



**Computer Science
from the
Metal Up**

**Object-oriented
Programming**

Richard Pawson

**Foreword by
Alan Kay**

Object-Oriented Programming (C# version)

by Richard Pawson

v1.0.0

©Richard Pawson, 2020. The moral right of the author has been asserted.



This document is distributed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License: <https://creativecommons.org/licenses/by-nc-nd/4.0/>.

The author is willing, in principle, to grant permission for distribution of derivative versions, on a case by case basis, and may be contacted as rpawson@metalup.org.

'Metal Up' is a registered trademark, number UK00003361893.

Author's acknowledgements

I am indebted to John Stout who has acted as a continuous reviewer throughout the writing of this book, offering many corrections and improvements to both text and program code. Responsibility for any remaining errors remains mine alone – please report any that you find to me as rpawson@metalup.org.

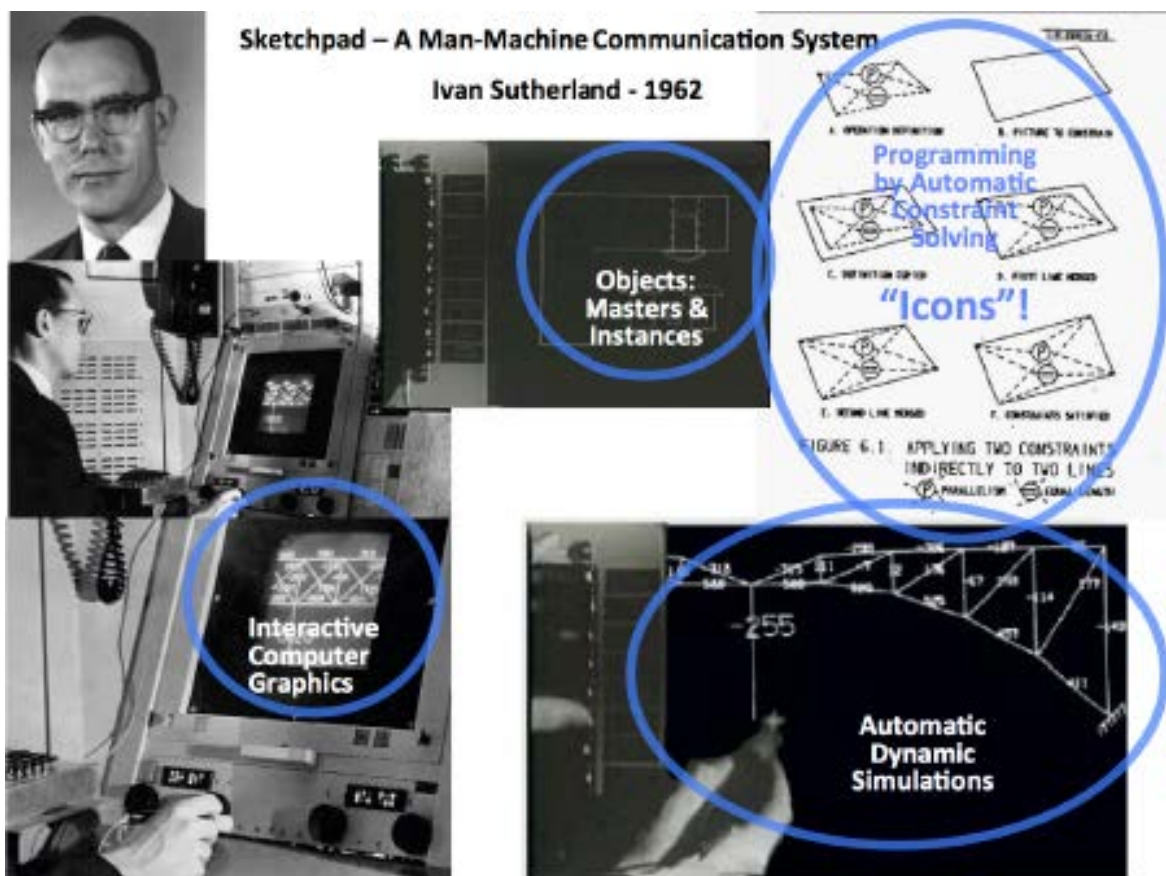
Foreword

by Alan Kay

In my first day in grad school in 1966, the head of the department gave me a thesis from just a few years earlier: “Sketchpad: A Man-Machine Communications System” by Ivan Sutherland at MIT. It completely turned upside down my ideas about computing.

It was not just the invention of interactive computer graphics, but also allowed the user to define complex “Ideas” — such as beams for a bridge, rivets, transistors, linkages, text characters, and much more — produce as many “instances” of each Idea as needed to make more complex assemblies (which could themselves be turned into Ideas) — then use goal-solving graphical programming to give the Ideas dynamic relationships and larger simulation behaviors.

For example, you could define a beam as being like a stiff spring, use many instances of them to make a bridge, hang simulated weights from the bridge, and Sketchpad would dynamically compute the tensions and compressions on the beams and show these numerically — all without the system knowing anything about beams or bridges, or even the numeral forms for numbers ahead of time.



A few days later I was asked to get what was thought to be Algol going on the university computer, but it turned out to be the first version of Simula, which was a new language that could also define “Ideas consisting of behaviors and properties”, and create parallel existing instances of them. Years later, many of today’s “OOP” languages — C++, Java, C#, etc. — emerged from Simula’s approach.

Once grokked, I could see there were already examples of Ideas and Instances: in the data-base world (but missing behaviors), in the multi-computing and time-sharing world of physical computers running quasi-parallel “processes” that were virtual encapsulated computers. There was the “Idea” of “physical computer” itself, with many physical instances about, and my ARPA research community was just starting the work to hook them together with what is today called the Internet -- many of these would act as “servers” for “services” requested and delivered over the network.

One of my undergraduate concentrations had been molecular biology, which revealed a vast scale of mechanisms. Even a single cell is vastly more complex than any computer, and a human body has more than 10 trillion of them – all instances of about 250 kinds of cell. This was serious scaling! But one which worked because of the modular highly protected nature of active cells whose interiors didn’t require a “central control”, nor did the enormous organizations that could be made from them. These were kind of like computers on a network ...

This got me thinking about the “idea of Ideas” as universal mechanisms. Since a computer can simulate any mechanism including any computer, it can make as many Ideas and instances of them as needed, and they will be universal. This meant that — if you don’t worry about efficiency, speed, space, etc. — that anything can be defined as a system of virtual computers, themselves composed of virtual computers, at any scale from individual gates and bits to systems larger than the Internet was going to be. If the communication between them is non-imperative — for example, they can make requests of each other but cannot command (very like biological cells) -- very large and safe organizations should be possible (like the Internet itself, which co-evolved with these ideas and was also done by my research community). My image of this was essentially a software version of what the Internet was going to do with hardware.

While pondering all this, someone asked me what I was working on and I foolishly replied “Object Oriented Programming” (a very poor choice of term for something much more inherently dynamic). But composite structures in those days were often called “objects”, and these new “things” were composites. Better choices might have been “Process Oriented Programming”, or “Simulation Oriented Programming”, or “Activity Oriented Programming” or “Agency Oriented Programming” or “Server Oriented Programming”, or even “Idea Oriented Programming”.

Too late now!

In any case, the most important thing to understand about all this is the idea of organizing dynamic complexities into sealed off systems that can safely intercommunicate, and whose components are themselves such systems. This provides a way to define and build anything from a single powerful principle, but it only suggests a little about how to go about designing and programming. For example, if everything is made from these dynamic building blocks, it will be possible for systems to “reflect” on their own makeup, properties and behaviors. What could this mean?

Why do I put it this way? Because in a field with the immense rapid scaling that ours is still going through, our tools will always be extremely out of date. For example, it takes years to even make a useable tool — especially a practical programming language — and old ideas die hard. Meanwhile the Moore’s Law explosion means that 40 years ago (when C++ was designed), personal computers were about 2,000 to 10,000 times slower (CPU), and about 10,000,000 times smaller (RAM). But little has changed in programming language design since then.

This means that even if you are learning a current favorite language, most of the concepts in it will not scale well today and you will need new concepts not in the today’s languages. So actual “learning to program” is not just “learning to code”, and not just “learning to design”. It is most

especially about learning how to deal with scalings and complexities by anticipating difficulties — including those in the tools — and building the tools needed in the inadequate languages of today, before trying to use their features directly.

This isn't fair to beginners, but far too many computerists today never learned how to deal with what's actually going on. If you start thinking about these issues now, while you are learning, then you will be much more resilient when dealing with what is actually needed.

The good news here is that Richard Pawson — the author of this book — is one of the most experienced practitioners of not just OOP and other forms of programming, but has done a number of real-world systems that required just what was suggested above about needing to go beyond the tools given to extending the tools in important ways. When we realize that the fundamental idea behind “objects” is that they can simulate anything we need, we can see that deeper use of objects can keep pace with scalings and other problems by extending our tools as required.

The second large example in this excellent book shows just how to do this with a realistic problem of “object bases” that is drawn from an actual massive and successful system that Richard designed and helped build a few years ago. This puts design, reflection, and extending tools front and center.

So this is not just a textbook, but a journey into the real-world of large scale problems and how they can be addressed, guided by an expert who has literally been there and done that.

While learning this example, see if you can imagine all computer systems as having many of the very same problems. We have to attend to them with object-bases, but all of our systems would be much more controllable if we could see them as needing what object-bases require.

Happy travels!



Alan Kay



Alan Kay likes to say, “No one owes more to his research community than I do”. He is one of the earliest pioneers of object-oriented programming, personal computing, graphical user interfaces, and children’s computing, all done within the context of the ARPA/Parc research communities. His recognition includes the ACM Alan Turing Award, the National Academy of Engineering Charles Stark Draper Prize, and the Kyoto Prize. For more info: <http://vpri.org/bio/> (Chrome is suggested).

Contents

<i>Introduction</i>	1
Part I – An object-oriented interactive drawing program	2
<i>Using ready-made object types</i>	3
<i>Creating new object classes</i>	12
<i>Polymorphism</i>	18
<i>Abstract classes</i>	24
<i>Implementation inheritance</i>	29
<i>Inheritance vs. delegation</i>	36
<i>Enriching the application</i>	40
<i>Association</i>	48
<i>Deleting and Duplicating objects</i>	55
<i>Suggested further enhancements and extensions</i>	59
Part II – An object-oriented records-management system	60
<i>OOP and records management</i>	61
<i>Objects in memory</i>	62
<i>Saving objects to a file</i>	73
<i>Persisting objects to a database</i>	76
<i>Introducing the Naked Objects framework</i>	91
<i>Enriching domain objects</i>	101
<i>Extending the model</i>	106
Appendices	117
<i>Technical pre-requisites</i>	118
<i>Troubleshooting</i>	119

Introduction

This book provides an introduction to object-oriented programming (OOP) in the C#VB programming language.

The book takes the form of two extended, guided exercises, to develop two applications, with very different forms both of user experience and underlying technical infrastructure:

- OOPDraw: an object-oriented interactive drawing program (Part I)
- OOPRecords: an object-oriented records management program (Part II)

The first of these extended exercises will teach you the core principles that define OOP today.

However, the second will give you more of an understanding of what is required to build more substantial applications, in particular the requirement to 'persist' (save) object states in some kind of 'object-base'. As such it will provide invaluable insight and experience for those students who might want to undertake their own projects involving objects and persistence.

Part I – An object-oriented interactive drawing program

In Part I we are going to learn all the core principles of OOP through the development of a single application: a simple interactive drawing program, called OOPDraw, that works in much the same way as off-the-shelf drawing applications that you may have used before, such as PowerPoint.

We have chosen this application because it aligns very well with object-oriented principles, and this is no coincidence. Ivan Sutherland's ground-breaking Sketchpad in 1963 was one of the things that inspired Alan Kay and his team at Xerox Parc to create the first *pure* object-oriented programming language, Smalltalk, and, through that language, the first implementation of what is today the most widespread form of graphical user interface.

Using ready-made object types

In Visual Studio 2019 create a new project using the **Windows Forms App** template:

Create a new project

Windows Forms

Clear all

C# Windows Other

No exact matches found

Other results based on your search

Recent project templates

- Windows Forms App (.NET Core) C#
- Console App (.NET Core) C#
- Class Library (.NET Core) C#
- MSTest Test Project (.NET Core) C#
- Class Library (.NET Framework) C#
- ASP.NET Core Web Application C#

Windows Forms App (.NET Core)
A project for creating an application with a Windows Forms (WinForms) user interface
C# Desktop Windows

Windows Forms Control Library (.NET Framework)
A project for creating controls to use in Windows Forms (WinForms) applications
C# Desktop Library Windows

Back Next

Name the project **OOPDraw**, and choose a suitable location to store it:

Configure your new project

Windows Forms App (.NET Core) Desktop Windows

Project name

OOPDraw

Location

C:\MetalUp\

Solution name ⓘ

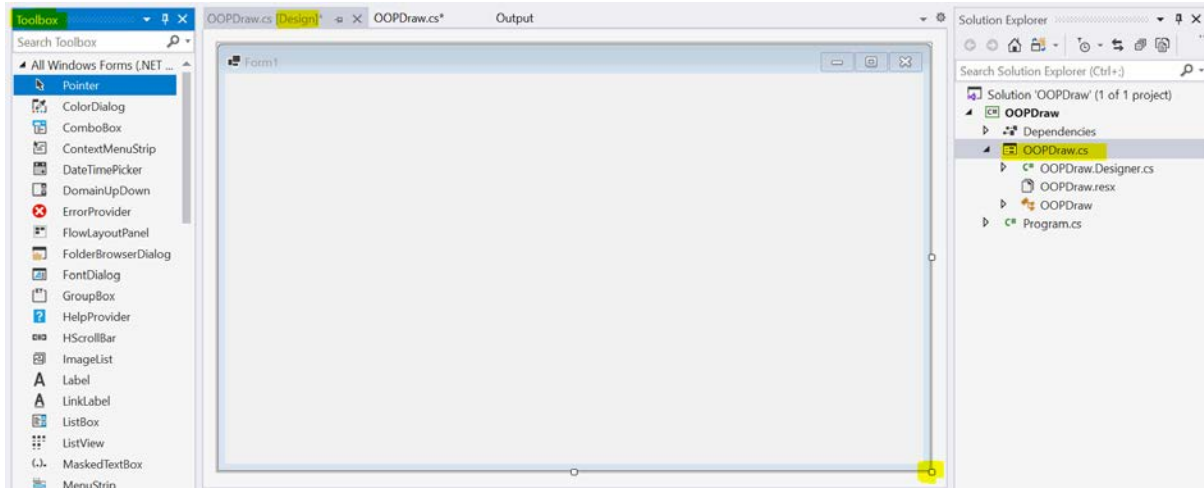
OOPDraw

Place solution and project in the same directory

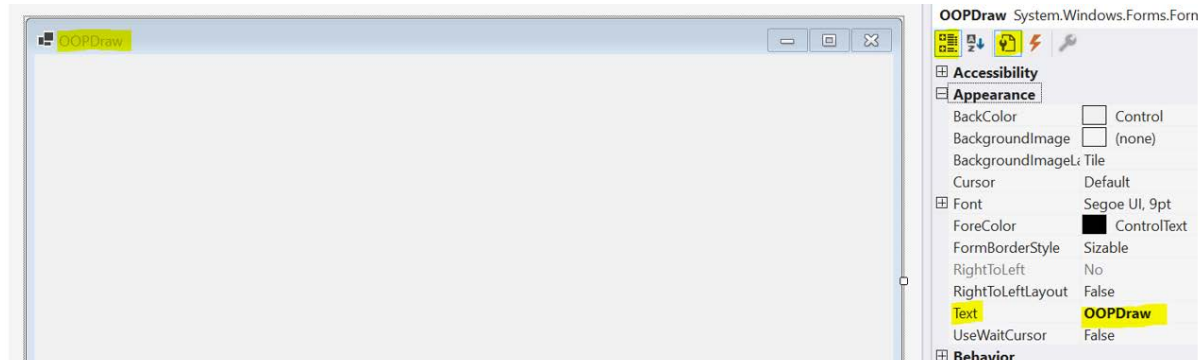
Back Create

The project will be created with a default Windows form named **Form1**. Right-click on this file in the **Solution Explorer** and select **Rename**, then edit the name to **OOPDraw** (clicking **Yes** if you are presented with a confirmation dialog). Then double click on this file to open the **[Design]** view of the form.

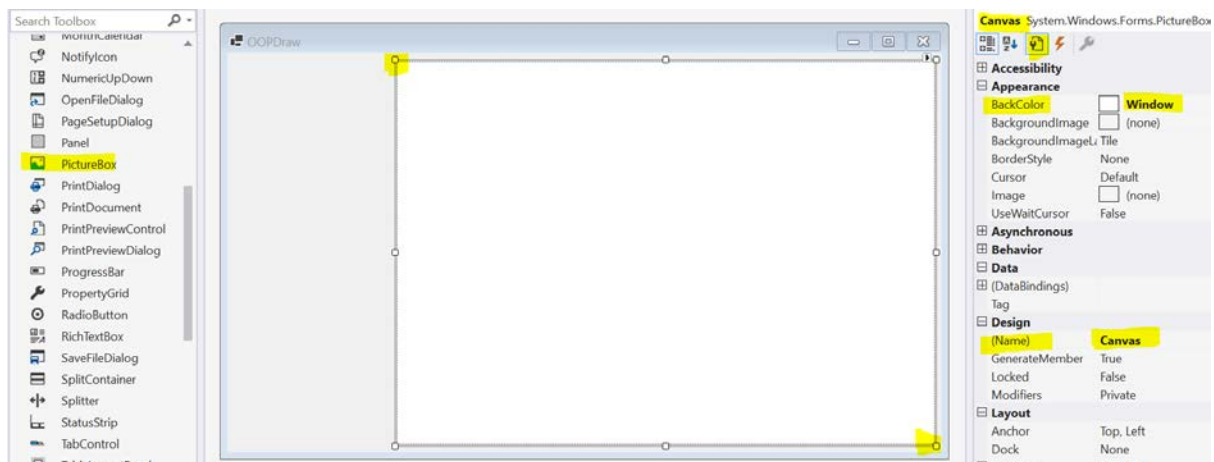
Open the toolbox the (**View > Toolbox**), then select and drag the lower-right corner of the form's visual representation so that it nearly fills the visible space (you can adjust this again later, if needed):



Right-click within the form to select **Properties**, and edit the **Text** property to also say **OOPDraw** (this changes the label at the top of the form in the **[Design]** view) . (If your view of the properties does not show them grouped by 'categories' as shown below, click the top-left icon – highlighted below – to show the categories.)

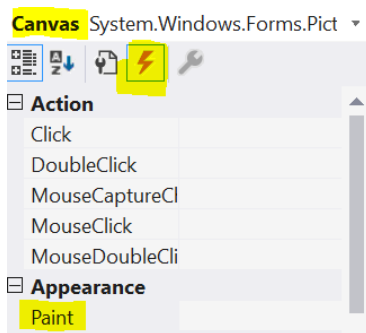


From the **Toolbox**, drag the **PictureBox** component into the form, then expand it to fill most of the form, leaving some space on the left where we will add some further components later (you can always re-adjust the size of the box later). Then within the **Properties** view for this new component (if not visible, right-click within the picture box and select **Properties**), edit the **(Name)** property to **Canvas**, and the **BackColor** property to **Window** (so that the background changes to white).



(From now on we will refer to this white picture box as 'the canvas').

Still within the **Properties** of the canvas click on the lightning symbol to show the **Events** for this component, then double-click on the **Paint** event:



which will create an empty `Canvas_Paint` method in the `OOPDraw.cs` file. This file is where we can add C# code that defines behaviour for the form (this file is also known as the 'code-behind' for the form and when we use this term we mean `OOPDraw.cs`). Make the changes highlighted below:

```
using System.Drawing;
using System.Windows.Forms;

namespace OOPDraw
{
    public partial class OOPDraw : Form
    {
        public OOPDraw()
        {
            InitializeComponent();
            DoubleBuffered = true; //Stops image flickering
        }

        Pen currentPen = new Pen(Color.Black);

        private void Canvas_Paint(object sender, PaintEventArgs e)
        {
            Graphics gr = e.Graphics;
            Point a = new Point(20, 30);
            Point b = new Point(400, 500);
            gr.DrawLine(currentPen, a, b);
        }
    }
}
```

Exercise 1

Run the application, which should show the canvas with a line drawn on it.

Capture a screen snippet showing this.

Then stop the program, either by closing the form, or by clicking on the red square – 'Stop Debugging' – icon at the top of Visual Studio

Looking again at the code you inserted (above), there are several things to note:

- There are lots of new *types* involved. Previous code that you will have written might have involved just simple types such as `int`, `bool`, or `string` and perhaps some common data structure types such as an array or a `List` of one of those simple types. Now, in these few lines of code, you are encountering multiple new types: `Graphics`, `Pen`, `Color`, `Point`, `PaintEventArgs`... . You might not understand *exactly* what those types do, but you can already see evidence of one of the first principles of OOP: you create applications using types that are *meaningful* to the context (also known as the 'domain') of the application: in this case all the type names relate in some way to graphics.
- OOP involves creating and manipulating instances of these 'domain types'. These instances may be assigned to variables, passed as parameters, and held in data structures: all the

same things you are used to doing with basic types. In the code above, we use one instance of the type `Graphics` held in the variable `gr`, one instance of type `Pen` held in the variable `currentPen`, and two instances of type `Point` held in variables `a` and `b`. Note that the latter two instances are created using the keyword `new`, followed by the type name with brackets. This code is calling the 'constructor' for a `Point` and the constructor specifies what data is needed to construct an instance of that type: in this case two integers representing the X and Y coordinates of the point being created.

- As well as creating and assigning instances, OOP involves accessing and using the 'members' of an instance. Broadly speaking, there are two kinds of member: *properties* and *methods*. The exact distinction between a property and a method in C# is subtle (see [Properties vs. Methods](#)), but as a first approximation, properties hold data – what the instance *knows* – and methods provide *behaviour* – what the instance *knows how to do*. Both kinds of member are accessed via 'dot syntax' – the instance variable is followed by a dot and then the name of the member required. So, `e.Graphics` accesses the `Graphics` *property* of `e` (an instance of type `PaintEventArgs`) and `gr.DrawLine(...)` accesses the `DrawLine` *method* on `gr` (an instance of type `Graphics`). Methods may also be thought of as functions that are associated directly with an object.
- The one absolute distinction is that accessing a method always requires brackets, typically containing one or more arguments, though in some cases the brackets are empty; properties are always accessed without brackets.

Exercise 2

1) In the code, hover the mouse over the name `DrawLine` and you will see a pop-up tooltip, telling you about that method. Describe, in general terms, at least three kinds of information that the tooltip is giving you.

On a new line, just underneath the line that calls `DrawLine`, type:

```
gr.
```

This will give you a pop-up list of all the members of an instance of type `Graphics` that you may access. The properties have a spanner icon next to them, the methods have a small cube icon. The members marked with a star are just the members that Visual Studio thinks you are most likely to use in the current context.

Select `DrawLine` and then type `(` and you will get a new pop-up indicating that you have four options. This means that there are four different versions of the `DrawLine` method – we say that `DrawLine` is ‘overloaded’.

2) What is the principal difference between the four versions of this method (give a general description of the principal difference, not a detailed explanation)? When done, delete the extra bracket that you added.

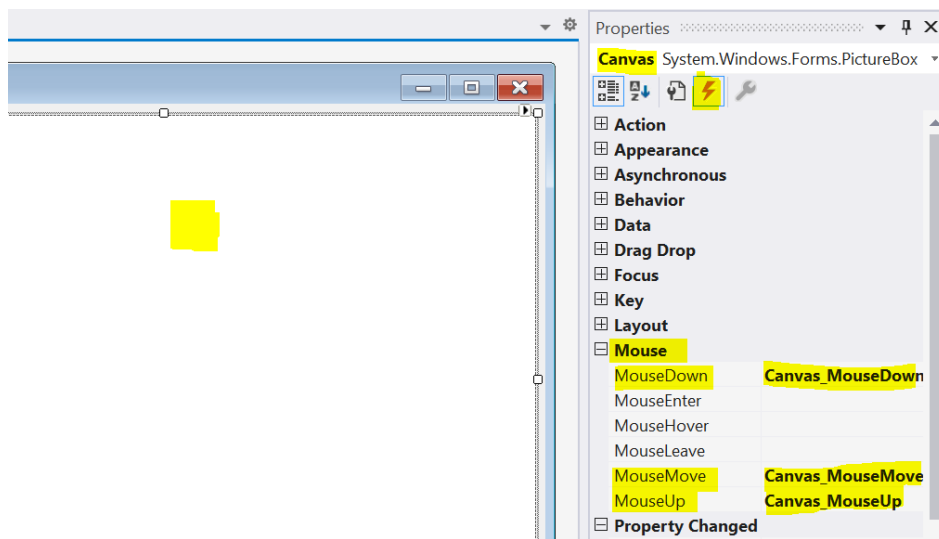
Into the `Canvas_Paint` method add a new line creating a third `Point`. Then add two more calls to `gr.DrawLine` (in both cases using the first version of that method) such that when the program is run you have drawn a large triangle on the form.

3) Capture screen snippets showing the code you have added, and the resulting triangle drawn.

Making OOPDraw interactive

We have drawn a triangle *programmatically* – meaning that it was executed by pre-defined code. However, we want to make OOPDraw *interactive*, such that a *user* may create their own drawing in the running application.

In the *Events* properties for *the canvas* locate the **Mouse** heading, and double-click on each of the three events: **MouseDown**, **MouseMove**, and **MouseUp**. Note that double-clicking on each event will add code into the code-behind, and you will need to re-select the **[Design]** view before selecting the next event. The properties should end up looking like this:



The code-behind should now contain three new, empty, methods: `Canvas_MouseDown`, `Canvas_MouseMove`, and `Canvas_MouseUp`, where you can add code that will be invoked when each of those events occurs.

Modify the code-behind, inserting the code highlighted below:


```
public partial class OOPDraw : Form
{
    public OOPDraw()
    {
        InitializeComponent();
        DoubleBuffered = true; //Stops image flickering
    }

    Pen currentPen = new Pen(Color.Black);
    bool dragging = false;
    Point startOfDrag = Point.Empty;
    Point lastMousePosition = Point.Empty;

    private void Canvas_Paint(object sender, PaintEventArgs e)
    {
        Graphics gr = e.Graphics;
        previous code deleted
        gr.DrawLine(currentPen, startOfDrag, lastMousePosition);
    }

    private void Canvas_MouseDown(object sender, MouseEventArgs e)
    {
        dragging = true;
        startOfDrag = lastMousePosition = e.Location;
    }

    private void Canvas_MouseMove(object sender, MouseEventArgs e)
    {
        if (dragging)
        {
            lastMousePosition = e.Location;
            Refresh();
        }
    }

    private void Canvas_MouseUp(object sender, MouseEventArgs e)
    {
        dragging = false;
    }
}
```

Exercise 3

- 1) Having made the changes above, run the program. Click and drag the mouse. Describe what happens.
- 2) Release the mouse button and move the mouse. Describe what happens.
- 3) Now draw a new line. Describe what happens.

Now go back to the code listing above and note the following:

- In the top highlighted line we are defining a boolean variable, `dragging`, that indicates whether or not the user is *currently* dragging the mouse (i.e. moving it with the button down). This is read and/or set to `true` or `false` within the three new event methods.
- The two variables `startOfDrag` and `lastMousePosition`, initialised to an empty value using `Point.Empty` (this is not essential, but good practice), hold the two ends of the current line as `Points`. Make sure you understand where these variables are being set and used, and why.
- The call to the `Refresh` method instructs the system to clear the canvas, which will then automatically result in the `Canvas_Paint` method being called, which redraws the line. This is necessary to get the ‘elastic band’ effect when dragging.

