An Introduction to Object Oriented Programming with C#

Dr Kieran Mulchrone
Department of Applied Mathematics,
University College, Cork.

September 2010

CONTENTS

Chapter 1
INTRODUCTION

1.1 A Brief History of C#

C# (pronounced see-sharp) is a relatively new language and can trace its ancestry back to C and C++ (as can Java). It is completely object oriented and was developed by Microsoft to work with .NET framework (see below). C# has been developed with the benefit of hindsight and incorporates the best features of other programming languages whilst clearing up any problems. C# has a simpler syntax than C++ and in general whatever you can do in C++ you can also do in C#. C# can sometimes be more verbose than C++ as the language is typesafe, in other words once data has been assigned a type it cannot be converted to another totally unrelated type. In C++ you could just take some data at a memory location and arbitrarily treat it as something else (and sometimes this can be very useful, but dangerous in the hands of a novice).

C evolved from two previous programming languages, B and BCPL. Martin Richards developed BCPL in 1967 designed mainly for writing operating systems and compilers. Ken Thompson developed B, modelled on BCPL, and early versions of the UNIX operating system were developed in this language in 1970 at Bell Laboratories.

Dennis Ritchie evolved the C language from B at Bell Laboratories in 1972. UNIX was developed in C and most major operating systems today are also developed in C. During the 70's C continue to evolve and in 1978 Kernighan and Ritchie published one of the most widely read computer books of all time, namely, "The C Programming Language". C was extremely portable (i.e. hardware and operating system independent) and provided much more powerful features than either BCPL or B.

By the early 80's many flavours of C were in use and were incompatible with each other, thus degrading the portability of C code. An international standard was set by ANSI (American National Standards Institute) in 1983.

Like C, C++ began life at Bell Labs, where Bjarne Stroustrup developed the language in the early 1980s. In the words of Dr. Stroustrup, "C++ was developed primarily so that the author and his friends would not have to program in assembler, C, or various modern high level languages. Its main purpose is to make writing good programs easier and more pleasant for the individual programmer". C++ was based on C because it is:

1. versatile, terse and relatively low-level.
2. adequate for most systems programming tasks.
3. extremely portable.
4. entrenched in the UNIX environment.

C was retained as a subset of C++ because of the millions of lines of C code written, thousands of useful C function libraries, and the millions of C programmers who only needed to learn a new set of enhancements to become proficient C++ programmers. Thus C++ is a superset of C and any valid C program is also a valid C++ program.

The name C++ comes from the C increment operator ++ and thus C++ may be thought of as C + 1, i.e. a better or spruced up C. More importantly C++ support the Object Oriented Programming (OOP) paradigm. So C++ is a hybrid language incorporating OOP and the more traditional procedural aspects of C. This combination of abilities has helped the spread of C++. However, the hybrid nature of C++ means that it is not regarded as a pure OOP language such as SmallTalk.

Object Oriented technologies bring closer the goal of rapid, high quality software development which is economically viable. Objects are reusable software components used to model aspects of the real world. When approaching a new project, development teams can draw on an extensive library of tried and tested object components, thus by reusing objects they reduce the effort involved and increase the quality.

1.2 What is the .NET framework?

The .NET framework is essentially a huge library of pre-written code (objects) that we can use in our programs to speed up development. Writing applications in the .NET framework involves writing code (i.e. a program) which usually incorporates pre-written elements which are part of the framework. Next our program is turned into a set of instructions the operating system on a target computer system can understand - this process is called compilation and is performed by a specialised program called a compiler. When a program is compiled operating system/hardware specific code is not immediately generated, instead Microsoft Intermediate Language (MSIL) code is generated. This resides in an assembly which can be either an executable (*.exe) file or a dynamic link library (*.dll) file. When the assembly is placed on a target operating system/hardware configuration, a just-in-time (JIT) compiler turns the MSIL into instructions than can be read by the specific operating system/hardware configuration. This means we no longer have to worry about target system peculiarities (all automatically handled). C# and the .NET framework can be deployed on all windows operating systems as well as Unix (see http://www.mono-project.com/Main_Page, for example).

1.3 The C# Programming Environment

1.3.1 Generic Considerations

These days programming is carried out using specifically designed applications usually referred to as "integrated development tools" or IDEs. Development tools come in many shapes and sizes and are usually, but not always, specific to a particular operating system. On this course Microsoft Visual Studio .NET is the development tool of choice and the specifics of this environment are dealt with in a little detail in the next section.

There are three steps required to produce a program, which can run on a computer:

1. A program is written with words and symbols understandable to human beings. This is the C# language. Any text editor can be used to produce these files which are typically named as "myfile.cs", and are generally referred as source files or source code. There is nothing special about the source code files used in IDEs although they often colour different elements of the code to help the programmer. A single program may contain one or more source files.

2. Each source file is compiled with a compiler program, which turns the human readable source file into MSIL. During this process the C# language is turned into a form which the computer can understand. Note that MSIL will not be created if the program contains a syntactical error. The compiler also checks the source file to make sure that it conforms to the rules of the C# language and in general will report back where such compiler errors occur.

3. An assembly file is built during a process known as linking with a program called the linker. If more than one source file is used, all the separate MSIL files need to be linked. The end result of this process is an executable file with extension ".exe". This is the file that can be run on a computer (and as noted above with the help of the JIT compiler).

If the program behaves as expected then that is the end of the development process. However, if it does not then the debugging process begins whereby problems are searched for and fixed.

1.3.2 Microsoft Visual C# express edition

Microsoft Visual C# express edition is the development tool of choice on this course. It is a free to download 32-bit user-friendly GUI (Graphical User Interface) application, which runs on various platforms.

The program is started in the usual manner i.e. either using the task bar (Start->Programs etc.) or by double clicking a desktop shortcut, if one has been created. You are presented with the window illustrated in figure 1.3.1. Along the top of the window are menus and toolbars, which are used to perform various actions. A

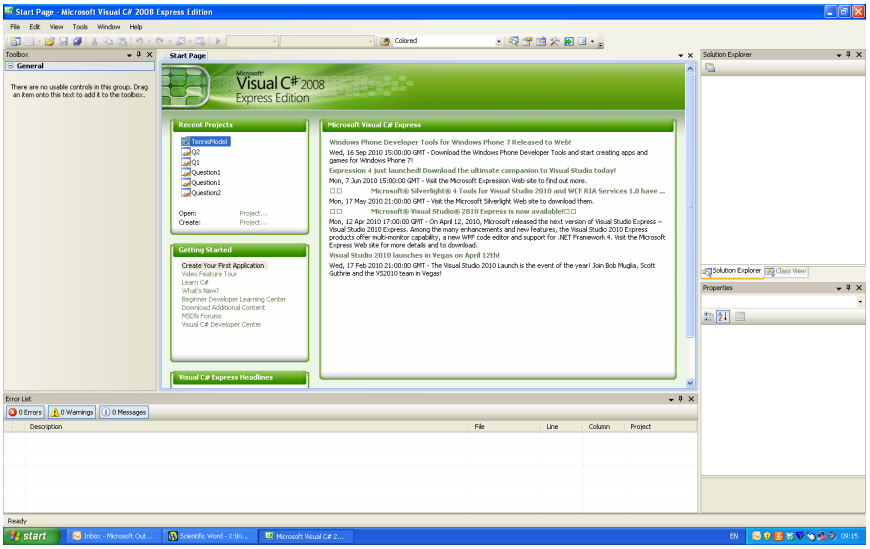

Figure 1.3.1 The startup screen in Visual C# express

message window is placed along the bottom of the window and error messages etc. are displayed here. In the middle of the window (usually) along the right hand side is the workspace window, which is used extensively for organising programming projects. The blank area on the left hand side is where the contents of source files, help files etc. are displayed. Feel free to explore all the various menu options and toolbar options. The primary purpose of this document is to explain the art of C# programming and therefore only a brief introduction to the Visual Studio C# environment is given. A fuller discussion of the use of Visual Studio C# can be found on-line or in the help system.

1.3.3 Creating your First C# Program "Hello World!"

In this section a step by step guide to creating your first C program is given followed by a discussion of the actual program itself. A new program is created by creating a new project. This can be achieved by either clicking the "New Project" button on the start page (see Fig. 1.3.1) or by select the menu option File->New->Project. You will then be confronted with the dialog box illustrated in Fig. 1.3.2. Make sure that "Console Application" is selected. Give the project a suitable name (this will be used to give default names in your program), I would suggest HelloWorld and click OK.

When you click OK Visual Studio C# creates a skeletal program/application for you. What you get for free is shown in Fig. 1.3.3. There are a few items to notice: on the upper right-hand side is a project management area for organising you work or for looking at different views. By default you are in the "Solution Explorer" tab (tabs are at the bottom of the window). A solution is an umbrella for one or more (usually related) projects. We will only be dealing with single project solutions on this

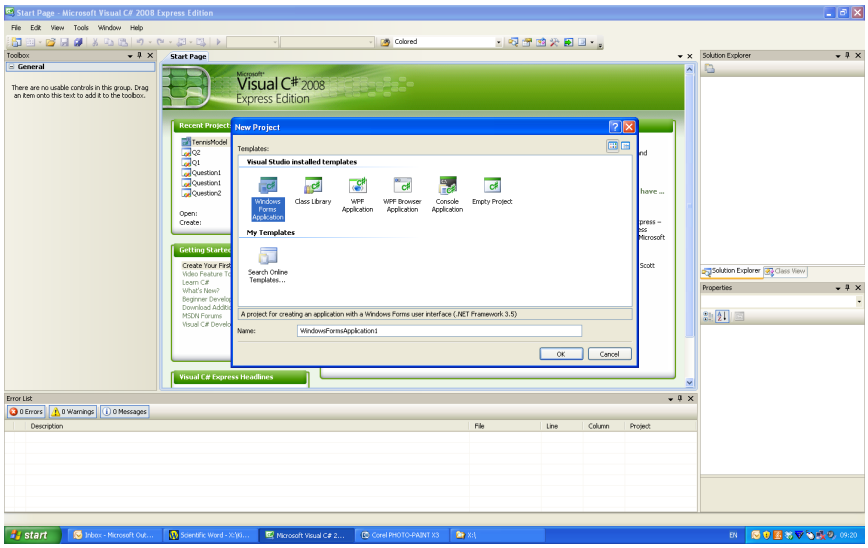

Figure 1.3.2 The new project dialog box.

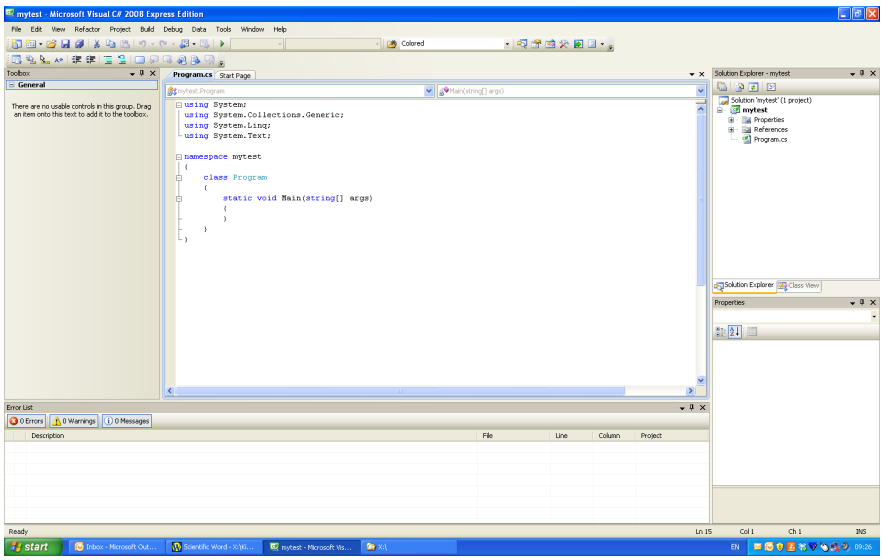

Figure 1.3.3 The default application created for you by Visual Studio.

course. By default the solution is given the same name as the project, although this can be changed if required. The project is identified by its name in bold typeface (I called my project "mytest"). Hanging off the project name are the files and resources associated with the project. The source code file for the project is called "program.cs" by default, but this is editable. At the bottom right-hand side of the IDE is a small window illustrating the properties of whatever is currently selected is displayed, this will be useful for more advanced applications. The main window contains the default code automatically generated for us by Visual Studio.

If you have never programmed before this may look a bit complex. Don't worry all will be explained! First of all lets look at the concept of a namespace. Namespaces are simply a means of organising code. This is useful because there is a vast quantity of programs out there and components of these programs will inevitably be given the same names. For example, if you were writing a program for scheduling work rosters, it is likely that you might have a component called "Schedule". If there were ten other people also writing a similar program then it is likely that they might also use the name "Schedule". Namespaces help us to distinguish between different versions of similarly named components. They also help us to arrange our code in a sensible manner. In the case of our "Schedule" component, I might have a namespace KieranM and my colleague might have namespace JohnM so that my Schedule component can be distinguished as follows KieranM.Schedule from my colleagues JohnM.Schedule. If there is no fear of confusion then we can drop the namespace prefix.

Lets look at the code! The first line using System; is an instruction that we may want to use one or more objects defined in the System namespace (this is a namespace full of goodies programmed by somebody else but available for us to use). Next we define our own namespace (this can be called anything). Inside the namespace we create a class called by default Program we can and should rename this to something more explanatory like CHelloWorld. Classes have to do with objects and we wont dive in to this yet so you can just ignore this for the moment. Inside the class we have method called main. This is where the program begins execution i.e. when the program runs on the computer the instructions placed in here are executed sequentially. For the first while we will be concentrating on writing simple programs inside this main method. At the moment there are no instructions to be executed here so the program does nothing.

Enter the following into the main function:

```
Console.WriteLine("Hello World!");
```

Before you can run and test the program the program needs to built (equivalent to compiler and linking as described earlier). This can be performed by selecting the menu option Build->Build Solution. If the program is syntactically correct then an executable will be create otherwise error messages will be reported at the bottom of the application. Part of the programmers task is to eliminate the error messages, although as you become more experienced these should become more and more infrequent. To run the program select the menu option Debug->Start. What do you

see?

1.3.4 Comments

"Hello World" is a program that is well known to all programmers as it is taken from the classic C reference text "The C Programming Language" by Brian Kernighan and Dennis Ritchie. Programs usually contain comments. Multiple line comments are enclosed between a starting `/*` and an ending `*/`. The compiler ignores anything between these two markers. Single line comments are preceded by `//`. Comments are a very important part of programming and you should generously comment your programs.

In industry every program usually begins with a large comment detailing the purpose of the program, the programmer and the date. This is then followed by a list of updates to the program giving the programmer, date and purpose in each case. Comments should also be scattered throughout a program in order to help others to understand the program. There is nothing worse than trying to read somebody else's un-commented programs.

Exercise 1 *Write a program to print the following message on the screen: "This is my first C# Program". Add a few blank lines before and after the message (hint: use '\n').*

Exercise 2 *Write a program to print the following message on screen:*

```
"C# programs contain source files.
Source files are compiled to MSIL.
MSIL is linked to make executable files.
Executable files run on the computer."
```

Chapter 2
OBJECT ORIENTATION

2.1 Introduction

In this chapter object oriented programming and the whole object oriented paradigm is discussed without reference to the details of the C# programming language. This is because object orientation is independent of any programming language. It is a concept or an approach to solving problems by computer. Many of the terms associated with object oriented programming are unfamiliar and discussed with a new and sometimes strange vocabulary. Sometimes students get confused between the syntax of the language and object orientation. To avoid this happening, the fundamental concepts of the object oriented approach are tackled in plain English with real-world examples. Hopefully this method of presentation allows the reader to come to grips with the object oriented vocabulary before exposure to the C# syntax which supports it. All of the concepts discussed here will be returned to at a later stage with solid programming examples; if you don't get it now, then don't panic!!

2.2 What is Object Oriented Programming (OOP)?

The traditional view of programming is that there is some input data which must be logically manipulated to produce some output data. The logical manipulation should consist of a step by step linear series of operations ultimately giving the required answer. Our program will simply consist of a long list of (hopefully logical) instructions to be performed. What if we had to do the some portion of the sequence of instructions twice? Well, we would have to copy the required sequence of commands to another place in our program. This could lead to large and unwieldy programs. Additionally if an error is detected in the code then we have to search through and find all occurrences of the erroneous sequence in order to fix the error.

This lead on to the idea of extracting useful sequences of commands into a single place (called a function or a procedure). Then in our main section of code we simply call the procedure, pass in any required data, and the commands of the procedure are processed before returning back to the main program with an answer. This simplifies programming greatly: our code is reduced in size, more organised and errors corrected in a procedure are automatically fixed for every procedure call. The main program is responsible for the creation of data and making sure it is the correct

type and format of data for the procedures. Sometimes it makes good sense to group related procedures into a single entity called a module.

The above approach can be problematic (although with diligence and patience, you can solve any problem using this approach). Something else (i.e. external to the procedure) is responsible for the creation and destruction of data as well as sending in valid correctly formatted data. The procedures and the data are decoupled, this tends to lead to a focus on the mechanics of the procedures rather than on the data structures.

Enter OOP. In OOP data and operations on data (data manipulation) are brought together into a single entity called an object. Our programs then consist of one or more objects interacting with each other to bring about a desired result. The object is responsible for its data and its data can only manipulated by a predefined list of acceptable operations. Object orientation aims to emulate the way humans interact with the world around them. In other words there is presumption that the object oriented approach to problem solving is somewhat akin to the natural approach. Consider a typical day, get out of bed, have a cup of coffee, catch a bus to work, go to a restaurant for lunch, go to your flat, eat your dinner with a knife and fork, watch television etc. It is possible to look on life as a series of interactions with things. These things we call objects. We can view all of these objects as consisting of data (called properties) and operations that can be performed on them (called methods). This is thought to closely resemble the way humans perceive the world, thus it is thought that object oriented system models are easier to understand.

2.3 Object Oriented Vocabulary

2.3.1 Objects

Sometimes it not obvious that the process of writing a computer program is an exercise in modelling. Often people perceive that computers are in some way exact. In reality all computer programs provide answers on the basis of models. OOP is one methodology (extremely popular these days) for turning real world problems into satisfactory models that can be solved by computer. An object is anything that can be modelled as data (properties) and operations on that data (methods).

Lets look at a simple example: the motor car. Cars have hundreds of properties (colour, make, model, year, engine cc, leather interior, location etc.), however, it is usually case in programming that for a particular problem we will only be interested in a subset of the properties. Lets concentrate on a single property: velocity. Every car everywhere has the property of velocity. If it is parked the velocity is zero, if it is on the road it is moving at a certain speed in a particular direction. What are the methods that can modify the cars velocity? We can press the accelerator to increase the speed, we can press the break to reduce the speed and finally we can turn the steering wheel to alter the direction of the motion. The speedometer is a method we can consult at any time to access the value of the speed component

of velocity. However, it is clear that without using any of the methods we cannot directly manipulate the velocity (unless we have a crash in which case the velocity goes to zero very rapidly).

2.3.2 Classes

A *class* is used to specify/define the sets of properties and methods applicable to an object (when we are programming this is usually not exhaustive, as we select only those relevant to the problem at hand). Classes are templates or blueprints which are followed to create an object. A class is not an object, no more than a house plan is a house. For example, if a car needs to be manufactured we go to the car class to find out the relevant properties and methods, so that the car can be created. In OOP we say that a physical car (in your hand) is an instance of the book class and this is equivalent to an object. An object is an instance of a class.

2.3.3 Encapsulation and Interfaces

Encapsulation refers to the process of hiding or encapsulating the implementation details of an object. A washing machine is a good example of an encapsulated object. We know that inside a washing machine are a whole series of complex electronics, however, we don't need to be able to understand them to wash our clothes. In fact if the nitty gritty electronics were exposed to us we might even be afraid to use it and the washing machine would be more fragile and in danger of breaking down. This was one of the disadvantages of procedural programming; the user of the procedure could break the procedure by giving it the wrong type of data.

In terms of our concept of an object, encapsulation hides the properties, some methods, all method implementation details of an object from the outside. For example, the velocity of a car cannot be magically changed, we have to press the accelerator or brake (methods that we don't need to know the details of), in this respect the velocity of the car is hidden from outside interference, but can be changed by visible methods

An interface is a simple control panel that enables us to use an object. In the case of a washing machine the interface consists of the powder drawer, the door, the program knob and the on/off switch. For a car we have the steering wheels, clutch, brake accelerator etc. The great benefit of the interface is that we only need to understand the simple interface to use the washing machine or car in order to use them. This is much easier than understanding the internal implementation details. Another benefit is that the implementation details can be changed or fixed and we can still use the car or washing machine. For example, Suppose your car breaks down and you take it to the garage and they replace the engine with a bigger and better engine. The car operates in exactly the same way as the interface has remained constant. Thus it is extremely important to design a good interface that will not change. The inner workings can be tinkered with and cause no external operational effect.

Taken together the encapsulation and interface concepts unite to produce another benefit. Once the user understands the interface, the implementation details

can be modified, fixed or enhanced, and once the interface is unchanged the user can seamlessly continue to use the object. Following our example, if you buy a new washing machine with a turbo charged spin cycle and a special vibrating drum you know how to use it, provided the manufacturer sticks to the traditional washing machine interface. Once you know how to drive one car you can drive them all as the interface remains constant.

2.3.4 Data Abstraction

Abstraction is a general concept in computer science and means disregarding the details of an implementation and focusing instead on the ideal being represented. For example, consider what happens when you click on the print button in a word processing application. It is possible to simply imagine some process which moves the contents of the document to the printer, which then prints the document. It would be complex and confusing to think about the actual software being executed on your computer, the network server and the printer in order to perform printing. Instead, an abstraction of the printing process is imagined.

Data abstraction is the process of creating an object whose implementation details are hidden (i.e. encapsulated), however the object is used through a well-defined interface. Data abstraction leads to an abstract data type (ADT). ADT's are objects whose implementation is encapsulated. For example, when you use a floating point number in a program you don't really care exactly how it is represented inside the computer, provided it behaves in a known manner.

ADT's should be able to be used independent of their implementation meaning that even if the implementation changes the ADT can be used without modification. Most people would be unhappy if they took their car to the garage for a service and afterwards the mechanic said "She's running lovely now, but you'll have to use the pedals in reverse". If this were the case the car would not be an ADT. However, the reality is that we can take a car to a garage for a major overhaul (which represents a change of implementation) and still drive it in exactly the same way afterwards.

2.3.5 Problem Solving with Objects and Object Interactions

OOP is essentially an approach to computer programming and problem solving. Pretty much everything can be modelled as an object. OOP focuses the mind on data and what needs to be done with that data to solve a particular problem.

