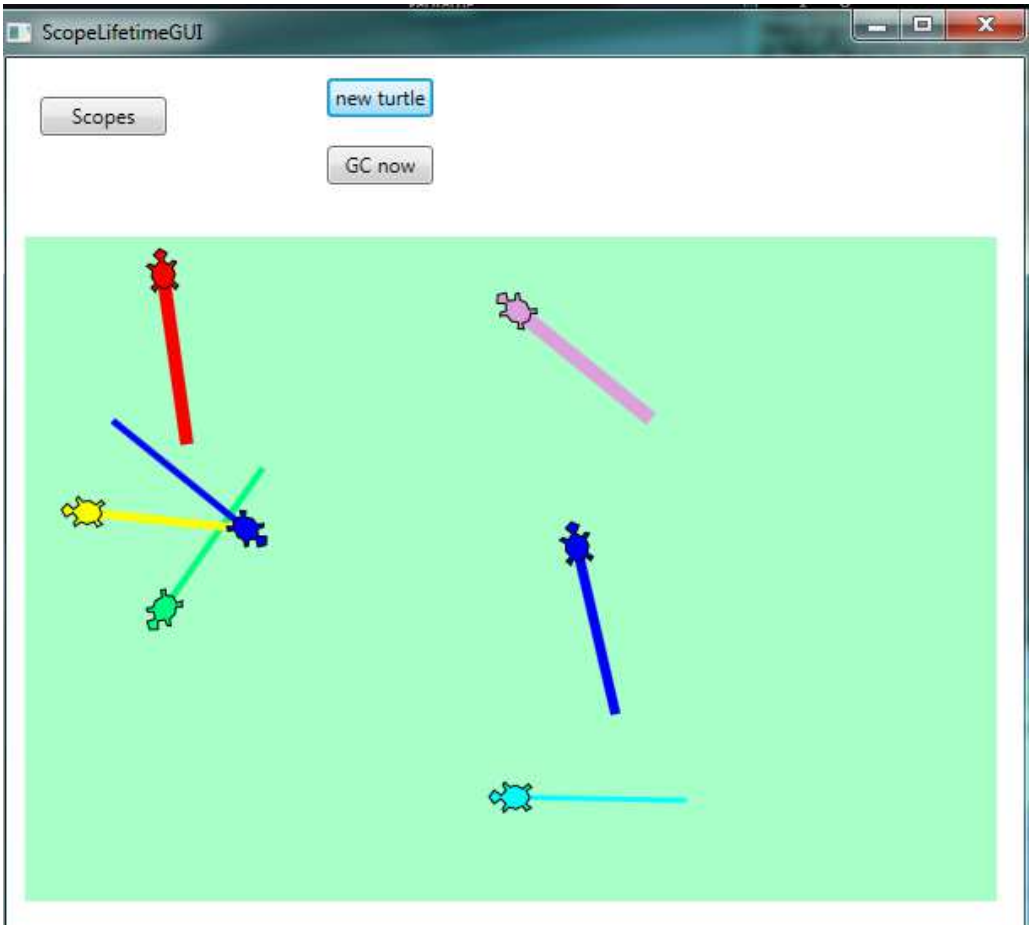
Often when we delete a file or email it gets moved to a *recycle bin* or a *deleted folder*. So it is not dead yet — just inaccessible from our "normal" filing system or inbox. When we do get around to emptying the recycle bin or our deleted folder, it is finally gone forever.

So for reference types, lifetimes are a bit more complicated than they are for value types.

Normally we need not concern ourselves with lifetimes: C# will handle most situations very nicely in the background. But we can have some fun exploring the mechanism, and we can build a really good understanding of the inner workings by doing this.

Let's go back and play with another Turtle program. When we create a Turtle, we create an object on the heap, but we also create some graphical lines and shapes on our playground canvas. When our turtle becomes inaccessible it won't die immediately — it hangs around until the garbage collector decides it is time to clean up and kill off all inaccessible objects. Just before a turtle dies, it gets an event from the garbage collector that allows it to do its "final actions". (Technically, we call the handler a *destructor*, the lifetime opposite of the *constructor*.) As its final action, a turtle will removes its shapes (lines, footprints, etc.) from the playground. This gives us a nice visual way for us to "see when the turtle actually dies".

```
1  Brush[] myFaves = { Brushes.Aqua, Brushes.Red, Brushes.Plum,
2                      Brushes.Yellow, Brushes.Blue, Brushes.SpringGreen };
3
4  private void btnNewTurtle_Click(object sender, RoutedEventArgs e)
5  {
6      Random rng = new Random();
7      Turtle tess = new Turtle(playground, rng.Next(400), rng.Next(400));
8      tess.Heading = rng.Next(360);
9      tess.BrushWidth = rng.Next(2, 10);
10     tess.LineBrush = myFaves[rng.Next(myFaves.Length)];
11     tess.Forward(100);
12 }
```



Each time the button is clicked we create a new turtle. We start it off at a random position in the playground, on a random heading, with a random brush width, and a random brush colour. Then we draw a single line.

Each time we click the button we get a new turtle on the heap, and will soon have something looking this screen-shot. But each time we exit the handler, we destroy variable `tess` — so we lose the reference to the turtle, and it becomes inaccessible, or garbage. (But not dead yet, so we can still see its shapes on the playground.) After a while the garbage collector springs into action, and all the shapes suddenly disappear from the playground as the inaccessible turtles die.

The garbage collector kicks in at unpredictable times — sometimes when we resize our window, sometimes when we click the button, sometimes when we just move the mouse over the window, occasionally when we're just sitting back watching our screen.

C# provides a `GC` class (in the `System` namespace) that provides a method for the programmer to explicitly ask for garbage collection. So we can add an extra button to our GUI, and give it a handler like this:

```
1  private void btnForceGC_Click(object sender, RoutedEventArgs e)
2  {
3      GC.Collect();
4  }
```

This lets us explicitly force a garbage collection whenever we want one.

Let us summarize the main idea of this section again: variables and objects have lifetimes. In the case of reference types, the objects in the heap become inaccessible, often because the variables that reference them are destroyed. But the object itself might live on a while longer in the heap, until the garbage collector gets around to cleaning it up and reclaiming its memory.

24.5. Static members, static classes

We've talked about instances of a type: `tess` and `alex` are instances of type Turtle. `button1` and `button2` could be instances of type `Button`. And every instance has its own members (fields, variables, or methods),

But sometimes we want to have members that don't apply to the object, but they apply to the *type*. Let us suppose we have 10 instances of Samsung S5 phones. They'll each have their own settings: phone numbers, address books, and so on. But there can be other information that applies to the *type* of phone, not just to individual instances. For example, the screen and camera resolutions, and the size and weight of the phone apply to all Samsung S5 phones.

A member (property, method, variable) can be defined to be **static** in a class. This means that it will be associated with the class, not with the objects that are instantiated from the class.

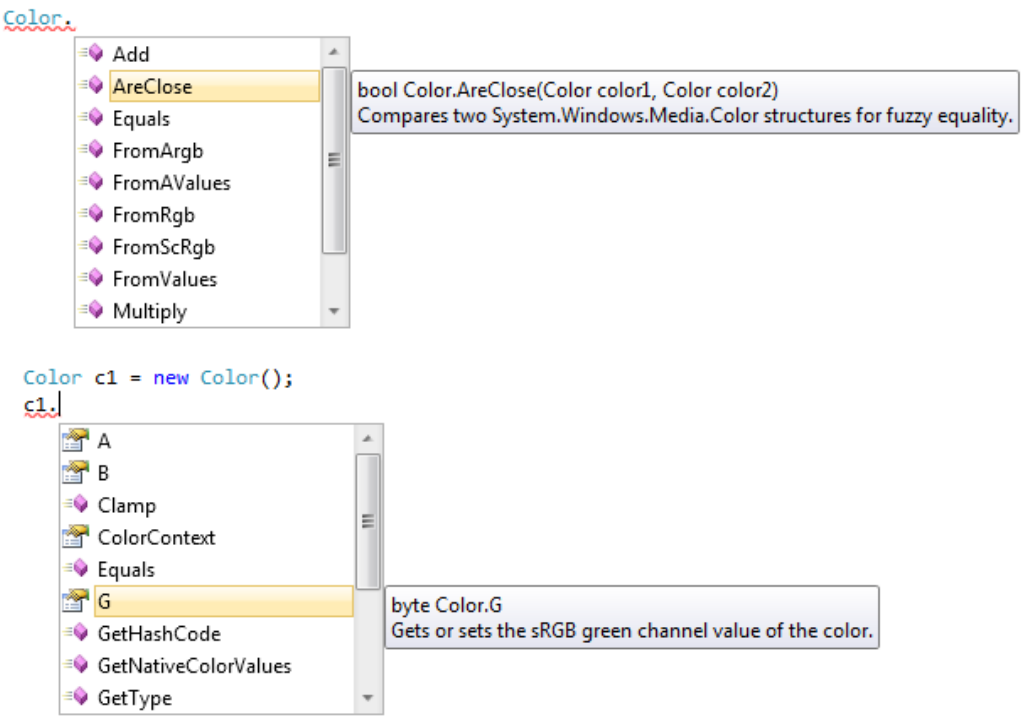
Let's see some examples that we know well already:

`Math` is a class in namespace `System`. It provides static methods like `Math.Sqrt` and `Math.Cos` that we can call. It also has a few static fields (`Math.E` and `Math.PI`) that hold some useful constants.

The important point is that we don't have to instantiate an object of type `Math` and then call methods of the instance: we use the methods directly from the class.

Static classes are classes that *cannot* be instantiated. They exist only for their static members. `Math` is a static class. You cannot create an instance of a Math object. We've also used methods from the static class `System.Convert`, e.g. `Convert.Int32("1234")`.

Some other classes we've seen have static members, but they can also be instantiated. (So they have static members, but they're not static classes.) For example, `Color` (from the namespace `System.Windows.Media`) is a class like this:



In the upper part of the image we see the IntelliSense showing us members that are static: the class name is `Color`, and IntelliSense pops up the static members.

In the lower part of the image we've instantiated a colour instance called `c1`. Now IntelliSense shows us the members that belong to the instance. They're not the same as the static members.

The help files for C# also use little icons to signal whether members are static or whether they belong to an instance. In the fragment of the help page from http://msdn.microsoft.com/en-us/library/system.windows.media.color.aspx notice the big red static markers that tell us that the first two methods belong the the class, the others belong to instances.

Methods

	Name	Description
S	Add	Adds two Color structures.
S	AreClose	Compares two Color structures for fuzzy equality.
	Clamp	Sets the ScRGB channels of the color to within the gamut of 0 to 1, if they are outside that range.
	Equals(Color)	Tests whether the specified Color structure is identical to the current color.
	Equals(Object)	Tests whether the specified object is a Color structure and is equivalent to the current color. (Overrides ValueType.Equals(Object).)

24.6. Glossary

scope

The region of program code in which a name can be used without the need for extra qualification.

class-level scope

Any name defined directly within a class, and visible to methods within the class.

garbage

Any object that has become inaccessible.

garbage collection

A periodic process that reclaims and recycles the memory used by inaccessible objects.

heap

An area of memory where objects are created and where they live.

inaccessible object

An object which has no references from the program pointing to it. This happens because a variable can be reassigned, and made to point to a different object, or because a variable gets destroyed when control leaves the scope in which it is defined.

lifetime

The time from when a variable or an object is created (e.g. from control entering a new scope or from object instantiation) until it is destroyed (from control exiting the scope, or the garbage collector reclaiming the memory associated with the object.)

local scope

The closest enclosing block or method in which names are defined.

static class

A class that cannot be instantiated. `Math` is a static class.

static member

A member that is accessible via the class name, not via an instance object. `Canvas.GetLeft` in our *More Event Handling* chapter is one example.

24.7. Exercises

1. The `GC` class can tell us how much heap memory our program is using. Create a button on your GUI, and bind it to a handler like this so that the memory usage gets shown in the title bar of the window:

```
private void btnMemUsage_Click(object sender, RoutedEventArgs e)
{
   this.Title = string.Format("Memory in use = {0}", GC.GetTotalMemory(false));
}
```

What is interesting is that each click of the button consumes quite a large number of bytes of memory (for processing the event, passing the arguments to the handler, redrawing the window, and so on.) Notice that if you click enough times the garbage collector will recover the memory.

2. Do an experiment to measure how much memory gets used each time we create a new array of int. Because there is a lot of system overhead on every click and redraw of the window, we need to find some sensible way of controlling our experiment. Here is a suggestion:

```
long m1 = GC.GetTotalMemory(false);
int[] xs = new int[10000];
long m2 = GC.GetTotalMemory(false);
this.Title = string.Format("Memory for the array = {0}", m2-m1);
```

3. Repeat the above experiment, but this time allocate an array of double.

25. GUIs for our Queens

We solved the N-Queens problem in an earlier chapter. Now we'd like to draw the boards for the solutions as we find them.

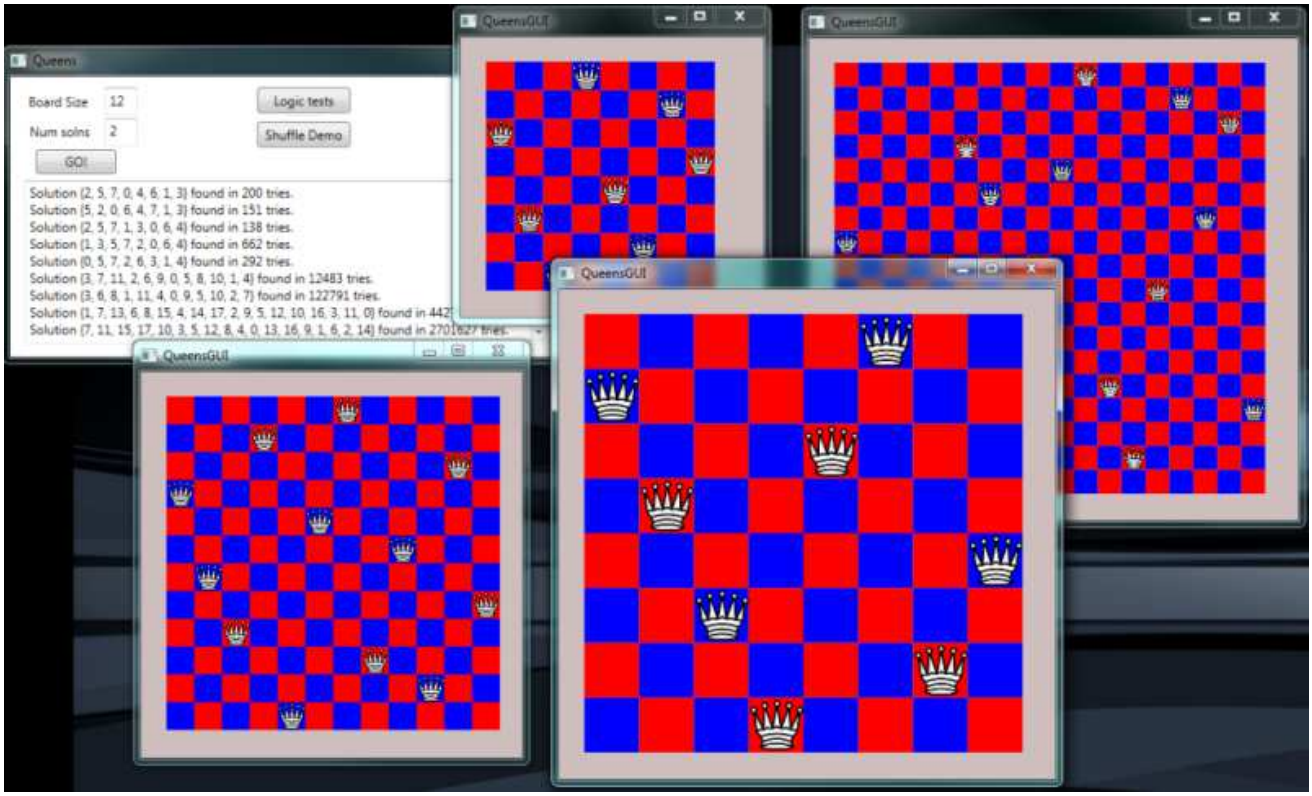
This chapter also takes us quite a bit deeper into WPF: we'll learn how to have and manage more than one window in our application, and we'll learn how to create visual controls in those windows *from our C# code*, rather than the usual way — from the XAML or the Designer in Visual Studio.

Along the way, our two underlying agendas remain:

- To become more familiar with objects: how we create and manipulate them, their states, their properties, their events, their lifetimes, etc.
- To build our algorithmic skills with loops, arrays, conditionals, and events.

In our earlier N-Queens puzzle-solver each solution was represented as an array of ints, like this: [6, 4, 2, 0, 5, 7, 1, 3].

Our goal now is to create a new window for each solution, and draw the boards, like this screen shot which shows two solutions for 8x8 boards, one for a 12x12 case, and one for an 18x18 case.



The 8x8 board consists of 64 rectangle objects that are placed and positioned on a canvas. Then we place 8 image objects on the canvas too, to draw the eight queens.