Drawing multiple lines

We have made a useful start, but we really want the ability to be able to draw multiple lines. To achieve this we will need to memorise previous lines and draw them all again each time within `Canvas_Paint`. (This might sound wasteful but it is how almost all graphics are drawn, now that modern computers have so much processing power.)

How should we store a representation of a line? We could hold the specification of each line as a `Point array` (holding two `Points`) or as an integer array holding 4 integer values – but, thinking ahead, what if we also want the lines to have different widths, or colours?

What we really want is a type called `Line` to hold all this information as properties. And if there isn’t a `Line` type already – or if there is one, but it isn’t compatible with what we need – then we can just create our own...

Creating new object classes

The code holding the definition of a new type in OOP is called a 'class'. You can think of a class as being like a template. Its principal role is to define the members (properties and methods, remember) that *each* instance of that type will have.

Right-click on the **OOPDraw** project icon (not the **OOPDraw** 'solution' icon above it) and select **Add > Class**. In the dialog specify the class name as **Line**. Then add the code highlighted below:

```
using System.Drawing;

namespace OOPDraw
{
    public class Line
    {
        public Pen Pen { get; private set; }
        public int X1 { get; private set; }
        public int Y1 { get; private set; }
        public int X2 { get; private set; }
        public int Y2 { get; private set; }

        public Line(Pen p, int x1, int y1, int x2, int y2)
        {
            Pen = p;
            X1 = x1;
            Y1 = y1;
            X2 = x2;
            Y2 = y2;
        }

        public Line(Pen p, int x1, int y1) : this(p, x1, y1, x1, y1)
        {
        }

        public void Draw(Graphics g)
        {
            g.DrawLine(Pen, X1, Y1, X2, Y2);
        }

        public void GrowTo(int x2, int y2)
        {
            X2 = x2;
            Y2 = y2;
        }
    }
}
```

We'll explore the new code shortly, but in the meantime, we'll make the necessary changes in our existing code to use the new **Line** type. Make the following changes to the code-behind, carefully noting that in several places the new highlighted code *replaces* one or more previous lines of code:

```
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Windows.Forms;

namespace OOPDraw
{
    public partial class OOPDraw : Form
    {
        public OOPDraw()
        {
            InitializeComponent();
            DoubleBuffered = true; //Stops image flickering
        }

        Pen currentPen = new Pen(Color.Black);
        bool dragging = false;
        Point startOfDrag = Point.Empty;
        Point lastMousePosition = Point.Empty;
        List<Line> lines = new List<Line>();

        private void Canvas_Paint(object sender, PaintEventArgs e)
        {
            Graphics gr = e.Graphics;
            foreach (Line line in lines)
            {
                line.Draw(gr);
            }
        }

        private void Canvas_MouseDown(object sender, MouseEventArgs e)
        {
            dragging = true;
            startOfDrag = lastMousePosition = e.Location;
            lines.Add(new Line(currentPen, e.X, e.Y));
        }

        private void Canvas_MouseMove(object sender, MouseEventArgs e)
        {
            if (dragging)
            {
                Line currentLine = lines.Last();
                currentLine.GrowTo(e.X, e.Y);
                lastMousePosition = e.Location;
                Refresh();
            }
        }

        private void Canvas_MouseUp(object sender, MouseEventArgs e)
        {
            dragging = false;
        }
    }
}
```

Exercise 4

Having made the changes listed above, use the program to draw multiple lines, and capture a screen snippet of your drawing.

We'll now go back and explore the new code, starting with the `Line` class definition:

- The `class` keyword indicates that what follows is a definition for a new type.
- The new class, named `Line`, defines four integer properties to hold the two pairs of coordinates that define the end points of the line. `Line` also defines one property, named `Pen`, and of type `Pen`.
- All the properties in this class take the form:
`public [type] [name] { get; private set; }`. This means that code *outside* the class definition may *read* the property value, but only code *inside* the class definition may *modify* that value.
- C# is case-sensitive. `X1` is not the same as `x1`, which is why you can have meaningful statements like `X1 = x1` (setting the property `X1` to the parameter value `x1`). However, this does mean that it is easy to make the mistake of writing `X1 = X1`, or `x1 = x1` neither of which will achieve the right result (or, indeed, have any effect at all).
- After the properties we find two definitions beginning `public Line`. These are *constructors* (which were briefly mentioned in the previous chapter). It is common to define a constructor for a class, but you *may* define more than one. Each time a new instance of a class is created, a constructor will be called; *which* constructor is called depends upon the arguments provided – because the constructors must differ in the number and/or type of the parameters they specify.
- The syntax rules that define any constructor are: it must have the same name as the class (`Line` in this case); it has no return type, not even `void`; and it must be `public` because it is necessary to be able to create a new instance from code outside the class.
- The first of the two constructors specifies that the calling code must provide a `Pen`, and four integer values representing the two pairs of coordinates. In the body of this constructor all the parameter values are copied into the properties with corresponding names.
- The second constructor defines only three parameters: a `Pen` and two integer values representing just one pair of coordinates. The code specifies that the second constructor delegates its work to the first constructor, and it fulfils the latter's requirement for four integer arguments (after the pen) by passing in its values for `x1` and `y1` in *twice*. This second constructor is a convenience: it allows us to create a `Line` of zero length, which is useful when starting to draw a line by dragging the mouse.
- At the bottom of the class definition we see two *methods* defined: `Draw` and `GrowTo`, either of which may be invoked on an instance of type `Line` using dot-syntax. We'll explore the significance of these methods below in the next section.

Turning now to the new code added into the OOPDraw form code-behind...

- We have defined a variable named `lines`, to hold a `List` of objects each of type `Line`.
- When the user starts drawing a line, a new `Line` instance is created (with zero length) and it is added to the `lines` list.
- As the mouse is dragged the method `Canvas_MouseMove` first retrieves the last line to be added to the list using `lines.Last()` then calls the `GrowTo` method on it, passing in the current mouse coordinates.
- The `Canvas_Paint` method now draws each of the lines in the list (including the current 'elasticated' line) using a `foreach` loop. It calls the `Draw` method of each `Line` in the `lines` collection, passing in the (common) `Graphics` object that is associated with the canvas.

Encapsulation and information hiding

The new code has allowed us to draw multiple lines by creating multiple instances of the type `Line`, each of which holds the details (a `Pen` and coordinates for the end points) of a line.

In addition to this data, the `Line` class also *encapsulates* the behaviour of a line: it *knows how to draw itself* (in collaboration with a `Graphics` object) and it *knows how to grow itself*. These two behaviours are implemented as methods with suitably 'intentional' names. What is the advantage of moving these behaviours, which existed previously but elsewhere in our code, into the `Line` class?

One reason is that it allows us to *hide* the data. Having defined the coordinates of the line when we created it (via one of the two constructors) the code *deliberately* does not allow any of those properties to be changed by any code outside the object itself. There is no *need* to change those properties directly: if you want to change the size of the `Line` you call `GrowTo`, which *can* modify the appropriate properties.

This is a fundamental principle of OOP known as 'information hiding'. By only modifying properties *via* the methods, it is easier to enforce rules that prevent the object from being put into an invalid state, with inconsistent properties for example.

Sometimes we can take this further by ensuring that properties are visible only inside the class, too.

However, the biggest advantage of encapsulation will become clear when we start to deal with multiple types of object – in the next chapter.

In the meantime, we will extend the functionality of our application.

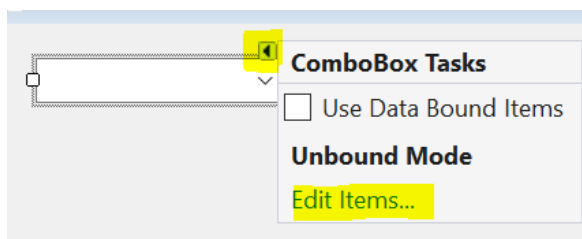
Changing the colour and line width

After stopping the program select the **[Design]** view, then follow these steps:

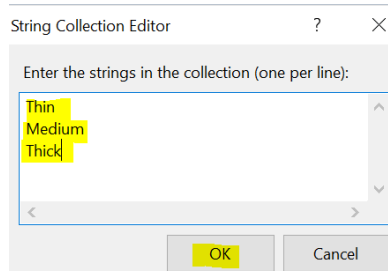
- 1) From the toolbox drag the **ComboBox** component into the form. (A combo box appears as a drop-down list in the running application.) In the **Properties** pane, set **(Name)** to **LineWidth** (no space):



2) click on the small triangle highlighted and select **Edit Items**:

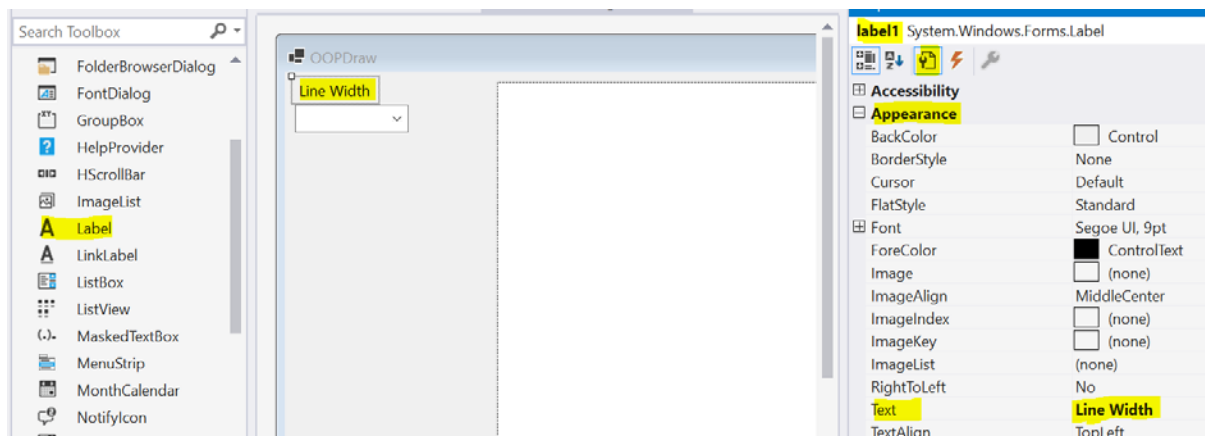


3) in the dialog enter the strings **Thin**, **Medium**, and **Thick** on separate lines and click **OK**:



4) double click-within the combo box, which will generate a method `LineWidth_SelectedIndexChanged` in the code-behind.

5) Back in the **[Design]** view, drag in a **Label** component to just above the combo box (move the box if necessary), and set the **Label's Text** property to **Line Width** (with a space):



Now repeat steps 1 to 5, adding a second combo box *renamed* to **Colour**, with text values for Red, Green, and Blue, plus a separate **Label**.

Add the code shown below into the two new methods in the code-behind:

```
private void LineWidth_SelectedIndexChanged(object sender, System.EventArgs e)
{
    float width = currentPen.Width;
    switch (LineWidth.Text)
    {
        case "Thin":
            width = 2.0F;
            break;
        case "Medium":
            width = 4.0F;
            break;
        case "Thick":
            width = 8.0F;
            break;
    }
    currentPen = new Pen(currentPen.Color, width);
}

private void Colour_SelectedIndexChanged(object sender, System.EventArgs e)
{
    Color color = currentPen.Color;
    switch (Colour.Text)
    {
        case "Red":
            color = Color.Red;
            break;
        case "Blue":
            color = Color.Blue;
            break;
        case "Green":
            color = Color.Green;
            break;
    }
    currentPen = new Pen(color, currentPen.Width);
}
```

Notes:

- Each of these methods uses a `switchSelect` statement on the value in the `Text` property of the corresponding combo box (i.e. on the value that the user has selected from the list).
- In both cases the outcome is to replace the `currentPen` with a new `Pen` object that has either a new value for the `color` or the `width`, taking the other value from the existing `Pen`.

Exercise 5

Make the changes above, then use the program to make a small drawing that involves lines of different widths and colours. Capture a screen snippet.

Polymorphism

Polymorphism is the most powerful principle of OOP. It applies when you have more than one type of object that have some similar properties or behaviours: for example, having different types of shape that can be drawn or grown.

We will start by creating another class `Rectangle` with this class definition:

```
using System;
using System.Drawing;

namespace OOPDraw
{
    public class Rectangle
    {
        public Pen Pen { get; private set; }
        public int X1 { get; private set; }
        public int Y1 { get; private set; }
        public int X2 { get; private set; }
        public int Y2 { get; private set; }

        public Rectangle(Pen p, int x1, int y1, int x2, int y2)
        {
            Pen = p;
            X1 = x1;
            Y1 = y1;
            X2 = x2;
            Y2 = y2;
        }

        public Rectangle(Pen p, int x1, int y1) : this(p, x1, y1, x1, y1)
        {
        }

        public void Draw(Graphics g)
        {
            int x = Math.Min(X1, X2);
            int y = Math.Min(Y1, Y2);
            int w = Math.Max(X1, X2) - x;
            int h = Math.Max(Y1, Y2) - y;
            g.DrawRectangle(Pen, x, y, w, h);
        }

        public void GrowTo(int x2, int y2)
        {
            X2 = x2;
            Y2 = y2;
        }
    }
}
```

Notes:

- The `Draw` method calls `DrawRectangle` on the `Graphics` object `g` (just as the `Draw` method in `Line` called `DrawLine`)
- Why the need for the first four lines of code in the `Draw` method? The reason is that `DrawRectangle` requires the width, the height, and the coordinates of the *top left* corner to be specified, but we don't want to force the user to have to draw the rectangle *always* starting from the top-left hand corner: the user should be able to click the mouse and move it in any direction and so this code works out which pair of coordinates represents the top-left corner.

Before we start using this code, complete this short exercise...

Exercise 6

Looking at the code for `Rectangle` alongside that of `Line`, compare *just the methods*.

Make a copy of the methods of `Rectangle`, and colour the lines *red* if they are different.

The key point from the exercise is that for *both* `Draw` and `GrowTo`, the method *signature* is identical between `Line` and `Rectangle`. The method *implementation* (body) of the `Draw` method is different between the two classes, though for `GrowTo` the implementations are identical.

Add the `Rectangle` class definition into a new file.

To test it, without adding all the code to draw it interactively, *temporarily* add this line at the end of `Canvas_Paint` in to draw a single `Rectangle` programmatically:

```
Rectangle rect = new Rectangle(currentPen, 100, 200, 300, 500);  
rect.Draw(gr);
```

Exercise 7

Run the program and capture a screen snippet of the result.

Then delete the two lines added above.

Now to make it interactive the first challenge we need to address is how we are going to store any rectangles that the user has drawn, as well as any lines. We *could* create a separate `List<Rectangle>` and then in `Canvas_Paint` work through the separate lists of lines and rectangles, but this code is going to become very repetitive when we go on to add other shapes in future. It would be much nicer if we could hold them in one list. One way to do this would be to re-define the list as a `List<object>` which can hold any type of object –both `Line` and `Rectangle` being types of object.

In the line:

```
private List<Line> lines = new List<Line>();
```

Important: when you rename a variable, property, or method – which you should do whenever the old name is no longer an accurate description – always do this by right-clicking on the name and selecting the option **Rename**, because it doesn't just edit the name where the cursor is: it finds every reference to that name in the code and changes it to match.

Change `List<line>` to `List<object>`, *manually* in two places (this is *not* a rename). Then make these changes, noting that one of the changes is a rename:

```
private void Canvas_MouseMove(object sender, MouseEventArgs e)
{
    if (dragging)
    {
        dynamic shape = shapes.Last();
        shape.GrowTo(e.X, e.Y);
        lastMousePosition = e.Location;
        Refresh();
    }
}
```

Change the `Canvas_Paint` method to read as follows:

```
private void Canvas_Paint(PaintEventArgs e)
{
    Graphics gr = e.Graphics;
    foreach (dynamic shape in shapes)
    {
        shape.Draw(gr);
    }
}
```

Then within the constructor, *temporarily* add this code to initialise the shapes list with a single, fixed, rectangle:

```
public OOPDraw()
{
    InitializeComponent();
    DoubleBuffered = true; //Stops image flickering
    shapes.Add(new Rectangle(currentPen, 100, 100, 300, 200));
}
```

Exercise 8

Run the code, adding some further lines using the mouse, and capture a screen snippet of the result.

We will explore the meaning and significance of the keyword `dynamic` shortly. For now, though, try changing either of the two uses of the keyword `dynamic` to `object` – which would seem more obvious, since we know every item in the `shapes` list is an object.

Exercise 9

What error message do you get from this change? Capture the error and then revert to using the keyword `dynamic`.

We have now seen that it is *possible* to hold both `Lines` and `Rectangles` in a single list and to call both the `Draw` and `GrowTo` methods on each object in the list.

Now we are using polymorphism. Here's a more formal definition of polymorphism, which is worth memorising:

Where two or more different types of object
define a member with the same signature, then
– even though the implementations may be different –
external code may access that member on an instance
without having to know the specific type of the instance.

You've just seen that in action: you are calling the method `Draw` on an instance named `shape`, without having to know whether `shape` holds a `Line` or a `Rectangle`.

Static typing and dynamic typing

Broadly speaking there are two *main* ways that polymorphism is supported in programming languages: using dynamic types, and using abstract static types. The example above uses dynamic types. Traditionally, programming languages are classified as either statically typed or dynamically typed.

In a *statically typed language*, the type of each variable or parameter, and the type of the value returned by each method/function (if it returns anything) must be specified in the source code. Though the value held by a variable may change, the type may not: you cannot define a variable `x` as an integer and subsequently assign a string to it. C#, Java, VB, are all examples of modern programming languages that are statically typed.

In a *dynamically typed language*, variables, parameters, and values returned by functions/methods do not have a pre-determined type. You can assign an integer to a variable named `x`, and subsequently assign a string to that same variable. Python and JavaScript are examples of modern programming languages that are dynamically typed.

Python, however, may *optionally* be made to behave *somewhat* like a statically-typed language by specifying the intended types – using 'type hints'. (Strictly speaking, Python ignores the type hints, but an IDE or other code development tool can use the hints to identify type inconsistencies in your code.)

Conversely, C# may *optionally* be made to behave more like a dynamically typed language by using the keyword `dynamic` where the language would otherwise require a type to be specified. This is what we did above.

There are advantages and disadvantages to both static and dynamic approaches. A dynamically typed language *typically* allows you to implement a given specification of requirements with slightly less code than if using a statically typed one. The dynamic typing also allows more flexibility in the rare cases where there is no way to know the type of the object you are dealing with in advance. (JavaScript was designed as a dynamically typed language because there is no way to know in advance about the structure – or ‘DOM’ – of a web page it is being asked to process, and this was the initial focus of JavaScript.)

Dynamic typing with OOP means that you can specify *any* method to call on an object. So, in the above example, you can call the member `Draw` on the variable `shape` even though the compiler cannot confirm that there definitely will be a `Draw` member (method, in this case) implemented for any given instance `shape`.

Whereas with a statically-typed variable or parameter, you can type a `.` after an instance and get a pop-up list of all the members available for that type, you won’t get any such list on a dynamically typed variable/parameter because *there is no way for the compiler to know what members the actual instance will have when the program is run*. Secondly, if you misspell a method name called on a dynamic type, or pass in arguments of the wrong type, the compiler cannot warn you in advance.

Within the `Canvas_Paint` method in delete this line:

```
shape.Draw(gr);
```

and instead type `shape .`

Confirm, for yourself that there is now no pop-up menu of available members.

Then complete the line as follows (with a deliberate misspelling):

```
s.Drew(gr);
```

Exercise 10

1) Run the program and capture a screen snippet showing what happens.

Then revert the line of code to its correct version.

By contrast, in the following line of code, `shapes` is a statically typed variable of type `List<object>`:

```
shapes.Add(new Rectangle(currentPen, 100, 100, 300, 200));
```

2) What happens if you mis-spell the call to the `Add` method (for example as `And`)?

(when finished, revert the change).

The principle that you’ve hopefully realised is that with dynamic typing you don’t find out that the member doesn’t exist, or that you’ve misspelled it, until run time. In the example above the error surfaced as soon as you ran it, but other such errors might surface only after a certain pattern of use.

In the next chapter we'll be finding out how to achieve polymorphism with static typing.

In the meantime, we need to get our application to allow the user to draw lines and Rectangles *interactively*.

Add a third combo box into the design view of the form, *renaming* the component to **Shape** and then specifying **Line** and **Rectangle** as the options. (There is no need to double click on this new combo box to get the `Shape_SelectedIndexChanged` method.) Then add a suitable **Label**.

To avoid creating unnecessary work for the user, you can also set the default selection for each combo box in the form's code-behind like this, *also deleting the previously-added line to draw a fixed rectangle*:

```
public OOPDraw()
{
    InitializeComponent();
    DoubleBuffered = true; //Stops image flickering
    LineWidth.SelectedItem = "Medium";
    Colour.SelectedItem = "Green";
    Shape.SelectedItem = "Line";
}
```

Make the following changes to the `MouseDown` method *replacing* the last line with the new highlighted code:

```
private void Canvas_MouseDown(object sender, MouseEventArgs e)
{
    dragging = true;
    startOfDrag = lastMousePosition = e.Location;
    switch (Shape.Text)
    {
        case "Line":
            shapes.Add(new Line(currentPen, e.X, e.Y));
            break;
        case "Rectangle":
            shapes.Add(new Rectangle(currentPen, e.X, e.Y));
            break;
    }
}
```

Exercise 11

When done, create a simple drawing using both lines and rectangles of different sizes and colours, and capture the drawing.

In the next chapter we are going to implement polymorphism using static typing, by defining an abstract class.

Abstract classes

An *abstract* class is one that does not have any instances, but defines members that are common to certain other classes. We are now going to define an ‘abstract’ class named `Shape`, which defines members that are common to our `Line` and `Rectangle` classes and will hopefully be common to other classes representing shapes that we might want to draw in future. `Shape` will then be the ‘superclass’ of `Line` and `Rectangle` – and they will each then be a ‘subclass’ of `Shape`.

Here is the first version of `Shape`:

```
using System.Drawing;

namespace OOPDraw
{
    public abstract class Shape
    {
        public abstract void Draw(Graphics g);

        public abstract void GrowTo(int x2, int y2);
    }
}
```

Notes:

- Our class defines two methods, `Draw` and `GrowTo`, each of which is marked `abstract`. This means that the method definition has only the *signature*; there is no *implementation*, so no braces follow the signature. The implementation must be provided by the subclass(es).
- The class itself is also marked `abstract` – in other words it will be of no use unless there is one or more subclasses that ‘inherit’ from it. Only an abstract class may define `abstract` methods.

Here is how we specify that `Line` inherits from (or ‘is a subclass of’) `Shape`:

```
public class Line : Shape
{
    ...

    public override void Draw(Graphics g)
    {
        g.DrawLine(Pen, Origin, Finish);
    }

    public override void GrowTo(int x2, int y2)
    {
        X2 = x2;
        Y2 = y2;
    }
}
```

Notes:

- The first line may be articulated as `Line` 'extends' `Shape`, or 'is a sub-type of', or 'inherits from' – these phrases all have the same meaning.
- `override` indicates that the method is deliberately acting in place of the (abstract) method of the same name defined in `Shape`.

Exercise 12

Add the new class `Shape` into the project. Make the changes shown to `Line`, and then make equivalent changes to `Rectangle`. Check that your code is compiling.

Record what happens if you make the following changes in turn. *After each change*, having captured the error, restore the code (using `Ctrl-z`) so it compiles again.

1) You delete the whole of the `Draw` method from `Line` (use `Ctrl-x` so you can restore it with `Ctrl-z`).

2) You remove `: Shape` from the `Line` class definition.

3) You remove the word `override` from one of the methods in `Line`.

4) You remove the word `abstract` from the line `public abstract class Shape`.

5) You remove the word `abstract` from either of the method definitions within `Shape`.

6) In any of the methods in `OOPDraw` (e.g. `Canvas_Paint`) you add the line `Shape s = new Shape();`

Read each error message carefully in each case to see how it relates to the code change – this will help you diagnose errors in future.

Notes:

- The error message arising from change number 3 indicates that you probably should have added the keyword `override`. The *other* option is best ignored for now – it is *rarely* the right answer.
- The error message arising from change number 6 clearly states the other significance of marking a class `abstract`: you cannot create instances of an abstract class – though you may create instances of a subclass (such as `Line` or `Rectangle`), provided *that subclass* is not itself marked as `abstract`.
- Classes not specified as `abstract` are sometimes referred to as 'concrete classes'. You *can* create instances of concrete classes.

Now we shall make use of this new abstract type.

In the code-behind, change:


```
private List<object> shapes = new List<object>();
```

to

```
private List<Shape> shapes = new List<Shape>();
```

and then replace the *two* places (in the `Canvas_Paint` and `Canvas_MouseMove` methods) where a variable `shape` has been defined as type `dynamic`, to now be of type `Shape`.

Check that the program works correctly, as before.

Exercise 13

1) Capture a screen snippet showing that if you now mis-spell the method name called in `shape.Draw(gr)` you will get a compile error.

Confirm also, for yourself, that temporarily deleting that line and typing just `shape.` will give you a pop-up list of the methods that includes both the method names defined on the abstract `Shape` class.

2) Does the pop-up list include any of the properties defined in `Line` and/or `Rectangle`?

3) There are also some other methods listed in the pop-up – what are the method names?

The four methods shown in addition to `Draw` and `GrowTo` are methods that are automatically implemented for *all* types of object by the C# compiler, behind the scenes. It is not necessary to know about these methods at this stage, but for those interested:

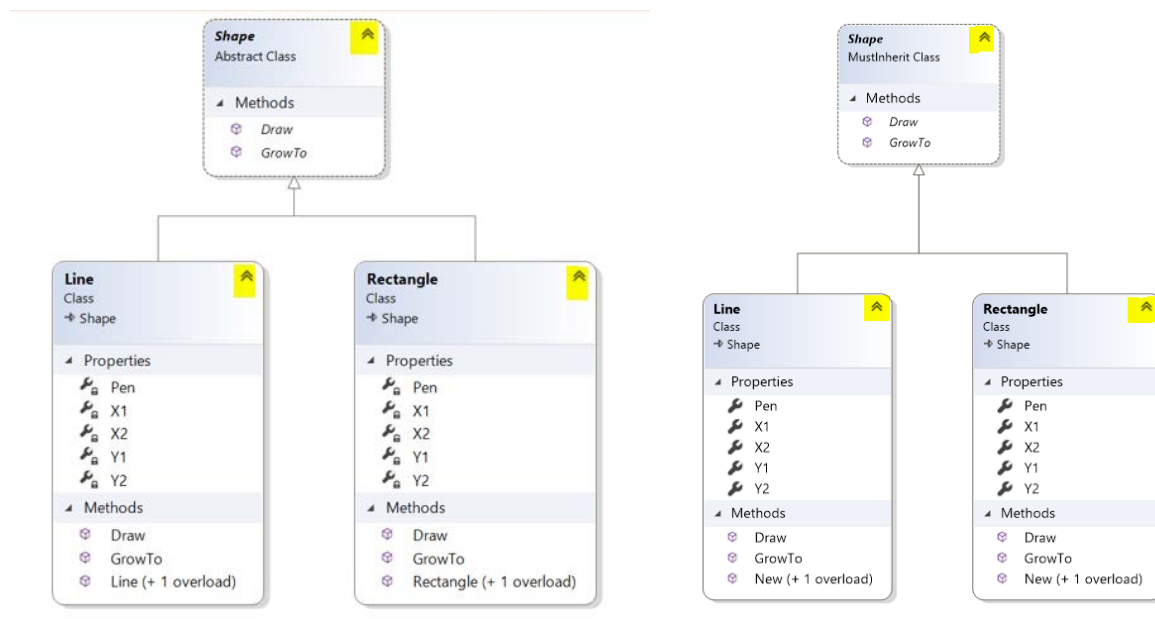
- `Equals` is a method to determine whether two instances have identical properties. Calling `a.Equals(b)` is not quite the same thing as `a == b`, because the latter checks that `a` and `b` are *the same instance*, rather than checking that two instances have the same properties.
- `GetHashCode` creates a 32-bit integer that constitutes a unique identifier for the object. This is used by the `Equals` method, but also has other uses, for example in `HashSet`, `Dictionary` and related data structures,
- `a.GetType()` will tell you the actual type that `a` is holding. You might have defined `a` as being of type `Shape`, but it might be holding a reference to a `Line`.
- `ToString` will provide a simple string representation of an object. This has many applications, but one simple one is to allow any object to be written using `Console.Write`. However, the string representation might not be what you want, in which case you have the option to override `ToString` to suit your own purposes.

