

LEVEL 4

**DESIGNING AND DEVELOPING OBJECT-ORIENTED
COMPUTER PROGRAMS**

Student Workbook

Modification History

Version	Date	Revision Description
V1.0	June 2011	For Release
V2.0	June 2017	Updated Progammig language changed from Java to C#
V2.1	September 2018	Updated TQT

© NCC Education Limited, 2017

All Rights Reserved

The copyright in this document is vested in NCC Education Limited. The document must not be reproduced by any means, in whole or in part, or used for manufacturing purposes, except with the prior written permission of NCC Education Limited and then only on condition that this notice is included in any such reproduction.

Published by: NCC Education Limited, The Towers, Towers Business Park, Wilmslow Road, Didsbury, Manchester M20 2EZ, UK

Tel: +44 (0) 161 438 6200 Fax: +44 (0) 161 438 6240 Email: info@nccedu.com
<http://www.nccedu.com>

CONTENTS

1.	Unit Overview and Objectives	6
2.	Learning Outcomes and Assessment Criteria	6
3.	Syllabus	7
4.	Related National Occupational Standards.....	9
5.	Resources	9
5.1	Additional Software Requirements.....	9
5.2	Code	10
6.	Pedagogic Approach.....	10
6.1	Lectures	10
6.2	Laboratory Sessions.....	10
6.3	Private Study	10
7.	Assessment	10
8.	Further Reading List	11
Topic 1:	An Introduction to the .NET Framework	12
1.1	Learning Objectives.....	12
1.2	Timings.....	12
1.3	Laboratory Session	13
1.3.1	Resources Required	13
1.4	Tasks	25
1.5	Private Study Exercises.....	27
Topic 2:	Event Driven Programming	28
2.1	Learning Objectives.....	28
2.2	Timings.....	28
2.3	Laboratory Session	29
2.3.1	Resources Required	29
2.4	Tasks	34
2.5	Private Study Exercises.....	36
Topic 3:	Programming Structures (1)	37
3.1	Learning Objectives.....	37
3.2	Timings.....	37
3.3	Laboratory Session	38
3.3.1	Resources Required	38
3.4	Tasks	45
3.5	Private Study Exercises.....	47
Topic 4:	Programming Structures (2)	48
4.1	Learning Objectives.....	48
4.2	Timings.....	48

4.3	Laboratory Session	49
4.3.1	Resources Required	49
4.4	Tasks	54
4.5	Private Study Exercises.....	56
Topic 5:	Object Orientation (1).....	58
5.1	Learning Objectives.....	58
5.2	Timings.....	58
5.3	Laboratory Session	59
5.3.1	Resources Required	59
5.4	Tasks	69
5.5	Private Study Exercises.....	70
Topic 6:	Consolidation (1).....	71
6.1	Learning Objectives.....	71
6.2	Timings.....	71
6.3	Laboratory Session	72
6.3.1	Resources Required	72
6.4	Tasks	74
6.5	Private Study Exercises.....	75
Topic 7:	Data Structures.....	76
7.1	Learning Objectives.....	76
7.2	Timings.....	76
7.3	Laboratory Session	77
7.3.1	Resources Required	77
7.4	Tasks	86
7.5	Private Study Exercises.....	87
Topic 8:	Object Orientation (2).....	88
8.1	Learning Objectives.....	88
8.2	Timings.....	88
8.3	Laboratory Session	89
8.3.1	Resources Required	89
8.4	Tasks	99
8.5	Private Study Exercises.....	100
Topic 9:	Consolidation (2).....	101
9.1	Learning Objectives.....	101
9.2	Timings.....	101
9.3	Laboratory Session	102
9.3.1	Resources Required	102
9.4	Tasks	103
9.5	Private Study Exercises.....	104

Topic 10: Testing and Error Handling	105
10.1 Learning Objectives.....	105
10.2 Timings.....	105
10.3 Laboratory Session	106
10.3.1 Resources Required	106
10.4 Tasks	114
10.5 Private Study Exercises.....	115
Topic 11: File IO	116
11.1 Learning Objectives.....	116
11.2 Timings.....	116
11.3 Laboratory Sessions.....	117
11.3.1 Resources Required	117
11.4 Tasks	124
11.5 Private Study Exercises.....	125
Topic 12: Databases in .NET	126
12.1 Learning Objectives.....	126
12.2 Timings.....	126
12.3 Laboratory Sessions.....	127
12.3.1 Resources Required	127
Laboratory and Private Study Sessions.....	133

1. Unit Overview and Objectives

This unit aims to give the learner a thorough grounding in programming methods, and a detailed knowledge of developing programs using C#.

2. Learning Outcomes and Assessment Criteria

Learning Outcomes; The Learner will:	Assessment Criteria; The Learner can:
1. Design object-oriented programs to address loosely defined problems	1.1 Identify a set of classes and their interrelationships to address the problem 1.2 Make effective use of encapsulation, inheritance and polymorphism 1.3 Select and reuse pre-existing objects and templates specialising as required 1.4 Structure the design so that objects communicate efficiently 1.5 Specify the properties and behaviour of classes to allow efficient implementation, selecting appropriate data types, data and file structures and algorithms 1.6 Record the design using well-established notations
2. Implement object-oriented programs from well-defined specifications	2.1 Produce a working program which satisfies the design specification 2.2 Make effective use of basic programming language features and programming concepts to implement a program that satisfies the design specification 2.3 Make effective use of the features of the programming environment 2.4 Make effective use of user interface components in the implementation of the program 2.5 Make effective use of a range of debugging tools
3. Develop object-oriented programs that reflect established programming and software engineering practice	3.1 Apply standard naming, layout and comment conventions 3.2 Apply appropriate data validation and error handling techniques
4. Develop test strategies and apply these to object-oriented programs	4.1 Develop and apply a test strategy consistent with the design identifying appropriate test data 4.2 Apply regression testing consistent with the test strategy 4.3 Use appropriate tools to estimate the performance of the program

5. Develop design documentation for use in program maintenance and end-user documentation	5.1 Record the final state of the program in a form suitable for subsequent maintenance 5.2 Provide end-user documentation that meets the user's needs
---	---

3. Syllabus

Syllabus			
Topic	Title	Proportion	Content
1	An Introduction to the .NET framework	1/12 1 hour of lectures 6 hours of laboratory sessions	<ul style="list-style-type: none"> Visual Studio IDE The Design of .NET programs Sequential Program Flow <p>Learning Outcomes: 2 & 3</p>
2	Event Driven Programming	1/12 1 hour of lectures 6 hours of laboratory sessions	<ul style="list-style-type: none"> Event Handling Mouse Events Paper Prototypes Wizard of Oz <p>Learning Outcomes: 2, 3, & 5</p>
3	Programming Structures (1)	1/12 1 hour of lectures 6 hours of laboratory sessions	<ul style="list-style-type: none"> Selections within .NET Branching Program Flow <p>Learning Outcomes: 2 & 3</p>
4	Programming Structures (2)	1/12 1 hour of lectures 6 hours of laboratory sessions	<ul style="list-style-type: none"> Repetition within .NET Iterative Program Flow <p>Learning Outcomes: 2 & 3</p>

5	Object Orientation (1)	1/12 1 hour of lectures 6 hours of laboratory sessions	<ul style="list-style-type: none"> • Classes • Objects • Encapsulation • Abstraction <p>Learning Outcomes: 1, 2 & 3</p>
6	Consolidation (1)	1/12 6 hours of laboratory sessions	<ul style="list-style-type: none"> • Worked example of material to date <p>Learning Outcomes: 1, 2, 3 & 5</p>
7	Data Structures	1/12 1 hour of lectures 6 hours of laboratory sessions	<ul style="list-style-type: none"> • Arrays • ArrayLists • Dictionary • Generics <p>Learning Outcomes: 1, 2 & 3</p>
8	Object Orientation (2)	1/12 1 hour of lectures 6 hours of laboratory sessions	<ul style="list-style-type: none"> • Inheritance • Polymorphism • Introduction to UML • Coupling and Cohesion <p>Learning Outcomes: 1, 2, 3, 4 & 5</p>
9	Consolidation (2)	1/12 6 hours of laboratory sessions	<ul style="list-style-type: none"> • Worked example of material to date <p>Learning Outcomes: 1, 2, 3 & 5</p>
10	Testing and Error Handling	1/12 1 hour of lectures 6 hours of laboratory sessions	<ul style="list-style-type: none"> • Testing strategies • Regression testing • Detection and correction of errors • Exception handling <p>Learning Outcomes: 4 & 5</p>

11	File IO	1/12 1 hour of lectures 6 hours of laboratory sessions	<ul style="list-style-type: none"> • File IO • Serialization <p>Learning Outcomes: 2, 3, & 4</p>
12	Databases with .NET	1/12 1 hour of lectures 6 hours of laboratory sessions	<ul style="list-style-type: none"> • Connection to databases • Querying data • Representing Data <p>Learning Outcomes: 2, 3, 4 & 5</p>

4. Related National Occupational Standards

The UK National Occupational Standards describe the skills that professionals are expected to demonstrate in their jobs in order to carry them out effectively. They are developed by employers and this information can be helpful in explaining the practical skills that students have covered in this unit.

Related National Occupational Standards (NOS)	
<p>Sector Subject Area: IT and Telecoms</p> <p>Related NOS: ESKITP5013 P1-6 - Carry out system development activities under direction; ESKITP5014v2 P1-5 - Perform systems development activities; ESKITP5014v2 P6-10 - Contribute to the management of systems development; ESKITP5022v2 - Perform software development activities; ESKITP5024 P6-12- Carry out IT/Technology solution testing activities under direction; ESKITP5034 P1-4 - Carry out IT/Technology solution testing.</p>	

5. Resources

Lecturer Guide: This guide contains notes for lecturers on the organisation of each topic, and suggested use of the resources. It also contains all of the model answers for the activities presented in the Student Workbook.

Student Workbook: This guides the student through the unit and presents details of the tasks they should undertake during the laboratory sessions. It also included the exercises which have been provided for private study time.

5.1 Additional Software Requirements

This unit also makes use of Visual Studio Community edition. Students will need to have access to this during laboratory and private study time. This is available from <https://www.visualstudio.com/vs/community/>.

If you are using Internet Explorer, a message should appear at the top of the browser window when you click on this link, saying “click here for options”. Click it and choose to download the software. After it downloads, choose to run it and it will then install.

5.2 Code

Code is included in the Student Workbook for use during the laboratory sessions and private study time and can be copy and pasted as required.

6. Pedagogic Approach

Suggested Learning Hours						
Guided Learning Hours				Assessment	Private Study:	Total:
Lectures:	Tutorial:	Seminar:	Laboratory:			
10	12	-	60	30 (assignment)	38	150

The contact time for this unit is comprised of lectures and laboratory sessions. The breakdown of the hours for each topic is given in the Topic Notes which follow.

6.1 Lectures

The lecture time is intended to provide short presentations to introduce new concepts as they occur in the topic. This time therefore needs to be scheduled with the laboratory sessions as a formal one hour lecture will not be given. Instead, lecturers should introduce new concepts with brief presentations (approximately 10-15 mins) at the start of the relevant laboratory sessions. Students should then work through the associated notes in the Student Workbook under the supervision and guidance of the lecturer.

6.2 Laboratory Sessions

These sessions take a task-based approach to student learning. The details of the practical tutorials and exercises which students should work through are provided in the Student Workbook. These sessions should be supervised by a C# competent instructor. If there is insufficient time to complete all tasks in the Laboratory Session, candidates may attempt some of the tasks during their Private Study time, with the lecturer checking their work during the Tutorial session.

6.3 Private Study

In addition to the taught portion of the unit, students will also be expected to undertake private study. Exercises are provided in the Student Workbook for students to complete during this time. Teachers will need to set deadlines for the completion of this work and then review the solutions with the students.

7. Assessment

This unit will be assessed by means of an assignment worth 100% of the total. This assessment will be based on the Learning Outcomes and Assessment Criteria given above and students will be expected to demonstrate that they have met the unit's Learning Outcomes. Sample assessments are available through the NCC Education website for your reference.

8. Further Reading List

A selection of sources of further reading around the content of this unit must be available in your Accredited Partner Centre's library. The following list provides suggestions of some suitable sources:

RB Whitaker (2015). *The C# Player's Guide* (2nd Edition). Starbound Software.
ISBN-10: 0985580127
ISBN-13: 978-0985580124

Joesph Albahari & Ben Albahri (2016). *C# 6.0 in a Nutshell: The Definitive Reference*. O'Reilly.
ISBN-10: 1491927062
ISBN-13: 978-1491927069

Andrew Troelsen & Philip Japikse (2016). *C# 6.0 and the .NET 4.6 Framework*. Apress.
ISBN-10: 1484213335
ISBN-13: 978-1484213339

Topic 1: An Introduction to the .NET Framework

1.1 Learning Objectives

On completion of this topic, students will be able to:

- Understand how the .NET framework functions;
- Understand the importance of Integrated Development Environments (IDE);
- Use the Visual Studio IDE;
- Design .NET program front-ends;
- Make use of sequential coding statements in C#;

1.2 Timings

Lectures:	1 hour
Laboratory Sessions:	5 hours
Private Study:	3 hours
Tutorial:	1 hour

1.3 Laboratory Session

1.3.1 Resources Required

Ensure that Visual Studio is installed on your computer:

1.3.1.1 Introducing Visual Studio

In this course we will be looking at how to build object oriented, event driven programs using the C# programming language. C# is part of Microsoft's .NET framework, and is a very popular language that is in considerable demand. For this, we'll be making use of Visual Studio as an **Integrated Development Environment**, or IDE.

In the early days of programming, developers would make use of several programs to write computer code. They'd use a text editor to write out code statements, a compiler to turn those statements into computer code, and a linker to hook together all those computer code fragments into a working program. If they needed to work with a database they'd use a dedicated tool for it.

An IDE is a software package that combines all of these separate tools into a single application so that everything is all contained within a unified package. This makes it much easier for software to be developed since there's no need to constantly shift between multiple different tools. Visual Studio is one such IDE, and one of the most powerful available. Other IDEs exist for other languages and platforms, such as Netbeans for Java, Android Studio for mobile development with Dalvik, and many more besides. They all do largely the same job, and there's no strict requirement for you to use Visual Studio to make C# programs. There are other IDEs, such as MonoDevelop, that allow you to do the same thing.

The IDE is your application, it is not the programming language itself.

1.3.1.2 The .NET Framework

C# is a programming language that is part of the .NET framework. This is a large and powerful programming framework that is aimed primarily, but not entirely, at Windows based platforms. The .NET framework is made up of five key parts:

- The Common Language Runtime (CLR)
- The base class library
- Common language specifications
- ADO .NET
- ASP .NET

The first thing it's important to know with .NET is that the language you use doesn't really matter. You can code in Visual Basic.NET, C#, J#, C++.Net or any of the 70+ supported .NET languages and the only thing that changes is the **syntax** of the code you use.

Every programming language is made up of a set of rules as to how various parts should go together, and a set of special words. When we put these together, we are working with the **syntax** of our language. C# is known as a C-type language, like Java, because it makes use of a **syntax** that is derived from the C programming language. As such, when you know how to

code in C# you should find it easy to move into PHP, Javascript, Java, or any number of other languages.

What we write is known as **source code**, and it's a set of instructions in a format that is relatively easy for humans to understand. It's not provided in a form that makes sense to the computer—it's something that's supposed to be easy for **us** to work with. All we have on our computers when we save a piece of source code is a text-file, not a computer program.

In order to turn our source code into a computer program we go through a process of **compilation**. The .NET framework takes the code we have written and turns it into a special kind of computer code that is understood by the Common Language Runtime. Every language, no matter what we use, **compiles** down into exactly the same computer code. This is known as **intermediate language (IL)**.

On any computer where the .NET framework is installed there is a piece of software running called the **Common Language Runtime (CLR)**. The CLR is a **virtual machine** – when we run our intermediate language it's not run on the computer itself. Instead, we provide instructions and requests to the CLR, and the CLR sends requests on to the operating system as required.

This might seem like a roundabout way to handle the whole thing, but there's a good reason for it. Once upon a time, when a program was compiled it ended up as **machine code** – a series of 1s and 0s that were understood by the electronics of the machine itself. Every different kind of processor and every different kind of operating system had its own way of representing and executing machine code. If you wanted to have one application that ran on lots of different systems you'd need to compile it on each – and it would never work as easily as you would like. **Porting** was the process of turning code written for one system into working code for another. It was time consuming, costly, and difficult to do correctly.

The use of a virtual machine means that for each platform where the code is supposed to work all that's needed is a version of the Common Language Runtime, and Microsoft writes that for us. We don't write code for the computer we're on – we write it for the CLR, and Microsoft makes a CLR available for all supported platforms. When we write our code for .NET, it will run anywhere we can get access to a CLR.

However, more than this the .NET framework is also a series of routines for simplifying common core actions for developers. It comes with the **base class library**, which is a set of data structures which are provided for us. When we need to connect to an internet location or load data from a database we don't need to write all of that for ourselves. When we want to put code up on a server, we don't need to do that either. ADO .NET is the data access engine for .NET and ASP .NET is the web deployment architecture.

.NET then is a powerful tool, and we're only going to scratch the surface during this course. However, what you learn here is going to open up a lot of doors for the future – any piece of software you need to write, you'll be able to do it with C# and .NET as your skills develop.

1.3.1.3 What is Programming?

Programming is a difficult task, unlikely almost anything you may have learned before – it's like learning how to do mathematics in a foreign language, it needs you to learn new syntax combined with a formally exact vocabulary. When you learn how to speak a different language, people can often make out meaning even when you make mistakes. A computer cannot do that. And worse, it does exactly what you tell it to do – that sounds great in theory, but it can be a huge problem.

Programming is essentially the task of taking a list of instructions, and writing them in such a way that the computer can follow your instructions to complete a task. To give a simple, real world

example – imagine the task of making scrambled eggs. Imagine too that you have an obedient, but simple-minded kitchen assistant, and you would like them to make some scrambled eggs for you.

They do everything you tell them to do, and they do it exactly as you tell them. It sounds like an ideal situation, but consider the following instructions:

1. Get two eggs from the fridge
2. Get some milk from the fridge
3. Get some butter from the fridge
4. Break the eggs into a jug
5. Whisk the eggs until they are well mixed.
6. Heat some milk and some butter in a pot.
7. Add the egg mixture.
8. Stir until scrambled eggs are made.

The instructions are pretty simple, but your assistant falters at the first instruction. He, or she, stands hitting the fridge door. You need to amend your instructions a little:

1. Open the fridge.
2. Get two eggs from the fridge
3. Get some milk from the fridge
4. Get some butter from the fridge
5. Close the fridge
6. Break the eggs into a jug
7. Whisk the eggs until they are well mixed.
8. Heat some milk and some butter in a pot.
9. Add the egg mixture.
10. Stir until scrambled eggs are made.

You set your assistant back to work and they open the fridge. They get two eggs out. Then they stand there with a confused look on their face. "What does 'some' mean?", they ask. We need to be more specific:

1. Open the fridge.
2. Get two eggs from the fridge
3. Get two tablespoons of milk from the fridge
4. Get a knob of butter from the fridge
5. Close the fridge.
6. Break the eggs into a jug
7. Whisk the eggs until they are well mixed.
8. Heat some milk and some butter in a pot.
9. Add the egg mixture.
10. Stir until scrambled eggs are made.

You start them running again.

"The fridge is already open", they say, and they give up until you can provide more instruction.:

1. If the fridge door is shut, open the fridge.
2. Get two eggs from the fridge
3. Get two tablespoons of milk from the fridge
4. Get a knob of butter from the fridge
5. Close the fridge.
6. Break the eggs into a jug
7. Whisk the eggs until they are well mixed.
8. Heat some milk and some butter in a pot.
9. Add the egg mixture.
10. Stir until scrambled eggs are made.

And so on, until you have a completely correct, entirely unambiguous list of instructions that anyone can follow without mistakes or confusion. This is the essence of what programming is all about. The only difference is, your instructions are given in C#, and your assistant is the .NET framework.

1.3.1.4 Variables and other technical terms

Most of what we do in C# is going to involve the use of a **variable** at some point. You can think of a variable as a little box with a name, into which we put some information so it's available for later use. Within C#, variables have a **name**, which we use to refer to it, and a **type** which tells C# what kind of data might be stored within. We create a variable like so:

```
<Type> <name>;
```

Some of the types we'll see a lot are known as **primitive** data types and make up the basic building blocks of the language. The ones we'll see are:

Type	Description
int	Whole numbers. Short for integer
double	Real numbers, making use of decimal points.
Boolean	A true or false value.
String	A sequence of alphanumeric characters

We'll see more as we go along. If we want to create a variable that holds a whole number, and we want that variable to be called **num**, we'd use the following line of code:

```
int num;
```

If we want to put something into that variable, we use what is known as an **assignment**:

```
num = 10;
```

We can do simple sums using variables. For that, we use the set of what are known as **arithmetic operators** – plus, minus, division (represented by /) and multiplication (represented by *). For example:

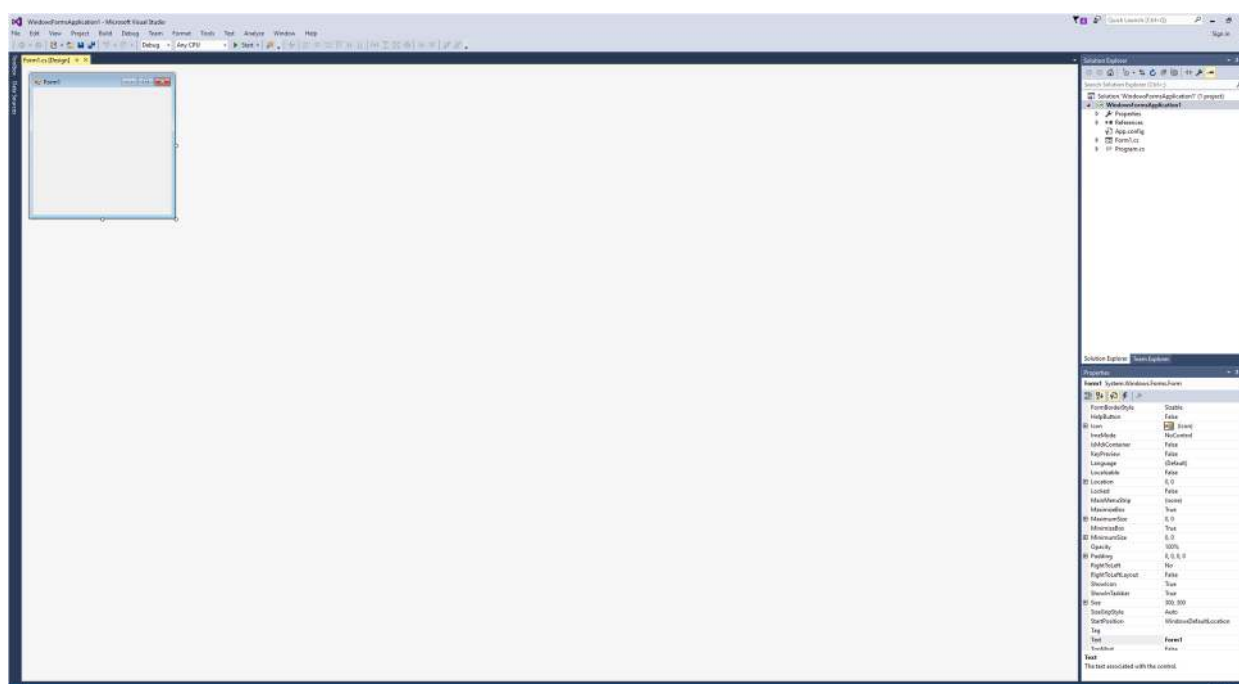
```
int num;  
int ans;  
  
num = 10;  
  
ans = num * 20;
```

There are a few other terms we should define before we go forward too, so that you're familiar with them when you encounter them:

Term	Description
Syntax	The formal parts of a language that define what is valid when writing code. For example, the order of words we use to create a variable is defined by the language syntax .
Syntax error	An error that is the result of us not obeying the syntax of the language. For example, if we declared a variable like so: <pre>num is an int;</pre> That would be a syntax error, and our program would not compile .
Compile	The act of turning the human readable code in the IDE into something the computer can run and execute upon.
Runtime error	An error that occurs when the program is running – usually caused by incorrect logic in the program code we provide.
Widget or Control	A specific user interface element we place on our application. These include buttons, text fields, and combo boxes amongst other things.
GUI	A Graphical User Interface – that's what our users will use to interact with the programs we write.