We can't do this statically in the XAML because we don't know ahead of time how big the board is going to be, or where the queen images need to be placed. So the correct number of rectangles and queens, and their sizes and positions can only be determined dynamically (while the program is running) — i.e. we're going to have to write some loops in our code to create these objects as each solution is found.

Like our previous canvas objects, we'll also make this canvas stretch when the window is resized. So whenever the canvas changes size, we'll have to recompute and adjust the positions and the sizes of all the rectangles and all the queens. Take another look at the picture above: one 8x8 board is a lot bigger than the other 8x8 board, because we resized its window. That means its rectangles and queens need to be bigger, and positioned differently.

25.1. Creating a new window for each queens solution

We have a solver that already finds solutions in one class — the code behind our main window. We now want a second type of window, which we'll call the `QueensGUI`. Some of the sub-problems we need to solve now are:

- How does our main window instantiate (create) and show a new `QueensGUI` object?
- How do we pass the board that we want drawn from the solver to the newly created `QueensGUI` object?
- Within a `QueensGUI` object, how do we create the Rectangle and Image objects, and keep track of them?
- Within a `QueensGUI` object, how do we lay out all the rectangles and images in their correct positions, with their optimal sizes?

To create a new Window class in our project we use Visual Studio, right-click on the project name in our Solution Explorer, and choose Add, and then choose Window. It will ask for the name for our new type — let's call it `QueensGUI`. Visual Studio will create the class, and open the familiar XAML designer for our window. At this stage, we can drag on a canvas (which we'll name `canvas1`. We'll also set some properties to make the canvas stretch when the window is resized.

This is the first time we've had more than one window class in a WPF program. WPF has a setting in one of the XAML files that tells it which window to instantiate and open when the application first starts running.

If we look at the XAML, we'll see the canvas is nested within a `Grid` — we'll give the grid a nice background colour, and give the canvas a different colour, and leave some border around the canvas so that it is visually easy to separate the two components.

Instantiating a new QueensGUI window (and showing it) is just a few lines of code. You'll put this code into your previous Queens solver so that after we've displayed the textual solution, we also create a new window. The changes are highlighted below:

```
1  private int[] findQueensSolution(int N)
2  {
3      // set up the initial board of the correct size
4      int[] bd = new int[N];
5      for (int i = 0; i < N; i++) bd[i] = i;
6
7      int tries = 1;
8      while (boardHasDiagonalClashes(bd))
9      {
10         shuffle(bd);
11         tries++;
12     }
13
14     // output the results ...
15     txtResults.AppendText(string.Format("Solution {0} found in {1} tries.\n", stringify(bd), tries));
16     txtResults.ScrollToEnd();
17
18     QueensGUI theWindow = new QueensGUI();    //   Create a new window object in memory.
19     theWindow.Owner = this;                   //   Let it know that we're its owner
20     theWindow.Show();                         //   Get the window to show itself on the screen.
21
22     return bd;
23 }
```

Line 18 creates a new QueensGUI window object in memory. At this point the GUI is not shown — the Window exists only in memory. Line 20 makes the window visible on our screen.

Line 19 can be left out, but it impacts the lifetime of the new child window. It sets the new window's `parent` window reference to the solver's window. What this means is that if the parent window is closed, all its children windows will automatically close too. Without line 19, the children windows will remain open even after the parent is closed, and you'll have to close every window manually before your application terminates.

So we began with a queens solver from Chapter 20. But now, each time it finds a solution, it opens a new child window. So let's get on with making each child window do something more interesting.

25.2. Drawing the Board

The child window needs to know what board (with Queens) it is trying to draw, so our next step is to pass the board array from the parent solver to the new child. On line 18 in the code above we call the window's constructor. We've seen before with the Turtle class and the Random class that we can pass arguments to a constructor when the object is created. So that's what we'll do: change line 18 in the code above to send in a (reference to) our board array:

```
1  QueensGUI theWindow = new QueensGUI(bd);  // pass the board to the new window
```

The compiler will now give an error because our constructor doesn't expect an argument, so now it is time to dive into the code behind the QueensGUI window and change the constructor.

Remember that the role of a constructor in a class is to set the new object up to its *factory default* settings when each new object is created. So it is time to think about "what needs to be in this new class, what should it do, and how should it do that?".

When a new QueensGUI object is created, it should

- Save N, the size of the board (e.g. 8 for an 8x8 board) in a class-level variable.
- Create the required NxN Rectangle objects, and give them alternating colours, and place them on `canvas1`. (At this stage we won't worry about sizes or positions — we'll defer the layout until later, after the window is shown, when we know how big the canvas is.)
- We will need to keep track of the Rectangles, so each time we create one we'll store a reference to it into an List of rows. We'll later want to run a loop over every rectangle and fix its position and size. Since the board is always going to be a two-dimensional square board, we'll use a List of List of Rectangles.
- We'll also need to create and keep track of the Queen images. Here we'll just use a List of Image, each holding a reference to an Image control that shows one of the queens.

Some code then: the first few lines of our QueensGUI class are going to look like this:

```
1  public partial class QueensGUI : Window
2  {
3      private List<List<Rectangle>> rects; // Define a reference to a list of lists
4      private List<Image> queenImages;     // Define a reference to a list
5      private int[] theBoard;              // Our own reference to the caller's board
6      private int N;                       // Number of squares on each edge, eg 8
7
8      public QueensGUI(int[] board)        // The constructor
9      {
10         InitializeComponent();           // Initialize all the bits defined in XAML
11         theBoard = board;                // Save the reference for later use
12         N = theBoard.Length;             // Save board size as class-level variable
13         rects = createRects();           // Create the NxN rectangles
14         queenImages = createQueenImages();  // Create the N queen images
15     }
```

Lines 3–6 define some private class-level member variables: they'll live as long as the QueensGUI window remains open. We remind ourselves that defining a variable that can hold a reference to a list does not actually create that list. That must be done separately.

Lines 5 and 11 are an important technique that we'll often see and use when writing our own classes. The values contained in the parameter `board` are passed as the arguments for the constructor when it is called. But `board` is local to the constructor, not visible to the other methods. So we define a variable at line 5, and at line 11 we save our parameter by copying it into the variable (it is a reference to an array in this case). This can now be accessed from other methods in the class.

At lines 13 and 14 we'll call on some other (private) methods to create our Rectangles and Queens. Notice that `rects` is defined as a reference to a List of Lists. We'll have to bear that in mind when we create and lay out the rectangles.

Now let's write the method to create all those rectangle objects. We'll create an initially empty List of rows. Then a nested loop will create one row at a time, and add it to our List of rows. While we do this we'll also need to set some properties for each rectangle, (particularly the brush that will set its colour), we'll need to remember to add each rectangle as a child of `canvas1` as seen in line 15 (so that canvas1 knows to paint all its children when we need to draw the board):

```
1  private List<List<Rectangle>> createRects()
2  {
3      List<List<Rectangle>> result = new List<List<Rectangle>>();
4      Brush[] bs = { Brushes.Red, Brushes.Blue };
5      for (int row = 0; row < N; row++)
6      {
7          List<Rectangle> thisRow = new List<Rectangle>();
8          int whichBrush = row % 2;
```

```
9           for (int col = 0; col < N; col++)
10          {
11              Rectangle rect = new Rectangle();
12              rect.Fill = bs[whichBrush];
13              canvas1.Children.Add(rect);
14              thisRow.Add(rect);
15              whichBrush = (whichBrush + 1) % 2;
16          }
17          result.Add(thisRow);
18      }
19      return result;
20  }
21
22  private List<Image> createQueenImages()
23  {
24      return null;   // TODO: we still need to write this later ...
25  }
```

Every time we execute the inner loop body we create a new Rectangle object, we save its reference into `thisRow` list, and add it to the canvas. Line 8 is responsible for making sure each new row starts on a different colour: `whichBrush` will always either have the value 0 or 1. Then, as we move along the row, at line 17, we keep swapping the brush index to go 0,1,0,1,0,1,0,1 ... So the colours of each block will alternate in each row.

It is quite common practice for programmers to write little stubs for functionality that they intend to provide later. Lines 24–27 are part of this scaffolding: we have to provide a method because we called it from the constructor, but we can leave it empty for the moment as we try to focus on one thing at a time.

25.2.1. Positioning and sizing the rectangles

Before we concern ourselves with drawing the queens, let's get the board drawn. When the window is first created, it has not been shown. Until the window gets shown, Windows won't compute how things are laid out. So we don't yet know how big the canvas will be, and can't decide yet how big to make the rectangles, or where to position them.

After the window is first shown, (or when the user resizes the window), the canvas gets a `SizeChanged` event. This is the right time for us to work out where we want each rectangle to be drawn, and how big it needs to be. And if we put this logic behind this event, it will automatically recompute whenever the window size changes.

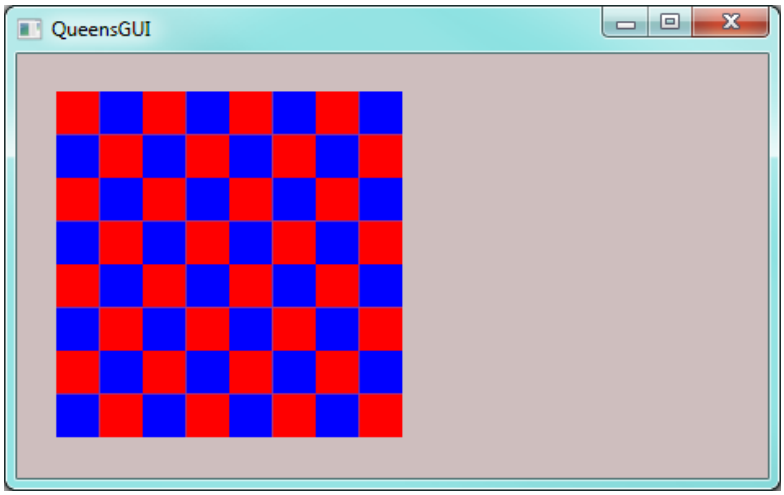
So using the XAML designer we attach a handler to the `SizeChanged` event, and use it to call some new methods that we will write to reposition our rectangles (and eventually) our queens.

```
1   private void canvas1_SizeChanged(object sender, SizeChangedEventArgs e)
2   {
3       layoutRectangles();
4       layoutQueens();
5   }
6
7   private void layoutRectangles()
8   {
9       double side = Math.Min(canvas1.ActualWidth, canvas1.ActualHeight);
10      double rectSz = side / N;
11
12      for (int row = 0; row < N; row++)
13      {
14          for (int col = 0; col < N; col++)
15          {
16              Rectangle rect = rects[row][col];
17              Canvas.SetLeft(rect, rectSz * col);
18              Canvas.SetTop(rect, rectSz * row);
19              rect.Width = rectSz;
20              rect.Height = rectSz;
21          }
22      }
23  }
24
25  private void layoutQueens()
26  {
27      // TODO:  we'll leave this as a stub till later.
28  }
```

We intend to keep the squares properly square. But the user can resize the window (and the canvas) narrow or higher, (i.e. not square). In line 9 we find the smaller side of the canvas. We know the board is NxN, (line 10), so we can work out `rectSz` — the width and height that every rectangle should be. We set each of our rectangles that we created earlier to that size in lines 19 and 20.

Each rectangle is positioned on lines 17 and 18. The distance from the top of the canvas depends on which row the rectangle is in, and the distance from the left depends on its column.

Great, now if we solve an 8x8 board, we'll get this:

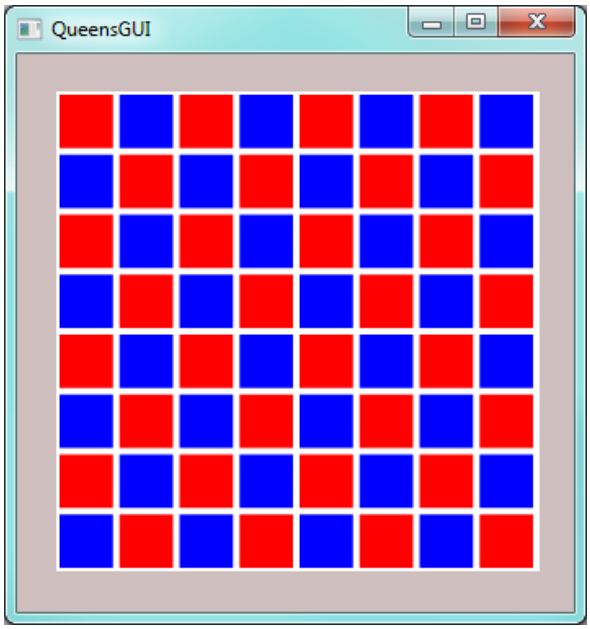


Notice that the board has sized to the smaller of the dimensions — its height, in this case. If we do resize the window we'll see the board stretching or shrinking.

Here is a fun idea that is a very easy change, and makes the board look a lot nicer. We could make the rectangles slightly smaller than what we computed above, but not change their positions. This would give a small open margin between each, so that they don't touch each other. The background colour of the canvas would show though the gaps. So here we demonstrate: we've set the canvas colour to white, and we change lines 17–20 in the code above like this:

```
1      Canvas.SetLeft(rect, rectSz * col + 0.05 * rectSz);
2      Canvas.SetTop(rect, rectSz * row + 0.05 * rectSz);
3      rect.Width = rectSz - 0.10 * rectSz;
4      rect.Height = rectSz - 0.10 * rectSz;
```

So we've made each rectangle 10% smaller in both width and height, and we've repositioned each rectangle by 5% of its size. Visually, it is quite impressive what a big difference such a small change makes:

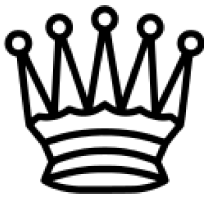


Tiled floors or walls always have “grout” between the tiles, like this!

25.2.2. Adding the Queens

A WPF Image control can render an image. Our approach in this section will be precisely the same as it was for the Rectangle controls, but we’ll use Image controls instead. When we instantiate the window, we’ll create N Image controls on the canvas, and we’ll also keep track of them in a list. Then we’ll complete the code for the `layoutQueens` method to position and size the Image controls correctly.

We’ll also need a picture for our queens which we’ll add to our project (as we did earlier with the bouncing ball, in the “More Event Handling” chapter). You might want to click and save one of these images, or find one of your own.



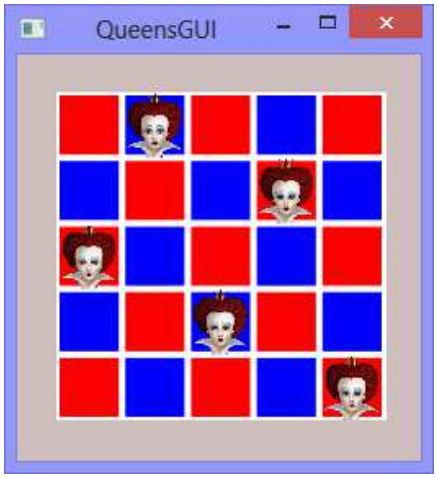
Here then is the code to create the N Image controls, and give them all the same picture. Notice that this method was already called earlier by the constructor for the Window.

```
private List<Image> createQueenImages()
{
    List<Image> result = new List<Image>();   // Create the List
    BitmapImage bm =             // Fetch the bitmap that we're going to use
         new BitmapImage(new Uri("pack://application:,,,/Queen_of_Hearts.png"));

    for (int i = 0; i < N; i++)     // Build the controls and the list.
    {
        Image im = new Image();     // Create the WPF control
        im.Source = bm;             // Tell it what image to display
        canvas1.Children.Add(im);   // Add the control to the canvas
        result.Add(im);             // And remember a reference to it.
    }
    return result;
}
```

The final tweak we need to make is to position and size each Image control correctly.

```
private void layoutQueens()
{
    double side = Math.Min(canvas1.ActualWidth, canvas1.ActualHeight);
    double rectSz = side / N;
    // Position and size the Image controls that display our queens
    for (int i = 0; i < N; i++)
    {
        Image q = queenImages[i];
        q.Width = rectSz;
        q.Height = rectSz;
        Canvas.SetLeft(q, i * rectSz);
        Canvas.SetTop(q, theBoard[i] * rectSz);
    }
}
```

And here’s one solution for a 5x5 board:



At this stage you can go back and generate multiple solutions for different size boards: each solution should display in its own window that is resizeable.

25.3. Can we see all the interim boards as the solver tries each shuffle?

Let's go back to the code for `findQueensSolution` which was at the top of this chapter. We did a search, and once we found a solution we created a window and displayed the solution. But another plan could be to create the Window and the board when we *start* the search, and as we try every new shuffling of the board, we could rearrange the queens so that they displayed every shuffling.