In this section lets consider a simple example of modelling a queue in a bank. Usually we don't model situations just for the laugh and there is some reason for doing the work. Lets suppose the bank manager wants to be able to deploy his staff effectively so that customer waiting time is minimised. This immediately tells us that the waiting time of customers is the key piece of data we need to get out of the model. Lets assume that there are four positions in the bank for serving customers. We'll call these service stations and name them SS1 to SS4. Obviously between 0 and 4 of these service stations might be operational at any given time, depending on customer demand, staff lunch breaks and a whole host of other factors. When a

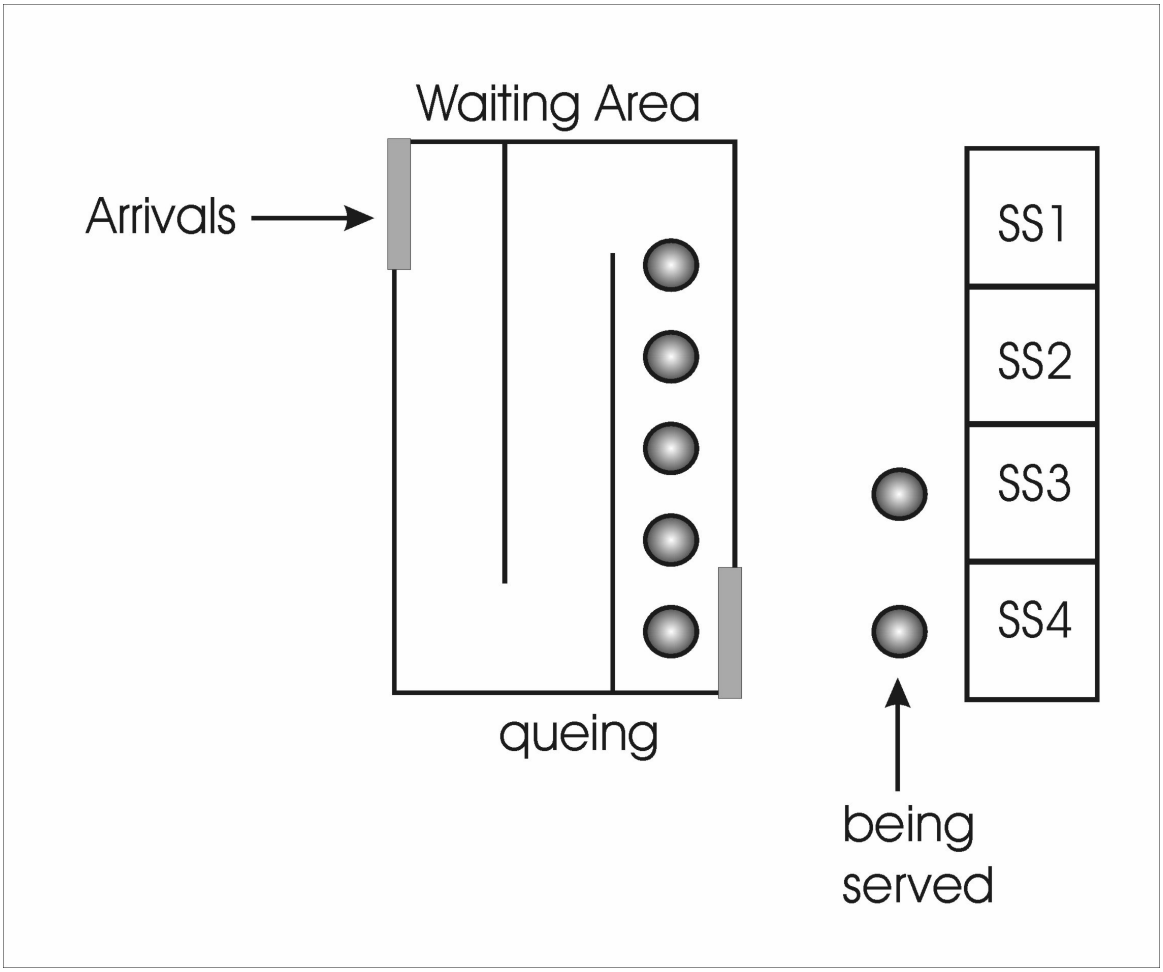

Figure 2.3.1 Abstract view of the bank queuing problem

customer arrives, he goes to an unoccupied service station, otherwise he has to wait in line. This again prompts some questions: what is the arrival pattern? Constant, varied, lunch-time etc. How long does it take to get your business done once you get to a vacant service station? The time spent being served is called the service time. A schematic illustration of the problem is given in Fig. 2.3.1.

There are some obvious objects in this scenario. Starting at the beginning of the process, there must be something which generates arrivals. In reality this is very complex as who knows why somebody decides to go the bank at any particular time. We can probably use a random number generator (and there is a Random object in .NET for this very purpose). However, because we are taking an OO approach, we needn't be overly concerned with the implementation details provided we get the interface and functionality right! Our Arrivals object basically creates customers who arrive at a particular time. Customers are a likely object as well, they would be aware

of the time they arrive, the time they get to a service station and the time they leave the bank. This is enough information to calculate the total time queuing etc. for every customer. The waiting area is an object which stores a list of customers in first come-first served order. We might be interested in knowing who is the next customer to be served? How many customers are waiting at any given time? A service station is also a likely candidate for being an object. It can be either active or inactive. It can have an average service time (i.e. some staff members may be slower than others).

Next we need to consider how the system will work in order to simulate the queuing situation. This involves interactions between the objects discussed above. When a customer object arrives it has to either go to the back of queue or go to an available service station. Each active service station object asks the waiting area for the next customer and serves that customer, if there are no more customers then it waits for a customer to arrive. We can make this problem more complicated by allowing service stations to shut down and start up during the day.

2.3.6 Inheritance

We live in a world that can be considered to be comprised of countless objects, but how do we think about objects? Do we treat each one on an individual basis? No, we tend to group objects into categories and in turn these form a hierarchy. For example consider a book. Lots of objects can be grouped together and called book objects. Computer books, school books, science fiction books, pulp fiction books are all different types of books which can all be described as books. A book is also part of a more general group called publications, in addition to books, publications include scientific journals, newspapers, comics, magazines etc. A graphical representation is given in figure **??**.

Books have certain attributes or properties that are shared by all books: They have a cover, chapters, no advertising, and so on. They also have attributes shared by publications in general: a title, a date of publication, a publisher, etc. They have attributes that are shared by all physical objects: a location, size, shape, and weight. This idea of shared attributes is very important in OOP. OOP models the concept of shared attributes using inheritance.

Each object has associated with it a set of actions that may be performed with or to it. This set of actions varies from object to object. For example, you can read a book, flip its pages, count its pages etc. These actions are largely unique to publications: you cannot flip the pages of a hammer or a chisel, for example. However, there are actions that are generic to all physical objects, such as picking them up. OOP takes this fact about the world into account as well, again using inheritance.

Objects therefore consist of a set of attributes (generally referred to as properties in object-speak) and a set of actions (termed methods in object-speak). To consolidate these concepts lets consider the car object. Its properties include colour, manufacturer, year of registration, registration number, value, velocity etc. The following methods apply to a car: open a door, switch on the engine, press clutch, press accelerator, change gear etc. A car is a good example of an object. The values which

properties contain are said to define the state of an object and methods can be used either to examine or change the state of an object.

Object hierarchies arrange objects in such a way that general objects are at the top and specialised objects are at the bottom. Considering objects in this way it is soon noticed that objects share properties and methods. Inheritance supports this hierarchical-shared view by allowing new classes (of objects) to be created simply by specifying the differences between the new class and a pre-existing more general class. In object-speak the new specialised class, termed the descendant class, is said to be inherited from the pre-existing more general class termed the ancestor class. In simple terms inheritance means that descendant classes automatically get the properties and methods of the ancestor class. The primary advantage of inheritance is that we can reuse the properties and methods of pre-existing classes. Reuse is an important concept in software engineering because tried and tested ancestor classes provide a solid foundation for future projects which can proceed rapidly and with confidence. Another powerful advantage of inheritance is that modifications applied to methods in ancestor classes are automatically applied to all inheriting classes. This greatly simplifies software maintenance.

2.3.7 Polymorphism

Probably the most exotic sounding and commonly feared part of object orientation is polymorphism, which literally means poly (many) and morph (forms). Objects interact by calling each others methods. How does some object A know the supported methods of another object B, so that it can call a valid method of B? In general there is no magic way for A to determine the supported methods of B. A must know in advance the supported methods of B, which means A must know the class of B. Polymorphism means that A does not need to know the class of B in order to call a method of B. In other words, an instance of a class can call a method of another instance, without knowing its exact class. The calling instance need only know that the other instance supports a method, it does not need to know the exact class of the other instance. It is the instance receiving the method call that determines what happens, not the calling instance.

Chapter 3
DATA TYPES, VARIABLES, I/O AND OPERATORS

3.1 Introduction

This is a fairly long chapter and introduces many of the basic requirements for writing programs in C#. Most of this material has nothing to do with OOP and applies to all programming languages, although the syntax will vary. Data is the fundamental currency of the computer. All computer processing deals with analysis, manipulation and processing of data. Data is entered, stored and retrieved from computers. It is not surprising then, to learn that data is also fundamental to the C# language. Humans are very clever at dealing with data and we tend to automatically know what kind of data we are dealing with from context. Computers are not so clever and we have to be very precise when specifying data.

In this chapter we look at the data types supported by C#, and how real life data is modelled by these data types. Variables allow us to represent data elements within a program so that the data can be manipulated. We will see how to get data in and out of a computer using the keyboard and screen and finally we take a look at the multitude of operators, which allow us to modify and process data.

3.2 Data Types

3.2.1 Introduction

C# is a strongly typed language, that is, every object or entity you create in a program must have definite type. This allows the compiler to know how big it is (i.e. how much storage is required in memory) and what it can do (i.e. and thereby make sure that the programmer is not misusing it). There are thirteen basic data types in C# and their characteristics are given in Table 3.2.1, note that 1 byte equals 8 bits and each bit can take one of two values (i.e. 0 or 1).

These data types are commonly referred to as intrinsic data types because the are the simplest data types in C# and all other data types are ,in some not always obvious way, composed of these basic data type. The following data types take integer values only: byte, sbyte, short, ushort, int uint, long, ulong. Whereas float and double accept floating point numbers (i.e. real numbers).

Data Type	Storage	.NET Type	Range of Values
byte	1	Byte	0 to 255
char	2	Char	unicode character codes (0 to 65,535)
bool	1	Boolean	True or false
sbyte	1	SByte	-128 to 127
short	2	Int16	-32,768 to 32767
ushort	2	UInt16	0 to 65,535
int	4	Int32	-2,147,483,648 to 1,147,483,647
uint	4	UInt32	0 to 4,294,967,295
float	4	Single	$\pm 1.5 * 10^{-45}$ to $\pm 3.4 * 10^{38}$, 7 significant figures
double	8	Double	$\pm 5.0 * 10^{-324}$ to $\pm 1.8 * 10^{308}$, 15 significant figures
decimal	12	Decimal	for financial applications
long	8	Int64	$-9*10^{18}$ to $9*10^{18}$
ulong	8	UInt64	0 to $1.8*10^{19}$

Table 3.2.1 Basic C data types. Storage is in bytes.

3.2.2 Data type char

Although char appears to hold a restricted range of integers, i.e. whole numbers, a char is used to represent a single alphanumeric character, such as 'A', 'a', 'B', 'b', '1','5' or ','. In C# a single character must be surrounded by single quotes. Unlike other computer languages such as C or C++, C# does not allow you to interchangeably use integer values and character values in a char. This means I cannot hold the number 10 in a char, but I can hold the character 'A'.

3.2.3 Integer Data Types

These data types are an exact representation of an integer value provided that the value is inside the accepted range for the data type, for example trying to store the value 300 in a byte will cause a compiler error (see Table 3.2.1).

3.2.4 Data Types double and float

Data represented by doubles and floats may contain a fractional part i.e. a decimal point and their ranges are given in table 3.2.1. You make ask at this stage, why have different numeric data types to store numbers? Can we store all numbers in type double and forget about the complexities described above? The answer is yes, but for the following reasons it is not a good strategy:

1. operations involving integer data types are generally faster.
2. less memory is required to store an integer data types (sometimes, depending on you needs).
3. operations with int types are always precise (see below for some exceptions), whereas round-off errors may be associated with operations involving doubles and floats.

3.3 Variables

3.3.1 Introduction

The memory locations used to store a program's data are referred to as variables because as the program executes the values stored tend to change. Each variable has three aspects of interest, its:

1. type.
2. value.
3. memory address.

The data type of a variable informs us of what type of data and what range of values can be stored in the variable and the memory address tells us where in memory the variable is located. It is important to think of variables as pieces of memory that store values of a certain data type. It is conceptually valuable to think about memory in the abstract. We can think of memory as a long contiguous sequence of spaces where variables can be stored. Each space has an address associated with it and we can arbitrarily assume they are label from 0 up to the capacity of the target machine (we don't need to worry about the implementation details i.e. the actual electronics or what memory actually is in reality). In actuality some variables take up more space than others but again lets not worry about that for know (perhaps we could imagine stretchy spaces that expand to hold a larger variable. The size required for variable depends on its data type.

3.3.2 Declaration of Variables

In C# before a variable can be used, it must be declared. Declarations associate a specific data type with each variable that is declared, and this tells the compiler to set aside an appropriate amount of memory space to hold values associated with the variable. In general, each code block (a piece of code between two curly braces {}) consists of declarations followed by C# statements. Declarations must precede statements. All statements in C# are terminated with a semi-colon.

All C# variables are declared in the following manner:

Syntax: `<type> <name>;`

Example

```
int i;
char a, b, ch;
```

In the first case only a single integer variable i was declared, whereas in the second statement three char variables were declared in one go. In this case each variable must be separated from the other by a comma.

3.3.3 Naming of Variables

The names of variables and functions in C# are commonly called identifiers. There are a few rules to keep in mind when naming variables:

1. The first character must be a letter or an underscore.
2. An identifier can consist of letters, numbers and underscores only.
3. Reserved words (e.g. words that already have a special purpose int he language such as int, char and double) cannot be used as variable names.

In addition, please note carefully that C# is case sensitive. For example, the identifiers Rate, rate and RATE are all considered to be different by the C# compiler. This is a common source of compiler error and programming error.

It is well advised to decide to use lowercase letters only for variable naming. It is also a good idea to give variables meaningful names, but without overdoing it.

EXAMPLE

```
int this_variable_holds_the_radius; //bad
double radius; //good
```

3.3.4 Initialising Variables and Assigning Values to Variables, C# requires definite assignment

When a variable is declared in a code block, memory to store its value is allocated, however, a value is not given to it. In fact the value is random and can be any number, there is no guarantee as to what value is in the variable. In order to be certain of the value you must initialise it, that is, assign a value to the variable. This can be achieved in one of two ways:

Initialise during variable declaration

Syntax: type var_name = constant;

EXAMPLE

```
int i = 20; //i declared and given the value 20
char ch = 'a' //ch declared and initialised with value 'a'
int i = 2, j = 4, k, l = 5; //i, j and l initialised, k not initialised
```

Declare first then assign

EXAMPLE

```
int i, j, k; //declare
i = 2; //assign
j = 3;
k = 5;
```

Type(s)	**Category**	**Suffix**	**Example**
bool	Boolean	none	true or false
int, uint, long, ulong	Integer	none	100
uint, ulong	Integer	u or U	100U
long, ulong	Integer	l or L	100L
ulong	Integer	ul, uL, Ul, UL, lu, lU, Lu, LU	100UL
float	Real	f or F	1.5f
double	Real	none	1.5
char	Character	none	'a' or '\n'
string	String	none	"hello"

Table 3.3.1 Literal constants in C.

It is clearly more concise to simply initialise variables. Assigning values to variables is done using the assignment operator =. Note that in C# definite assignment is required, this means that if you don't assign a value to a variable in advance of using the variable a compiler error is generated.

3.3.5 Constants

There are three types of constants available in C#: literal, symbolic and enumerations. Literals are most frequently used.

Literal constants

Literal constants are fixed values that cannot be altered by the program and may be numbers, characters or strings, see Table 3.3.1 for some examples. Typically constants are used to initialise variables or as data passed to objects.

The final literal constant type in Table 3.3.1 is string. This is not an intrinsic data type as it is made up of a sequence of characters, although unlike C and C++, it comes as part of the C# language. In effect a string is an objects whose data consists of a list of characters. String literals are an important component of most programs and even though the string object will be discussed in more detail later, it is worth considering string literals at this stage. First of all, not every character has a visible representation (for example a tab or a return), these characters and some other commonly used elements in programming (the single and double quote mark) are given special character sequences to represent them called escape sequences (see Table 3.3.2). Note that the string constant is enclosed in double quotes whereas a character literal is enclosed in single quotes.

For example, suppose you wish to have a string literal representing: "Hello", he said. Because double quotes have a special meaning in C# (i.e. enclosing string literals), then putting this sentence into a string literal would be tricky without the help of escape sequences. First of all take the string and put in an escape sequence of each double quote to get: \"Hello\", he said and then enclose the whole lot in double quotes to get: "\"Hello\", he said".

Escape Sequence	Meaning
\0	null character
\a	audible alert
\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\’	single quote
\”	double quotes
\\	back slash
\?	question mark

Table 3.3.2 Escape sequences in C and their meaning.

There is a short cut for string literals containing lots of escape sequences (however double quotes in string literals always need to be referred to by an escape sequence). For example to store a file name such as: c:\temp\mystuff\text.txt would be converted to "c:\\temp\\mystuff\\text.txt" as a string literal. To avoid having to put in all the escape sequences we can make the string literal verbatim by prefixing it with the @ symbol. Therefore in our example the string literal "c:\\temp\\mystuff\\text.txt" is exactly equivalent to @"c:\temp\mystuff\text.txt". Obviously verbatim string literals are optional.

Symbolic constants

Symbolic constants assign a name to a constant value. The syntax is:
const data-type identifier = value;
Although this may look like a typical variable declaration, it is not. You must initialise a symbolic constant and after initialisation the value cannot change. For example:

```
const int maxiters = 10000;
maxiters = 1000; //fails because maxiters is a constant
```

The value of symbolic constants is in code readability and because they are initialised at a single location then changing the value of the constant there, changes it everywhere it is used.

Enumerations

Enumerations are a handy way to group symbolic constants (if it makes logical sense). For example you may have an application which uses a series of temperatures. Rather than declaring each relevant temperature constant as a symbolic constant they can be grouped together in an enumeration as they are all related. For example:

```
enum Temperatures
{
VeryCold = -20;
Freezing = 0;
Warm = 18;
Hot = 25;
Roasting = 35;
}
```

We could then make use of these constants as follows:

```
Console.WriteLine("Sometimes the temperature drops to {0}", Temperatures.VeryCold);
Console.WriteLine("But usually is a comfortable {0}", Temperatures.Warm);
```

3.4 Console Input/Output (I/O)

3.4.1 Introduction

Console applications (also known as DOS applications) are not very common these days and most applications are written for the windowed environment running under Windows or Unix. That being said, console applications are fairly easy and allow us to concentrate on the language and OO features without investing heavily in the look of the output. In this section we briefly consider how we get data into and out of our console application. In doing this we will be introduced to two objects from the .NET framework: the Console Object and the Convert object. This highlights one of the benefits of OOP. We can just learn how to use somebody else's objects without knowing the nitty gritty details. Provide these objects work and behave as expected, then our development time is greatly reduced.

Both the Console and Convert objects are automatically created and ready for use in our programs without having to explicitly create them ourselves. This is because they are such commonly used and useful objects. The Console object is an abstraction of process which connects the program to the keyboard (for input) or the screen (for output). It is clear that the details are going to be very complicated (first off, think of all the different screens and keyboards there are), but we insulated from this with the Console object.

3.4.2 Output

We will only consider a subset of the functionality of the Console object as we will be mainly writing windowed applications. For output to the screen the most useful method of the Console class is the WriteLine method. Its syntax is as follows:

Syntax: `Console.WriteLine(<control_string>,<optional_other_arguments);`

This means that this method expects to be passed at least one argument, the control string. For example:

```
Console.WriteLine("Hello World!");
```

In this case we are outputting a string literal which cannot change. However, we usually want to output the result of a calculation, which can only be known during

the execution of the program. This is where the optional arguments come in. These optional arguments are a series of one or more values that need to be incorporated into the output. For example, we may wish to calculate the average of three numbers (say 2,3,4) the we expect to able to output to the screen "The average is 3". Each additional optional argument is number from zero up. Consider the following code snippet:

```
int a = 2, int b = 3, c = 0;
c=a+b;
Console.WriteLine("c has the value {0}", c);
Console.WriteLine("{0} + {1} = {2}", a, b, c);
```

Here the symbols {0}, {1} etc. are placeholders where the values of the optional arguments are substituted. In the first statement the value of c (the zeroth optional argument) is substituted instead of {0}. In the second statement a is the zeroth, b the first and c the second optional argument. Therefore the value of a is substituted for {0}, that of b for {1} and that of c for {2}.

WriteLine automatically outputs a newline at the end of its output. We could get very sophisticated about formatting the output but as console applications are only for teaching purposes we won't go to town on it!

3.4.3 Input

An important part of any program is getting input from the user. This usually takes the form of prompting the user to input data (using WriteLine), taking the data input by the user and placing it in a variable for later use in the program. The most commonly used method of the Console object is the ReadLine method. The syntax is very simple:

Syntax: `string Console.ReadLine();`

The string before the method means that whatever the user types on the keyboard is returned from the method call and presented as a string. It is up to the programmer to retrieve that data. An example is:

```
string input = "";
int data = 0;
Console.WriteLine("Please enter an integer value:  ");
Console.ReadLine(); //user input is stored in the string input.
data = Convert.ToInt32(input);
Console.WriteLine("You entered {0}", data);
```

In this simple piece of code we take a value from the user and use the Convert Object to convert the input from a string to an integer. Note that you have no control over what the user inputs. If you are expecting an integer and they type in something else then your program will fail because the Convert object fails in its task. Later on we will see how to cope with these situations (called exceptions), for the moment, be careful!!

3.5 Operators

A strong feature of C# is a very rich set of built in operators including arithmetic, relational, logical and bitwise operators. In this section the most important and commonly used operators are introduced.

3.5.1 Assignment =

Syntax: <lhs> = <rhs>;

where lhs means left hand side and rhs means right hand side. The assignment operator copys the value of the rhs variable to the variable on the lhs. In C# the assignment operator returns a result, i.e. the value assigned. We will see the importance of this property later.

EXAMPLE

```
int i, j, k;
i = 20; // value 20 assigned to variable i
i = (j = 25); /* in C#, expressions in parentheses are always evaluated first, so j is assigned the value 25 and the result of this assignment (i.e. 25) is assigned to i */
i = j = k = 10;
```

The last statement of the above example demonstrates right to left associativity of the assignment operator, i.e. in the absence of parentheses the assignment operator is evaluated right to left and the return value of the assignment operator is passed to the left. Hence the statement in question is equivalent to

```
i = (j = ( k = 10 ) );
```

After this statement i, j and k all have the value 10. This is handy for initialising many variables simultaneously.

Implicit and Explicit Conversion

Assignment is pretty straightforward when both the lhs and rhs are of the same type. However, if this is not the case, the value of the rhs is converted to the type of the lhs. This can cause errors during compilation. The compiler can deal with mismatched types in certain cases by performing an automatic or implicit conversion. This requires no effort on the programmers part.