Class hierarchy

It can sometimes be helpful to represent the relationship between different types diagrammatically.

Right-click on the *project* icon for **OOPDraw** and select **Add > New Item**, then from the next dialog select **Class Diagram**. (If this option does not appear in the list see [Adding Class Diagram capability to Visual Studio 2019](#).) Give it the name **OOPDraw.cd**.

Drag the icons for **Shape**, **Line**, and **Rectangle** from the **Solution Explorer** view into the new **OOPDraw.cd** tab. In each case expand the view in the diagram by clicking the double-arrow at the top of each box (highlighted below), and then move them around until your diagram has approximately this layout:



Exercise 14

- 1) The methods **Draw** and **GrowTo** are defined in abstract form in **Shape**, and in concrete form (i.e. with an implementation) in **Line** and **Rectangle**. What is the visual distinction between abstract and concrete methods in the diagram?
- 2) What happens if you double click, in the diagram, on a property or method in one of the classes?
- 3) Listed under the 'methods' in **Line**, is an entry also called **Line**. Try double clicking on it. What it takes you to is distinct from the other methods. What is it called?

The 'inheritance relationship' between classes is indicated by an arrow with an open *triangular* head. Try moving the **Shape** box so that it is physically underneath the other two. Notice the arrow.

4) Describe which way the arrow points in terms of its *meaning* (not in terms of the spatial positions of the boxes).

5) Move **Line** and **Rectangle** further apart and place **Shape** directly between the two. Capture a screen snippet showing what has happened to the arrow.

When done, edit the layout so that it looks like the original one above.

Notes:

- Another way of describing the ‘inheritance relationship’ in this example is that **Shape** is the ‘superclass’ of **Line** and **Rectangle** (we already learned that **Line** and **Rectangle** are subclasses of **Shape**).
- It is only a *convention* that superclasses are drawn above their subclasses. The hard and fast rule is that the arrow points *from* a subclass to a superclass.

Implementation inheritance

Where all the subclasses of a given superclass have one or more properties, or method *implementations*, in common, the definition may be ‘pulled up’ to the superclass. This has three benefits:

- It eliminates duplication. (In keeping with the DRY principle of coding – Don’t Repeat Yourself.)
- It makes those properties accessible to any variable where the type is defined by the superclass.
- Any new subclasses that you create automatically inherit these ready-made capabilities.

Select all five *properties* defined in `Line`, cut them from `Line`, and paste them into the `Shape` class (near the top). Now delete the same five properties from `Rectangle`.

This change will have resulted in a lot of compile errors. In the `Line` class, hover the mouse over any reference to the property `X1` (which will have a red squiggle) and read the pop-up error message.

Exercise 15

What does the error message say?

The problem is that we marked the `set` for all the properties `private`, so they can’t be modified outside the code of the `Shape` class. We could mark them `public`, but then they become modifiable by code everywhere.

Fortunately, there is a ‘half-way house’ between `private` and `public`: `protected`. A `protected` member may be accessed by code within the same class, or within any subclass, but not by other code.

In `Shape`, change the `set` for all five properties from `private` to `protected`, and confirm that the code now compiles without errors.

Then make the equivalent changes for the other four properties, including `Pen`. Confirm that the code now compiles.

Next, in `Line`, select the complete code for the method `GrowTo` (which is identical for `Line` and `Rectangle`), cut it from `Line`, and paste it into `Shape` in place of the abstract definition for `GrowTo` already in `Shape`.

Exercise 16

1) The `GrowTo` method in `Shape` will now cause a compile error. What does the message say?

Fix this error by deleting the keyword `override`. `GrowTo` should now look like any regular method definition.

This will then produce a new error in the `GrowTo` method in `Rectangle`, but we no longer need this as it is inherited from `Shape` – so delete `GrowTo` from `Rectangle`.

Confirm that the code compiles and the application is still working as before.

2) Why can we not do the same thing for `Draw`, that we have just done for `GrowTo`?

It is even possible to pull common code from constructors up into a superclass, by defining a constructor in the superclass. It might seem odd defining a constructor on an *abstract* class, such as `Shape`, which cannot be instantiated. However, the pattern works because each of the subclasses still defines its own constructor, but these *delegate the work* to the constructor in the superclass.

Using the code below, define two constructors in the `Shape` class (adopting the *convention* of placing constructors immediately after the property definitions). The code is similar to the constructors for `Line` and `Rectangle`:

```
public Shape(Pen p, int x1, int y1, int x2, int y2)
{
    Pen = p;
    X1 = x1;
    Y1 = y1;
    X2 = x2;
    Y2 = y2;
}

public Shape(Pen p, int x1, int y1) : this(p, x1, y1, x1, y1)
{
}
```

Now modify the constructors in `Line` as shown below, noting that the existing lines of code in the body of each constructor should be removed.

```
public Line(Pen p, int x1, int y1, int x2, int y2) : base(p, x1, y1, x2, y2)
{
}

public Line(Pen p, int x1, int y1) : base(p, x1, y1)
{
}
```

And then make equivalent changes to `Rectangle`. Confirm that the code compiles, and the application still works as before.

Modifications of this kind, where you move code between classes, are known as 'refactoring'. In this refactoring you have saved some code in the subclasses, but you needed to add some code to the superclass. The net reduction in code may be small, or even negative...

Exercise 17

- 1) Why might this refactoring result in greater savings to net code *going forwards*?
- 2) In addition to saved effort, what other advantage(s) can you see from eliminating the duplication of code?

Notes:

- The `base` keyword refers to the superclass of the class in which it is used. So here we are saying that both the constructors delegate to constructors with equivalent signatures in the superclass.

Adding a new subclass

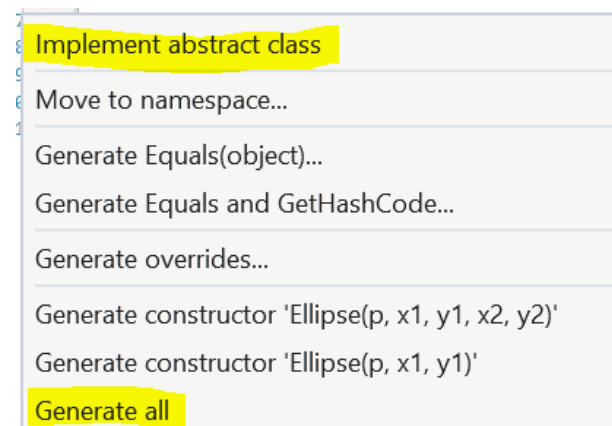
We are now going to see just how easy it is to add new types of shape into our OOPDraw application.

Add a new class named `Ellipse`, specifying that it extends/inherits from `Shape` as follows:

```
namespace OOPDraw
{
    public class Ellipse: Shape
    {
    }
}
```

The compiler will show some errors for `Ellipse`, indicating that it does not yet fulfil the requirements for a `Shape`.

Right-click on the word `Ellipse`, select **Quick Actions and Refactorings**, and from the options offered select the two shown highlighted below – which you will need to do as two separate actions:



Your `Ellipse` class should now have two constructors that mimic those of `Line` and `Rectangle`, and a `Draw` method, which just contains the line:

```
throw new System.NotImplementedException();
```

This line deliberately throws an 'exception', which would cause the program to halt and show an error if the Draw method were invoked. This is a safety mechanism. If we don't remember to write a proper implementation and just run the code we can see, when that method is called, that the cause of the error was that the method had not been implemented, rather than because something had gone wrong somewhere else in the code.

Replace that line with the following:

```
public override void Draw(Graphics g)
{
    int x = Math.Min(X1, X2);
    int y = Math.Min(Y1, Y2);
    int w = Math.Max(X1, X2) - x;
    int h = Math.Max(Y1, Y2) - y;
    g.DrawArc(Pen, x, y, w, h, 0F, 360F);
}
```

Notes:

- The implementation of this method uses the DrawArc method on Graphics. The first five parameters are identical to those of DrawRectangle: they define the enclosing rectangle for an arc, and so we have added the same code that we used in Rectangle to calculate the values.
- The last two parameters in DrawArc define the start and end of the arc in degrees – in this case 0 and 360, because we want a complete ellipse. The F after both those values is to convert the value from an integer to a float (short for 'single-precision floating-point number') which is the type required by the method.
- You will need to have using System; at the top to access the Math library; if you don't already have it, **Quick Actions and Refactorings** will offer this as an option.

Exercise 18

- 1) Capture your complete code for the Ellipse class.
- 2) Drag Ellipse from the Solution Explorer view into the class diagram and neaten-up the layout, ensuring that all the properties and/or methods are showing on each class. Paste a screen snippet of the expanded diagram.

Exercise 19

Make suitable modifications to the definition of the **Shape** combo box in the **[Design]** view, and then to the **Canvas_MouseDown** method in the code-behind, to allow you to draw an ellipse.

1) Capture the code change you made in **Canvas_MouseDown**.

Run the program and try to create an ellipse. If you are successful, try drawing several more. Sooner or later you will encounter an error.

2) When you do encounter the error, capture a screen snippet showing the error.

Stop the program and make the following change to the **Draw** method in **Ellipse** (the reason for the change will be explained after the exercise):

```
public override void Draw(Graphics g)
{
    int x = Math.Min(X1, X2);
    int y = Math.Min(Y1, Y2);
    int w = Math.Max(X1, X2) - x;
    int h = Math.Max(Y1, Y2) - y;
    if (w > 0 && h > 0)
    {
        g.DrawArc(Pen, x, y, w, h, 0F, 360F);
    }
}
```

Exercise 20

Then run again and confirm you can now create ellipses.

Capture a screen snippet showing at least four drawn ellipses of varying sizes (and/or colours and line widths).

Notes:

- If you were surprised to get the 'parameter is not valid' error message, so was the author! The ultimate cause of this error is, in the author's opinion, *a poor design decision in Microsoft code*. The methods **DrawLine**, **DrawRectangle**, and **DrawArc** all take four parameters (amongst others) that define an 'enclosing rectangle' for the shape. For the first two methods that enclosing rectangle may have zero height or width (which is what happens when, in our code, the corresponding shape is first created in **Canvas_MouseDown**). However, **DrawArc** will not accept a zero value for the height or width parameters – *and for no obvious reason!*
- Since we cannot (easily) edit the Microsoft code, we must 'work around' this problem. Although annoying this is a useful lesson in practical software development.
- Our fix is to draw the arc only if the height and width are greater than zero. It means that no ellipse will be drawn until the mouse is dragged at least one pixel away from the origin in both axes, which is fine.

- Warning: poor design like this in commercial software is *rare*. If you get exceptions when calling such functions, the *most likely* cause is a problem in your own code. You need to read the documentation for those functions carefully and test your own code thoroughly.

Adding a Circle class

We are now going to add a `Circle` class. Here is the complete code, with the parts that we want to draw your attention to highlighted:

```
namespace OOPDraw
{
    public class Circle : Shape
    {
        public Circle(Pen p, int x1, int y1) : base(p, x1, y1)
        {
        }

        public Circle(Pen p, int x1, int y1, int x2, int y2) : base(p, x1, y1, x2, y2)
        {
            GrowTo(x2, y2);
        }

        public override void Draw(Graphics g)
        {
            int x = Math.Min(X1, X2);
            int y = Math.Min(Y1, Y2);
            int w = Math.Max(X1, X2) - x;
            int h = Math.Max(Y1, Y2) - y;
            if (w > 0 && h > 0)
            {
                g.DrawArc(Pen, x, y, w, h, 0F, 360F);
            }
        }

        public override void GrowTo(int x2, int y2)
        {
            int diameter = Math.Max(x2 - X1, y2 - Y1);

            X2 = X1 + diameter;
            Y2 = Y1 + diameter;
        }
    }
}
```

Notes:

- The difference between the `Circle` class and the `Ellipse` is that we need to ensure that the height and width of the enclosing rectangle are kept the same. So instead of using the standard implementation of `GrowTo`, that is defined in `Shape` and inherited by all the other shapes as is, we are intending to *override* that standard implementation with a new one, just for this class. Note that this is not the same thing as providing an implementation for an *abstract* method (which has no standard implementation) – as we do with the `Draw` method, for example.

- This circle-specific implementation of `GrowTo`, calculates the diameter of the intended circle as the *greater* of the increment specified for either the x or the y axis. Then this diameter is used to calculate the new values for both X2 and Y2.
- Similar logic is required within the constructor, but rather than repeat it, we can simply delegate the calculation to the `GrowTo` method.

Add a new `Circle` class to the project and replace the default implementation with the code above. You will get at least one error.

(If you have more than one, first check whether you need to add any `using` statements.)

You still be left with an error in `GrowTo` saying that this method '*cannot override ...*'. An inherited member (unless it is marked `abstract` may only be overridden if the programmer has explicitly permitted it to be overridden. In C# the keyword to indicate this is `virtual`. So, change the `GrowTo` as highlighted below:

```
public virtual void GrowTo(int x2, int y2)
{
    X2 = x2;
    Y2 = y2;
}
```

Now make the necessary modifications to the **OOPDraw [Design]** view, and its code-behind, so that you have the option to draw circles. Run the program and confirm that as you grow the circle, it remains circular irrespective of whether you move the mouse horizontally or vertically.

Exercise 21

Capture a screen snippet showing circles of different sizes (or, if you prefer, a simple drawing that uses circles with other shapes).

Inheritance vs. delegation

You might have noticed that implementation of the `Draw` method is identical for `Ellipse` and `Circle`. And this implementation has four lines in common with that of the `Draw` method in `Rectangle`. Code duplication violates the 'DRY' principle of coding: Don't Repeat Yourself!

Could we use inheritance to eliminate this duplication? We *could* pull the implementation of `Draw` up to `Shape`, and then make `Line` and `Rectangle` override the implementation with their own versions, but this achieves no net savings. And worse: it doesn't *feel* right – it is hard to argue that the *default* implementation for any shape should use `DrawArc`!

Another option to consider would be to make `Circle` inherit from `Ellipse`, or *vice versa*.

The important principle here is that subclassing represents an 'is-a' relationship: a rectangle *is a* shape, a cod *is a* fish, a pupil *is a* person.

The Liskov substitution principle

The idea of the 'is-a' relationship was formalised by computer scientist Barbara Liskov, who was the first woman to receive the coveted IEEE John von Neumann Medal, and the second to receive the Turing Award.

Barbara stated that an object of type `T` may be substituted with any object of a subtype `S` without altering any of the desirable properties of the program. This has become known as the 'Liskov substitution principle'.

Picture credit and further information:

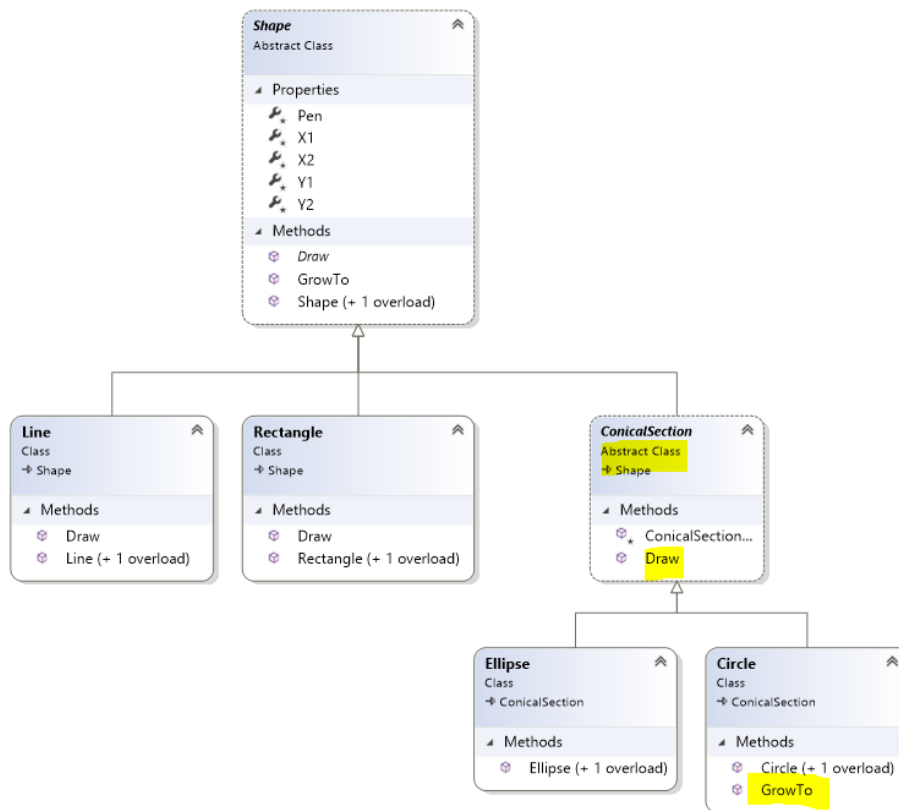
https://en.wikipedia.org/wiki/Liskov_substitution_principle



It may be argued, from algebra, that a circle is a specialised example of an ellipse, where the major and minor axes are both the same length. But it makes no sense to say that an ellipse is a specialised type of circle.

Problems arise, however, if you define methods on an `Ellipse`, such as `StretchHorizontally` or `StretchVertically`, that cannot be honoured by a `Circle`. A great deal of ink has been expended on the so-called 'circle-ellipse problem' in OOP (or its near-equivalent, the 'square-rectangle problem'), and its implications for the theoretical limits of the object-oriented approach.

One way to resolve the dilemma would be to add a new *abstract* class called say, `ConicalSection` (in formal 3D geometry, circle and ellipse are both 'conical sections'). We would now have two levels of inheritance (you can have as many as you wish) and the class diagram would be as shown below. Notice that the *implementation* of `Draw` is defined on `ConicalSection` and that `Circle` still overrides `GrowTo`.



This is *technically* a valid solution (though it still does not resolve the code duplication with `Rectangle`) – but it is *ugly*! The concept of a conical section might be correct in 3D geometry, but it has little meaning to a programmer designing a 2D drawing application. It is *too* abstract an idea.

Delegation

The better approach when you have duplicated code and no natural choice of common superclass to pull it up to, is to use *delegation*. Here the idea is to extract the *parts* that two or more methods have common and define that common code as a ‘helper’ method on a superclass or, when appropriate, as a free-standing function.

Helper method

There is no *hard* distinction between a ‘helper method’ and methods in general: we tend to use the former term when talking about a method that is deliberately designed just to provide part of the implementation of other methods, rather than implementing completely a desired behaviour of an object.

Define the following helper method within the `Shape` class (the syntax of this line will be explained after the exercise):

```
public (int, int, int, int) EnclosingRectangle()
{
    int x = Math.Min(X1, X2);
    int y = Math.Min(Y1, Y2);
    int w = Math.Max(X1, X2) - x;
    int h = Math.Max(Y1, Y2) - y;
    return (x, y, w, h);
}
```

Now reduce the `Draw` method of `Rectangle` to this:

```
public override void Draw(Graphics g)
{
    (int x, int y, int w, int h) = EnclosingRectangle();
    g.DrawRectangle(Pen, x, y, w, h);
}
```

Make *equivalent* changes to the `Draw` methods of `Ellipse` and `Circle`. Check that the code compiles, and that the application performs correctly. Then take this pattern a step further. Into the project, add a new class named `DrawingFunctions` using this code:

```
using System.Drawing;

namespace OOPDraw
{
    public static class DrawingFunctions
    {
        public static void DrawClosedArc(Graphics g, Shape shape )
        {
            (int x, int y, int w, int h) = shape.EnclosingRectangle();
            if (w > 0 && h > 0)
            {
                g.DrawArc(shape.Pen, x, y, w, h, 0F, 360F);
            }
        }
    }
}
```

Then replace the code in the `Draw` methods of `Ellipse` and `Circle` with just this function call:

```
DrawingFunctions.DrawClosedArc(g, this);
```

Again, check that the code compiles, and that the application performs correctly.

Exercise 22

- 1) Capture the final implementation of the `Draw` method in *each of* `Rectangle`...
- 2) `Ellipse`...
- 3) and `Circle`.

Notes:

- `(int, int, int, int)` defines a 'four-tuple'. A tuple is a simple and convenient mechanism to pass multiple values into or out of a method or function. In this case the tuple specifies four values, all integers in this case although the values of a tuple can be of different types.
- `(int x, int y, int w, int h) = EnclosingRectangle();` takes the four-tuple that will be returned by `EnclosingRectangle` and assigns the four values, separately, into four different new integer variables, `x`, `y`, `w`, and `h`. This coding pattern is known as 'deconstruction' (of a tuple).
- `DrawClosedArc` is defined as a `static` method. This means it is just a freestanding function – it does not belong to any object instance. Although it is defined within a class named `DrawingFunctions`, that class is also `static`. A static class is just a placeholder for freestanding (static functions) – it is not like the other classes we have defined so far, from which instances are created.
- The keyword `this` in the method call `DrawingFunctions.DrawClosedArc(g, this);` passed current object (which will be either a `Circle` or an `Ellipse` here) to provide the second parameter (defined as a `Shape`) that the method requires.

The changes that you made in the previous exercise are an example of the principle of 'favouring delegation over inheritance'. In this example, rather than create a complex and artificial-looking hierarchy of inheritance to eliminate duplication, we have instead extracted common code into helper methods and delegated the execution of other methods to those helper methods.

This has *not* eliminated 100% of the code duplication, but it has eliminated *almost all* of it, without introducing awkward looking abstract object classes.

As well as delegating execution to a helper method, it is possible to delegate implementation to an instance of another type of object, and we shall return to this in the chapter on [Association](#).

Enriching the application

In this chapter we will be extending the functionality of the application, learning new principles of OOP along the way.

First, having drawn shapes, it would be useful to be able to move them around. Since we would expect to move a shape continuously – by dragging the mouse – it would be better to specify the movement as an increment, rather than a destination, so `MoveBy` rather than `MoveTo`:

```
public void MoveBy(int xDelta, int yDelta)
{
    X1 += xDelta;
    Y1 += yDelta;
    X2 += xDelta;
    Y2 += yDelta;
}
```

This method will work on all types of shape so it can be defined on the abstract `Shape` class.

We can select the most recently drawn object – with `shapes.Last()` – and then call the `MoveBy` method within `Canvas_MouseMove`, but we need to specify that the mouse movement should move the last drawn shape rather than draw a new one.

Add the `MoveBy` method (above) into the `Shape` class.

In the form **[Design]** view add a fourth combo box. Name it **Action**, specify the string options as **Draw** and **Move**, and add a **Label** for it. In the `OOPDraw` constructor set the default option for the `Action` combo box as `Draw`.

In the `Canvas_MouseDown` method, select *all* the lines highlighted below, together, then right-click and select **Quick Actions and Refactorings > Extract method**.

```
private void Canvas_MouseDown(object sender, MouseEventArgs e)
{
    dragging = true;
    startOfDrag = lastMousePosition = e.Location;
    switch (Shape.Text)
    {
        case "Line":
            shapes.Add(new Line(currentPen, e.X, e.Y));
            break;
        case "Rectangle":
            shapes.Add(new Rectangle(currentPen, e.X, e.Y));
            break;
        case "Ellipse":
            shapes.Add(new Ellipse(currentPen, e.X, e.Y));
            break;
        case "Circle":
            shapes.Add(new Circle(currentPen, e.X, e.Y));
            break;
    }
}
```

Rename the extracted NewMethod to **AddShape**.

Then change Canvas_MouseDown again, to:

```
private void Canvas_MouseDown(object sender, MouseEventArgs e)
{
    dragging = true;
    startOfDrag = lastMousePosition = e.Location;
    if (Action.Text == "Draw")
    {
        AddShape(e);
    }
}
```

Now make the following change to Canvas_MouseUp:

```
private void Canvas_MouseUp(object sender, MouseEventArgs e)
{
    dragging = false;
    lastMousePosition = Point.Empty;
    Refresh();
}
```

And to Canvas_MouseMove:


```

private void Canvas_MouseMove(object sender, MouseEventArgs e)
{
    if (dragging)
    {
        Shape shape = shapes.Last();
        switch (Action.Text)
        {
            case "Move":
                if (lastMousePosition == Point.Empty) lastMousePosition =
e.Location;
                shape.MoveBy(e.X - lastMousePosition.X, e.Y -
lastMousePosition.Y);
                break;
            case "Draw":
                shape.GrowTo(e.X, e.Y);
                break;
        }
        lastMousePosition = e.Location;
        Refresh();
    }
}

```

Exercise 23

Run the application, draw two different shapes. Change the **Action** to **Move** and drag the mouse to move the last drawn shape.

- 1) Which shape is moved?
- 2) Does it matter, when moving a shape, whether you start the drag from inside the object or not?

The next step is to be able to move any previously-drawn shape – ideally multiple shapes at once. A common gesture for selecting multiple objects is dragging the mouse to draw a *temporary* rectangle around the objects of interest, and to highlight the objects enclosed in some way. We will start with the latter part.

Add a new property to the `Shape` class, located near the top with the other properties:

```
public bool Selected { get; private set; }
```

then add these two new methods below the other *methods*:

```
public void Select()
{
    Selected = true;
    Pen.DashStyle = DashStyle.Dash;
}

public void Deselect()
{
    Selected = false;
    Pen.DashStyle = DashStyle.Solid;
}
```

(You will need to add the appropriate `using` statement, but this will be offered as an option under **Quick Actions and Refactorings**).

Note: The `Selected` property is `public` but its 'setter' is marked `private`. So code outside the class can read the property value, but will only be able to modify it by calling the `Deselect` method defined. This is to enforce that the shape cannot be selected without changing the `Pen`.

In the `OOPDraw` code-behind add this new method:

```
private void DeselectAll()
{
    foreach (Shape s in shapes)
    {
        s.Deselect();
    }
}
```

To test this, in the `AddShape` method *temporarily* add this line *at the start of the method body*:

```
DeselectAll();
```

And the following line at the very end of the method body:

```
shapes.Last().Select();
```

Exercise 24

Run the program. Draw multiple shapes, *changing the colour OR line width between drawing each new shape*. You should see that the shape is shown with a dashed line, but this will change to a solid line when the next shape is drawn.

1) Capture a screen snippet showing this.

2) Now draw two shapes (can be the same shape or different shape) *without changing the colour or line width in between*. Capture a screen snippet.

Then change the colour or line width and draw another shape.

3) Describe what you observe happening (repeating the process if you are not sure).