We'll need one new (public) method called `RefreshQueenPositions` in the GUI class, and we'll make a small change to `findQueensSolution`:

```
1  private int[] findQueensSolution(int N)
2  {
3      // set up the initial board of the correct size
4      int[] bd = new int[N];
5      for (int i = 0; i < N; i++) bd[i] = i;
6
7      QueensGUI theWindow = new QueensGUI(bd);
8      theWindow.Owner = this;
9      theWindow.Show();
10
11     int tries = 1;
12     while (boardHasDiagonalClashes(bd))
13     {
14         shuffle(bd);
15         theWindow.RefreshQueenPositions();
16         tries++;
17     }
18
19     txtResults.AppendText(string.Format("Solution {0} found in {1} tries.\n", stringify(bd), tries));
20     txtResults.ScrollToEnd();
21
22     return bd;
23 }
```

There is one critically important aspect of this code that we must understand. In line 7, when we passed the array `bd` to the constructor for the new window, we passed a *reference* to the array, not its values. Our new QueensGUI object stored that reference into one of its own variables. *But there is still only one underlying array, with more than one reference pointing to it.* So the array is aliased. When we shuffle the array here on line 14, it means that the QueensGUI object has a reference to the newly shuffled array. So our call in line 15 will use the latest shuffling to position the queen images.

Let us write `RefreshQueenPositions` in the QueensGUI class like this:

```
1  public void RefreshQueenPositions()
2  {
3      layoutQueens();
4      Dispatcher.Invoke((Action)delegate{}, System.Windows.Threading.DispatcherPriority.SystemIdle);
```

```
5       // Magic spell to force the GUI to get updated immediately
6   }
```

Line 3 recomputes all positions and sizes of the Image controls, on the basis of the permutation that is currently in the shuffled array.

Line 4 is a messy workaround that we'll need, but we won't need to understand it in too much detail. When WPF changes something related to the GUI, it creates a task that says "I must remember to redraw the screen at some future time". This gets put in a "tasks to be done" queue with other tasks that it might have waiting (for example, responding to a click event on a button, responding to a timer tick event, etc.). It gives top priority to tasks that run the user's code, and only when it becomes idle, will it go back and do the work from the task queue.

Unfortunately for us, WPF doesn't think that updating the screen is important: so it postpones the work as a "future task to be done later when I have nothing better to do".

Line 4 forces WPF to re-prioritize things. It effectively says (with some weird syntax) *"pause here until you have cleared your backlog of work in your task queue"*. So it draws the screen before it starts the search for the next possible permutation.

With these changes in place you can watch as the algorithm tries every shuffle.

25.4. Key Ideas

- We've shown how to work with multiple windows in our programs.
- As each new window is created, we've created WPF controls and set properties from our own code, rather than from the XAML or Visual Studio's designer.
- We've written a handler for the canvas' `SizeChanged` event, and done our own layout of all the rectangles and images.
- We've passed information from the parent window to the child window by calling the constructor with an argument.
- The constructor has saved that information, and saved all the Rectangles and Queen references into class-level variables. This ensures that they will live while the Window lives, and that they'll be accessible to the other methods of the class.

25.5. Exercises

1. The `QueensGUI` class doesn't really have much interesting internal state or behaviour yet, apart from being able to resize itself. Add a keyboard handler so that each time the F2 key is pressed the board gets a different colour scheme. Provide at least three colour schemes, and cycle through them if the user repeatedly pushes the F2 key.
2. Add some logic so that the caller can pass in the two brushes for painting the rectangles.
3. Experiment with different kinds of brushes. A good hint is to look up help for `System.Windows.Media.Brush` and expand its inheritance hierarchy to see what specific kinds of brushes are available. For example, one could try a `RadialGradientBrush`.
4. Drawing the screen is quite slow which is why WPF made it "low priority". So in the last section of this chapter we got it to draw a lot. Run some timing experiments to estimate how many tries per second our solver can do a) if it is not having to update the GUI on each try, and b) if it is. Try the experiment for different size boards, and for windows that are stretched big, or made small.

26. Writing our own Classes

A class is a *type*. Up until this point in the book we've seen many classes and objects — Button, Canvas, Turtle, Random, Timer, etc. are all classes that we've used, but we've not yet designed our own classes (other than GUI windows). In this chapter we'll learn to design our own (non-GUI) classes.

 It is useful to think of a class as a *factory* for creating many objects of the type.

So we've been able to use a class like a factory, e.g. the Button class gives rise to many Button instances, or we can create many Timer instances, or many Turtle objects:

```
1  Turtle tess, alex;     // define two variables that can refer to type Turtle ...
2
3  // and somewhere else in the code
4    tess = new Turtle(playground, 10, 10); // Ask the factory to create a turtle
5    alex = new Turtle(playground, 90, 90); // and now manufacture another turtle!
```

So breaking our programs into multiple classes (instead of doing everything in one big class) has two really important advantages:

- It allows us to create many objects, all independent and disentangled from each other.
- It is an essential tool for managing bigger tasks with more complexity.

An object, as we've already seen, allows us to encapsulate, or chunk together, two main things: some *state* of the object, and some *functionality* for the object. The state can be internal to the object — private — or it can be exposed via public properties. Think about turtles again — each one has its own `Heading`, its own `BrushWidth`, and so on. They also have functionality — the things they can do. These are the public methods, like `Forward` or `Stamp`.

26.1. Object-oriented programming

In the Events chapter we animated some traffic lights. Go back to the traffic-light example there, and notice that we had three important abstractions, or "mental chunks" for the problem:

- We used a *state machine* to represent the *Model* (the internal mechanism or the "business logic"), of how traffic lights should work.
- There was a *View* of the traffic lights — some pictures that we changed in the GUI as the program ran.
- There was a source of events — timer tick events in this case, which was a *Controller* for the system. Buttons, scrollbars, menus, etc. are common parts of the controller in a GUI program.

The separation of problems so that we organize them as three different parts, a Model, a View, and a Controller, has turned out to be a really useful technique for designing software and games. The controller responds to the keyboard, mouse, gamepad, touchscreen, etc.), and it drives the changes in the model. Then the controller gets the view to show some representation of the model and its changes.

The way that we break up and organize programs into component parts is called a **software architecture**. Model-View-Controller (also called MVC for short) is one popular architecture.

We can get a nice overview at http://en.wikipedia.org/wiki/Model-view-controller.

Our working problem

Here we're going to re-implement the traffic lights, with some twists.

We now require three different sets of traffic lights. Each will have its own controller: one can be controlled by a timer, one by a keypress, and one by having the user explicitly click a button.

We'll need three separate state machines — internal engines for the model of how the lights work. Here we'll separate the logic for what the state machine should do into its own separate class, and we'll create three instances of that class, one for each of the different sets of traffic lights.

And finally, we'll have three different ways to visualize our traffic lights. One can use our pictures from the earlier chapter, one can just have some text output to say what the light is doing, and ... (we'll make up something when we get there...).

26.2. Designing a class

We'll want our own class to encapsulate (encapsulate means to enclose, as if in a capsule) the state and the functionality for a State Machine. One of the popular ways of figuring out how to break a complex system into more manageable components is to to write this information on a index card that is divided into three parts:

- What is the name of the class?
- What responsibilities will it (or the objects we instantiate from it) have,
- What other (types of) objects will it collaborate with?

See http://en.wikipedia.org/wiki/Class-responsibility-collaboration_card for more detail, but this is enough to get us started.

- Let's call our class `TrafficLightFSM` (FSM for finite state machine).
- Its responsibilities will be
  - to keep track of what state the state machine is currently in,
  - to advance (or transition) from the current state to its next state.
- Its collaborators will be
  - the controller logic in the GUI part of our program,
  - the viewer logic in the GUI part of our program.

One we've figured out what classes we want, (just the one in this case) we'll decide what properties and methods we need to give our objects so that they can fulfil their responsibilities.

For our very simple objects, let's start with one property — the `CurrentState` that the machine is in, and two methods: the constructor method to initialize new objects, and a method called `AdvanceState()`.

To get a new class in our project in Visual Studio, right-click on our project in the Solution Explorer pane, and choose Add | Class... You'll be prompted for a name (enter `TrafficLightFSM.cs`). Visual Studio will now generate a new class skeleton and add it to our project. It will open the code, and will look something like this:

```
1  using System;
2  ...
3
4  namespace Fragments
5  {
6      class TrafficLightFSM
```

```
7       {
8       }
9   }
```

Your namespace may be different to the example above but will be the same as the namespace used in your `MainWindow`, and in any other classes we create later. (Putting all the classes in our project into the same namespace makes it easier for us, because they can all "see" each other without having to add any extra `using` directives at the top.)

Now we can write our methods and our property definitions or private variable definitions inside the class. Here's what we'll put in place of lines 6–8 now:

```
1   class TrafficLightFSM
2   {
3       public int CurrentState { get; private set; }
4
5       public TrafficLightFSM()
6       {
7           CurrentState = 0;
8       }
9
10      public void Advance()
11      {
12          switch (CurrentState)
13          {
14              case 0:
15                  CurrentState = 1;
16                  break;
17              case 1:
18                  CurrentState = 2;
19                  break;
20              case 2:
21                  CurrentState = 0;
22                  break;
23          }
24      }
25  }
```

Lines 5–8 define the constructor for our new class — the code necessary to initialize a new object of this type to its factory settings. The distinguishing thing about the constructor is that it has no return type — not even the keyword `void` — and its name must be identical to the class name. If we do provide a constructor, it will be automatically called whenever we instantiate a new object of this type.

Lines 10–24 provide the functionality we wanted — the same logic as we saw in the "More Events Handling" chapter of this book.

Line 3 is new: it shows that we're defining a new property of type `int`. A property is like a variable, but with finer control over how it can be used — this one says that the property is public — it can be accessed outside the class (others can *get* its value), but the `private` modifier on `set` restricts the property so that it can only be set (assigned a new value) by methods within this class.

If we right-click on the class name in the Solution Explorer and choose the option "View Class Diagram",

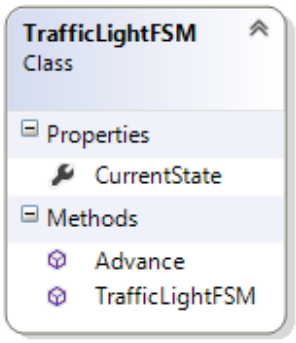
Properties can do more...

There are more complex ways of using properties that we don't cover.

They're preferred over variables. Public variables mean that an external collaborator might change them, so the object loses control of its own state.

we'll get a diagram of the class (this feature may not be in all versions of Visual Studio):

> But the reason why we encapsulate state in the first place is so that the object itself can stay in charge.



A class diagram is a widely used notation that allows software developers to express different aspects of their design. Many people will tell us that we should draw our diagram first, and write our class according to our diagram. In Visual Studio, because we can get the diagram from a class it seems sometimes easier just to start with the class.

Well done! We've written our very first (non GUI) class. Now let's make some objects of this type, and put them to good use.

26.3. Three sets of traffic lights — The Controllers and the Views

In the kinds of programs we write the Controllers and the Views are both intertwined in our Window class, so they're not fantastically well separated. We'll make a special effort to separate them in our thinking, at least.

Here's what the finished product will look like:

Let's begin by just getting the first set of lights to work. Most of the code is identical to what we saw previously in the *More Event Handling* chapter, except that the logic for the state machine is now in its own class.

```
1     private BitmapImage[] thePics;
2     private TrafficLightFSM model1, model2, model3;
3     private System.Windows.Threading.DispatcherTimer theTimer;
4
5     public TrafficLightsGUI()
6     {
7         InitializeComponent();
8
9         model1 = new TrafficLightFSM();
10        model2 = new TrafficLightFSM();
11        model3 = new TrafficLightFSM();
12
13        string inThisProject = "pack://application:,,,/";  // the magic spell ...
14        thePics = new BitmapImage[] {
```

```
                    new BitmapImage(new Uri(inThisProject + "TrafficLightGreen.png")),
                    new BitmapImage(new Uri(inThisProject + "TrafficLightAmber.png")),
                    new BitmapImage(new Uri(inThisProject + "TrafficLightRed.png")) };

        theTimer = new System.Windows.Threading.DispatcherTimer();
        theTimer.Tick += theTimer_Tick;
        theTimer.Interval = TimeSpan.FromMilliseconds(500);
        theTimer.Start();
    }

    private void theTimer_Tick(object sender, EventArgs e)
    {   // advance the model
        model1.Advance();
        // update the view
        image1.Source = thePics[model1.CurrentState];
    }
```

In line 2 we define variables for all three models, and at lines 9–11 we instantiate all three. Otherwise the code is pretty similar to what it was before: our class constructor instantiates all the objects it needs, and starts the timer. When the timer ticks we advance our model, and then we update the view by asking the model for its current state, and using that to choose the picture that is displayed by the image control.

Our second set of traffic lights is controlled by a keyboard event, so we create an event handler for the Window's `KeyDown` event, and respond to a keypress like this:

```
    private void Window_KeyDown(object sender, KeyEventArgs e)
    {
        switch (e.Key)
        {
            // Controller for light set 2 - press the N key for Next
            case Key.N:
                model2.Advance();
                switch (model2.CurrentState)  // update view
                {
                    case 0: lblOutputSet2.Content = "GO!";
                        break;
                    case 1: lblOutputSet2.Content = "PAUSE!";
                        break;
                    case 2: lblOutputSet2.Content = "STOP!";
                        break;
                }
                break;
        }
    }
```

So our "view" in this case is just some text that is displayed in a label. To reinforce the idea that the view is now independent from the model and controller, perhaps we can consider how easily we could change the view to display Spanish rather than English in the label.

Finally, the third set of traffic lights is controlled by clicking the button. So here is the code behind the button click event:

```
    private void btnAdvance_Click(object sender, RoutedEventArgs e)
    {
        model3.Advance();
        switch (model3.CurrentState)   // update view
        {
            case 0: progressBar1.Value = 33;
```

```
                progressBar1.Foreground = Brushes.Green;
                break;
            case 1: progressBar1.Value = 66;
                progressBar1.Foreground = Brushes.Orange;
                break;
            case 2: progressBar1.Value = 100;
                progressBar1.Foreground = Brushes.Red;
                break;
        }
    }
```

We've not used a `ProgressBar` control before, but they're quite easy. We drag one onto our Window, and we set its orientation property to make it grow vertically instead of horizontally. It shows progress according to the number we assign to its `Value` property. (By default, 0 represents no progress, 100 represents "100% progress".) We also change the brush colour to view the different states.

26.4. The keyword `this`

The keyword `this` is available in a class, and works like the English word "me". So every object can use a reference to itself. So `this.CurrentState` means "my CurrentState".

Normally we don't need to use the keyword explicitly. But it is really convenient in one particular situation — in a class constructor. So let us change our `TrafficLightFSM` constructor. Instead of always creating an FSM which begins in state 0, we now want the caller to be able to pass an argument to the constructor, to set the initial state:

```
1   class TrafficLightFSM
2   {
3       public int CurrentState { get; private set; }
4
5       public TrafficLightFSM(int CurrentState)
6       {
7           this.CurrentState = CurrentState;
8       }
9
10  ...
11  }
```

The tricky bit is at line 7. We now have a parameter at line 5 with exactly the same name as our property in line 3. We recall our discussion about scope lookup rules in Chapter 24: `CurrentState` means the most closely nested definition of the name. Which is the parameter on line 5.

Our new keyword, `this`, can be used to explicitly qualify the a name to mean *"the CurrentState that belongs to me, the object"* (i.e. the one defined at line 3, in *class-level scope*.)

It is sometimes easier just to choose another name for the parameter, but using the same names is popular practice in C# (and in Java). So we'll need to use `this` to work around the scope rules to let us access the class-level variable of the name. So we're sure to see it often.

26.5. Summary

By making `TrafficLightFSM` into its own class we've simplified the logic, especially when we want multiple instances, each with their own state. It would be tricky to have all the logic and variables for three separate state machines tangled up inside our Window class.

Our first `TrafficLightFSM` class was really simple: one property and one method (apart from the constructor). But the powerful thing is the idea: as we put more state and logic into our objects, we'll see more benefits from this approach of breaking programs into separate components that interact with each other. (Consider how complicated the behaviour of a Window or a Turtle object is, or for any of the GUI controls that we use so easily. Having classes to organize and manage this complexity is essential!)

26.6. The bigger picture

C# is a fully object-oriented programming language: every method must belong to a class, and the only way to organize computation is by having classes that give rise to objects that can interact with each other. Java is like this too.

An earlier style of organizing computation is called *procedural* (or *imperative*) programming. Look it up. Fortran, C, BASIC and Pascal are popular languages that support this style.

Some procedural languages added objects and classes to their procedural core, and they now have a (sometimes messy) mixture that can be used in either style. C++, Python, Visual Basic, and some newer versions of Pascal (e.g. Delphi Pascal) work like this.

26.7. Glossary

class

A class defines a new type of object. A class can also be thought of as a template or a blueprint for the objects that are created according to its specifications.

constructor

Every class may have special method that is invoked automatically to initialize a new object to its factory-default state.

instance

An object whose type is of some class. Instance and object are used interchangeably.

instantiate

To create an instance of a class, and to run its constructor, if one exists.

method

A method is defined inside a class definition and is invoked on objects of that class. Methods give objects their behaviour.