EXAMPLE

```
int x = 10;

char c = 'a';

float f;
```

Type	Can be implicitly converted to:
byte	short, ushort, int, uint, long, ulong, float, double
sbyte	short, int, long, float, double
short	int, long, float, double
ushort	int, uint, long, ulong, float, double
int	long, float, double
uint	long, ulong, float, double
long	float, double
ulong	float, double
float	double
char	ushort, int, uint, long, ulong, float, double
\\	back slash
\?	question mark

Table 3.5.1 Implicit conversions by the compiler.

```
c = x; // not allowed possible loss of information

x = c; // fine but check the value placed in x!

x = f; // not allowed possible loss of information

f = x; // value of x converted to a floating point
```

Explicit conversion, on the other hand, places the responsibility for making sure conversions are sensible with the programmer. In tis case we override the compiler objects using a cast. IN the previous example neither x = x nor x = f were allowed by the compiler. To override these objects (i.e. you are essentially sayings thanks for the warning but I'm doing it anyway!!) the code would be as follows:

```
c = (char) x;
f = (float) x;
```

These are called casts. Casts will not work in every situation.

*3.5.2 Arithmetic Operators (+, -, *, /, %)*

Introduction

\+ addition
\- subtraction
* multiplication
/ division
% modulus

\+ and - have unary and binary forms, i.e. unary operators take only one operand, whereas binary operators require two operands.

EXAMPLE

```
x = -y; // unary subtraction operator
p = +x * y; // unary addition operator
x = a + b; // binary addition operator
y = x - a; // binary subtraction operator
```

Rules for Evaluation of Arithmetic Expressions

These rules are equivalent to the standard rules of mathematics and are included for completeness:

All parenthesised sub-expressions must be evaluated separately and are evaluated first. Nested parenthesised sub-expressions must be evaluated inside out, with the innermost expression evaluated first.

Operator precedence implies that operators in the same sub-expression are evaluated in the following order:

1. unary +, -
2. *, /, %
3. binary +, -

Associativity rules. Unary operators in the same expression at the same precedence level are evaluated right to left (right associativity). Binary operators in the same expression at the same precedence level are evaluated left to right (left associativity).

If you stick to these rules, you will never come to any harm and your expressions will be evaluated as expected. However, adding parentheses gives expressions more readability and reduces the possibility of error.

EXAMPLES

```
x = 7 + 3 * 6 / 2 - 1 // x = 15, think about it
x = 7 + ( 3 * 6 / 2 ) - 1 // x = 15, a bit clearer
x = (7 + 3) *6 / (2 - 1) // a completely different way of evaluation
```

3.5.3 Integer Division

When applied to two positive integers, division returns the integral part of the result of the division. So 7/2 is 3.5, the integral part of which is 3, therefore in integer division 7/2 equals 3. This can be a source of errors in a program. Always check for integer division and make sure that's what you want to occur.

EXAMPLES

```
3/15 = 0        18/3 = 6
15/5 = 5        16/-3 varies
16/3 = 5        0/4 = 0
17/3 = 5        4/0 undefined
```

Operator	Meaning
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

Table 3.5.2 Relational operators in C.

3.5.4 The Modulus Operator

The modulus operator returns the remainder of the result of dividing the lhs by the rhs. Typically the modulus operator is applied only to integers, however, unlike many other languages, you are free to use the modulus operator with floating point data types as well in C#. Therefore 7%2 equals 1 and 11.1%2.5 equals 1.1

3.5.5 Increment and Decrement operators (++, - -)

Increment (++) and decrement (- -) are unary operators which cause the value of the variable they act upon to be incremented or decremented by 1 respectively. These operators are shorthand for a very common programming task.

EXAMPLE

```
x++; //is equivalent to x = x + 1;
```

++ and - - may be used in prefix or postfix positions, each with a different meaning. In prefix usage the value of the expression is the value after incrementing or decrementing. In postfix usage the value of the expression is the value before incrementing or decrementing.

EXAMPLE

```
int i, j = 2;
i = ++j; // both i and j have the value 3
i = j++; // now i = 3 and j = 4
```

*3.5.6 Special Assignment Operators (+=, -=, *=, /=, %=, &=, >>=, <<=, |=, ^=)*

Syntax: lhs operator= rhs

which is equivalent to lhs = lhs operator (rhs)

EXAMPLE

```
x += i + j; // this is the same as x = x + (i + j);
```

These shorthand operators improve the speed of execution as they require the expression and variable to be evaluated once rather than twice.

3.5.7 Relational Operators

These operators are typically used for comparing the values of two expressions..

There is a common C# programming error associated with the equal to (==) operator and even though you will be forewarned, no doubt you too will make this

mistake. To test if a variable has a particular value we would write:
`if ( x ==10 )` etc.,
however, commonly this is inadvertently written as:
`if (x = 10)` i.e. only one equals sign,
which is the *assignment* operator.

Unlike C and C++, this statement causes a compilation error in C# because the result of x = 10 is an int whereas the if statement is expecting a bool and the compiler cannot implicitly convert int to bool.

3.5.8 Logical Operators

&& is the logical AND, || is the logical ORand ! is the logical NOT
EXAMPLE
if (x >= 0 && x < 10)
Console.WriteLine("x is greater than or equal to 0 and less than 10");
EXAMPLE

```
2 > 1; //TRUE value true
2 > 3; //FALSE value false
i = 2 > 1; //TRUE value true assigned to i (which must be declared as a
bool)
```

Every C# expression has a value, remember that even the assignment operator returns a value. Thus the expression 2 > 1 has a value of 1 which may be assigned. Consider the following piece of code:

```
int i = 0;
if (i = 0)
i = 1;
else
i = -1;
```

What value does i have? Will the program compile?

Care must be exercised when using the logical operators and can be the source of many errors. You are well advised to test logical conditions with real data as it is very easy to get confused.

3.5.9 Bitwise Operators

Introduction

These are operators that apply to integer data types only. They allow the programmer to get closer to the machine (or low) level by allowing direct manipulation at the bit-level.

Bitwise operators are commonly used in system level programming where individual bits represent real-life entities that are on or off, i.e. 1 or 0, true or false. For example, specific bits are often set to request various operations such as reading or writing from a hard-drive. When these operations complete other bits may be set to indicate success or failure. It is therefore necessary to be able to manipulate bits and C# supports this activity. Before continuing lets refresh our memories about

Operator	Meaning
&	bitwise AND
|	bitwise OR
^	bitwise XOR
~	complement
>>	shift right
<<	shift left

Table 3.5.3 Bitwise operators in C.

converting from decimal (i.e. base 10) numbers to binary numbers (i.e. base 2).

Converting from Decimal to Binary and vice versa

Decimal numbers are to the base 10 and consist of the digits (0 - 9), binary numbers are to the base 2 and consist of the digits 0 and 1 only. We are familiar with the meaning of decimals but less so with binary numbers.

Consider the decimal number 142, how is this number composed? Well it consists of 2 ones, 4 tens and one hundred. In table form this is:

Row	100's	10's	1's	
A	10^2	10^1	10^0	
B	1	4	2	
A*B	100	40	2	**Total: 142**

Using a similar table we can convert binary to decimal

Row	8's	4's	2's	1's	
A	2^3	2^2	2^1	2^0	
B	1	1	0	1	
A*B	8	4	0	1	**Total: 13**

Thus the decimal equivalent of 1101 is 13.

To convert from decimal to binary find the highest power of 2 that divides into the number.

EXAMPLE

Convert 1278 into binary. 2^{10} (=1024) goes in once with 254 left over. Next find the highest power of 2 that goes into the remainder 254 which is 2^7. This process is then repeated until the full binary is found. The full binary representation is an 11 digit binary number 10011111110.

There are other simpler methods to convert from decimal to binary. One approach is to simply divide the decimal value by 2, write down the remainder and repeat this process until you cannot divide by 2 anymore, for example let's take the decimal value 157:

$157 \div 2 = 78 \quad with a remainder of 1$

$78 \div 2 = 39 \quad with a remainder of 0$

$39 \div 2 = 19 \quad with a remainder of 1$

$19 \div\ 2 = 9 \quad with a remainder of 1$
$9 \div\ 2 = 4 \quad with a remainder of 1$
$4 \div\ 2 = 2 \quad with a remainder of 0$
$2 \div\ 2 = 1 \quad with a remainder of 0$
$1 \div\ 2 = 0 \quad with a remainder of 1$
$Next write down the value of the remainders from bottom to top (in other words write down the bottom remai$
10011101 = 157.

C# Data Types and bits

A byte is stored in one byte, which is composed of 8 bits. Each bit can contain a 1 or a 0. Hence the maximum number which can be stored in a byte is a 8 digit binary number 11111111 or 255, alternatively if one bit indicates sign the range is from -127 to 128 i.e. -1111111 to 1111111.
Bits are viewed as follows

Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
0	0	0	0	1	1	0	1

This is a bitwise representation of the decimal number 13. Bit 0 is normally termed the Least Significant Bit (LSB) and Bit 7 the Most Significant Bit (MSB). Bitwise operators also work with larger integer data types (i.e. int, which is represented by 32 bits). For the sake of simplicity we deal only with the byte data type here.

The most important thing to remember about bitwise operators is that they operate on the corresponding bits of the operands.

Bitwise AND, &

For this and all binary bitwise operators, the n^{th} bit of the lhs operand is compared with n^{th} bit of the rhs operand. The following truth table indicates the effect of &.

lhs	**rhs**	**result**
1	1	1
1	0	0
0	1	0
0	0	0

EXAMPLE

```
    10110010  (178)
 &  00111111  (63)
-------------------
    00110010  (50)
```

Bitwise OR, |

Truth table

lhs	rhs	result
1	1	1
1	0	1
0	1	1
0	0	0

EXAMPLE

```
    10110010  (178)
 &  00001000  (8)
-------------------
    10111010  (186)
```

Bitwise XOR, ^

Truth table

lhs	rhs	result
1	1	0
1	0	1
0	1	1
0	0	0

EXAMPLE

```
    10110010  (178)
 &  00111100  (60)
-------------------
    10001110  (142)
```

Bitwise Shift, << and >>

These operators move all the bits of a variable either to the left or the right and pads new bits with 0's.

Syntax: var_name << number_of_places
var_name >> number_of_places

EXAMPLE

`2<<2 //= 8 as 00000010 becomes 00001000`

In general, a left shift of n places multiplies by 2^n, and a right shift of n places divides by 2^n.

Bitwise Complement, ~

Reverses the state of each bit.

Truth table

Bit state	result
1	0
0	1

EXAMPLE
```
~10110010 (178) //= 01001101 (77) i.e.  ~178 = 77
```

Masks

A mask is a variable, which allows us to ignore certain bit positions and concentrate the operation only on bits of specific interest. The value given to a mask depends on the bitwise operator involved and the result required.
EXAMPLE
To clear the 7th bit, set it equal to false
```
byte b = 136, mask = 127; // 136 = 10001000 and 127 = 01111111
b = b & mask; //b now has value 00001000
```
To set the 1st bit to true
```
byte i = 136, mask = 2; //136 = 10001000, 2 = 00000010
i |= mask; //the value of i is now 10001010
```

3.6 Type Overflow, Type Underflow and Numerical Inaccuracies

3.6.1 Overflow and Underflow

When the value to be stored in variable is larger than the maximum value allowed we have type overflow. Similarly, if the value to be stored is smaller than the minimum value permitted we have type underflow.

Overflow and underflow is most problematic when dealing with integer variables. By default C# ignores the error in this case and continues as if a problem did not occur. Consider the following EXAMPLE
```
byte dest = 1;
short source = 281;
dest = (byte)source;
Console.WriteLine("source is {0}", source);
Console.WriteLine("dest is {0}", dest);
Console.ReadLine();
```
What is the output? Note there are no compiler or runtime errors. Why did this happen? Lets look at the binary numbers:
```
281 = 100011001 (note nine significant bits)
25 = 000011001 (if we put 281 into a byte)
255 = 011111111
```
Because a byte has only 8 bits then the ninth bit of the short is lost giving 25. We can force these errors to be flagged by enclosing the (potentially) offending code in checked(), as follows:
EXAMPLE
```
byte dest = 1;
short source = 281;
dest = checked((byte)source);
```

```
Console.WriteLine("source is {0}", source);
Console.WriteLine("dest is {0}", dest);
Console.ReadLine();
```

This code now results in an exception being raised. We will get on to dealing with exceptions later. We can also wrap the statement in unchecked() which gives the same result as the first example above.

If we do the same with a signed byte (sbyte) we also have the possibility of getting a negative number because the sign bit may be overwritten.

EXAMPLE

```
sbyte dest = 1;
short source = 155;
dest = (sbyte)source;
Console.WriteLine("source is {0}", source);
Console.WriteLine("dest is {0}", dest);
Console.ReadLine();
```

Floating point (floats and doubles) overflow and underflow is not a problem as the system itself is informed when it occurs and causes your program to terminate with a run-time error. If this happens simply promote your variables to the largest possible type and try again.

3.6.2 Numerical Inaccuracies

Type double cannot always accurately represent real numbers, just as decimals cannot always accurately represent fractions (e.g. 1/3 becomes 0.33333). For certain real numbers the fractional part is not well represented and leads to representational (or round-off) errors.

EXAMPLE

```
double d = 0.1, e;
e = (d*1000) - (0.1*999) - 0.1;
Console.WriteLine("The value of e is {0}", e); // it should be 0 but it's
not
```

Because of this error equality comparisons of two doubles can lead to surprising results. In the above example the magnitude of a small error is enlarged by repeated computation.

Other problems occur when manipulating very large and very small real numbers. When a very large number and a very small number are added, the large number may cancel out the small number. This is termed a cancellation error. So if x much greater in value than y then x + y has the same value as x.

EXAMPLE

```
float x = 1.234e+24, y = 1.234e-24, z;
z = x + y;
Console.WriteLine(" The value of z is {0}", z); // z has the same value
as x
```

Try the following code for solving a quadratic equation:

```
float a=1,b=100000f,c=4f;
float root1, root2,q;
//standard way
root1 = (-b + (float)Math.Sqrt(b*b-4*a*c))/(2*a);
root2 = (-b - (float)Math.Sqrt(b*b-4*a*c))/(2*a);
Console.WriteLine(" The value of root1 is {0}", root1);
Console.WriteLine(" The value of root2 is {0}", root2);
//numerically sound way
q = -(b+(float)Math.Sign(b)*(float)Math.Sqrt((b*b-4*a*c)))/2;
root1 = q/a;
root2 = c/q;
Console.WriteLine(" The value of root1 is {0}", root1);
Console.WriteLine(" The value of root2 is {0}", root2);
Console.ReadLine();
```

Try to solve this problem in mathematica (using Solve), which way is most accurate. This occurs because if either a or c are small, then the solution involves the subtraction of b from a very nearly equal quantity.

Exercise 1 *Declare two char variables (a and b), and one int variable (i). Assign the value 97 to a, the character '0' to b and the number 1054 to i. Print the values of a and b so that they come out as actual numbers and also print them as characters. Print the value of i to the screen as well. Assign the value of i to a. Print it out as a number and as a character. Can you explain whats going on?*

Exercise 2 *Write a program which reads a single character from the keyboard and writes out its ASCII representation. Then write a program which reads in an integer from the keyboard and prints out its character representation. Make certain you carry out appropriate bounds/error checking.*

Exercise 3 *What value does x contain after each of the following where x is of type float.*

1. `x = 7 + 3 * 6 / 2 - 1 ;`
2. `x=2%2+2*2?2/2;`
3. `x=(3 *9*(3+(4*5/3)));`
4. `x = 12.0 + 2 / 5 * 10.0;`
5. `x=2/5+10.0*3-2.5;`
6. `x=15>10&&5<2;`

Exercise 4 *Write a program to read Fahrenheit temperatures and print them in Celsius. The conversion formula is C = (5/9)(F - 32). Use variables of type double in your program.*

Exercise 5 *Write a program that reads in the radius of a circle and prints the circle's diameter, circumference and area. Use the value 3.14159 for* π.

Exercise 6 *Write a program to read in two points and calculate the distance between them. Given 2 points* (x_1, y_1) *and* (x_2, y_2)*, the distance (d) between them is given by:* $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. *Note that square root is implemented in Math.Sqrt().*

Exercise 7 *Get the user to input an integer number of seconds (i.e. 5049). Write a program to convert this in hours, minutes and seconds. (The modulus operator will come in handy). For example 5049 = H:1 M:24 S:9.*

Exercise 8 *A drinks machine manufacturer requires software for dispensing change. One euro (100 cent) is the maximum that can be put into the machine. Given the amount inserted and the cost of the item devise the change required. For example suppose you put in 1 euro for an item costing 45 cent then your program should generate output as follows:*
Number of 50 cent coins : 1
Number of 20 cent coins : 0
Number of 10 cent coins : 0
Number of 5 cent coins : 1
Number of 2 cent coins : 0
Number of 1 cent coins : 0

Chapter 4
STATEMENTS

4.1 Introduction

This chapter investigates C#'s small selection of statements. We re-visit statements we have already used and describe in detail those which have not yet been presented. By the end of this chapter we should be in a position to start writing more interesting programs.

4.2 Expression Statements

Expression statements are C#'s simplest statements. An expression statement is quite simply an expression followed by a semi-colon.
syntax: expression;
We have already seen many examples of expression statements in the previous chapter such as:

```
x = 1;
```

This statement assigns the value 1 to the variable x. The statement consists of the assignment operator and the terminating semi-colon. The assignment operator is an expression. In general an expression consists of one or more of C#'s operators acting on one or more operands. However, unlike many programming languages, a function call is also treated as an expression so that: `Console.WriteLine("Hello World!");` is an expression statement also.

Multiple expressions can be strung together to form more complex statements:

```
x = 2 + (3 * 5) - 23;
```

Individual expressions inside in a complex expression are usually termed sub-expressions.

An expression is executed by evaluating the expression and throwing away the result. To be useful, it must have a side effect, such as invoking a function or assigning the result of an expression evaluation to a variable. Consider the following legal but useless statement:

```
x * p;
```

Here the value of p is multiplied by x, but nothing is done with the result. Compilers don't always warn about such superfluous statements, which are usually the result of typographical errors.

4.3 Compound Statements or Blocks

When writing programs we normally need to group several statements together such that they are executed sequentially. These collections of statements are referred to as compound statements or more commonly as code blocks. Blocks are enclosed in a pair of curly braces.

syntax:

```
{
statement
statement
statement
}
```

This syntax has already been encountered in the Hello World program in Chapter 1, whereby the statements comprising the main function are enclosed in curly braces.

4.4 Decision Statements

4.4.1 Introduction

In this section a class of statements are described which allow one or another block of code to be executed depending on some logical condition. These statements are extremely common and invaluable in programming

4.4.2 if Statement

This is a simple and intuitive construct

syntax 1:

```
if ( condition )
    true statement block;
else
    false statement block;
```

syntax 2:

```
if ( condition )
    true statement block;
```

If the condition in parentheses evaluates to true then the true statement block is executed, otherwise the false statement block is executed. In the second syntactical variation the true statement is executed only if the expression is true otherwise nothing is done.

EXAMPLE

```
int numerator, denominator;
Console.WriteLine("Enter two integer values for the numerator and denominator");
numerator = Convert.ToInt32(Console.ReadLine());
denominator = Convert.ToInt32(Console.ReadLine());
if (denominator != 0)
Console.WriteLine("{0}/{1} = {2}", numerator, denominator, numerator/denominator);
```

```
else
Console.WriteLine("Invalid operation can't divide by 0");
```

The statement body can include more than one statement but make sure they are group into a code block i.e. surrounded by curly braces.

EXAMPLE

```
int x, y, tmp;
Console.WriteLine("Please enter two integers");
x = Convert.ToInt32(Console.ReadLine());
y = Convert.ToInt32(Console.ReadLine());
if ( x > y)
{
tmp = x;
x = y;
y = tmp;
}
```

4.4.3 Nested if Statements

Nested if statements occur when one if statement is nested within another if statement.

EXAMPLE

```
if (x > 0)
if ( x > 10)
Console.WriteLine("x is greater than both 0 and 10");
else
Console.WriteLine("x is greater than 0 but less than or equal to 10");
else
Console.WriteLine("x is less than or equal to 0");
```

Nested if's are not complicated but can be a source of error. Notice that the code of the above example is indented in such a way as to clarify the programmers' intentions. Indentation is not necessary but makes a big difference to program readability and error spotting. Indentation is compulsory on this course. Another way to avoid errors in nested if's is to use the rule "an else is matched to the nearest unfinished if".

EXAMPLE

```
x = y;
if( y <= 12)
if(y == 0)
x = 12;
else
x = y - 12;
```

To which "if" does the else belong? Note that indentation can be misleading and anyway the compiler ignores it.

4.4.4 if - else - if Ladders

If a program requires a choice from one of many cases, successive if statements can be joined together to form a if - else - if ladder.

syntax:

```
if ( expression_1 )
statement_1;
else if ( expression_2 )
statement_2;
else if ( expression 3 )
statement_3;
else if (condition_n)
statement_n;
else
statement_default;
```

The statement_default is executed only if all preceding expressions evaluate to false. Caution is advisable when coding if-else-if ladders as they tend to be error prone due to mismatched if-else clauses.

EXAMPLE

```
int age;
Console.WriteLine("Please enter your age\n\n");
age = Convert.ToInt32(Console.ReadLine());
if ( age > 0 && age <= 10 )
Console.WriteLine("You are a child?\n");
else if ( age > 10 && age <= 20 )
Console.WriteLine("You are a teenager?\n");
else if ( age > 20 && age <= 65 )
Console.WriteLine("You are an adult?\n");
else if ( age > 65 )
Console.WriteLine("You are old!\n");
```

4.4.5 Conditional Operator ?:

There is a special shorthand syntax that gives the same result as

```
if (expression )
true_statement;
else
false_statement;
```

syntax: `expression ?  true_statement :  false_statement;`

The ?; requires three arguments and is thus ternary. The main advantage of this operator is that it is succinct.

EXAMPLE

```
max = x >= y ?  x :  y;
```

which is the equivalent of

```
if ( x >= y)
max = x;
else
max = y;
```

4.4.6 switch Statement

This statement is similar to the if-else-if ladder but is clearer, easier to code and less error prone.

syntax:

```
switch( expression )
{
case constant1:
statement1;
break;
case constant2:
statement2;
break;
default:
default_statement;
}
```

The value of expression is tested for equality against each of the constants in the order written until a match is found. If no match is found then the optional default option is executed. The statements associated with a case statement are executed until a break statement is encountered, at which point execution jumps to the statement immediately after the switch statement. Note that a break statement is not compulsory, however, in the absence of a break statement execution continues to the next statement in the switch statement. This may be logical in certain cases, but this is not usually the desired behaviour.

A default section is optional but it is usually a good idea to include one, so that all possibilities are covered. If absent and no match is found nothing is done.

The following limitations apply to the switch statement:

1. You an only test for equality against integer or string constants in case statements.
2. No two case statements can have the same value.
3. Character constants are automatically converted to integers e.g. switch('a') is converted to switch(97) and case 'a': is converted to case 97:.