Reference Types

The behaviour of the application is *not* what we had intended. However, the bug illustrates an important principle of OOP. Objects are *reference types*, not value types (such as `int`, or `bool`): when you pass an object instance as an argument to a method, the method *does not make a copy* of the instance, it continues to work with a reference to the *same* instance that you passed it. When you create two shapes (they can even be different shapes) *without* changing the colour or line width, the *same instance* of `Pen` (held in `currentPen`) is passed into each object and a reference to it held in each instance's `Pen` property. When you draw the first shape, its `Select()` method is called. This changes the style of the `Pen` to a dashed line. When you create the second shape, all shapes are deselected, but then the same `Pen` is immediately changed back to selected (dashed) mode.

Very often, this sharing of references is the desired behaviour, but occasionally – as in this case – it is not desirable. Here is the fix:

Change the *first line* in the first constructor of the `Shape` class as highlighted:

```
public Shape(Pen p, int x1, int y1, int x2, int y2)
{
    Pen = new Pen(p.Color, p.Width);
    . . .
}
```

Exercise 25

Run the program again and confirm that if you draw more than one shape with the same line width and colour, only the last one drawn is selected (dashed).

Why (before this change) did the program work correctly provided you had changed either the colour or line width? Hint: look at the code (you added it some while back) that is called when you change the selected item in either of those combo boxes. Write your answer as text.

Remove the lines at the top and bottom of the `AddShape` method that you temporarily added in the previous exercise i.e.

```
private void AddShape(MouseEventArgs e)
{
    DeselectAll();
    . . .
    shapes.Last().Select();
}
```

Notes:

- The modified first line of the `Shape` constructor now creates a new `Pen`, but using the same colour and width values as for the pen passed in as a parameter. This way the `Shape` object will be created with a reference to its own `Pen`, that won't be subject to unintended changes as a result of changes to a `Pen` made somewhere else.

Now we will add the ability to select multiple objects by dragging a 'selection rectangle' around them. If we are careful, we can re-use the existing `Rectangle` class to draw this selection rectangle, but using a distinctive `Pen`. We will also need some functionality to determine whether any given shape falls within the current bounds of the selection rectangle.

We *could* consider defining a method in `Shape`:

```
public bool FallsWithin(Rectangle selection)
{
    ...
}
```

This could be made to work, but the superclass (`Shape`) would then 'know about' one of its subclasses (`Rectangle`). This is what is known by professional programmers as a 'design smell'. Subclasses necessarily 'know about' their superclass, but it should never be the other way around.

We can fix this by *inverting* our conception of the method (and, hence, the name too) and then placing the method in `Rectangle` instead:

```
public bool FullySurrounds(Shape s)
{
    (int x, int y, int w, int h) = this.EnclosingRectangle();
    (int xs, int ys, int ws, int hs) = s.EnclosingRectangle();
    return x < xs && y < ys && x + w > xs + ws && y + h > ys + hs;
}
```

Add the new `FullySurrounds` method into the `Rectangle` class. Then add the following new property and new method within the `OOPDraw` code-behind:

```
Rectangle selectionBox;

private void SelectShapes()
{
    DeselectAll();
    foreach (Shape s in shapes)
    {
        if (selectionBox.FullySurrounds(s)) s.Select();
    }
}
```

In the **[Design]** view, add a new option **Select** into the **Action** combo box. Modify the `Canvas_MouseDown` method in the code-behind to use a `switch` statement covering the **Draw** and **Select** actions:

```
private void Canvas_MouseDown(object sender, MouseEventArgs e)
{
    dragging = true;
    startOfDrag = lastMousePosition = e.Location;
    switch (Action.Text)
    {
        case "Draw":
            AddShape(e);
            break;
        case "Select":
            Pen p = new Pen(Color.Black, 1.0F);
            selectionBox = new Rectangle(p, startOfDrag.X, startOfDrag.Y);
            break;
    }
}
```

and then add a new case into the `switch` statement in `Canvas_MouseMove`:

```
case "Select":
    selectionBox.GrowTo(e.X,e.Y);
    SelectShapes();
    break;
```

Now add new code at the end of the `Canvas_Paint` method to draw the selection box:

```
if (selectionBox != null) selectionBox.Draw(gr);
```

And in the `Canvas_MouseUp` method, add the following line (*above* the call to `Refresh`) to remove the box when the selection is complete:

```
selectionBox = null;
```

Exercise 26

Run the program, drawing multiple shapes then use **Action > Select** and drag to select some, but not all, of your shapes. Capture a screen snippet.

Notes:

- In adding the `FullySurrounds` method to `Rectangle`, notice that this subclass of `Shape` has a method that does not exist in its superclass. This is quite normal: a subclass is a specialised version of the superclass: it may implement inherited methods differently and it may define *additional* properties or methods. However, a subclass cannot *remove* methods or properties defined in the superclass.

Now that we can select multiple shapes, the next task is to make the **Move** action work with the shapes selected.

Into the form's code-behind add this new method:

```
private List<Shape> GetSelectedShapes()  
{  
    return shapes.Where(s => s.Selected).ToList();  
}
```

Note that this code uses a 'LINQ' statement (see panel after the exercise) – **Where** – to filter the list of all shapes, finding those where the **Selected** value is true. Remember that we deliberately specified that the **Selected** property should be publicly *readable* though not writeable.

Then add the following new method, which calls **GetSelectedShapes** and then calls the **MoveBy** method on each of them.

```
private void MoveSelectedShapes(MouseEventArgs e)  
{  
    foreach (Shape s in GetSelectedShapes())  
    {  
        s.MoveBy(e.X - lastMousePosition.X, e.Y - lastMousePosition.Y);  
    }  
}
```

Then change the **Move** case within **Canvas_MouseMove** to:

```
case "Move":  
    MoveSelectedShapes(e)  
    break;
```

Also, *move* the line `Shape shape = shapes.Last();` from above the **switch** statement to being the first line *after* case **"Draw"** :, as this is no longer relevant to the **Move** case.

Run the program. **Draw** several shapes, then **Select** and **Move** some of them.

This must be working correctly before you can progress to the next chapter.

LINQ

LINQ, which stands for 'Language INtegrated Queries', may be thought of as a query language. It is *somewhat* like SQL – you might recognise the function names **Where** and **Select**, for example – but, unlike SQL, is fully integrated within the **C#VB** language, and able to operate on objects in memory (as well as on databases, incidentally, via the Microsoft 'Entity Framework'). Most professional programmers writing **C#VB**, today, make extensive use of LINQ.

Association

Objects may hold references to, or know how to access, other objects. An object might rely on such ‘associated’ objects to help them fulfil either their own know-what or know-how-to responsibilities.

We have already seen an example of association, although we have not previously described it as such: each `Shape` has an associated `Pen` object held, as a reference, in the property named `Pen`. The `Shape` uses the `Pen` object to help fulfil the responsibility of being able to draw itself on the canvas.

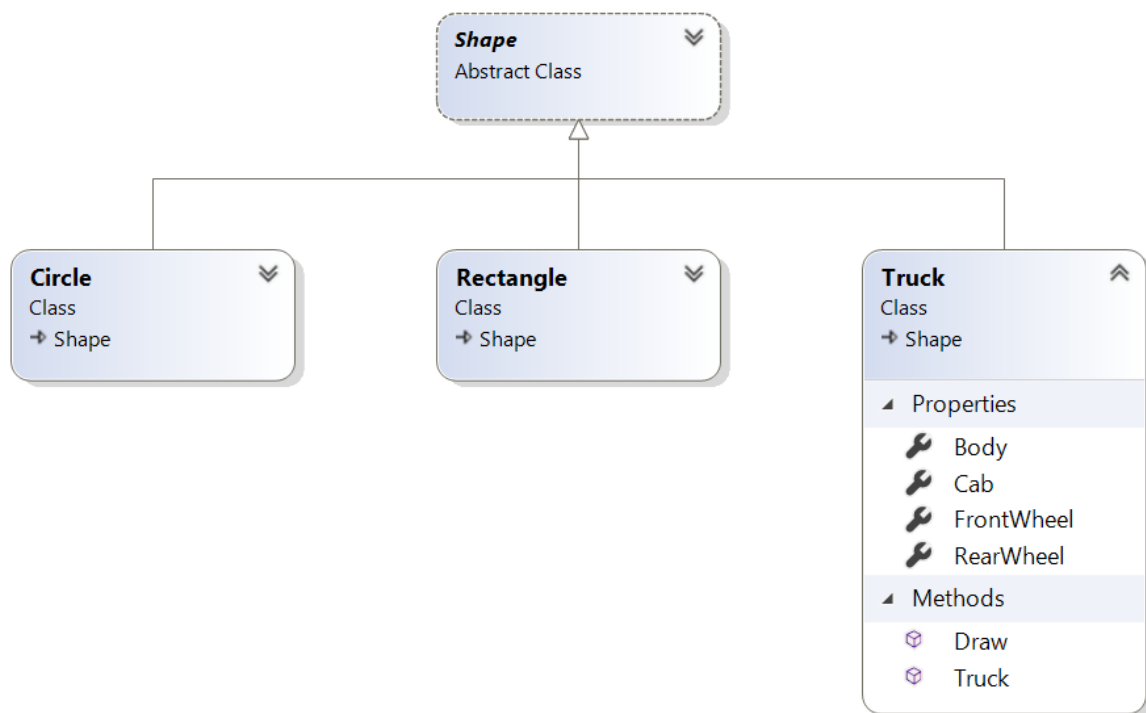
The association pattern becomes more obvious when we start forming associations between object classes that we have defined in our application. In our example we are going to create associations between `Shapes`.

One way to use an association between `Shapes` would be to build more a specialised `Shape` class that included other `Shapes`, potentially different subclasses of `Shape`, held as properties. For example, we might envisage a class representing a simple 2-dimensional drawing of a `Truck` that looked something like this:

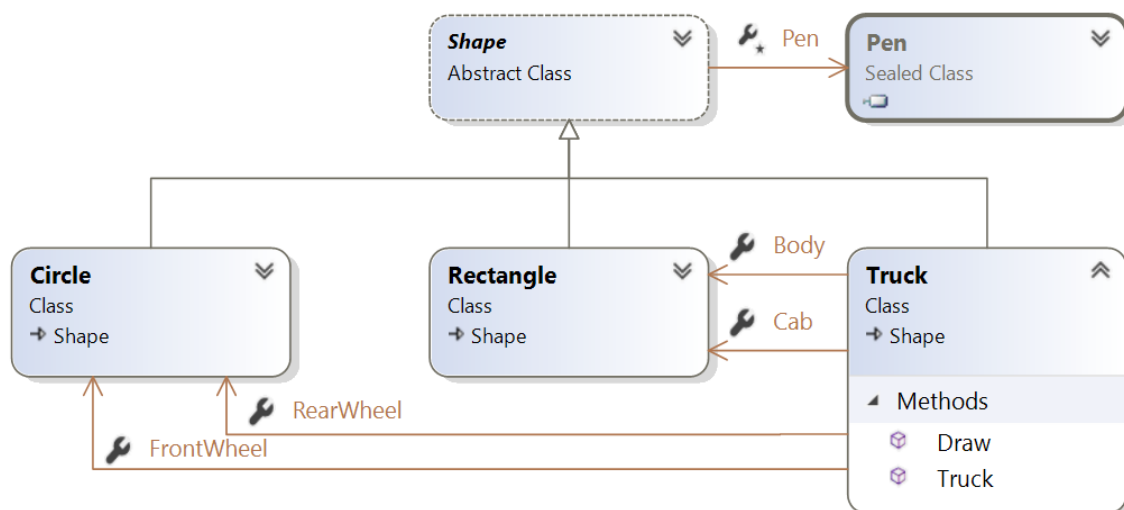
```
public class Truck : Shape
{
    public Rectangle Cab { get; private set; }
    public Rectangle Body { get; private set; }
    public Circle FrontWheel { get; private set; }
    public Circle RearWheel { get; private set; }

    // + other properties, constructors, and methods
}
```

The following class diagram shows the *inheritance* relationships, with `Truck` showing its *additional* properties. The other classes are shown in collapsed view, and we have excluded some other shapes, for clarity. (As a general rule: class diagrams are best used to illustrate or explain the relationships between just a handful of classes, not to capture all the classes in an application.)



It is also possible to show the *association* relationships diagrammatically, including the association to the Pen class in the System.Drawing library:



(A **Sealed Class** is one that, deliberately, may not have any subclasses derived from it).

Notice that the arrows defining *associations* are visually distinct from those showing inheritance. The exact format of the arrows will depend on the visual diagramming tool used. The most important thing to understand is that inheritance and association are fundamentally different relationships:

- Inheritance defines an 'is-a' relationship, for example: a `Rectangle` *is a* (specialised kind of) `Shape`.
- Association defines a 'has-a' relationship, for example: a `Shape` has a `Pen`; a `Truck` has a `Cab` (`Rectangle`) and a `Body` (`Rectangle`).

A common mistake made by newcomers to OOP is to confuse these two things, for example in deciding that because a customer has an address the `Customer` class should perhaps inherit from `Address`. No: `Customer` should have an *association* to an `Address`.

Our hypothetical `Truck` class *might be* of use in specialised application where drawing standard simple representations of trucks in different colours and line-widths was a frequent requirement (perhaps as part of a traffic management application). For a general-purpose drawing application like `OOPDraw`, however, the user wants the ability to create their own drawing from multiple components, and then move it around as a unit, make duplicates and so forth. If you've used an off-the-shelf drawing package such as **PowerPoint** you will know that you have the option to *group* shapes, and that a group may then be manipulated just as if it was a simple shape.

What we need is a class, inheriting from `Shape`, but holding an association to an arbitrary number of other shapes, of different types. We could call this new class, 'Group', but a more descriptive name is: `CompositeShape`.

Here is a starting point:

```
namespace OOPDraw
{
    public class CompositeShape : Shape
    {
        private List<Shape> Components { get; set; }

        public CompositeShape(List<Shape> components)
            : base(new Pen(Color.Black, 1.0F), 0, 0, 0, 0)
        {
            Pen.DashStyle = DashStyle.Dash;
            Components = components;
            CalculateEnclosingRectangle();
        }

        private void CalculateEnclosingRectangle()
        {
            X1 = Components.Min(m => Math.Min(m.X1, m.X2));
            Y1 = Components.Min(m => Math.Min(m.Y1, m.Y2));
            X2 = Components.Max(m => Math.Max(m.X1, m.X2));
            Y2 = Components.Max(m => Math.Max(m.Y1, m.Y2));
        }
    }
}
```

Notes:

- `CompositeShape` defines one new property (in addition to those it inherits from `Shape`) named `Components`. The type of this property is not a single domain object type, but `List<Shape>`, so that it can hold a list of other `Shapes` (of different types, potentially).
- Another name for this specific pattern is a 'collection association'.
- `CompositeShape` defines a single constructor that takes in a list of components. It extends the standard base constructor, passing in a specific new `Pen` object. We will see how this pen is used later on, but note, for now, that each of the component (`Shapes`) will have its own `Pen` to draw itself with. The call to the base constructor also specifies `0` for each of the four coordinate values, because ...
- The body of the new constructor calls `CalculateEnclosingRectangle` which calculates the correct values for the four coordinate properties based on the minimum and maximum values for the coordinates of all the components. The net effect is that the enclosing rectangle should enclose all the enclosing rectangles of the components.

Add the `CompositeShape` class into the project, adding the necessary `using` statements.

Then add these two methods into `CompositeShape`:

```
public override void Draw(Graphics g)
{
    foreach (Shape m in Components)
    {
        m.Draw(g);
    }
    if (Selected) g.DrawRectangle(Pen, X1, Y1, X2 - X1, Y2 - Y1);
}

public override void MoveBy(int xDelta, int yDelta)
{
    foreach (Shape m in Components)
    {
        m.MoveBy(xDelta, yDelta);
    }
    X1 += xDelta;
    Y1 += yDelta;
    X2 += xDelta;
    Y2 += yDelta;
}
```

Exercise 27

You will get compile error on the `MoveBy` method. Look at the error message: you have seen a message like this before (hint: when you created `Circle`).

Fix the code and explain what you had to do, and what the change means.

Notes:

- The implementation of the `Draw` method on `CompositeShape` just delegates responsibility to the `Draw` method on each of its components in turn. This is a powerful example of OOP.

- At the end of the `Draw` method is code that, if the `CompositeShape` is `Selected`, will draw a thin black dashed-line around the enclosing rectangle – this is to make the selection of a `CompositeShape` distinct from a manual selection of independent shapes. Since the `CompositeShape` has no need of a `Pen` to draw itself, we might as well use the inherited `Pen` property to hold a suitable `Pen` for drawing the enclosing rectangle as an outline – set up in the constructor.
- The `MoveBy` method also passes on the same instruction to each of its components. It must also adjust the enclosing rectangle, by adjusting each of the coordinates.

Exercise 28

- 1) There is a simpler way by which `MoveBy` could adjust the coordinates of the enclosing rectangle. What is it? Describe the change and then implement it.
- 2) If, instead of drawing a thin black dashed-line around the enclosing rectangle, we instead decided to show that a `CompositeShape` was selected by showing each of the components as selected, what is the simplest way to achieve this, and where would the changes have to be made. Describe your approach – do not implement it.

To create a `CompositeShape` the user can select multiple existing shapes, using the existing **Select** action, and could then invoke a new **Group** action. Note, however, that unlike the other actions in the **Action** combo box, **Group** needs to be enacted immediately (assuming that more than one shape *has* already been selected – something we will need to check).

In the **[Design]** view, add `Group` as a new string option for the **Action** combo box. Then *double-click* on the **Action** combo box to create the method `Action_SelectedIndexChanged` in the code-behind. This is a method that will be called *whenever the user selects a new action*.

Make the following changes to that method:

```
private void Action_SelectedIndexChanged(object sender, System.EventArgs e)
{
    switch (Action.Text)
    {
        case "Group":
            GroupSelectedShapes();
            break;
    }
}
```

(We don't *need* a `switch` statement here as we are reacting only to the selection of the **Group** action – we are just anticipating the likelihood that we will be adding more actions that need to be handled this way.)

Then define the new method that is being called from the method above:

```
private void GroupSelectedShapes()
{
    var members = GetSelectedShapes();
    if (members.Count < 2) return; //Group has no effect
    CompositeShape compS = new CompositeShape(members);
    compS.Select();
    shapes.Add(compS);
    foreach (Shape m in members)
    {
        shapes.Remove(m);
        m.Deselect();
    }
    Refresh();
}
```

Exercise 29

Run the program, draw multiple shapes, **Select** some of them and then invoke the **Group** action.

1) Capture the resulting drawing, showing the composite shape selected.

Prove to yourself that you can now move the composite shape.

2) Is it possible to create new composite shapes from other composite shapes as well as from individual shapes? Why?

It can be helpful to view the relationships of the `CompositeShape` class in our class diagram.

Exercise 30

Drag `CompositeShape` into the class diagram and adjust the layout if needed such that all the subclasses of shape are below `Shape` (though you might stagger them up and down to save horizontal space). Ensure that `CompositeShape` is at one end, though.

1) Expand the view of `CompositeShape`, but show all the other objects collapsed. Capture the class diagram.

2) Right-click on the `Components` property and select **Show as Collection Association**. Capture the revised class diagram.

3) What is the visual distinction (for this modelling tool) between an association and a collection association in diagrammatic form?

4) Right-click on the arrow now representing the collection association and select **Show As Property**. The right-click on the new property and select **Show as Association**. (Not 'Collection Association' this time). Capture the new class diagram.

Types of association

There are many ways to distinguish types of association between classes. For example:

- **Direct vs. Indirect**. A direct association is where the associated object is held as a property of an object; an indirect association is where an object is passed the associated object as a

parameter into a method, or where the object knows how to go and find another object of interest.

- **Single vs. Multiple.** A multiple association is also known as a 'collection association' – we have seen this used in `CompositeShape`.
- **Unidirectional vs. Bidirectional.** In our class diagram, a `Shape` holds a reference to the `Pen` that is it using, but a `Pen` does not hold a reference to the `Shape` or `Shapes` that use it. But there are situations where the ability to navigate from one object to an associated object *both ways* is useful.
- **Mandatory, or Optional.** Some properties are allowed to hold a `null` value, some not. Some collection associations may hold an empty list, some require that the list hold at least one domain object, and some might place more specific rules on the minimum and/or maximum number of members in that collection association.
- **Aggregation vs. Composition.** Composition is a form of direct association where the associated object(s) are said to be owned, exclusively, by an object.

Aggregation and Composition in OOPDraw

In OOPDraw the association between `CompositeShape` and its component `Shapes` may be described as *composition*. This is because in the current design of OOPDraw (in common with most drawing tools) any specific `Shape` instance may not be a component of more than one `CompositeShape`. And if the `CompositeShape` were to be deleted (we'll come to that in the next chapter) then its component `Shapes` would also be deleted. (Note, however, most drawing programs permit a group, or composite shape, to be ungrouped – and this could be added to OOPDraw. In this event the `CompositeShape` is deleted but the component shapes now have an independent existence again).

Early on in this book, the relationship between `Shape` and `Pen` was non-exclusive – so it could *then* have been described as *aggregation*. However, we found in the previous chapter, that this sharing of a `Pen` between `Shapes` had unwanted side effects, so we changed the implementation such that each `Shape` now creates its own separate *copy* of the `Pen` instance passed into the constructor. As the implementation *now* stands, the relationship of `Shape` to `Pen` is therefore *composition*, and if the `Shape` were to be deleted, then its own `Pen` object would be deleted, too.

Deleting and Duplicating objects

Deleting objects

Now that the user can select one or more of the shapes drawn it would be useful to add a simple action that would delete those selected shapes. This is simple to implement.

In the **[Design]** view add a **Delete** option to the **Action** combo box.

In the code-behind, add this new method:

```
private void DeleteSelectedShapes()
{
    foreach (Shape s in GetSelectedShapes())
    {
        shapes.Remove(s);
    }
    Refresh();
}
```

Then in `Action_SelectedIndexChanged`, add this new case into the existing `switch` statement:

```
case "Delete":
    DeleteSelectedShapes();
    break;
```

Exercise 31

Run the application and draw several shapes, then **Select** some of them and invoke the **Delete** action. Verify that the selected objects have disappeared from the canvas.

What happens if you invoke **Delete** without having selected any shapes? Explain why, with reference to the code in the `DeleteSelectedShapes` method.

Although the last exercise has the desired result of removing the selected Shapes from the canvas, has it really *deleted* the objects in memory? And is that an important question?

When any new object instance is created, space in memory (specifically in an area of working memory known as the ‘heap’) will be reserved for that object – the amount of memory required will be determined by the number and type of properties in the class definition. Even the smallest modern computer now has so much memory (compared to early computing devices) that the programmer seldom has to worry about running out of memory. But what if a user of `OOPDraw`, in creating a *very* complex drawing, repeatedly added new shapes then removed them. Would the heap eventually fill up – or, if it dynamically expanded, would the heap eventually occupy all the available memory? Is there specific code needed to ensure that an object is removed from memory (or, rather, the memory reserved for that object is marked as being available to be used for another purpose)?

The answer to both those questions is: ‘no’. This is thanks to a powerful capability of all modern OOP languages, known ‘garbage collection’. Early OOP languages had, by comparison, only a limited form of garbage collection – or none at all – leaving the programmer with much more responsibility for ‘memory management’.

Garbage collection, which operates in the background, frees up space in the heap whenever an object instance is no longer needed. How does it know when an object is no longer needed? Answer: when nothing is holding a reference (also known as a ‘pointer’) to that object, in other words:

- No current variable holds a reference to that instance
- No reference to that instance is held in a data structure such as a list
- No other object instance has a direct association to that object (i.e. a property/field holding a reference to it).

But is it right to assume that if there are no references to an instance that it is no longer needed? Yes – because if there is no reference to that instance there is no way (through normal coding) to get hold of that instance again, anyway.

Going back to OOPDraw does that mean that, having selected some shapes, then as soon as you invoke **Delete** and the shapes disappear from the canvas, that the garbage collector will then remove them from memory? Well, in the `DeleteSelectedShapes` the call to `GetSelectedShapes` will create a new list containing references to all the shapes (from the shapes list) that have their `Selected` property set to `true`. The `foreach` loop then removed each of these selected shapes *from the shapes list*, but it does not remove them from the new list of just the selected shapes. However, this list of selected shapes, was created *within the scope of the DeleteSelectedShapes code*, and since this list is not returned as a result it will automatically be deleted as soon as the `DeleteSelectedShapes` method is completed. At that point, nothing will be holding a reference to any of those selected shapes, and garbage collection will free up the memory.

Duplicating or ‘cloning’ objects

It would be convenient to be able to duplicate *any* existing shape – including a composite shape – where the duplicate has the same size, line width, colour(s) and any other properties that we might add in future.

We have already seen an example of duplicating (this is sometimes called ‘cloning’) an object: in the constructor of `Shape` we clone the `Pen` object passed in as `p`, thus:

```
Pen = new Pen(p.Color, p.Width);
```

Exercise 32

- 1) Write a method in the class `Line`, named `Clone`, which defines no parameters, but returns a new `Line` that is clone of the current one.
- 2) Now write an equivalent method in `Circle`.
- 3) Your two methods should be similar in structure, though clearly not identical. This will be true for all the subclasses of `Shape` (that we have created so far) *except* for `CompositeShape`. Without writing the method, explain in words what the `Clone` method for `CompositeShape` would have to do.

The specific challenge we face when writing a `Clone` method for `CompositeShape`, or if we want the user to be able to duplicate multiple shapes that have been selected on screen, is that we need to be able to call the `Clone` method on each shape, *without knowing the specific type of that shape*.

Exercise 33

- 1) What principle of OOP is being described in the sentence above?
- 2) What specific problem prevents us from using the `Clone` methods in this way?

We could solve this problem using either the static typing or dynamic typing approach (see [Static typing and dynamic typing](#)): we will stick with the first of those. The trick here is to define all the `Clone` methods as returning a `Shape`. Note that what each `Clone` method actually returns is an instance of a specific subclass of `Shape`, but this fine – it is still a `Shape`.

Exercise 34

In the `Shape` class, define an *abstract* method named `Clone`, that returns a `Shape`.

For your implementations of `Clone` on `Line` and `Circle`, there is no need to change the body of those methods, but you must make two changes to the method signature *in each case*.

- 1) What are the *two* changes that you must make to each of those two `Clone` methods?

Now create equivalent methods for `Rectangle` and `Ellipse`.

For `CompositeShape`, the implementation of `Clone` should follow this logic.

- Create a new `List` of type `Shape`.
- Iterating through the `Components`, clone each component and add the clone to the new list.
- Create, and return, a new `CompositeShape`, passing in the list of clones to the constructor.

Implement this method and confirm that your solution is compiling without errors.

- 2) Capture your implementation of `Clone` method on `CompositeShape`.

To use this new capability, we will need to add a new action, which we will call **Duplicate**, and which will invoke the `Clone` method on each of the selected shapes, and add the clone(s) into the

shapes list. (We could name the action 'Clone', but that word has more meaning to programmers than to users of a drawing program).

However, since a cloned shape has identical properties to the original, it will be drawn exactly over the original – leaving the user with the impression that nothing has happened. So, after creating the clone we shall offset it from the original by 50 pixels in both X and Y axes. (You might be aware that most off-the-shelf drawing packages do something similar.)

Exercise 35

In the **[Design]** view add a new option **Duplicate** to the **Action** combo box.