1.3.1.5 Let's do some programming

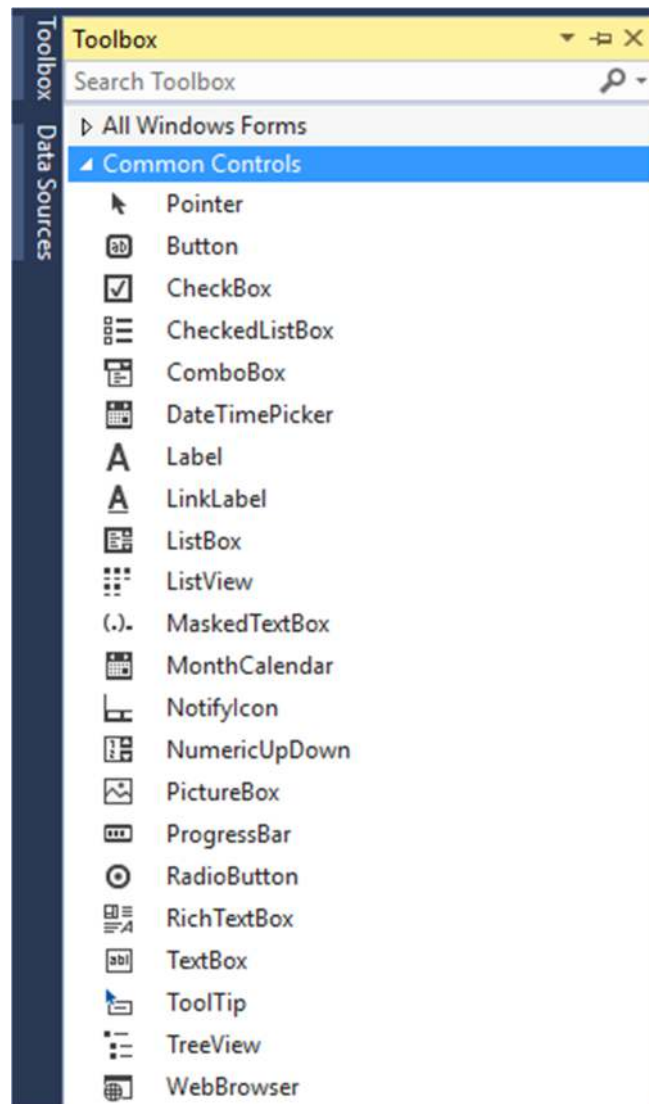
Start up Visual Studio, and you'll be presented with your programming environment. We'll be creating a **windows form application** and doing some simple things with it.



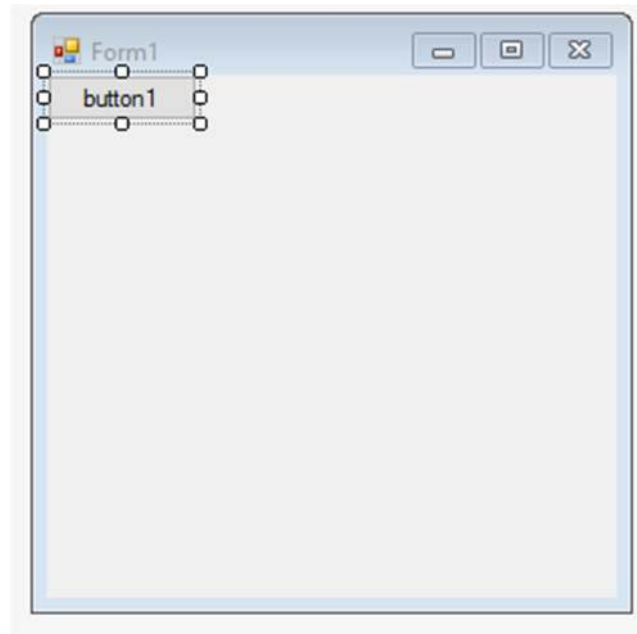
At the top left of this interface is your **form**. This is the window of the application you'll be creating. Along the right hand side at the top is the **solution explorer**, which lets us look through all the files that are part of the project. We'll talk about that more as we go through the unit. Underneath that is the **property explorer**. We'll be using that a lot as we go through this course.

There is a **toolbox** tab at the top left of the interface. Clicking this will open the toolbox of components we can select. We want to expand **common controls** which will reveal the standard

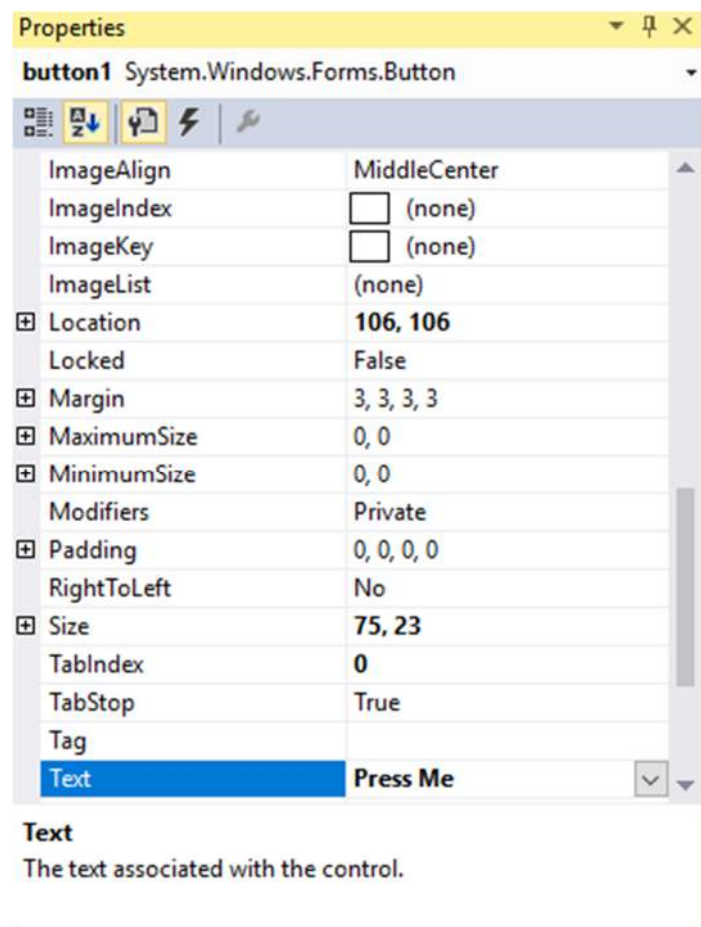
GUI components we'll be using. To begin with, we use Visual Studio to draw how we want an application to look. Later we'll see more powerful ways to set up the programs we're creating.



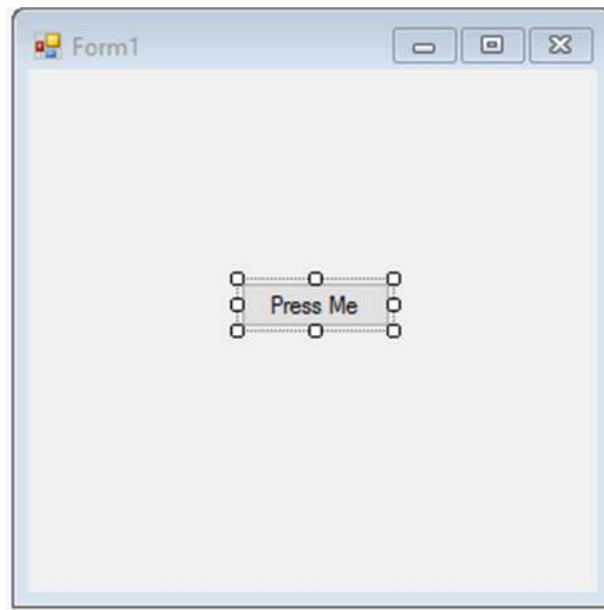
Double click on **Button**, and it'll place a button on your application. It won't do anything interesting yet. But this is how we begin with writing a C# program.



We can drag and drop this button to where we want it to go, but more importantly we can also change the way it looks and behaves by changing its **properties**. Drag the button into the centre of your form, and then find the **Text** property. Change it to **press me**.



This will give us the following interface on our application:



Scroll up to the (name) property and change it to cmdPress. This won't change anything in the application, but it changes the name we use when we want to refer to this button in code. We'll usually do this by giving it a name that begins with a three letter prefix that tells us what kind of control it is. We don't have to, but it's good practise because it means we'll know in the code what kind of control we're working with.

1.3.1.6 Our first C# Program

What we've drawn here on the form is a **button control**. Controls are the building blocks of C# programs. We've set its **text** property to change how it looks, but there are many more properties that we haven't touched. We'll see more of these as we go along.

C# is an **event driven language**, which means that when we write code we associate it with **events** that occur in the lifetime of a program. When we drag a window around the display we're creating events. When we move the mouse across it, we create events. When we click on a button, we create an event. Most of the events that happen we're not interested in. Some of them we are. For example, we want to know when someone clicks that button we added.

Double click on the button within the IDE and it'll bring up the code editor for this program. Don't let this intimidate you – it's easier than you think because Visual Studio does most of the work for us here.

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WindowsFormsApplication2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
```

```

        InitializeComponent();
    }

    private void cmdPress_Click(object sender, EventArgs e)
    {
    }
}

```

There is a part of this code that contains the **stub** where we'll put the code relating to our button:

```

private void cmdPress_Click(object sender, EventArgs e)
{
}

```

This tells C# that we have a control called cmdPress, and we want to handle its **click** event – that is, when someone clicks on it. Add in the following code:

```

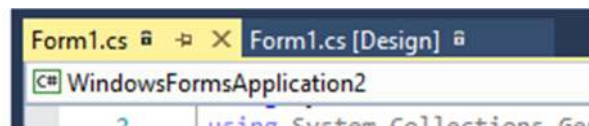
private void cmdPress_Click(object sender, EventArgs e)
{
    MessageBox.Show("Hello World");
}

```

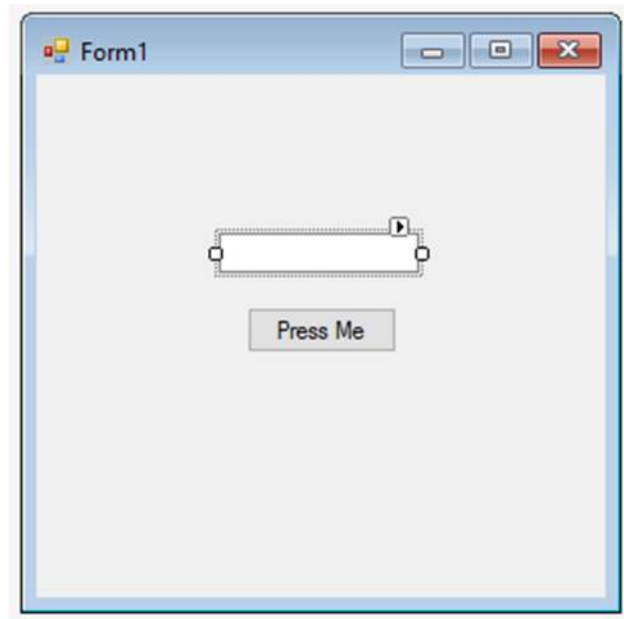
Be sure to do this exactly – C# is not forgiving of errors, and it expects that you'll obey the **syntax** of the language. We'll talk more about this as we go on.

Now, click on the play symbol at the top of Visual Studio and press the button when your application starts up. You'll see it displays a message box containing the text we entered. Press the stop button, and it'll stop and return you to the IDE.

Note you can switch between the code view and the **design view** by selecting the tab at the top of the screen.



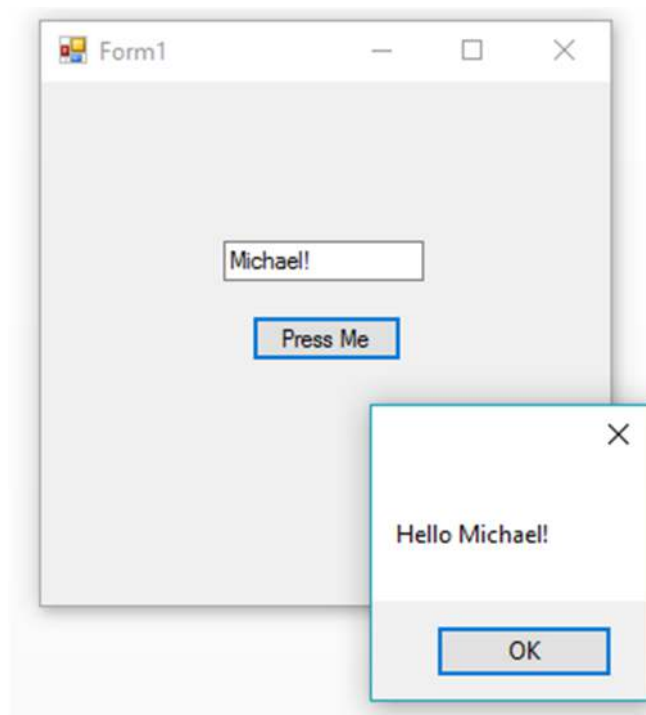
Making use of the Toolbox as before, add in a TextBox control. Call it txtName, and position it above the button on your form.



Now adjust the code so it says the following:

```
private void cmdPress_Click(object sender, EventArgs e)
{
    MessageBox.Show("Hello " + txtName.Text);
}
```

Run the program again, and type something into the text box. Press the button, and you'll see it appears in the messagebox that appears:



And if you wanted to put something **into** the textbox when you press the button:

```
private void cmdPress_Click(object sender, EventArgs e)
```

```
{  
    txtName.Text = "My content in here";  
}
```

This is the crux of software development in Visual Studio. We add controls to a form, and then in the code that we write we make use of the properties of those controls to make things happen.

We will have to wait until we've covered some more of C#'s design before we can talk about what most of this code does, but we'll get to it. For now though, congratulations on completing your first C# program!

1.4 Tasks

Task 1

Your first task is to become comfortable with the Visual Studio interface. You'll be spending a lot of time working with this, so it's important you understand how to make it do what you want. So, here are some things to try:

1. In the design view, add a PictureBox to your application.
 - a. Change its image property to an image of your choice. You'll do this by importing a local resource and finding it on your file system.
 - b. See what happens when you change its SizeMode property to different values.
 - c. Change its BackColour and BorderStyle.
 - d. Change the visible property
2. Start a new project and add a ComboBox control
 - a. Add some things into its Items property.
 - b. Those will appear in the combo box when you run your program.
3. Start a new project and create a progress bar. Call it PrgCounter
 - a. Change its Value to 50.
 - b. Add in a second progress bar. Call it PrgCounterTwo.
 - c. Change its Value to 25.
4. Create a new project. Add two text boxes.
 - a. Set one so it is a multiline control through the multiline property.
 - b. See what happens when you try to resize both of them in terms of length and height.
5. Replace the multiline control with a WebBrowser and explore how its URL property works.
6. Create a new project and add two checkboxes and two radio buttons. See how they differ.
7. Click on the form itself. Explore how its properties work.

Task 2

Once you're comfortable with the different available controls, it's time to try manipulating properties in code. Create a new project for this.

Add in two textboxes and a combo box. Add a button at the bottom. Add a checkbox. Set appropriate properties for all of these for names and text.

Making use of the cmdPress_click function stub, implement the following:

1. Place the contents of the first textbox into the second.
2. Making use of the SelectedItem property of the combobox, display in a message box which item from the list has been selected.
3. Make it so when pressing the button it activates a checkbox by setting its Checked property to true.

Task 3

Making use only of button and textbox controls, create an application with the layout of a standard calculator. It should contain buttons for each number along with a display along the top for the numbers that have been typed it. It should also contain buttons for arithmetic operations. It isn't necessary for the calculator to do anything yet, but the layout should be complete.

1.5 Private Study Exercises

Exercise 1

Describe what is meant by the .NET Framework. Draw a diagram of how the CLR and IL interacts with source code.

Exercise 2

Describe the advantages of an IDE.

Exercise 3

Explain what is meant by a Virtual Machine. Outline the benefits that go along with its use.

Exercise 4

Explain what is meant by the term **Form**. Draw a diagram showing the relationship between Forms, Controls and Properties.

Exercise 5

Explain what is meant by the term **Control**. Give some examples of controls you have used so far, and what role they fill. Make a note of any particularly useful properties they have, and how you have used them.

Exercise 6

Explain what a **property** is and how it is changed. Give some examples of properties you have used, and what they did when you changed them. Start to build up a log of what the properties you have available are, so you can refer to this as you develop future programs.

Topic 2: Event Driven Programming

2.1 Learning Objectives

On completion of this topic, students will be able to:

- Understand how events in C# work;
- Make use of event handling to design responsive programs;
- Understand the role of wireframes in UI design;

2.2 Timings

Lectures:	1 hour
Laboratory Sessions:	5 hours
Private Study:	3 hours
Tutorial:	1 hour

2.3 Laboratory Session

2.3.1 Resources Required

Ensure that Visual Studio is installed on your computer:

2.3.1.1 Event Handling

We've already made use of an event when building our first C# program. Events are the most important element of the .NET framework, at least at the moment, because they are the entry points we provide into our program. When developing an event driven program, we need to consider the range of what a program must do and how we will present the options for functionality to the user. We also need to make sure that the programs we develop are easy to use, and make sense in the wider context of user interfaces. That can be challenging.

Events in the .NET framework work according to what is known as an **observer pattern**. This is a design that permits one part of the program (usually the UI widget we're working with) to send a message whenever a particular thing happens. When we click on a button, the button sends out a message to say 'Someone just clicked on me'. When we move the mouse over the button, it sends out a message saying 'Someone just moved the mouse over me'. This is known as a **dispatch**, and the button itself is an **event dispatcher**.

Often, these dispatches go nowhere because no part of the program is very interested in when particular events occur. In order for us to indicate an interest in a particular event on a particular control, we must register a **listener**. Each **event dispatcher** is responsible for maintaining a list of all the other parts of the program that are listening for the event, and then when the event occurs a message is sent to each of these in turn.

When we double click on a GUI component in the builder, we are telling Visual Studio to register our form as a **listener** for the button. There is code that is automatically generated when this happens, but we don't need to work with it directly. Visual Studio tells the Button 'When someone clicks on you, make sure you tell the form to trigger the code with the name we provide'. In most cases for this, it is `<name_of_button>_click`.

The combination of the listener and the function name to be called is known as a **delegate**. This delegate defines where the **handler** for an event is to be found. So, a GUI component **dispatches** an event to all its **listeners**, each of which has registered a **delegate** that works as an event **handler**. It is the handler that contains the code that works in response to the event.

2.3.1.2 An Event Context

We often don't know when an event will be triggered when we're writing code. We don't know in what order events will be triggered, and we don't know what may have happened previously in the application. As such, events have to work on the assumption that they can be triggered at any time. This is a considerable difference from the way older programs are written. These would prompt the user for information, in a particular order, process it, and then query the user for more information before showing the output. The **locus of control** in such an application was with the computer.

In event driven programs, users may trigger events before information has been provided, trigger them when the application is not ready for them, or send the wrong information into the application. The **locus of control** in an event driven program is with the user. It is necessary for our events to be **robust** enough to deal with this. They must ensure the information that

comes their way is correct and fully present before proceeding to processing. That needs us to design events with error checking and data handling built right into them.

However, as part of sending a **dispatch**, the event dispatcher will send two pieces of information our way so as to simplify our task. The first of these is the **sender**, which is the GUI object that triggered the event. Usually each of our functions will handle a single event for a single widget, but not always. The second piece of information we get is the **event arguments**, and these contain important information about the state of the system when an event was triggered. For example, if we receive a mouse button event, we may wish to know where the mouse was, what button was clicked, how many times it had been clicked, and so on. All of this is provided as event arguments, and the specific set of information we receive will depend on the type of event itself.

Otherwise, the only things we have available to build the context of our code are the various states of the various widgets in the program, and whatever internal variables we are tracking ourselves. We must think of the events we trigger as small, self-contained **packages** of functionality.

2.3.1.3 Manually Registering an Event Listener

To see all of this working, let's look at how we manually register events in C#. For this, we'll start a new project and register a set of mouse listeners. Create a new form, call it **frmMouse** and double click on the window to bring up the **Load** event sub for the form. This event gets called when a form is first loaded into memory, and can be used to handle anything that must be set up at the start of execution in a program. Events are happening all the time in C# - whenever you need a thing to happen, there is almost certainly an event you can find.

Next, add the following line of code into the stub:

```
private void frmMouse_Load(object sender, EventArgs e)
{
    this.MouseDown += new MouseEventHandler(this.frmMouse_MouseDown);
}
```

MouseDown is the event for which we're adding a listener. We'll talk more about what **new** is doing later in the unit, but for now all we need to know is that we're adding a mouse event handler, and telling it that it'll find the event handler in the current form (that's what **this** does) and it'll be called **frmMouse_MouseDown**. If we wanted to add a similar event listener for another widget (say, something called **cmdPressMe**), we'd use the name of that component instead of **this**. For example, **cmdPressMe.MouseDown**.

We need to add that event handler function too – Visual Studio won't do this for us because we're setting up the relationship ourselves rather than going through the builder. At this point, Visual Studio assumes that we know what we're doing.

```
private void frmMouse_MouseDown(object sender, MouseEventArgs e)
{
}
```

Now, we can make use of this code to trigger something whenever the mouse button is pressed. We can even make it so it does a different thing when different buttons are pressed, although we'll get to that later.

Let's make it so that every time we press the button, the background of our form gets set to a different colour. For that, we'll need to make use of a random number generator and generate random numbers in red, green and blue values. That code looks like this:

```
private void frmMouse_MouseDown(object sender, MouseEventArgs e)
{
    Random r = new Random();
    this.BackColor = Color.FromArgb(r.Next(0, 256), r.Next(0, 256), r.Next(0, 256));
}
```

Random is the random number generator in C#, and Next gives us the next random number according to a minimum and maximum bound – here, between 0 and 256. FromArgb turns three random numbers from 0-255 into an RGB code such as #43ab28, and we then set the BackColor of the form to that random RGB colour code.

Run this program, and click the mouse a few times, and you will see that the colour changes.

Let's add something now that tracks when we're **moving** the mouse. We're only registering an event here for **MouseDown**, which is a button click. If we want something to happen when the mouse moves, we register a new handler. Fittingly, this one goes on **MouseMove**. To begin with, let's just make it change colour again every time we move the mouse. We'll pretty quickly want to change that though:

```
private void frmMouse_Load(object sender, EventArgs e)
{
    this.MouseDown += new MouseEventHandler(this.frmMouse_MouseDown);
    this.MouseMove += new MouseEventHandler(this.frmMouse_MouseMove);
}

private void frmMouse_MouseDown(object sender, MouseEventArgs e)
{
    Random r = new Random();
    this.BackColor = Color.FromArgb(r.Next(0, 256), r.Next(0, 256), r.Next(0, 256));
}

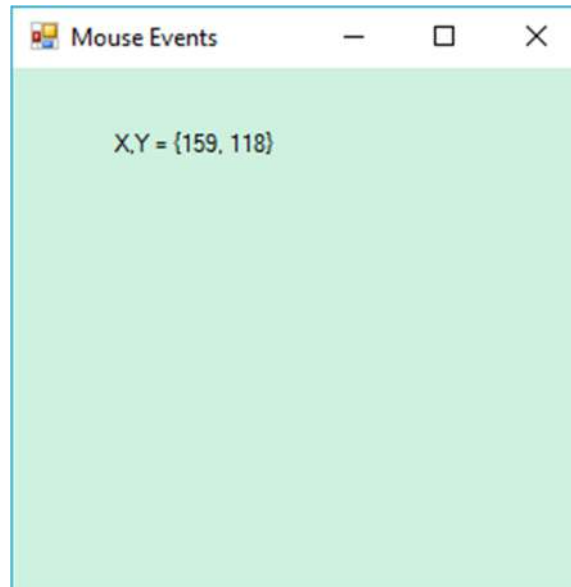
private void frmMouse_MouseMove(object sender, MouseEventArgs e)
{
    Random r = new Random();
    this.BackColor = Color.FromArgb(r.Next(0, 256), r.Next(0, 256), r.Next(0, 256));
}
```

Instead, let's see what we can get out of the event arguments. Let's add a label (lblMouse) on the form, and we'll update it every time the mouse moves by getting the X and Y co-ordinates of the mouse when the event is triggered.