Model-View-Controller (MVC)

A popular way of breaking software systems into three co-operating components.

object

The run-time entity that is often used to represent a real-world thing. It bundles together the state (data) and the behaviour appropriate for that kind of thing. Instance and object are used interchangeably.

software architecture

The overall design of a large software system.

`this`

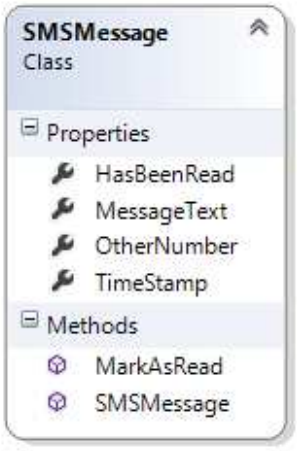
A C# keyword that refers to the current instance of the class.

object-oriented programming

A powerful style of programming in which data and the operations that manipulate it are organized into objects.

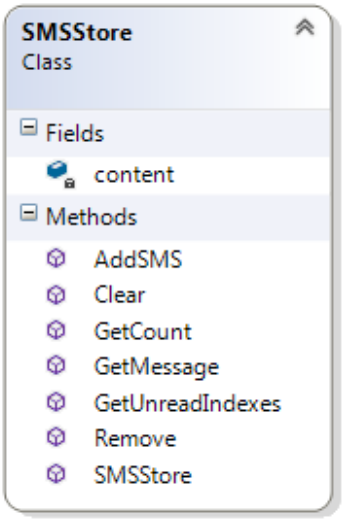
26.8. Exercises

1. Write a class definition for an SMSMessage that implements the class diagram here. It should contain four properties that are privately settable, but publically gettable. The constructor should initialize the sender's number and the message from parameters, and it should also set `HasBeenRead` to false, set the `TimeStamp` to `DateTime.Now`. The `MarkAsRead` method should change `HasBeenRead` to true.



2. Create a new class, SMSStore, that can be used for an SMS inbox or outbox on a cellphone. It should implement the class diagram here.

```
SMSStore myInbox = new SMSStore();
```



This store can hold multiple `SMSMessage` objects. (Hint: use a `List<SMSMessage>`!)

An SMSStore object should provide methods with these signatures:

```
void AddSMS(string fromNumber, string textOfSMS);
  // Makes a new SMSMessage object, inserts it after other messages.

int GetCount();
  // Returns the number of sms messages in the store.

List<int> GetUnreadIndexes();
  // Returns list of indexes of all not-yet-viewed SMS messages

SMSMessage GetMessage(int i);
  // Return message[i].
  // If there is no message at position i, throw an exception.

void Remove(int i);     // Delete the message at index i (or throw an exception)
void Clear();           // Delete all messages from the store
```

Write the class, create a message store object, write tests for these methods, and implement the methods.

3. Make a GUI like the one shown. In one half of the screen you can be in the role of a friend, and "send" new SMS messages to your inbox. In the other half of the screen you can be the phone

"user" — you'll be able to refresh the status (the view) of your inbox. (In this sample I used two Canvas controls and set different background colours to provide the two halves of the screen.)

You can read any message by entering its index number.

4. Extend your SMS GUI so that you don't need the "Refresh Status" button — the status automatically updates itself whenever you add a message to the Inbox, or when a message is read. Extend the interface so that the user can delete a message, or clear all messages from the Inbox.
5. Add another class called AddressBook to your program. Pre-program some phone numbers and names into the address book. (We won't worry about adding new contacts or deleting contacts, or updating their information, although this could be a nice exercise too!) Now change your GUI from the previous question so that if the SMS arrives from a known contact, their name will be shown in the GUI instead of their number.
6. Do the following experiment on your phone. Find an SMS in your Inbox for which you have no contact details. Add a dummy contact for that number. Does your Inbox now show the dummy contact name, or does it still show the original number? What happens if you now delete the dummy contact? Do you think your phone searches your contact list when the SMS arrives, or when you want to see your Inbox, or does it update your Inbox whenever you make changes to your contact list?
7. Take an existing class, and add the qualifier `this` to every use of a member (variable, method, property) so that you build a good understanding that an unqualified name often resolves (using the normal scope rules) to the local scope. But the `this` keyword explicitly forces the name to be at class-level scope.

27. In the Caves — a Case Study

This case study builds a skeleton of a game using a few more classes that we'll write ourselves.

It is *strongly recommended* that this chapter needs to be "hands-on" — get the code copied and running in Visual Studio as you work through the case study.

We'll stay with the same theme — state machines, state diagrams, and state transitions. This is a very useful formalism in Computer Science, with a wide range of applications. So perhaps it is time to be a little more precise.

Definition of a Finite State Machine (FSM)

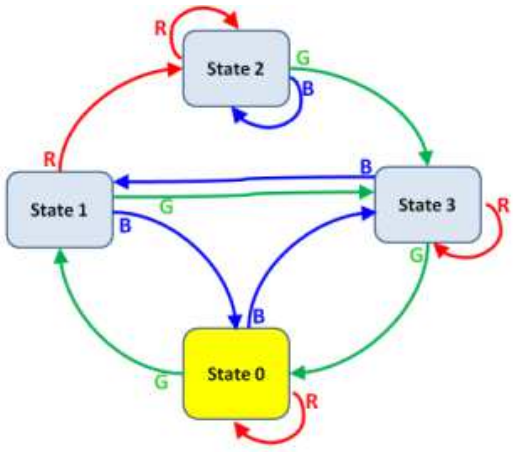
An FSM is described by a few components (all finite):

- A set of states that the machine can be in. One of these must be designated as the *starting state* of the machine. The machine can only be in *one* of its states at any possible moment.
- A set of possible inputs (or events) to the machine. (We'll call these FSM events, because we don't want to confuse them with C# events.)
- A state transition function. For each state that the machine can be in, and for each possible input, this mechanism describes what the next state will be.
- A set of *additional actions* that can occur. Actions can be triggered in one of three situations:
  - Whenever a particular *transition* occurs between one state and another,
  - Whenever a particular state is entered,
  - Whenever a particular state is exited.

Our earlier traffic light FSM can now be described in terms of this definition: It had three states, and only one input. After every transition we arrived in a new state. There were no additional actions (besides state transitions) built into our machine.

As another example, suppose a vending machine needs three coins to buy a chocolate: it might have three cyclic states representing the monetary balance (zero, one, two). When it gets an additional coin in the "two" state it would transition back to "zero" state, but attached to that transition we'd want an additional "dispense chocolate" action.

We'll typically draw a State Diagram to describe our machines. For example, this state diagram describes a machine with four states, and three possible inputs. The highlighted state is the current state. The transition arrows describe transitions. There are no actions represented in this diagram.



Maps, Caves, and Finite State Machines

A number of games involve being somewhere (i.e. in some state) in a cave that is part of a bigger system of caves, or on an island in a bigger system of islands. We have a limited number of choices (these are the inputs to the FSM) — perhaps doors that we can exit from, or passages in the caves, or directions that we can go, or ships that we can catch — to trigger the transition to the next cave, or the next island.

The four states in our diagram could represent four caves, or four islands, or four stations in an underground tube system. The R/G/B triggers could represent Red, Green, or Blue ships that we could catch to the next island, or exit passages from one cave to another, or platforms where we could catch the next tube train.

For a game, some of the states can contain some treasure, some reward, or some danger. But we don't want the user to know the "map" of the machine. That would make finding the treasure too easy. So the user needs to explore the possible states and transitions and find the treasure without going around in circles forever.

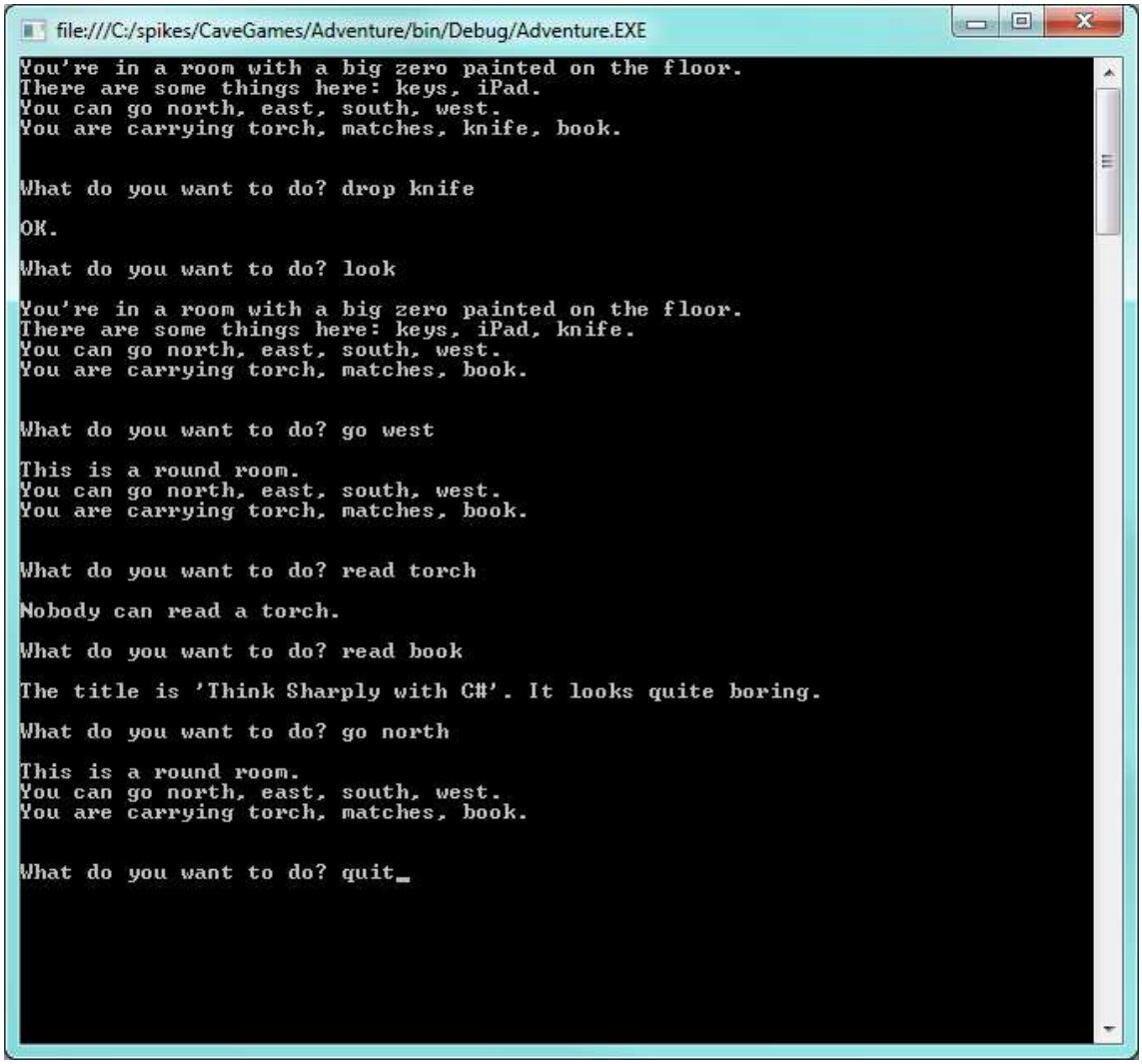
See http://csunplugged.org/finite-state-automata/ where the finite state machine is presented as a hunt for pirate treasure on some islands.

Our task now: write an "engine" for games like this, and then use our engine to build one or two cave or island games.

The game was inspired by Will Crowther's original "Colossal Cave Adventure" game — possibly the first ever dungeon game done on a computer. It has a fascinating history, and you can download the original code (or an executable that runs on a PC), or play the game on-line. Prepare to devote a good number of hours to this! http://en.wikipedia.org/wiki/Colossal_Cave_Adventure. Also read the really fascinating page http://rickadams.org/adventure/b_cave.html.

In the spirit of the original game, we'll do this as a Console Application rather than use WPF. In the original game everything was in upper-case too, (that is all computers could manage then), so take a peek at the very retro-looking screen shot on Wikipedia. We won't go that far! (Refresher: Section 4.3 shows how to create and use a Console Application.)

Here is a short interaction, showing what we aim to achieve:



When we create our Console Application we'll get a class called `Program` that contains the starting point — a static `Main` method. We'll add just two statements to that class, and write everything else in our own classes. So let's begin by changing the skeleton program like this:

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          Game g = new Game();
6          g.RunGameLoop();
7      }
8  }
```

Lines 5 and 6 instantiate a new game, and call its `RunGameLoop` method. So we now need to add a `Game` class to our project.

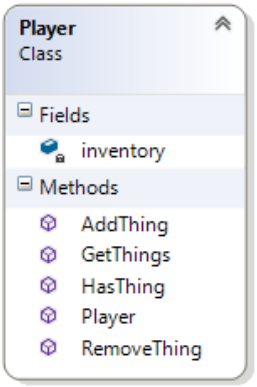
Designing classes is a huge topic in its own right. So we're not really going to spend too much time here arguing the merits of what classes we need and what responsibilities each should have. Instead, we'll just present and implement a design. Our approach is based on the advice given earlier: write down a few user stories, identify the nouns, and these often make good candidates to turn into classes of their own.

We'll introduce four additional classes:

- A `Player` class. The only responsibility (and state) at this time will be to keep an inventory of things that they are carrying (like the torch, or the keys).
- A `State` class. This will correspond to a state in our FSM, and will represent a Cave or an Island in the game. It needs a description, and also needs to keep an inventory of things that are in the room.
- An `FSM` class, as we had in the last chapter. It implements the map of the game, and is responsible for moving from one state to another.
- The `Game` class, which controls the game. It will do all the interaction with the user via the Console, and will instantiate and "own" the finite state machine object, and the player object. It must "understand" what the user types in, and be the controller for the game. After each change to the state, it must update the view — i.e. give feedback to the user about the new situation.

The one other obvious noun that might qualify for its own class would be the "thing" — torch, iPad, knife, etc. At this stage, things don't have any behaviour of their own (for example, the torch cannot run out of battery power, and we cannot turn on the iPad), so we will just represent each "thing"" as a string.

27.1. The `Player` class



The class diagram here shows the members that we'll want for a player. These diagrams show private and public members. We are only interested in the public members when we're figuring out how this component will interact with other components. We're interested in the private members because they tell us about the state that an object can be in, or private operations it can do. Recall that the tiny padlock icon next to `inventory` means "private".

You'll also notice that the class diagrams have collapsible sections for the Fields, Properties, Methods and all the detail of the Class. This allows us to choose to view the level of detail and the "mental chunking" that is appropriate for our task.

Let's implement the player class (we haven't shown the `using` directives and the namespace sections of the code:)

```
1  class Player
2  {
3      private List<string> inventory;
4
5      public Player(params string[] initialThings)
6      {
7          inventory = new List<string>(initialThings);
8      }
9
10     public void RemoveThing(string thing)
11     {
12         if (!inventory.Contains(thing))
13         {
14             throw new Exception("You don't have one.");
15         }
16         inventory.Remove(thing);
17     }
18
19     public void AddThing(string thing)
20     {
21         inventory.Add(thing);
22     }
23
24     public string GetThings()
25     {
26         return string.Format("You are carrying {0}.\n", string.Join(", ", inventory));
27     }
28
```

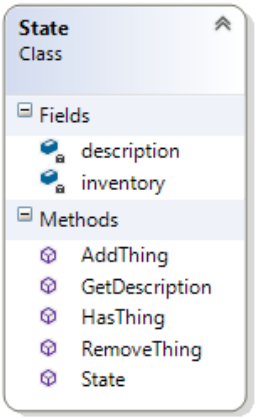
```
29      public bool HasThing(string t)
30      {
31          return inventory.Contains(t);
32      }
33  }
```

There are some noteworthy aspects here:

- Because we need to dynamically add and remove items from the player's inventory, a `List<string>` is a good choice of data structure.
- The inventory needs to get initialized. The collaborator that instantiates the class must pass in the things. At line 7 we turn the array into a list, because we need it to be dynamic. Building our own list also means that the list really is private and encapsulated in the class. If we got our collaborator to create the list for us, and we just saved a reference, they'd have a "back-door" way of modifying the player's inventory.
- On line 14 we throw an exception on trying to remove a thing that we don't have. So the caller will need to catch that exception and pass a message back to our user.
- On line 26 we used `string.Join` to join all our inventory items into a single description string, with commas between each of the items. It works, but may not be the ultimately satisfying way to do it.
- Our boss is going to complain that we haven't documented any of our methods decently. We'll cover that in the next chapter.

27.2. The `State` class

Here we need to deal with each cave or island in our game. It too needs an inventory (and we've chosen the same method names and representation as we used for the player class).

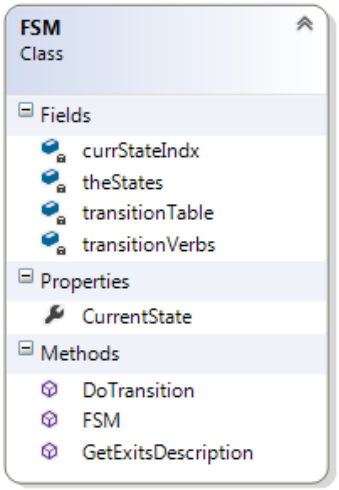