EXAMPLE

```
double num1, num2, result;
char op;
Console.WriteLine("Enter number operator number\n");
```

```
num1 = Convert.ToInt32(Console.ReadLine());
op = Convert.ToChar(Console.ReadLine());
num2 = Convert.ToInt32(Console.ReadLine());
switch(op)
{
case '+':
result = num1 + num2;
break;
case '-':
result = num1 - num2;
break;
case '*':
result = num1 * num2;
break;
case '/':
if(num2 != 0)
{
result = num1 / num2;
break;
} //else fall through to error statement
default:
Console.WriteLine("ERROR- invalid operation or divide by 0.0\n");
}
Console.WriteLine("{0} {1},{2} = {3}\n", num1, op, num2, result);
```

4.5 Iterative Statements

4.5.1 for Statement

Simple for Loops

A statement or block of statements may be repeated a known number of times using the for statement. The programmer must know in advance how many times to iterate or loop through the statements, for this reason the for statement is referred to as a counted loop.

syntax:

```
for([initialisation];[condition];[action])
[statement_block];
```

Square braces indicate optional sections. Initialisation, condition and action can be any valid C# expression, however, there are common expressions which are recommended for each part.

initialisation: executed once only when the for loop is first entered, usually used to initialise a counter variable.

condition: when this condition is false the loop terminates.

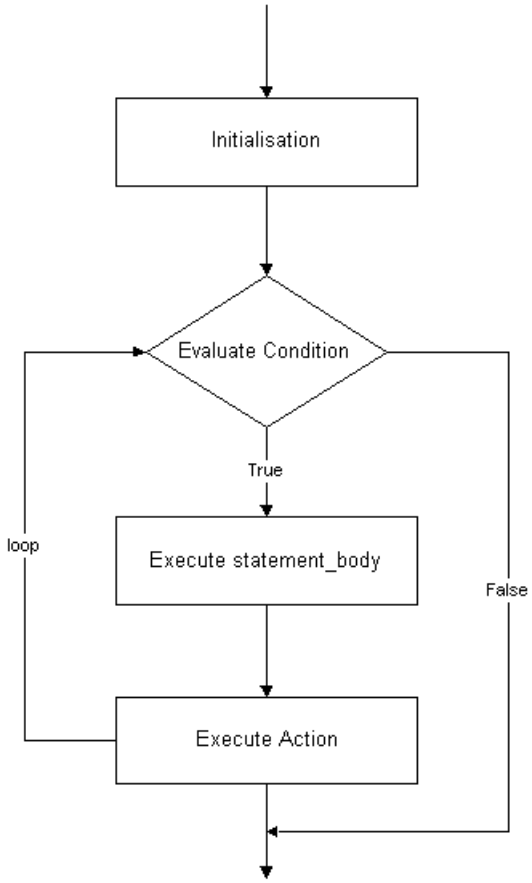

Figure 4.5.1 Flow chart for execution of for statement

action: executed immediately after every run through statement_block and typically increments the counter variable controlling the loop.

The following steps illustrate exactly how a for statement is executed:

1. Evaluate initialisation.
2. Evaluate condition, if true continue, if false exit for statement, execution jumps to first statement after the for statement.
3. Execute statement_block;
4. Evaluate action.
5. Repeat steps 2 to 4 until test evaluates to false.

Figure 4.5.1 illustrates the execution sequence graphically.

EXAMPLE

```
int x;
for (x = 1; x <= 100; x++)
Console.WriteLine("{0}", x);
```

The above example prints out the numbers from 1 to 100.

EXAMPLE

```
int x, sum = 0;
for (x = 1; x <= 100; x++)
{
Console.WriteLine("{0}", x);
sum += x;
}
Console.WriteLine("Sum is {0}", sum);
```

Prints the numbers from 1 to 100 and their sum.

Advanced for Loops

Multiple expression statements can be tied together into one statement using the comma operator and we have seen this already in the way we declare multiple variables.

EXAMPLE

```
int i = 0, j = 0, k = 1; // the comma operator is in use here
i += j;
j += k;
k *= i;
```

we could simplify this to

```
int i = 0, j = 0, k = 1; // the comma operator is in use here
i += j, j += k, k *= i; // comma operator joins three statements in one
```

Applying this syntax to a for statement allows multiple initialisations in a for loop.

EXAMPLE

```
for( x = 0, sum = 0; x <= 100; x++)
{
Console.WriteLine("{0}", x);
sum += x;
}
```

We can omit any or all of the four section of the for loop statement, however the semi-colons must always be present.

EXAMPLE

```
for ( x = 0; x < 10; )
Console.WriteLine("{0}", x++);
```

EXAMPLE

```
x = 0;
```

```
for ( ; x < 10; x++)
Console.WriteLine("{0}", x);
```

Infinite loops are possible as follows

```
for( ; ; )
statement_body;
```

infinite loops can also be created erroneously by including a faulty test condition. Sometimes we may wish to create a for loop with no statement_body in order to create a time delay, for example

```
for(t = 0; t < big_number; t++);
```

or we could printout the numbers from 1 to 100 by

```
for(x = 1; x <= 100; printf("%d\n", x++));
```

Don't forget that the initialisation, condition and action parts of a for loop can contain any valid C expression.

```
for(x = 12 * 4; x < 34 / 2 *47; x += 10) printf("%d",x);
```

It is also possible to nest for loops.

EXAMPLE

```
for(x = 0; x < 65535; x++)
for(y = 0; y < 65535; y++);
```

The following program produces a table of values

```
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
int j, k;
for (j = 1; j <= 5; j++)
{
for( k = j; k < j + 5; k++) {
Console.WriteLine("{0}", k);
}
Console.WriteLine();
}
```

4.5.2 while Statement

In contrast to the for statement, the while statement allows us to loop through a statement block when we don't know in advance how many iterations are required.

syntax:

```
while( condition )
statement_body;
```

A flow chart of the execution sequence for the while statement is shown in figure 4.5.2.

EXAMPLE

```
int sum = 0, i = 100;
```

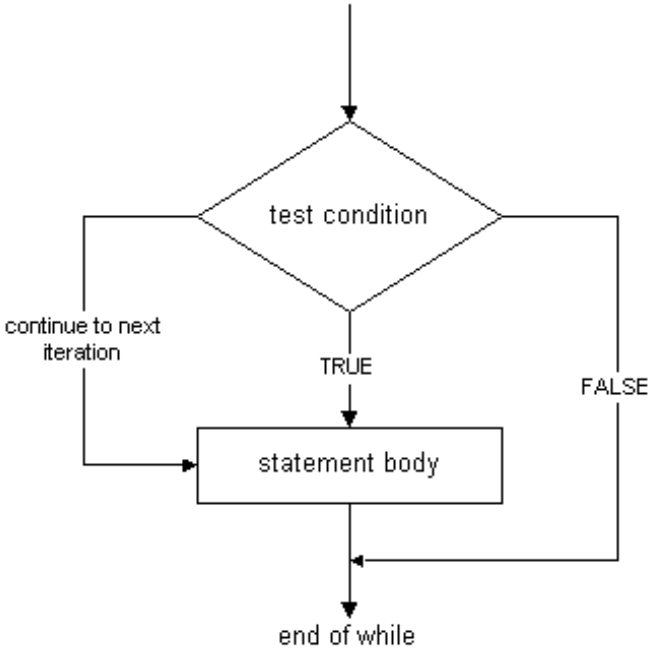

Figure 4.5.2 Flow chart for while statement

```
while(i != 0) // this condition evaluates to true once i is not equal to
0
sum += i- -; // note postfix decrement operator, why?
Console.WriteLine("sum is {0}", sum);
```

This program calculates the sum of 1 to 100.

Words of warning, although succinctness is tidy and usually desirable, do not sacrifice readability for it. If other people (not just C gurus) cannot read your code or it is too much effort, they may become nervous. What happens when you leave the company? Who is going to maintain your code?

Like for loops while loops may also be nested.

EXAMPLE

A program to guess a letter

```
char ch, letter = 'c', finish = 'y';
while ( finish == 'y' || finish == 'Y')
{
Console.WriteLine("Guess my letter - only 1 of 26!");
while((ch = Convert.ToChar(Console.ReadLine())) != letter)
{
Console.WriteLine("{0} is wrong - try again\n", ch);
}
Console.WriteLine("OK you got it \n Lets start again.\n");
```

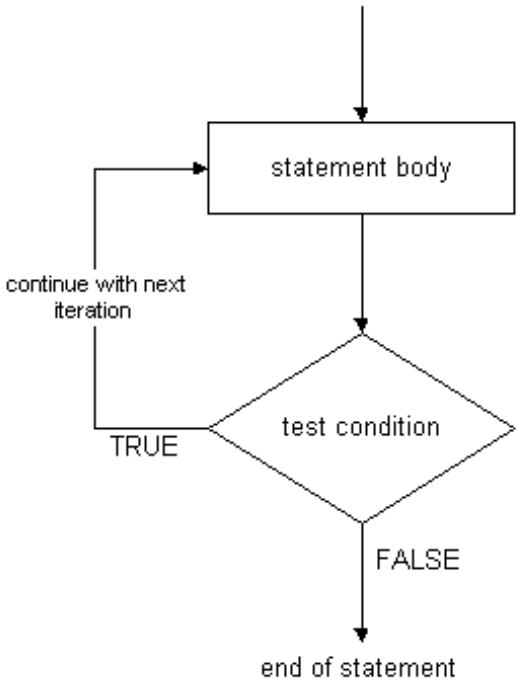

Figure 4.5.3 Flowchart for do while statement

```
letter += (char)3;
Console.WriteLine("Do you wish to continue (Y/N)?");
finish = Convert.ToChar(Console.ReadLine());
}
```

4.5.3 do while Statement

In both the for and while statements the test condition is evaluated before the statement_body is executed. This means that the statement_body might never be executed. In the do while statement the statement_body is always executed at least once because the test condition is at the end of the body of the loop.

syntax:

```
do
{
statement_body;
} while ( condition );
```

A flowchart of the statement is present in figure 4.5.3.

EXAMPLE

Keep reading in integers until a value between 1 and 10 is entered.

```
int i;
do
{
```

```
i = Convert.Toint32(Console.ReadLine());
} while( i >= 1 && i <= 10);
```

4.5.4 break Statement

When a break statement is encountered in a for, while, do while or switch statement the statement is immediately terminated and execution resumes at the next statement following the loop/switch statement.

EXAMPLE

```
for (x = 1; x <= 10 ; x++)
{
if ( x > 4)
break;
Console.Write("{0} ", x);
}
Console.WriteLine("Next executed"); //Output is 1 2 3 4 Next executed
```

4.5.5 continue Statement

The continue statement terminates the current iteration of a for, while or do while statement and resumes execution back at the beginning of the statement_body of the loop with the next iteration.

EXAMPLE

```
for (x = 1; x <= 5; x++)
{
if (x == 3)
continue;
Console.Write("{0} ", x);
}
Console.WriteLine("Finished loop\n"); // output is 1 2 4 5 Finished loop.
```

4.6 Generating Random Numbers in C#

Although this is not a C# statement it is a handy feature to have to hand when writing programs. Random numbers are generated in C# courtesy of the unimaginatively named Random object. Unlike Console and Convert a Random object is not automatically created when your program runs. This means we have to create a Random object before we can use it. The following code illustrates an example:

```
int r, i = 0 ;
Random ran = new Random();//creates a Random object
while(i++ < 100)
{
r = Convert.ToInt32(ran.Next(0,10));
Console.WriteLine("{0}", r);
};
Console.ReadLine();
```

Please check out the various features of the Random class yourself in the help file.

Exercise 1 *Write four statements which each add 1 to an integer variable x.*

Exercise 2 *Given the following integer variables and their initial values in brackets x(1), y(2), z(2), count(11), total(7), q(5) and divisor(2), write C statements to accomplish the following:*

1. *Assign the sum of x and y to z and increment the value of x by 1 after the calculation.*
2. *Test to see if the value of count is greater than 10. If it is then output "Count is greater than 10".*
3. *Decrement the variable x by 1 and then substract it from total.*
4. *Calculate the remainder after q is divided by divisor and assign the result to q. Can you do this in two different ways?*
5. *Print out the values of the variables to the screen.*

Exercise 3 *Write a program that does the following:*

1. *Declare variables x, y and sum to be of type int.*
2. *Initialise x to be 3, y to be 2 and sum to be 0.*
3. *Add x to sum and assign the result into sum.*
4. *Add y to sum and assign the result into sum.*
5. *Print out "The sum is : " followed by the value of sum.*
6. *Can you replace steps 3 and 4 with a single C# statement?*

Exercise 4 *Write a program to calculate the sum of the first n integers where n is specified by the user e.g. if the user inputs a value of 10 for n your program should calculate* $1 + 2 + \ldots + 9 + 10$. *Write code to do this (a) using a while loop and (b) using a for loop. Your program should print out "The sum of the first n integers is result", where n and result are the appropriate numbers.*

Exercise 5 *Consider two integer variables x and y with initial values of 5. What are their values after* `x *= y++;`*? What are they after* `x /= ++y`*?*

Exercise 6 *What is wrong with the following piece of code?*

```
while(z >= 0)
sum += z;
```

Exercise 7 *Write a program which invites the user to input 10 numbers and then finally prints out the largest one entered.*

Exercise 8 *Write a program which prints out the ASCII values of all characters input at the keyboard terminating only when the character 'q' or 'Q' is entered.*

Exercise 9 *Write a program to keep count of the occurrence of a user specified character in a stream of characters of known length (e.g. 50 characters) input from the keyboard. Compare this to the total number of characters input when ignoring all but alphabetic characters. Note: The ASCII values of 'A' - 'Z' are 65 - 90 and 'a' - 'z' are 97 - 122.*

Exercise 10 *Write a program to find the roots of a user specified quadratic equation. Recall the roots of ax2 + bx + c = 0 are* $\frac{-b \pm \sqrt{b^2-4ac}}{2ac}$*. The user should be informed if the specified quadratic is valid or not and should be informed how many roots it has, if it has equal or approximately equal roots* $(b^2 = 4ac)$*, if the roots are real* $(b^2 - 4ac > 0)$ *or if the roots are imaginary* $(b^2 - 4ac < 0)$*. In the case of imaginary roots the value should be presented in the form (x + i y). Note that C# has a standard function Math.Sqrt( ), which returns the square root of its operand, and whose prototype can be found in the help system. NB: The floating point number system as represented by the computer is "gappy". Not all real numbers can be represented as there is only a limited amount of memory given towards their storage. Type float for example can only represent seven significant digits so that for example 0.1234567 and 0.1234568 are deemed consecutive floating point numbers. However in reality there are an infinite number of real numbers between these two numbers. Thus when testing for equality, e.g. testing if* $b^2 = 4ac$*, one should test for approximate equality i.e. we should test if* $|b^2 - 4ac| < 0.00001$ *where we are basically saying that accuracy to five decimal places will suffice.*

Exercise 11 *Write a program that allows the user to read a user specified number of double precision floating point numbers from the keyboard. Your program should calculate the sum and the average of the numbers input.*

Exercise 12 *Write a program to print out all the Fibonacci numbers using short integer variables until the numbers become too large to be stored in a short integer variable i.e. until overflow occurs, (a) using a for loop construction and (b) using a while loop construction. Which construction is most suitable in your opinion? Note: Fibonacci numbers are (1, 1,2,3,5,8,13, etc.).*

Exercise 13 *Write a program which simulates the action of a simple calculator. The program should take as input two integer numbers then a character which is one of +, -, *, /, %. The numbers should be then processed according to the operator input and the result printed out. Your program should correctly intercept any possible erroneous situations such as invalid operations, integer overflow, and division by zero.*

Exercise 14 *Implement a version of the rock, scissors, paper game where by the user plays the computer. The user inputs their choice and the computer randomly chooses a option and the options and winner are output to the screen.*

Exercise 15 *Write a program to find the sum of the following series :* $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} \ldots$ *terminating when a term less than 0.0001 is encountered.*

Chapter 5
CREATING OBJECTS WITH CLASS

5.1 Introduction

In an earlier chapter, OOP was discussed without reference to a programming language. In this chapter, the main features for supporting object orientation in C#, the **class**, is investigated. The class construct is used to support encapsulation, data abstraction, interfaces and implementation hiding. Objects consist of properties and methods and we have already had experience of using objects created by others such as Console, Convert and the Random object. For example we used the WriteLine method of Console to send output to the screen and the Next method of Random to generate a random number. Notice that in each case we really didn't need to know the internal implementation of the object to successfully use it. By creating classes we can define our own objects by giving them properties and methods.

Up until now, we have been using intrinsic data types to manipulate numeric data. With classes we can create new data types to suit our own particular problem. A class is like a template or definition which is used to create an object. An object is then an instance of the class. A class specifies the data and methods relevant to the object. The problem to be solved guides us in choosing appropriate data and methods. In general terms a class consists of a hidden part and an exposed part, which users of the object can "see" or use. In the hidden part there is private data and private methods, and the user of an object cannot see or use these data or methods, this is equivalent to hiding the implementation details. Public methods have their internals hidden from the user, however, the user can invoke these methods. Public properties expose the private data and provide a controlled interface whereby users can modify the private data. There are also special methods called constructors that are called to help creating an object (i.e. make sure the object is initially in a good state). Sometimes there may be a need to tidy up data when an object is being terminated and these methods are called destructors.

5.2 Complex numbers with class

5.2.1 Defining the class

C# classes enable programmers to model real-world objects as objects in a program which have properties and methods. A class creates a new data type in C#. The

keyword `class` is used in C# to declare a class. Its syntax is as follows:
syntax:
```
[access-modifier] class identifier [:base-class]
{class-body}
```
Items in square brackets are optional. Access modifiers are discussed later and typically the public keyword is used with classes. Identifier gives the class a name. The optional base class is for advanced classes dealt with later on. Finally the class body is where the data and methods are placed.

5.2.2 Access Modifiers

Access modifiers are placed before data and methods in the class body. The two most commonly used are public and private. Public means that users of the object either have direct access to data or can invoke a method. Private means that object users cannot directly access data or invoke the method, essentially hiding them.

5.2.3 Defining Data

Data is declared inside the class body in almost exactly the same way as we have seen already. That is:
syntax:
```
[access-modifier] data-type identifier [= intial-value];
```
For example
```
private int x = 10;
public double radius = 10.5;
```
Its as easy as that! The variable x is hidden, whereas the variable radius is usable.

5.2.4 Defining Methods

Methods define the behaviour of an object and are where we encapsulate pieces of useful functionality that can operate on class data (both public and private). In addition, data may be passed into methods as optional arguments and data may also be returned by the method (i.e. the result of the manipulation). Here is the syntax of a method:
syntax:
```
[access-modifier] return-data-type identifier ([data-type param1], [data-type
param2],...)
{method-body};
```
For example:
```
public int add(int a, int b)
{
int result;
result = a + b;
return result;
}
```
In this example we are defining a method visible to object users, which must be passed in two arguments (each must be of type int). Inside the method an int is declared

which temporarily stores the result of the addition. Finally the result is returned as the result. Notice that in the method definition the return type was specified to be int. You must return a data type compatible with the declared return type. If no data is to be returned from a method call then the keyword `void` is used. Most people use argument and parameter interchangeably, however strictly speaking a parameter is the definition and an argument is what is actually passed in to a method. Arguments must be individually typed, that is, `public int add(int a, b)` is invalid. Before going any further lets create a simple class.

5.2.5 Putting it All Together

The following example is a first attempt at implementing a complex class. It illustrates the fundamental syntax of the class construct, with private data, a private method and three public methods.

```
public class Complex {
//data section
private double real = 0;
private double imag = 0;
//method section
private void zero()
{
real = 0.0;
imag = 0.0;
}
public void set_real(double r)
{
real = r;
}
public void set_imag(double i)
{
imag = i;
}
public double magnitude()
{
return Math.Sqrt(real * real + imag * imag);
}
}
```

It is important when developing classes and objects that they are tested. This should involve some code that outs the object through its paces. It is usually good practice to test often, rather than waiting for a huge volume of code, which is harder to debug. The complete code including testing the in main method is:

```
namespace MyComplex
{
/// <summary>
/// Summary description for Class1.
/// </summary>
class TestClass
{
/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main(string[] args)
{
double r;
complex c = new complex();
c.set_imag(2);
c.set_real(-2);
r = c.magnitude();
Console.WriteLine("the magnitude is :  {0}", r);
Console.ReadLine();
}
}
class complex
{
//data section
private double real = 0;
private double imag = 0;
//method section
private void zero()
{
real = 0.0;
imag = 0.0;
}
public void set_real(double r)
{
real = r;
}
public void set_imag(double i)
{
imag = i;
}
public double magnitude()
```

```
{
return Math.Sqrt(real * real + imag * imag);
}
}
}
```

Notice that we create an object in much the same way as we did a Random object. There is a subtle point to be made here. A complex object is created by:
`complex c = new complex();`
However we could also break it down to:
```
complex c;
c = new complex();
```
This is because the `complex c` part is creating a reference object which will refer to a complex object (it itself is not the complex object). The `new complex()` part creates a complex object in memory and returns a reference to it. In other words if wrote:
`new complex();`
a new unnamed complex object would be created in memory but I would have no way of modifying it later because I have no reference to it. Therefore the two operations go hand in hand.

5.2.6 Relating the C# class to OOP

Before continuing further with the Complex class lets review some of the concepts introduced in the last chapter and see how C# classes support these object oriented notions. A class is a template for creating objects and in C# the class construct is exactly that. Objects are entities with properties and methods; in C# a class contains data members corresponding to properties and methods corresponding to OO methods. Ecapsulation is the process of hiding the implementation details of an object; in C# the private access modifier is used to create hidden data and methods (i.e. these are not directly available to the user of an object). Note also that the implementation details of public methods cannot be interfered with by object users. An object should have an interface which allows the object to be manipulated, in C# this interface is made up of the public parts of the class definition. Clearly then the C# class construct allows the creation of abstract data types (ADT's) with hidden implementation details and a consistent interface. Note however that C# does not twist our arms into creating ADT's. I could have made everything public in the above example however such a class is neither encapsulated nor defines an ADT, and such an approach is considered poor programming practice.

5.3 Constructors, Initialisation and Copy Constructors

When an object is instantiated it is a good idea to be able to get the object into a satisfactory initial state (for example a default good state) or alternatively allow the user creating the object to specify suitable values to initialise the object data. If the

methods associated with the class are well-designed then starting with a good state there should be no possibility of getting into a bad state. In the absence of either initialisation or constructors, intrinsic data types get a default value (this is quite different to other languages), numeric data is set to zero, bools to false, chars to the null character '\0' and references to null.