In the code-behind create a new method named `DuplicateSelectedShapes`, and implement this method using the description for the algorithm:

For each selected shape

- Deselect that shape
- Clone that shape into a new variable
- Move the clone by 50 pixels on each axis
- Select the clone
- Add the clone to the shapes list.

When all selected shapes have been cloned, call `Refresh` to re-draw the canvas.

1) Capture your implementation of `DuplicateSelectedShapes`.

In the `Action_SelectedIndexChanged` method, add a case for the **Duplicate** action, which calls the new `DuplicateSelectedShapes` method.

2) Run the program, make a drawing using multiple shapes. **Group** them. **Select** the composite shape, and **Duplicate** it. Then **Move** the duplicate so it is alongside the original. Capture a screen snippet of the duplicated drawing.

Suggested further enhancements and extensions

Although you have now completed all the formal exercises in this book, hopefully you have gained an appetite to do more OOP. Perhaps you already have ideas for a new project using OOP. If not, then you can gain a lot of useful practice by continuing to enhance and extend the functionality of OOPDraw can be enhanced and extended to emulate many of the features found in off-the-shelf drawing programs. Here are just a few of the possibilities that could be added, using the programming techniques already learned in this book.

- 1) Add further types of shape such as a (partial) Arc, a Multi-segment line (each mouse click starts a new line extending from the end of the previous one), or a Freehand drawing that captures *each* tiny movement of the mouse as a line from the previous position.
- 2) With one or more shapes selected, make it so that if the user selects a new option in the Width or Colour combo boxes, that changes the Pen for each selected shape. (But remember to make a copy of the new Pen in each object).
- 3) Allow all shapes to be filled with a solid colour
- 4) Allow selected shapes to be grown (or reduced) in size.
- 5) Ability to 'flip' or 'invert' selected shapes (only useful for non-symmetrical shapes).
- 6) Allow a selection of multiple shapes to be aligned vertically or horizontally, or 'distributed' with equal spacing, vertically or horizontally.
- 7) Improve usability by 'anticipating' the next action. For example, having completed a **Duplicate** action, the action combo box could be programmatically set to the **Move** action. Create an option to **Undo last action**. At the start of each action, create a separate list of clones of all the current shapes. **Undo last action** then simply replaces the current list of shapes with the clones of the previous ones.

Part II – An object-oriented records-management system

OOP and records management

Many programmers associate OOP with the kind of interactive graphical program that we developed in Part I. However, OOP is applicable to almost all forms of programming.

To make this clearer, in Part II we are going to develop an entirely different type of application, one that we might call 'Records Management'. Such applications typically place much less emphasis on the richness of the user interface, and much more on the structure and integrity of the *data*. Indeed, they are sometimes referred to as 'data-centric' applications. They account for a huge proportion of the commercial application of computing: airline reservation systems, banking and financial trading systems, government administration, accounting, e-commerce, and manufacturing systems. At a more modest level of complexity, it includes school management systems.

OOP has been slower to make an impact on these traditional records management applications. But not only is OOP *suitable* for writing such applications, it offers huge advantages. The author of this book has designed many such projects, including one of the largest and most complex commercial systems - that uses pure OOP - in the world. It has more than 5000 object classes, of which more than 95% correspond to business domain 'entities' recognisable to a business user: Customer, Payment, Address, Document, Pension, MedicalCertificate, Doctor, Company, Employee, Officer... . The instances of these classes – tens of millions of them for some classes – are stored in a database, measured in terabytes. And the systems are used *all-day every day* by thousands of government officers, and on an occasional basis by millions of citizens.

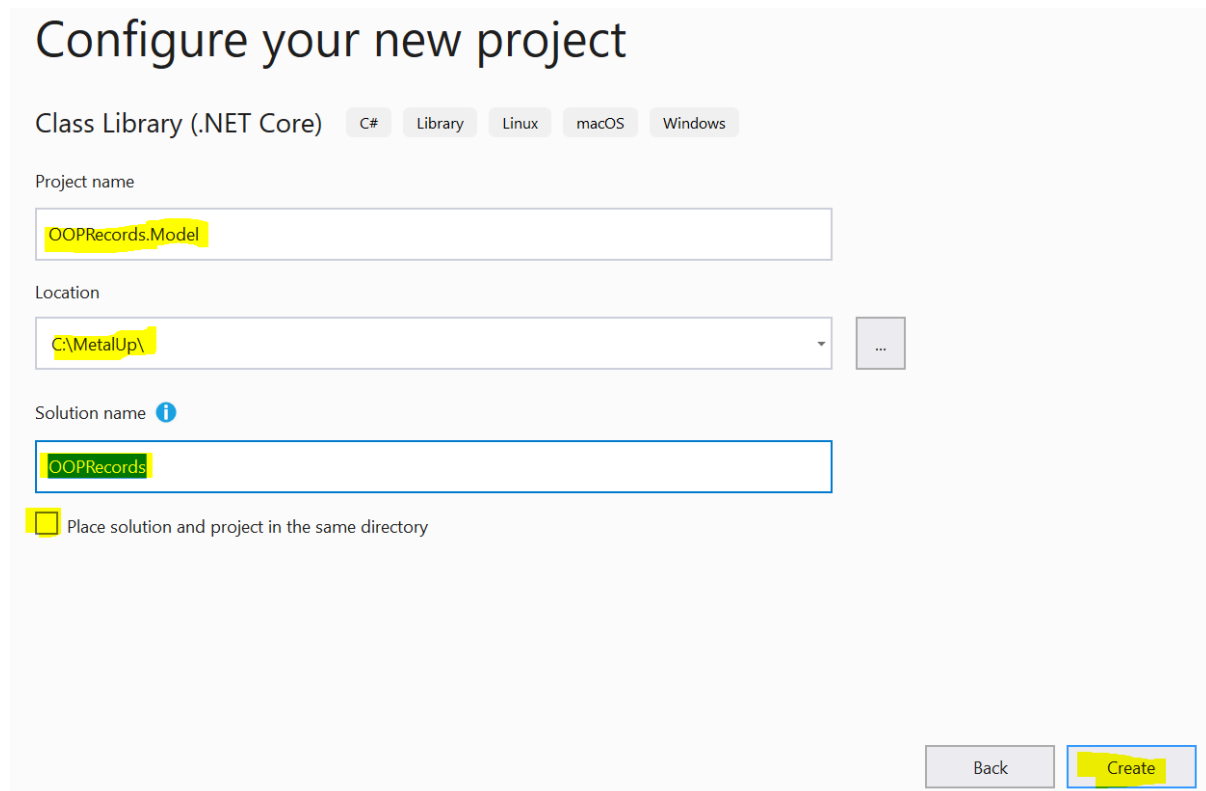
The application we shall be developing together is called OOPRecords, applied to school administration. We'll start with just a single class `Student` and working only in-memory – all data being lost when you quit the application. Then we'll add a very simple form of 'persistence' that allows the state of the objects that have been created or modified, to be saved to a text file, and read back in. But to build a realistic school management system – one that could support many users simultaneously – we need to be using database technology. After a brief review of some of the options for using OOP with databases, we will use one of the most popular options for commercial systems development today, an 'object-relational mapper' or ORM – specifically Microsoft's Entity Framework.

We'll learn, very briefly, about one of the most advanced concepts in OOP known as 'reflection' whereby a program can 'reflect' or 'introspect' on one of its own objects to find out what its capabilities, and how Entity Framework uses this to decide how an object model should be mapped onto a database schema.

Finally, we will introduce another technology – known as 'Naked Objects' – which also uses reflection to build a user-interface, 100% automatically, from the same object model without any further coding. Not only does it generate a user interface that shows the data in the objects, and allow the user to navigate the associations between them; it also translates the methods on the objects, where appropriate, into behaviour that can be observed and/or invoked by the user. As well as discovering that this makes for a highly-productive programming environment, you will discover that the OOP becomes a very natural way to think about the business problem domain. And lest you think that this is just a 'toy' approach designed for simple problems: the hugely complex government system referred to above was built using exactly those techniques that you will have learned here.

Objects in memory

Start by creating a new project of type **Class Library (.NET Core)** (you might need to search for this template name) to hold the object classes that will form the core of the application: Student, Teacher, Subject, TeachingSet, SubjectReport, Test, Sport, and so forth. First name the *solution* **OOPRecords**, and then name the *project* **OOPRecords.Model**, as shown below.



Configure your new project

Class Library (.NET Core) C# Library Linux macOS Windows

Project name

OOPRecords.Model

Location

C:\MetalUp\

Solution name ⓘ

OOPRecords

Place solution and project in the same directory

Back Create

Notes:

- The reason for naming the solution **OOPRecords**, but the project **OOPRecords.Model**, is that the solution will ultimately hold more than one project.
- ‘Model’ commonly refers to the collection of core object classes, sometimes also called the ‘domain classes’, that represent the domain of the application – in this case a school.

In the **Model** project, add the first of our intended domain classes: **Student**:

```
using System;

namespace OOPRecords.Model
{
    public class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime DateOfBirth { get; set; }

        public int Age()
        {
            var today = DateTime.Today;
            int ageToday = today.Year - DateOfBirth.Year;

            if (today.Month < DateOfBirth.Month || (today.Month ==
DateOfBirth.Month && today.Day < DateOfBirth.Day))
                ageToday --;
            return ageToday;
        }

        public override string ToString()
        {
            return $"{FirstName} {LastName}, Age {Age()}";
        }
    }
}
```

Notes:

- The class *initially* defines three properties: two of type `string`, and one of type `DateTime`.
- `DateTime` is a good example of a ready-made class: it has properties such as `Day`, `Month`, and `Year`, but also a rich set of methods such as `AddDays`, as well as the standard 'operators' such as `+` and `-`, which may be thought of as methods accessed via a more succinct syntax.
- We have used these properties and operators within a method named `Age`, which calculates the `Student`'s current age in years.
- We have overridden the inherited `ToString` method (that all object types in C# have) to provide a convenient summary of the object's properties as a single string. (If you have not previously seen the C# syntax used in the body of this method – i.e. a string starting with `$` – look up 'C# string interpolation' on the web).

Exercise 36

What is a good reason or reasons – there are multiple - for encapsulating a method to calculate `Age` on the `Student` object?

Now we are going to add another class, named `StudentRepository`, which has responsibility for storing and retrieving Students. Why don't we just make these things the responsibility of the `Student` class? We can't, for example, make *retrieving* a `Student` instance the responsibility of an instance, because if we haven't already got a reference to an instance, we wouldn't be able to call a method on it. A possible solution to this would be to implement the retrieval method as a *class method* rather than an *instance method* (in C# a class method is signified by the keyword `static`).

However, the primary reason for moving these responsibilities outside the `Student` class altogether, and into a `StudentRepository`, is the principle of 'separation of [technical] concerns'. The implementation of the responsibilities for storing and retrieving students will change according to the 'persistence' (storage) technologies being used. In this book we are going to change the persistence technology at least twice. Yet the fundamental structure and behaviour of the `Student` objects won't be changing. The design principle of separation of concerns says that you should keep code concerned with specific technologies as separate as possible from code that is generic to the application logic. We'll use this principle more than once in this project.

Add the `StudentRepository` class shown below:

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

namespace OOPRecords.Model
{
    public class StudentRepository
    {
        private List<Student> Students = new List<Student>();

        public void Add(Student s)
        {
            Students.Add(s);
        }

        public IEnumerable<Student> AllStudents()
        {
            return Students;
        }

        public IEnumerable<Student> FindStudentByLastName(string lastName)
        {
            return from s in AllStudents()
                   where s.LastName.ToUpper().Contains(lastName.ToUpper())
                   select s;
        }

        public Student NewStudent(string firstName, string lastName, DateTime dob)
        {
            var s = new Student();
            s.FirstName = firstName;
            s.LastName = lastName;
            s.DateOfBirth = dob;
            Add(s);
            return s;
        }
    }
}
```

Notes:

- In this chapter, we are just going to hold the Students in memory (persistent storage will be introduced in the next chapter), so the repository implementation has a simple `List<Student>` that the other methods use directly, or indirectly.
- The method `FindStudentByLastName` uses LINQ, which you have already met in Part I (), but here we are using LINQ's 'query syntax'. Previously, we used LINQ's 'method syntax'. These are just two different forms of syntax expressing the same logic. The implementation of `FindStudentByLastName` using method syntax would be:
`return AllStudents().Where(s => s.LastName.ToUpper().Contains(lastName.ToUpper()));`

- The reason we have switched to query syntax here is because it more closely resembles the Structured Query Language (SQL) that you will have learned in connection with relational databases. The keywords `from`, `where`, and `select` appear in both – though they are composed in a different order. But there are two other differences between SQL and LINQ. The first is that LINQ is a fully integrated part of the C# language – so the query logic does not have to be written inside strings with quotation marks. The other difference, and by far the most important, is that LINQ queries are written in terms of the domain objects, not the internal representations used by the database (tables, columns etc) or other persistent storage mechanism. In the query, when you wrote `S`, you would have been presented with a pop-up list of the properties of a `Student` object.
- `IEnumerable` can be thought of as a generalised type of collection or data structure that can be ‘enumerated over’. It gives us the standard methods available on all such data structures without us having to know the specific type of the data structure (this will come in handy when the underlying implementation changes).

The final class we need to add is the `Initializer`, which will be responsible for creating the *initial* instances of `Student` (and later of other domain classes), so that we don’t have to begin every run of our evolving application by manually creating each instance from scratch.

This is another example of separation of concerns: when, how, and indeed, *whether* initial data needs to be created will vary according to the form of persistence that we will use. Add the following class:

```
using System;

namespace OOPRecords.Model
{
    public class Initializer
    {
        public void Seed(StudentRepository students)
        {
            var alg = NewStudent(students, "Alie", "Algol", "19/02/2004");
            var frt = NewStudent(students, "Forrest", "Fortran", "22/09/2003");
            var jav = NewStudent(students, "James", "Java", "24/03/2004");
            var cee = NewStudent(students, "Celia", "Cee-Sharp", "12/09/2003");
            var vee = NewStudent(students, "Veronica", "Vee-Bee", "05/09/2003");
            var sim = NewStudent(students, "Simon", "Simula", "31/07/2003");
            var typ = NewStudent(students, "Tilly", "TypeScript", "14/01/2003");
            var pyt = NewStudent(students, "Petra", "Python", "17/06/2003");
            var has = NewStudent(students, "Harry", "Haskell", "08/04/2003");
            var cob = NewStudent(students, "Corinie", "Cobol", "28/02/2003");

            private Student NewStudent(StudentRepository students, string firstName,
string lastName, string dob)
            {
                var s = new Student();
                s.FirstName = firstName;
                s.LastName = lastName;
                s.DateOfBirth = Convert.ToDateTime(dob);
                students.Add(s);
                return s;
            }
        }
    }
}
```

Notes:

- Seed is the term commonly used for initializing a program with data. The Seed method has the repository passed into it as a parameter, because when it creates new instances of Student it needs to tell the repository to add them.
- There is no real need to assign the newly created students to variables (e.g. `var alg = ...`) as those variables are not used, and you might even get a warning to this effect from Visual Studio. However, these variables may come in useful when we later enrich the model.
- NewStudent is a helper method (so that a new student can be created more succinctly).

Finally, add a constructor into the StudentRepository (near the top, we suggest, just to follow a convention) to call the Seed method on the Initializer:

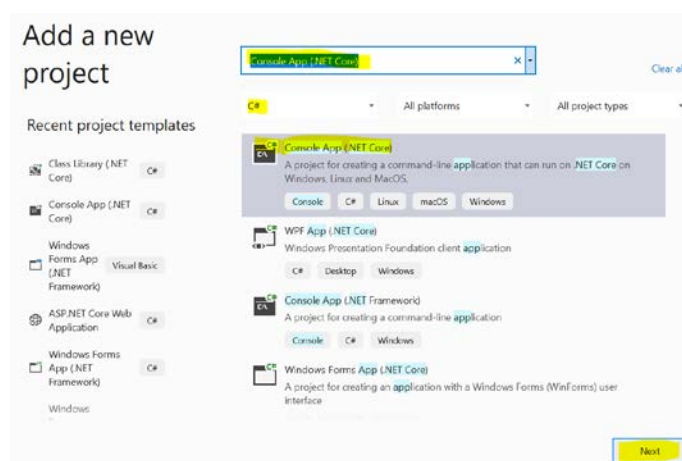
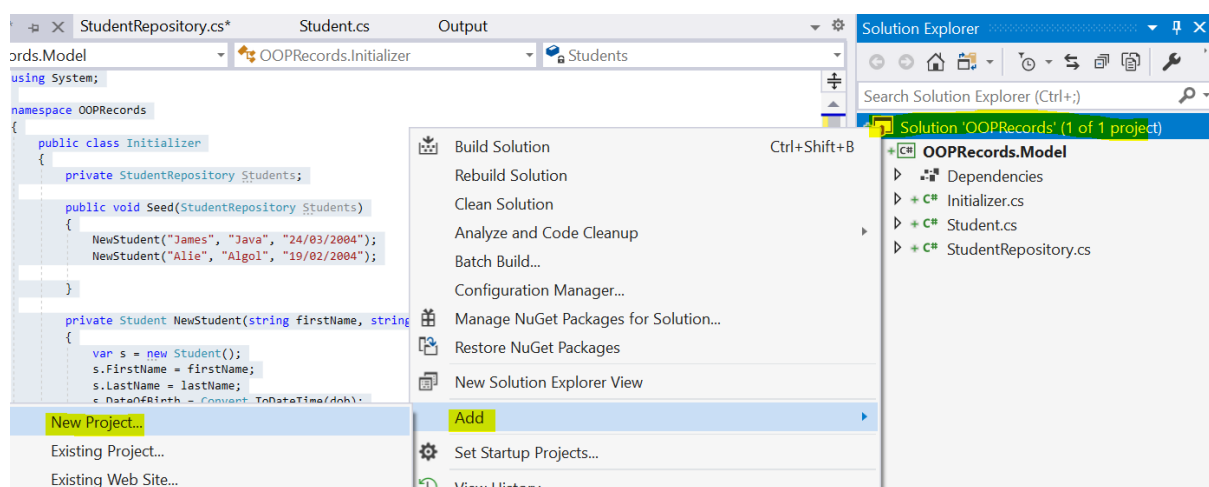
```
public StudentRepository()
{
    var initializer = new Initializer();
    initializer.Seed(this);
}
```

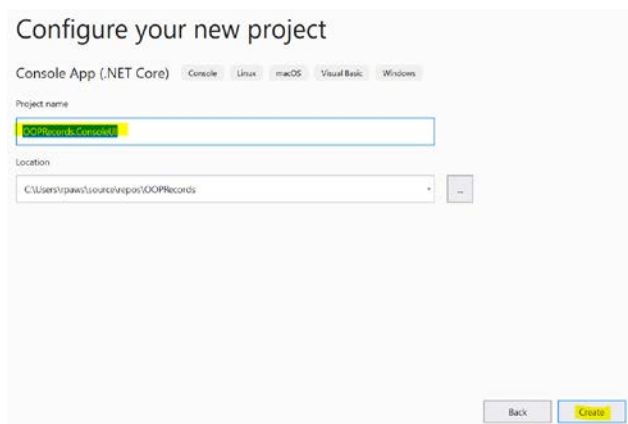
Adding a simple console user interface

To turn our very simple object model for a Student Records system into an application, we need a user interface. We will start with the simplest form of UI, a text-based or 'console' application. We will introduce this as a separate new project within the OOPRecords solution.

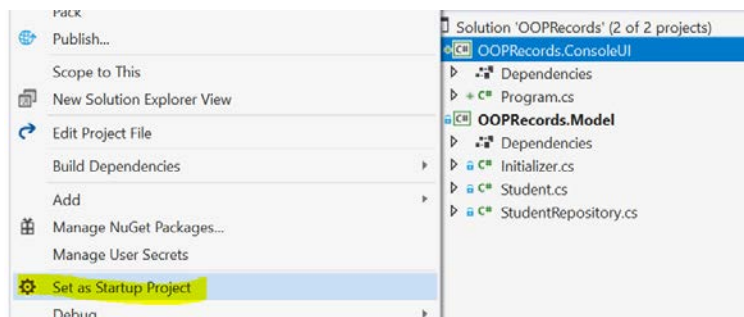
This is yet another example of the applying the principle of 'separation of concerns'. By keeping the knowledge of the user interface technology *out* of the domain object model, our intention is to facilitate the introduction of one or more alternative user interfaces at a future point *without having to change the Model project*.

Right-click on the **OOPRecords** solution, and select **Add > New Project**, of type **Console App (.NET Core)** and with the name **OOPRecords.ConsoleUI**:

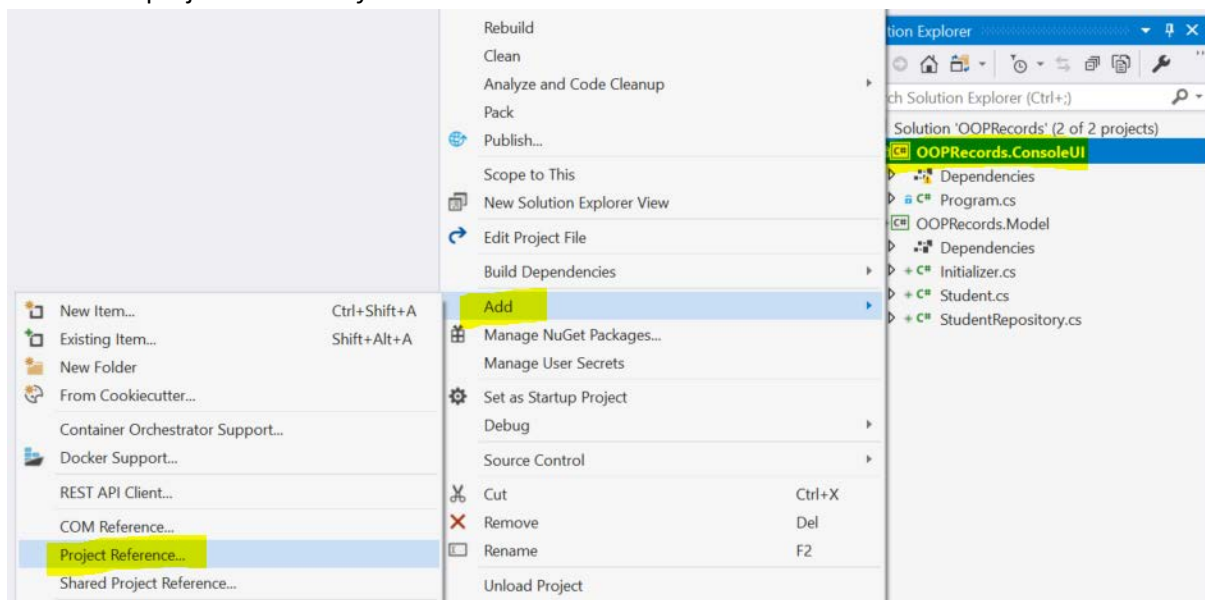


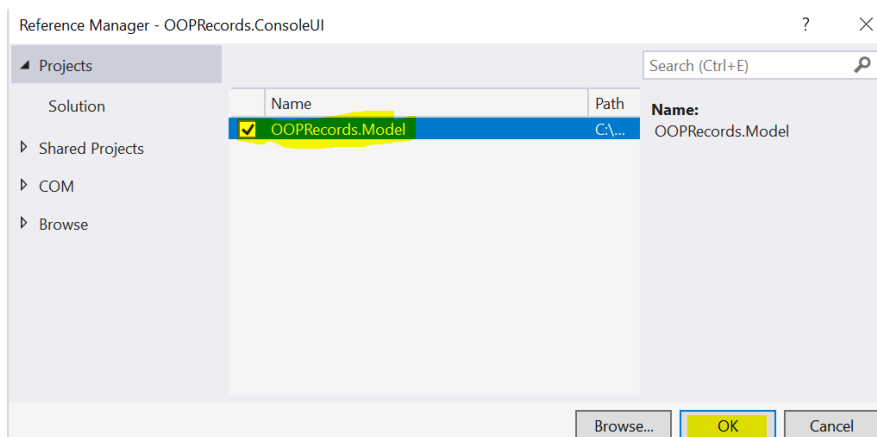


Right-click on the new project and **Set as Startup Project** (indicating that this is the project to be run):



Then add a project reference *from* **OOPRecords.ConsoleUI** to **OOPRecords.Model**:

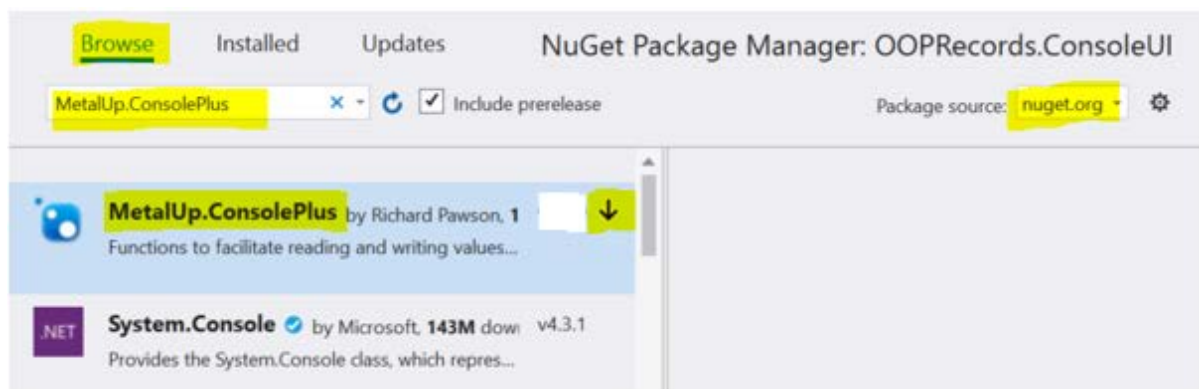




This means that the **ConsoleUI** project ‘knows about’ and is therefore ‘dependent upon’ the **Model** project, but *not vice versa*. If you were to try to add a reference *from* the **Model** project to the **ConsoleUI** you would encounter an error as this would be attempting to create a ‘circular reference’. This is a limitation of Visual Studio, but it is a *useful* limitation since it forces the programmer to think carefully about which project should depend on which and hence help achieve separation of concerns.

Finally we are going to add a package (library) to the new project, which offers some ready-made functions for common console interactions – because the objective of this book is to learn about OOP, not about the specifics of writing console code – which we will be abandoning within a couple of chapters, anyway.

Right-click on the new project and select **Manage NuGet Packages**, then **Browse** the NuGet Gallery (**nuget.org**) for the **MetalUp.ConsolePlus** package, downloading and installing the latest version by clicking the black arrow, **OK**ing the confirmation dialog:



Now replace the code in the **Program** file with this:

```
using MetalUp;
using OOPRecords.Model;
using System;

namespace OOPRecords.ConsoleUI
{
    class Program
    {
        static void Main()
        {
            var students = new StudentRepository();
            while (true)
            {
                Console.Clear();
                Console.WriteLine("Main Menu");
                Console.WriteLine("1. Create Student");
                Console.WriteLine("2. Find Student");
                Console.WriteLine("3. All Students");
                int selection = ConsolePlus.ReadInteger("Select option: ", 1, 3);
                Console.Clear();
                switch (selection)
                {
                    case 1: CreateStudent(students); break;
                    case 2: FindStudent(students); break;
                    case 3: AllStudents(students); break;
                }
                Console.WriteLine("Press any key to continue ...");
                Console.ReadKey();
            }
        }

        private static void AllStudents(StudentRepository students)
        {
            ConsolePlus.WriteList(students.AllStudents(), "\n");
        }

        private static void FindStudent(StudentRepository students)
        {
            Console.WriteLine("Find Student");
            string match = ConsolePlus.ReadString("Last name or part of last name: ", 1);
            ConsolePlus.WriteList(students.FindStudentByLastName(match), "\n");
        }

        private static void CreateStudent(StudentRepository students)
        {
            Console.WriteLine("Create Student");
            string firstName = ConsolePlus.ReadString("First Name: ", 1);
            string lastName = ConsolePlus.ReadString("Last Name: ", 1);
            DateTime dob = ConsolePlus.ReadDate("Date Of Birth: ", -10000, -1000);
            students.NewStudent(firstName, lastName, dob);
        }
    }
}
```

This code generates a simple menu-driven console application.

Exercise 37

Run the application. Invoke menu option 3.

Then use menu option 1 to create a new student.

Run option 3 again to show that your new student is now in the list.

1) Capture a screen snippet showing this.

Use option 2 to search for one or more students matching a *partial* surname of at least one letter.

2) Capture a screen snippet showing this interaction.

3) In the implementation of `FindStudentByName` on `StudentRepository`, why is the `ToUpper` method being called (twice)?

The big problem with even this minimal application is that the objects don't survive after the application is closed. (You can prove this for yourself by creating a new student, closing the application and re-running it – you'll be back to just the 'seed' objects.)

We need a mechanism to 'persist' (save) the objects, and we'll start with the simplest possible one: saving all the `Student` objects to a single file and then re-loading them all when the application is run.

Saving objects to a file

To save objects to a file we need to convert their 'state' – all the data held in their properties – into a standard text format. (The *methods* are not persisted – they will automatically become available when the object is brought back into memory as an instance of a class). The process for converting the state to text is known as 'serialization', and most OOP languages support some form of serialization. It may be used both for persisting objects, and also for transferring an object across a network (we'll see an example of this in a later chapter).

Add the following code into the `StudentRepository` replacing the highlighted part of the filename with the path to where your `OOPRecords` code is located.