```
1   class State
2   {
3       private string description;
4       private List<string> inventory;
5
6       public State(string theDescription, params string[] initialThings)
7       {
8           description = theDescription;
9           inventory = new List<string>(initialThings);
10      }
11
12      public string GetDescription()
13      {
14          if (inventory.Count > 0)
15          {
16              return string.Format("{0}\nThere are some things here: {1}.",
17                                   description, string.Join(", ", inventory));
18          }
19          else
20          {
21              return string.Format("{0}", description);
22          }
23      }
24
25      public void AddThing(string thing)
26      {
27          if (inventory.Contains(thing))
28          {
29              throw new Exception("There is already one here.");
30          }
31          inventory.Add(thing);
32
```

```
    }

    public void RemoveThing(string thing)
    {
        if (!inventory.Contains(thing))
        {
            throw new Exception("There isn't one here.");
        }
        inventory.Remove(thing);
    }

    public bool HasThing(string t)
    {
        return inventory.Contains(t);
    }
}
```

The `GetDescription` method returns the description of the cave, and its inventory. But it takes a bit of extra care not to report an empty inventory.

27.3. The `FSM` class




```
class FSM
{
    private int currStateIndx = 0;
    private int[][] transitionTable;
    private State[] theStates;
    private List<string> transitionVerbs;

    public State CurrentState { get; private set; }

    public FSM(State[] states, int[][] stateTransitionTable, params string[] eventVerbs)
    {
        theStates = states;
        transitionTable = stateTransitionTable;
        CurrentState = states[0];           // the initial state
        transitionVerbs = new List<string>(eventVerbs);
    }

    public void DoTransition(string fsmEventVerb)
    {
        int indx = transitionVerbs.IndexOf(fsmEventVerb);

        if (indx < 0)
        {
            throw new Exception("Go where?");
        }

        int newState = transitionTable[currStateIndx][indx];
        if (newState >= 0)
        {
            currStateIndx = newState;
            CurrentState = theStates[currStateIndx];
        }
        else
        {
```

```
35              throw new Exception("That's not possible!");
36          }
37      }
38
39      public string GetExitsDescription()
40      {
41          string exitDescription = "You can go ";
42          string separator = "";
43          for (int col = 0; col < transitionVerbs.Count; col++)
44          {
45              if (transitionTable[currStateIndx][col] >= 0)
46              {
47                  exitDescription += separator + transitionVerbs[col];
48                  separator = ", ";
49              }
50          }
51
52          if (separator == "") return "There is no way out!";
53          return  exitDescription + ".";
54      }
55  }
```

The key ideas here are as they were in the previous chapter with the Traffic Lights finite state machine, but we've made a few improvements.

The machine keeps a list of the state objects, and exposes a `CurrentState` property (on line 8) that only it can set, but the collaborators can get. It sets the property initially at line 14, and changes it at line 31.

Looking back to our previous finite state machine examples (the traffic light example done twice, once in Chapter 12 and again with a separate class in the previous chapter), we had a very simple state diagram with only one event that caused transitions. Now we have a more complex FSM: at each state there are many possible events. `transitionTable` is an array of an array of integers because the first array specifies which cave you are in and the second is an array of integers which represent all the possible exits and where they lead to for the cave you are in.

So there are two important generalization built into this example.

1. Firstly the transition table is not hard-coded in the FSM class. So each time we create a new FSM instance, we can provide a different transition table specific to that FSM.
2. The set of event words are also not hard-coded as part of the FSM logic. The verbs that trigger the transitions are also set up in the constructor. So the `DoTransition` method now takes the FSM event verb as a string, and it looks up the corresponding index to get to the correct element in the transition array.

These generalizations serve us well. We could re-use the same FSM class to have a version of the caves game in another language, or we could use this FSM class to build a controller for our Traffic Lights.

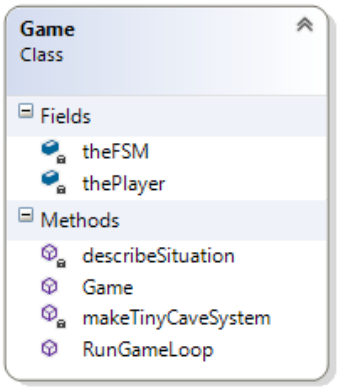
Finally, in lines 39–54 we go to quite a lot of trouble to build up a description of the possible exits from each room. The way we use the `separator` variable is a bit tricky: we put separators before each new possibility, but we get around the fact that the first item doesn't need a comma before it. Then on top of that complication, we also use the variable at line 52 to determine whether we found any exits at all. (Of course, this part of the code is pretty specific for the caves game!)

And if you want to understand why we chose that specific text for the exception at line 35, it is a quote from Boris the Animal.

27.4. Tying it all together — the `Game` class

When we create the Game instance it must set up the caves (States), create the FSM for the transitions and create the player object.

We also need a method to output the current view, and we'll call that too.




```
1  class Game
2  {
3      private FSM theFSM;
4      private Player thePlayer;
5
6      public Game()
7      {
8          theFSM = makeTinyCaveSystem();
9          thePlayer = new Player("torch", "matches", "knife", "book");
10         describeSituation();
11     }
12
13     private FSM makeTinyCaveSystem()
14     {
15         int[][] transitionTable =
16         {   new int[] {0,   1,   0,   2},    // 0 state (start state)
17             new int[] {2,  -1,   0,  -1},    // 1
18             new int[] {2,   1,   3,   0},    // 2
19             new int[] {-1, -1,  -1,  -1}     // 3 final state, no way out.
20         };
21
22         State[] States =
23         {
24            new State("You're in a room with a big zero painted on the floor.", "keys", "iPad"),
25            new State("You're in a bright yellow room."),
26            new State("This is a round room."),
27            new State("As you enter the room a rockfall closes the entrance!")
28         };
29
30         FSM result = new FSM(States, transitionTable, "north", "east", "south", "west");
31         return result;
32     }
33
34     private void describeSituation()
35     {
36         Console.WriteLine("{0}\n{1}\n{2}",
37                           theFSM.CurrentState.GetDescription(),
38                           theFSM.GetExitsDescription(),
39                           thePlayer.GetThings());
40     }
41 }
```

As already mentioned, the FSM now understands what event verbs should cause transitions, so on line 32 we set this up. Our cave system consists of just four rooms. One of them is initialized with two things, the others have nothing in them. The player is initialized with a few things in their inventory too.

If we look back at the `Main` method right at the top of this chapter, getting the game to play was a two-step process: instantiate the game (and we now have the code completed for that step), and then call `RunGameLoop` to enter the play loop.

Many games have a game loop, and it is worthwhile thinking in larger abstractions about what it needs to do. Each iteration of the loop needs two main steps:

- Get some input from the user and break it up (parse it) into its words,
- Respond to the user's command.

The game loop will run continuously until the user types *quit*.

It will be a good idea to use a `switch` statement with one case to cater for each of the possible actions that a user can do.

So we could start with a skeleton like this:

```
1  public void RunGameLoop()
2  {
3      while (true)
4      {
5          Console.Write("\nWhat do you want to do? ");
6          string response = Console.ReadLine().ToLower();
7          Console.WriteLine();
8
9          // Split the user's input into a verb and another word
10         string[] words = response.Split(new char[]{' '}, StringSplitOptions.RemoveEmptyEntries);
11         if (words.Length > 2)
12         {
13             Console.WriteLine("The most I can cope with is two words at a time!");
14             continue;
15         }
16         string theVerb = words[0];
17         string theObject = "";
18
19         // Respond to the user's command
20         switch (theVerb)
21         {
22             case "quit": return;     // leave the game.
23
24             case "help":
25                 Console.WriteLine("Valid verbs are go, drop, take, look, help, throw, read, quit, ");
26                 Console.WriteLine("and perhaps some others. Some verbs must be followed by another");
27                 Console.WriteLine("object word to make sense, e.g.  'go west', or 'read book'.");
28                 break;
29
30             default:
31                 Console.WriteLine("Huh?");
32                 break;
33         }
34     }
35 }
```

The game loop at line 3 runs “forever”, but the return at line 22 overrides that, leaves the method, and the game will end.

Lines 10–17 parse the user’s input and do a bit of error checking. We now need to expand that logic so that we check that the user does supply an object word for the verbs like “go” and “drop” that require one, and that they don’t supply an argument if the verb doesn’t require one.

```
1  List<string> verbsRequiringObjects = new List<string>() { "go", "read", "drop", "take", "throw" };
2
3      ...
4      string theVerb = words[0];
5      string theObject = "";
6      bool verbNeedsAnObject = verbsRequiringObjects.Contains(theVerb);
7      if (verbNeedsAnObject && words.Length == 1)
8      {
9          Console.WriteLine("{0} what or where?", theVerb);
10         continue;
11     }
12     if (!verbNeedsAnObject && words.Length == 2)
13     {
14         Console.WriteLine("I don't know how to do that.");
15         continue;
16     }
17     if (verbNeedsAnObject) theObject = words[1];
```

We keep a list of words that need object words. Lines 7–16 check and give errors for the two “invalid” cases: the user types “read” or the user types “help east”. At line 17, if an object is required (and is present) we can safely access the

`words` array at index position 1 and extract the object word. That completes our simple parser for the user input.

Now we can incrementally add cases to our `switch` statement for the rest of the verbs. But before we can do that, we need to remember that some of our methods can throw exceptions. So let's also wrap the whole `switch` statement in a `try ... catch`: (Notice that this is all still inside the game loop, so an exception won't end the game!)

```
1  try
2  {
3      switch (theVerb)
4      {
5          case "go":
6              theFSM.DoTransition(theObject);
7              describeSituation();
8              break;
9
10         case "drop":
11             thePlayer.RemoveThing(theObject);
12             theFSM.CurrentState.AddThing(theObject);
13             Console.WriteLine("OK.");
14             break;
15
16         case "look":
17             describeSituation();
18             break;
19
20         case "read":
21
22             if (theObject == "book")
23             {
24                 if (thePlayer.HasThing(theObject) || theFSM.CurrentState.HasThing(theObject))
25                 {
26                     Console.WriteLine("The title is 'Think Sharply with C#'. It looks quite boring.");
27                 }
28                 else
29                 {
30                     Console.WriteLine("There is no {1} here to read.", theObject);
31                 }
32             }
33             else
34             {
35                 Console.WriteLine("Nobody can read a {0}.", theObject);
36             }
37             break;
38
39         ...
40     }
41 }
42 catch (Exception ex)
43 {
44     Console.WriteLine(ex.Message);
45 }
```

Lines 5–8 handle our movement between the caves. Lines 10–14 transfer an item from the player's inventory to the room's inventory. We haven't yet written the logic for picking up things, but it should be quite similar. When asked to "read", we make sure that the thing we're asked to read is either in the player's inventory, or the room's inventory. And we currently only allow the user to read the book. But allowing them to read the iPad should be an easy change.

27.5. Summary

We've constructed an application with five different classes. The object-based approach has allowed us to break a fairly complicated application into manageable chunks. Together they collaborate to make a non-trivial application.

27.6. Exercises

1. Create your Console Application, add the classes you need, and cut and paste all the code from this chapter into the respective classes to make the game play.
2. The game class we presented here didn't implement all the verbs. Complete it.

3. You're lazy. Change the game so that we can just type "go n", "go e", "go s", "go w" instead of "go north", "go east", "go south", "go west".
4. You're even lazier. Change the game so that you can also just type "n", "e", "s", "w" instead of "go n", "go e", "go s", "go w".
5. Change the player class so that a player can only carry a maximum of four items at any one time. ("Your hands are full!")
6. Extend the cave system so that you can also go "up" or "down" and add a few more passages, rooms, and treasures.
7. Put some food in one of the caves. Allow the user to take, drop, or eat the food, ("Yummy!"). Don't let the user eat inappropriate things (use your imagination). And make sure that once something is eaten, it is gone.
8. As the game stands, we use short strings like "food" so that the user can easily refer to the item when typing input at the console. But perhaps having a key word "food" that maps to a more elaborate description like "a steaming hot chilli pizza" would add some spice to the game!
9. The Colossal Cave adventure game has a few magic spell words that transport you to another part of the cave. This requires "breaking" the rules of an FSM — you have to create a way to get the machine into a specific state without following the transitions that are available. Implement a magic word "plugh" that always takes you back to a fixed cave.
10. The Colossal Cave adventure game has some situations in which a rockfall might close down some passages, or open new passages. In our system, this would require "rewiring" the transition table a bit. Create a new state in the cave system — a treasure room — but make it initially unreachable from any other state. Then, on some trigger (maybe when the user tries to eat the book, or you may want to provide some dynamite that the user can light) you should provide an unexpected side effect that reconfigures the caves and opens a passageway to the treasure room.
11. The Colossal Cave adventure game has a torch with batteries that run out after a while. But in one of the caves there is a vending machine that dispenses new batteries. But the vending machine needs coins that have to be collected in other parts of the cave system. To implement features like this we would need to allow interactions between the "things" in our system. Presently, things don't react or interact with one another — they're just represented as simple strings that we can take, drop, throw, or eat. What changes would you have to make to the program to represent a thing as an object with its own internal behaviour and state?
12. The Colossal Cave adventure game has a dangerous troll that wanders about the caves in an unpredictable fashion. How could you add a troll to your game?

28. Inheritance

We've spoken a few times in the book about the *is-a* relationship: the idea that a `FileNotFoundException` is-a `IOException` which in turn is-a `Exception`, which is-a `Object`.

We sometimes use the terms **subtype**, **derived class**, **child class** or **descendant class** to mean the same as is-a. A `FileNotFoundException` is a subtype (or derived class, or child class) of `Exception`.

Conversely, we say `Exception` is a **supertype**, a **base class**, a **parent class** or an **ancestor class** of `FileNotFoundException`.

We say that a derived class *specializes* the base class.

The `Exception` type is important because it can be used in a `try ... catch` construct. But when we say some base type can be used in some way, we really mean that any of its derived (or child) types will also work in that same situation.

This is a very powerful idea: something that works for a base type T will also be able to work for any class derived from T.

We've seen this idea repeatedly in our GUI work too. All the controls we've used — `Canvas`, `Button`, `TextBox`, etc., on are derived from `UIElement`. A window knows how to position, layout, and display any `UIElement`, and how to interact with the control for mouse click events, key press events, giving the control focus, and so on.

So what is **inheritance**? When we derive a child class from a parent class, the child automatically inherits all the capabilities of the parent. So a `Button` can do everything that its ancestors can do. And it then gets some extra specialization — it might look different, or know about Click events.

All the GUIs we've built are specialized types of the `Window` class. So our own windows have their own buttons and canvases, but they're also able to be resized on the screen, they know how to respond to clicking the close button, they already know how to work with the Title bar, key presses, mouse movement, timers, etc. And we didn't have to do anything to get that all for free! It was all inherited.

Imagine if human inheritance worked like this: you always inherited your mom's capabilities. She might be a great athlete, a concert pianist, and a master chef. You'd start off already having those capabilities! (Unlike humans, C# classes only ever have one parent class!)

28.1. Let's do it!

We're going to create a new class that inherits from an existing class, and we'll give it some new capabilities that it doesn't already have.

We want a `TurtleGTX` type [1]. We'll start by adding two new methods to our specialized turtles. The `Jump(distance)` method will make our turtle hop the specified distance without drawing a line. And the `Spin()` method will spin our turtle around a few times and leave it facing in a random direction. These should be *additions* to what the turtles can already do, so we still expect our new turtles to be able to move forward, change the brush, stamp a footprint, and do everything else.

[1] GTX was a suffix used on earlier motor cars, standing for *"Grand Tourismo X"*. The GTX models had higher performance, more features, were fancier, and cost more. Check out some retro images of the 1971 Plymouth GTX!

We create a new class called `TurtleGTX`, add a few `using` directives and provide a constructor. So we start with this:

```
1  using System;
2  ...
3  using System.Windows.Media;
4  using ThinkLib;
5
6  namespace Fragments
7  {
8      public class TurtleGTX: Turtle
9      {
10         Random rng;
```