The simplest method of getting an object into a good state is to use initialisers (see the complex example above). Basically a constant value is assigned to a data member. This is the very first thing that happens when an object is created. If a constructor also exists then initialisation occurs first followed by the call to the constructor. A constructor is a public method with the same name as the class with no return type, which is called once upon object creation. Constructors may be passed one or more arguments. Typically constructors are used to initialise the object data, checking that values are valid. Any construct with no arguments is called the default constructor and if no constructors are written then the compiler creates a default constructor (which does nothing) for you. There can be more than one constructor associated with an object if this is useful. The compiler must be able to distinguish between each constructor so that it knows the correct one to invoke. This essentially means that each constructor must have unique signature, that is, the types and number of arguments are unique. This is called overloading, something which can be done with all methods (see later).

As an example we could add three constructors to the complex class above:

```
public complex()  //no arguments default constructor
{
zero(); //make the values zero
}

public complex(double preal, double pimag)  //two double arguments
{
real = preal;
imag = pimag;
}

public complex(double pval)  //one double argument
{
real = image = pval;
}
```

Lets now look at code to invoke the various constructors when creating complex objects.

```
complex c1 = new complex()  //uses the default constructor
```

```
complex c2 = new complex(3.2, -5.6);  //invokes second constructor
complex c3 = new complex(5.4);  //etc.
```

Sometimes you may want to create a new object using the data of an existing object. A copy constructor is required if you wish to support this activity. Typically this is only used for objects with complex data. Code exemplifying this procedure is shown below:

```
public complex(complex p)
{
real = p.real;
imag = p.imag;
}

complex c = new complex(1.2, 3.4); //create a complex object
complex b = new complex(c); //create another one based on the data of c
```

5.4 Variable Scope and this

5.4.1 Variable scope

It is important to know where an when you can access the various elements of an object. This can be confusing initially. When we are writing objects we need to be aware of the end user of the object. In addition we also need to be aware of what we can and cannot do as we create the template for object creation. You should already be aware of what an object user can and cannot do with an object as we (the object developers) determine this using access modifiers.

However, there can be some confusion about what we can do as we developers. First of all, variables declared inside methods and method arguments only exist during a call to the method and cannot be accessed elsewhere. Class data members can be accessed inside any method of the class (public or private). Indeed in the case of the copy constructor in the last section when an object is passed in, we can even access the private data of that object. this is because we are creating the object. A normal object user would not be able to access the object internals.

5.4.2 The this Keyword

The keyword this refers to the current instance of an object. It is used in a variety of settings. Firstly it can be used in the case of ambiguous and unrecommended naming. For example:

```
public complex(double real)
{
this.real = real;
}
```

here this distinguishes between the passed in real and the real of the current object instance. This vagueness is preferably avoided by using a sensible naming strategy. The second typical use is for passing the current object instance into the method of another class. We may come across some real examples later on.

5.5 Static and Instance Members

The properties and methods of class can be either instance or static. By default properties and methods are instance (as they have been up to now) and this means that they are to be used only with a specific instance of a class. Sometimes it makes sense to have data and methods that operate at a class level, that is, independently of any particular instance. Such methods are prefixed with the keyword static in their declaration and can only be accessed using the class name rather than through a particular instance. Static data and methods can be accessed/invoked inside in instance methods however static methods can only access static data and obviously don't have access to the this object reference.

As an example consider the main() method where have been doing most our coding so far. Notice that this is a static method. Suppose we decide that we wish to have an instance method to run our code as follows:

```
public void Run()
{
//lots of lovely code here
}
```

However because Run is an instance method it cannot be called directly in main(). We have to create an instance of the object first (in main()) and then invoke the Run() instance method for the instance.

```
using System;
namespace testspace
{
class test
{
static void main()
{
test t = new test();
t.Run();
}

public void Run()
{
```

```
//lots of lovely code here
}

}
}
```

Again we will see some real examples later.

5.6 Destroying Objects

When objects are created we use constructors to get them into a valid state, correspondingly when an object is destroyed we may need to tidy things up and in many other languages (C++, for example) destructor methods are used. Although destructors are available in C# there are used less frequently due to automatic tidying up that occurs. Although this releases the programmer from much tedious and error prone coding, the tidying up process in non-deterministic. In C# when an object is no longer in use the garbage collector will at some stage (not guaranteed to be quick) tidy up any resources associated with the object and call the destructor. However, in some objects we may be referencing critical resources related to graphics, files or databases. Therefore in C# an alternative approach is recommended. We are not yet in a position to discuss this approach, however we will tackle it later on.

Exercise 1 *Create a rectangle class with two data members length and width which default to 1.0. Provide both set and get functions to change the member data values and the set functions should ensure that values entered are between 0 and 20. Provide member functions to calculate the perimeter and area of the rectangle and a function square which returns true if the rectangle is a square otherwise it returns false.*

Exercise 2 *Write a class to simulate a bank account allowing for initialisation, deposits, withdrawals, setting the interest rate, adding interest and giving the current balance. Provide some code to demonstrate the object in action.*

Exercise 3 *Modify the complex class developed by adding methods which allow complex addition, subtraction, multiplication and division. You are required to do this in two ways (i) using instance methods and (ii) using static methods. Also provide a method which outputs the current state of a complex object. Could this method be static? Provide some code to illustrate complex objects in action.*

Exercise 4 *Write a program that can convert temperatures from the Fahrenheit scale to Celsius and back. The relationship is* $C = (5/9)(F - 32)$*. Your program should read a temperature and which scale is used and convert it to the other, printing out the results. Write one or more objects to carry out the actual conversion.*

Exercise 5 *Write a program that reads in the radius of a circle and prints the circle's diameter, circumference and area. Write an object for appropriate logical tasks in your program.*

Exercise 6 *Write a program which calculates two random integer numbers between 1 and 12 inclusive and then outputs: "What is 4 times 9?". Check the answer and if it is correct print out "Very Good", if it is wrong print "Close but no banana, try again". Keep them trying until they get it right. The program should continue asking questions until the user input indicates they wish to terminate. Think about the object(s) you might write to implement this program.*

Exercise 7 *Write and test an object to convert a byte to binary notation.*

Exercise 8 *Write and test an object that packs two sbyte variables into one short variable. It should also be able to unpack them again to check the result.*

Exercise 9 *An integer is said to be prime if it is divisible only by one and itself. Write an object which determines whether an integer is prime or not. To test your object, write a program that prints out all prime numbers between 1 and 10,000.*

Chapter 6
MORE ON METHODS

6.1 Introduction

Methods allow us to segregate the functionality of an object in logical sections. They encapsulate the functionality of an object and are the basic units of work in an application. In this chapter we look in more detail at methods and arguments and how they can be used.

6.2 Method Overloading

We already encountered overloading in the case of constructors. We can apply the same logic to all methods, that is we can create many versions of the same method, provided a unique signature is present. Overloading was developed to reduce the number of different method names to be created by the programmer and it also makes life easier for the end object user. Although there is no restriction on what functionality goes into methods of the same name, it makes good logical and functional sense that the behaviour should be in some way related. For example, a method named add shouldn't implement subtraction.

6.3 C# Properties

Perhaps confusingly C# has a construct named the property. It is akin to the property notion in OOP and it is a great labour saving device. Whenever we have some private data, we have to provide a mechanism to access and/or modify the data. This usually involves writing a get and a set method for each private data member, which can get tedious. C# introduces properties as a way of rolling all this up into a simpler syntax. There is still a requirement for a private data member, but for each private data member we provide a public property which simulates direct access, however, it is controlled behind the scenes (i.e. the implementation details are hidden form the user). As an example suppose we have a private data member which is supposed to have positive integer values. Our class definition will contain (as per normal):

```
private int positive;
```

next we implement the property section:

```
public int Positive  //notice different name
{
get
{
return positive;
}
set
{
if(value > 0)
postive = value;
}
}
```

Suppose now we have an instance of the class called obj. We can directly access the pubic property (Positive) to indirectly modify the private data (positive):

```
obj.Positive = 10;
obj.Positive = -100; //no effect
Console.WriteLine("positive has value {0}", obj.Positive);
```

For the first call the set section is executed to change the value of the private data. Similarly with the second line, however, this time we are trying to assign a negative number so the value is not changed. Finally the get method is used to access the private data. This is a convenient and time saving syntax.

6.4 Passing Arguments and Objects

In C# there are two basic data types: value types and reference types. All intrinsic data types (e.g. byte, int , float, double) are value types and when they are passed into a method a copy is made (see next section). However after an object is created it is referenced by a reference type. For example:

```
complex c = new complex();
```

here c is a reference type, it is not a complex object, it is a reference to a complex object. The data of the object is stored elsewhere, c only holds a reference (i.e. directions) to the object. This is an important and subtle point. Value and reference types behave differently when being passed to a method and this is investigated in this section.

6.4.1 Passing by Value

Arguments are used to pass in additional data required by a method. What actually happens when we call a method and pass in data? Quite a bit actually. Consider the following method:

```
public int add( int a, int b)
{
int res;
res = a+b;
return res;
}
```

suppose we call it using some instance:

```
int p = 2, q = 3, res = 0;
res = obj.add(p, q);
```

The sequence of events is:

1. The integer values p, q and res are created in memory and initialised.
2. The add method is invoked
3. Local variables a and b are created in memory and the values from p and q are copied in respectively.
4. Local variable res is created in memory (this is different to the external res variable) and calculation is performed.
5. The value of the result of the calculation is copied into the external variable res.

Local variables res, a and b are removed from memory.

This means that whatever changes are made to the method parameters a and b, are separate from p and q. One does not affect the other. This is termed pass by value and is the default way that data is passed into methods.

6.4.2 Passing by Reference

Sometimes passing by value is not enough, we may want to modify the data in parameters (as a way of passing back more than one return value for example). Suppose we wished to write a swap method that swaps the values in two variables. At present we might try:

```
public void swap(int a, int b)
```

Type	int	int		ref int	ref int	
Name	j	k		a	b	
Value	2	3		1000	1004	
Address	1000	1004		2000	2056	

Figure 6.4.1 Memory situation during invokation of swap method

```
{
int temp;
temp = a;
a = b;
b = temp;
}
```

this approach wont work because we are dealing with copies and the code has no effect on any variables outside of the method. Enter the reference. Reference parameters don't pass the values of the variables used in the method invocation - they use the variables themselves. Rather than creating a new storage location for each variable in the method declaration, the same storage location is used, so the value of the variable in the method and the value of the reference parameter will always be the same. Reference parameters need the `ref` modifier as part of both the declaration and the invocation - that means it's always clear when you're passing something by reference. Our swap example would then become:

```
public void swap(ref int a, ref int b)
{
int temp;
temp = a;
a = b;
b = temp;
}

int j = 2, k = 3;
swap(ref j,ref k);
```

Whats happens when the swap function is invoked? It works quite differently to the previous example:

1. When swap is called two reference data types are created, these are not ints but types which can hold a reference to an int (in reality this amounts to holding the memory address of the actual int).

2. The reference types don't take the actual values of j and k but their memory addresses. This means that inside the method a is a reference to j and b is a reference to k.

3. When a and b are being modified inside the method we are actually modifying the data in j and k outside the method.

It is important to know whether we are dealing with reference of value types when we are passing in data to a method. It is advisable to play around with some examples, to be sure you understand the implications. Because objects are always referred to by reference then clearly whenever we pass objects into a method (and we don't have to specify the ref keyword) we can modify the object inside the method.

6.4.3 The Keyword out

As a minor detail there is the keyword out. Sometimes we wish only to put data into reference parameters and we are not interested in the values they hold before being passed in. Because definite assignment is supported in C# (i.e. all variables must be initialised before use) we can get compiler errors in the following situation:

```
int p;

public void modify(ref int a)
{
a = 10;
}

modify(ref p);
```

This is because p hasn't been initialised prior to its use, even thought the method does nothing with the value of p. There are two solutions (i) initialise p or (ii) use the out keyword. out works in the same way as ref except that you are not allowed to use an out parameters value. The correct example then becomes:

```
int p;

public void modify(out int a)
{
int j = 5;
//j = a; this line would cause an error
a = 10;
}

modify(out p);
```

Exercise 1 *Write a method that takes two integer numbers and returns both their sum and difference.*

Exercise 2 *Write a method which takes an integer number of seconds and converts it to hours, minutes and seconds.*

Exercise 3 *Write a time object which stores time as an integer number of seconds. Add methods/properties to set the hour,minute and seconds always storing the total time as seconds. Write a method which takes another time object as a parameter and calculates the time difference in hours minutes and seconds.*

Exercise 4 *Write a method which increments an integer argument by one.*

Programming Assignment No 1: Non-linear Equations.

Your basic task in this assignment is to write a C program which will allow the user to solve the following non-linear equation

$$f(x) = \sin\left(\frac{2x}{5}\right) - x + 1 = 0$$

to an accuracy of 10^{-5} using both the Bisection method and Newton's method.

In order to be able to solve equations using these methods you will need to provide your program with suitable initial estimates to the actual root, an interval over which the sign of $f(x)$ changes in the case of the Bisection method and an initial estimate x_0 where $f(x_0)$ is not very small in the case of Newton's method.

To help the user to provide such estimates your program should first tabulate the function $f(x)$, repeatedly until the user is satisfied, over a user specified interval $[x_1, x_2]$ and using a user specified tabulation step.

This should allow the user to begin by tabulating the function over a large interval with a coarse tabulation step and to fine down the interval and step until he / she can determine the behaviour of $f(x)$ in or about the actual root.

Once the user is satisfied with the tabulation your program should read in the appropriate initial estimates from the user and test their validity and then solve the equation using both methods. Finally your program should compare the number of iterations required to compute the root using both methods and print out the value of the root.

You should break your program up into appropriate logical functions and try to intercept all erroneous situations as soon as possible and take appropriate action. *Note:* The exact solution to the above equation is 1.595869359.

Chapter 7
ARRAYS AND STRINGS

7.1 Introduction

Arrays and strings are introduced in this chapter. We will see how to declare arrays, access the elements of an array and how to pass them as parameters to functions. Additionally we investigate string objects and how we can manipulate them.

7.2 Simple Arrays

7.2.1 Introduction

An array is a named collection of values, all of which have the same data-type. When discussing variables earlier we saw that a variable consists of a memory location and a value. An array is conceptually similar except that with an array we can reference many values using a single identifier. We use an index to indicate which value we want to reference. The values of an array are stored in contiguous memory locations (i.e. side by side).

7.2.2 Declaration and Creation

The syntax for declaring an array is:

```
data-type[] array_name;
```

For example:

```
int[] myArray;
```

Arrays are reference types and we are creating a reference type with this declaration which can be used to refer to an array object. In C# all arrays are in fact instances of the System.Array class, such that arrays have a set of methods and properties going with them. This makes the programmers life easier. Next we need to create an array object as follows:

```
myArray = new int[10];
```

This assigns into myArray a reference to the array object created with the new keyword. Putting 10 in the square brackets indicates that enough space for 10 ints is required. These will be stored side by side contiguously somewhere in memory.

Unlike C and C++ arrays, C# arrays can be dynamically sized at run-time.

```
Console.WriteLine("Please enter an integer value");
int size = Convert.ToInt32(Console.ReadLine());
```

```
int[] myArray = new int[size];
```

In contrast to C and C++ we don't have to worry about deallocating the dynamically allocated memory as the garbage collector mechanism tidies up automatically.

When an array of intrinsic types is created the elements of the array are given default values. However, when an array of reference types is created they are initialised to null (indicating they are not referencing anything). Using a reference to null will cause an exception to be raised. So for example if we were to create an array of 10 complex objects we would have to do the following:

complex[] carr = new complex[10]; //creates 10 references to null
for(i = 0; i <10; i++)
complex[i] = new complex(1,2); //creates the objects to refer to

7.2.3 Indices and Size

When a new array is created the size of the array is in the square brackets. If we don't want to remember the size of each array we can use a property (called Length) of the underlying array object to find out the size of the array at any time. When we want to refer to an individual element of an array we use index notation to do so. For example to access the an element of the complex array above we could write carr[3] to access the fourth element. Indices use zero counting, so the first element is zero etc. The valid values for an index are 0 up to the array size minus one.

```
myArray = new int[10];
for(i = 0; i < myArray.Length; i++)
myArray[i] = 0;
```

If you index into an array outside of the valid bounds then an exception will occur.

7.2.4 The foreach Statement

Very often we need to go through an array from start to finish i.e. the zeroth to the (size-1)th element. C# has a statement which makes this routine (and negates the need for counter variables in loops) call foreach. The syntax is:
foreach (type identifier in expression) statement
An example of its use is:

```
int[] arr = new int[20]

foreach(int t in arr)
Console.WriteLine("{0}", t);
```

each time around the loop t takes on the value of the next element starting with

the first. This may be of use in certain situations where you don't need the index value. You don't have to check the array size or maintain a counter variable.

7.2.5 Initialising Arrays

Usually the data for arrays comes form file or a database, for completeness we discuss how to initialise arrays in code. First of we could construct a list of elements by one of the two equivalent syntaxes below:

```
int[] myArray = new int[5] {1,3,2,5,8};
int[] myArray = {1,3,2,5,8};
```

In the second syntax the array creation also occurs implicitly. In the case of a reference type we need to create the actual objects as well i.e.:

```
complex[] c = new complex[3] {new complex(1,2), new complex(3,2), new complex(5,1)};
complex[] c = {new complex(1,2), new complex(3,2), new complex(5,1)};
```

7.2.6 Passing arrays as Parameters and the params keyword

The main feature of passing arrays is that arrays are reference types and therefore we may inadvertently modify the data of an array inside a method (and if this is not what is expected it could lead to unexpected result, otherwise its fine). Also when an array is passed in to a method there is no restriction placed on the size of the array so you always have to figure out the size of the array before you start working with it. For example:

```
int[] data = {3,5,7,2,5};

public float average(int[] p)
{
float sum = 0;
foreach(int item in p)
sum += item;
return sum/p.Length;
}

average(data);
```

The params keyword is available in C# and allows you to create methods which take a variable number of parameters. It is simple to implement. For example:

```
public void Display(params int[] p)
{
//just treat p as an ordinary array in here
```

```
foreach(int item in p)
Console.WriteLine("{0}", Item);
}

Display(1,2,3); //p contains {1,2,3}
Display(5,1,2,3,5); //guess what p contains!!
```

7.3 Multidimensional Arrays

7.3.1 Rectangular Arrays

Perhaps the most commonly used multidimensional arrays are rectangular, that is they consist of a number of rows and columns (and onwards if you are using higher dimensional arrays). The syntax is similar to that above and they are reference types. For example:

```
int[,] marr = new int[10,5];

marr[0,4] = 10;
```

This creates an array of ints with 10 rows and 5 columns. You are free to size these arrays dynamically as well. Valid indices range from 0 to rows-1 and from 0 to columns-1. They can be passed to methods in much the same way as simple arrays, except that there is a bit more to figuring out the size. The Length property will give the total number of elements (so in the case of marr this will be 10 X 5 = 50). Using marr as defined above:

```
void Display(int[,] p)
{
int rank = p.Rank; //gives the dimension of the array i.e.  2 in this case
int rows = p.GetLength(0);
int cols = p.GetLength(1); //up to Rank-1
int i, j;
for(i = 0; i < rows; i++)
{
for(j = 0; j < cols; j++)
Console.Write("{0} ",p[i,j]);
Console.WriteLine(); // puts a return after each row
}
}

Display(marr);
```

7.3.2 Jagged Arrays

A jagged array is really an array of arrays and in two dimensions it consists of rows of varying with hence the name. Declaration, indexing and initialisation of jagged arrays differs from that of rectangular arrays, although the syntax is similar enough to cause confusion. As with rectangular arrays jagged arrays are reference types. A reference to a rectangular array is as follows:

```
type [][] identifier;
```

e.g.

```
int [][] myJarray;
```

Creating the jagged arrays is a two step process (i) define the number of rows and (ii) identify the length of each row e.g.:

```
int [][] myJarray = new int[5][]; //5 rows
myJarray[0] = new int[2]; //create a row
myJarray[1] = new int[20];
myJarray[2] = new int[7];
myJarray[3] = new int[4];
myJarray[4] = new int[10];
```

When accessing a jagged array the syntax is:

```
identifier[row][column];
```

compare this with a rectangular array. As with all other arrays jagged areas can be dynamically sized. Clearly care needs to be taken when accessing a jagged array as an exception will be raised if you exceed the bounds of the array.

Jagged arrays can be passed into methods as before, however we get at array information differently:

```
//intialise a jagged array
Random ran = new Random();
int max = 20,i=0;
int[][] arr= new int[max][];
for(i=0; i < max;i++)
arr[i] = new int[ran.Next(1,10)];
//pass it to a method
```

```
output(arr);

//method details

public void output(int[][] a)
{
//number of rows
Console.WriteLine("Length :  {0}",a.Length);
int i;
for(i=0; i< a.Length; i++)
//length of each row
Console.WriteLine("Length :  {0}",a[i].Length);
}
```

7.4 Strings

7.4.1 Introduction

C# provides an object to deal with strings. Unlike C and C++, this makes strings in C# powerful and easy to use. In addition regular expressions (not covered here) can be used with strings to provide processing and matching capabilities. Strings are a reference type (like all objects) and the methods that are used to modify the string do not modify the actual string but produce a modified copy (this is called immutability). A string is declared as follows:

```
string identifier;
```

and are typically initialised with literal strings:

```
string str = "Hello World!";
```

It may be a good idea to refer back to the section on constants to see how we deal with string constants, escape characters and verbatim strings.

All the intrinsic data types and most objects support a method called ToString() which converts the contents of an object to a string. For example:

```
int a  = 10;
string s = a.ToString();
```

s now contains the string "10". This can be handy.

Method	**Type**
Compare()	static
Copy()	static
Equals()	static
Format()	static
Length	property
PadLeft()	instance
PadRight()	instance
Remove()	instance
Split()	instance
StartsWith()	instance
Substring()	instance
ToCharArray()	instance
ToLower()	instance
ToUpper()	instance
Trim()	instance
TrimEnd()	instance
TrimStart()	instance

Table 7.4.1 Common string methods

7.4.2 Manipulating Strings

The C# string object has a rich set of methods to support string manipulation (see Table 7.4.1 for commonly used methods).
You should familairise yourself with the capabilities of strings and their methods by tackling the exercises at the end of this chapter. Remember that static methods can only be accessed using the class name (i.e. string) and instance methods can only accessed by an instance name. The help files give information and examples of how these functions should be used.