```
private const string fileName =
@"C:\MetalUp\OOPRecords\OOPRecords.ConsoleUI\StudentsFile.json";

public void Load()
{
    using (StreamReader reader = new StreamReader(fileName))
    {
        string json = reader.ReadToEnd();
        Students = JsonSerializer.Deserialize<List<Student>>(json);
    }
}

public void SaveAll()
{
    using (StreamWriter writer = new StreamWriter(fileName))
    {
        var options = new JsonSerializerOptions { WriteIndented = true };
        string json = JsonSerializer.Serialize(Students, options);
        writer.Write(json);
        writer.Flush();
    }
}
```

Notes:

- You will need to add the necessary `using` statements to get the code to compile.
- Both the methods make use of the in-built `JsonSerializer`: `SaveAll` calls the `Serialize` method and `Load` calls the `Deserialize` method. The name indicates that the text will be formatted as JSON, which we'll explain shortly.
- `@` is needed in front of the string containing the filename, because the latter includes the `\` symbol, representing a directory folder. `@` tells the compiler to treat the `\` symbols literally – not to read them as 'escape sequence' (the escape sequence `\n`, for example would mean 'newline').

Now alter the constructor on `StudentRepository` adding the new code highlighted below:


```

public StudentRepository()
{
    if (File.Exists(fileName))
    {
        Load();
    }
    else
    {
        var initializer = new Initializer();
        initializer.Seed(this);
        SaveAll();
    }
}

```

We should also call the `SaveAll` method at the end of any repository method that adds, or updates (we haven't covered that behaviour yet), any `Student` object(s).

```

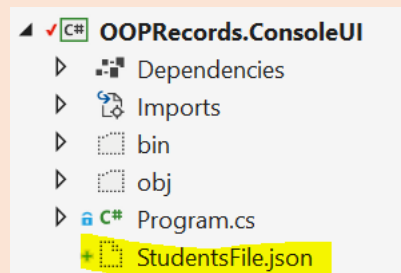
public Student NewStudent(string firstName, string lastName, DateTime dob)
{
    var s = new Student();
    s.FirstName = firstName;
    s.LastName = lastName;
    s.DateOfBirth = dob;
    Add(s);
    SaveAll();
    return s;
}

```

Exercise 38

Run the program. Create a new student (as you did in the previous chapter, but which will since have been lost). List **All Students** to confirm that it has been added.

In the **Solution Explorer**, click the **Show All Files** icon (highlighted below) and then open **StudentsFile.json**.



Capture a partial screen snippet showing just a few lines from this file include your new `Student` object as well as one other seeded by the `Initializer`.

Close the application and run it again, to confirm that the last `Student` you added has been reloaded into memory.

Introducing JSON

JSON is a standard for representing structured data. It is an alternative to XML, which we *could* have chosen to use as the format for storing our serialized objects. JSON is slightly more succinct than XML, and most people find it slightly easier to read, but it is important to remember that it isn't really intended to be read by humans: it is intended to be written and read by machines, whether for the purposes of persisting data or transmission over a network.

JSON stands for JavaScript Object Notation, and it works especially well with the JavaScript language. But its use is not restricted to JavaScript – we have successfully used it here for representing the state of C# objects.

Limitations of file-based persistence

File-based persistence of objects is easy to implement but it is extremely limited. Each time you wish to save a change (when an object is added, deleted, or updated), *all* the objects must be written anew. For a small application this matters little, but if your school management, or other records-based application, handle many thousands of records, this would be a big overhead.

Also, we would need to keep regular back-ups of the file, in case the system failed while overwriting the previous data.

In our current design, there would be one file of Students, another file for persisting Teachers, Sets, Reports, Tests, Sports and so on. While this does ameliorate the first problem slightly, it does not facilitate storing relationships (association) between objects of different types, which is an important requirement of most records management applications. (This *can* be managed with file-based persistence, but it is awkward.)

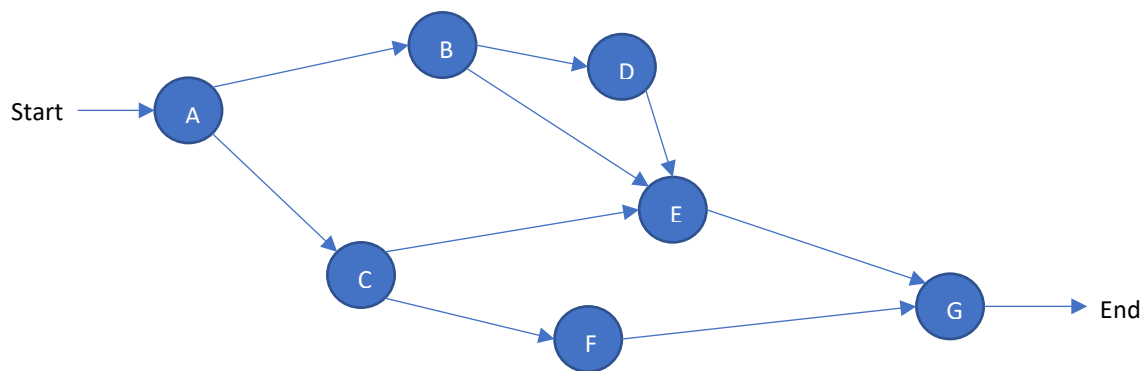
However, the biggest limitation of file-based persistence of objects is that it cannot readily support multiple parallel users, all potentially both accessing and modifying the objects.

The technology for addressing all these problems should be well-known to you: a database. The challenge is: how can we persist objects to a database?

Persisting objects to a database

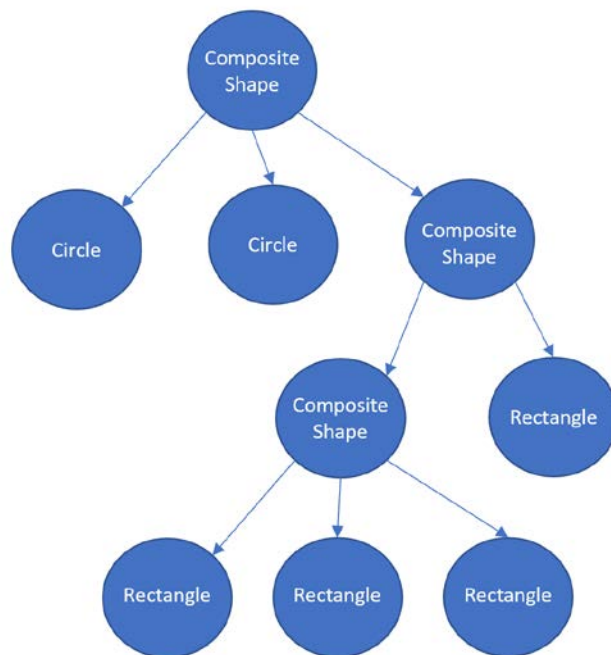
Object-oriented Database Management systems (OODBMS)

Early in the evolution of OOP, purpose-designed object-oriented database management systems (OODBMS) started to appear. As we have seen in the previous chapter, when you persist an object, all you are persisting is its 'state' (data) – not its methods, so how was an OODBMS fundamentally different to existing kinds of databases? The main difference lay in the nature of the relationships *between* objects. Objects can have associations to other objects; at a simple level, these are broadly equivalent to the relationships between tables in a relational database (RDBMS). But OOP makes it easy to construct complex 'graphs' (the term used in mathematics and in computer science for what we might call 'networks' in everyday language), as illustrated below:



An application that models the road, rail, or footpath, network of a whole country, or, in the case of GoogleMaps of all three types and for the whole world, relies on a vastly complex graph structure. The same goes for most 'Geographical Information Systems' (GIS), that manage the physical assets of energy or water utilities, or of local government. But graph structures are not limited to geographical representations...

When you developed OOPDraw in Part I, you added a `CompositeShape`, which was made up of other `Shapes`, each of which might be a `Rectangle`, `Circle`, ..., or another `CompositeShape`. So the graph representing a single drawing might look something like this:

**Notes:**

- The diagram above is *not* a class diagram, it represents the relationship between object instances *in a specific drawing* - a drawing of a vehicle in this case.
- You might describe this diagram as being a 'tree' – remember that in computer science a tree is just a graph with certain constraints added.

Computer Aided Design (CAD) systems follow broadly the same structure as OOPDraw, though there are many more classes of objects and those classes have much richer behaviour. Modern CAD systems can handle the design of a complete passenger aircraft - over 4 million components in the case of an Airbus A380 – where the object representing each component must model the whole shape (meaning that it may itself be a graph of shape objects), as well as the mass, structural strength, mechanical operation, heat dissipation, and more properties and behaviours.

In theory the structure of an all-world road map, or the components in an Airbus A380, could be persisted on a relational database. The problem is that it would be very slow to navigate, following *each* link of a randomly-connected graph: in some circumstances, following each link would require a separate query to the database. An OODBMS, however, typically allows a graph to be navigated across an unlimited number of links as a single query.

Today, the term OODBMS has largely been superseded by the term 'No SQL database', often associated with 'big data' applications, but the principle is similar.

Persisting objects on a relational database

However, focussing on the most complex applications, misses the fact that a huge proportion of applications, especially business systems, fit very well with relational databases, one reason why the latter have been so successful for so many decades.

The point about these examples and, indeed, *most* business applications today, is that the data elements, and hence the classes, have very *regular* relationships: they seldom involve random interconnections or graph structures.

OOP was slow to make inroads into such applications, however. Although people had connected object-oriented application code to relational database as early as the 1980s, the problem was that the amount of ‘glue’ code (between the objects in memory and their stored state in the database) that you had to write undid much of the advantage of writing the domain code as pure OOP.

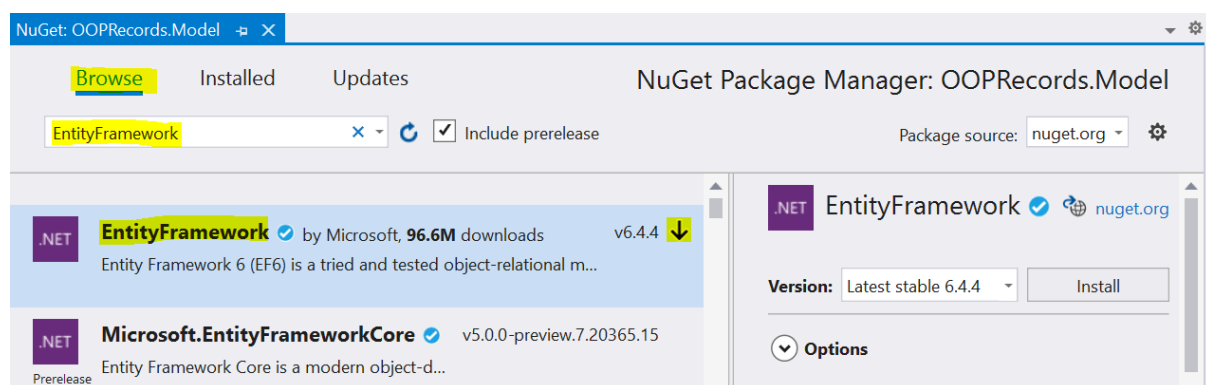
What was needed was to use the power of information technology itself to automate the ‘mapping’ from object classes to database tables, and thus eliminate all that glue code. Much of the progress in software design over the last 70 years has been through the development of more and more powerful ‘abstractions’ – allowing the programmer to focus more directly on the logic of the problem domain rather than on the mechanics of the computer operation.

The specific abstraction that fixed the above problem became known as an Object-Relational Mapper, or ORM. Early ORMs were better than writing glue code by hand, but still awkward. Today, ORMs have become so powerful that it is possible to persist many kinds of object model onto a relational database without even being aware of the that database.

We are now going to use just such a powerful ORM: Microsoft’s Entity Framework.

Using Entity Framework

Start by installing Entity Framework into the **OOPRecords.Model** project, via the **NuGet Package Manager**:



Note: Make sure that you install **EntityFramework**, *not* **EntityFrameworkCore**. (The latter is a more recent development, which may eventually displace the current Entity Framework, but still has some limitations at present.)

To use Entity Framework (‘EF’ henceforward), you need to define a ‘database context’, which inherits from the `DbContext` class that is a part of the EF library (`System.Data.Entity`). Add the following class, `DatabaseContext` into the Model:

```

using System.Data.Entity;

namespace OOPRecords.Model
{
    public class DatabaseContext : DbContext
    {
        public DatabaseContext(string dbName) : base(dbName)
        {
            Database.SetInitializer(new Initializer());
        }

        public DbSet<Student> Students { get; set; }
    }
}

```

Notes:

- The new class will not initially compile because we need to alter the `Initializer` also.
- The new class defines a property named `Students`, of type `DbSet<Student>`. As we add further classes – such as `Teacher`, `Subject`, `Report` – we will need to add an equivalent property for each type.
- The database context may optionally be set up with an initializer, somewhat like the one we have been using so far, but you will get a compile error because the current `Initializer` class is not (yet) fully compatible with EF.

Now make the following changes to the existing `Initializer` class:

```

using System;
using System.Data.Entity;

namespace OOPRecords.Model
{
    public class Initializer : DropCreateDatabaseIfModelChanges<DatabaseContext>
    {
        protected override void Seed(DatabaseContext context)
        {
            var students = context.Students;
            var aa = NewStudent(students, "Alie", "Algol", "19/02/2004", "HM287");
            ...
        }

        private Student NewStudent(DbSet<Student> students, string firstName,
            string lastName, string dob)
        {
            ...
        }
    }
}

```

Notes:

- Our `Initializer` class now inherits from one of several standard initializers in the EF library.
- The specific one we have chosen, `DropCreateDatabaseIfModelChanges`, will initialize the database when the application is first run, but, thereafter only when the object model changes i.e. when we add a new class or modify the properties/associations of an existing class. Otherwise the database will be left in the state it was previously.
- When such a change is made to the model the existing database will be 'dropped' (deleted) a new database created, and the initial data seeded into the new database. The reason why the database must be dropped and re-created is that the object class structure has changed, *so the database schema will need to change too*. This does mean, though that every such change will mean losing other changes we have made to the data.
- There is a more sophisticated way to use EF, known as 'data migration' which, in the event of a model change, *migrate* existing data from the old database schema automatically (where possible). This is useful in a commercial context when modifying applications that are already running with live data. However, this is technique is more advanced, and involves more programming, than we need at the early stages of designing our student records system.
- Another option would be for the `Initializer` to inherit from `DropCreateDatabaseAlways`, but this will initialize the database with the seed data *each run*, whether or not the model has changed (sometimes this can be useful during development).

We must also make some changes to the `StudentRepository`. Start by deleting the `Load` and `SaveAll` methods, and the calls to each of those methods from elsewhere in the code, together with the `file` property that they made use of. Then change the top of code in the `StudentRepository` as shown below to:

```
public class StudentRepository
{
    private List<Student> Students = new List<Student>();
    private DatabaseContext Context;

    public StudentRepository(DatabaseContext context)
    {
        Replace the whole body of the constructor with just this line:
        Context = context;
    }

    public void Add(Student s)
    {
        Context.Students.Add(s);
        Context.SaveChanges();
    }

    public IEnumerable<Student> AllStudents()
    {
        return Context.Students;
    }
    ...
}
```

Notes:

- You may remove two using statements that are now redundant, and which will already have been greyed-out by Visual Studio.
- The StudentRepository now uses the DatabaseContext in place of the in-memory list of students.
- When new objects are added (or existing objects changed) it is necessary to call SaveChanges on the context to tell it to update the database. (This is not necessary in the Initializer because Entity Framework automatically calls SaveChanges itself after the Seed method has been called).

As the StudentRepository now requires an instance of the DatabaseContext, we must set this up in the Program (in the ConsoleUI project):

```
static void Main()
{
    var context = new DatabaseContext("OOPRecords");
    var students = new StudentRepository(context);

    while (true)
    ...
}
```

Notes:

- The string OOPRecords that we are passing into the constructor for DatabaseContext is simply the name we want to give to the database that will be created (we could have chosen anything).

The only change that we need to make to `Student`, and any other domain class that will be persisted, is to add an `Id` property, that can be used as the ‘primary key’ in the database. Primary keys don’t have to be integers, but those are the simplest form to use. Add this property at the top of the `Student` class:

```
public int Id { get; set; }
```

Notes:

- We don’t have to set up a value for this property in our code (the database will automatically set the value) or reference it at all within our code. The `Id` field is just there to associate a given object instance with a specific row in a table
- Neither the `StudentRepository` nor the `Initializer` classes need an `Id` property, because they are *not* persisted in the database. These are both considered to be ‘stateless’ classes so there is nothing to persist anyway.

Exercise 39

Run the project. Verify that it starts with the expected list of students.

If you encounter any errors, refer to [Technical pre-requisites](#) and/or to [Troubleshooting](#).

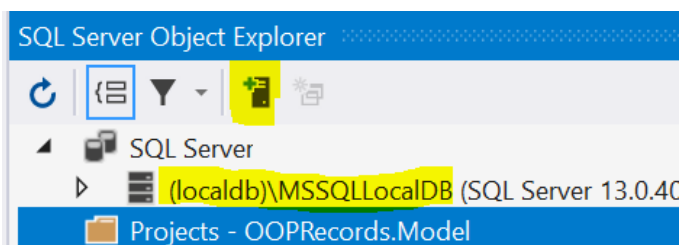
Add a new student. Close the console program and re-run it, verifying that the new student you added is still there.

Believe it or not, you have successfully created a relational database representing your simple object model, and connected the two together. Yet you have not used any database management application, written any SQL, or specified any schema! How do we even know that there *is* a database?

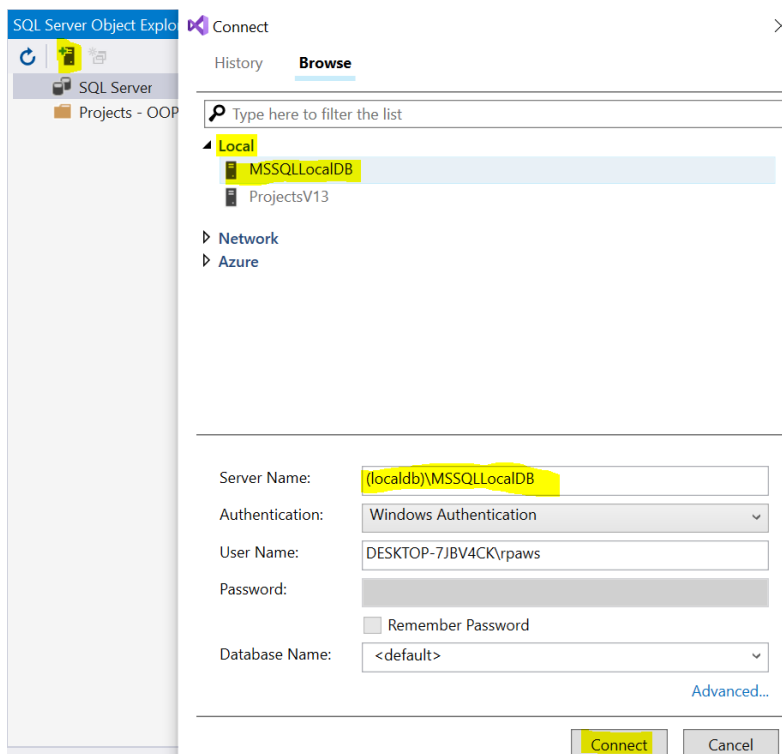
Viewing the created database directly

From Visual Studio’s main menu select **View > SQL Server Object Explorer**.

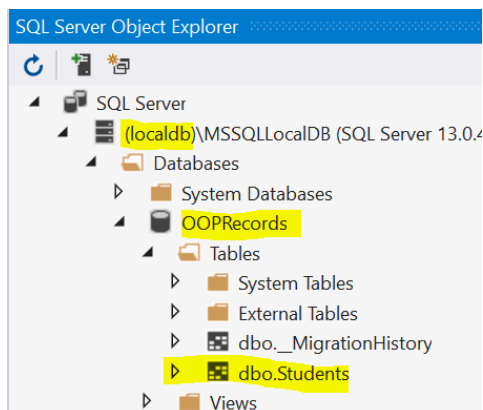
If, when expanded, the SQL Server icon does not show an entry for `(localdb)\MSSQLLocalDB` as shown below:



then select the **Add SQL Server** icon (highlighted above) and in the dialog, connect to **Local > MSSQLLocalDB** as shown below:



Back in the **SQL Server Object Explorer** expand the view of the **localdb** server to reveal the **OOPRecords** database (the name we passed into the constructor for **DatabaseContext**, remember), and the table named **Students** (EF automatically took the name of the class **Student** and pluralized it to use as the table name):



Exercise 40

Expand the **dbo.Students** table to show the table's columns.

1) Capture a screen snippet showing the column names.

Then right-click on the **dbo.Students** icon and select **View Data**.

2) Capture a screen snippet showing the data in the **Students** table.

Edit the bottom line of the table (currently containing NULL values) adding the two name fields and a date of birth for an additional student named **Barry Basic**. Note: enter the date of birth in the same format that you see for the existing records in the database – though you don't need to add the time on the end.

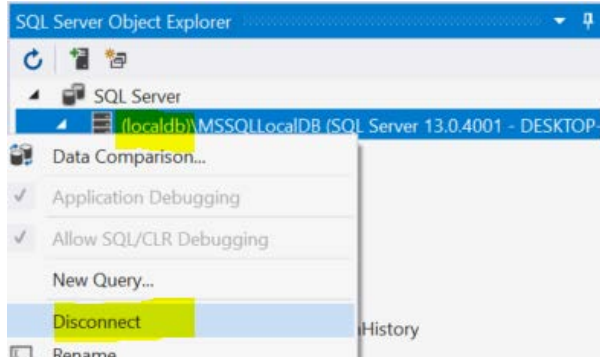
Go back to the console window (or, if you had stopped the program, run it again). Select the menu option to show All Students and verify that **Barry Basic** has now appeared.

3) Capture a screen snippet of the console display showing **Barry Basic**.

Extending the model

We are now going to expand the object model, but first:

Very important: Close any tabs in Visual Studio that are showing database information, and also right-click on the **localdb** server and select **Disconnect**:



If you forget to disconnect and/or have tabs showing database views then – in the next exercise – you will see an error message. (See [Troubleshooting](#)).

Stop the program (if it is still running) and in the **OOPRecords.Model** project add a new class, **Teacher**:

```
using System.Collections.Generic;

namespace OOPRecords.Model
{
    public class Teacher
    {
        public int Id { get; set; }

        public string Title { get; set; }

        public string LastName { get; set; }

        public string JobTitle { get; set; }

        public ICollection<Student> Tutees { get; set; } = new List<Student>();

        public override string ToString()
        {
            return $"{Title} {LastName}, {JobTitle}";
        }
    }
}
```

Notes:

- As well as an integer `Id`, and three string properties, `Teacher` has a `ICollection<Student>` named `Tutees`, initialized as a new `List<Student>`.

Now modify the `Student` class adding the following property, which represents an association to a single `Teacher` in the role of `Tutor` for that student:

```
public Teacher Tutor { get; set; }
```

Add a `DbSet` for the teachers into the `DbContext`:

```
public DbSet<Teacher> Teachers { get; set; }
```

And modify the `Initializer` to add some initial data for teachers, and to specify the personal tutor for *some of* the students:

```

public class Initializer : DropCreateDatabaseIfModelChanges<DatabaseContext>
{
    protected override void Seed(DatabaseContext context)
    {
        ...
        var teachers = context.Teachers;
        var dec = NewTeacher(teachers, "Mr.", "Deckerd");
        var tyr = NewTeacher(teachers, "Dr.", "Tynell");
        var maj = NewTeacher(teachers, "Maj.", "Major");
        var dou = NewTeacher(teachers, "Mrs.", "Doubtfire");
        var doo = NewTeacher(teachers, "Dr.", "Doolittle");
        var str = NewTeacher(teachers, "Dr.", "Strangelove");
        var iss = NewTeacher(teachers, "Ms.", "Issippi");
        var and = NewTeacher(teachers, "Ms.", "Andrist");
        var jek = NewTeacher(teachers, "Dr.", "Jekyll");
        var hyd = NewTeacher(teachers, "Mr.", "Hyde");
        var rob = NewTeacher(teachers, "Mrs.", "Robinson");
        var wor = NewTeacher(teachers, "Mrs.", "Worthington");
        var hu = NewTeacher(teachers, "Dr.", "Hu");
        var ove = NewTeacher(teachers, "Cpt.", "Over");

        alg.Tutor = dec;
        frt.Tutor = tyr;
        jav.Tutor = maj;
    }
    ...
    private Teacher NewTeacher(DbSet<Teacher> teachers, string title, string
lastName)
    {
        var t = new Teacher();
        t.Title = title;
        t.LastName = lastName;
        teachers.Add(t);
        return t;
    }
}

```

Notes:

- We have, deliberately, not specified a tutor for all students. This is to demonstrate that this association is *optional* not mandatory.
- We have not explicitly specified the tutees for each teacher, but, as we shall see, shortly, the association is inferred from the fact that we have specified the tutor for *some of* the students.
- In other words, Tutor and Tutees represent two ends of a 'one-to-many' association that is 'bi-directionally navigable' – see [Types of association](#).

Exercise 41

Referring, if needed, to [Class hierarchy](#) create a class diagram in Visual Studio showing just the **Student** and **Teacher** classes, but with the **Tutor** and **Tutees** properties shown as an association, and a collection association, respectively.

- 1) Capture a screen snippet of the class diagram.
- 2) What is the visual distinction between the two associations, apart from their names?

Run the application and then select the option to show **All Students**. You will not see anything different at this stage, but you need to run the application to re-create the database. If you get an error, refer to [Troubleshooting](#).

Exercise 42

Re-open a connection to the **localdb** (as before).

Expand the view of the database to show that it now has two tables, and expand each table to show the columns. Using your understanding of relational database, answer the following questions:

- 1) The **Students** table now has a column named **Tutor_Id**. In database terms, what is the nature of this field?
- 2) Why do you think that the **Teachers** table has nothing representing the tutees?

Right-click on the **Students** table icon and select **View Code** to see the table definition represented as SQL.

- 3) Capture a screen snippet showing the SQL definition for the **Students** table.
- 4) The SQL defining the **Id** column contains **NOT NULL**. Why?
- 5) The SQL defining the **Tutor_Id** column contains **NULL**. Why?

How does EF work?

EF is simple to use, but there is a great deal going on behind the scenes. The communication between Entity Framework and the database – both to create/update the database schema, and to execute queries at runtime – is all in the form of SQL statements, generated automatically and invisibly (it is possible to view the SQL if you really need to). In this sense, EF is doing something similar to what a compiler is doing when it translates your source code into the machine language that can be executed by the processor – but in this case is translating C# code into SQL.

To generate the necessary SQL, EF first has to identify the object classes that need to be persisted and their various properties and relationships (associations to other classes). This is achieved using ‘reflection’, which is a feature of most OOP languages, though some languages favour the term ‘introspection’ for the same capability.

In C#, for example, if you have a variable named `x` holding an object then