```
private void frmMouse_MouseMove(object sender, MouseEventArgs e)
{
    lblMouse.Text = "X,Y = {" + e.X + ", " + e.Y + "}";
}
```

Click the button and the colour will change. Move the mouse and the label will change. These are two completely independent event handlers doing small things, each contributing to the overall function of the program. That is how event handling works when building a complex

program. Every part does something self-contained, and working together they accomplish the goals of the system.



This raises the obvious question – ‘how do we know what events we’ll need’? For that, we need to consider some **program design**.

2.3.1.4 Paper Prototyping

When working out how a visual program should function, it’s important to spend a bit of time putting together a **sketch** of how you expect it to look. We do this by providing a drawn version of the interface you are proposing, making use of annotations to explain what should happen when the user interacts with the system. This is usually a less costly process than building the program, and permits rapid changes in collaboration with your expected users.

This is part of a philosophy of development called **paper prototyping**, and the idea behind this is that you use very crude, **low fidelity** sketches of how you expect a program to work. It’s very quick and very easy to draw them out, put them in front of people, and find out what they think. You can use this to quickly and effectively explore the design space of an application and make sure that the flow of the program is sensible. A paper prototype should not take you a long time to put together – it should be rough enough that you won’t hesitate for a moment before ripping it up and starting again. You can see some examples of paper prototyping here: <https://speckyboy.com/10-effective-video-examples-of-paper-prototyping/>.

As part of the documentation you end up putting together for your C# program, you’ll be encouraged to develop paper prototypes for any of the programs you develop. You’ll be likewise encouraged to sit down with your classmates and ask for their views. This is done through a **Wizard of Oz simulation**. Consider the following paper prototype:

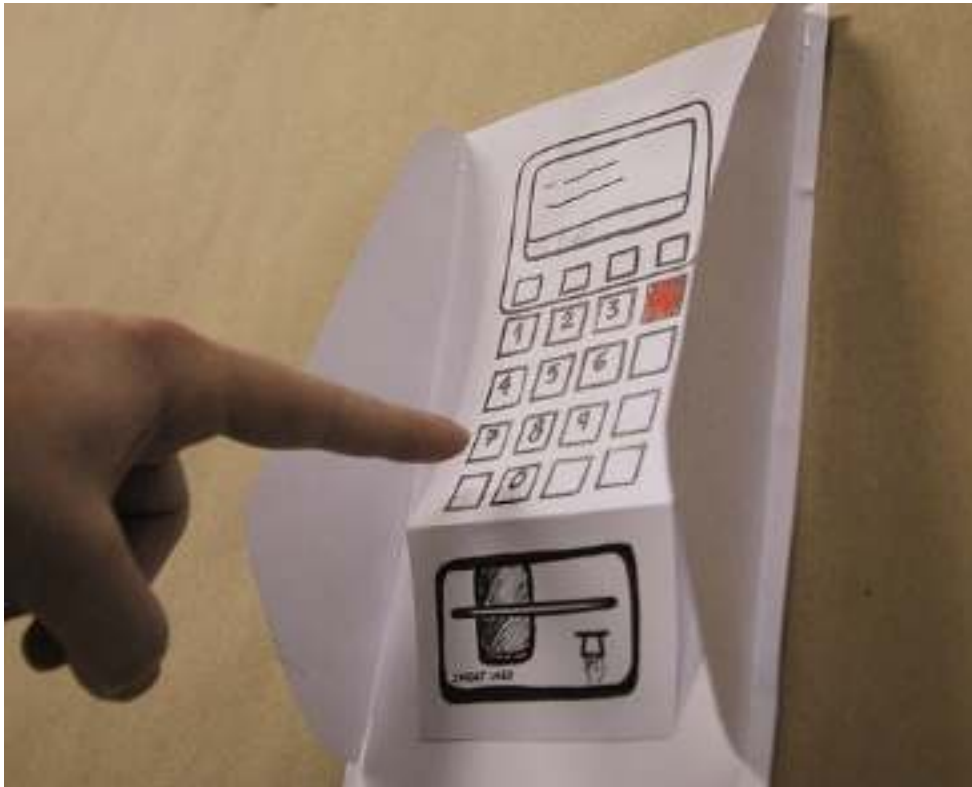


Figure 1 - <https://www.flickr.com/photos/rosenfeldmedia/3978126963>. CC BY 2.0

When testing such a prototype, a set of tasks are provided to the user outlining the key things a user might be expected to do. For the cash machine above, these might include:

1. Check your balance, and ask for a printed statement.
2. Withdraw £50 after checking your balance.
3. Check your balance and leave without withdrawing money

Obviously, the paper prototype will not do anything by itself, and so you as the designer or developer step in and role-play the part of the computer. As they type into the pad, you make whatever noises would be expected. When the screen should change, you swap in the displays that would be shown. If a user's actions would take them to a different interface, you'd swap in the paper prototype for that interface. This is the **Wizard of Oz** procedure – it is **you** that is the wizard behind the curtain.

This process is important, because it allows you to see what makes sense to your users, what doesn't, and what functionality is missing. This in turn allows you to refine your prototype, test it iteratively with other people, and settle on the design of an application before you ever write a single line of code.

You can find a set of templates for paper prototyping online if you want to ensure things look the best they can, but my advice is to simply grab a piece of paper of the right scale and draw it out with as little concern as possible. See <https://www.boardofinnovation.com/resources-tools-for-paper-prototyping/> for some useful resources.

Before beginning any complex user interface, of the kind we'll see later in this unit, do some paper prototyping and make sure you understand the workflow of your own program before you give it to anyone else to use. You might be surprised how far your expectations are from the reality of your users.

2.4 Tasks

Task 1

Put together a paper prototype for an application that mimics a simple word processor. It should have facilities for loading, saving, typing, changing fonts and font sizes. It should permit for images to be inserted, for different text styles and headings to be applied, and for standard text manipulation activities (cut, paste, find, replace and so on) to be performed.

Once you have this paper prototype in place, test it with someone and get them to work through the following tasks:

1. Type their name
2. Apply a heading to a chunk of text
3. Load a document (which you have provided as a separate prototype)
4. Search through that document for a particular string of text
5. Replace that text with a different piece of text
6. Spell check

Make notes of how they respond to your design, and when you are done design a second draft of the interface and test it with a **different user**. Repeat this one more time. Make a note of the evolution of the design, and what changes you made, as part of a change log.

Task 2

Create a new Visual Studio project. Making use of the mouse handling code we have outlined above, implement a program that shows a single button that changes location every time the mouse cursor moves over it. You can change the position of a GUI component at runtime with the following code:

```
<component>.Location = new Point (<x_coordinate>, <y_coordinate>);
```

Once you have it moving randomly, have it move **away** from the cursor as if it's being chased.

Task 3

Making use of the program you developed in Task 2, add a label that changes colour every time the mouse moves over it. It should change **only** when the mouse moves over the label, not just when it moves across the application.

Task 4

Making use of the DoubleClick event and the program you developed in Task 3, have the label return to its original colour when the user double clicks anywhere on the form.

Task 5

Making use of the program you developed in Task 4, and using the MouseWheel event, have the label grow in size whenever the mouse wheel is moved up, and shrink when it is moved down. The Delta property of your event arguments will help here. To change the size of a component, you can use the following code:

```
<component>.Size = new Size (<length>, <height>);
```

2.5 Private Study Exercises

Exercise 1

Referring back to the paper prototyping you did for Task 1 of the Laboratory Session, write a reflective log on how people responded to your original design, and how the design evolved as you tried it with additional users. Discuss whether you think it would be better to test the iterated design three times with one user, or once each with three different users.

Exercise 2

Draw a diagram of the various parts involved in an event being triggered and handled in C#. Annotate this diagram with reference to your program in Task 3 of the Laboratory Session.

Exercise 3

The process for setting location and size in Tasks 2 and 5 of the Laboratory Session is more complicated than just changing X and Y values. This is because they are **read only properties**. Research why read only properties exist, and outline your findings here.

Exercise 4

Outline the benefits that come from making use of a **Wizard of Oz** simulation when developing paper prototypes.

Exercise 5

Explain the reasoning behind paper prototypes being a documentation format that can be ripped up when no longer needed. What is the likely impact on the design process of prototypes being so easy to abandon and modify?

Topic 3: Programming Structures (1)

3.1 Learning Objectives

On completion of this topic, students will be able to:

- Make use of selection in C#;
- Apply conditional logic to programs;
- Make effective use of switch statements;

3.2 Timings

Lectures:	1 hour
Laboratory Sessions:	5 hours
Private Study:	3 hours
Tutorial:	1 hour

3.3 Laboratory Session

3.3.1 Resources Required

Ensure that Visual Studio is installed on your computer:

3.3.1.1 Selection Structures

Over the course of the past two topics we have looked at making use of events to build C# programs. However, we've had to do this in a limited way, through the use of **sequential statements**. When we trigger an event, our code currently works from the beginning of the associated functions stub and executes each line in turn until it gets to the end. That's fine, as it stands, but we should expect more of our programs. We want to be able to alter the **flow of execution** so that we can add in branches, or loops, that let us selectively execute code or repeat it a certain number of times. In this topic, we're going to look at how we add branches to our flow of execution, providing code statements that execute, or not, depending on the state of other parts of the program. This opens up a lot of potential computer programs to us that until now we simply couldn't write.

3.3.1.2 The If-statement

The if-statement is used to change the flow of execution through a program by making a section of the code dependant on a particular condition elsewhere. For example, 'if user has ticked this checkbox, do something', or 'if there is text in a text box, do something with it'. These are examples of conditional statements - the performing of an action is dependent on a certain situation being true. If they are not true, the action shouldn't execute. It lets us create *branches* in our flow of execution, giving multiple potential paths through a program. The path we take is dependent on how that condition is evaluated at run-time – that is to say, **when the program is running**. This is different from **design time**, which is when we're writing the program. If-statements let us write programs that respond to information we won't have available as we write our code.

An if-statement has a particular syntax, and it looks like this:

```
if (something_to_check)
{
}
}
```

The 'something_to_check' part here is known as a **condition**, and that's the trigger for whether the code between the braces will function. For this, we need to assess whether the condition **evaluates to true** or **evaluates to false**. If it is false, the program will simply skip over the code in the braces. This introduces us to a new kind of variable - the **boolean**. This is a variable that has one of two values – true, or false:

```
Boolean something_to_check;

something_to_check = true;

if (something_to_check)
{
}
}
```

A condition in an if-statement is looking for a boolean value, although it need not be provided as a variable. This will usually be handled as a kind of **logical comparison** in which we compare a

variable against some particular criteria. For example, 'is age less than ten' or 'has this button has been clicked before'?

To explore this, let's create a new program. It's going to be very simple – just a checkbox (chkChecked) and a button (cmdPressMe). It's going to have the following code attached to the button:

```
private void cmdPressMe_Click(object sender, EventArgs e)
{
    if (chkChecked.Checked == true)
    {
        MessageBox.Show("It's checked!");
    }
}
```

Run this application and press the button. It will do nothing if the checkbox isn't checked, but if it is a message box will flash up. This is our first selection structure, and you can see the condition inside the if-statement that drives it. 'If the checked property of chkChecked is true, then...'

Note here that we're using two equals signs together, not just one as we have previously. That's because we're doing something different. A single equals sign is known as an **assignment operator** and it takes the right hand side of the equals and puts it into the left hand side. Two equals is an **equivalence operator** and gives a true or false value depending on whether the left hand side is the same thing as the right hand side. Don't worry, you'll get used to this eventually.

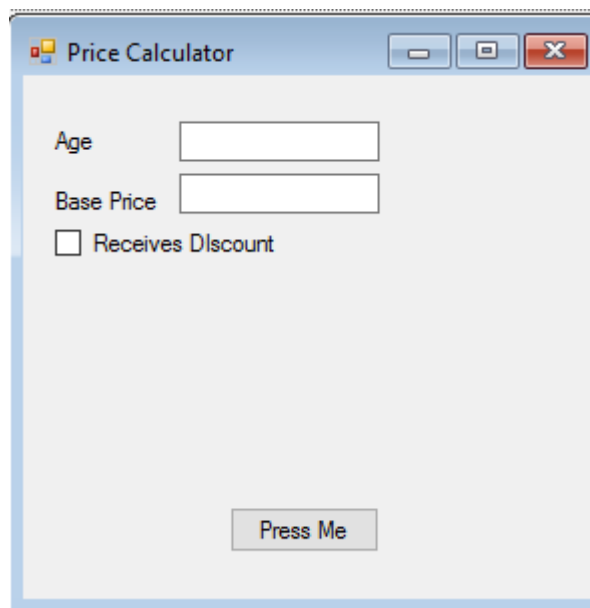
We usually want to do something more interesting here than compare one thing against another for whether they are the same thing. We want to be able to compare things in all kinds of different ways. For that, we introduce the set of powerful **comparison operators** offered by C#.

3.3.1.3 Comparison Operators

There are five primary comparison operators that are used to build convenient and powerful if-statements:

Operator	Example	Will evaluate to true when...
>	a > b	a is <i>greater than</i> b
<	a < b	a is <i>less than</i> b
>=	a >= b	a is <i>greater than or equal to</i> b
<=	a <= b	a is <i>less than or equal to</i> b
!=	a != b	a <i>does not equal</i> b

As limited as these may seem, they allow for very flexible control over what happens when we're working within a particular computer program. When we want to make a comparison, we need to consider what it is that we're actually doing and use the appropriate operator. Let's expand our simple program a little and see how it might work. We're going to add a text box (txtAge), one for a price (txtPrice) and a label for the cost of a ticket to a zoo (lblOutput). The checkbox is going to be used to determine whether someone gets a discount. It'll be called chkDiscount:



Now, we're going to implement a little bit of logic here that takes a base price and calculates what it should be based on various concessions. To begin with, if the age is less than 18, the price should be half. If they receive a discount (indicated by the checkbox being checked) the price is reduced by a further 50% (to a total of 25% of the base price). When we calculate this, we output it to the label.

Before we can do that, we need to talk about how we take the string of a textbox and turn it into a number. We need to **parse** this, using a special piece of code:

```
int price;  
  
price = Int32.Parse(txtPrice.Text);
```

This code will trigger an error if the string inside the textbox can't convert into a number, but we'll talk about how to solve that later. For now, this takes a string of text such as "42" and turns it into the raw number 42. All the standard data types have equivalent processes that we'll see as time goes by. We do the same thing for the age, giving us two integer variables containing the numbers typed into the textboxes.

```
private void cmdPressMe_Click(object sender, EventArgs e)  
{  
    int price, age;  
  
    price = Int32.Parse(txtPrice.Text);  
    age = Int32.Parse(txtAge.Text);  
  
}
```

Now we need to implement our logic – if the age is less than eighteen, the price should be halved:

```
if (age < 18)  
{  
    price = price / 2;  
}
```


And if there is a discount, we half again:

```
if (chkDiscount.Checked == true)
{
    price = price / 2;
}
```

And then finally we output that to the label:

```
lblOutput.Text = "" + price;
```

Notice that we're doing this in a slightly awkward way – this is the reverse of the `Int32.Parse` process. Here we're adding an integer number onto a string to create a string holding that number, which we can then slot into the text property. Put this all together and you get the following function:

```
private void cmdPressMe_Click(object sender, EventArgs e)
{
    int price, age;

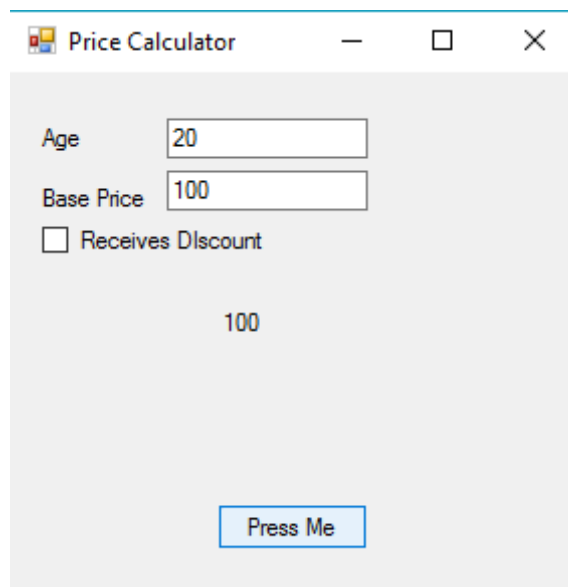
    price = Int32.Parse(txtPrice.Text);
    age = Int32.Parse(txtAge.Text);

    if (age < 18)
    {
        price = price / 2;
    }

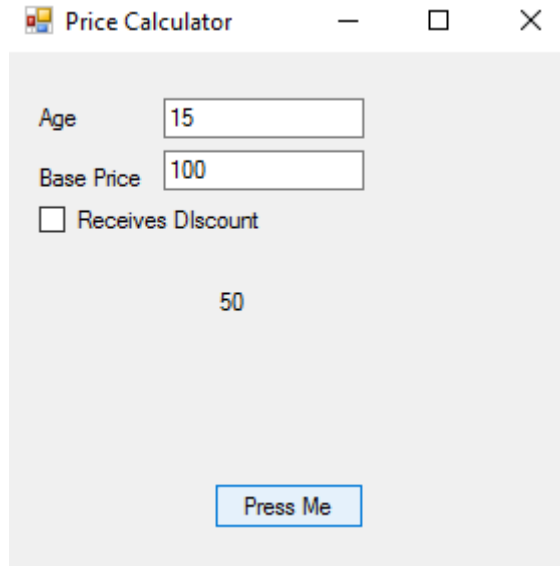
    if (chkDiscount.Checked == true)
    {
        price = price / 2;
    }

    lblOutput.Text = "" + price;
}
```

Run this program and you'll see that it now applies the age and discounts appropriately, based on what you select. If no discount should apply:

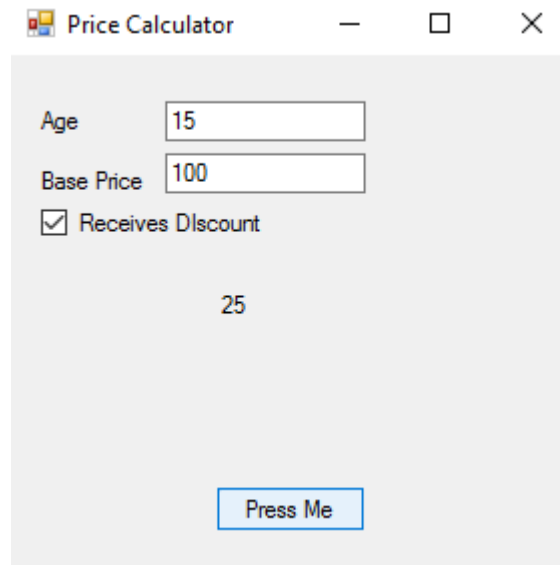


If age alone should handle the discount:



A screenshot of a Windows-style application window titled "Price Calculator". It contains two input fields: "Age" with the value "15" and "Base Price" with the value "100". Below these is an unchecked checkbox labeled "Receives Discount". In the center of the window, the number "50" is displayed. At the bottom is a blue button labeled "Press Me".

And if both should apply:



A screenshot of the same "Price Calculator" window. The "Age" field is "15" and the "Base Price" field is "100". The "Receives Discount" checkbox is now checked. The calculated price in the center is "25". The "Press Me" button remains at the bottom.

Already you can undoubtedly see how useful this is going to be to us as we go along.

3.3.1.4 If-Else

Sometimes we don't want to model a piece of code that is purely conditional - sometimes we want to choose between one of two mutually exclusive options. 'If this is true, do something. Otherwise, do something different'. The if-statement comes in a number of different flavours, and the first of these is an if-else structure which lets us do exactly that. Let's say for our example above we want it so that anyone who doesn't have a discount is in fact going to pay double. That means we're either going to half the price, or double it – one or the other, never both.

```
if (chkDiscount.Checked == true)
```

```

{
    price = price / 2;
}
else
{
    price = price * 2;
}

```

This creates a branch of two different courses of action, but sometimes even that isn't enough. The good news is we can extend the if-statement indefinitely through the use of the **else if** construction. Think of the if-else like this:

```

if condition then
    do something
otherwise
    do something else

```

The else-if extends this:

```

if condition then
    do something
otherwise if a second condition is true
    do something else

```

So if we wanted to offer a wider range of discounts, we might want to offer a discounted rate for those of 65 years old or older. We can do that:

```

if (age < 18)
{
    price = price / 2;
}
else if (age >= 65)
{
    price = price / 4;
}

```

We can extend this as much as we like to offer as many different conditions as we need – as many branches of execution as we could ever want. We do this by chaining together else-if-statements until we're done, potentially ending with one final **else** at the end. Let's say we want to offer a rate for very young children too:

```

if (age < 5)
{
    price = price / 10;
}
else if (age < 18)
{
    price = price / 2;
}
else if (age >= 65)
{
    price = price / 4;
}

```

Choosing between an if- and an else-if statement is a powerful tool you have available to shape the way your program responds to events beyond your control at design time.

3.3.1.5 Compound Conditionals

Sometimes we want to be able to base an if-statement's execution on more than one condition - for example, if we want to check that a variable falls within two values. Unfortunately, C# cannot make sense of valid mathematical expressions such as:

```
if ( 10 < x > 100 ) {  
}
```

C# won't know how to interpret that, and so it will give you an error. Instead, you must simplify it a little and break it up into two separate conditions. 'If X is greater than 10' and 'if X is less than 100'.

To handle this, we make use of C#'s library of *logical operators*. These can be used to join single conditionals together into a **compound condition** which has more than one part to it.

The first of these we will look at is the 'and' operator, and it takes the form of two ampersands side by side: &&. We would write the check above as follows:

```
if ( x > 10 && x < 100 ) {  
}
```

When C# encounters a compound conditional, it tries to assess it as a whole - the code belonging to the if-statement will only be executed if every condition in the compound evaluates correctly. This is where it starts to become complicated. The logical operators that we use indicate to C# how each of the conditionals making up the compound must evaluate in order for the whole to be true. There are two main kinds of logical operators - AND, and OR. There are others (such as the Exclusive Or (XOR) operator, but we won't look at these now).

Logical Comparison	Symbol
AND	&&
OR	

So let's make our program a little more powerful still. We're going to make it so the discount can only apply if someone is **not** permitted one of the other discounts. It's available only to someone that is older than 18 and younger than 65:

```
if (chkDiscount.Checked == true && age >= 18 && age < 65)  
{  
    price = price / 2;  
}  
else  
{  
    price = price * 2;  
}
```

We can build compounds of any complexity through chaining together logical comparisons. In this way, our selections can be truly adaptive taking into account multiple conditions that need to be considered as part of a single decision.

3.4 Tasks

Task 1

Create a new project. Create a new form, consisting of four radio buttons offering the user a choice of addition, subtraction, multiplication and division. Add two text boxes to allow users to input two numbers. Have the correct calculation performed when the user clicks a button and the result displayed to the user.

Task 2

Making use of the program you developed in Task 1, add some validation to your textboxes. If the user presses the button without having entered anything into the text fields, have it flash up a message box reminding them to do so.

Task 3

Making use of boolean variables to track when an arithmetic operation has been performed, modify your program from Task 2 so that it needs the user to change their arithmetic operation each time they press the button. This will mean they can't perform two additions in a row, or two subtractions, but an addition followed by a subtraction, followed by an addition would be fine.

Task 4

Making use of the program you wrote in Task 3, add a set of labels that will correctly output the following information when the button is pressed:

- Which of the numbers is the largest
- Whether the result of the indicated sum is smaller or greater than the first number
- Whether the result of the indicated sum is smaller or greater than the second number

Task 5

Create a new program. This should have a single textbox and a button. The textbox should accept a number between 0 and 100, making use of validation to ensure users don't enter an invalid number. When the button is pressed, it should output an alphabetic grade according to the following logic:

- A grade higher than 70 counts as an A
- A grade higher than 60 and less than or equal to 70 counts as a B
- A grade higher than 50 and less than or equal to 60 counts as a C

- A grade higher than 40 and less than or equal to 50 counts as a D
- A grade higher than 30 and less than or equal to 40 counts as a E
- Anything less than or equal to 30 counts as an F

Task 6

Modify the program you wrote for Task 5 to provide a series of three text boxes. Upon pressing a button, an alphabetic grade should be presented for each. It should also output which of the grades was highest.

3.5 Private Study Exercises

Exercise 1

Create a paper prototype of an application that would permit for specifying a computer based on radio buttons for CPU, memory, hard-drive, and graphics card. Add checkboxes for other optional peripherals. Outline the logic that would be needed to turn these selections into C# code that calculated the price of the computer accordingly.

Exercise 2

Research the concept of a **truth table**, and develop truth tables for the following logical comparisons:

- A AND B
- A OR B
- A AND B OR C
- A OR B OR C
- A AND B AND C
- A AND B AND NOT C

Exercise 3

Research the **switch statement**, and outline its relationship to the if-else-if structure outlined above. Consider when you would choose to use one over the other.

Exercise 4

Research the term **lazy evaluation** as it is related to logical comparison, and consider what it implies about the logical operations you examined in Exercise 2.

Exercise 5

Consider the evaluation you did in Task 6 of the Laboratory Session to compare three numbers, identifying which is the largest. Using nothing more than the comparison operators we have discussed in this section, how might you determine which is the largest of ten numbers without needing compound conditionals that assess all possible combinations?

Topic 4: Programming Structures (2)

4.1 Learning Objectives

On completion of this topic, students will be able to:

- Make use of iteration in C#;
- Apply conditional logic to unbounded loops;
- Understand the role of repetition on shaping program flow;

4.2 Timings

Lectures:	1 hour
Laboratory Sessions:	5 hours
Private Study:	3 hours
Tutorial:	1 hour

4.3 Laboratory Session

4.3.1 Resources Required

Ensure that Visual Studio is installed on your computer:

4.3.1.1 Repetition Structures

In the last chapter we looked at the power of selection structures to shape the flow of execution through a program – these permit us branches that we can follow, or not, to have the program respond to the information it has at run-time. We often don't know at design time what the state of individual variables in a program might be and so we make use of these structures to handle uncertainty. However, often we don't want to simply choose between two or more course of action – sometimes we want to repeat the **same** course of action several times. This is where **iteration** enters the picture.

4.3.1.2 The For Loop

The For Loop is an example of a structure known as a **bounded loop**. Bounded loops repeat a certain section of code a known number of times – perhaps not a value known when we write the code, but known when the loop is reached in the running program. For example, we might have a textbox that contains a number, and that number will determine how many times a piece of code should execute. We don't know when we write the program what people will type into the textbox, but when the loop triggers we'll know what's in there. Bounded loops work on a counter, which we compare against a condition to see if we should still be repeating the code. An individual execution of the code that belongs to a loop is called an **iteration** of that loop.

The For Loop is the most complicated programming structure we have looked at yet. It has the following basic structure:

```
for (initialisation; continuation; upkeep)
{
    // Code to repeat
}
```

When our program reaches the for loop, these special sections are executed in a particular order:

1. Initialization occurs at the point we reach the loop, and never again. We use this to set up whatever counters we plan to use to track the number of times we've repeated.
2. Continuation is checked at the start of each iteration of the loop. Making use of the comparison operators we saw in the last chapter, we check to see if our counter meets some threshold. If it does, the loop continues. If it doesn't, the loop ends.
3. Upkeep is executed at the end of each iteration of the loop, and is used to update the counters we're using.

Let's look at this with an example – we're going to provide two textboxes (txtBase and txtPower) in a program, and we're going to compute the power of one to the other when we press a button (cmdCalculate). To do this, we take a number and multiply it by itself the number of times indicated by the power. 2^2 is calculated as 2×2 for a total of four. 2^3 is $2 \times 2 \times 2$ for a total of eight. This is a perfect example of a situation in which we might need to make use of a loop to handle a calculation.

First of all, let's look at the **pseudocode** of this process:

1. Get the base number
2. Get the power to which the base number is to be raised
3. Set the current total to the base number
4. Set the number of times to repeat to be the power minus one
5. Repeat the following number of times equal to the number of times to repeat
 - a. Set the total to be the total times the base number
6. Output the result

Putting that together in code would give us the following:

```
private void cmdCalculate_Click(object sender, EventArgs e)
{
    int baseNum, powerNum;
    int ans;
    int numRepeats;

    baseNum = Int32.Parse(txtBase.Text);
    powerNum = Int32.Parse(txtPower.Text);

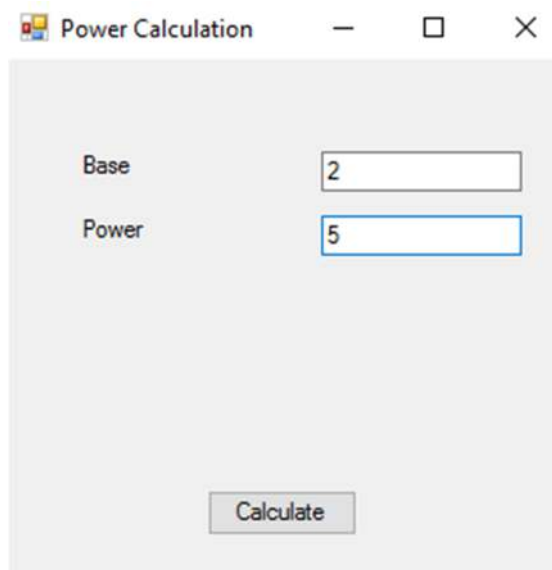
    ans = baseNum;

    numRepeats = powerNum - 1;

    for (int counter = 0; counter < numRepeats; counter++)
    {
        ans = ans * baseNum;
    }

    MessageBox.Show("The total is " + ans);
}
```

Imagine our user sits down in front of our application and hits the button:



The screenshot shows a standard Windows application window with the title bar 'Power Calculation'. Inside the window, there are two labels, 'Base' and 'Power', each followed by a text input box. The 'Base' box contains the number '2' and the 'Power' box contains the number '5'. At the bottom center of the window is a button labeled 'Calculate'.

We're looking here for two raised to the power of five. Running through the initial setup of our code will give us the following set of variables:

baseNum	2
powerNum	5
Ans	2
numRepeats	4
Counter	0

Let's look at that loop in more detail:

```
for (int counter = 0; counter < numRepeats; counter++)
{
    ans = ans * baseNum;
}
```

The first thing that we do when we enter the loop is create an integer called counter, and set it to 0. This happens once, and never again. When the loop has finished, the counter variable will disappear – it exists only as long as this loop is running. That wouldn't have been true if we created it in the same place as the other variables, but as long as we don't need it later it's safe to declare it here.

Next, we evaluate the **condition** of the loop – here, the format is exactly the same as with the if statements we saw in the previous chapter. To begin with our counter is zero, so the **continuation** condition of the loop is 'is the counter less than the numRepeats'. Or, 'is zero less than four?'. It is, and so the code within the braces triggers. This then resolves down into:

```
ans = 2 * 2;
```

At the end of the first iteration of the loop, ans is 4.

We then move into the **upkeep** phase, and this adds one to the counter. After the first execution of the loop, this is what our variables contain (changes marked in bold):

baseNum	2
powerNum	5
Ans	4

numRepeats	4
Counter	1

Because it is a loop, the program now goes back to the continuation condition and asks 'is one less than four?' It's still true, and so the code triggers one more time, changing Ans to 8. At the end of that, upkeep is triggered which increases the counter by one again:

baseNum	2
powerNum	5
Ans	8
numRepeats	4
Counter	2

This continues until counter is increased to four in the upkeep. At that point, we go back to the continuation condition and ask 'Is four less than four?' It's not and so the loop terminates and control is handed back to the line of code that follows it.

4.3.1.3 The While Loop

There is a second kind of loop we often need to use – the **unbounded loop**. The bounded loop works on the assumption we know how many times to repeat, but that's not always true. Sometimes we only know we should stop repeating when a thing happens. For example, we might ask our user 'do you want to continue?' and have no idea how many times they'll say yes. The unbounded loop permits us to loop based only on a condition, rather than a counter.

Strictly speaking, both kinds of loops can be used for all kinds of situations. The for loop though has fixed sections for counter management and if you wish to use a while loop to handle bounded conditions you'll need to do all the continuation and upkeep calculations yourself. It's best to get into the habit of using the right tool for the right purpose.

Here, we're going to change our code a little – instead of asking up front how many times we should raise to a power, we're going to ask the user. For that, we'll also need to look at how to provide the user with a 'yes/no' dialog instead of a simple message box. When they press 'Yes' we'll raise to another power. When they press 'no' we'll output the result.

```
private void cmdCalculate_Click(object sender, EventArgs e)
{
    DialogResult response;
    int ans;
    int baseNum;
    int times = 1;
```

```

baseNum = Int32.Parse(txtNum.Text);

ans = baseNum;

response = MessageBox.Show("Should I raise to another power?", "Keep on Powering?", MessageBoxButtons.YesNo);

while (response == DialogResult.Yes) {
    ans = ans * baseNum;
    times = times + 1;
    response = MessageBox.Show("Should I raise to another power? Currently " + baseNum + "^" + times, "Keep on Powering?", MessageBoxButtons.YesNo);
}

MessageBox.Show("Raised " + times + " to " + ans);
}
}

```

The DialogResult is what we use to determine which button was pressed – you can explore what MessageBoxButtons offers if you want to see other options for presenting these confirmatory dialogs to users. We ask the user what we should do, and then we enter a loop – note there's no counter here, we continue until the user presses something other than the yes button. The while loop checks this continuation clause right at the very start, so we need to have it prepared for when it checks. Inside the loop, we ask the question again and again until the user decides to stop. We don't know when that's going to happen.

Note here the code is a little clumsy, repeating the same MessageBox code twice. There's a variation of this called the **do while** loop. The only difference is that it does the code within the brackets at least once before checking the continuation. It can be used to tidy up some of your code, like so:

```

do
{
    response = MessageBox.Show("Should I raise to another power? Currently " + baseNum + "^" + times, "Keep on Powering?", MessageBoxButtons.YesNo);

    if (response == DialogResult.Yes)
    {
        ans = ans * baseNum;
        times = times + 1;
    }
}
while (response == DialogResult.Yes);

```

For unbounded loops, if we want to iterate zero or more times, we use a while loop. If we want to iterate one or more, we use a do-while loop. Which is most appropriate will depend on your specific scenario.

4.3.1.4 Conclusion

Loops have just put a lot of additional power into your hands – structurally they are similar to the selections we looked at in the previous chapter, but provide additional opportunities for seriously impacting on the flow of your project. As you can see in the example above, they can even be placed inside each other – you can have if statements inside a for statement, for statements within a while loop, and any combination besides. This is called **nesting** and we'll have cause to see lots of examples of this as time goes by. In the meantime, mastering loops is going to offer you a much greater opportunity to create meaningfully interesting projects – that's especially true when we come to look at **arrays** in a couple of chapters time.

4.4 Tasks

Task 1

Create a new project. Allow the user to enter an integer number. Calculate the sum of all the numbers that precede it – so, 4 would be $4 + 3 + 2 + 1$. 5 would be $5 + 4 + 3 + 2 + 1$, and so on.

Task 2

Create a new project. Allow the user to enter an integer number. The program should output the multiplication table of the provided value to a label on the form.

Task 3

Create a new project. Allow the user to enter two numbers. The first will represent the upper range of numbers, so as an example 'zero up to 10' or 'zero up to 5'. Iterating over each of the numbers leading up to the first provided number, output all of those that are cleanly divisible by the second without leaving a remainder. You will need to use the modulus arithmetic operator (%) for this – it gives the remainder of a division.

Task 4

Create a new project. This should have a textbox and a label large enough to cover the majority of the form. When the user enters a number and presses a button, it should output a triangle of stars made up of the number of lines indicated by the user. If they enter 4, it should display the following:

```
*
**
***
****
```

Task 5

Making use of the program you developed in Task 4, modify it so that once it reaches the number indicated by the user it starts to draw lines of decreasing length, like so:

```
*
**
***
****
***
**
*
```

Task 6

Create a new project, and write the code that will compute the factorial of a number provided by the user.

4.5 Private Study Exercises

Exercise 1

Making use of the format shown above for tracking variables in a loop, write out the variable states at each iteration of the following loop:

```
int num = 2;
int tmp = 5;

for (int i = 0; i < tmp; i = i + 2)
{
    num = num + i;
}
```

Exercise 2

Making use of the format shown above for tracking variables in a loop, write out the variable states at each iteration of the following loop:

```
int num = 2;
int tmp = 5;

for (int i = 10; i >= 0; i = i - 1)
{
    num = num - i;
}
```

Exercise 3

Explain the difference between **bounded** and **unbounded** loops, giving an example of usage for each. Explain the circumstances under which one might be preferred over the other.

Exercise 4

Making use of the format shown above for tracking variables in a loop, write out the variable states at each iteration of the following loop:

```
int num;

for (int i = 0; i < 5; i++) {
    for (int j = 0; j < 5; j++) {
        {
            num = i * j;
        }
    }
}
```


Exercise 5

Investigate the Sieve of Eratosthenes – research this on the internet, and describe what it does. Outline in pseudocode how it might be implemented, and identify which elements of the algorithm are currently not possible to implement with the techniques discussed to date.

Exercise 6

In the example program above calculating powers using a for loop, we don't use the number provided by the user – we use one less than that. Provide an explanation of why, and suggest some alternate for loops that would have correctly calculated the number.

Topic 5: Object Orientation (1)

5.1 Learning Objectives

On completion of this topic, students will be able to:

- Understand the structure of object orientation;
- Create and make use of objects and classes;
- Employ encapsulation for data protection;

5.2 Timings

Lectures:	1 hour
Laboratory Sessions:	5 hours
Private Study:	3 hours
Tutorial:	1 hour

5.3 Laboratory Session

5.3.1 Resources Required

Ensure that Visual Studio is installed on your computer:

5.3.1.1 Object Orientation

We have been using object orientation all the way through this unit, but we have not yet actually taken the time to look at what objects are in theory and in practice. These serve as the fundamental building blocks of modern OO programming languages – hardly surprising when we consider that OO stands for Object Orientation.

The concept can be quite difficult to understand at the beginning. Object orientation involves breaking a problem down into separate components in a way that is contradictory to how we would approach such a thing in our minds. Despite this, object orientation is a powerful programming paradigm and the tools it provides can lead to the development of richer and more elegant programming solutions than is possible with other, earlier development frameworks.

5.3.1.2 The Theory of Object Orientation

Consider the code we've using to date to create the programs we build:

```
namespace WindowsFormsApplication5
{
    public partial class FormClassExample : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

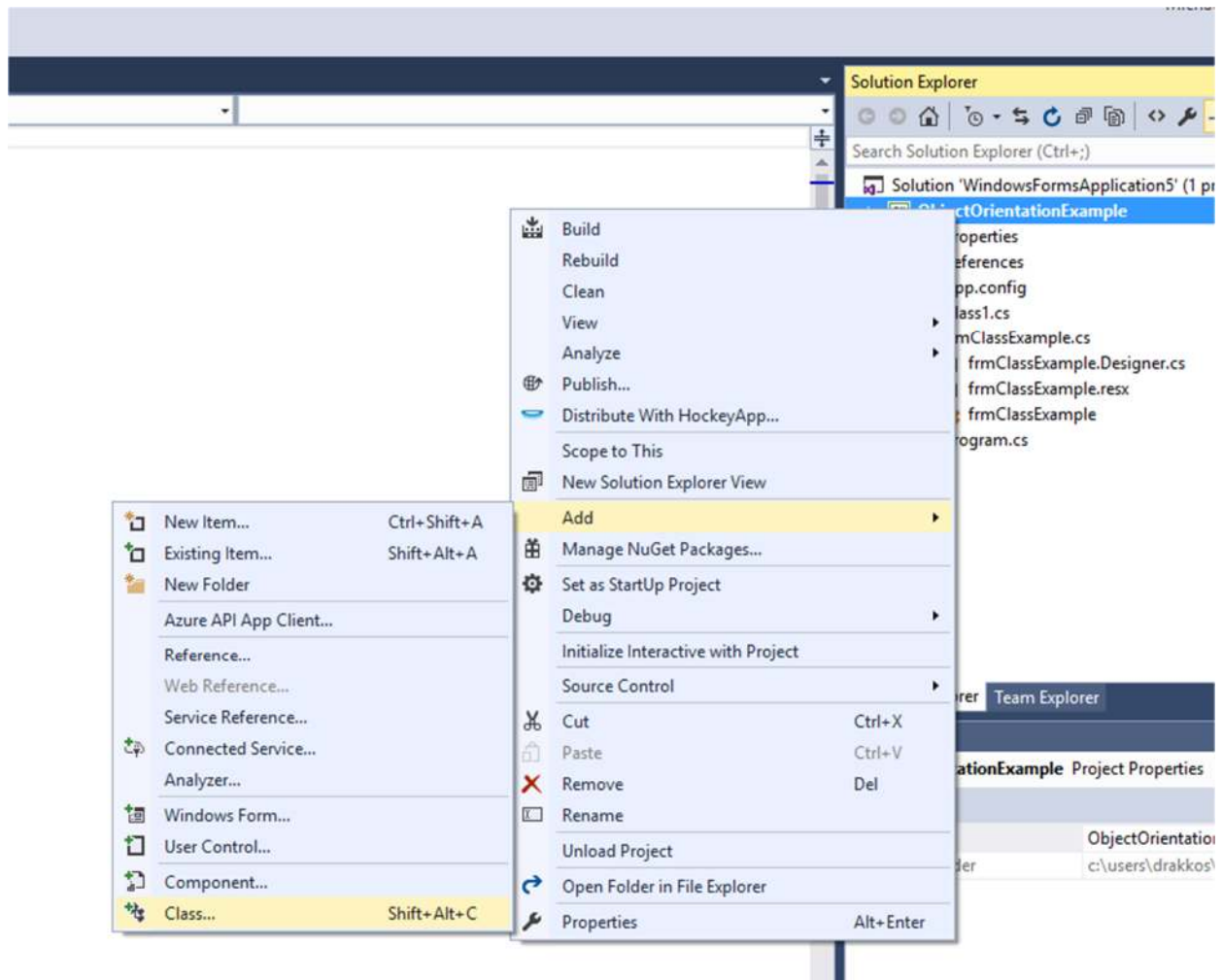
        private void Form1_Load(object sender, EventArgs e)
        {
        }
    }
}
```

We haven't made a big deal of it, but what we're doing here is creating a **class**. In object oriented programs, a class is a blueprint from which we derive instances of an **object**. Consider the chair on which you are (perhaps) sitting – there may be many chairs like this, but they all stem from a central design. That design is the class, and the specific chair you are sitting on is an object. There may be variations in objects – you may have chairs with the same design that have different colours, sizes, materials, and so on. It is our job as programmers to write classes that are sufficiently flexible that they can adapt to the real world as needed.

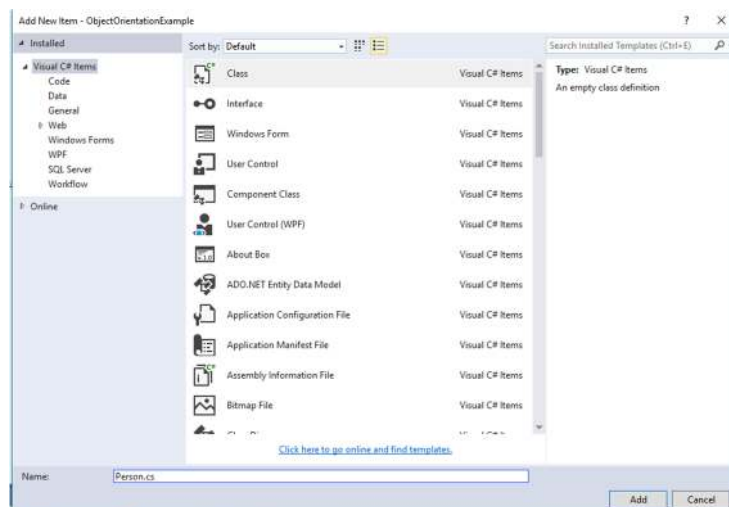
When we create the class above, what we are doing is building a template for a Windows application. When we run the program, this class gets **instantiated** into an object. Usually we only have one of these, but there would be nothing to stop us having two, three, or three hundred objects **instantiated** from that simple class.

This is a difficult concept to understand without a concrete example, so let's look at that. For this, we'll need to add a new kind of file to our project. We do that from the **solution explorer** – a

part of the IDE we haven't had much cause to do anything with up until now. We add a new class by right clicking on the project name, choosing 'add' and picking **class**:



This in turn will bring up the new item dialog. Click 'class' and give it a name. In this case, Person.cs.



Once we've done this, we'll get an empty class that we're going to expand so that it can be used to hold a set of useful data elements. The code we'll get is the following:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ObjectOrientationExample
{
    class Person
    {
    }
}

```

Don't worry too much about the **namespace** here – it uses the name we give our project for this, so it doesn't matter if it doesn't match what we have here. The key element is what's within the braces there – the **class Person** portion of the code. We're about to make that do something very powerful.

5.3.1.3 Classes and Objects

As discussed above, the class is the template and the object is a specific instance of that template. The class tells the object what data and functions it should have, and the object tells the program what **state** the data has. In other words, a class tells an object what it should be, and the object tells the program what it **is**. For our purposes, a person might have a name and an age. That's the information we store about a person. It's only when we make an object that we give specific values to those fields. Let's see how.

```

class Person
{
    private String name;
    private int age;
}

```

Don't worry at the moment about what the **private** part of this means just yet– just remember it should always be there when we add data elements. What we've done here is create two **attributes** that will be used to define what information an object might contain. Think here of an object as a way to create your own data-types that are made up of collections of other data types. Not a string, or a button, but something new that might have different compartments that contain strings **and** buttons. While it's not wholly accurate, it's useful to think of a class as a new data type, of our own design, that has many different drawers in which we might store information. This is a technique known as **encapsulation** – storing the data in a class along with the methods that are going to act upon this data.

However, this information currently isn't accessible to any other part of the program. For that, we need to add in some **accessor** functionality. In some languages this is done by creating a **set** and **get** function for each variable we have, like so:

```

void setAge(int a)
{
    age = a;
}

int getAge()
{
    return age;
}

```

This works perfectly well, but the .NET framework has a better system – **properties**. These let us hook our objects up to the usual IntelliSense¹ that we see when we type code into the IDE. To create a property, we provide the following code:

```
public int Age
{
    get
    {
        return age;
    }

    set
    {
        age = value;
    }
}
```

Note here that we use **public** rather than **private**. This is true of properties and methods, but not of attributes. We'll talk about why later in the unit. We can add a property for each variable we want to expose to the rest of our program. Adding in properties for both our attributes will give us a class that looks like this:

```
namespace ObjectOrientationExample
{
    class Person
    {
        private String name;
        private int age;

        public int Age
        {
            get
            {
                return age;
            }

            set
            {
                age = value;
            }
        }

        public string Name
        {
            get
            {
                return name;
            }

            set
            {
                name = value;
            }
        }
    }
}
```

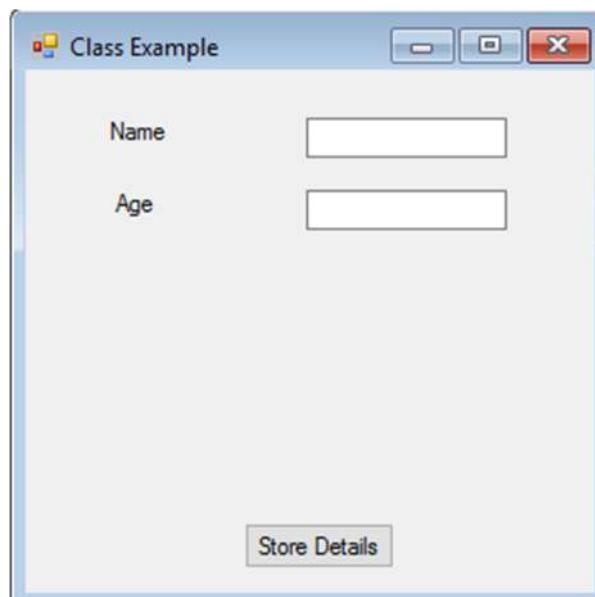
¹ *IntelliSense* is Microsoft's intelligent code completion feature:
https://en.wikipedia.org/wiki/Intelligent_code_completion

The code inside *set* and *get* here can be manipulated too, to change the way the object behaves when we try to query or assign values. This can be a powerful technique and we will have cause to see it in the future.