Exercise 1 *Write a program that allows the user to read a user specified number of double precision floating point numbers from the keyboard, storing them in an array of maximum size 100 say. Your program should then calculate the sum and the average of the numbers input.*

Exercise 2 *Modify your program in the above exercise so that the maximum and minimum values in the data are found and are ignored when calculating the average value.*

Exercise 3 *Write a program that allows the elements of a user input array of doubles to be reversed so that first becomes last etc. Use a separate swap method to carry out the switching of elements.*

Exercise 4 *Write a program that reads an array of 9 integers from the keyboard and produces a histogram of the values as indicated below. A two dimensional array of characters should be used to produce the histogram, filling it with asterisks appropriate to the data values and then just printing out the array. Some scaling will be required if the values are allowed to exceed the number of asterisks that can fit on the screen vertically.*

```
|    *
|   **
|   **
|   ****
| * **** *
| ****** *
|*********
|132467413
```

Exercise 5 *Write a program to accept two strings from the keyboard, compare them and then print out whether or not they are the same.*

Exercise 6 *Write a function to test whether or not a word is a palindrome e.g. MADAM.*

Exercise 7 *Modify the function in above so that white space characters are ignored i.e. so that MADAM IM ADAM for example is deemed a palindrome.*

Exercise 8 *Write a program which accepts a string form the user and outputs it in reverse word order i.e. I am Kieran becomes Kieran am I.*

Exercise 9 *Write a program to simulate the rolling of two dice. Use Random to roll the first die (i.e. to get an integer between 1 and 6) and to separately roll the second die. Calculate the sum of the two rolls, note that the sum is a integer between 2 and 12. Your program should roll the dice 36,000 times and store the number of times each sum occurs in an array. Print out in tabular form the percentages for the occurrence of each sum. Are the results what you'd expect?*

Exercise 10 *The Sieve of Eratosthenes. A prime integer in as integer which can only be divided by 1 and itself. The Sieve of Eratosthenes is a method for finding primes in the first n integers. First create an array of n integers all initialised to 1. Note that non-primes will be set to 0 during the program. Starting at array element 1, find the next array element with value 1, initially this is 2, loop through the rest of the array setting array elements which are multiples of 2 i.e. 2, 4, 6, 8 to value 0. Continue this process until the end of the array is reached. Do we need to go to the end of the array? Using this method print out the primes between 1 and 1000.*

Exercise 11 *Write an object to encapsulate a two 2 X 2 matrix. Provide methods for addition, subtraction and multiplication. Provide a constructor which takes a string argument like the following: "{{1,2},{3,4}}" and initialise the matrix accordingly. Write a test program which should read in the individual matrices and display them in standard format for the user to check them. The user should be allowed to correct/alter any one element of the matrix at a time and check the revised matrix until satisfied. The results of the operation should be displayed along with the two component matrices. Recall that*

$$\begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} * \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix} = \begin{pmatrix} a_1a_2 + b_1c_2 & a_1b_2 + b_1d_2 \\ a_2c_1 + d_1c_2 & a_2d_1 + b_2d_2 \end{pmatrix}$$

Exercise 12 *A real polynomial p(x) of degree n is given by* $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ *where the coefficients an are real numbers. Write an object to encapsulate a polynomial of degree n which can evaluate the polynomial for a particular value x. Internally use an array to stored the coefficients. Write a test program. Provide two methods for calculating the value at x (i) use straightforward calculations to compute the value and (ii) employ Horner's Rule to calculate the value. Recall Homer's Rule for a three degree polynomial for example is* $p(x) = a_0 + x(a_1 + x(a_2 + x(a_3)))$. *Compare the efficiency obtained using both methods (i.e. by timing a test run of 1000000 evaluations in each case).*

Chapter 8
VECTORS AND MATRICES

8.1 Introduction

In this chapter we briefly review the mathematics of vectors and matrices with particular reference to their use in visualisation. We will refer to 2D vectors but the definitions and results can be easily tranferred to 3D.

8.2 Vectors

A vector is an ordered collection of two or more real numbers (technically we don't need to restrict ourselves to real numbers). Normally they are used to represent physical quantities which need two or more numbers to describe them such as velocity, force or acceleration. Symbolically we represent the vector $\mathbf{u}$ and $\mathbf{v}$ as:

$$\begin{aligned} \mathbf{u} &= \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \\ \mathbf{v} &= \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \end{aligned}$$

where u_1, u_2, v_1 and v_2 are real numbers.

8.2.1 Addition, Scalar Multiplication, Subtraction and Basis

Addition is defined as:

$$\mathbf{u} + \mathbf{v} = \begin{bmatrix} u_1 + v_1 \\ u_2 + v_2 \end{bmatrix}$$

and multipication by the scalar a as:

$$a\mathbf{u} = \begin{bmatrix} au_1 \\ au_2 \end{bmatrix}$$

Subtraction follows from these definitions.

$$\mathbf{u} - \mathbf{v} = \mathbf{u} + (-1 \times \mathbf{v}) = \begin{bmatrix} u_1 - v_1 \\ u_2 - v_2 \end{bmatrix}$$

The idea of a basis is that there is a set of vectors from which any other can be made simply by multiplying each basis vector by a scalar and adding the resulting vectors

together. In 2D the basis vectors are:

$$\mathbf{e}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \mathbf{e}_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

so that:

$$\mathbf{v} = v_1\mathbf{e}_0 + v_2\mathbf{e}_1$$

Informally we can think of $\mathbf{e}_0$ being parallel to the $x-$axis and $\mathbf{e}_1$ being parallel to the $y-$axis.

8.2.2 Vector Length and Unit Vector

For most applications the length of a vector is the Euclidean Norm or plain old length in the colloquial sense:

$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2}$$

A unit vector has length 1 and we can find the unit vector ($\mathbf{n}$) parallel to $\mathbf{v}$ as follows:

$$\mathbf{n} = \frac{1}{\|\mathbf{v}\|}\mathbf{v} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

8.2.3 Dot Product

The dot product of two vectors is:

$$\mathbf{u} \cdot \mathbf{v} = u_1v_1 + u_2v_2$$

alternatively if there is an angle θ between the vectors then:

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \, \|\mathbf{v}\| \cos\theta$$

These definitions are equivalent and follow from the cosine rule. If the vectors are orthogonal or perpendicular then $\mathbf{u} \cdot \mathbf{v} = \mathbf{0}$, if $\mathbf{u} \cdot \mathbf{v} > \mathbf{0}$ then the angle is less than 90°, and if $\mathbf{u} \cdot \mathbf{v} < \mathbf{0}$ then the angle is greater than 90°.

The dot product is used to define the projection on one vector onto another. That is the projection of a vector $\mathbf{v}$ onto another vector $\mathbf{w}$ is:

$$proj_{\mathbf{w}}\mathbf{v} = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|^2}\mathbf{w}$$

This is the component of $\mathbf{v}$ along the direction of $\mathbf{w}$. The component perpendicular to $\mathbf{w}$ is:

$$perp_{\mathbf{w}}\mathbf{v} = \mathbf{v} - \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|^2}\mathbf{w}$$

which follows from the addition of vectors.

8.2.4 Cross Product

The cross product computes a new vector orthogonal to the plane containing two vectors. The length of the new vector is equal to the area of the parallelogram subtended by the two vectors. The cross product only really makes sense in 3D therefore consider the 3D vectors:

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Then the cross product is defined to be:

$$\mathbf{v} \times \mathbf{u} = \begin{bmatrix} v_2u_3 - v_3u_2 \\ v_3u_1 - v_1u_3 \\ v_1u_2 - v_2u_1 \end{bmatrix}$$

The magnitude of the cross product is:

$$\|\mathbf{v} \times \mathbf{u}\| = \|\mathbf{u}\| \, \|\mathbf{v}\| \sin\theta$$

where θ is the angle between the vectors. If the two vectors are parallel then $\sin\theta = 0$ and the zero vector results. The rule used most often for figuring out the direction of the resulting vector points is by alligning the index finger of the right-hand along $\mathbf{v}$ and the middle finger along $\mathbf{u}$ then $\mathbf{v} \times \mathbf{u}$ points along the thumb.

8.3 Matrices

A matrix is simply a rectangular 2D array of values. For example:

$$\mathbf{A} = \begin{bmatrix} 1 & 3 \\ 5 & 2 \end{bmatrix}$$

Each individual element of a matrix is called an element. Each matrix has m rows and n columns and is referred to as an m by n matrix. Rows are numbered 0 to $m-1$ and columns are numbered 0 to $n-1$. The numbering can also be from 1 to m and 1 to n depending on prefered notation of the writer. An element is referred to by specifying the row and column to which is belongs. In notation this maybe $(\mathbf{A})_{ij}$ or a_{ij}. So $a_{11} = 2$ above and i and j are referred to a s indices. If $m = n$ then the matrix is called square. If all elements are zero then it is called a zero matrix. If two matrices have the same values for m and n then they have the same size. If two matrices have the same size and corresponding elements have the same values then the matrices are equal. The set of elements whose indices are equal are called the diagonal elements (i.e. a_{00} and a_{11} above) and form the main diagonal. The trace of a matrix is the sum of the diagonal elements. If all elements below the main diagonal are zero the matrix is called upper triangular whereas if the elements above the main diagonal are all zero

then the matrix is called lower triangular. A matrix is called diagonal if all elements off the main diagonal are zero. The identity matrix is square with diagonal elements of one and all other elements equal to zero. Note that a vector can be thought of as a matrix with m rows and one column (a column vector). Other writers prefer to specify that a vector has one row and multiple columns (a row vector). These notations are euivalent and there are adherents to each depending on discipline.

8.3.1 Matrix Addition, Subtraction and Scalar Multiplication

Addition and subtraction are performed component wise just like in vectors:

$$\mathbf{S} = \mathbf{A} + \mathbf{B}$$

thus:

$$s_{ij} = a_{ij} + b_{ij}$$

for each i and j. $\mathbf{A}$ and $\mathbf{B}$ must be the same size to able to perform addition or subtraction and the resulting matrix $\mathbf{S}$ will also have the same size. Multiplication by scalar i.e. $\mathbf{P} = c\mathbf{A}$ is done component wise e.g.:

$$p_{ij} = ca_{ij}$$

for each i and j.

8.3.2 Transpose

An m by n matrix $\mathbf{A}$ has transpose $\mathbf{A}^T$ of size n by m such that at the component level $a_{ij}^T = a_{ji}$. For example:

$$\begin{bmatrix} 1 & 2 & 4 \\ 3 & 6 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 6 \\ 4 & 9 \end{bmatrix}$$

8.3.3 Multiplication of two matrices

A matrix $\mathbf{A}$ of size r by n may be multiplied by a matrix $\mathbf{B}$ of size n by c to produce a matrix $\mathbf{C} = \mathbf{AB}$ of size r by c. In other words the number of columns in the left hand matrix must equal the number of rows in the right hand matrix. The rule for multiplication is:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

One way to think about this is to the matrices up as follows (ignore the symbol $\frown$):

$$\begin{array}{ccc} & \frown & \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \\ \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} & \frown & \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \end{array}$$

Pick an element in $\mathbf{C}$, say c_{23}, then multiply the corresponding row in $\mathbf{A}$ by the corresponding column in $\mathbf{B}$ i.e:

$$c_{23} = a_{21}b_{13} + a_{22}b_{23}$$

In this example $\mathbf{A}$ of size 3 by 2 may be multiplied by a matrix $\mathbf{B}$ of size 2 by 3 to produce a matrix $\mathbf{C} = \mathbf{AB}$ of size 3 by 3 so that in our formula $n = 2$ therefore:

$$\begin{aligned} c_{23} &= \sum_{k=1}^{2} a_{2k}b_{k3} \\ &= a_{21}b_{13} + a_{22}b_{23} \end{aligned}$$

Naturally we can multipy matrices and vectors i.e. suppose we have:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}, \mathbf{C} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

Then:

$$\mathbf{Cv} = \begin{bmatrix} c_{11}v_1 + c_{12}v_2 + c_{13}v_3 \\ c_{21}v_1 + c_{22}v_2 + c_{23}v_3 \\ c_{31}v_1 + c_{32}v_2 + c_{33}v_3 \end{bmatrix}$$

which is a vector.

Chapter 9
OPERATOR OVERLOADING

9.1 Introduction

Consider the division operator introduced earlier and the following code segment:

```
int i = 2, j = 5, ires;
float f = 2f, g = 5f, fres;
ires = i/j;    //division #1
fres = f/g;    //division #2
```

In the first case integer division is performed and the result is 0, whereas in the second case floating point division is performed and the result is 0.4. In both cases the same division operator is used, however depending on the context, a different division implementation is applied. This is operator overloading. In C# nearly all operators can be overloaded to be aware of context and we can provide the appropriate implementation to suit the context. So, for example, the +, -, * and / operators could be overloaded to know what to do if they are being used with complex objects. The end result of this is that we can use objects in much the same way as basic data types such int and float are used. This enhances code readability and makes objects simpler to use for the programmer.

9.2 The Basics

Programmers have a rich set of operators available to them for use with built-in data types. In C# programmers can create their own sophisticated data-types using classes and objects. Operator overloading allows programmers to define how operators behave with their own data types, however the programmer cannot created new operators i.e. the programmer can only redefine the behaviour of existing C# operators.

In C# operators are static methods. When operators are overloaded, methods are written as normal except that they are named **operator** plus the name of the operator to be overloaded. For example a function to overload the addition operator would be named `operator+` and to overload division the function must be named `operator/`. Before you can use an operator with an object the operator must be overloaded for use with that objects of that class.

Operator overloading is most appropriate for mathematical objects such as

complex numbers, vectors and matrices but can also be useful for other objects provided it makes logical sense. There are no restrictions on how you overload an operator and so this technique is open to abuse. For example you could overload + to behave as - and \ to behave as *, but a programmer using such objects and operators would be extremely confused. In general the overloading of operators should be intuitive, easy to understand and correspond in some way to the mathematical notion of the operator e.g. the + operator should always be used to add two objects in some way.

Most operators can overloaded in C#. The following operators can be overloaded

unary	+	-	!	~	++	- -	true	false		
binary	+	-	*	/	%	&	|	^	<<	>>
comparison	==	!=	>	<	>=	<=				

There are several restrictions which apply to operator overloading. Operator precedence (discussed earlier) cannot be changed by overloading, if this poses a problem, however, a programmer using your objects can use parentheses to force a particular precedence if required. Likewise operator associativity cannot be changed by overloading. It is not possible to make a unary operator to behave as a binary operator or vice versa. This single ternary operator (?:) cannot be overloaded. Operators which have both unary and binary forms (i.e. +, -, &, *) must have their unary and binary forms overloaded separately if so desired. You cannot change how an operator behaves with built-in types (e.g. how the + operator behaves with ints). If the + operator is overloaded then the += operator is automatically overloaded as well. This applies to all such composite operators, so we can't overload composite operators directly.

In fact in C# there are certain operators that must be overloaded in pairs, for example if you overload the == operator you must also overload the != operator likewise for > and <, <= and >=.

9.3 Implementing operator overloading

Operator overloading is implemented in a straightforward fashion. Lets revisit the complex class and overload the binary and unary forms of + and - operator for complex objects. Consider the following code:

```
class complex
{
private double real, imag;
private void zero()
{
real = 0.0; imag = 0.0;
}
//some C# properties
public Real
{
```

```
get
{
return real;
}
set
{
real = value;
}
}
public Imag
{
get
{
return imag;
}
set
{
imag = value;
}
}

public Complex()
{
zero();
}

public double magnitude()
{
return Math.Sqrt(real * real + imag * imag);
}

//binary
public static complex operator+(complex left, complex right)
{
complex tmp = new complex();
tmp.real = left.real + right.real;
rmp.imag = left.imag + right.imag;
return tmp;
}

//binary
public static complex operator-(complex left, complex right)
```

```
{
complex tmp = new complex();
tmp.real = left.real - right.real;
rmp.imag = left.imag - right.imag;
return tmp;
}

//unary
public static complex operator+(complex val)
{
complex tmp = new complex();
tmp.real = val.real;
rmp.imag = val.imag;
return tmp;
}

//binary
public static complex operator-(complex val)
{
complex tmp = new complex();
tmp.real = -val.real;
rmp.imag = -val.imag;
return tmp;
}
}
```

Next heres some code using the overloaded operators:

```
complex a = new complex();
complex b = new complex();
a.Real = 2;
a.Imag = -4;
b.Real = -2;
a.Imag = 7;
complex c = a + b;
```

The statement `complex c = a + b;` in the code above invokes the `operator+` method as follows: `complex c = complex.operator+(a,b);`. It is important to think about how the operator statement is converted to a call to a method. Notice that both the + and - operator methods return new objects, this enables us to tie

several calls to operators together and the statement `complex c4 = c1 - c2 - c3` is interpreted by the compiler as: `complex c4 = complex.operator-(c1,complex.operator-(c2,c3))` (note: left to right associativity).

9.4 Type Conversions

C# supports both implicit (i.e. int to long) and explicit (i.e. long to int using a cast) conversion of data types. Sometimes it is useful to be able to support the same operations for objects we create. To do this we must overload the conversion operator which comes in two flavours: implicit (compiler applies the operator without fuss) and explicit (the programmer must use to the cast syntax to avoid an error from the compiler). These conversion operator overloads are often referred to as cast operators also.

Conversion operator overloads are public and static (because it makes sense to convert from one type to another without an instance of the type being converted to being in existence). The syntax is as follows:

```
public static [implicit, explicit] operator data_type1(data_type2 parameter)
{
}
```

One of the implicit or explicit keywords must be used to describe whether or not a cast syntax must be used. The data_type1 is used as a method name and this is the data type resulting from the conversion, whereas data_type2 is the parameter is the method argument and is the type which is undergoing the conversion.

A simple example should clarify matters. Consider a class which encapsulates fractional data:

```
public class Fraction
{
private int numerator;
private int denominator;
//constructor #1
public Fraction(int numerator, int denominator)
{
this.numerator = numerator;
this.denominator = denominator;
}
//constructor #2
public Fraction(int wholenumber)
{
numerator = wholenumber;
```

```
denominator = 1;
}
//convert int to a fraction implicitly
public static implicit operator Fraction(int pInt)
{
//create and return a new Fraction using constructor #2
return new Fraction(pInt);
}
//convert a Fraction to an int
public static explicit operator int(Fraction pFract)
{
//create and return a new Fraction using constructor #2
return pFract.numerator/pFract.denominator;
}
/*  lots more stuff omitted here */
}
```

Programming Assignment No 2: Vector and Matrix class.

You are required to implement a class to represent a vector and a matrix which supports all the operations defined in the previous chapter. Provide some code to test your implementation and provides output for verification.

Chapter 10
INHERITANCE, INTERFACES AND POLYMORPHISM

10.1 Introduction

When we model the world with objects and consider the benefits of abstract data types (i.e. data protection, robustness, reusability), it is natural to want to be able to build upon software (i.e. a collection of objects) that has already been created and tested and appears to work. This is usually a good starting point for future developments. Early on we considered the concepts of specialisation and generalisation. In this chapter we consider the language mechanism for incorporating specialisation and generalisation into our OO programs using inheritance. Like most mechanisms, this is optional and should really only be used where it makes sense. For example, does it make sense to say that a dog is a specialised human, that barks instead of talks?

Inheritance is all about taking an existing class as a basis and then creating a new class by either adding functionality (properties and methods) or modifying existing functionality (usually methods only). In OOP terminology the class used as a foundation is called the parent class, base class or ancestor class whereas the new class is termed the child class or derived class. In order to facilitate our discussions in this chapter, we will be using the Animal class as our base class.

```
public class Animal
{
private int age = 0;
public int Age
{
get { return age; }
set { age = value; }
}
private int health = 0;
public Animal()
{
age = 1;
}
public Animal(int age)
{
```

```
this.age = age;
}
public void eatFood()
{
health++;
}
public void doExercise()
{
health++;
}
public void liveOneday()
{
health--;
}
public void outPut()
{
Console.WriteLine("Health = {0}", health);
}
}
```

We will also consider the related concepts of interfaces and polymorphism in this chapter.

10.2 Inheritance

Unlike other OO languages C# only allows a class one base class. It does not support multiple inheritance, probably because you can get into a complete muddle with multiple inheritance. The syntax for implementing inheritance is simple:

```
public class derived_class :  base_class
{
//implementation details
}
```

Lets look at this in our example and see what we get for free.

```
public class Chicken :  Animal
{

}
```

Here we are specifying that the Chicken class inherits (or is derived) from the Animal class. Consider now the following code:

```
Animal a = new Animal();
Chicken c = new Chicken();
a.doExercise();
a.eatFood();
a.liveOneday();
a.outPut();
c.doExercise();
c.eatFood();
c.liveOneday();
c.outPut();
```

Simply by inheriting we get all the properties and methods of the Animal class in the Chicken class. Check out what happens if a default constructor is not provided for the Animal class.

However, this isn't very useful as we are basically recycling the Animal class. The first thing we could do is add new properties and methods to the child class. Lets focus first on derived class properties and constructors and then look at derived class methods.

10.2.1 Derived class properties and constructors

One way to enhance the behaviour of the derived class is by add more properties (or data). This is quite simple, just add definitions to the derived class:

```
public class Chicken :  Animal
{
private int numEggs = 0;
public int NumEggs
{
get { return numEggs; }
set { numEggs = value; }
}
}
```

Because numEggs is part of the definition of the Chicken class we can directly access it in the methods of the Chicken class (as usual, a user of a Chicken object cannot directly access the numEggs private data). What about the base class data members (age and health)? For example, in a constructor it would be nice to be able to specify the age and the number of eggs. For example I may try this:

```
public class Chicken :  Animal
{
private int numEggs = 0;
public int NumEggs
{
get { return numEggs; }
set { numEggs = value; }
}
public Chicken()
{
age = 2; //***WRONG***
numEggs = 3; //This is fine
}
}
```

The age property was defined as private to the Animal class therefore even in a derived class we cannot directly access the data (we can however manipulate using interface methods defined in the base class, in the current case you could use the property Age). We can directly access the numEggs private data as that is defined in the Chicken class. This leaves with a problem how do we initialise data in a constructor? We could use base class methods in the derived class constructor or we could invoke the base class constructor to do the work. Lets explore these options:

```
public class Chicken :  Animal
{
private int numEggs = 0;
public int NumEggs
{
get { return numEggs; }
set { numEggs = value; }
}
public Chicken(int age, int numEggs)
{
this.numEggs = numEggs;
Age = age; //note using the base class public property
}
}
```

Alternatively the base class constructor can be invoked using the keyword `base`.

```
public class Chicken :  Animal
{
```

```
private int numEggs = 0;
public int NumEggs
{
get { return numEggs; }
set { numEggs = value; }
}
public Chicken(int age, int numEggs) :  base(age)
//invoke base class constructor to initialise age
//the recommended approach
{
this.numEggs = numEggs;
}
}
```

There is one other option that can be used, however, it is not recommended because it contravenes the basic rules of OOP (data hiding in particular). Making data public gives access to an object internals and misuse could cause your object to fail. Making it private implies that no user of the object, including derived classes, can interfere with the data in an uncontrolled manner. However, this can be tedious in derived classes. In this situation we can make data in the base class protected (this means that for users of the base class this data is hidden just like private), but derived classes can directly access the data. This is a dangerous practice because if the base class implementation changes then your derived class may fail and/or require a rewrite. Clearly the OO philosophy of data hiding is being broken.

10.2.2 Derived class methods

Adding extra functionality

Simply use the standard method for adding methods to your class. Provided there is not a method with the same name and signature in the base class there should be no difficulties.