```
x.GetType()
```

will return you the *type* (class) for that instance. (Remember that in C#, `GetType` is one of the [four methods](#) that *every* type of object has.)

Type is itself an object, which can provide information about the type, starting with the fully-qualified type name, such as:

```
OOPRecords.Model.Student
```

The type can also provide a list of the properties and/or methods that an instance of that type has, for example:

```
x.GetType().GetProperties()
```

```
x.GetType().GetMethods()
```

We can continue to ‘drill-down’ like this to find out the name, and return type, of each property or method. All of this information *about* an instance is referred to as the ‘metadata’, meaning ‘data about data’, as distinct from the data that the object itself holds (such as the student’s name).

At run-time, EF accesses all this metadata about the object model and transforms it into a corresponding database schema. It will then compare this schema to the schema of the existing database (if any) and if the two do not match *in any respect* then EF will throw an exception or, if the initializer is set up to permit this (as it is in our code here) then it will drop and re-create the database so that the schema matches the object model structure.

EF also relies heavily on the simple idea of ‘programming conventions’. For example, if the `Student` class defines a property either named `Id` or `StudentId`, and the type of that property is an acceptable type for a database key (for example, an integer or string) then this property is *assumed* by EF to be the key. Another convention we have already seen, is that, by default, the table name will be a pluralised version of the class name.

In a large, complex project it is sometimes necessary to override these conventions, and specify the mappings explicitly. For example there might be a reason, perhaps for historical compatibility, why you want the table holding student data to be named `StudentRecords`, or the column holding the `LastName` to be called `Surname`. For this book, however, we shall rely on the conventions alone – if you want to know more about overriding the automated mapping, see <https://www.entityframeworktutorial.net/code-first/configure-classes-in-code-first.aspx> or other online source.

Perhaps the most intriguing part about EF is the way that it makes use of LINQ. When you originally wrote this method on the `StudentRepository` class:

```
public IEnumerable<Student> FindStudentByLastName(string lastName)
{
    return from s in AllStudents()
           where s.LastName.ToUpper().Contains(lastName.ToUpper())
           select s;
}
```

The objects provided by `AllStudents()` were all already in memory, held in a `List<Student>`. Yet we are now using the same method to search students *on the database* -

without any alteration. You might assume that the first thing that method now does is to read *all* the Students into memory and then search them: but it does *not* do this. That would be fine if the application dealt with a handful of students, but it would be horribly inefficient when querying all the students in a school/university, let alone all the student registrations held by an exam board. In fact, EF translates this LINQ query into a SQL query, at run-time, and then delegates that query to the database, translating only the table rows *returned by the query* back into Student objects in memory. How does it know that it should now do this instead of searching objects in memory?

The answer is that that the collection of objects returned by AllStudents, as well as being an IEnumerable<Student>, like all collections, is now *also* an IQueryable<Student> and IQueryable means, in simple terms, that it is capable of executing *lazily* – it fetches students only when a subsequent query is applied to it, and then only the results, not all up front.

Prove this by changing the return type to IQueryable<Student> on two of the methods in StudentRepository as shown below:

```
public IQueryable<Student> AllStudents()
{
    return Students;
}

public IQueryable<Student> FindStudentByLastName(string lastName)
{
    return from s in AllStudents()
           where s.LastName.ToUpper().Contains(lastName.ToUpper())
           select s;
}
```

If you had attempted this change back when we were dealing with objects in memory, or from persisted in a file, you would have received a compile error, because a simple List<Student> cannot execute queries lazily. By successfully changing the code as above, we have proved that we are in fact now dealing with IQueryable types, but this change will also prove useful for the next chapter *so leave the changes in place*.

Updating the user interface

Although we have seen that extending our model to include Teacher objects and the Tutor/Tutees relationship has generated a new database schema, for the user to be able to make use of this we would need to make multiple changes to the ConsoleUI. However...

- Even this small change will mean new menu options, new sub-menus, and changes to several existing options.
- None of these code changes will be difficult, but there will be a lot of code to write. This takes considerable time, and hampers the process of extending the object model.
- It also introduces lots of scope for errors, large and small, and hence increases the burden of testing the system with each iteration.
- The most subtle problem is that writing lots of simple UI code increases the risk that application logic that *should* be encapsulated as methods on the domain objects, ends up being written into the UI code instead, which reduces its scope for re-use. To take one very

simple example: a sensible application rule is that a date of birth entered by a user should *never* be after today, and, for most schools, should be constrained to a specific allowed range of years. The *right* place to put that logic is on the `Student` object, not in the UI code, even though that is often where it ends up in many systems.

Most advances in software development provide more powerful ‘abstractions’ that allow the software designer to focus more attention on analysing the problem domain rather than on the specific technical approach of the computer system. This is most obvious in the evolution of higher-level programming languages, but it applies also to the invention of powerful software ‘frameworks’ such as EF.

So if EF can reflect/introspect over a domain model written in C# and generate a database schema from it 100% automatically, should it not be possible to do something similar for the user interface?

Introducing ... ‘Naked Objects’, a software framework that, in conjunction with EF, allows you to create a complete application just by writing domain classes, and encapsulating all the application logic that you need as methods on those classes, without any reference to either the database or the user interface, both of which are generated automatically from the domain code. In the next chapter we’ll get our tiny model working with Naked Objects and then you will be astonished at the ease with which you can then extend the model.

Introducing the Naked Objects framework

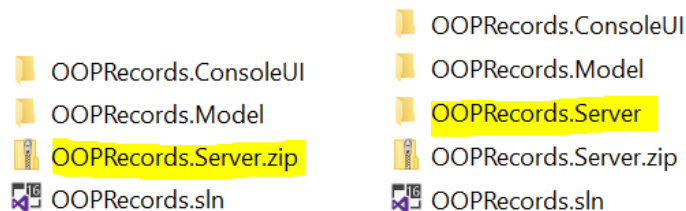
Naked Objects is an open-source framework that uses the power of reflection/introspection to create a sophisticated user-interface automatically from a domain model, keeping up with any changes to that model. It is compatible with EF. Naked Objects manages this in two stages:

- By wrapping the domain model and database in a web-server that transforms the domain model into a 'RESTful API', which may be accessed by any browser via HTTP.
- Using a generic 'Single Page (web) Application' (SPA) running in a browser, to transform the (JSON) representations of the objects delivered by the RESTful API into a fully operational user interface that allows objects to be viewed, and their methods to be invoked as actions.

While it is possible to use Naked Objects without any knowledge of these two separate transformations, we are deliberately going to make them more explicit by applying Naked Objects in two stages, and exploring the RESTful API between the two.

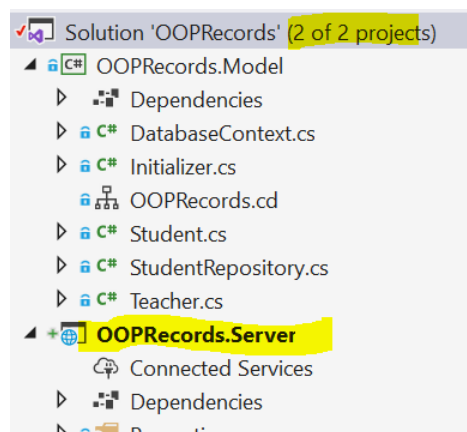
Creating a RESTful API from the object model

Download **OOPRecords.Server.zip** (see [Technical pre-requisites](#)) and move the zip file into your **OOPRecords** folder, as shown below left, then right-click on it and **Extract All ...** so that you end up with a new folder **OOPRecords.Server** at the same level as the other two project folders and alongside the **OOPRecords.sln** (below right):



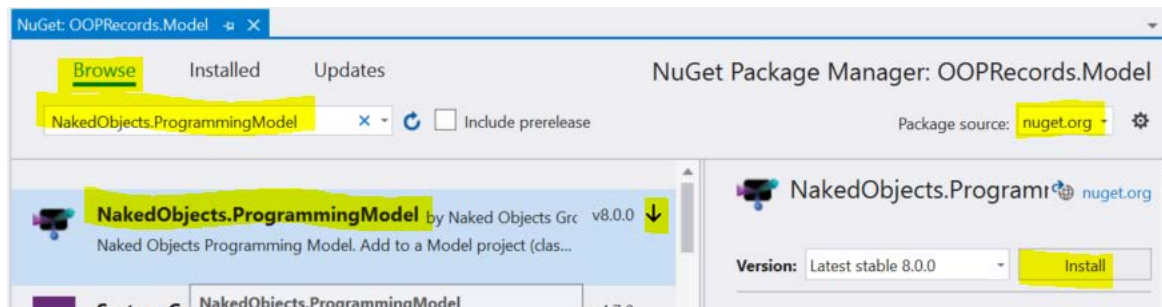
Back in the Visual Studio **Solution Explorer** right-click on the OOPRecords *solution* icon then **Add > Existing Project** and navigate to the new project file - **OOPRecords.Server.csproj**.

Set the new **Server** project as the **Start Up Project**, and then **Remove** the **OOPRecords.ConsoleUI** project so your solution then looks like this:



Right-click on **OOPRecords.Server** and **Add > Project Reference** to your **OOPRecords.Model** project (just as you had to do for the ConsoleUI project previously).

We must now make some *small* changes to the existing domain model to make it compatible with the new framework. Start by installing the latest version of the package **NakedObjects.ProgrammingModel** into the **OOPRecords.Model** project using the **NuGet Package Manager** as before:



The first change is that all properties on persisted objects must be marked **virtual**. (This is not a requirement imposed by Naked Objects: it is a requirement imposed by EF, for the specific way that Naked Objects uses EF).

Here are the changes shown for the class **Teacher**:

```
using System.Collections.Generic;

namespace OOPRecords.Model
{
    public class Teacher
    {
        public virtual int Id { get; set; }

        public virtual string FirstName { get; set; }

        public virtual string LastName { get; set; }

        public virtual string JobTitle { get; set; }

        public virtual ICollection<Student> Tutees { get; set; } = new
List<Student>();

        public override string ToString()
        {
            return $"{FirstName} {LastName}, {JobTitle}";
        }
    }
}
```

Note that there is no need to mark *methods* **virtual**, just the properties (including collections).

Make these changes, and then make equivalent changes to the **Student** class. *If you fail to do this*, then when you run you will get the second error message shown in [Troubleshooting](#).

Now some small changes to the StudentRepository:

```
using NakedObjects;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;

namespace OOPRecords.Model
{
    public class StudentRepository
    {
        private DatabaseContext Context;

        public StudentRepository(DatabaseContext context)
        {
            Context = context;
        }

        public void Add(Student s)
        {
            Context.Students.Add(s);
            Context.SaveChanges();
        }

        public IDomainObjectContainer Container { set; protected get; }

        public IQueryable<Student> AllStudents()
        {
            return Container.Instances<Student>();
        }

        public IQueryable<Student> FindStudentByLastName(string lastName)
        {
            return from s in AllStudents()
                   where s.LastName.ToUpper().Contains(lastName.ToUpper())
                   select s;
        }

        public Student NewStudent(string firstName, string lastName, DateTime dob)
        {
            var s = Container.NewTransientInstance<Student>();
            s.FirstName = firstName;
            s.LastName = lastName;
            s.DateOfBirth = dob;
            Add(s);
            Container.Persist(ref s);
            return s;
        }
    }
}
```

Notes:

- The repository methods no longer deal with the `DatabaseContext` directly. Instead they work through methods on the `IDomainObjectContainer` (referred to in the text as ‘the container’) a type defined in the `NakedObjects` library.
- Calling `Container.Instances<Student>` is equivalent to calling `context.Students`, but by going through the new container, the Naked Objects framework is made aware of the request and can make necessary interventions in the background.
- Similarly, `Container.NewTransientInstance<Student>()` is equivalent to writing `new Student()`, but, again, alerts Naked Objects that a new instance is being created that it needs to keep track of. ‘Transient’, here, means ‘not yet persisted’.
- When the new object, `s`, has been set up with the properties required, `Container.Persist(ref s)` tells the framework to persist that object in the database.
- How does the container get into the `Container` property of `StudentRepository` in the first place? This is done using ‘dependency injection’, a technique is now widely used in professional OOP development, not just with Naked Objects. The principle of dependency injection (sometimes also known as ‘Inversion of Control’ or IoC) is that an object should not have to go and *find* services that it needs, those services should be ‘injected’ into the object, when it is created, by the framework it is running within. It is another example of an abstraction that allows the programmer to focus on the external requirements of the application, and less on the internals of the computer system.

Now add (into the **OOPRecords.Model** project) a new class `AppConfig` as shown below:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;

namespace OOPRecords.Model
{
    public static class AppConfig
    {
        public static Type[] Services()
        {
            return new[] {typeof(StudentRepository)};
        }

        public static IDictionary<string, Type> MainMenus()
        {
            return new Dictionary<string, Type>()
            {
                ["Students"] = typeof(StudentRepository)
            };
        }

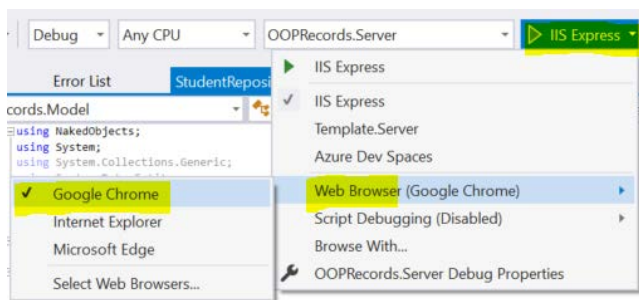
        public static DbContext CreateDbContext()
        {
            return new DatabaseContext("OOPRecords");
        }
    }
}
```

Notes:

- This class acts as a ‘bridge’ between the model and the server, providing information about the model that is needed by the server. The **Server** project has already been set up to look for file named **AppConfig** in the namespace **OOPRecords.Model**, and to access the three methods it defines.
- **Services** tells the Server project which of the classes in the model are to be registered as services (rather than as persisted domain classes).
- **MainMenus** specifies the names of user menus to be created – just **Students** initially – and the actions to be added to those menus (in this case, all the methods from the **StudentRepository**).
- **CreateDbContext** will be called by the Server project when it needs to create a database context.

Right-click on the solution icon and **Build Solution**. This might take a little while the first time, as the new project must download several NuGet packages in the background.

You should now be able to run the new server project. As this project is configured as a web-server, it will launch a browser. You have the option to specify which browser as shown below; we shall be using Chrome for our examples and recommend that you do the same, if possible:



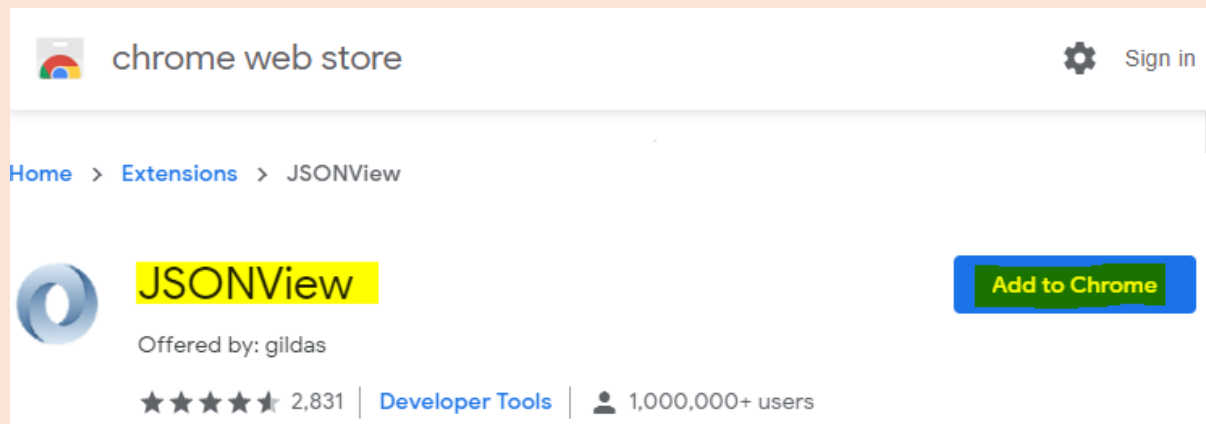
After a short delay, the browser should open on the url localhost:5000 (5000 is the 'port number'), and display something like this:

```
{
  "links": [
    {
      "rel": "self",
      "method": "GET",
      "type": "application/json",
      "profile": "urn:org.restfulobjects:repr-types/homepage",
      "charset": "utf-8",
      "href": "http://localhost:5000/"
    },
    {
      "rel": "urn:org.restfulobjects:rels/user",
      "method": "GET",
      "type": "application/json",
      "profile": "urn:org.restfulobjects:repr-types/user",
      "charset": "utf-8",
      "href": "http://localhost:5000/user"
    },
    {
      "rel": "urn:org.restfulobjects:rels/services",
      "method": "GET",
      "type": "application/json",
      "profile": "urn:org.restfulobjects:repr-types/list",
      "charset": "utf-8",
      "x-ro-element-type": "System.Object",
      "href": "http://localhost:5000/services"
    },
    {
      "rel": "urn:org.restfulobjects:rels/menus",
      "method": "GET",
      "type": "application/json",
      "profile": "urn:org.restfulobjects:repr-types/list",
      "charset": "utf-8",
      "x-ro-element-type": "System.Object",
      "href": "http://localhost:5000/menus"
    },
    {
      "rel": "urn:org.restfulobjects:rels/version",
      "method": "GET",
      "type": "application/json",
      "profile": "urn:org.restfulobjects:repr-types/version",
      "charset": "utf-8",
      "href": "http://localhost:5000/version"
    }
  ],
  "extensions": {}
}
```

This is a page of JSON, generated by the RESTful API, but unformatted, so it is not as easy to read as the JSON we saw in earlier (see [Introducing JSON](#)). Since this JSON will never be seen by a real user we need not worry, but since – for the purpose of this book – we would like to be able to explore the RESTful API a little, it would be nice to have the JSON well formatted. To do this you need to install the free JSONView component into Chrome:

Exercise 43

Search the online **Chrome Web Store** for **JSONView** and install it:



With the server program running, go back to **localhost:5000** (clicking **Refresh** on it if necessary) so that you now see some nicely formatted JSON. You are now looking at the 'home' representation created by the RESTful API.

1) Capture a screen snippet of this home representation.

Notice that it contains several links. Follow the links indicated below, which will take you through several JSON representations. *Be careful to find and click on the exact link specified in each case – as pages may contain several links that differ only slightly* (some highlights have been added below to draw your attention to specific details in the URL):

<http://localhost:5000/menus>

<http://localhost:5000/menus/StudentRepository>

<http://localhost:5000/services/OOPRecords.Model.StudentRepository/actions/AllStudents/invoke>

<http://localhost:5000/objects/OOPRecords.Model.Student/8>

You are now viewing a JSON representation of one of the Student objects.

2) Capture a screen snippet of the JSON showing the top of the representation only, sufficient to show the student's name

Use the +- symbols (these are added by JSONView, they are not part of the JSON itself) to collapse the view so that you can see the list of the object's **members**.

Expand the **Tutor** member then follow these two links:

<http://localhost:5000/objects/OOPRecords.Model.Student/8/properties/Tutor>

<http://localhost:5000/objects/OOPRecords.Model.Teacher/1>

You should now be looking at a JSON representation of a Teacher object.

3) Capture a screen snippet showing the top part of the JSON for this teacher.

You've been exploring a complex RESTful API, created automatically from the domain object model. One of the features of a true RESTful API (many APIs are *described* as RESTful, but don't follow all the rules) is that it is possible to navigate to any resource by following links from the home resource. However, this does not mean you must navigate it that way: you can always just specify the URL for the object of interest directly.

Exercise 44

Copy these URLs and paste them into the browser's URL bar and press the **Enter** key, capturing a partial screenshot (showing just the first few lines of the returned JSON):

- 1) `http://localhost:5000/objects/OOPRecords.Model.Student/4`
- 2) `http://localhost:5000/objects/OOPRecords.Model.Teacher/2`

Resources include actions as well as objects. Try this one:

`http://localhost:5000/services/OOPRecords.Model.StudentRepository/actions/FindStudentByLastName`

- 3) The top line of the returned JSON representation gives the **Id** as `FindStudentByLastName`. But what information is the next field down telling us about this action. (Describe the information in words, don't paste a screen snippet.)

As we stressed before, the RESTful API is not a user interface – users are not expected to read JSON – but what you have hopefully realised is that the RESTful API contains sufficient information and functionality to allow us to create a user interface.

Adding a 'Single Page Application' client

The modern way to build user interfaces to applications intended for use via the web is to create a Single Page Application (SPA) – a rich user interface that is executed by the browser's JavaScript engine. We're now going to add a ready-made one.

In order to build and use this client you will need to have **Node.js** installed. (See [Technical pre-requisites](#))

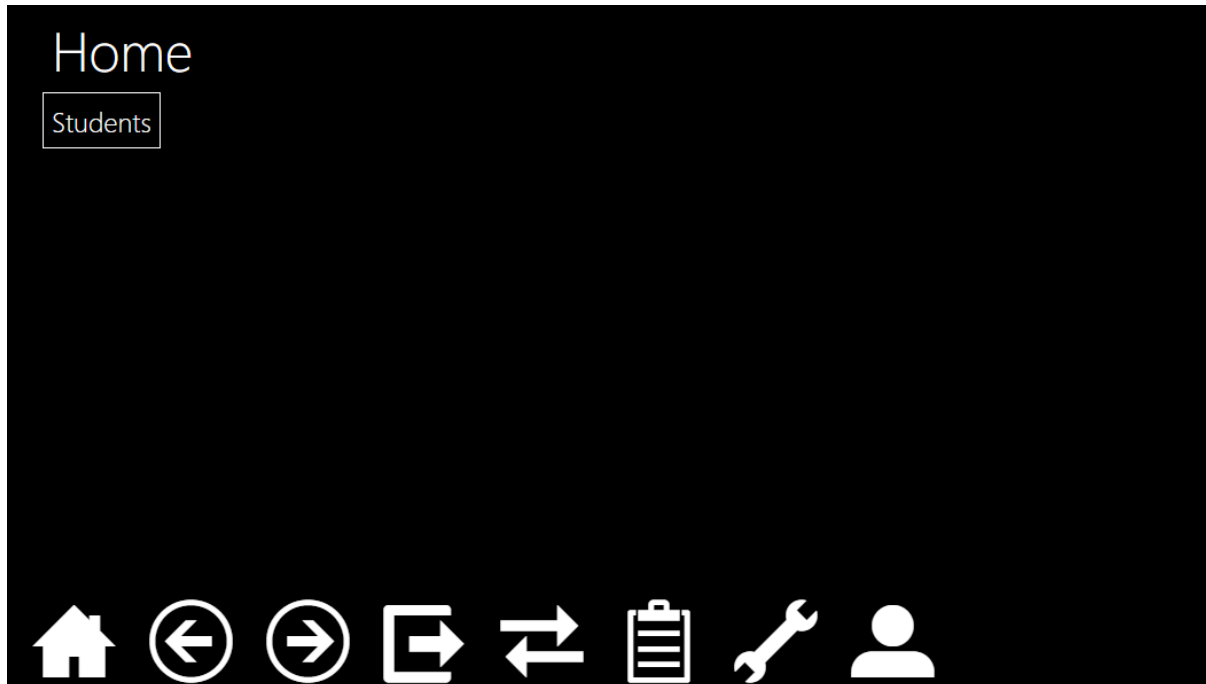
Locate the file **OOPRecords.Client.zip** (see [Technical pre-requisites](#)) and place it alongside where you placed the **OOPRecords.Server.zip**, and then **Extract All**.

Open the **OOPRecords.Client.csproj** file by double-clicking on it from within the **File Explorer**. This should open the client project in *a separate instance of* Visual Studio – so you should now have two instances of Visual Studio on your task bar, just as you might have two instances of Word viewing separate documents. While we are *developing* the application we need to be able to run the client and the server *separately* – even though they will be communicating with each other, and the easiest way to manage this is using two separate instances of Visual Studio. (If, or when, this application is eventually deployed for *live* use, the client and server will be deployed as two separate resources on the same web server).

Make sure that the **OOPRecords.Server** project is still running. (You can check this by accessing `localhost:5000` from the browser). Leaving this running, go to the **OOPRecords.Client** project

and run it. The *first time* you run it this may take some while to start as multiple packages need to be downloaded and installed.

Eventually, a separate browser instance will be launched on a *separate port* of the localhost web server (`localhost:5001`), and you will end up with a display like this:



Exercise 45

Click on the Students menu and then on **All Students**, which should return you a list of student objects. Click on one for an expanded view of that object.

Use the **Back icon** at the bottom of the screen (as with most SPA projects it is best to avoid using the browser's own back-button) to go back to the list of all students. This time right-click on one of the students.

1) What is the difference between left-clicking and right-clicking on an object in this view?

Explore what the other icons at the bottom of the screen do. Not all their roles will be clear to you, but several should be.

By whichever means, get back to a view of **Alie Algol** and notice that the **Tutor** property contains a link to the tutor **Harry Haskell** and that you can open a view of that object (by either left- or right-clicking).

Finally, go back to the **Students** menu (clicking **Home** first if necessary) and invoke **Find Student By Last Name**. Unlike All Students, which executed immediately, this action generates a dialog box.

2) Where has the client got the information necessary to create this dialog box?

Apart from the fact that we now have a nicer, more functional, user interface than we had before, the really nice feature of our set-up now is that is that we can extend our domain model, with new classes, properties and methods, and these new capabilities will be automatically result in

corresponding changes both to the RESTful API, and to the user interface without our having to write or edit any additional code (in most cases). We'll now prove that.

Enriching domain objects

For the remainder of this book we shall be modifying and extending the domain object model (**OOPRecords.Model**). Each time you will need to stop the **OOPDraw.Server** project and then run it again when the changes are made. However, there is *no need* to stop the **OOPRecords.Client** project, which is running in a separate instance of Visual Studio and displays in a separate browser instance or tab. This client will update automatically to reflect the new capabilities offered by the server (if it does not update immediately, just click the browser's **Refresh** icon).