```
11
12          public TurtleGTX(Canvas playground, double homeX=50, double homeY=100)
13              : base(playground, homeX, homeY)
14          {
15              rng = new Random();
16              LineBrush = new RadialGradientBrush(Colors.LimeGreen, Colors.HotPink);
17              BrushWidth = 10;
18          }
19      }
20  }
```

Line 8, the definition of the class, says "this class inherits from Turtle", so `TurtleGTX` is-a `Turtle` and can already do everything that our ordinary plain turtles can do.

When we create instances of our new class the constructor at line 12 will be called. What is needed now is to *also* run our inherited constructor code — because it does the setup steps for those parts of the new object that are the "ordinary" turtle, and then we can do a few extra initialization steps that apply to the "extended" parts of the object that are only found in the fancier GTX models.

So line 13 calls the constructor code that was originally written in the base class. In a child class, the keyword `base` always refers to "as inherited from my parent". So `base(...)` means "call my constructor code as I initially inherited it", and `base.Forward(10);` means "call my `Forward` method, as I initially inherited it".

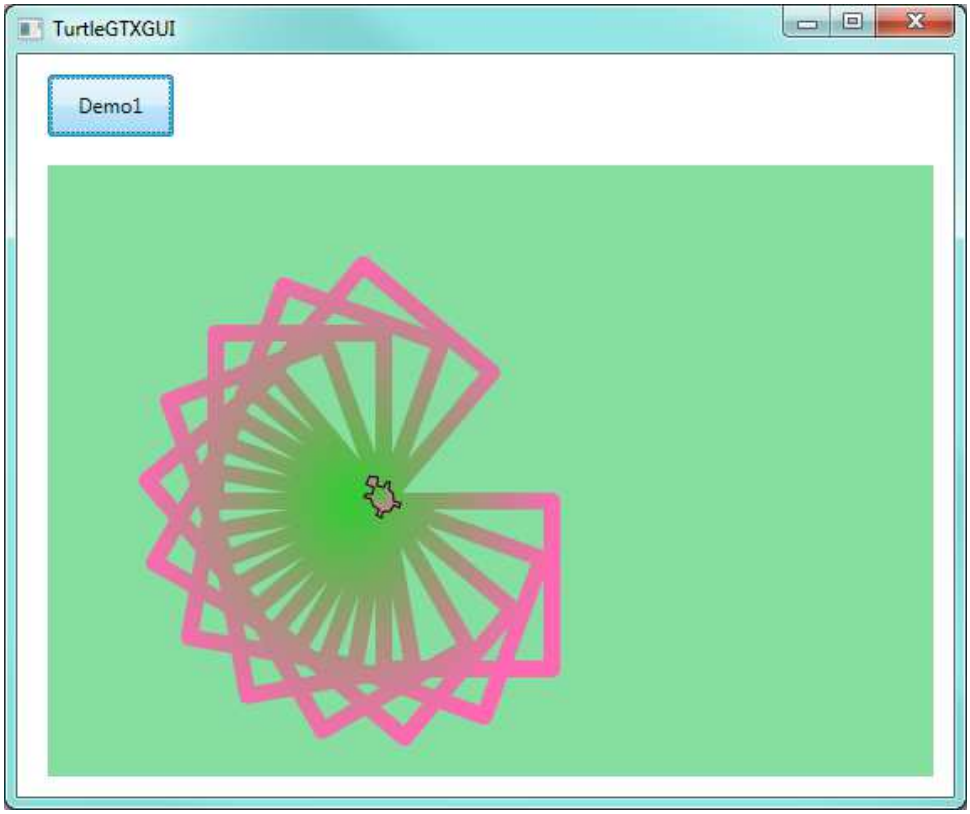
Lines 15–17 are executed after the base constructor completes its execution. We instantiate a random number generator object (we'll need it shortly for the `Spin` method we intend to add), and we change two of the default settings: whereas "ordinary" turtles come out of the factor with thin magenta brushes, we'll do the factory initialization of our GTX turtles with fat brushes and a fancy brush. This will help us to easily see that we're working with a GTX turtle, not a plain one.

Notice in lines 16 and 17: `LineBrush` and `BrushWidth` are properties of a turtle instance. So usually, we'd have to qualify them with an object, and say `Alex.BrushWidth`. But because of inheritance, these are now our own methods for our own object. We can call the method without qualification *because it is part of this class* — not because we wrote it explicitly, but because we inherited all the parent's public or protected members.

Let's set up our usual GUI with a canvas and a button, instantiate a turtle of type `TurtleGTX`, and get it to do some ordinary things.

```
1   public partial class TurtleGTXGUI: Window
2   {
3       TurtleGTX tess;
4
5       public TurtleGTXGUI()
6       {
7           InitializeComponent();
8           tess = new TurtleGTX(playground, 200, 200);
9       }
10
11      private void btnDemo_Click(object sender, RoutedEventArgs e)
12      {   // Draw three squares, and turn a bit after each one...
13          for (int i = 0; i < 3; i++)
14          {
15              // Draw a square
16              for (int side = 0; side < 4; side++)
17              {   tess.Forward(100);
18                  tess.Right(90);
19              }
20              tess.Right(20);
21          }
22      }
23  }
```

The only changes now are the type of tess, in lines 3 and 8. Otherwise, everything that we inherited from the plain turtle type works exactly as before. Clicking the button a few times produces this:



Here's a further observation: in our first chapter on methods we created a method to get a turtle to draw a square. Let's re-use that code, and replace lines 11-22 above with this new code:

```
1   private void drawSquare(Turtle t, double sz)
2   {
3       for (int side = 0; side < 4; side++)
4       {
5           t.Forward(sz);
6           t.Right(90);
7       }
8   }
9
10  private void btnDemo_Click(object sender, RoutedEventArgs e)
11  {   // Draw three squares, and turn a bit after each one...
12      for (int i = 0; i < 3; i++)
13      {
14          drawSquare(tess, 100);
15          tess.Right(20);
16      }
17  }
```

It still works. But, notice a very subtle thing: the type of argument `tess` line line 14 is `TurtleGTX`, but the type of the parameter in `drawSquare` is `Turtle`. Yet the compiler doesn't give an error, and the code works. This is because a `TurtleGTX` is-a `Turtle`.

A object of a descendant type can be assigned to, or be used as if its type is an ancestor type. We can use `tess` as a plain old turtle, assign her to a plain old turtle type of variable, pass her to a parameter of type `Turtle`. So any methods we already have to work with `Turtle` can automatically also work with any types of objects that are descendant from `Turtle`.

This feature is called **subtype polymorphism**. Polymorphism means many (poly) types or forms (morph). The `drawSquare` method is polymorphic in that it can work with different types of objects, just as long as they're all descendant types of `Turtle`.

OK. We are now ready for making some more serious improvements to the GTX version of the turtle.

28.2. Adding new behaviour

Adding new behaviour is as easy as adding some new public methods to the `TurtleGTX` class. Let's implement the two methods we wanted:

```
public void Jump(double distance)
{
   if (BrushDown)
   {
        BrushDown = false;
        Forward(distance);
        BrushDown = true;
    }
    else
    {
        Forward(distance);
    }
}

public void Spin()
{
    int theta = rng.Next(360);
    for (int t = 0; t < 199; t++)
    {
        Right(theta);
    }
}
```

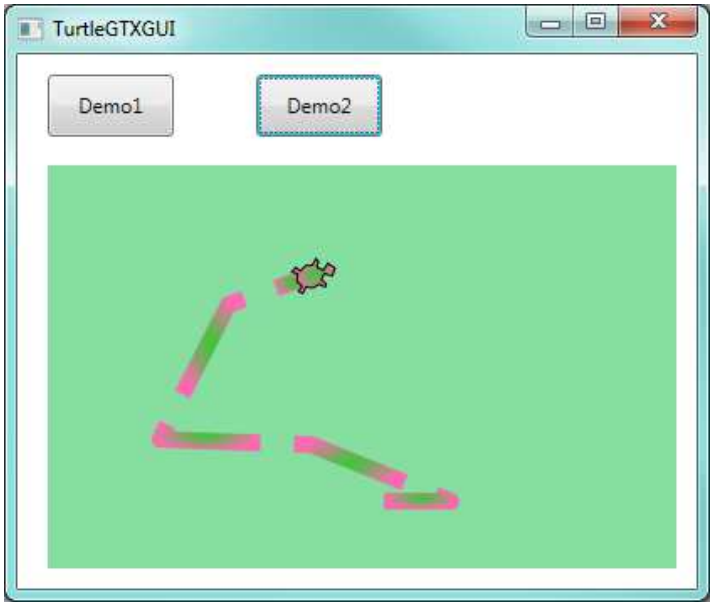
If the brush is down, jumping requires picking up the brush and putting it back down after moving. To spin the turtle a random amount, we pick a random angle and then repeatedly turn by that angle, so that we can briefly see the turtle spinning on the playground before it sets off on its (random) new heading.

We go back to the GUI, add another button and another handler, and this test code:

```
private void btnDemo2_Click(object sender, RoutedEventArgs e)
{
    tess.Forward(40);
    tess.Spin();
    tess.Forward(10);
    tess.Jump(20);
    tess.Forward(20);
}
```

Now we can use the extended features of the new class (and, if you watch IntelliSense, it knows about them too.)

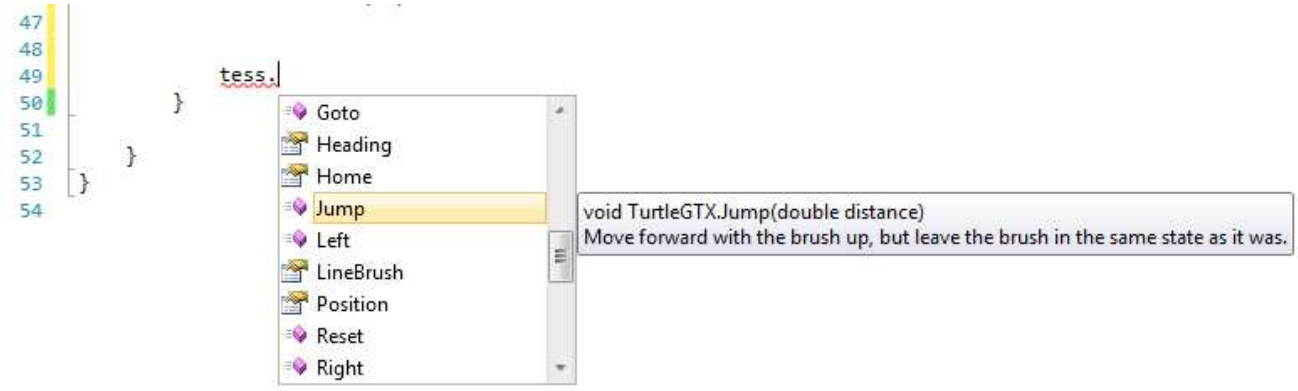
Clicking the new button a few times gives us something like this ...



Let's make one more refinement to complete this section. The IntelliSense for the new `Jump` and `Spin` methods is not helpful. We usually expect some help explaining briefly what a method does, and what it's parameters and return values are. In C#, we can write special comments (they start with three forward slashes `///`) before any definition. These will be picked up by the compiler and become part of the IntelliSense mechanism. (You'll find in Visual Studio that the editor understands the three slashes, and will write the skeletons of the comments for you.) So we go back to the TurtleGTX class now, and add suitable comments:

```
1  /// <summary>
2  /// A fancy type of Turtle to illustrate some ideas about inheritance.
3  /// </summary>
4  public class TurtleGTX: Turtle
5  {
6      ...
7
8      /// <summary>
9      /// Move forward with the brush up, but leave the brush in the same
10     /// state as it was.
11     /// </summary>
12     /// <param name="distance">How far to hop forward.</param>
13     public void Jump(double distance)
14     ...
15
16     /// <summary>
17     /// Spin the turtle a few times and leave it facing in a random heading.
18     /// </summary>
19     public void Spin()
20     ...
```

With these comments added (notice we've commented above the class and the two new methods) we'll see the results in IntelliSense:



28.3. Changing existing behaviour

We've added some new behaviour to our `TurtleGTX` type. But so far, all the underlying behaviour of the base Turtle class remains unchanged.

There is a second use for inheritance — to **override** behaviour from the base class, so that our child class objects work differently.

Where might this be useful?

Suppose we have a spelling checker with a `CheckSpelling(string word)` method. Perhaps we could make a specialized spelling checker that knows local slang or domain-specific words. So we'd like to change how `CheckSpelling` goes about its business, by making it also look in our slang dictionary.

Or consider a `FileStream` object — it allows reading and writing to a stream that it connects to an underlying file. Typical file systems have some potential security risks. If your program writes sensitive passwords or data to disk, some other program might be able to access it or corrupt it. So Windows provides what is called *Isolated Storage* — a private storage area that belongs to a single program and user. (Other users will get their own isolated storage, and other applications get their own isolated storage too.) So there is a class called `IsolatedStorageFileStream` that is

derived from `FileStream`. It can only open, create, read and write data in isolated storage. So in this case, it must *restrict* what the base class is capable of.

You can think up other examples too. You might want to inherit from a Web Browser control (there is one in WPF), but modify some existing behaviour by overriding its methods so that your users can only access "approved" web sites.

28.4. Let's do it!

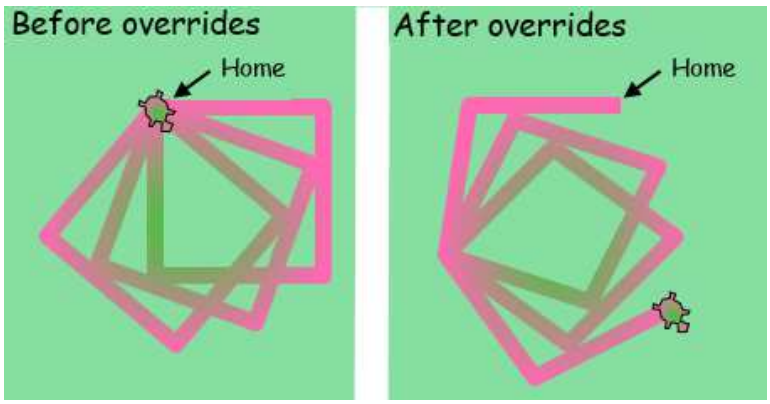
We'll continue with the `TurtleGTX` class. Unfortunately, our turtle has just become a teenager [2] and does the opposite of whatever they're told to do. And they'll only complete 90% of whatever is asked. So we'll override the `Left` method to make it turn `Right` instead, and vice-versa, and `Forward` will move backward. We add the following methods to the class:

[2] This is not a bias against teenagers. But it makes a fun and understandable way to explain some pretty complicated ideas...

```
1  public override void Left(double degrees)
2  {
3      base.Right(degrees*0.9);
4  }
5
6  public override void Right(double degrees)
7  {
8      base.Left(degrees*0.9);
9  }
10
11 public override void Forward(double distance)
12 {
13     base.Forward(-distance*0.9);
14 }
```

The `override` keyword says that we're changing our existing behaviour that we inherited from the parent class. In our modified interpretation of what `Left` should do, at line 3, we call our inherited `Right` method, (but we only turn 90% of what was requested). So calling `Left(90)` on a GTX turtle will turn it to the right by 81 degrees. Similarly, we negate the distance: our new `Forward` method calls our inherited `Forward` method, but with a different argument.

We go back to the first demo that drew three squares. On the left we show how it used to work. On the right we show what the same calling code produces now after we've overridden the behaviour of these three methods.



Now let's make a really important observation, one which is key to having a good understanding of inheritance and behaviour that is overridden. Our code made use of this `DrawSquare` method.

```
1  private void drawSquare(Turtle t, double sz)
2  {
3      for (int side = 0; side < 4; side++)
4      {
5          t.Forward(sz);
6          t.Right(90);
7      }
8  }
```

As we remarked earlier, the parameter type here is `Turtle`. Because of subtype polymorphism, we can pass a `TurtleGTX` argument to it.

So what does the method call at line 5 actually do? Does it use our modified `Forward` method (that actually goes backwards), or does it use the original `Forward` method because `t` is defined as a `Turtle`? Or, using our silly analogy, does the teenager still behave like a teenager even though they're here in the role of "person"?

Yes, `t` is still a teenager! `t` references an object: that object is a `TurtleGTX`. The fact that the variable can reference a `Turtle` doesn't change what is in memory — in memory, we have a `TurtleGTX`, with its own behaviour and methods. So at line 6, its `Right` method actually turns to the left.

28.5. A popular override

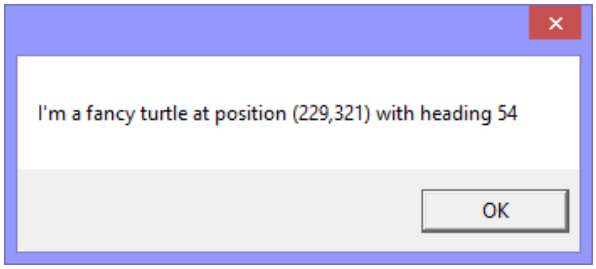
Every type ultimately inherits from the `object` base class. So every object that we can possibly create in C# has four methods that it inherits from `object`. One of those is `ToString()`. Whenever C# needs to convert an object to a string (as it needs to do if we convert the object to a string using, say, string formatting), this method gets called to do the conversion. Let' see what happens when we call `ToString()` on tess:

```
1   MessageBox.Show(tess.ToString());
```

We'll get our message box popping up, and showing the *type* of tess. That is what the default behaviour is. But, of course, we could override it for our TurtleGTX.