Now we've got a class of our own – a *Person*. This is now a thing we can use in the rest of the program. So let's do that.

5.3.1.4 Using an Object

Let's go back to our form – we're going to add two textboxes (*txtName* and *txtAge*) fields to this, and a button (*cmdStore*) that lets us store the values within an object of our choosing.



First of all, we're going to create what is called a **class wide variable** – this sits outside of any of our methods, which means it's available to all of them. We define this underneath the class definition:

```
public partial class frmClassExample : Form
{
    Person me;
```

Here, we're saying 'we have a variable called **me** that is of type **Person**'. We wrote that *Person* class ourselves, so C# accepts it without question. In a very real sense, we're taking part in actively constructing the language of C#.

Next, we need to **instantiate** this object. We do this using the **new** keyword, and usually we do it in the load event of the form.

```
private void frmClassExample_Load(object sender, EventArgs e)
{
    me = new Person();
}
```

And then finally, we use the click event of our button to actually store the values in the object we created.

```
private void cmdStore_Click(object sender, EventArgs e)
{
    String name = txtName.Text;
```

```

    int age = Int32.Parse(txtAge.Text);

    me.Name = name;
    me.Age = age;

}

```

Now, we have the name and age from the textboxes stored happily in our object. We can prove it too. Let's add a second button called cmdDisplay:

```

private void cmdDisplay_Click(object sender, EventArgs e)
{
    txtName.Text = me.Name;
    txtAge.Text = "" + me.Age;
}

```

Run this program, and type in a name and age. Press the store button, and then delete the text you had put into the textboxes. Press the display button and you'll see both pieces of data get restored.

5.3.1.5 Methods

In addition to providing attributes for our class, we can also provide **methods**. Methods are functions that exist within a class, and normally permit us to act upon the data within. As noted above, storing the data and the methods that act upon that data together is known as **encapsulation**.

Consider the following scenario. You've got to write a program that tracks the names and ages of several people, and also permits people to add to and subtract from their ages as necessary. Upon instruction the age should increase, or decrease by a set value. Think how we might do that using the tools we discussed before this chapter – we'd need to keep track of two variables for each person, work out how much to increase or decrease ages by, work out which person to do that for and hope that nothing falls out of sync. However, with object orientation we have the object acting as the glue keeping the name and age of someone together. All we need to do is provide a set of **methods** that permit someone to modify the contents. We can do that like so:

```

class Person
{
    private String name;
    private int age;

    public void adjustAge(int amount) {
        age = age + amount;
    }

    public int Age
    {
        get
        {
            return age;
        }

        set
        {
            age = value;
        }
    }
}

```



```

public string Name
{
    get
    {
        return name;
    }

    set
    {
        name = value;
    }
}
}

```

Now, we'll go back into our form and adjust it in several ways.

- We'll add a textbox to store the amount to adjust ages (txtAmount)
- We'll make the program use three objects rather than one
- We'll add a textbox to indicate which object we wish to work with (txtWhich)
- We'll add a button to permit the adjustment (cmdAdjust)

We're going to do this in a somewhat clumsy way to begin with because we haven't yet spoken about arrays – we'll revisit this program in a later chapter and see a better way to accomplish this. For now, we can see a problem with scaling up our program to larger numbers of people – it needs us to do too much work.

First our interface – it's going to look like this:

And then our code. First of all, we need to create three objects instead of one. We'll also change their names to p1, p2 and p3:

```

public partial class frmClassExample : Form
{
    Person p1, p2, p3;

    public frmClassExample()
    {
        InitializeComponent();
    }
}

```

```

private void formClassExample_Load(object sender, EventArgs e)
{
    p1 = new Person();
    p2 = new Person();
    p3 = new Person();
}

```

To choose which object we are going to work with, we're going to give ourselves a function that takes the appropriate text box content, works out which number refers to which object, and returns that object from the function so that we don't need to go looking for it. This is where the clumsy part of the code comes in, and we'll fix it later.

```

private Person getDesiredPerson()
{
    Person whichOne;
    int num;

    num = Int32.Parse(txtWhich.Text);

    if (num <= 1)
    {
        whichOne = p1;
    }
    else if (num >= 3)
    {
        whichOne = p3;
    }
    else
    {
        whichOne = p2;
    }

    return whichOne;
}

```

Next, we need to adjust our display and store methods accordingly:

```

private void cmdDisplay_Click(object sender, EventArgs e)
{
    Person selected;

    selected = getDesiredPerson();

    txtName.Text = selected.Name;
    txtAge.Text = "" + selected.Age;
}

private void cmdStore_Click(object sender, EventArgs e)
{
    String name = txtName.Text;
    int age = Int32.Parse(txtAge.Text);
    int index;
    Person selected;

    selected = getDesiredPerson();

    selected.Name = name;
    selected.Age = age;

}

```

And then finally, we implement the code for permitting the user to adjust the age. Note here the call to `adjustAge` in the selected object – this is how we invoke a method:

```
private void cmdAdjust_Click(object sender, EventArgs e)
{
    Person selected;
    int amount;

    amount = Int32.Parse(txtAmount.Text);

    selected = getDesiredPerson();

    selected.adjustAge(amount);
}
```

Methods like this, along with properties, can be used to give objects powerful functionality. We're going to see that option become very valuable soon.

5.3.1.6 Constructors

We have one last thing to cover before we're done with our initial discussion of objects. Isn't it awkward to have to store all the details in each person before we can do anything else with the program? Wouldn't it be useful if we could give the objects some initial starting values? Well, we can! We can provide each object with something called a **constructor** – a method that's executed when we create the class with the **new** keyword. Let's say we wanted to provide the object with a name and an age when created – we'd add the following constructor to our class definition:

```
public Person(String n, int a)
{
    name = n;
    age = a;
}
```

The thing that defines a constructor is that it has exactly the same name as the class itself (in this case, `Person`) and does not have a return type. Once we provide a constructor like this we no longer create a `Person` using an empty parameter list – instead, we either need to always provide a name and age, or provide a **second** constructor that works with no parameters:

```
public Person(String n, int a)
{
    name = n;
    age = a;
}

public Person()
{
    name = "Unset";
    age = 21;
}
```

This is called **method overloading** and is performed by giving a method the same name as one that already exists, but a different **method signature**. The method signature is made up of the order and type of parameters passed into the function. Here now, the following will work:

```
private void formClassExample_Load(object sender, EventArgs e)
{
```

```
p1 = new Person("Michael", 100);  
p2 = new Person("Pauline", 20);  
p3 = new Person();  
}
```

Run the program with this code and you'll see that we don't need to store data to begin with – we can give ourselves some starting test data that makes the program easier to work with.

5.4 Tasks

Task 1

Create a new project. Create a new class called `StatisticalData`. Give it three integers that can be set from the main program. Give the main program a set of five objects in a similar fashion to that discussed in the chapter.

Implement methods in the class that will calculate the following values from the three integers in each:

- Largest
- Smallest
- Average

Task 2

Create a new project. Create a class called `Book` that stores a title, author name, ISBN, and genre. Write the interface that would permit a user to set and query each of the elements in a set of four of these.

Task 3

Create a new project. Create a class called `Movie` that stores a title, year of release, and runtime. Have the main program store four of these, and permit the user to find the title of the first movie that has a runtime within a certain minimum and maximum range. For example, 'show the first movie that has a runtime between 60 and 120 minutes'.

Task 4

Create a new project. Create a class called `Album` that stores a title, artist and genre of an album. Have the main program store six of these. Permit the user to enter the name of a genre, and output a list of each `Album` that matches that genre.

5.5 Private Study Exercises

Exercise 1

Research the topic of **data hiding**, and explain its relationship to **encapsulation**.

Exercise 2

Research the topic of **encapsulation**, and provide an overview of the reasons as to why it might be desired in an object oriented computer program.

Exercise 3

Method overloading is a concept that has been first introduced in this chapter. Research the technique, and outline the benefits it provides for writing clear and maintainable code.

Exercise 4

Investigate how programs were written before object orientation became a dominant form of software development. Explain the benefits that are purported to arise from adoption of object orientation.

Exercise 5

Imagine that any of our practical programs had required us to permit the user to search through a database of one hundred movies, books or artists. Outline the process by which this could be done but **do not write the program**.

Exercise 6

Explain the relationship between an object and a class, and what role each has in the programs we construct.

Topic 6: Consolidation (1)

6.1 Learning Objectives

On completion of this topic, students will be able to:

- Integrate all unit material to date into a worked example.

6.2 Timings

Lectures: 0 hours

Laboratory Sessions: 5 hours

Private Study: 4 hours

Tutorial: 1 hour

6.3 Laboratory Session

6.3.1 Resources Required

Ensure that Visual Studio is installed on your computer:

6.3.1.1 Introduction

This chapter is going to change the format of the unit briefly, permitting you time to strengthen your understanding of the material covered to date. Here, we'll look at a worked example that draws everything we have discussed together into a single, coherent program that you will develop independently before the solution is discussed as a class.

Refer to the previous chapters for guidance on how to make use of all the tools you have available. All of these will be important. You'll need to make use of:

1. Events
2. GUI components
3. Selection structures
4. Objects

Ensure that you are comfortable with these before beginning.

6.3.1.2 The Electronic Cash Machine

A company called 'Money Marketplace' has contacted you to build a prototype of the automated cash machine system they are intending to put into production. Before building large, expensive hardware devices that are installed in shops and other convenient locations, they wish to test a small scale simulation for usability. They've hired you to do that work.

A user interface designer attached to the project has provided you with a storyboard from which you should work:

Balance	<h2 style="text-align: center;">Money Marketplace</h2> <div style="text-align: center; margin-top: 50px;"> Enter Your Pin Number to login. </div>			
Withdraw				
Withdraw with Receipt				
Confirm				
Deny				

7	8	9
4	5	6
1	2	3
A	0	C

The user for this application should make use of the keypad you provide to enter a four digit personal identifier number (PIN). This will be checked against a number of objects stored in the program to see which account should be made active. Pressing 'Balance' will show the balance of the account on the main display. Pressing 'Withdraw' will prompt the user to type a value to withdraw into the display before pressing 'Confirm' or 'Deny' to approve or deny the transaction. 'Withdraw with Receipt' will show a window on the main display that lists the last transaction performed on the account.

The underlying bank simulation should ensure that it's not possible to withdraw money if there is no sufficient sum available in the account.

Your task is to write the code for this application.

6.4 Tasks

Task 1

Create the user interface, as shown above. Variation from this is permitted, but all the requested functionality should be permitted through UI elements.

Task 2

Implement the keypad, which should permit numbers to be entered into the system at the appropriate points in the program's functioning, specifically at login or when withdrawing money. They should do nothing otherwise. Pressing C should clear the previously entered number. Pressing A should do nothing, for now.

Task 3

Create four objects representing customer accounts in the system. These should implement the following methods:

- Compare PIN, which takes in a string representing a PIN and returns true if it matches the one stored and false if it doesn't.
- Withdraw money, which takes in a number representing a number and adjusts the balance appropriately. It should return true if the transaction was successful and false if it wasn't.
- Query transaction, which returns a string containing the last successful transaction, and "no transaction" if there wasn't one.
- A constructor that allows for the PIN and balance to be set.

These four objects should come with PIN and balances set when the program loads up.

Task 4

Implement the functionality for the other buttons – Withdraw, Withdraw with Receipt, Balance, Confirm and Deny.

6.5 Private Study Exercises

Exercise 1

Make a note of some **limitations** in this program. For example, the fact it stores only four accounts is a problem. We'll be addressing some of these in the second consolidation exercise at the end of the unit.

Exercise 2

Provide a copy of your code to a colleague in the class. Have them work through the program and comment on any areas where the functionality seems unusual or broken.

Exercise 3

Consider the importance of **correctness** when writing a piece of software like this – research how you might test it to make sure it does what it's supposed to do.

Exercise 4

Outline some improvements that could be made to the underlying bank code. For example, you might wish to support an overdraft, or user information beyond simply a PIN number. Outline how you could do that within the context of this program.

Topic 7: Data Structures

7.1 Learning Objectives

On completion of this topic, students will be able to:

- Make use of arrays to build scalable programs;
- Combine arrays and objects for powerful data solutions;
- Use Lists;
- Understand the Generics system used for .NET programs;

7.2 Timings

Lectures:	1 hours
Laboratory Sessions:	5 hours
Private Study:	3 hours
Tutorial:	1 hour

7.3 Laboratory Session

7.3.1 Resources Required

Ensure that Visual Studio is installed on your computer:

7.3.1.1 Data Structures

In the tasks for Topic 6, you will have discovered that certain elements you might expect from such an application were impossible to implement in a sensible way. Most banks don't have three or four accounts – they might have three or four million. Most banks don't need you to carefully select between individual accounts to find the right one – they can locate them almost instantly. In this chapter, we are going to discuss the tremendous impact that data structures have on your programs.

In a very real sense, a good data structure is what will determine the effectiveness of a program you write. A good data structure is easy to manipulate and quick to access – it makes everything you do easier, and as a result your program code will be cleaner, more efficient, and more maintainable. A poor data structure will need you to write complicated code to do simple things, with a corresponding negative impact on code quality. For the moment, we have largely been dealing with poor data structures – individual objects which we must check one at a time, for example. From now on, we're going to make our own lives much easier.

7.3.1.2 The Array

Here's the problem we have at the moment – our variables live an isolated existence with no way, in code, to link them together. If we create five integers to represent age, we can't tell the program to group them together into 'a list of ages'. If we could, we'd be able to do things like 'read each age in this list and give me all of them that are greater than 18' or 'count how many ages in this list are equal to 30'. Since we can't do that, we have to write the code that groups them together ourselves, with individual comparisons. The more variables we have, the more comparisons we have to make.

The good news is, we **can** create that list – we do it through the use of a data structure called an **array**. This allows us to create a single variable that has many compartments within it – in each compartment, we store an individual piece of data. Instead of using age1, age2, age3 and so on we can create an array called ages, and store each age in its own compartment:

Ages	
	20
	25
	18

Each of these compartments also has an address that we can use to extract a specific age if we need it. In C#, we begin counting at zero (you've seen this in how we do for loop counters) and arrays are no different. Every compartment gets its own numerical address – this is known as an **index**:

Ages	
0	20
1	25
2	18

When we make use of an array in code, we use a slightly different notation – we create a variable for the array in the usual way, but we put square brackets on the type information:

```
int[] ages;
```

This creates a container for the array, but it doesn't put anything in it. We have to actually **create** the array at an appropriate point (such as in the constructor of an object, or the load event of a form). When we do this, we tell the program how big we want the array to be, like so:

```
ages = new int[3];
```

The index means that we can access individual ages with the same precision as we do for a variable, but that we can do it with a single, consistent variable name. In C#, we access the contents of an array index (known as an **element**) with a square bracket notation. If we wanted to change the second age in our array to 40, we'd do this:

```
ages[1] = 40;
```

And at the end of this line of code our array would look like this:

Ages	
0	20
1	40
2	18

If we wanted to put the third element in a text box, we'd do something similar:

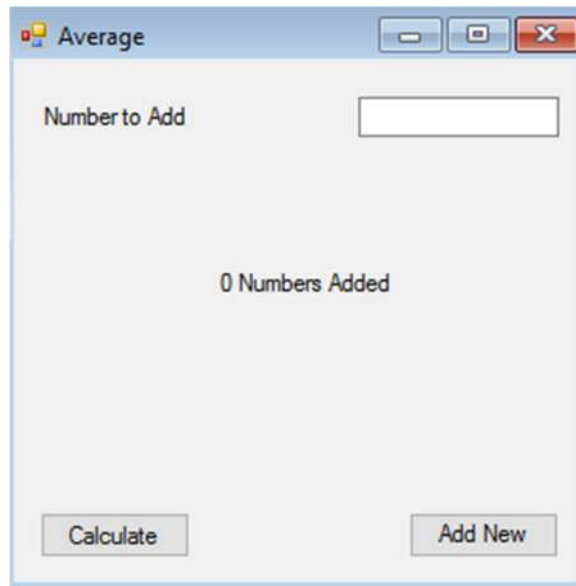
```
txtOutput.text = "The third age is " + ages[2];
```

It's easy to be underwhelmed here, but there's a secret feature of arrays that makes them the best new friend you have ever had. The size of an array is **whatever we want it to be**, which means the code difference between an array of ten, twenty, or twenty-thousand elements is **nothing**. This means our programs, for the first time, become truly **scalable** – as we need larger amounts of data stored, our code stays as complex, or otherwise, as it was when we needed to store smaller amounts. That's because of the other secret feature – **you can use a variable to provide the index**.

This is, you will find, an amazing benefit to someone writing a computer program. Let's see what we can do!

7.3.1.3 An Example of an Array

We're going to create a program here that will give us the average of ten numbers we input. This is an amount of numbers we have never even dreamed of before. Consider a program making use of a textbox (txtToAdd) and two buttons (cmdAdd and CmdCalculate). We'll also use a label, lblNum that shows us how many numbers we've already got added. It'll look like this:



We need to add code to create our array, which we'll call **nums**:

```
int[] nums;

private void frmAverage_Load(object sender, EventArgs e)
{
    nums = new int[10];
}
```

We'll also need the code that lets us add a number to our array. For this, we'll also need to keep track of the next index into which we will insert the element. We'll also make sure we can't overfill the array (if we do, the program will crash – we'll talk about how to fix that in a later chapter).

```
int[] nums;
int currentIndex;

private void frmAverage_Load(object sender, EventArgs e)
{
    nums = new int[10];
    currentIndex = 0;
}
```

The array we have at this point looks like this:

Nums	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

We want to start filling these compartments with what the user enters, so we implement the code for our add button:

```
private void cmdAdd_Click(object sender, EventArgs e)
{
    int num;

    if (currentIndex == 10)
    {
        MessageBox.Show("You have filled the array. Press calculate.");
        return;
    }

    num = Int32.Parse(txtToAdd.Text);

    nums[currentIndex] = num;

    currentIndex = currentIndex + 1;

    lblNum.Text = currentIndex + " numbers in array.";
}
```

If our user has the number 20 in the textbox, then this function checks to see if we have a `currentIndex` that does not equal ten – if we do, we'd be trying to put something into position 11 of the array (remember, we start counting from zero) and we only have ten. If that's not a problem, we parse the number out of the text box, and then put the number into the array at the position indicated by `currentIndex`. To begin with, we set `currentIndex` to zero – the end result is that our array now looks like this:

Nums	
0	20
1	
2	
3	
4	
5	
6	
7	
8	
9	

At the end of this, `currentIndex` is increased by 1 (to one), and the label is updated to show how many numbers we've entered. The next time we press the button, the number will go into index 1, and then 2, and then 3, and then so on until we reach the end.

Note here that we don't need to store the variable anywhere other than the array – the **num** variable we are using within the function is only to hold the temporary result of the integer parsing. The array is the only place we keep this data.

That means that we can then easily calculate our average by summing up the numbers we have and then dividing them by the current index. If we were doing this the bad old way, it would look like this:

```
total = nums[0] + nums[1] + nums[2]... + nums[9];
```


We're not going to do that though – we're going to make use of a loop to do it for us. The fact that we can index an array using a variable is incredibly powerful, because it reduces summing up to this:

```
private void cmdCalculate_Click(object sender, EventArgs e)
{
    int total;

    total = 0;

    for (int i = 0; i < currentIndex; i++)
    {
        total = total + nums[i];
    }

    MessageBox.Show("The average is " + total / currentIndex);
}
```

But that's not all. Because here's why arrays are going to change your world – this code looks **exactly the same** if we're averaging ten numbers, or ten thousand. All we need to do is change the size of the array:

```
nums = new int[10000];
```

And the check that stops us from overfilling:

```
if (currentIndex == 10000)
{
    MessageBox.Show("You have filled the array. Press calculate.");
    return;
}
```

Or even quicker still – an array is an object, and it has properties. One of these properties is **Length**, and that gives us back the size of the array:

```
if (currentIndex == nums.Length)
{
    MessageBox.Show("You have filled the array. Press calculate.");
    return;
}
```

All we need to change, whenever we want to scale up to massive amounts of data, is the code that deals with how many elements we want. Nothing else, not even the average calculation, needs to change at all.

That's amazing, but sometimes even that isn't enough. Sometimes we don't even want to say how many things should be there in the first place. For that, we need to use a slightly more complex data structure – the List.

7.3.1.4 The List

Think of the List as an array that grows and shrinks as you need it. This is a special, powerful object in C# that uses up some more computing resources (CPU and memory) but in exchange for a flexible data structure that doesn't need us to know when we create the code how many

things we'll be dealing with. Think of the array above as a **bounded** data structure and the List as an **unbounded** structure and you'll understand the circumstances under which we might wish to use either.

The List is a more advanced kind of structure known as a Collection, and we need to tell .NET where to find it. We add the following line to the top of any program making use of a List:

```
using System.Collections;
```

However, when we declare a List we need to use a slightly odd syntax known as the **Generic** system. Think back to when we created our classes in a previous chapter – imagine if we could provide data types when we actually created a class, and those would be reflected in the variables we used inside it. We won't talk about how to do that, not here, but many classes in .NET do this through the use of generics. We provide type information when we create the object, and when we declare its variable – like so:

```
List<int> nums;
```

And when we create the object:

```
nums = new List<int>();
```

Otherwise, we make use of this the same way we do any object. We add things on to the end to the List using the Add method, but we can still use the square bracket notation if we want specific elements. What we don't need to do though is track what number of the index we're currently at, because the List does it for us. If we want to find out how many things have been added, we make use of the Count property of the list. And, since we don't set a size of the List, there's no need for us to have a check to make sure it isn't overfilled:

```
private void cmdAdd_Click(object sender, EventArgs e)
{
    int num;

    num = Int32.Parse(txtToAdd.Text);

    nums.Add(num);

    lblNum.Text = nums.Count + " numbers in array.";
}
```

And:

```
private void cmdCalculate_Click(object sender, EventArgs e)
{
    int total;

    total = 0;

    for (int i = 0; i < nums.Count ; i++)
    {
        total = total + nums[i];
    }

    MessageBox.Show("The average is " + total / nums.Count);
}
```

This program works exactly the same way as the original one we did for arrays, but with a greater degree of flexibility because the List will grow as we need.