Modifying base class functionality

In our example the Chicken object has an extra data member numEggs, when we call the outPut() method it would be more accurate to output both the age and the number of eggs. Therefore we need a different implementation of the outPut() method in the derived class. We do this by using the `new` keyword (`new` has a different meaning in this context and should not be confused with its use in creating objects). For example:

```
public class Chicken :  Animal
{
```

```
private int numEggs = 0;
public int NumEggs
{
get { return numEggs; }
set { numEggs = value; }
}
public Chicken(int age, int numEggs) :  base(age)
//invoke base class constructor to initialise age
//the recommended approach
{
this.numEggs = numEggs;
}
public new void outPut()
{
base.outPut();
Console.WriteLine("I lay {0} eggs", numEggs);
}
}
```

In this example the `new` keyword is used to indicate that for Chicken objects the derived class version of outPut() is to be used instead of the base class version. We can call the base class version inside in our derived class version using the keyword `base`. This means that we can build upon existing functionality without having to start for scratch. If the new keyword were omitted in this case, a compiler warning would be generated.

10.3 Polymorphism I (Inheritance)

Inheritance benefits software development by facilitating code reuse. Another important feature is polymorphism (poly meaning many and morph meaning form). The basic idea is that we are able to use many different forms of objects without knowing the exact details. We will return to polymorphism later in the chapter after interfaces have been discussed. A good example of polymorphism is a ringing telephone signal. There are many types of telephone handsets with different ring tones etc. The phone company dose not need to send a different signal to each handset, instead it sends the same signal without knowing the details and each handset responds correctly in some fashion. In our chapter example, we would like to be able to call the outPut() method of an object without knowing whether it is of type Animal or Chicken and get the correct version of the method invoked. We do this using polymorphic methods.

10.3.1 Polymorphic Methods

Polymorphic methods are create using the `virtual` keyword. Methods intended to be polymorphic need only be qualified as virtual in the base class. Let us change our

Animal definition to:

```
public class Animal
{
private int age = 0;
public int Age
{
get { return age; }
set { age = value; }
}
private int health = 0;
public Animal()
{
age = 1;
}
public Animal(int age)
{
this.age = age;
}
public void eatFood()
{
health++;
}
public void doExercise()
{
health++;
}
public void liveOneday()
{
health--;
}
public virtual void outPut()
{
Console.WriteLine("Health = {0}", health);
}
}
```

The method outPut() is now designated as polymorphic. We have two choices for the outPut() method in Chicken i) `new` and ii) `override`. If we use new then the method behaves as before (i.e. it has a new version of outPut() that can only be invoked with an instance of the Chicken class. On the other hand if we use the override keyword polymorphic behaviour occurs. Again the idea is that we modify the functionality of

outPut() for Chicken objects.

First lets look at an example with new (i.e. with the Chicken class definition above):

```
//inside the default class of a console application
static void main()
{
Animal a = new Animal(2);
Chicken b = new Chicken(4, 10); //age 4, 10 eggs
a.doExercise();
b.eatFood();
b.doExercise();
Process(a);
Process(b);

}
static void Process(Animal a)
{
Console.WriteLine("Process");
a.outPut();
}

//Resulting output
Process
Health = 1
Process
Health = 2
```

ON the other hand if I modify the new to override in the Chicken class:

```
public class Chicken :  Animal
{
private int numEggs = 0;
public int NumEggs
{
get { return numEggs; }
set { numEggs = value; }
}
public Chicken(int age, int numEggs) :  base(age)
//invoke base class constructor to initialise age
//the recommended approach
{
this.numEggs = numEggs;
```

```
}
public override void outPut()
{
base.outPut();
Console.WriteLine("I lay {0} eggs", numEggs);
}
}
```

The resulting output is:

```
//Resulting output
Process
Health = 1
Process
Health = 2
I lay 10 eggs
```

This is because, even though the Process() method is being passed a reference to an Animal object, the polymorphic nature of the outPut() method means that the correct version is called even though the exact type of object is unknown in the Process() method.

10.3.2 Why bother with new and override?

Suppose a company A wrote the Animal class and Company B wrote the Chicken class using animal as a base class. B has no control over Animal and A has no control over Chicken. To begin with suppose B created an additional method called `public int GetWeight()`. Now, in a bizarre coincidence, Company A decides to add a method `public virtual int GetWeight()` to Animal in an updated version. What happens to the Chicken class of company B? In C++, polymorphic behaviour would occur, but this may not be the required behaviour. In C# Company B will have to decide which behaviour they require because a compiler error forces then to designate their version as either new (non-polymorphic) or override (polymorphic behaviour). C# gives greater flexibility and control.

10.4 Abstract and Sealed Classes

Consider the example we have been using to date; does it make sense to be able to instantiate an Animal object? Is their such a thing as an "Animal"? In the real world there are only specialised types of animal (i.e. chicken, goat, dog etc.) but there is no physical object which is an animal. An animal is really an abstract concept. However, if someone said they saw an animal we know that the animal will have certain properties (eyes, nose, teeth etc. usually). Abstract classes bring this idea into OOP. Sometimes it makes sense to define a class, which cannot be instantiated,

but which contains definitions for properties and methods, which derived classes **are required to implement**. Abstract methods have no implementation and by making one method abstract the whole class becomes abstract (i.e. cannot be instantiated). An abstract class establishes a fixed set of methods which derived classes must implement. Making a method abstract is accomplished by placing the keyword abstract at the start of the method, the method contains no implementation and the method definition is terminated by a semi-colon. When one method is abstract then the keyword abstract must precede the class definition as well. For example:

```
abstract public class Animal
{
private int age = 0;
public int Age
{
get { return age; }
set { age = value; }
}
private int health = 0;
public Animal()
{
age = 1;
}
public Animal(int age)
{
this.age = age;
}
public void eatFood()
{
health++;
}
public void doExercise()
{
health++;
}
public void liveOneday()
{
health--;
}
abstract public void outPut(); //abstract method
}
```

When implementing the outPut() method in the derived class you should notice

that it must be an override (e.g. try new, virtual and no qualifier).

Whereas an abstract class is designed for inheritance, we can designate a class as sealed, whereby it cannot act as a base or parent class. This is accomplished by prefixing the class definition with the keyword sealed. Sealing is used to avoid accidental inheritance and should always be applied to classes consisting of static properties and methods only.

10.5 Interfaces

An interface is a relatively simple concept akin to an abstract base class but with much greater flexibility. An interface is essentially a set of unimplemented methods (signatures) that can be thought of as a contract. Because an implementation is not required, interfaces can come in handy during the design phase of a project, i.e. we do not need to worry about coding details. Interfaces are not useful in themselves, but when they are subscribed to by an **implementing class** (i.e. we say the class agrees to the interface contract, also termed subscribing to the interface or implementing the interface), the implementing class agrees to provide an implementation for every method signature in the interface. Finally, a client of an object of the implementing class can find out if the object subscribes to a particular interface, therefore without knowing the exact type of an object the client knows that certain methods are guaranteed to exist (if this sounds a bit like polymorphism, it is; see the next section for more details). Using the interface construct in programs facilitates the OOP concept of the interface and thereby interoperability. In Ireland all our plugs and electrical sockets are the same. When you move house or apartment you don't have to put new plugs on all your electrical appliances. The uniform interface promotes interoperability. Likewise in our programs the interface construct means that if we know an object supports an interface then we can call the interface methods without difficulty.The syntax for interfaces in C# is:

```
[access_modifier] interface interface_name [:  base_list] {interface_body}
```

The access modifiers are typically public or private and interfaces are named beginning with a capital I by convention. Lets look at an example:

```
public interface IOutput
{
void Output();
void Print(int numcopies);
int Status { get; set; }
}
```

The interface consists of a list of methods without implementations (this is provided

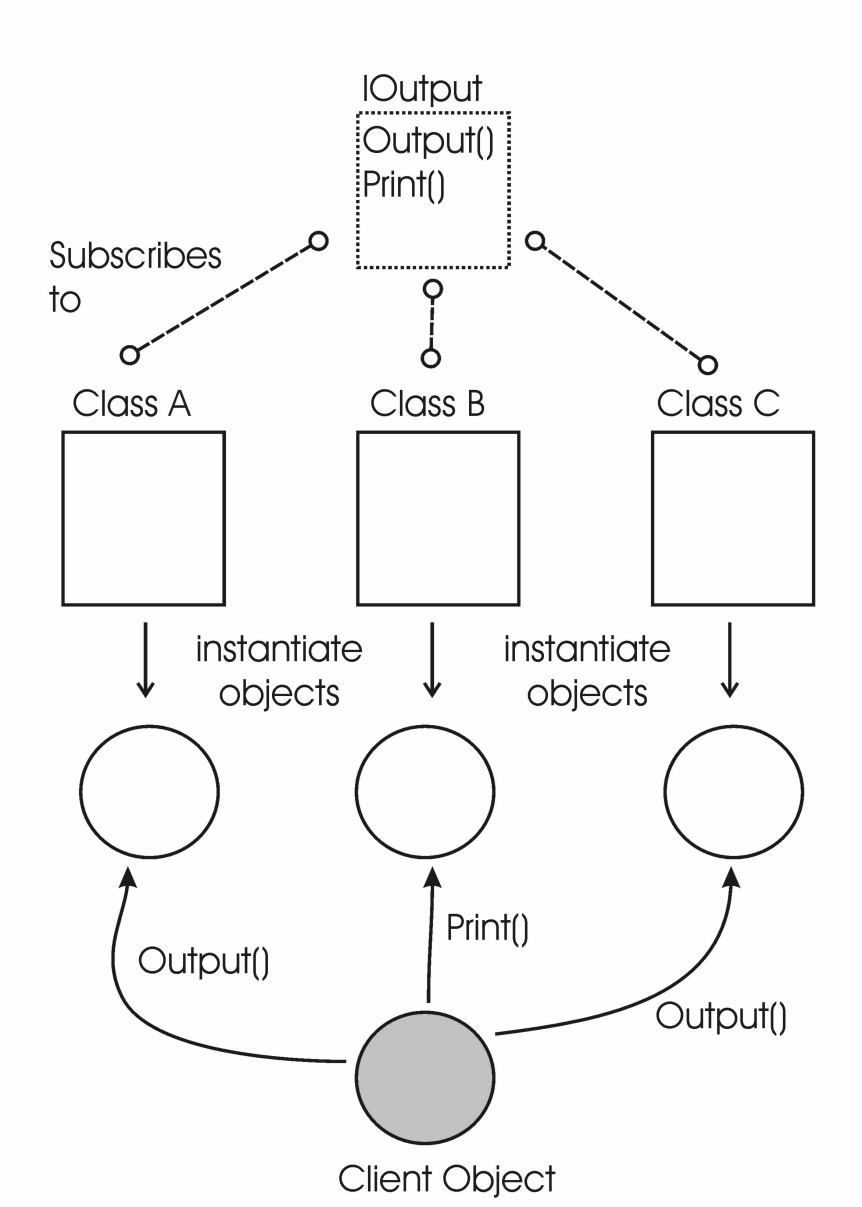

Figure 10.5.1 Schematic representation of the role of an interface. Classes, not necessarily from the same inheritance tree, subscribe to an interface. Clients of objects can then call interface methods without knowing the exact type of the object.

by subscribing classes). Notice also I can specify properties to be part of the interface, but again the implementation details are up to the subscribing class. Suppose the Chicken class subscribed to the above interface. If I fail to implement the methods and property of IOutput, compiler errors will prevent the program from building.

```
public class Chicken :  IOutput
{
private int numEggs = 0;
public int NumEggs
{
get { return numEggs; }
set { numEggs = value; }
}
public Chicken(int age, int numEggs) :  base(age)
//invoke base class constructor to initialise age
//the recommended approach
{
this.numEggs = numEggs;
}
public void Output()
{
Console.WriteLine("I'm a chicken");
}
public void Print(int numcopies)
{
for(int i = 0; i < numcopies; i++)
Console.WriteLine("I'm a chicken");
}
private int status = 0;
public int Status {
get { return status;}
set { status = value;}
}
}
```

Notice that the subscribing class uses the same signature as the interface and that we provide the private int status to support the property implementation. Notice that we do not specify access modifiers for the methods in the interface they are public due to the interface being public. If the interface was private then in order to match the method signatures we would need to prefix our implementations by the access modifier private.

Unlike C# inheritance classes are free to subscribe to more than one interface,

this is done by for example `class X : IOutput, ICompress` etc. We wont be doing anything so sophisticated here, but if you are interested try it out. What happens if two interfaces support the method signatures and a class subscribes to both. Does it cause a problem?

Clearly an interface is not instantiable (because in itself it has no implementation), however we can create variables which can refer to object which implements a particular interface. In the case of our example:

```
Chicken c = new Chicken(4, 3);
IOutput opref = (Chicken)c;
opref.Output();
opref.Print();
opref.Status = 1;
```

Notice that the cast operator is used to force the compiler to accept the assignment. As before we are ignoring the compilers misgivings and essentially saying "trust me I know it implements the IOutput interface". Additionally it is important to see that the creation of a variable which references an interface does not create any new object no more than writing `Chicken c` would create an object. Once the interface reference is assigned we can use it to invoke methods of the interface and because it is referring to the Chicken object it will perform Chickens implementation. This is a fragile system though. I could inadvertently use casting to assign an object which does not support the interface to an interface reference. The result would be an exception (fatal error).

Fortunately C# provides two operators which allow us to basically interrogate an object about the interface(s) it supports and then call methods only if it is safe to do so. The first one is the **is** operator. This is used to find whether an object subscribes to an interface or not. For example:

```
Chicken c = new Chicken(4,3);
if(c is IOutput)
{
IOutput opref = (IOutput)c;
opref.Output(); //can't fail because we know here
}
```

The is operator returns true if an object supports an interface and (you've guessed it) false otherwise.

The **as** operator combines the is and cast operators. First it checks to see if the cast is valid (i.e. like the is operator) and then it casts the object to the interface. If it fails the interface reference hold the null value. For example:

```
Chicken c = new Chicken(4,3);
IOutput opref = c as IOutput;
if(opref != null)
opref.Output(); //can't fail because we know here
}
```

A class which implements an interface may mark any of the interface methods as virtual. Classes derived from the implementing class can override or provide new implementations. Consider the following example:

```
interface IStorable
{
void Read();
void Write();
}

public class Document :  IStorable
{
public Document(string s)
{
Console.WriteLine("Creating document with {0}",s);
}
public virtual void Read()
{
Console.WriteLine("Document read method for IStorable");
}
public void Write()
{
Console.WriteLine("Document Write Method for IStorable");
}
}

public class Note :  Document
{
public Note(string s) :  base(s)
{
Console.WriteLine("Creating note with {0}",s);
}
public override void Read()
{
Console.WriteLine("Overriding the Read Method for Note!");
```

```
}
public new void Write()
{
Console.WriteLine("Implementing the Write Method for Note!");
}
}

//Some test code

Document theNote = new Note("Test Note");

theNote.Read();
theNote.Write();

IStorable isNote = theNote as IStorable;
if(isNote != null)
{
isNote.Read();
isNote.Write();
}

Console.WriteLine("********* Part II ************");

Note note2 = new Note("Second test");

note2.Read();
note2.Write();

IStorable isNote2 = note2 as IStorable;
if(isNote2 != null)
{
isNote2.Read();
isNote2.Write();
}
```

The output from the program is as follows:

```
//constructors
Creating Document with:  Test Note
Creating note with:  Test Note
//using base class reference to derived object i.e.  theNote
```

```
Overriding the Read Method for Note!
Document Write Method for IStorable
//cast to Istorable interface
Overriding the Read Method for Note!
Document Write Method for IStorable

********* Part II ************

//constructors
Creating Document with:  Second Test
Creating note with:  Second Test
//derived class reference to derived class i.e.  note2
Overriding the Read Method for Note!
Implementing the Write Method for Note!
//cast to Istorable interface
Overriding the Read Method for Note!
Document Write Method for IStorable
```

Whats going on here? The base class reference theNote holds a reference to a note object. Because read is virtual theNote. Read() invokes the note version. By contrast write is non-virtual and calling write for theNote calls the base class (Document) version because the reference is of type Document. The same output is obtained from the interface reference. This is notable, the interface reference calls virtual functions as we would expect but for the non-virtual methods (e.g. Write()), but the base class version is called for read. Can you explain the second part of the output?

10.6 Polymorphism II (Interface)

Interfaces allow us to use polymorphism in a much more general sense. Whereas for polymorphism in inheritance we need to know that some object is part of a particular inheritance tree, interfaces allow us to implement the same behaviour except across inheritance trees. Using the is and as operators we can come across an object and find out which interfaces it supports and then use that knowledge to be able to invoke methods for the object. this all happens without knowing the exact type of the object.

10.7 Using Standard Interfaces to Add Functionality

Back when we considered arrays for the first time we noted that there were some features (such as using the Array.Sort(), a static method) that come for free with arrays of intrinsic data types. C# uses the interface mechanism to enable developers to make arrays of their own objects be compatible with these functionalities.

10.7.1 Array.Sort(), `IComparable` and `IComparer`

An array of objects cannot be sorted using the static method Array.Sort() unless the object subscribes to the `IComparable` interface and implements the method `int CompareTo(object obj)`. This method returns a postive int value if the instance is greater than obj, a negative value if the instance is less than obj and zero if they are equal. Consider the following code:

```
class Person :IComparable
{
private string firstname;
public string Firstname
{ get { return firstname; } set { firstname = value; } }
public int CompareTo(object obj)
{
Person p = obj as Person;
//just use the normal string comparison functionality
return this.firstname.CompareTo(p.firstname);
}
}
class Program
{
static void Main(string[] args)
{
Person [] p = new Person[3];    //make an array of objects
p[0] = new Person();    //create and assign objects
p[0].Firstname = "Kieran";
p[1] = new Person();
p[1].Firstname = "Teddy";
p[2] = new Person();
p[2].Firstname = "James";
Array.Sort(p);    //sort the objects
}
}
```

By subscibing to the `IComparable` interface you have to implment the method `int CompareTo(object obj)`. In this case, a standard lexicographic comparison is used as implemented by the CompareTo() method of the string class.

Suppose now that we want to have a more sophisicated comparison method where the user of the object can decide on the method of comparison used. Achieving this requires a little more effort. For the sake of argument suppose we wish to be able to sort on the length oft he name and also the names in reverse. This technique can also be used if you want to make an object you didn't develop sort in a different man-

ner. The steps are: (1) create a comparer object which subscribes to the IComparer interface and implement the Compare method (2) as well as the array to be sorted pass an instance of this comparer object to the Array.Sort() method. Consider the code:

```
class Person :IComparable
{
private string firstname;
public string Firstname
{ get { return firstname; } set { firstname = value; } }
public int CompareTo(object obj)
{
Person p = obj as Person;
return this.firstname.CompareTo(p.firstname);
}
}
class PersonComparer :  IComparer
{
private PersonCompareType pct;
public PersonCompareType Pct
{
get { return pct; }
set { pct = value; }
}
public PersonComparer(PersonCompareType ipct)
{
pct = ipct;
}
public int Compare(object x, object y)
{
Person px = x as Person;
Person py = y as Person;
switch(pct)
{
case PersonCompareType.Firstname:
return px.Firstname.CompareTo(py.Firstname);
case PersonCompareType.Length:
return px.Firstname.Length - py.Firstname.Length;
case PersonCompareType.Reverse:
return ReverseString(px.Firstname).CompareTo(ReverseString(py.Firstname));
}
```

```
return 0;
}
private string ReverseString(string val)
{
char[] ch = val.ToCharArray();
Array.Reverse(ch);
return new string(ch);
}
}
public enum PersonCompareType
{
Firstname,
Length,
Reverse
}
class Program
{
static void Main(string[] args)
{
Person [] p = new Person[3];
p[0] = new Person();
p[0].Firstname = "Kieran";
p[1] = new Person();
p[1].Firstname = "Teddy";
p[2] = new Person();
p[2].Firstname = "James";
Array.Sort(p);
PersonComparer pc = new PersonComparer(PersonCompareType.Length);
Array.Sort(p, pc);
pc.Pct = PersonCompareType.Firstname;
Array.Sort(p, pc);
pc.Pct = PersonCompareType.Reverse;
Array.Sort(p, pc);
}
}
```

The PersonComparer object is used to aid the Array.Sort() method as exactly how the Person objects are compared.

10.7.2 Enumerations

If you are writing an object, which contains within it a number of elements, it may be nice to be able to go sequentially through each element using the foreach statement.

In order to be able to do this your class must implement the IEnumerable interface.

Now suppose we have a class called stringcoll which essentially encapulates an array of string objects and we wish to use the foreach statement. Then the aim is to able to write:

```
stringcoll c = new stringcoll(); //array of strings hard-coded
foreach(string s in c)
Console.WriteLine(s);
```

The compiler actually turns the above into the following:

```
IEnumerator enumerator = c.GetEnumerator();
while(enumerator.MoveNext())
{
string s = (string)enumerator.Current;
Console.WriteLine(s);
}
```

Now the IEnumerable interface is:

```
interface IEnumerable
{
public IEnumerator GetEnumerator();
}
```

and that of IEnumerator is:

```
interface IEnumerator
{
public object Current;
public void Reset();
public bool MoveNext();
}
```

Considering the definitions and the code the compiler produces the mechanics of what is supposed to happen should be becoming clear. The following gives the detailed implementation in an old way (perfect but takes a fair amount of coding) and in a new way (very short). The old way is very clear and we could use it to enumerate our object data in any which way we choose. We may loose this flexibility with the new implementation, it certainly is less obvious due to it's succinctness.

```
//old implementation
class stringcollold :  IEnumerable
{
string[] arr = new string[10] { "John", "Edward", "Pat", "Mary", "Dan",
"Denis", "Liya", "Anca", "Tom", "Kathleen" };
/********MAKE A CLASS WHICH IMPLEMENTS IEnumerator ********/
private class stringcollEnumerator :  IEnumerator
{
//a reference to the obj being enumerated
private stringcollold theobj;
//an index to know where we are in the enumeration
private int index;
//constructor
public stringcollEnumerator(stringcollold itheobj)
{
theobj = itheobj;
index = -1;
}
//implement the interface
public bool MoveNext()
{
index++;
if (index >= theobj.arr.Length)
return false;
else
return true;
}
public void Reset()
{
index = -1;
}
public object Current //a property
{
get
{
return(theobj.arr[index]);
}
//no set implies its readonly
}
}
/*****END MAKING CLASS******/
public IEnumerator GetEnumerator()
```

```
{//required for IEnumerable
return (IEnumerator)new stringcollEnumerator(this);
}
}
//new implementation
class stringcoll :  IEnumerable
{
string[] arr = new string[10] { "John", "Edward", "Pat", "Mary", "Dan",
"Denis", "Liya", "Anca", "Tom", "Kathleen" };
public IEnumerator GetEnumerator()
{
int i;
for (i = 0; i < arr.Length; i++)
yield return arr[i];
}
}
```

The code for using each version is identical:

```
//old implementation
stringcollold c = new stringcollold();
foreach (string s in c)
Console.WriteLine(s);
Console.WriteLine();
//new implementation
stringcoll cnew = new stringcoll();
foreach (string s in cnew)
Console.WriteLine(s);
```

The `yield return` statement returns one element and moves the the position to the next element. There is also a `yield break` statement which terminates the iteration. A `return` statement is invalid in this context. These statements can only exist inside a method which has return type IEnumerable or IEnumerator.