```
1   public override string ToString()
2   {
3       string msg = string.Format("I'm a fancy turtle at position ({0:f0},{1:f0}) with heading {2:f0}",
4               this.Position.X, this.Position.Y, this.Heading);
5       return msg;
6   }
```

Now we get something more special:



It is quite usual to override the `ToString()` method and to have it report some useful information about its internal state.

28.6. A touch of sanity

Wouldn't it be nice if we could inherit from our bank account class, and override its withdrawal method so that each time we withdraw money we could call the inherited deposit method instead!

There are two security mechanisms that help.

- We can mark a class as `sealed`. It is not possible to derive any subclasses from a sealed class. Many of the .NET framework classes are sealed, so that we cannot maliciously or erroneously change their behaviour. Try to inherit from the `string` class, and see what happens. Or read the documentation for the `string` class.
- You cannot override a method in a class unless the class definition itself catered for that when it was originally written. A class writer can mark individual methods or properties in a class with a keyword `virtual`. This means "can be overridden by my descendants". So the Turtle class has given us permission to override methods like `Left` and `Forward`, but at the same time there are some of its methods (`FlushToPlayground` is one) that it does not allow us to override.

28.7. Glossary

ancestor class

Same as base class.

base class

A class which is used as a basis for another more specialized derived class.

child class

Same as derived class.

derived class

A class which defined in terms of another class by inheriting the functionality of the base base.

descendant class

Same as derived class.

hierarchy

A tree-like structure. In this chapter, we've been interested in how classes are related to each other, so the hierarchy is the "family tree" of classes.

inherit

To receive capability and behaviour from a base class.

instance

An object whose type is of some class. Instance and object are used interchangeably.

`object`

This is the ultimate base class of all other classes in the .NET framework, and in C#. It is the root of the class hierarchy from which every other class is derived. (In the framework, its name is spelled with a capital 'O', in C#, with a lower-case 'o'.)

`override`

A C# keyword that indicates that a method definition is modifying the behaviour that it inherited from its base class.

parent class

Same as base class.

polymorphism

Able to work with many types. The origin of the word is from "poly) (many) and "morph" (forms).

sealed class

A class that cannot be derived or used as a base for a new class.

subtype

Same as derived class.

subtype polymorphism

Polymorphism that is possible because anything that works for type T will automatically also work for any subtypes of T.

There are other mechanisms for polymorphism too. Contrast this idea to the "parametric polymorphism" we get with generic types, e.g. `List<Turtle>` or `List<Button>`. There is no subtype relationship between `Turtle` and `Button`, yet `List` can handle the different types.

supertype

Same as base class.

28.8. Exercises

1. Look up the `IO.Stream` class at http://msdn.microsoft.com/en-us/library/system.io.stream.aspx, expand its inheritance hierarchy and determine the key purpose or responsibilities of at least three of its derived classes.
2. Add an `Odometer` property to `TurtleGTX` so that a turtle keeps track of the mileage (total distance) it has travelled since it was created. To keep it simple, only count the distance from the `Forward` method.
3. If you travel backwards (`tess.Forward(-50)`) does your odometer count up, or does it count down? What should it do? Fix it. (On older cars with mechanical odometers, some unscrupulous car dealers run the odometer backwards to improve the resale value. Drive a car backwards down the street, and determine whether the odometer counts forward or backwards.)
4. Our TurtleGTX has a brush that it uses to draw its lines. But brushes need ink. Give the turtle a supply of ink. On each move that draws a line (i.e. only when the pen is down) consume some of the ink supply. When the ink runs out, throw an "Ink Cartridge Empty" exception. Any attempt to make the turtle draw after that should throw the exception. Provide a way for the client to interrogate how full the ink tank is, and provide a `ChangeInkCartridge` method that can fix our broken turtle. (Bonus exercises could be to make the rate at which the ink gets consumed depend on the width of the brush, or, like inkjet printers, to have differernt ink cartridges for Cyan, Magenta, Yellow and Black ink supplies. The possibilities seem ink-credible.)
5. In our Recursion chapter we learned about great-looking Koch fractals. Our teenage TurtleGTX becomes fascinated by the nesting of the patterns. So whenever he/she is asked to move `Forward` by any distance, the turtle playfully does a detour, making a Koch order 2 curve instead. Provide an override for the `Forward` method to achieve this.
6. Add to the previous exercise: provide a `KochOrder` property that the user can set. Give it an initial default value of zero: this will mean the turtle takes no detours, since a Koch line of order 0 is just a straight line. Then if the user sets the value of KochOrder to be 3, the turtle will detour with order 3 Koch curves.

29. Dictionaries

Arrays, strings and lists all have one thing in common: the elements can be accessed by their index position.

Dictionaries are different. They associate **values** with **keys**. The keys can be one type, the values can be another (possibly different) type. Every entry in the dictionary is a *pair* with two parts: the key, and its associated value. We'll refer to the pair as a key-value pair.

Let's begin with a simple dictionary that keeps track of how many emails we have have sent to others. The key will be the email address that we've sent email to, the value associated with each key will be the number of emails we've sent to that address since we began counting.

```
1  private void btnDemo1_Click(object sender, RoutedEventArgs e)
2  {
3      Dictionary<string, int> emailCounter = new Dictionary<string, int>();
4
5      emailCounter["a.n.other@ru.ac.za"] = 3;
6      emailCounter["joe123@google.com"] = 12;
7      emailCounter["president@gov.za"] = 2;
8      emailCounter["abbey@foxpictures.com"] = 5;
9
10     emailCounter["joe123@google.com"] += 1;
11
12     foreach (string k in emailCounter.Keys)
13     {
14         txtResult.AppendText(string.Format("Key {0} has value {1}.\n", k, emailCounter[k]));
15     }
16     txtResult.AppendText(string.Format(
17         "There are {0} key-value pairs in the dictionary.\n", emailCounter.Count));
18 }
```

Take note:

- In line 3 we define a generic dictionary and supply the type of the keys, `string`, and the type of the values, `int`. On the right-hand-side of the assignment we instantiate a new empty dictionary.
- Lines 5-8 add new items to the dictionary. But notice now that the "index" is a key — a string in this case.
- Line 10 changes the value associated with a key, by incrementing it.
- In line 12, we see that we can get an enumerator for the keys, so `foreach` can look at each key in turn. Each key is a string, which is assigned to `k` in the loop.
- Line 14 outputs each key and the value associated with it.
- Line 17 shows that we can use the `Count` property to discover how many pairs are in the dictionary.

Here's what we get:

Hashing

Our output here shows the pairs in the same order as we inserted them, but this is misleading! In general, the order in which a dictionary stores its pairs is unpredictable. (So the order in which we'll get them delivered by a `foreach` becomes unpredictable.) C# uses complex "hashing" algorithms, designed for very fast access, to determine where the key-value pairs get stored in memory. In small dictionaries like ours, C# might choose a list, but for large dictionaries it will switch techniques. If you want to learn more about hashing, Wikipedia has some interesting information!

29.1. Dictionary Initializers

Similar to what have for integer, doubles, arrays, strings, and lists, there is also some useful initializer syntax in C#. This lets us provide some pairs when we instantiate the dictionary. Lines 5–8 in our example can also be coded like this:

```
1      Dictionary<string, int> emailCounter = new Dictionary<string, int>()
2          {  {"a.n.other@ru.ac.za",    3},
3             {"joe123@google.com",    12},
4             {"president@gov.za",      2},
5             {"abbey@foxpictures.com", 5} };
```

If we try to get the value associated with a key that doesn't exist in our dictionary, we'll get a `KeyNotFound` exception. So

```
1      int n = emailCounter["bill@microsoft.com"];  // Exception
```

is not going to work!

29.2. Dictionary Methods

Here are a couple of useful methods:

- `ContainsKey(key)` can (safely) test whether a key is in a dictionary.
- `Remove(key)` removes the key and its associated value from the dictionary.
- `Clear()` will remove all key-value pairs and leave the dictionary empty.

A dictionary is a reference type. Because it can be updated, we need to be aware of aliasing. Whenever two variables refer to the same object, changes to one affect the other.

Dictionaries and lists are your friends!

As you learn more C# you'll discover more exotic ways of organizing your collections: we've briefly covered lists and dictionaries, but you'll find queues, stacks, sets, deques, and possibly others.

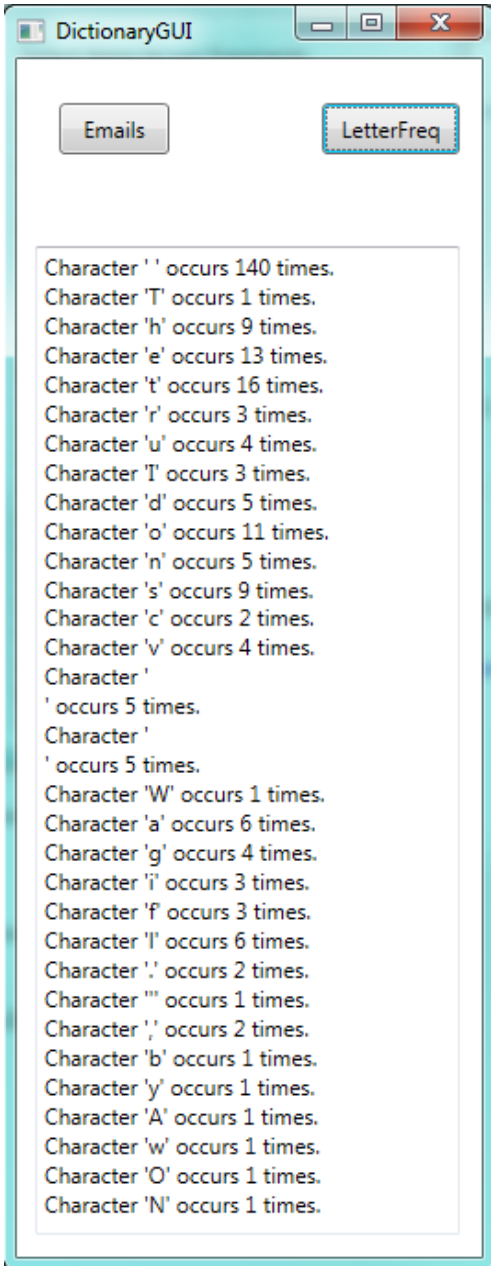
But most of the time the most practical data structure of choice will be either a list (if you don't need key-value associations), or a dictionary (if you do).

If you want some extra reading on C# dictionaries, try http://www.dotnetperls.com/dictionary.

29.3. Counting letters in a string

We'll build a frequency table of the letters in a string. Such a frequency table might be useful for compressing a text file. Because different letters appear with different frequencies, we can compress a file by using shorter codes for common letters and longer codes for letters that appear less frequently.

Dictionaries provide an elegant way to generate a frequency table:




```
1  private Dictionary<char, int> letterFreqs(string theText)
2  {
3      Dictionary<char, int> result = new Dictionary<char, int>();
4      foreach (char c in theText)
5      {
6          if (result.ContainsKey(c))
7          {
8              result[c]++;
9          }
10         else
```

```
11          {
12              result[c] = 1;
13          }
14      }
15      return result;
16  }
```

We start with an empty dictionary. For each character in the string we test whether it is the first time we've encountered it (and it becomes a new association in our dictionary at line 12), or we bump (increment) the counter of the number of times we've already seen it (line 8).

With the code below to exercise our method we get the output here.

```
1   private void btnLetterFreq_Click(object sender, RoutedEventArgs e)
2   {
3       string poem =
4   @"   The truth I do not stretch or shove
5       When I state that the dog is full of love.
6            I've also found, by actual test,
7               A wet dog is the lovingest.
8
9                                       Ogden Nash";
10
11      Dictionary<char, int> freqs = letterFreqs(poem);
12      foreach (char k in freqs.Keys)
13      {
14          txtResult.AppendText(string.Format(
15              "Character '{0}' occurs {1} times.\n", k, freqs[k]));
16      }
17  }
```

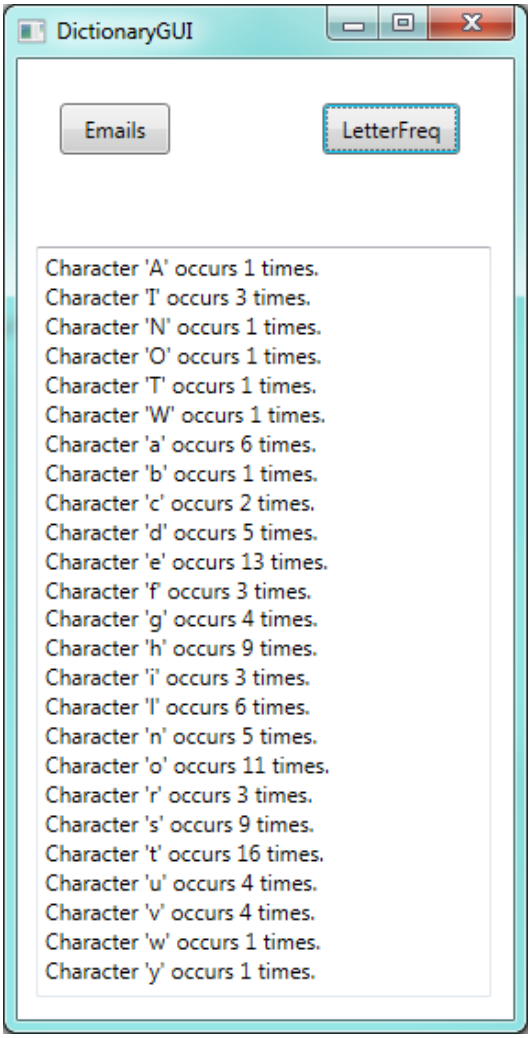
Those broken lines in the middle of the output correspond to the carriage return and newline characters at the end of each line of the poem. We'll change our frequency counter to ignore them. And it might be more appealing to get back a frequency table in alphabetical order.

29.4. Slightly fancier dictionaries

Dictionaries (and lists) are important enough that the .NET Framework gives us a number of different flavours of each that we can choose from. This allows a smart programmer to choose a type that is most effective for the problem.

In the `System.Collections.Generic` namespace (we're already using it for type `Dictionary`) we also find a `SortedDictionary` type. And if we want more choice, the `System.Collections.Specialized` namespace offers us a `ListDictionary` type, an `OrderedDictionary` type and a `StringDictionary` type.

Let's put some changes into our code:




```
1  private IDictionary<char, int> letterFreqs(string theText)
2  {
3      SortedDictionary<char, int> result = new SortedDictionary<char, int>();
4      foreach (char c in theText)
5      {
6          if (char.IsLetter(c))
7          {
8              if (result.ContainsKey(c))
9              {
10                 result[c]++;
11             }
12             else
13             {
14                 result[c] = 1;
15             }
16         }
17     }
18     return result;
19 }
```

- At line 6 we test that `c` is a letter before we count it.
- At line 3 we use a `SortedDictionary` instead of a plain one. This guarantees that when we use the `foreach` we will get the keys out in sorted order. Otherwise all the code is the same.
- At line 1 we've done something special: we're returning a type called `IDictionary`. We're going to talk about that in the next chapter.

We make one change to the calling code, — change the definition at line 11 to also be an `IDictionary` instead of `Dictionary`, and we get the new output shown here. (Join the dots... recall that for characters, all

upper-case letters are ordered before the lower-case letters).

You could lower-case the string if you wanted capitals and small letters to be treated equivalently. There are also advanced options (beyond the scope of this book) to set up a dictionary that uses our own "comparer" method that we supply. That would give us complete control over decisions about when two keys are considered less than or equal to each other.

29.5. Dictionary, array, or list?

If we think about it, lists and arrays are also "mappings" of keys to associated data. An array of the names of the days of the week would have position 0 associated with "Sunday", position 1 with "Monday", and so on.

In an array or list the index (or key) must always be an integer and the lowest index is always zero. For dictionaries this is not the case.

So, specifically for the situation when our keys are integers, we should ask *"Which of list, or array, or dictionary, will be a good choice?"*.

So here are two sample problems, let us think about the best data representation for each:

> a) We want to check whether a random number generator really does give a uniform distribution of numbers in the range [0,100). We'll generate ten million random numbers and count how many times each one occurs. For a uniform generator, we'll expect the frequencies should be close to each other. (And our statistics wizards will be able to define exactly what is meant by "close enough".)
>
> b) People work in a building that uses an access card to allow them through specific doors. The security card uses a date of birth followed by two random digits, for example 1994052674, as a key, and the associated data is a string of characters describing the doors they can open — "A103,B12,Foyer,North Seminar Room,Stairwell".

In the first case the keys are "dense", and there is no need for a list that can grow or shrink dynamically. So an integer array of 100 elements sounds ideal. Each time we generate a new random number we can immediately know which array element to update. So `int[] freqs = new int[100];` will be a great way to store this data.

In the second problem the nature of the keys is very different. We might only have eighty people who can access the building, but their keys are all very large integers, and far apart from one another. If we plotted all the keys on a number line they would be very "sparse". In this case, a `Dictionary<int, string>` would be a better choice.

Will the large access code fit into an `int`?

We should double-check that our `int` type is suitable for storing large access codes like this. The biggest number that can be stored in a 32-bit integer is 2,147,483,647.

So yes, our codes will fit, but our program could go wrong when someone born after 2147 needs access to our building.

In the first case, we, the programmers, are really determining where to store the elements. In the second case, we've handed that problem over to the dictionary.

29.6. Glossary

dictionary

A collection of key-value pairs that maps from keys to values.

key

A data item that is *mapped to* a value in a dictionary. Keys are used to look up values in a dictionary. Each key must be unique across the dictionary.

key-value pair

One of the pairs of items in a dictionary. Values are looked up in a dictionary by their key.

29.7. Exercises

1. Perform a frequency count on randomly generated numbers in the range [0,10000). Do it twice: once with an array, and once with a dictionary. Devise an experiment to compare the performance of the two alternatives. Try to control for the overheads of generating the random numbers and running the loops, so that your experimental results become an accurate measure of the speeds of the underlying storage mechanisms.
2. Suppose every computer lab user has a usercode like g09m1234. Associated with the usercode is a `UserProfile` object that holds their name, permissions, desktop preferences, and course groups that they belong to. Give a type definition for a dictionary that could maintain the mapping between the usercode and its profile.
3. In our "List and Array Algorithms" chapter we worked extensively with *`Alice's Adventures in Wonderland`*. Re-use as much of that code as you can, and produce an alphabetic frequency count of the words in the book, something like this:

   ```
   Word              Count
   ======================
   a                 631
   a-piece           1
   abide             1
   able              1
   about             94
   above             3
   absence           1
   absurd            2
   ```

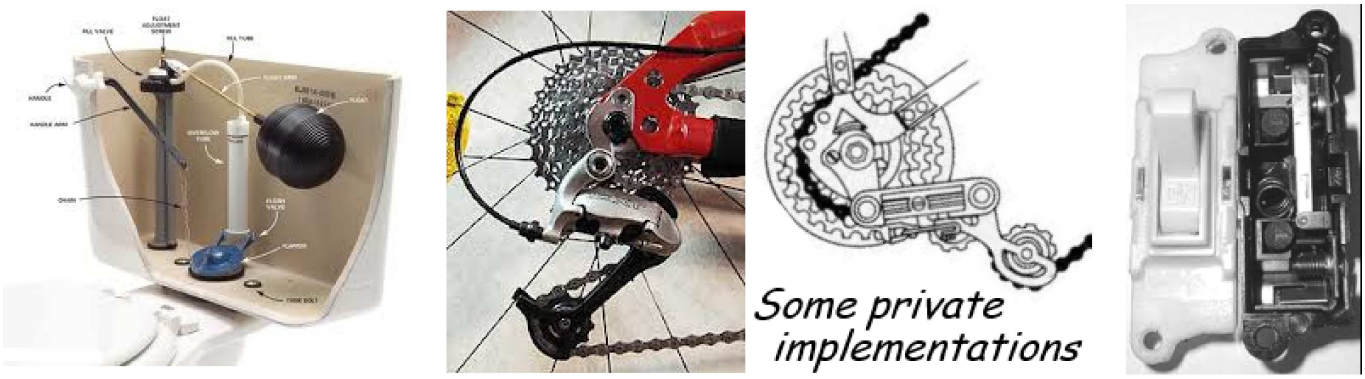
   How many times does the word "alice" occur in the book?
4. What is the longest word in Alice in Wonderland? How many characters does it have?

30. Interfaces

30.1. The Bigger Picture

Up until this point in our book, an object's *type* has determined its state and behaviour, and how we use it. Because `tess` is a Turtle, we can call `Turn` and `Forward` methods, or get the `Heading` property. Because `playground` is a `Canvas`, we can set its background brush or get its size.

The word *interface* is used with two different meanings in programming. The real-world meaning is *"the way we interact with an object"*. This is separate from *"how it works"*, or its implementation. In our classes we've made some members public (that's its interface to the outside world), and some members private (that's how we chose to implement the interface).



So when our boss says *"What's the interface to the Random class?"*, he is probably asking us to describe its public methods and members.

In C# (and some other languages, notably Java) we have a second more formal meaning the word. An **interface** is a language construct which contains some method signatures (and possibly a few other things) that defines a contract.

Object-oriented thinking in programming arose because people looked at the world around them, saw objects, grouped them into types, and realized that they had certain attributes, or states, and were capable of different behaviours. So we thought that would be a good way to organize software too!

But in real life, its not just "type" that determines how things can be used. Objects can *be* one type of thing, but also have different "roles". A person might have roles like a "parent", or a "chess player" or a "student".

So let's look at some devices that we use every day: our digital camera, our smart-phone, our Internet provider, our USB flash stick, our portable music player, our TV, our tablet PC, our alarm clock, our FM radio.

They're all different types of things, but some of them can perform in roles that others can't. Which of the devices can act in the role of "memory device for storing a file"? Which can act in the role of "music player"? Which can act in the role of "video player"? Which of these devices can act in the role of "camera" and take a photograph? Which can act as a "email reader", a "calculator", an "alarm clock" or a "radio receiver"?

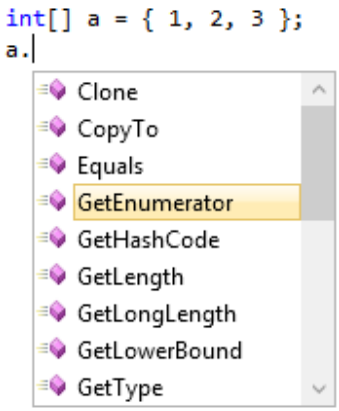
So in programming, in addition to defining something's type, (i.e. what it *is*), we want some extra mechanism to describe and add extra roles or different ways in which it can be used.

This is what a formal programming **interface** is all about: it describes a *role* that an object can play when it interacts with other objects.

When we plug our phone into our computer's USB port and copy some files into its memory, our computer doesn't need to know the *type* of the object we've plugged in — whether this is a phone, an external hard drive, or a camera, or a USB flash stick. All it needs to know is that this object is capable of the role "memory device for storing a file".

It turns out that we've seen this repeatedly already in our book, but we just have not fully identified where it has been happening.

Let's think about the `foreach` loop. We use it with strings, with arrays, with lists, with dictionary keys, etc. In C#, a class needs to provide a method called `GetEnumerator`. It returns an object with methods that allow `foreach` to get at every item in turn. So if we look at IntelliSense for a string, an array, a list or a dictionary, we'll always find a method called `GetEnumerator`.



This is the magic that lets the `foreach` traverse different types of things: they can all play the particular role required by `foreach`, even though they're all objects of fundamentally different types.

For another take on Interfaces, see Dave's page
http://www.hitthebits.com/2012/11/what-are-interfaces.html

30.2. Back to the low-level code

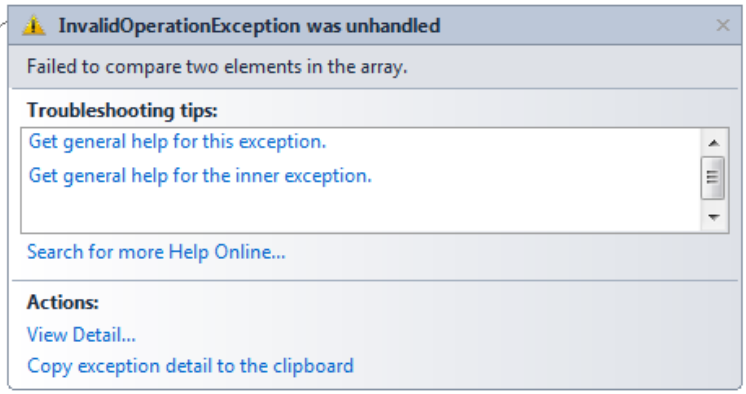
Learning to define and implement our own interfaces is a bit beyond the scope of this book. But we will look at one or two interfaces that are already defined in the .Net Framework, and see some ways in which we can use them.

An interface is really a "contract" between a class, and the "consumer" that wants to use the class. In C#, this contract specifies what members must be implemented in order to fulfil the contract.

30.2.1. Example 1

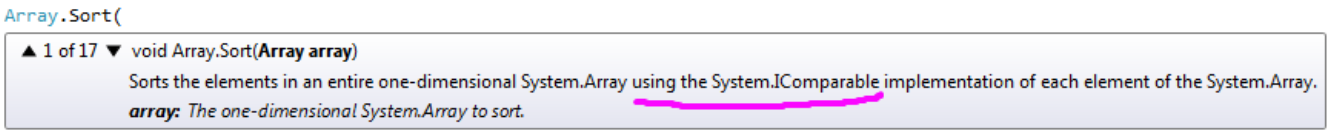
In our chapter on list and array algorithms we sorted an array of strings using `Array.Sort`. We also used this in the Queens algorithm, where we shuffled our queens by using random integers as the keys for the sort.

```
27          private void btnDemo1_Click(object sender, RoutedEventArgs e)
28          {
29              int[] nums =        { 12, 4, 7 };
30              string[] names =    { "Joe", "Bill", "Mary" };
31              DateTime[] times =  { new DateTime(1995,07,24), new DateTime(1995,07,11), new DateTime(1993,07,24) };
32              Rectangle[] rects = { new Rectangle(), new Rectangle(), new Rectangle() };
33
34              Array.Sort(nums);
35              Array.Sort(names);
36              Array.Sort(times);
37              Array.Sort(rects);
38          }
39      }
40  }
41
```



Take a close look at this code: we've created four arrays, three of them were able to be sorted just fine, but we've got a run-time error for the array of Rectangles. It failed to compare two elements of the array. Why?

Here's a hint from IntelliSense:



What the message is telling us is that it doesn't mind what type of objects are in the array, but whatever they are, they must have an `IComparable` interface. The `IComparable` contract specifies that there must be a method called `CompareTo` (recall that we've used this before when we used it to compare strings to each other).

So ints, strings, chars, DateTime have `IComparable` interfaces, (and, by contract, they therefore must have a `CompareTo` method). If we know how to compare them, we know how to sort them!

But we don't have a way to compare `Rectangle` objects to each other. `Rectangle` does not implement the interface `IComparable`.

When we define our own classes we need to ask whether it makes sense to compare our new types of objects to each other. For example, it might be sensible to allow `Student` objects to be ordered by student numbers. In that case we might decide that our class should implement `IComparable`, and we'd be required to write a `CompareTo` method in our new class to achieve this.

30.2.2. Example 2

Here is a contract of what an object needs to do to be able to fulfil the role `IDictionary`. (It is a convention in C# to always name interface contracts with a name starting with a capital I.)

```
1  public interface IDictionary<TKey, TValue> :
2                   ICollection<KeyValuePair<TKey, TValue>>,
3                   IEnumerable<KeyValuePair<TKey, TValue>>,
4                   IEnumerable
5  {
6      ICollection<TValue> Values { get; }
7      TValue this[TKey key] { get; set; }
8      void Add(TKey key, TValue value);
9      bool Remove(TKey key);
10     bool TryGetValue(TKey key, out TValue value);
11 }
```

Notice the keyword `interface` in line 1.

In the contract, all we have are member definitions and type signatures. So the contract specifies "what" has to be available, not "how" any particular class chooses to do it.

Lines 2–4 says that in order to act in the role of `IDictionary`, we've also got to be able to act in the role of `ICollection` and `IEnumerable`. Those contracts have a few more methods and members that are required. But more importantly, it shows that interface contracts can depend on, or inherit, contract requirements from other interfaces.

Now when we (or the Microsoft developers) write a new class, we can provide the "promise" side of the contract. We can say "We're defining a new Turtle class, and we want our turtle objects to also be able to act in the role of a dictionary, as specified by the `IDictionary` interface". So the class *implements* the interface.

Now the magic happens. If we have a turtle object, and if turtles can act as dictionaries (our turtles in this book cannot, by the way), then code like this would be possible:

```
1  Turtle tess = new Turtle(...);
2
3  IDictionary<string, int> tdict = tess;
4  tdict["distance covered"] = 25;
5  tdict["year of manufacture"] = 2013;
6
7  foreach (string k in td.Keys)  ...
```

Line 3 is the magic line. It says *"we're not interested in interacting with tess as a turtle, we're only interested in interacting with the object in terms of its IDictionary role"*. So we can define variable `tdict` so that its type is the *interface* type. Now it is possible to assign any object to variable `tdict`, provided it knows how to fulfil the role demanded by the interface.

30.3. Three flavours of polymorphism

Recall that polymorphism means "able to work with many types".

We've used subtype-based polymorphism: a method that requires a `Turtle` argument can also work for a subtype of `Turtle`, e.g. our `TurtleGTX`. This is what inheritance gives us.

The second type we've seen is parametric polymorphism: we use type parameters in `List<T>` or in `Dictionary<K,V>`. Another name for parametric polymorphism is *generic*.

In this chapter we've now seen *interface-based polymorphism*. In the code example above, we're able to treat tess as a dictionary. So all the dictionary methods are able to work with many types.

30.4. Type testing and casting

Sometimes we might need to "undo" the polymorphism. We have an object that we know is some type of `Turtle`, but we'd really like to a) know if it really a more special kind of `TurtleGTX`, and b) if it is, take advantage of its extra capabilities.

So let's go back to our `DrawSquare` method from the Inheritance chapter. We saw there that it works with any turtle. Now we'll add this new requirement: if the turtle has extra capabilities, (i.e. it is really a `TurtleGTX`), then get it to spin on each corner as it is draws the square.

```
1  private void drawSquare(Turtle t, double sz)
2  {
3      for (int side = 0; side < 4; side++)
4      {
5          t.Forward(sz);
6          t.Right(90);
7          if (t is TurtleGTX)
8          {
9            TurtleGTX tgtx = (TurtleGTX) t;
10           tgtx.Spin();
11         }
12     }
13 }
```

Line 7 is a **type test**. It allows us to ask whether this turtle `t` is actually a more derived (more capable) `TurtleGTX`. Line 9 defines a new variable that can reference a `TurtleGTX`. The type name in the parentheses on the right is called a **cast** or a **type cast**. It allows us to treat our turtle as a `TurtleGTX`. Once we have our reference to a fancy turtle, we can call its `Spin()` method.

You will get an exception if you attempt to type cast a plain turtle to a `TurtleGTX`. It just doesn't have the capabilities! That is why we tested first on line 7, to make sure it could work. (You could wrap the type-cast in a try...catch block, but it is generally considered bad style to write code that you expect to routinely throw exceptions.)

The mechanism we've shown above works in C#, Java, and a few other languages. But C# also has an alternative mechanism that is a widely used idiom:

```
1  private void drawSquare(Turtle t, double sz)
2  {
3      for (int side = 0; side < 4; side++)
4      {
5          t.Forward(sz);
6          t.Right(90);
7
8          TurtleGTX tgtx = t as TurtleGTX;
9          if (tgtx != null) {
10             tgtx.Spin();
11         }
12     }
13 }
```

The `as` keyword does a "safe" type cast, and returns `null` if the cast was not possible.

Type testing and type casting also work if the type being tested and cast is an interface type.

30.5. Back to the high-level view

Let's revisit the last example in the previous chapter now. The different specialized implementations of dictionaries (`Dictionary`, `SortedDictionary`) all implement the interface `IDictionary`. So in the last example in the previous chapter, we set the return type of the method to `IDictionary`, and we changed the calling code to work in terms of that *role* rather than work with the *type* of the object.

This is a powerful technique. A role is more abstract than a type. By moving our thinking up to "what roles is each object capable of" rather than "What type of thing is this", we create opportunities for phones that can also take photographs, or act as music players.

And putting a separate and explicit "contract" between our components — one side promises to fulfil the contract, the other side only uses the object in terms of what has been promised — leads to improved reliability in our systems.

It is quite easy now for us to go back to our example at the end of the last chapter and choose some different dictionary implementation in the `letterFreqs` method. As long as our new choice implements `IDictionary`, nothing else will need changing.

Interfaces promote a kind of plug-and-play approach to software components, just as our USB interface allows a kind of plug-and-play way of hooking devices up to each other. As long as the interface is supported, we no longer have to care about the type of the object.

30.6. Glossary

implement an interface

A class that agrees to an interface contract is agreeing to provide certain functionality. It does so by providing properties and methods (and even other kinds of members) that fulfil its obligations.

interface (the informal usage)

The way something interacts with other components outside of itself. A gear lever in a motor car is an interface to the gearbox. In programming we use the word loosely to mean "the public members of a class".

interface (a formal programming interface)

A specification of what methods and members an implementing class promises to the consumer of the functionality. Classes can implement many different interfaces.

role

Something that an object can *do*, as opposed to what it fundamentally *is*.

type cast

A conversion from one type to another.

type test

A test that lets us determine if an object is convertible to a specific type.

30.7. Exercises

1. We want an interface called `IMusicPlayer`. What methods and members should an object have so that others can interact with it in its role as a music player?
2. Some types are `ICloneable`. Look up the interface, discover what members it promises. Find at least two types that are cloneable, and two that are not. Are arrays cloneable? Are lists cloneable? What about `DateTime` and dictionaries?
3. Serialization of an object means *"to convert it into some textual representation"*. For example, our XAML text is a serialized representation of the GUI window we design. Many types of objects can be serialized, so, as we might expect, there is an interface called `ISerializable`. Look up help for `ISerializable` and determine what members a class must provide in order to fulfil this role.