7.3.1.5 Objects

Both of these data structures are very powerful when dealing with ints, strings and the other primitive data types. They become incredibly powerful when dealing with objects – and they can do exactly that. Consider the following class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _7._2
{
    class Person
    {
        private String name;
        private int age;

        public Person(String n, int a)
        {
            name = n;
            age = a;
        }

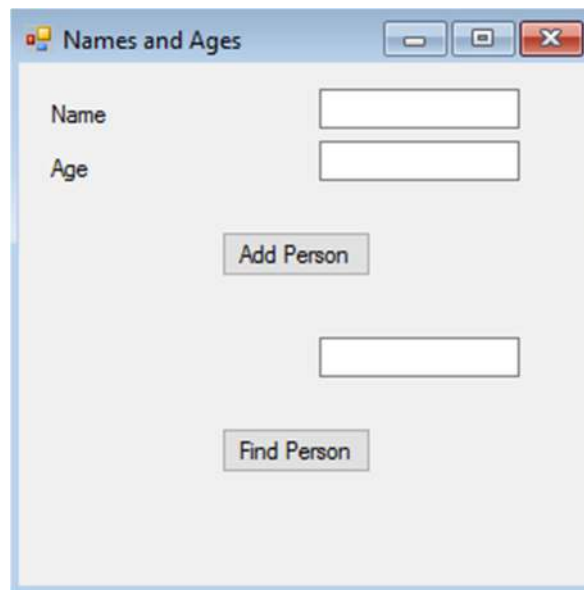
        public string Name
        {
            get
            {
                return name;
            }

            set
            {
                name = value;
            }
        }

        public int Age
        {
            get
            {
                return age;
            }

            set
            {
                age = value;
            }
        }
    }
}
```

If we wanted to make a simple database making use of arrays, lists and objects, we can do that. Consider the following interface. There are three text boxes (txtSearch, txtName and txtAge), and two buttons (cmdAdd, cmdFind).



Here, when we press Add Person we are going to take their name and age, create an object, and store it in a List. If we wanted to use an array, we'd create it like so:

```
Person[] people;

private void frmNamesAndAges_Load(object sender, EventArgs e)
{
    people = new Person[10];
}
```

We're not going to do that, but the technique is very similar. The only difference is that we'd need to keep track of our currentIndex if we were doing this with a standard array. We'll be using a List, derived from Generic syntax:

```
List<Person> people;

private void frmNamesAndAges_Load(object sender, EventArgs e)
{
    people = new List<Person>();
}
```

Creating the object follows the pattern we discussed in Topic 5, while putting it in the array based on the Add method we discussed above:

```
private void cmdAdd_Click(object sender, EventArgs e)
{
    string name;
    int age;
    Person p;

    name = txtName.Text;
    age = Int32.Parse(txtAge.Text);

    p = new Person(name, age);

    people.Add(p);
}
```

To find the right person, we'll go over each object in the list and see if their name matches the one that we entered into the text box. If it does, we'll display their age:

```
private void findPerson_Click(object sender, EventArgs e)
{
    String name;

    name = txtSearch.Text;

    for (int i = 0; i < people.Count; i++)
    {
        if (people[i].Name.ToLower() == name.ToLower())
        {
            MessageBox.Show(name + " is " + people[i].Age);
        }
    }
}
```

Now, think how useful that would have been in the previous chapter when it came to finding an account that matched a PIN. This kind of system is hugely powerful, and we'll be seeing it a lot as we progress through this module.

7.4 Tasks

Task 1

Modify the program above for finding averages to provide a button that will find the highest and lowest values to be found in the numbers list.

Task 2

Modify the program above for finding averages to output a count of all numbers that are greater than five and less than 10.

Task 3

Modify the program above for finding averages to create a list of one hundred randomly generated numbers that are shown on a new label (lblNumbers) on the form.

Task 4

Making use of the simple Person database above, implement it using an Array rather than a List.

Task 5

Create a new program. Add a class called Book that stores a title, author and year of publication. Set up an array of these with some predetermined values. Allow the user to enter a year, and output to a label all of the books that were published in that year.

Task 6

Making use of the book program you created in Task 5, add two new properties to the class. 'Checked out', which should be a boolean, and ID which should be a unique number for each book stored. Add a new search interface that allows the user to search by book ID, and mark the book as checked in or out as appropriate.

Task 7

Making use of the book program you wrote for Task 6, modify the search from Task 5 so that it only outputs books that have not been checked out.

7.5 Private Study Exercises

Exercise 1

Consider the previous programs you have written for this unit. Select one of these and explain how the use of an array would have improved the software quality. You do not need to re-write the program.

Exercise 2

Explain the circumstances under which you might choose to use an Array rather than a List. Provide an example for a scenario in which both might be preferred.

Exercise 3

Research the topic of **generics** and provide an explanation of why this style of software development emerged, and the advantages it brings.

Exercise 4

Research the concept of an **associative array** such as the Dictionary, and give a brief description of how it works and why someone may wish to use it.

Topic 8: Object Orientation (2)

8.1 Learning Objectives

On completion of this topic, students will be able to:

- Make use of inheritance;
- Understand and apply polymorphism;
- Override methods through the use of virtual keywords

8.2 Timings

Lectures:	1 hours
Laboratory Sessions:	5 hours
Private Study:	3 hours
Tutorial:	1 hour

8.3 Laboratory Session

8.3.1 Resources Required

Ensure that Visual Studio is installed on your computer:

8.3.1.1 Objects Revisited

We saw in Topic 5 how useful objects could be for grouping together sets of related data into a single unit we could reference throughout the program. We discussed how we could **encapsulate** data and the methods that act upon that data together. We saw in the last chapter how we could make lists of objects through the use of array type data structures and create our own simple databases. In this chapter, we're going to discuss the real reasons why objects are so intensely powerful – the role of **inheritance** and the flexibility offered by **polymorphism**.

These, together with encapsulation, represent the three key pillars of Object Orientation, and effective use of this programming style depends on being comfortable with all of them.

8.3.1.2 Inheritance

Often when building an object oriented program we will encounter a scenario when we need to duplicate large amounts of functionality within two objects. Consider our cash machine scenario – we have only a single kind of account, but most banks offer many kinds of accounts with many different features. Some offer overdrafts. Some have a minimum deposit amount per month. Some offer only delayed withdrawals. Some have both an authorised and unauthorised overdraft facility. In all cases, they'll share a certain set of attributes – the owner of the account, and the balance. Some though will have more things they track, and some will treat the common things differently.

Throughout the course of this unit we've been looking at the ways we can reduce or eliminate code duplication. Arrays permit us to cut down on the number of variables we use. Loops allow us to have the program handle repetition rather than the programmer. However, with objects we've introduced the problem once more – we're now in the situation that if we want objects that do similar but different things, we have to duplicate all the code.

Or rather, we **would** have to duplicate were it not for inheritance. Inheritance permits us to set a class as the **parent** of another class. The **child** of that class takes on all the attributes and methods of the parent, and can provide additional functionality either through **extension** (the provision of more attributes and methods) or **specialisation** (changing how existing methods work). Most child classes will make use of a combination of these approaches.

Let's look at a simple example – a User class that serves as the **parent** to both a NormalUser and an AdminUser. For both of the child classes we'll want to store a real name, a user name, and a password. The class definition would be as follows:

```
class User
{
    private String realname;
    private String username;
    private String password;

    public string Realname
    {
        get
        {
            return realname;
        }
    }
}
```

```

    }

    set
    {
        realname = value;
    }
}

public string Username
{
    get
    {
        return username;
    }

    set
    {
        username = value;
    }
}

public string Password
{
    get
    {
        return password;
    }

    set
    {
        password = value;
    }
}
}

```

To create our NormalUser, we don't need to duplicate any of these. We simply create a new class as normal, and as part of our class definition indicate our desire to **inherit** the methods and attributes of a parent:

```

class NormalUser : User
{
}

```

The colon there indicates that we're specifying an inherited relationship. The name that follows it is the class from which we are inheriting. You've seen this notation before – it's what's there in all the forms we create:

```

public partial class frmInheritance : Form

```

This is why we don't need to do anything about telling C# what a Form is or how it should work. There's a Form class already written, and we're saying 'one of those, please'. When we add event handlers and the like we are **extending** the functionality of the Form class.

Right from the start, our NormalUser now has all the methods and attributes of its parent class. If we added nothing else to our definition, it would function exactly as a User class. You can see that easily enough by attempting to access its properties:

```

NormalUser u;

```

```
u = new NormalUser();

u.Username = "mjh";
```

Although we didn't add any of the attributes or the properties, they're declared in the parent class so we get them for free in the child. Similarly if we create an AdminUser:

```
class AdminUser : User
{
}
```

And then attempt to make use of its properties:

```
NormalUser u;
AdminUser a;

u = new NormalUser();
a = new AdminUser();

u.Username = "mjh";
a.Username = "admin1";
```

That's interesting, but it's not useful yet. Let's look at making these objects do something worthwhile.

8.3.1.3 Making good use of inheritance

We're going to differentiate these classes based on how many times we can fail to logon before we get locked out. If we fail to enter the right password for an admin account three times, the account locks. A normal user never gets locked out. That means the following:

- Both accounts are going to need some code to check if a password is correct
- The admin account will need to track how many unsuccessful logins there have been, and whether the account is locked.
- The admin account only will refuse to validate a login if it is locked

The first of these is shared functionality, so this is where inheritance starts to become useful – we don't write the code for this in either child class. We put it into the parent so they can share it:

```
class User
{
    private String realname;
    private String username;
    private String password;

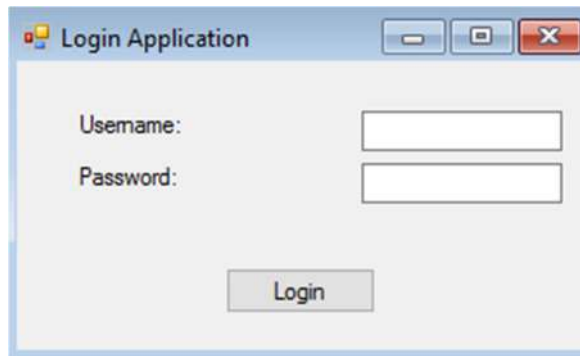
    // Property setup as usual

    public Boolean checkPassword(string pass)
    {
        if (pass.ToLower().Equals(password.ToLower()))
        {
            return true;
        }

        return false;
    }
}
```

```
}
```

Just like that, both NormalUser and AdminUser have a checkPassword method available. Let's try it out, making use of a simple login application:



We're going to create a list of our NormalUser objects, and then provide the appropriate 'access granted' or 'access denied' message when the login button is pressed. We'll populate this to begin with three accounts:

```
public partial class frmInheritance : Form
{
    List<NormalUser> accounts;

    public frmInheritance()
    {
        InitializeComponent();
    }

    private void addAccount(String name, string user, string pass)
    {
        NormalUser u;

        u = new NormalUser();

        u.Realname = name;
        u.Username = user;
        u.Password = pass;

        accounts.Add(u);
    }

    private void frmInheritance_Load(object sender, EventArgs e)
    {
        accounts = new List<NormalUser>();

        addAccount("Michael Heron", "mjh", "bing1");
        addAccount("Alan Moon", "am1", "ticket");
        addAccount("Aoife Lockhart", "awl", "euro");
    }
}
```

Our job now is going to be to find the account the user enters, and then check the provided password. First, to find an account:

```
private NormalUser findAccount(string us) {
    for (int i = 0; i < accounts.Count; i++)
    {
```

```

        if (accounts[i].Username.ToLower() == us.ToLower())
        {
            return accounts[i];
        }
    }

    return null;
}

```

Here we step over each account in our list, checking to see if the lower case of the username matches the lower case of the string we provided in the username textbox. If it does, we return the account object found in the current position of the list. Returning null means we didn't find anything.

Our login code at the end looks like this:

```

private void cmdLogin_Click(object sender, EventArgs e)
{
    NormalUser user;

    user = findAccount(txtLogin.Text);

    if (user == null)
    {
        MessageBox.Show("No such account.");
        return;
    }

    if (user.checkPassword(txtPassword.Text))
    {
        MessageBox.Show("Access granted.");
    }
    else
    {
        MessageBox.Show("Access Denied.");
    }
}

```

When we run this program we find that we get the appropriate messages based on our login. If we replaced every mention of NormalUser with AdminUser, we'd find exactly the same thing. At the moment, our classes are identical. That's about to change. But first...

8.3.1.4 Polymorphism

You might have noticed a problem here – if we are using a typed list, how can we have accounts of different types stored within? We could of course create a list of normal users and a list of admin users, but then we need to be constantly checking through two different lists and that seems like more work than it's worth. Here's where the next powerful object oriented technique comes into play – **polymorphism**.

Polymorphism is the code convention of treating instances of a specialised object (a child) as instances of a more general object (one of its parents). Our NormalUser is a NormalUser, but it's also a specialised type of User. Within object oriented programming, we can treat it as either case.

That doesn't sound useful until we realise that AdminUser is an AdminUser, but it is **also** a specialised kind of User. If we can treat both NormalUser **and** AdminUser as a type of User, then we have a solution to our problem. When we provide type information, we provide it using

the most general form the object will take. In this case, User. Our program above, if written polymorphically, would look like this:

```
public partial class frmInheritance : Form
{
    List<User> accounts;

    public frmInheritance()
    {
        InitializeComponent();
    }

    private User findAccount(string us) {
        for (int i = 0; i < accounts.Count; i++)
        {
            if (accounts[i].Username.ToLower() == us.ToLower())
            {
                return accounts[i];
            }
        }

        return null;
    }

    private void addAccount(String name, string user, string pass)
    {
        NormalUser u;

        u = new NormalUser();

        u.Realname = name;
        u.Username = user;
        u.Password = pass;

        accounts.Add(u);
    }

    private void frmInheritance_Load(object sender, EventArgs e)
    {
        accounts = new List<User>();

        addAccount("Michael Heron", "mjh", "bing1");
        addAccount("Alan Moon", "am1", "ticket");
        addAccount("Aoife Lockhart", "awl", "euro");
    }

    private void cmdLogin_Click(object sender, EventArgs e)
    {
        User user;

        user = findAccount(txtLogin.Text);

        if (user == null)
        {
            MessageBox.Show("No such account.");
            return;
        }

        if (user.checkPassword(txtPassword.Text))
        {

```

```

        MessageBox.Show("Access granted.");
    }
    else
    {
        MessageBox.Show("Access Denied.");
    }
}
}

```

Now, we need to look at what's actually happening here because it looks like we're randomly choosing when to use `NormalUser` and `User`. When we add an account, we create a `NormalUser` variable as usual, but when we use `findAccount` we are asking only for a `User`. This is perhaps the most difficult thing to become comfortable with in Polymorphism, and it's to do with the **contract** our program forms with the underlying virtual machine.

When we provide code that makes use of methods or variables that don't exist, the compiler gives up on the job of trying to turn it into a working program. This is because we are asking it to do something it can't do. In many cases, that's obvious at the time because key information is missing. With polymorphism though, we may not know that there's missing information until the program is already running. When the compiler lets us do something, what it's essentially saying is 'Okay, when the virtual machine gets to this I'm confident that it'll have the information it needs'.

When we create a user, we use `NormalUser` because we know that's the kind of user we want. However, the compiler doesn't know in what order we'll add things to our list. So we tell it to treat everything on the list as a `User`, but what we add is the **specialised** object that we actually want. We don't lose any information with Polymorphism – the object we add to the list isn't a `User`, it's a `NormalUser` with all the specialisations and extensions that implies. It's just that under most circumstances the virtual machine is going to treat it in a more general sense.

When it is time for the virtual machine to execute the `FindAccount` method, it thinks 'okay, everything in this list is **at least** of the level of being a `User`. That means that anything that's been defined as part of the `User` class is available to me'. That's the contract we're forming with polymorphism – that when we use the more general case we're going to restrict ourselves to the functionality that is guaranteed to be there by the parent class structure. We'll see why that's important soon.

To make our `AddAccount` method work polymorphically, we'd do this:

```

private void addAccount(String name, string user, string pass, Boolean admin)
{
    User u;

    if (admin == true)
    {
        u = new AdminUser();
    }
    else
    {
        u = new NormalUser();
    }

    u.Realname = name;
    u.Username = user;
    u.Password = pass;

    accounts.Add(u);
}

```

```
}
```

Here we're saying 'u is going to be a User of some kind, but I'm not sure what kind until the function is called'. We put the appropriate specialised object into the polymorphic User object, and everything works the way we'd expect. Again, we'll come back to that in a little bit. But now, let's adjust our code a little to take into account the new AdminUser objects we're using:

```
addAccount("Michael Heron", "mjh", "bing1", true);
addAccount("Alan Moon", "am1", "ticket", false);
addAccount("Aoife Lockhart", "awl", "euro", true);
```

Now mjh and awl are admin, but am1 is not. Let's make that actually **mean** something.

8.3.1.5 AdminUser extensions

Now we need to set the admin account as being potentially locked, and also keep track of the number of failed logins. The attributes are the easiest to do, so let's add them:

```
class AdminUser : User
{
    private int failedLogins;
    private Boolean locked;

    public int FailedLogins
    {
        get
        {
            return failedLogins;
        }

        set
        {
            failedLogins = value;
        }
    }

    public bool Locked
    {
        get
        {
            return locked;
        }

        set
        {
            locked = value;
        }
    }
}
```

Here's one of the areas where polymorphism has an impact – whenever we're dealing with the more general class, we can't use any properties, attributes or methods that are defined in a more specialised one. We can't have our findAccount method for example make use of the FailedLogins property because there is no way to guarantee that it's available in every child of the User class. NormalUser, as an example, doesn't have these. If we want to get access to these values, we need to be doing it either **within** the class itself, or with a specialised object that is typed appropriately.

We don't have the latter of those, but what we can do is **specialise** the behaviour of our login code within the class. For this, we use a process called **overriding**. We discussed **overloading** earlier in this unit – that's the technique of providing two methods with the same name but different types of parameters. Overriding is when we take a method name and provide a more specialised version of it in a child class. Like so:

```
public override Boolean checkPassword(String pass)
{
    return base.checkPassword (pass);
}
```

Note here that we also define this as an **override** function – that's because C# won't let us accidentally override a function. We have to explicitly indicate our intention. Also, to do this, we need to indicate that the method we're overriding can be overridden. We do this by giving it the **virtual** modifier. We do that in the parent class:

```
public virtual Boolean checkPassword(string pass)
{
    if (pass.ToLower().Equals(password.ToLower()))
    {
        return true;
    }

    return false;
}
```

We use **base** to refer to the parent class, saying here 'run the checkPassword routine in the parent, and return that as the result'. However, we can choose to do that whenever we like, meaning that we can have a checkPassword function that works a bit differently:

```
public override Boolean checkPassword(String pass)
{
    Boolean ret;

    if (locked == true)
    {
        return false;
    }

    ret = base.checkPassword (pass);

    if (ret == false)
    {
        failedLogins += 1;

        if (failedLogins == 3)
        {
            locked = true;
        }
        return false;
    }
    else
    {
        failedLogins = 0;
        return true;
    }
}
```

Here, we still have the parent class responsible for telling whether a password is correct, but we provide some guard conditions around when that is called. If the account has been locked, we don't even bother checking to see if the password is correct.

The really clever thing about how polymorphism works though is that while it will only allow you to checkPassword if it can guarantee every child of the User class has the method, when it comes to execute the function it'll use the most specialised version of that function available in the object. When we call checkPassword on a NormalUser, it'll find the one in the base User class. When we call checkPassword on an AdminUser, it'll find the more specialised one defined in AdminUser. As such, we now have objects with two different login regimes that work off of a common stem. This is incredibly powerful. We'll see more examples as time goes by as to what we can do with this.

8.4 Tasks

Task 1

Write a class called Animal. It should expose a name property. Create child classes called Dog, Cat and Horse. The Animal class should have methods called move() and talk(), and a queryType() function that returns the kind of animal it is. The base animal class should return “unspecified” for its type.

Task 2

Provide polymorphic methods for the program you wrote in Task 1, so that the following return values are given from the child classes:

Class	Move	Talk	Type
Cat	Walk	Meow	Cat
Dog	Walk	Woof	Dog
Horse	Run	Whinny	Horse

Task 3

Write a program that permits a user to maintain a list of animals, where they give a name, a type, and instances of these are added to the list in a polymorphic fashion. Upon pressing a button, the program should output the name, type, movement and talk values of each.

Task 4

Create a pair of child classes for the cat. These should be HouseCat and GreatCat. Provide appropriate implementations of the queryType method for these.

Task 5

Create a pair of child classes for the dog. These should be GreatDane and YorkshireTerrier. Provide appropriate implementations of the queryType method for these.

Task 6

Adjust the program you wrote in Task 3 so that it permits people to select Horse, Great Cat, House Cat, Great Dane and Yorkshire Terrier.

8.5 Private Study Exercises

Exercise 1

Research the **class diagram** that is to be found as part of the UML specification. Provide an example of a class diagram inspired by an element of the class structure you created for Task 5 above.

Exercise 2

Consider the cash machine example we discussed in Topic 6. Provide an overview of the parts that would benefit from the adoption of inheritance and polymorphism.

Exercise 3

Research the use of polymorphism, and provide two examples of a situation in which it might be useful.

Exercise 4

Research the concepts of **coupling** and **cohesion**, and discuss how they might impact on object oriented programming designs.

Topic 9: Consolidation (2)

9.1 Learning Objectives

On completion of this topic, students will be able to:

- Integrate all module material to date into a worked example.

9.2 Timings

Lectures: 0 hours

Laboratory Sessions: 5 hours

Private Study: 4 hours

Tutorial: 1 hour

9.3 Laboratory Session

9.3.1 Resources Required

Ensure that Visual Studio is installed on your computer:

9.3.1.1 Introduction

In Topic 6, we discussed a consolidation exercise that worked to integrate all the unit content to date. In this chapter, we're going to look at how we can apply the principles of **incremental development** to modify the program we have written, making use of the techniques we have learned in the previous two topics – specifically, data structures, inheritance, and polymorphism.

As with Topic 6, this material is presented primarily as an exercise for you to work towards, with your lecturer being able to work through the solutions with you as the class goes on.

9.3.1.2 Our Cash Machine

We saw in Topic 6 how to develop a simple cash machine using the tools we had available. However, because we had yet to discuss a number of key concepts in software development we were forced to implement a number of pieces of functionality that didn't work especially well. Consider for example that our cash machine has to check each object individually to find which account we want, and how it has a hard limit of four such accounts.

In this chapter we're going to fix some of the deficiencies in our program, making use of the new tools we have discussed. Specifically, we're going to do the following:

1. Make use of a List to store Account objects.
2. Make use of inheritance to provide implementations of two separate kinds of account.
3. Make use of polymorphism to ensure that these new accounts work well together in the main program.

We are not going to write a new version of the program to accomplish this – instead, we're going to take the one we've already written and fix it. This is an important skill in software development – we often write simpler versions of a program before we go back and write the proper version. This allows us to concentrate on the things we know how to do while waiting to find out about the things we don't. The other approach is to wait until we know how to do everything, but that's rarely truly possible. Instead, we write 'placeholder' implementations of functionality with the expectation that we'll improve upon it later.

Your task in this chapter to write the code for this application, as outlined in the tasks below.

9.4 Tasks

Task 1

Expand your Account class so that instead of using a PIN only as an identifier, it also requires an account number. In the real world this would normally come from a card inserted into the machine, but for our purposes we're going to have it be one of the things a user enters into the system. Modify your login procedure appropriately.

Task 2

Modify your existing code so that instead of permitting only four accounts to be stored, the system can have a potentially unlimited number. Provide a method in your program that lets you create an account from values that you provide as parameters.

Task 3

Consider your findAccount method. Modify this so that it works through the use of a technique that would permit it to search through the list of accounts for something that matches the account and PIN numbers, returning the appropriate account.

Task 4

Create two children classes from Account. One will be BasicAccount, and will give users a fixed overdraft of £100. Users can withdraw money until their overdraft limit is hit. The second account will be an ExtendedAccount, and will provide a variable interest rate. The ExtendedAccount will provide appropriate functionality for setting this, whereas BasicAccount will not. Both should indicate, when the user logs in, what kind of account they have.

Task 5

Modify your findAccount method so that it polymorphically supports accounts of either type you have implemented. Make available an addAccount method for each type of account you created.

9.5 Private Study Exercises

Exercise 1

Outline why the polymorphic approach to bank accounts is more powerful than having two separate lists.

Exercise 2

Why might we want to use an Account Number *and* PIN number to check through large sets of account data? Why was that not necessary for our earlier version?

Exercise 3

What other kinds of account functionality might we put into a third kind of account? How much would you need to change in your program to support it?

Exercise 4

How might we modify our Account object to provide a list of transactions? Support your answer with code.

Topic 10: Testing and Error Handling

10.1 Learning Objectives

On completion of this topic, students will be able to:

- Employ black box testing;
- Understand the importance of regression testing;
- Employ effective test case design to detect errors;
- Handle exceptions through the try-catch architecture.

10.2 Timings

Lectures:	1 hours
Laboratory Sessions:	5 hours
Private Study:	3 hours
Tutorial:	1 hour

10.3 Laboratory Session

10.3.1 Resources Required

Ensure that Visual Studio is installed on your computer:

10.3.1.1 Introduction

Writing a program is only one step of the process. We also need to write programs that work **correctly**. When we write a piece of code, we usually run the program and informally test to see if it works. However, this often leaves bugs and incorrect assumptions that will need to be corrected – something that looks like it works correctly might not actually be functioning as intended. For example, the answer to $2 + 2$ and $2 * 2$ is 4 in both cases. If we are expecting numbers to be multiplied but they're actually being added, getting a correct answer doesn't necessarily mean it was for the right reasons.

In this topic we're going to discuss some strategies for testing, and how we can make use of the C# exception handling system to deal with errors we can't otherwise prevent.

10.3.1.2 Test Cases

The first step in fixing a program is to ensure you can replicate the circumstances under which it goes wrong. The worst kind of bugs to fix are those that are **intermittent** – their causes can be many and varied and staring at the code hoping it reveals its secret is rarely very productive. In software development, we encourage a methodical approach to testing that focuses on developing good **test cases** and then using those to assess program correctness.

Consider the following simple function:

```
public Double findHypotenuse(int side1, int side2)
{
    int hyp = side1 * side1 + side2 * side2;

    return Math.Sqrt(hyp);
}
```

This is a simple implementation of Pythagoras theorem, which states that to find the square of the length of a hypotenuse of a right angled triangle you take the square of two known sides and sum them. This is a well understood system that we know how to evaluate – we'll know if we get the right answers out of this function. However, we need to make sure that we can be confident the answers check all the possible ways this function might go wrong. For that, we need to develop test cases.

Test cases are defined by their input values (what we'll send into the function), an **expected** return value (what we know the answer to be) and an **actual** return value (what the function gives us). We can't exhaustively test all possible combinations of numbers, so we use good test cases to build up a reasonable amount of confidence.

The first test cases we develop should focus on creating **equivalence cases** where we pick a few representative samples in a set of related inputs. For example, 'all positive numbers', 'first is positive, second is negative', and 'first is negative, second is positive', 'all negative numbers' and so on. What the correct equivalence categories will be will vary from function to function. We might want to look at 'well-formed strings' or 'fully populated objects'. We get to decide what these equivalences might be for the needs we have.

We then build these up into a testing strategy, comprising what we intend to be the set of data we want to test. This is a common format:

Test Case	Input 1	Input 2	Expected	Actual
Positive 1	10	5	11.18	
Positive 2	20	100	101.98	
Positive 3	100	100	141.42	
Positive 4	9999	8	9999.00	
Positive 5	1	2	2.23	
Positive 6	5	10	11.18	
Positive 7	100	20	101.98	
Positive 8	8	9999	9999.00	

Note here that we have a range of values – large numbers with small numbers, numbers that are the same, and ‘mirrors’ of earlier test cases. We can’t put every combination of numbers in here, so we want a range of them. Remember, we need to calculate by hand the **expected** value, or use a properly calibrated software tool in which we have confidence to do it for us.

To determine if our program works, we then input each of these test cases into a program and then see what it tells us. We can do that by hand, or we can write a (properly tested) **test harness** to input the values and then output the results for us. In either case, we can then fill out our **actual** results to see whether the program is working correctly:

Test Case	Input 1	Input 2	Expected	Actual
Positive 1	10	5	11.18	11.18
Positive 2	20	100	101.98	101.98
Positive 3	100	100	141.42	141.42
Positive 4	9999	8	9999.00	9999.00
Positive 5	1	2	2.23	2.23
Positive 6	5	10	11.18	11.18
Positive 7	100	20	101.98	101.98
Positive 8	8	9999	9999.00	9999.00

You can see here than in every case the **expected** answer matches the **actual** answer. This function seems to be working correctly, for this set of data. Eight test cases isn’t enough to say it works perfectly for all positive numbers – just that it works correctly for these. The more test cases we have, the more confident we can be that the results will correctly represent the untested possibilities. We can never be sure however. Programming is inherently a discipline in which we must be prepared to accept uncertainty.

Buoyed by these results, we might declare the function correct. But that’s only with this single equivalence case. Let’s try it with positive and negative numbers. We should expect it to work, but we need to be sure. So:

Test Case	Input 1	Input 2	Expected	Actual
Pos & Neg 1	10	-5	11.18	11.18
Pos & Neg 2	-20	100	101.98	101.98
Pos & Neg 3	100	-100	141.42	141.42
Pos & Neg 4	-9999	8	9999.00	9999.00

That seems to work, and so too when we add in test cases for using negative numbers only. Within those equivalence classes, we get the correct results. But there are other test cases we need to consider.

Core to the testing process is **retesting** once you change anything in the code. It's a common industry benchmark that for every two bugs you fix, you'll often inadvertently introduce another. As such, **regression testing** is the process of making sure that every single fix is accompanied by a rigorous repetition of the test data that you've already entered. This is where the idea of writing some code to automate this becomes especially valuable. That's something that needs done on an individual basis, but usually involves the writing of a class which is populated with test data, and then a for loop which executes the tested function for each test data object.

10.3.1.3 Boundary Cases

Boundary cases are those that exist at the point one part of a piece of code will change based on a value. For example, if we have a for loop that should iterate ten times, we would write test cases that would ensure it triggers correctly by clustering test cases around that boundary. We'd check that it works correctly for 8, 9, 10, 11 and 12. And if there were other loops inside the system, we'd check those too. If there are if statements, we check to see what happens if they are true or false, and cluster around those conditions. Let's consider a second example function to test:

```
public int calculatePower(int baseNum, int powerNum)
{
    int ans;
    int numRepeats;

    ans = baseNum;

    numRepeats = powerNum;

    for (int counter = 0; counter < numRepeats; counter++)
    {
        ans = ans * baseNum;
    }

    return ans;
}
```

We've encountered this earlier in unit – here, we're separating it out into a function of its own for ease of testing. We'd cluster test cases around the **numRepeats** boundary, like so:

Test Case	Input 1	Input 2	Expected	Actual
Boundary 1	10	4	10000	
Boundary 2	11	4	14641	
Boundary 3	9	4	6501	
Boundary 4	10	3	1000	
Boundary 5	10	5	100000	
Boundary 6	10	2	100	
Boundary 7	10	6	1000000	

Note here we're not wildly varying these cases – we're targeting them around a potential point of weakness in the code. Wherever a program branches, there's a chance that our code will function incorrectly. When we run our test cases, we'll find out if that boundary check is likely to be safe (at least, for this set of data):

Test Case	Input 1	Input 2	Expected	Actual
Boundary 1	10	4	10000	100000
Boundary 2	11	4	14641	161051
Boundary 3	9	4	6501	59049

Boundary 4	10	3	1000	10000
Boundary 5	10	5	100000	1000000
Boundary 6	10	2	100	1000
Boundary 7	10	6	1000000	10000000

None of those values are correct, and it's because in the process of writing this function we broke the code a little. We need to fix this line:

```
numRepeats = powerNum;
```

So that it says:

```
numRepeats = powerNum - 1;
```

Make that fix, and then run the test cases again:

Test Case	Input 1	Input 2	Expected	Actual
Boundary 1	10	4	10000	10000
Boundary 2	11	4	14641	14641
Boundary 3	9	4	6501	6501
Boundary 4	10	3	1000	1000
Boundary 5	10	5	100000	100000
Boundary 6	10	2	100	100
Boundary 7	10	6	1000000	1000000

That's much better. And now we can see that not only does the code work correctly, it works correctly around the boundary. It's not the case we'd use equivalence cases **or** boundary cases. We'd use both together in a single testing regime.

10.3.1.4 Unusual Data

Next, we want to throw a whole set of 'unusual' data at a function to see what happens. For example, we might want to throw in the maximum number that can be represented, or floating point numbers when we expect ints. Or booleans when we expect strings, or strings when we expect booleans. Perhaps binary data when we should be providing numbers. Remember, for most programs the values that we provide here will come from users, and often they'll be extracted from user elements such as text fields. We can't guarantee a user will enter anything sensible in there, so we need to know what's going to happen if they don't. Let's go back to our calculatePower function and go out of our way to break it:

Test Case	Input 1	Input 2	Expected	Actual
Unusual 1	MaxInt	MaxInt	MaxInt	
Unusual 2	MaxInt	0	0	
Unusual 3	0	MaxInt	0	

We can get the MaxInt value using Int32.MaxValue, so that's what we'll do. Run these test cases and we get...

Test Case	Input 1	Input 2	Expected	Actual
Unusual 1	MaxInt	MaxInt	???	ERROR
Unusual 2	MaxInt	0	1	2147483647
Unusual 3	0	MaxInt	0	0

The problem here is that the function we have is not robust. We can't deal with MaxInt cleanly because we can't actually raise MaxInt to a power. So now we know we need to fix our code so that first situation can't occur:

```
public int calculatePower(int baseNum, int powerNum)
{
    int ans;
    int numRepeats;

    if (baseNum == Int32.MaxValue && powerNum > 1) {
        return Int32.MaxValue;
    }
    ans = baseNum;

    numRepeats = powerNum - 1;

    for (int counter = 0; counter < numRepeats; counter++)
    {
        ans = ans * baseNum;
    }

    return ans;
}
```

That fixes our first case, which now becomes what we set:

Test Case	Input 1	Input 2	Expected	Actual
Unusual 1	MaxInt	MaxInt	2147483647	2147483647
Unusual 2	MaxInt	0	1	2147483647
Unusual 3	0	MaxInt	0	0

The second use case problem is that any number, when raised to a power of zero, should become one. We don't do that – we set the starting number to be the base number. So we need to have another special case to handle a power of zero:

```
public int calculatePower(int baseNum, int powerNum)
{
    int ans;
    int numRepeats;

    if (baseNum == Int32.MaxValue && powerNum > 1) {
        return Int32.MaxValue;
    }

    if (powerNum == 0)
    {
        return 1;
    }

    ans = baseNum;

    numRepeats = powerNum - 1;

    for (int counter = 0; counter < numRepeats; counter++)
    {
        ans = ans * baseNum;
    }

    return ans;
}
```


And then that will give us the following:

Test Case	Input 1	Input 2	Expected	Actual
Unusual 1	MaxInt	MaxInt	2147483647	2147483647
Unusual 2	MaxInt	0	1	1
Unusual 3	0	MaxInt	0	0

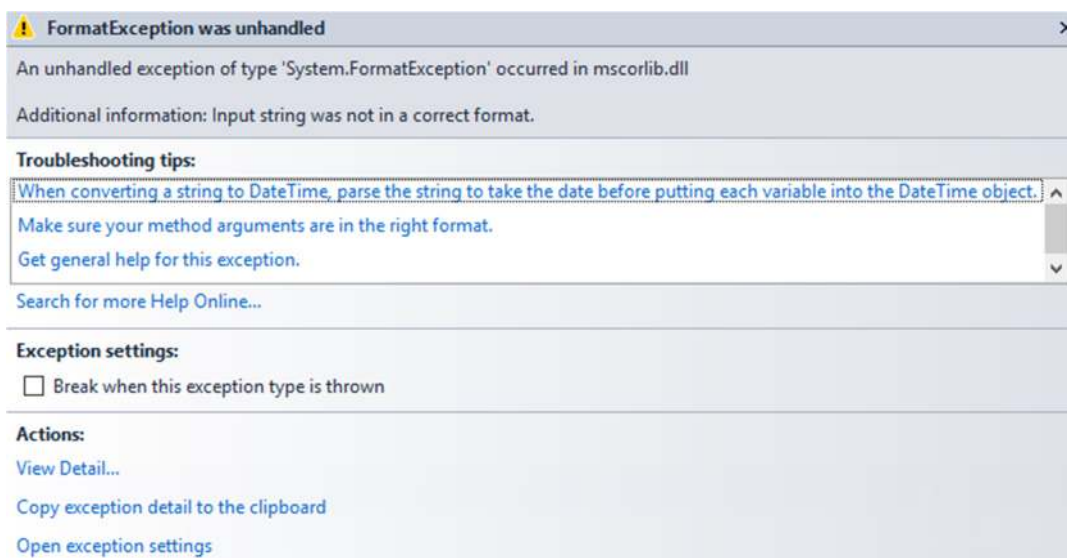
As we add new and interesting unusual data into our test cases, we'll often find strange things happen. Good test cases will reveal flaws in our program – we want to really treat our code as viciously as we can to reveal the things we need to fix and improve.

10.3.1.5 Exception Handling

Sometimes we can't actually test for a scenario because it is user dependant. If we ask a user to enter a number into a textbox, we can't be sure that they won't write "eight" when we expect 8. If we ask a user to pick an element out of an array, we won't necessarily be able to know they'll enter a valid index. We can write code to check such input, but there are other scenarios where we won't have control. We won't know, for example, if a hard-drive fails as we're writing to a file, or if a database becomes corrupted as we read from it. If we're trying to connect to a remote website, we won't necessarily be able to be sure the internet connection is sustained through the transaction. Those are **exceptional** circumstances, and C# gives us a framework for dealing with them. They're called **exceptions**.

With exception handling, we place code that might potentially go wrong inside a **try** block, which says to the virtual machine 'This might go horribly wrong so treat it with a bit of care'. If the code encounters a problem, the virtual machine will generate an **exception**, which is something that needs to be dealt with in some way. It **throws** that exception to us, and we handle it in a **catch** block polymorphically typed to the class of exception. Some of these are known as **unchecked** exceptions, which means that they'll happen at runtime and we don't necessarily need to deal with them – usually we can write code that ensures it doesn't happen. Other exceptions are **checked** exceptions and must be dealt with before the program will compile. We'll see some of those in the next two topics.

Let's consider a common example we've used throughout the module – letting someone enter a number into a text box. We parse that using `Int32.Parse`, but that doesn't work especially well if we try to parse a non-numerical string:



That's what an **unchecked exception** looks like when it's thrown. If we want to avoid that happening when a program runs (and we do), we need to **try** the parsing and then provide an appropriate response when it occurs. Like so:

```
try
{
    num = Int32.Parse(txtNum.Text);
}
catch (FormatException ex)
{
    MessageBox.Show("Please enter a real number");
}
```

Note here that we catch a `FormatException` – that's what the dialog above told us was being thrown. We can also catch the general case `Exception`, because all exceptions have `Exception` in their inheritance tree and the catch statement works polymorphically. That's handy, but not exactly user friendly – we want to be able to give meaningful guidance, and that only happens when we know **what** exception they just triggered.

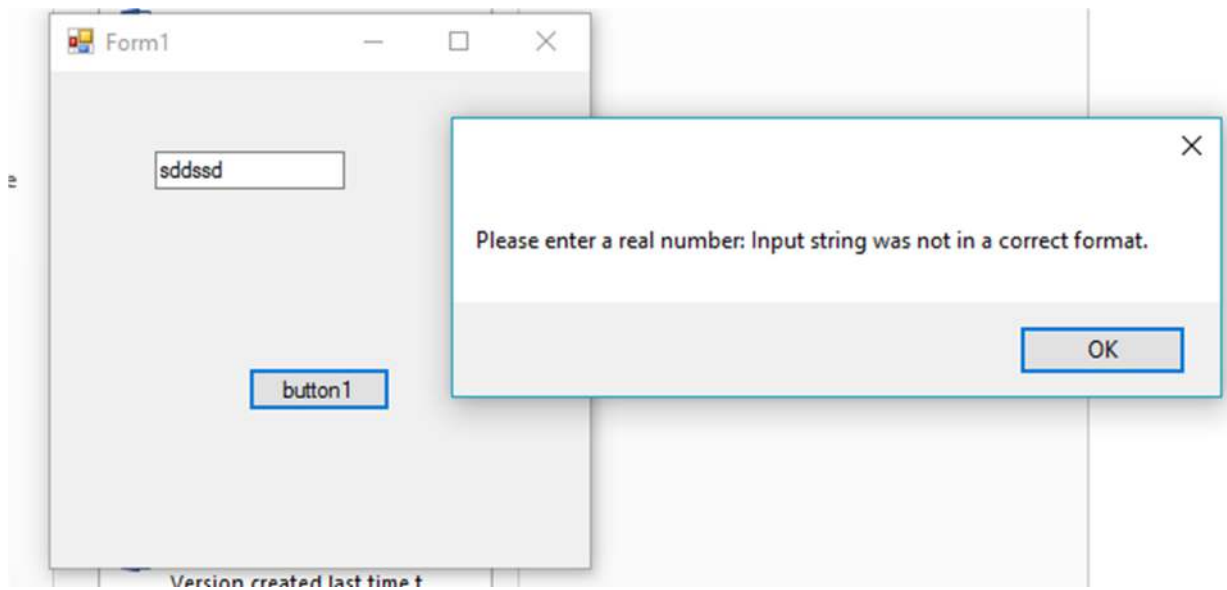
We can however 'tier' these responses to ensure robust programs that are also helpful for the user. As with an if/else-if we can add Catch statements indefinitely – the only requirement is that the most specialised exception has to be at the top of the structure, or more general cases will catch it first:

```
try
{
    num = Int32.Parse(txtNum.Text);
}
catch (FormatException ex)
{
    MessageBox.Show("Please enter a real number");
}
catch (Exception ex)
{
    MessageBox.Show("Something went wrong. I don't know what.");
}
```

Note too that we're giving the exception we catch a name – `ex`. The exception is an object like much of what we work with within C#. It also contains full details about the actual cause of the exception, which we can access:

```
catch (FormatException ex)
{
    MessageBox.Show("Please enter a real number: " + ex.Message);
}
```

Usually these messages are not ready for users, but they can be useful to us when we're catching exceptions to make sure that we, while developing, are aware of what's going on:



10.4 Tasks

Task 1

Write a simple testing harness for the `calculatePower` method. Create a class called `TestData`, populate it with the values in the testing table, and then run it against the function and output the results.

Task 2

Pick a function you have written during the course of this unit. Develop a full testing regime for it, including equivalence classes, boundary checks, and unusual sets of data. Note any exceptions that you identify in this process, and put together an exception handling routine for each.

Task 3

Go through the previous programs you have written, and add exception handling for each time you have prompted a user for numerical data. Provide meaningful user feedback to advise what should be done to correct the problem.

Task 4

Create a function of your own design, that will test someone's ability to write test cases. It should have a reasonably well understood function that maps on to the real world, but have numerous logical paths that must be followed in order to fully test the system.

10.5 Private Study Exercises

Exercise 1

Research **equivalence cases**, and give some examples of additional cases that you have encountered during your search.

Exercise 2

Research the concept of the Exception, and identify some common unchecked exceptions you might be likely to encounter during the normal course of coding the programs we've discussed during the module.

Exercise 3

Exception handling is only one way we can deal with errors. Give an outline of how we might properly validate numerical data that comes from a text box without trying and catching the potentially troublesome code.

Exercise 4

Research the statistics of bugs in computer programs, and give an overview of how important testing is in ensuring software is delivered in as functional a form as possible.

Topic 11: File IO

11.1 Learning Objectives

On completion of this topic, students will be able to:

- Read data from external files;
- Write data to external files;
- Use dialog choosers for files;
- Serialization of objects;
- Deserialization of objects;

11.2 Timings

Lectures:	1 hours
Laboratory Sessions:	5 hours
Private Study:	3 hours
Tutorial:	1 hour

11.3 Laboratory Sessions

11.3.1 Resources Required

Ensure that Visual Studio is installed on your computer:

11.3.1.1 Introduction

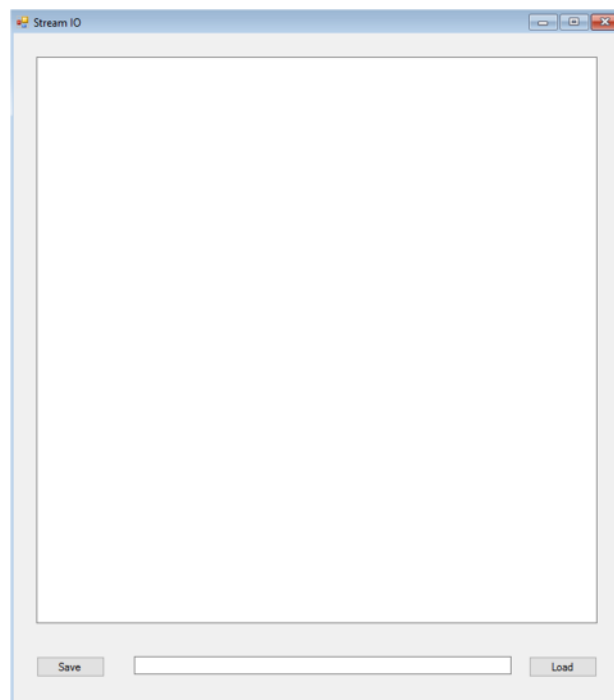
Up until this point, we have covered a large number of the fundamental building blocks of C#, but we are hugely restricted in that we cannot possibly write 95% of real world applications. That's simply because we have no way of storing the state of an application between executions. We run our code, we put in some data, we close the application down – and everything is lost. We might use constructor methods to populate test data, but as soon as the program stops running our data is gone.

Luckily though the C# programming language provides us with a wide range of methods and objects for dealing with File Input/Output (File IO) requirements. In this chapter, we'll be looking at one of the simpler frameworks for loading and saving data, that of **stream based IO**. Then we'll look at serialization as a way of storing object data indefinitely.

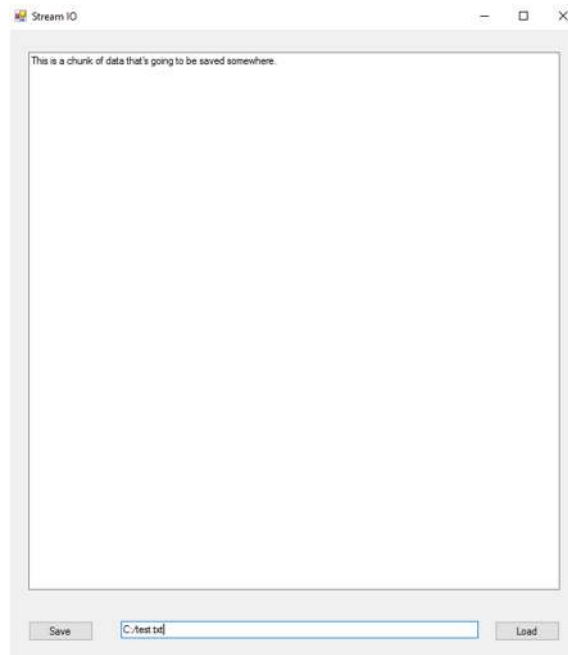
11.3.1.2 Writing to Files

Let's look at how we might write a very simple text editor. This is going to have a large text area in the centre, and two buttons – 'load' and 'save'. We'll allow the user to enter a file path, and then that's where in the file system we'll work with the document.

With stream based IO we have a three phase process that lets us make a connection to a file. First, we open a connection to a file location that we specify. We do this through the use of a StreamWriter. We'll need to add the System.IO **using** definition at the top of our programs to get access to this. Our application is going to look like this:



Our largest text area is named txtInput, the smaller txtLocation. The buttons are cmdSave and cmdLoad respectively. All we're going to do here when we press the save button is take the text in txtInput and store it in a file indicated by the txtLocation textbox, like so:



Adding the first bit of code for our save button gives us the following:

```
private void cmdSave_Click(object sender, EventArgs e)
{
    StreamWriter output;

    output = new StreamWriter(txtLocation.Text);
}
```

The StreamWriter object contains a number of methods that give us access to the functionality we need to write things to the file. One of the things it has is a WriteLine function that lets us take a chunk of text and write it directly into the file we specify. If the file doesn't exist, it'll be created for us. Once we're done, we **close** the file to complete the interaction and release the file back to other applications to use if they want:

```
private void cmdSave_Click(object sender, EventArgs e)
{
    StreamWriter output;

    output = new StreamWriter(txtLocation.Text);

    output.WriteLine(txtInput.Text);

    output.Close();
}
```

For alphanumeric data, this is fine. Objects won't work with this system, but we'll see how to fix that a little later in this chapter.

11.3.1.3 Reading from Files

Next, we want to be able to load in a file that we previously created. As you might imagine, this works in a very similar way except we use a **StreamReader** instead. We also make use of the `ReadToEnd` method to read in the entirety of the data in a single go:

```
private void cmdLoad_Click(object sender, EventArgs e)
{
    StreamReader input;

    input = new StreamReader(txtLocation.Text);

    txtInput.Text = input.ReadToEnd();

    input.Close();
}
```

While reading in text input is roughly the same complexity as writing it, we often have to do other things to turn that input into useful data. For example, we might write out an array of integers into a string and then have to turn the string that comes back into an array of integers. This is a process called **parsing** and it is highly individualised for each program. We won't deal much with that in this topic, but you'll encounter it in more depth elsewhere.

11.3.1.4 Choosing Files

The problem with this program is that we're currently using a textbox to provide a filename, and that's very fragile and unfriendly to users. There are reasons we might want to do it, but usually it's better to give the user an easier option – a file open dialog. Let's look at changing our program now so that the `txtLocation` box goes away and we use a **chooser** instead. Let's look at how it works for saving:

```
private void cmdSave_Click(object sender, EventArgs e)
{
    StreamWriter output;
    SaveFileDialog ofd;
    String file;
    DialogResult selected;

    ofd = new SaveFileDialog();

    selected = ofd.ShowDialog();

    if (selected == DialogResult.OK)
    {
        file = ofd.FileName;
    }
    else
    {
        return;
    }

    output = new StreamWriter(file);

    output.WriteLine(txtInput.Text);

    output.Close();
}
```

First, we create the `SaveFileDialog` and tell it to show itself. What comes back from that is a `DialogResult` object that we use to compare to some fixed results that are held in the `DialogResult` class itself. If the OK button was pressed, we grab the filename that was selected. Otherwise, we return from the function. The save file dialog will do all the rest of the work of handling confirmation and checking for the existence of the files. It's very handy.

Opening a file works in a very similar way, except we use an `OpenFileDialog` like so:

```
private void cmdLoad_Click(object sender, EventArgs e)
{
    StreamReader input;
    OpenFileDialog ofd;
    String file;
    DialogResult selected;

    ofd = new OpenFileDialog();

    selected = ofd.ShowDialog();

    if (selected == DialogResult.OK)
    {
        file = ofd.FileName;
    }
    else
    {
        return;
    }

    input = new StreamReader(file);

    txtInput.Text = input.ReadToEnd();

    input.Close();
}
```

Not only does this give us a lot of functionality for free, it also makes the application work better for the user and behave consistently with all the other applications we might use on a day to day basis. It adds a little bit of extra work to every File IO interaction, but makes them much stronger as a result.

11.3.1.5 Serialization

Objects are a little more complex than string data – they're usually made up of a number of other data fields of their own, for one thing. The process of **serialization** is used to turn an object into a stream of bytes that can be stored on a file (or a database, or transmitted via a network, or whatever you want to do with it). This is how we save the state of an object between invocations. Let's go back to our cash machine and see how we could use serialization to save the state of our bank.

We're going to want to be able to **serialize** the classes we have created for our objects – nothing else in this program needs to save. And then, when we save we're going to want to take the List of those accounts and write them out to a file. When loading, we want to take the serialized stream of bytes and turn it back into a list of objects. Luckily, it's easy enough to do but we do need to change our classes a little to indicate which of our attributes are serializable. If we had an object made up of other objects, we'd need to go all the way down the object chain making sure every single part of it could be converted into a byte. If we don't do this, we'll get an exception when we make the attempt.

First, we amend our classes like this:

```
[Serializable()]
class Account : ISerializable
{
```

What we're saying here is that we're going to have a set of methods in our account that implement a special kind of structure called an **Interface**. Don't worry about it, but what it means is that we now need to add in two new methods. One is a **deserialization constructor** that is used when we load in an object, and the other is `GetObjectData` which is what we use to strip off key elements of data and put them into a key/value pair. We'd implement the constructor like so:

```
public Account (SerializationInfo info, StreamingContext context) {
    PIN = (String)info.GetValue ("PIN", typeof(String));
    accountNumber = (String)info.GetValue ("Account Number", typeof(String));
    balance = (int)info.GetValue ("Balance", typeof(int));
    lastTransaction = (List<String>)info.GetValue ("Last Transaction",
typeof(List<String>));
}
```

Note here that there's an unusual syntax, but all you need to know is that for the bracketed part before the `GetValue` call, you use the data type of the attribute you want to set. For the `typeof` you do exactly the same thing. The first string that goes into `GetValue` is something we'll decide for ourselves, as part of the `GetObjectData` call:

```
public virtual void GetObjectData (SerializationInfo info, StreamingContext context)
{
    info.AddValue ("PIN", PIN);
    info.AddValue ("Account Number", accountNumber);
    info.AddValue ("Balance", balance);
    info.AddValue ("Last Transaction", lastTransaction);
}
```

You don't have to worry about calling either of these functions. It'll be done for us. But we also need to make sure all our specialised classes are **also** serialized. We use the same basic system for this, except we need to call the base constructor (as we have seen before) and override `GetObjectData` with our new data, like so:

```
[Serializable()]
class BasicAccount : Account, ISerializable
{
    private int overdraft;

    public BasicAccount (SerializationInfo info, StreamingContext context) : base (info,
context) {
        overdraft = (int)info.GetValue ("Overdraft", typeof(int));
    }

    public override void GetObjectData (SerializationInfo info, StreamingContext context)
    {
        base.GetObjectData (info, context);
        info.AddValue ("Overdraft", overdraft);
    }
}
```

And:

```
[Serializable()]
class ExtendedAccount : Account, ISerializable
{
    private double interestRate;

    public ExtendedAccount (SerializationInfo info, StreamingContext context) : base
(info, context) {
        interestRate = (double)info.GetValue ("Interest Rate", typeof(double));
    }

    public override void GetObjectData (SerializationInfo info, StreamingContext context)
{
        base.GetObjectData (info, context);
        info.AddValue ("Interest Rate", interestRate);
    }
}
```

With this, we're ready to save our objects, which we'll do whenever any state changes in the main program.

11.3.1.6 Using Serialization

Serialized data is **binary**, and it uses a different kind of system. Usually we do this with a base stream and a binary formatter object, like this:

```
private void saveData() {
    Stream output;
    BinaryFormatter bf = new BinaryFormatter();

    output = File.Open ("atmdata.atm", FileMode.OpenOrCreate);

    bf.Serialize (output, myAccounts);

    output.Close();
}
```

And when we load it in, we deserialize via a similar formatter like so:

```
private void loadData() {
    Stream output;
    BinaryFormatter bf = new BinaryFormatter();

    output = File.Open ("atmdata.atm", FileMode.Open);

    if (output.Length != 0)
    {
        myAccounts = (List<Account>)bf.Deserialize (output);
    }

    output.Close();
}
```

Now, all we need to do is call these functions at the appropriate time. We'll call `loadData` at the load event, and if there is nothing that comes out of it we'll populate our usual test data:

```
private void frmCashMachine_Load(object sender, EventArgs e)
{
    enteredText = "";
    enteredPin = "";
    enteredNumber = "";

    loadData();

    if (myAccounts == null) {
        myAccounts = new List<Account>();

        addBasicAccount ("20", "1234", 10000);
        addBasicAccount ("30", "4321", 55000);
        addExtendedAccount ("40", "5555", 1000, 1.5);
        addExtendedAccount ("50", "9966", 150000, 2.0);
    }

    inputPermitted = true;
}
```

And finally we add the `saveData` call when we manage a successful deposit:

```
        if (ret == true)
        {
            lblOutput.Text = "Transaction successful. You have a " + current.queryType()
+ " account";
            saveData();

            ...
        }
    }
}
```

Run your program, try some transactions, and then shut it down. Start it up again, and you should find the state of your data is the same as how you left it. It needs a little bit of extra class setup to make this work, but the good news is that you only need to do it for classes that you want to save. Nothing else needs the extra work.

11.4 Tasks

Task 1

Write a program that outputs an array of ten random numbers to a file, in text format, each on a line of its own.

Task 2

Making use of the program you wrote in Task 1, read in those numbers and store them in an array as they were originally.

Task 3

Add a new class to your ATM program. This one should be called SerializedClass and include name, address and email for the account holder. Make this work with the other classes you have in the program.

Task 4

Add a new integer to the basic account object, 'Times loaded'. This should increase by one every time the program starts up. Every ten times an Extended Account is loaded up, it should apply the interest rate set in the class.

11.5 Private Study Exercises

Exercise 1

Research **serialization** and give a few possible use cases for which it could be successfully applied.

Exercise 2

Explain why serialization is necessary for objects, and why it's not sufficient to simply write them out as text files.

Exercise 3

Research the concept of Random Access Files and explain how they work in comparison to Stream based files.

Exercise 4

Consider the GetObjectData function and the serialized constructor we added to our classes. Why might these be the approaches taken with Serialization? What benefit comes from this, or what problem is avoided?

Topic 12: Databases in .NET

12.1 Learning Objectives

On completion of this topic, students will be able to:

- Connect a Database to a C# program;
- Create adapters and data sets;
- Read data from a Data set and display it;
- Navigate through currency managers;

12.2 Timings

Lectures:	1 hours
Laboratory Sessions:	5 hours
Private Study:	3 hours
Tutorial:	1 hour

12.3 Laboratory Sessions

12.3.1 Resources Required

Ensure that Visual Studio is installed on your computer. You will also need access to a database. One of these will be provided as part of the material for this chapter. It is called Test.MDB.

12.3.1.1 Introduction

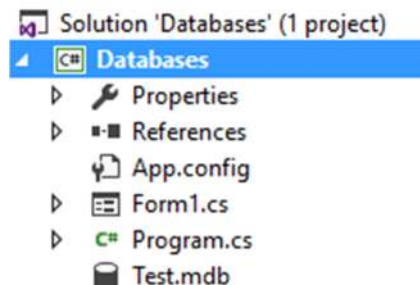
Classes and objects are tremendously powerful but they do suffer from some problems. They're not generally speaking **interoperable**. While serialization, as we discussed in the previous chapter, permits some degree of data interchange the technique is fraught and not universally supported. Sometimes we want to gain access to powerful query tools, or interact with an existing set of data. That's where databases come in, and in this chapter we're going to look at how we use them with C#.

For the purposes of this section, we'll assume you already know the basics of how databases work, including the theory behind how they are structured and how they are queried. You should draw on your knowledge developed in the Databases unit to assist you here.

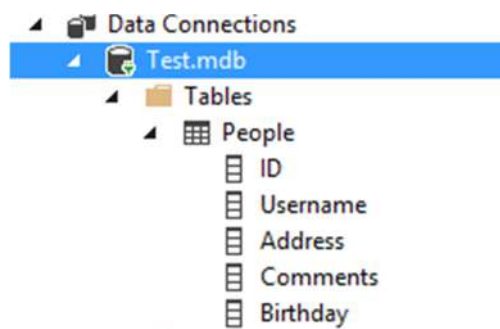
12.3.1.2 Connecting to a Database

A database is an external file, and so to make use of it we need to hook it up to our program. We do this by choosing **add > existing item** and navigating to where it is stored on our system. We will likely have to change the file-type to **data files** to get it to show up.

Once we've added it, we'll have it appear as part of our solution explorer, like so:



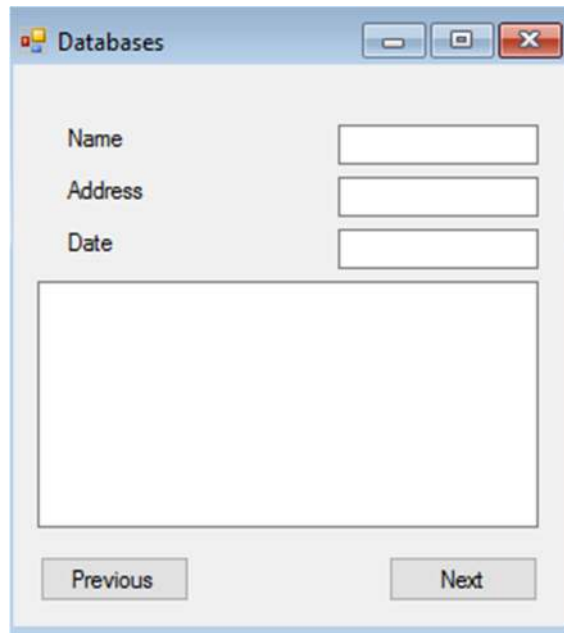
This will also reveal the server explorer, which can be used to examine the structure of the database you just connected. Click into it to make sure the connection works – if it doesn't, you may need to make sure you have the appropriate database drivers and packages installed on your system. We're using, for the purposes of this chapter, an **mdb** file which should be broadly compatible with most systems.



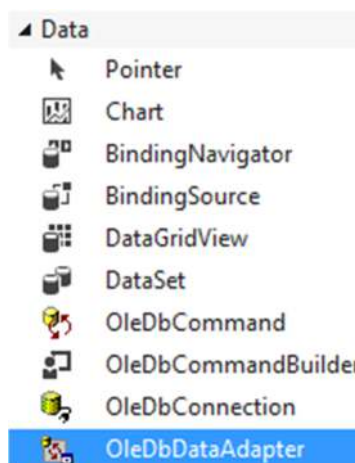
Once you've got this connected, it's ready for us to start making connections and pulling information out.

12.3.1.3 Setting up the Data Architecture

Many UI components in Visual Studio are known as **data aware components**, which means they can be **bound** to a data source and draw their content from it. This allows us to easily connect up textboxes, combo boxes and more to an underlying database without much work. Let's create the following interface, which we'll use to explore this:



First of all, we need to create a **data adapter** which sits between our program and the database. This is used to provide a layer that means our code doesn't need to change if different databases are plugged in. For this, we need to use the **Data** tab in our toolbox and add an **OleDbDataAdapter**. If that's not there, right click on the toolbox, choose 'choose items' and add the four OleDb components you'll find.

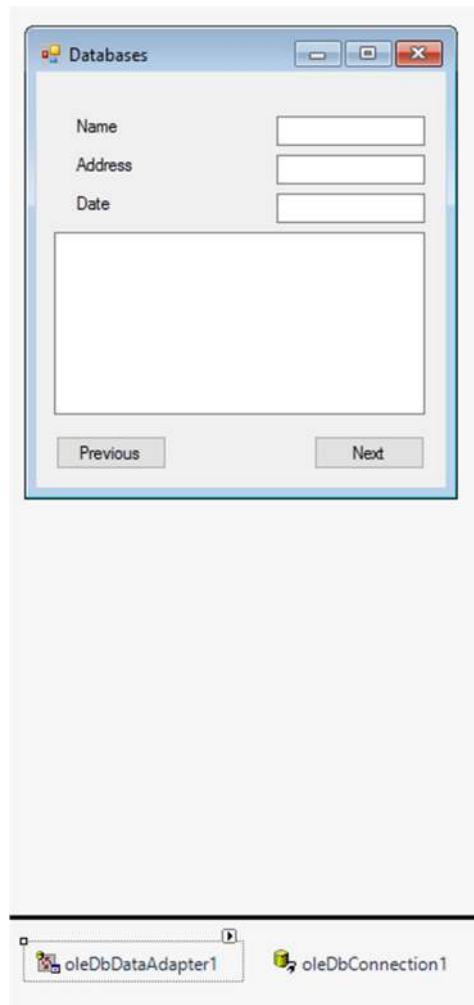


The data adapter goes onto your form in the same way as any control, but it'll be invisible. When you drag it onto the form, you'll go through a series of dialogs that step you through the process. You'll pick the database we added (test.mdb), choose 'Use SQL Statements', and then provide the following simple SQL statement:

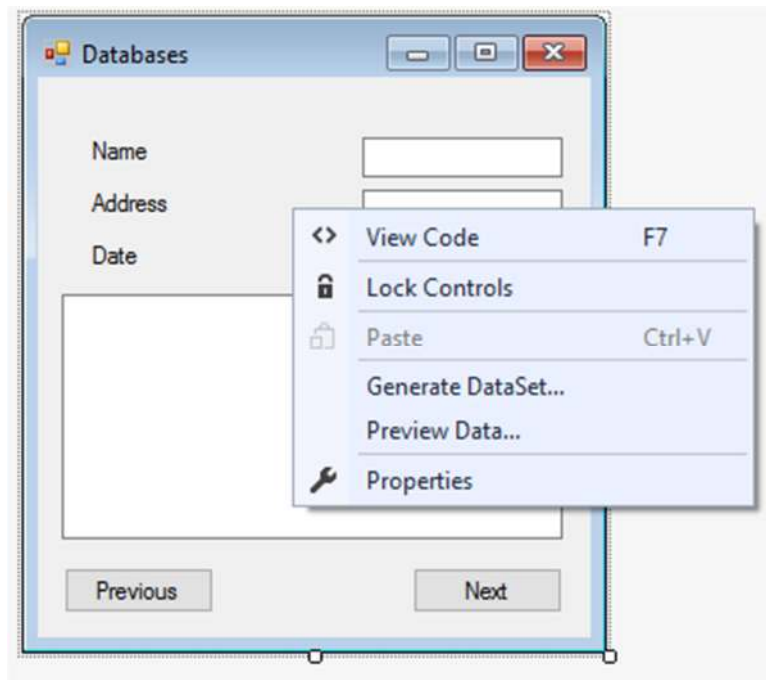

```
SELECT * FROM PEOPLE
```

You can do whatever you like with the SQL here if you know how it works, but this basic statement will just pull every record from the People table. You can see how you might be able to do more interesting things with a bit of SQL magic.

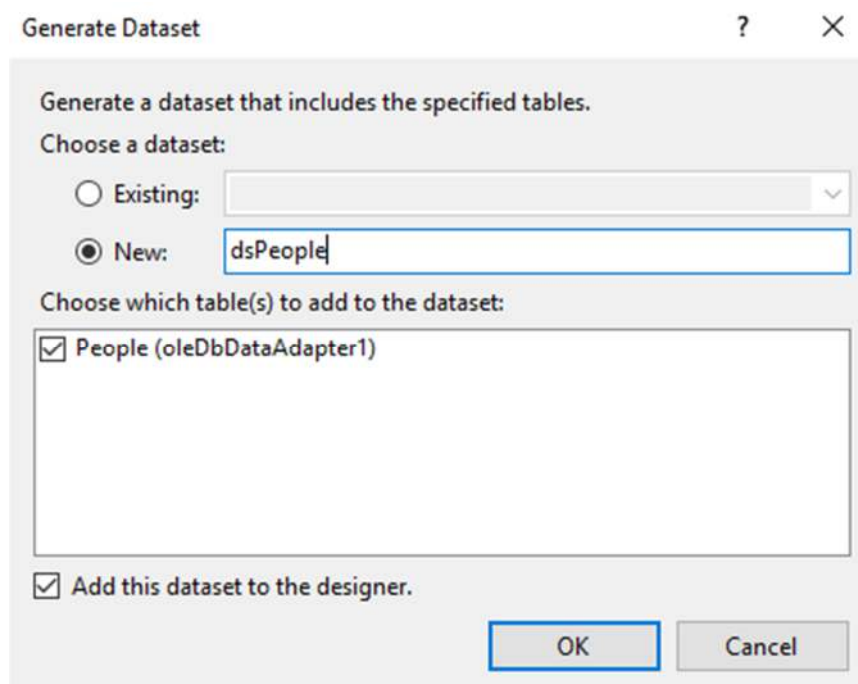
The fully configured adapter we drag across will now appear in the 'tray' of our application – a separate part of the builder that hides the 'invisible' things from the main form:



If we want to draw a different set of data from a different query we'd use a second adapter, but they'll all be connected to the same data. That in turn is done through an object called a **Dataset** which represents a snapshot of a specific set of the database – content from tables, queries or SQL statements. To get our dataset, we right click on our form and choose **generate dataset**:



This will bring up the wizard for generating a dataset – we'll give it a name (dsPeople) and then click 'ok':

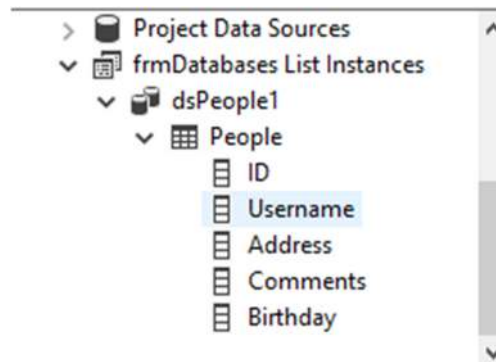


And then **finally** we've made the connection from our application to the database, and we can start hooking up our data aware controls.

12.3.1.4 Data Aware Controls

We need to decide which controls are going to display specific parts of our dataset. We've already done that though, so all we need to do is **bind** the right control to the right part of the data.

Each data aware control has a property called (DataBindings), and that in turn has a sub property called Text. By exploring the drop down menu it gives us, we can drill down into a single field of a single table in our project:

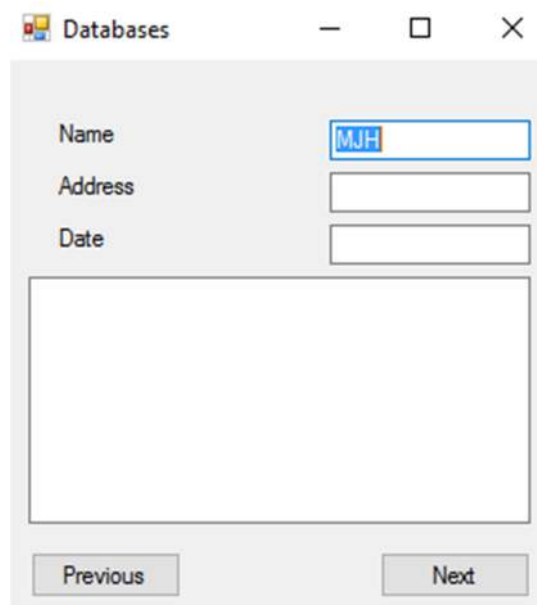


Note that you'll want the data sets that are associated with the List Instances of our project – those are the ones that are properly configured.

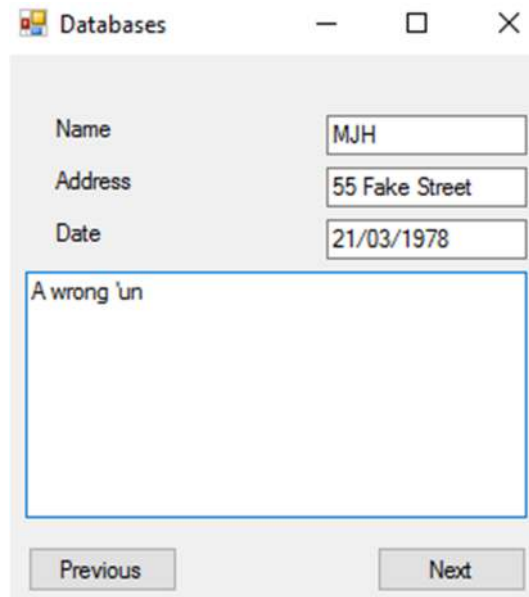
We'll select **Username** here, which sets the **binding** between our data set and the text box control. We're not quite done yet though. The last thing we need to do is tell our application to link up the adapter to the dataset. You might think we've done that already, but all we did is set up the architecture. We actually need to tell the program to do it in code, such as in our Load event:

```
private void frmDatabases_Load(object sender, EventArgs e)
{
    dsPeople1.Clear();
    OleDbDataAdapter1.Fill (dsPeople1);
}
```

And with that, finally, we can run the program and see the data from the table represented in that text box:



We then go through the other text boxes and bind them to the appropriate locations in the database to display a full record for the data set we've created:



If we change the underlying database and rerun our program, we'll get a newly filled adapter full of the data that's just changed. As you can imagine, this is very powerful if we're looking to share data between applications. But this is a passive view of data already there – we also need to know how to change it and navigate.

12.3.1.5 Navigation

Each of the current records being shown in the database is handled by a system called a **CurrencyManager**. Its job is to keep track of which controls are pointing to which fields of which records. You don't need to worry about the details of this – all we need to know is that every form in C# has a **BindingContext** object that keeps track of all these CurrencyManagers, and that gives us the tool we need to move things around.

We get access to the BindingContext through the use of the **this** keyword. We then give it the dataset we're manipulating, and the table we want to change. That then gives us access to the Position and Count properties, which we can use to implement moving forwards and backwards through the dataset. For moving backwards through the dataset, for example:

```
private void cmdPrev_Click(object sender, EventArgs e)
{
    if (this.BindingContext [dsPeople1, "People"].Position == 0) {
        this.BindingContext [dsPeople1, "People"].Position = this.BindingContext
[dsPeople1, "People"].Count - 1;
    }
    else {
        this.BindingContext [dsPeople1, "People"].Position -= 1;
    }
}
```

As we change the position, the contents of our textboxes will change accordingly (provided they are all bound to the correct data set). Putting in code for a next, a first, and a last button are done in the same way – by changing the position property to whatever we want,

Laboratory and Private Study Sessions

At this point you have now covered the full set of material for the unit – congratulations! Databases are introduced here as an interesting ‘further skills’ section but it is not anticipated that you make use of them in your assessment.

Use your Private Study time to make sure you’ve implemented each of the previous tasks and are comfortable with the technical tools and theoretical concepts we have developed over the course of the module.

Use the Tutorial session to ask your tutor to revisit any areas of the unit that you are not fully confident with.