Adding a new property

The first modification will be to add a new property into the **Student** class, add a new property, **StudentNumber** – of type **string** so that it may optionally include alphabetic characters as well as numeric digits.

```
public virtual string StudentNumber { get; set; }
```

and modify the **Initializer** class to make use of this new property as follows:

```
protected override void Seed(DatabaseContext context)
{
    var students = context.Students;
    var alg = NewStudent(students, "Alie", "Algol", "19/02/2004", "HM287");
    var frt = NewStudent(students, "Forrest", "Fortran", "22/09/2003", "LX046");
    var jav = NewStudent(students, "James", "Java", "24/03/2004", "HW531");
    var cee = NewStudent(students, "Celia", "Cee-Sharp", "12/09/2003", "LX033");
    var vee = NewStudent(students, "Veronica", "Vee-Bee", "05/09/2003", "HM119");
    var sim = NewStudent(students, "Simon", "Simula", "31/07/2003", "HW309");
    var typ = NewStudent(students, "Tilly", "TypeScript", "14/01/2003", "LX008");
    var pyt = NewStudent(students, "Petra", "Python", "17/06/2003", "LX144");
    var has = NewStudent(students, "Harry", "Haskell", "08/04/2003", "HM200");
    var cob = NewStudent(students, "Corinie", "Cobol", "28/02/2003", "HW442");
    ...
}

private Student NewStudent(DbSet<Student> students, string firstName, string
lastName, string dob, string number)
{
    var s = new Student();
    s.FirstName = firstName;
    s.LastName = lastName;
    s.DateOfBirth = Convert.ToDateTime(dob);
    s.StudentNumber = number;
    students.Add(s);
    return s;
}
```

Exercise 46

Run the server again, and on the client invoke the menu action to show **All Students**.

Right-click on **Alie Algol** to show a detailed view on the right-hand side of the display, which should now show the **Student Number**.

Back in the left-hand side, click on the small icon above the list of students (it has the tooltip **View As Table**).

1) Capture a screen snippet showing both halves of the display.

Click on the row for **Simon Simula**, who, currently, does not have a tutor, and then click the **Edit** action above.

From the right-hand side, *drag* the link to the teacher **Mr. Deckerd** from the view of **Alie Algol** into the **Tutor** field of **Simon Simula**.

Change Simon's Date Of Birth to tomorrow's date, using the calendar helper to the right of the field (you can go to Today and thence to tomorrow's date). Then click **Save**.

2) Capture a screen snippet showing both the students.

3) Notice that the 'title' for Simon (at the top of the view) has now changed, reflecting his new (albeit improbable!) age. Where in the code has this title been picked up from? (hint: look at the **Student** class)

4) It might be tempting to say that Simon now has a 'copy' of the Mr. Deckerd object, as his tutor, but this would not be correct terminology – as no *copy* has been made. What is the correct way to describe what has just happened in OOP terminology?

5) The properties are not laid out in the order in which they are specified in the code. In what order are they laid out?

One very important thing to notice is that we changed our domain model (to include Student Number) but we did not have to make any change to the Client code. This is the defining concept of the Naked Objects Framework – development of most web applications is far messier!

However, we can make several immediate criticisms of the application as it stands:

1. In the last exercise, you found that there was nothing to prevent us entering a future date of birth, which is clearly inappropriate. We need some sort of 'input validation' logic. It is tempting to code this logic in the user interface – and many developers do – but it is much better to encapsulate this functionality with the Student object that has the `DateOfBirth`. That way it can be re-used in any user interface built on top of the domain model.
2. It would be convenient if the user could look up teachers from a main menu.
3. Since the `Id` property is needed only for the database (to match up an object instance with a row in the table), it need not be displayed to the user.
4. The other properties of Student (and Teacher) are not laid out in a helpful order.
5. Although drag and drop allows us to specify the Tutor for a given Student, this relies on us having the Teacher available on screen. A more convenient mechanism would be to

implement 'auto-complete' on the Tutor field: start typing the teacher's name and then select a Teacher object from a drop-down list of matches.

We'll now fix each of these in turn.

Improving the user experience

1) Add `using NakedObjects;` at the top of the `Student` class, then add the following new method. Although it can be placed anywhere in the code, a recommended style is to place it just after the definition of the `DateOfBirth` property:

```
public string ValidateDateOfBirth(DateTime dob)
{
    return dob > DateTime.Today ? "Date of Birth cannot be after today" : null;
}
```

If this syntax used in the one-line *body* of the method new to you, it is just a more succinct way of writing:

```
public string ValidateDateOfBirth(DateTime dob)
{
    if (dob > DateTime.Today)
    {
        return "Date of Birth cannot be after today";
    }
    else
    {
        return null;
    }
}
```

2) Add a new class `TeacherRepository` to the **Model** project. Use the `StudentRepository` as a *guide*, but making changes as needed to fit the properties of a `Teacher` object.

Within the `AppConfig` file to register the new repository and create a main menu (named **Staff**) from its methods:

```
public static Type[] Services()
{
    return new[] {
        typeof(StudentRepository),
        typeof(TeacherRepository)
    };
}

public static IDictionary<string, Type> MainMenus()
{
    return new Dictionary<string, Type>()
    {
        ["Students"] = typeof(StudentRepository),
        ["Staff"] = typeof(TeacherRepository)
    };
}
```

3) Immediately above the `Id` property in *both* `Student` and `Teacher` add a `Hidden` attribute as shown here (add using `NakedObjects`; if needed):

```
[Hidden(WhenTo.Always)]
public virtual int Id { get; set; }
```

An attribute is not a piece of *executable* code: it is not ‘called’ like a function or method. Instead it provides additional information – the technical term is ‘meta-data’ – about the code that follows the attribute. The attribute(s) may be ‘read’ by other code before calling or using the executable code that follows the attribute(s). In this case the Naked Objects Framework reads the attributes and uses this to specify some change to the presentation, or the behaviour, of an object (or its properties or methods) at the user interface.

4) On the `Student` object, add a `MemberOrder` attribute above each of the properties (except `Id`, which will not now be displayed) as shown in this example:

```
[MemberOrder(1)]
public virtual string StudentNumber { get; set; }
```

You should change the number in brackets for each property *within the same object* to specify the order in which they should be displayed. Make the order: Student Number, First Name, Last Name, Date Of Birth and Tutor. Now do the same for `Teacher`, specifying the order of the (visible) fields as First Name, Last Name, Job Title, and Tutees.

Exercise 47

The numbers specified in the `MemberOrder` attributes within a class do not have to be contiguous (e.g. 1,2,3). Some programmers prefer to specify them as 10, 20, 30). Why do you think the latter be a good practice?

5) Add into the `Student` class (at the top, we recommend):

```
public TeacherRepository TeacherRepository { set; protected get; }
```

In the same way as we added code to inject an `IDomainObjectContainer` into both the `StudentRepository` and the `TeacherRepository`, the code above will result in a reference to the `TeacherRepository` to be injected into each instance of `Student`. We will now make use of this by adding the following new method into `Student`, just underneath the `Tutor` property (you will need to add a further `using` statement):

```
public IQueryable<Teacher> AutoCompleteTutor([MinLength(3)] string match)
{
    return TeacherRepository.FindTeacherByLastName(match);
}
```

Exercise 48

Make all the changes above, run the Server project again. Then from the browser viewing the Client, click **Refresh**. Confirm that:

- There is now a Staff menu, offering actions to create or retrieve Teachers.
- The Id property is no longer visible on either Student or Teacher
- The other properties appear in the intended order.

Edit a student specifying a date of birth that is after today, and click **Save**.

1) Capture a screen snippet showing the result, and then enter a valid date.

Edit the Student again. If the Tutor property already specifies a Teacher, click the X at the right of the field to clear the reference. Then start typing the last name of a Teacher. When the matching teacher appears underneath, click on it and **Save** the object.

2) What is the significance of the `[MinLength(3)]` in the code above. Why is this a good idea?

Notes:

- The `ValidateDateOfBirth` and `AutoCompleteTutor` methods are further example of the principle of 'programming by convention'. Naked Objects looks for the `Validate...` and `AutoComplete...` prefixes on member names. Provided that the rest of the name matches an existing member, and the types used also match, the framework will use that method to provide behaviour at the user interface.
- The `Hidden` and `MemberOrder` attributes, provided as part of the `NakedObjects.ProgrammingModel` library also modify the behaviour of the system. `MinLength` is an attribute provided by the standard .NET library, but recognised and interpreted by Naked Objects framework.
- There are many more attributes and method name conventions recognised by Naked Objects that allow you to add rich behaviours to your objects. We'll use just a few more of them in the next chapter, but if you are keen to enrich the application in your own ways, or start a brand new OOP project using this framework you'll need to read the Naked Objects Developer Manual, which you can download from here: <https://github.com/NakedObjectsGroup/NakedObjectsFramework/blob/master/Documentation/DeveloperManual.docx>.

Extending the model

Our Student Records application is still, frankly, quite trivial. But the same programming principles can be used to create large-scale, complex, commercial applications.

We are going to finish this part by adding three more domain entity classes: `Subject`, `TeachingSet` (meaning a group of pupils taught together for the same subject), and `SubjectReport`, with repositories for two of them. Both for the new classes, and the existing `Student` and `Teacher` classes, we are going to encapsulate more *behaviour* in the form of richer methods, several of which will show up on the **Actions** menu for an individual object.

The programming patterns being used are all fully documented in the Naked Objects Developer Manual, which may be downloaded from:

<https://github.com/NakedObjectsGroup/NakedObjectsFramework/blob/master/Documentation/DeveloperManual.docx>.

In the code modifications that follow, please note:

- For any given class, generally we have shown only existing lines that must be changed, or new lines that must be added. Any code not explicitly shown in the listing remains unchanged.
- New classes need to be added within the same namespace as existing domain objects i.e. within `OOPRecords.Model`.
- Required `using` statements have been omitted to save space here – by now you should be familiar with how to identify and fix any missing `using` statements.

We'll start by adding the three new domain entity classes: `Subject`, `TeachingSet`, and `SubjectReport`, with repositories for the first two. (We have no immediate need of a repository for `SubjectReport` because we will always be viewing, or creating, these objects *from within another domain object* – such as a `Student` - and so won't need a home menu called, say, **Reports**. However, you may add one later if you wish.)

```
public class Subject
{
    [Hidden(WhenTo.Always)]
    public virtual int Id { get; set; }

    [MemberOrder(1)]
    public virtual string Name { get; set; }

    public override string ToString()
    {
        return Name;
    }
}
```

```
public class SubjectRepository
{
    public IDomainObjectContainer Container { set; protected get; }

    public Subject CreateNewSubject()
    {
        return Container.NewTransientInstance<Subject>();
    }

    public IQueryable<Subject> AllSubjects()
    {
        return Container.Instances<Subject>();
    }

    public IQueryable<Subject> FindSubjectByName(string name)
    {
        return AllSubjects().Where(c =>
c.Name.ToUpper().Contains(name.ToUpper()));
    }
}
```

```
public class TeachingSet
{
    [NakedObjectsIgnore]
    public virtual int Id { get; set; }

    [MemberOrder(1)]
    public virtual string SetName { get; set; }

    [MemberOrder(2)]
    public virtual Subject Subject { get; set; }

    [MemberOrder(3), Range(9,13)]
    public virtual int YearGroup { get; set; }

    [MemberOrder(4)]
    public virtual Teacher Teacher { get; set; }

    [MemberOrder(5)]
    public virtual ICollection<Student> Students { get; set; } = new
List<Student>();

    public void AddStudentToSet(Student student)
    {
        Students.Add(student);
    }

    public void RemoveStudentFromSet(Student student)
    {
        Students.Remove(student);
    }

    public IList<Student> Choices0RemoveStudentFromSet()
    {
        return Students.ToList();
    }

    public override string ToString()
    {
        return SetName;
    }
}
```

Note that the Range attribute is defined in the namespace `System.ComponentModel.DataAnnotations`. You might need to add the using statement by hand as it is not always offered as a quick fix.

```
public class SetRepository
{
    public IDomainObjectContainer Container { set; protected get; }

    public TeachingSet CreateNewSet()
    {
        return Container.NewTransientInstance<TeachingSet>();
    }

    public IQueryable< TeachingSet > ListSets([Optionally] Subject subject,
                                             [Optionally] int? yearGroup)
    {
        var sets = Container.Instances<TeachingSet>();
        if (subject != null)
        {
            int id = subject.Id;
            sets = sets.Where(s => s.Subject.Id == id);
        }
        if (yearGroup != null)
        {
            sets = sets.Where(s => s.YearGroup == yearGroup.Value);
        }
        return sets.OrderBy(s => s.YearGroup).ThenBy(s => s.Subject.Name);
    }
}
```

```
public class SubjectReport
{
    public TeacherRepository TeacherRepository { set; protected get; }
    public SubjectRepository SubjectRepository { set; protected get; }

    [Hidden(WhenTo.Always)]
    public virtual int Id { get; set; }

    [MemberOrder(1)][Disabled]
    public virtual Student Student { get; set; }

    [MemberOrder(2)]
    public virtual Subject Subject { get; set; }

    public IQueryable<Subject> AutoCompleteSubject(string match)
    {
        return SubjectRepository.FindSubjectByName(match);
    }

    [MemberOrder(3)]
    public virtual string Grade { get; set; }

    public IList<string> ChoicesGrade()
    {
        return new List<string> { "A*", "A", "B", "C", "D", "E", "U" };
    }

    [MemberOrder(4)]
    public virtual Teacher GivenBy { get; set; }

    public IList<Teacher> ChoicesGivenBy()
    {
        return TeacherRepository.AllTeachers().ToList();
    }

    [MemberOrder(5)]
    [Mask("d")]
    public virtual DateTime Date { get; set; }

    public DateTime DefaultDate()
    {
        return DateTime.Today;
    }

    [MemberOrder(6)][MultiLine][Optionally]
    public virtual string Notes { get; set; }

    public override string ToString()
    {
        return $"{Subject}, {Date.Date}";
    }
}
```

Since these are all persisted objects, we should add corresponding DbSets into the DatabaseContext:

```
public class DatabaseContext : DbContext
{
    ...
    public DbSet<Student> Students { get; set; }
    public DbSet<Teacher> Teachers { get; set; }
    public DbSet<Subject> Subjects { get; set; }
    public DbSet<TeachingSet> Sets { get; set; }
    public DbSet<SubjectReport> SubjectReports { get; set; }
}
```

Within the AppConfig file, register the two new repositories, and specify that their methods are to appear in two new main menus:

```
public static Type[] Services()
{
    return new[] {
        typeof(StudentRepository),
        typeof(TeacherRepository),
        typeof(SubjectRepository),
        typeof(SetRepository)
    };
}

public static IDictionary<string, Type> MainMenus()
{
    return new Dictionary<string, Type>()
    {
        ["Students"] = typeof(StudentRepository),
        ["Staff"] = typeof(TeacherRepository),
        ["Subjects"] = typeof(SubjectRepository),
        ["Sets"] = typeof(SetRepository)
    };
}
```

Now we will enrich the existing Student and Teacher classes, to add relationships to the new objects, and also to add new methods, that will show up as options on the **Actions** menu for each type of object:

```
public class Student
{
    ...
    public IDomainObjectContainer Container { set; protected get; }
    ...
    [MemberOrder(4)][Hidden(WhenTo.OncePersisted)]
    public virtual DateTime DateOfBirth { get; set; }

    public void ConfirmDateOfBirth(DateTime dateOfBirth)
    {
        var message = dateOfBirth == DateOfBirth ? "CORRECT" : "INCORRECT";
        Container.InformUser($"The date of birth you entered is {message} for this
student.");
    }
    ...
    [MemberOrder(6)][Eagerly(EagerlyAttribute.Do.Rendering)]
    [TableView(false, "Subject", "SetName", "Teacher")]
    public virtual ICollection<TeachingSet> Sets { get; set; } = new
List<TeachingSet>();

    [Hidden(WhenTo.Always)]
    public int Age()
    ...
    public IQueryable<SubjectReport> ListRecentReports()
    {
        int id = this.Id;
        var studentReps = Container.Instances<SubjectReport>().Where(sr =>
sr.Student.Id == id);
        return studentReps.OrderByDescending(sr => sr.Date);
    }

    public SubjectReport CreateNewReport()
    {
        var rep = Container.NewTransientInstance<SubjectReport>();
        rep.Student = this;
        return rep;
    }
}
```

```
public class Teacher
{
    public IDomainObjectContainer Container { set; protected get; }
    ...
    [MemberOrder(3)][Optionally]
    public virtual string JobTitle { get; set; }
    ...
    [MemberOrder(5)]
    [Eagerly(EagerlyAttribute.Do.Rendering)]
    [TableView(false, "Subject", "YearGroup", "SetName")]
    public virtual ICollection<TeachingSet> SetsTaught()
    {
        int id = this.Id;
        return Container.Instances<TeachingSet>().Where(s => s.Teacher.Id ==
id).OrderBy(s => s.Subject.Name).ThenBy(s => s.YearGroup).ToList();
    }
}
```

Finally, we will update the Initializer to create some instances of the new types and associate them with Students:


```

public class Initializer : DropCreateDatabaseIfModelChanges<DatabaseContext>
{
    protected override void Seed(DatabaseContext context)
    {
        ...
        var subjects = context.Subjects;
        var csc = CreateNewSubject(subjects, "Computer Science");
        var math = CreateNewSubject(subjects, "Maths");
        var eng = CreateNewSubject(subjects, "English");
        var phy = CreateNewSubject(subjects, "Physics");
        var chem = CreateNewSubject(subjects, "Chemistry");
        var bio = CreateNewSubject(subjects, "Biology");
        var his = CreateNewSubject(subjects, "History");
        var fre = CreateNewSubject(subjects, "French");
        var ger = CreateNewSubject(subjects, "German");

        var sets = context.Sets;
        var CS12 = CreateNewSet(sets, "CS12", csc, 12, dec);
        var CS13 = CreateNewSet(sets, "CS13", csc, 13, dec);
        var MA09_1 = CreateNewSet(sets, "MA09_1", math, 9, rob);
        var MA10_1 = CreateNewSet(sets, "MA10_1", math, 10, rob);
        var MA11_1 = CreateNewSet(sets, "MA11_1", math, 11, hu);
        var MA09_2 = CreateNewSet(sets, "MA09_2", math, 9, dou);
        var MA10_2 = CreateNewSet(sets, "MA10_2", math, 10, dou);
        var MA11_2 = CreateNewSet(sets, "MA11_2", math, 11, dou);
        AssignStudents(CS12, alg, cee, frt);
        AssignStudents(CS13, vee, sim);
    }
    ...
    private Subject CreateNewSubject(DbSet<Subject> subjects, string name)
    {
        var obj = new Subject() { Name = name };
        subjects.Add(obj);
        return obj;
    }

    private TeachingSet CreateNewSet(DbSet<TeachingSet> sets, string name, Subject subject,
int yearGroup, Teacher teacher)
    {
        var obj = new TeachingSet() { SetName = name, Subject = subject, YearGroup =
yearGroup, Teacher = teacher };
        sets.Add(obj);
        return obj;
    }

    private void AssignStudents(TeachingSet set, params Student[] students)
    {
        foreach (Student stu in students)
        {
            set.Students.Add(stu);
        }
    }
}

```

Now we are ready to run the application again. In the following exercise you will be asked to use the application, and then to answer questions about how, or in some cases, why certain functionality/behaviour has been implemented. You are not *expected* to necessarily *know* the

answers immediately. Instead, the challenge is to explore the code and make good guesses about the relationships between the observed behaviour and the code written. (You are also allowed to consult the Naked Objects Developer Manual).

Exercise 49

If, through the user interface you create a new student, you are required to provide a **Date Of Birth**, but for other students, retrieved from the database, the **Date Of Birth** field is not visible – although, deliberately, you can still see the student’s current age (in years) in the title.

1) How (in the code) has this been achieved?

2) *Why* might the application designed have chosen to make this change?

Viewing the student record for Harry Haskell, click on the **Actions** menu in the top-left corner to show what you can do to, or with, that student object.

3) Capture a screen snippet showing those new actions.

Invoke the **Confirm Date Of Birth** action *twice*, first with the date 07/04/2003 and then as 08/04/2003.

4) Capture a screen snippet that shows how the system responds in each case.

Still on Harry Haskell, invoke the action **Create New Report**. You will be presented with an Unsaved Subject Report to complete and **Save**. The **Grade** field is a string type, but you are presented with a drop-down list of choices from A* to F.

5) How have these options been specified in the code, and what ties them to the **Grade** field?

The **Given By** field also offers a drop-down list of choices. However, the implementation of the method that does this slightly different (find it in the code).

6) Why is the code pattern different, and how is it obtaining the results to return?

7) The **Notes** field is a single string, but unlike other string fields allows carriage returns. What code makes this possible? And what happens if you type more than six lines of text?

Unlike all the other properties, you can Save a subject report without entering any notes.

8) How is this specified in the code? And how does the user know in advance that the **Notes** field may be left blank?

The unsaved subject report automatically shows the student who it is for and does not allow the user to change this (though they can **Cancel** the report without saving.)

9) How does the subject report object know which student it is for (i.e. how has this been specified in code), and what prevents that **Student** property from being changed by the user (unlike other properties containing object references such as **Given By**)?

Create and save a subject report. Go back to the student and invoke the action List Recent Reports.

10) Capture a screen snippet showing a split screen with the list of reports on the left and an expanded view of that report on the right.

11) What is the significance of the `[Eaglerly(EaglerlyAttribute.Do.Rendering)]` and `TableView` attributes used in two places in the new code. (You can look this up in the Naked Objects documentation, or just make a guess and *confirm it by removing the attribute(s) and running the application again*.)

We'll finish by reviewing the structure of the domain object model as it currently stands.

Exercise 50

Update the class diagram (or create a new one if need be) to show the five classes: `Student`, `Teacher`, `Subject`, `TeachingSet`, and `SubjectReport`, with all associations between those classes shown as arrows.

1) Lay out the classes such that the associations may be clearly read and capture a screen snippet of the diagram.

2) Looking at the class diagram, how would you characterise the relationship between `Student` and `TeachingSet`? (It might be helpful to refer to [Types of association](#)).

3) From your understanding of relational databases, how would you expect this relationship to be represented in the database schema?

4) Now reopen a connection to the database and explore the new schema. Paste a screen snippet that highlights the representation of the `Student/TeachingSet` relationship and shows data representing specific instances of this relationship.

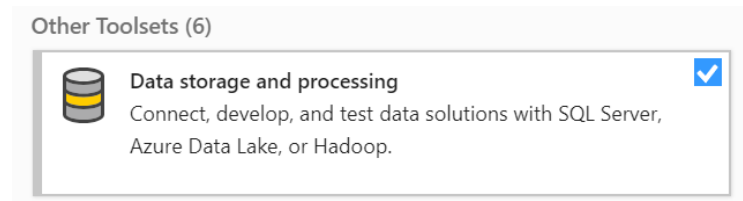
Appendices

Technical pre-requisites

Visual Studio 2019

This book is designed for use with **Visual Studio 2019 Community Edition**, which may be downloaded from <https://visualstudio.microsoft.com/downloads/>.

You should ensure, when installing that you include the Data storage and processing option:



Browser

Chrome is used as the browser in Part II of the book. It should work with any browser, though you will need to ensure that it is able to display JSON files. Most browsers offer plug-in options for this; for Chrome, the recommended plug-in is **JSONView**.

Naked Objects

Version 11 of the **Naked Objects** framework is used for the last few chapters, where you will be instructed how to install it as a **NuGet** Package, from the NuGet Public Gallery. Naked Objects is fully open source, hosted on: <https://github.com/NakedObjectsGroup/NakedObjectsFramework>. For those who wish to experiment further with this powerful framework, a comprehensive Developer Manual may be downloaded from: <https://github.com/NakedObjectsGroup/NakedObjectsFramework/blob/master/Documentation/DeveloperManual.docx>.

The **OOPRecords.Server.zip** and **OOPRecords.Client.zip** folders may be downloaded from: <https://community.computingschool.org.uk/resources/6176/single>

Node.js

In order to build and use the Naked Objects client you will need to have **Node.js** installed, which you can download from: <https://nodejs.org/en/download/>.

Troubleshooting

Error message: Cannot Drop database ... because it is in use

A very common error arising from using Entity Framework is a message that the database cannot be 'dropped' because it is 'in use'. This means that there is a connection open to the database in Visual Studio. Typically this either be an explicit connection (visible in the **Server Connection**) or an implicit connection caused by having a view of the database open (as a document/tab). This error is usually fixed by closing the connection in the **Server Explorer** and/or closing any open tabs with views of the database.

Occasionally, Visual Studio seems to hold onto a hidden connection the database that defies the above advice; if this happens, the advice is to restart Visual Studio.

Error message: System.Data.Entity.DynamicProxies... is not a IEntityWithChangeTracker

If you see the above message (the ... will be the name of a domain class followed by a long number), the end of the error message gives the real clue:

*(all properties must be made virtual/Overrideable
and all collection properties should be of type ICollection<T>)*

The most likely cause is that you have missed off the keyword `virtual` from one or more of the properties (including collections) on that domain class.

A second possibility is that you have a collection property that is not *returning* an `ICollection<DomainType>`. Note that this refers specifically to the *return type*, not the type that the collection property is initialised with (e.g. a `List<DomainType>`).

Server project fails to start but does not show an error message

The lack of an error message may be caused by your specific run settings in Visual Studio. However, all errors are 'logged' in the **nakedobjects.log** file, which you can access from the **Solution Explorer**. Note that as well as any errors, this file will contain a lot of INFO messages, which may be ignored. Search the file for the *first* ERROR message.

Each time the server is run, any *existing* log file will be renamed (to include the date and run-number) and **nakedobjects.log** will be started afresh.

New seed data not added to the database

The `Initializer` class has been specified to inherit from `DropCreateDatabaseIfModelChanges`. If you add new Seed data without changing the model, the database will not be re-created and hence the Seed method will not be run. One solution to this is to drop (delete) the database manually through the **Server Explorer** and then re-run. Another option is to change the `Initializer` so that it inherits from `DropCreateDatabaseAlways`. The latter is the easy option, but it also means that any data you add/change during a run will always be lost on the next run.

Error parsing a date in the Initializer

If an exception is thrown within the call to `Convert.ToDateTime(dob)`, within the `NewStudent` method in the `Initializer` class, the most likely cause is that your system configuration has

overridden the default Short Date format. If you don't want to reset the format, you can make the code work by changing all the dates that are specified as strings in the `Initializer` class to an alternative format, for example "2003-12-27". You will, however, also need to enter dates via the console in the same format.

Object-oriented programming (OOP) is a radical way of writing software, where the fundamental building blocks have both 'state' (data) and 'behaviour' (functionality). As Alan Kay – ACM Turing Award recipient and one of the pioneers of OOP – writes in the Foreword, this is like building a program out of thousands of tiny self-contained computers intercommunicating via their own internal network, an idea that was partly inspired by his understanding of molecular biology.

This is the first OOP textbook that has been written specifically for the context of A-level Computer Science. It will give you a solid grounding both in the theory of OOP, and in the practical application of the theory to real-world problems.

Part I teaches the fundamental principles of OOP – encapsulation, polymorphism, inheritance, association, and more – through the development of an interactive drawing program from scratch. Each principle is introduced as the response to a real design challenge.

Part II introduces some of the techniques appropriate for building much larger scale applications – including the 'persistence' of objects in a database – in the context of building a simple 'records management' system for school administration.



Richard Pawson worked in the computing industry for 40 years before teaching A-level Computer Science at Stowe School. He has a BSc in Engineering, a PhD in Computer Science, and a PGC in Intellectual Property Law. He now splits his time between managing the open source Naked Objects Framework, writing books, and restoring a vintage car – also 'from the metal up'.