We can however use the yield return method to quickly produce varied behaviour:

```
//new implementation
class stringcoll :  IEnumerable
{
string[] arr = new string[10] { "John", "Edward", "Pat", "Mary", "Dan",
```

```
"Denis", "Liya", "Anca", "Tom", "Kathleen" };
public IEnumerator GetEnumerator()
{//this is the default
int i;
for (i = 0; i < arr.Length; i++)
yield return arr[i];
}
public IEnumerable Reverse()
{
int i;
for (i = arr.Length -1; i >= 0; i--)
yield return arr[i];
}
public IEnumerable Subset(int start, int count)
{//note should have lots of checking for validity of inputs
int i;
for (i = start; i < start + count; i++)
yield return arr[i];
}
}
```

Test code:

```
//new implementation
stringcoll cnew = new stringcoll();
foreach (string s in cnew)//default
Console.WriteLine(s);
Console.WriteLine();
foreach (string s in cnew.Reverse())
Console.WriteLine(s);
Console.WriteLine();
foreach (string s in cnew.Subset(2,4))
Console.WriteLine(s);
```

Chapter 11
NUMERICAL INTEGRATION

11.1 Introduction

It is not possible to find a simple solution to every integral. For example statisticians need know the value of the following integral quite regularly:

$$p(\alpha) = \int_0^{\alpha} e^{-x^2} dx,\ \alpha \in [0, \infty)$$

However there is no known simple function which when differentiated gives the integrand. The traditional solution is to give the function a special name and to tabulate its values. In the case of the current example it is called the normal distribution and its values are found in most standard mathematical tables. If we wish to find a value on a computer we have to find an approximate, but accurate, value. If we wish to compile a table of values we need to do the same procedure many times. The question addressed in this section is how best to do these types of calculations.

11.2 Finite element interpolation integration

11.2.1 Piecewise constant interpolation (q_o)

The is simplest (and most rough) finite element interpolation which is rarely used in practice. Under this scheme the function is approximated throughout each element by its value at either the left or right node of the element (see figure **??**)..

$$q_0^{\text{left}}(x) = f(x_i),\ x \in [x_i, x_{i+1}),\ i = 1, 2, ..n-1$$

$$q_0^{\text{right}}(x) = f(x_{i+1}),\ x \in (x_i, x_{i+1}],\ i = 1, 2, ..n-1$$

Clearly by increasing the number of elements the average element width decreases giving a more accurate solution (particularly if we are using equally spaced nodes).

11.2.2 Trapezoidal or piecewise linear interpolation (q_1)

Intuitive formulation

This is a commonly used scheme and we will study it in some detail. Suppose we wish to approximate $f(x)$ on the interval $[a, b]$ by the piecewise function $q_1(x)$, note in this case left or right is irrelevant. The interval $[a, b]$ is subdivided into $n-1$ elements

by choosing ordered nodes $a = x_1 < x_2 < \ldots < x_{n-1} < x_n = b$. The i^{th} element is $[x_i, x_{i+1}]$, $i = 1, 2, \cdots, n-1$. In each element $f(x)$ is approximated by a first order polynomial. Considering the i^{th} element (see **??**)we find that:

$$q_1(x) = f(x_i)\frac{x - x_{i+1}}{x_i - x_{i+1}} + f(x_{i+1})\frac{x - x_i}{x_{i+1} - x_i}$$

Do a quick sanity check:

$$\begin{aligned} q_1(x_i) &= f(x_i) \\ q_1(x_{i+1}) &= f(x_{i+1}) \end{aligned}$$

Using this scheme we have a first order polynomial approximation on each element, however the coefficients vary from element to element. There is no single simple formula for $q_1(x)$, it is a piecewise function.

11.2.3 Generalities

In this section we take a general approach to numerical integration based on our piecewise finite element interpolation schemes. We wish to evaluate:

$$\int_a^b f(x)\,dx$$

as accurately as necessary. First we divide the interval into $n-1$ elements $a = x_1 < x_2 < \ldots < x_{n-1} < x_n = b$ such that the i^{th} element is $[x_i, x_{i+1}]$, $i = 1, 2, \ldots, n-1$. We approximate $f(x)$ by $q_m(x)$ a piecewise finite element interpolation (we only previously studied $m = 0, 1, 2$ by the way). So that:

$$\begin{aligned} \int_a^b f(x)\,dx &\approx \int_a^b q_m(x)\,dx = \int_{x_1}^{x_n} q_m(x)\,dx \\ &= \sum_{i=1}^{n-1}\int_{x_i}^{x_{i+1}} q_m(x)\,dx \end{aligned}$$

As we have already derived formulae for $q_m(x)$, $m = 0, 1$ we proceed by deriving numerical integration techniques for each variety.

11.2.4 Rectangular rule (based on $q_0(x)$)

Using the piecewise constant interpolation ($q_0^{\text{left}}(x)$) derived earlier we have that:

$$\begin{aligned} \int_a^b f(x)\,dx &\approx \sum_{i=1}^{n-1}\int_{x_i}^{x_{i+1}} q_0^{\text{left}}(x)\,dx = \sum_{i=1}^{n-1}\int_{x_i}^{x_{i+1}} f(x_i)\,dx \\ &= \sum_{i=1}^{n-1} f(x_i)\int_{x_i}^{x_{i+1}} dx = \sum_{i=1}^{n-1} f(x_i)(x_{i+1} - x_i) = \sum_{i=1}^{n-1} f(x_i)\,h_i \end{aligned}$$

where $h_i = x_{i+1} - x_i$. The formulation using $q_0^{\text{right}}(x)$ is similar.

11.2.5 Trapezoidal rule (based on $q_1(x)$)

From our intuitive formulation earlier we found that:

$$q_1(x) = f(x_i)\frac{x - x_{i+1}}{x_i - x_{i+1}} + f(x_{i+1})\frac{x - x_i}{x_{i+1} - x_i}$$

and

$$\int_a^b f(x)\,dx \approx \int_a^b q_1(x)\,dx = \frac{1}{2}(x_{i+1} - x_i)(f(x_i) + f(x_{i+1}))$$

and using the usual notation:

$$\int_a^b f(x)\,dx \approx \frac{h_i}{2}(f(x_i) + f(x_{i+1}))$$

In the special case of equally spaced nodes we have:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}[f(x_1) + 2\{f(x_2) + \ldots + f(x_{n-1})\} + f(x_n)]$$

the better known composite trapezoidal rule.

11.2.6 Adaptive Quadrature

Equally spaced nodes is one approach to numerical integration, however we may wonder if there is a better way? Could we get an equally accurate answer with less nodes by choosing our nodes in a more intelligent way? We would be happy with this approach if the total cost of computation were lower i.e. the problem was solved faster (less computer time), human time input was not increased (i.e. in setting up the problem, coding the algorithm etc).

We now look at a simplified version of this problem. We wish to calculate an accurate answer with the least number of nodes. Our approach is to adapt the nodes to suit the particular function being integrated and in particular we will derive an adaptive trapezoidal rule. Intuitively it is clear that where the function varies slowly we need to deploy few nodes whereas many nodes are required where the function is varying rapidly.

We wish to find:

$$\int_a^b f(x)\,dx$$

so we start with just two nodes and our approximation is:

$$I(a,b) = \frac{h}{2}[f(a) + f(b)]$$

where $I(a,b)$ is the estimate of $\int_a^b f(x)\,dx$ and $h = b - a$. If this approximation is sufficiently accurate we are done. However, how do we assess the accuracy, since in

general we will not be able to calculate the exact value for comparison (anyway why calculate it numerically if you already know the answer?).

However, if we subdivide the existing interval into two we would get a better estimate and we could approximate the error of the initial estimate using the better second estimate. So we divide the existing interval of length h into two intervals of length $\frac{h}{2}$ and our better estimate is:

$$I\left(a, a+\frac{h}{2}\right)+I\left(a+\frac{h}{2}, b\right)$$

The true error of our initial estimate is:

$$\int_a^b f(x)\,dx - I(a,b)$$

and we approximate it by:

$$E = I\left(a, a+\frac{h}{2}\right)+I\left(a+\frac{h}{2}, b\right) - I(a,b)$$

Furthermore if $E < \epsilon$, the required accuracy, we are done and we stop otherwise we continue. Suppose we must continue, then we need to assess the accuracy of $I\left(a, a+\frac{h}{2}\right)+I\left(a+\frac{h}{2}, b\right)$. In actual fact we are evaluating two problems i.e.:

$$\int_a^{a+\frac{h}{2}} f(x)\,dx$$

and

$$\int_{a+\frac{h}{2}}^b f(x)\,dx$$

one or both of which may be sufficiently accurate. Our original problem has been replaced by two similar problems. We repeat this problem subdivision approach until sufficient accuracy has been achieved. We must be careful regarding the accuracy requirement as we specified ϵ for the original problem, what should be specified for the subdivided problem? We could just use $\frac{\epsilon}{2}$, however this is not optimal but this shouldn't be a worry right now.

Programming Assignment No 3: Adaptive Quadrature.

This is an assignment. Implement a class which implements numerical integration using adaptive quadrature based on the trapezoidal rule. Using the mechanisms of inheritance the class should be able to used as a base class, whereby the derived class needs only to provide the function (method) to be integrated.

Chapter 12
ITERATIVE SOLUTION OF SYSTEMS OF LINEAR EQUATIONS

12.1 Introduction

One of the most common problems in many application areas involves solving a system of linear equations. In general terms we can write this problem as:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \vdots &= \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m \end{aligned} \tag{12.1.1}$$

This can be written in matrix notation as follows:

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

or in more compact notation:

$$\mathbf{Ax} = \mathbf{b} \tag{12.1.2}$$

There are many direct methods for solving such systems but here we will only be concerned with iterative methods which basically take an initial guess at the solution and then iterate (hopefully) towards the correct solution. These methods are easier to understand and implement. In the course of this chapter we will develop suitable vector and matrix objects with overloaded operators.

12.2 Gauss-Jacobi Iteration

For this method to work $\mathbf{A}$ must have **non-zero diagonal elements**. We split the matrix $\mathbf{A}$ into three matrices as follows:

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U} \tag{12.2.1}$$

where $\mathbf{L}$ is the lower triangular matrix (and zeros on the diagonal), $\mathbf{D}$ is the diagonal and $\mathbf{U}$ is the upper triangular matrix (with zeros on the diagonal). For example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -3 & 2 \\ 3 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & -3 & 0 \\ 0 & 0 & -1 \end{bmatrix} + \begin{bmatrix} 0 & 2 & 3 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \tag{12.2.2}$$

If we rewrite Eq. 12.1.2 taking account of Eq. 12.2.1 we get:

$$\begin{aligned} (\mathbf{L} + \mathbf{D} + \mathbf{U})\,\mathbf{x} &= \mathbf{b} \\ \mathbf{D}\mathbf{x} &= \mathbf{b} - (\mathbf{L} + \mathbf{U})\,\mathbf{x} \end{aligned} \tag{12.2.3}$$

because $\mathbf{D}$ has non-zero diagonal elements we can fins its inverse ($\mathbf{D}^{-1}$), remembering that $\mathbf{D}^{-1}\mathbf{D} = I$, the identity matrix:

$$\mathbf{x} = \mathbf{D}^{-1}\mathbf{b} - \mathbf{D}^{-1}\,(\mathbf{L} + \mathbf{U})\,\mathbf{x} \tag{12.2.4}$$

This equation is valid only if $\mathbf{x}$ is the solution. However if $\mathbf{x}$ is not the solution then we can write an iterative scheme which should converge towards the correct solution. Naturally there are theorems about when this convergence will occur etc., but we not deal with that here. Our scheme then becomes:

$$\mathbf{x}^{k+1} = \mathbf{D}^{-1}\mathbf{b} - \mathbf{D}^{-1}\,(\mathbf{L} + \mathbf{U})\,\mathbf{x}^{k} \tag{12.2.5}$$

where $\mathbf{x}^k$ is the present solution and $\mathbf{x}^{k+1}$ is a better solution.

From this scheme we see that we need to be able to multiply a matrix by vector (the result is a vector). We need to be able to add two matrices and we need to able to add and subtract vectors. We need to be able to invert a diagonal matrix. A diagonal matrix is easy to invert (check on Mathematica) if $\mathbf{D} = d_{ij}$, then $\mathbf{D}^{-1} = \frac{1}{d_{ij}}$. For example:

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & -3 & 0 \\ 0 & 0 & 6 \end{bmatrix}^{-1} = \begin{bmatrix} \frac{1}{2} & 0 & 0 \\ 0 & -\frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{6} \end{bmatrix} \tag{12.2.6}$$

Notice that the components of the equation $\mathbf{D}^{-1}\mathbf{b}$ and $\mathbf{D}^{-1}\,(\mathbf{L} + \mathbf{U})$ do not change during the calculation. They can be calculated once and then used from there on in without change. In component terms we can write for the ith component of $\mathbf{x}$:

$$x_i^{k+1} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^k}{a_{ii}} \tag{12.2.7}$$

A question that arises with all iterative methods is: how do I know when to stop iterating? A typical approach is to specify that the calculation should stop when the difference between $\mathbf{x}^{k+1}$ and $\mathbf{x}^k$ becomes small enough. This leads to the question: what is the difference between two vectors? First lets define the norm of a vector (of which there are several). The infinite norm of a vector $\mathbf{v}$ is written as $\|\mathbf{v}\|_\infty$ is the magnitude of the element of greatest absolute value in the vector. Our stopping condition occurs when $\left\|\mathbf{x}^{k+1} - \mathbf{x}^k\right\|_\infty < \epsilon$, where ϵ is a small value.

12.3 Gauss-Seidel Iteration

The Gauss-Jacobi method does not make full use of the available information. That is, as we calculate new values of $\mathbf{x}$ we continue to use the old values of $\mathbf{x}$ instead using the new values in our calculations as they become available. The Gauss-Seidel iteration scheme incorporates this observation and usually converges to a solution quicker than Gauss-Jacobi. Recall the componential form of the Gauss-Jacobi iteration:

$$x_i^{k+1} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^k}{a_{ii}} \tag{12.3.1}$$

it is clear that x_j^k for $j < i$ have already been updated, but these updated values are not being used. This is a wasteful approach and is corrected in the Gauss-Seidel iteration. Returning to our original problem and matrix splitting we have:

$$\begin{aligned} \mathbf{Ax} &= \mathbf{b} \\ (\mathbf{L} + \mathbf{D} + \mathbf{U})\,\mathbf{x} &= \mathbf{b} \\ \mathbf{Dx} = \mathbf{b} - \mathbf{Ux} & \quad -\mathbf{Lx} \\ \mathbf{x}^{k+1} &= \mathbf{D}^{-1}\left(\mathbf{b} - \mathbf{L}\mathbf{x}^{k+1} - \mathbf{U}\mathbf{x}^k\right) \end{aligned} \tag{12.3.2}$$

In componential terms this is:

$$x_i^{k+1} = \frac{b_i - \sum_{j=0}^{n-1} l_{ij} x_j^{k+1} - \sum_{j=0}^{n-1} u_{ij} x_j^k}{a_{ii}} \tag{12.3.3}$$

We can improve on this in componential terms by seeing that:

$$\sum_{j=0}^{n-1} l_{ij} x_j^{k+1} + \sum_{j=0}^{n-1} u_{ij} x_j^k = \sum_{j=0}^{i-1} a_{ij} x_j^{k+1} + \sum_{j=i+1}^{n-1} a_{ij} x_j^k \tag{12.3.4}$$

As an example suppose we have a 3 X 3 matrix as follows:

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} 1 & 2 & 3 \\ 2 & -3 & 2 \\ 3 & 1 & -1 \end{bmatrix} \\ \mathbf{L} &= \begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 1 & 0 \end{bmatrix} \\ \mathbf{D} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & -3 & 0 \\ 0 & 0 & -1 \end{bmatrix} \\ \mathbf{U} &= \begin{bmatrix} 0 & 2 & 3 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \end{aligned}$$

and $\mathbf{b}$ is:

$$\begin{bmatrix} 6 \\ 14 \\ -2 \end{bmatrix}$$

to being with take (any guess will do)

$$\mathbf{x}^k = \mathbf{x}^{k+1} = \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix}$$

then the LHS of Eq. 12.3.4 for $i = 0$ becomes:

$$\begin{aligned} \sum_{j=0}^{n-1} l_{ij} x_j^{k+1} &= l_{00}x_0^{k+1} + l_{01}x_1^{k+1} + l_{02}x_2^{k+1} = 0 \\ \sum_{j=0}^{n-1} u_{ij} x_j^{k+1} &= u_{00}x_0^{k+1} + u_{01}x_1^{k+1} + u_{02}x_2^{k+1} = 0 + 4 + 6 = 10 \end{aligned}$$

therefore

$$\sum_{j=0}^{n-1} l_{ij} x_j^{k+1} + \sum_{j=0}^{n-1} u_{ij} x_j^{k+1} = 10$$

Taking the RHS of Eq. 12.3.4:

$$\begin{aligned} \sum_{j=0}^{i-1} a_{ij} x_j^{k+1} &= 0, \text{ note } i - 1 = -1 \\ \sum_{j=i+1}^{n-1} a_{ij} x_j^{k} &= a_{01}x_1^k + a_{02}x_2^k = 4 + 6 \end{aligned}$$

therefore:

$$\sum_{j=0}^{i-1} a_{ij} x_j^{k+1} + \sum_{j=i+1}^{n-1} a_{ij} x_j^{k} = 10$$

After this iteration we have that:

$$\begin{aligned} \mathbf{x}^k &= \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \\ \mathbf{x}^{k+1} &= \begin{bmatrix} \frac{(6-10)}{1} = -4 \\ 2 \\ 2 \end{bmatrix} \end{aligned}$$

Calculating the next element ($i = 1$), then the LHS of Eq. 12.3.4 becomes:

$$\begin{aligned}\sum_{j=0}^{n-1} l_{ij}x_j^{k+1} &= l_{10}x_0^{k+1} + l_{11}x_1^{k+1} + l_{12}x_2^{k+1} = 2 \times -4 + 0 + 0 = -8 \\ \sum_{j=0}^{n-1} u_{ij}x_j^{k+1} &= u_{10}x_0^{k+1} + u_{11}x_1^{k+1} + u_{12}x_2^{k+1} = 0 + 0 + 4 = 4\end{aligned}$$

therefore

$$\sum_{j=0}^{n-1} l_{ij}x_j^{k+1} + \sum_{j=0}^{n-1} u_{ij}x_j^{k+1} = -4$$

Taking the RHS of Eq. 12.3.4:

$$\begin{aligned}\sum_{j=0}^{i-1} a_{ij}x_j^{k+1} &= a_{10}x_0^{k+1} = 2 \times -4 = -8, \text{ note } i - 1 = 0 \\ \sum_{j=i+1}^{n-1} a_{ij}x_j^{k+1} &= a_{12}x_2^{k+1} = 2 \times 2 = 4\end{aligned}$$

therefore:

$$\sum_{j=0}^{i-1} a_{ij}x_j^{k+1} + \sum_{j=i+1}^{n-1} a_{ij}x_j^{k} = -4$$

Notice that we get the same answer for much fewer calculations using the RHS of Eq. 12.3.4. This efficiency consideration should taken into account when deciding how to implement the algorithm (i.e. can overloaded operators adapt to this situation)? What happens for bigger matrices; is it worth trying to implement the more efficient approach?

12.4 Indexers

Some classes lend themselves to index notation. When we create an array (e.g. `int [] arr = new int[10]`) is it natural to access the elements using index notation (e.g. `arr[2] = 4`). It would be convenient to be able to access Matrix and Vector objects in a similar fashion e.g.

```
Matrix m = new Matrix(3,3);
m[1,1]=5; //assign a value
Console.WriteLine("Element [0,0] is {0}", m[0,0]); access a value
```

It is no surprise then that C# supports this behaviour using indexers. This is akin to operator overloading (and the property syntax) and involves implementing a method

to handle the details. The syntax is:

```
type this [type argument1, type argument2] {get; set;}
```

Suppose we have a matrix class as follows:

```
public class Matrix
{
private int rows;
private int cols;
private double [,] data;
public Matrix(int rows, int cols)
{
if(rows > 0)
this.rows = rows;
else
this.rows = 1;
if(cols > 0)
this.cols = cols;
else
this.cols = 1;
data = new double[rows, cols];
}
public Matrix(Matrix val)
{
rows = val.rows;
cols = val.cols;
data = new double[rows, cols];
for(int i = 0; i < rows;i++)
for(int j = 0; j < cols; j++)
data[i,j]=val.data[i,j];
}
public static Matrix operator+(Matrix a, Matrix b)
{
if(a.rows != b.rows || a.cols != b.cols)
return null;
Matrix tmp = new Matrix(a.rows,a.cols);
for(int i = 0; i < a.rows;i++)
for(int j = 0; j < a.cols; j++)
tmp.data[i,j]=a.data[i,j]+b.data[i,j];
return tmp;
```

```
}
public static Matrix operator-(Matrix a, Matrix b)
{
if(a.rows != b.rows || a.cols != b.cols)
return null;
Matrix tmp = new Matrix(a.rows,a.cols);
for(int i = 0; i < a.rows;i++)
for(int j = 0; j < a.cols; j++)
tmp.data[i,j]=a.data[i,j]-b.data[i,j];
return tmp;
}
public static Matrix operator+(Matrix a)
{
Matrix tmp = new Matrix(a);
return tmp;
}
public static Matrix operator-(Matrix a)
{
Matrix tmp = new Matrix(a.rows,a.cols);
for(int i = 0; i < a.rows;i++)
for(int j = 0; j < a.cols; j++)
tmp.data[i,j]=-a.data[i,j];
return tmp;
}
public void Output()
{
for(int i = 0; i < rows;i++)
{
for(int j = 0; j < cols; j++)
Console.Write("{0,4:F2}\t", data[i,j]);
Console.WriteLine();
}
}
public void setValue(int row, int col, double val)
{
if(!(row >= 0 && row < rows && col >= 0 && col < cols))
return;
data[row,col] = val;
}
public static explicit operator Matrix(double d)
{
Matrix tmp = new Matrix(3,3);
```

```
for(int i = 0; i < 3;i++)
for(int j = 0; j < 3; j++)
tmp.data[i,j]=d;
return tmp;
}
//indexer function
public double this[int rowindex, int colindex]
{
get
{
if(rowindex < rows && colindex < cols)
return data[rowindex, colindex];
else
return 0;
}
set
{
if(rowindex < rows && colindex < cols)
data[rowindex, colindex] = value;
//else do nothing
}
}
}
```

This enables us to get and set values of a matrix object in the following fashion:

```
Matrix m = new Matrix(3,3);
Random r = new Random();
for(int i = 0; i < 3; i++)
for(int j = 0; j < 3; j++)
m[i,j] = r.Next(0,10);
```

12.5 Measuring Time in C#

When writing numerical programs efficiency is an important consideration. If we can accurately measure the time an algorithm spends at different calculations we can usually direct improvements at the right location. Commonly the time spent at a single section of code can be extremely short. This problem can be fixed by repeating a code segment a million times and finding out the total time taken and subsequently dividing by one million.

C# provides the DateTime and TimeSpan structures for coping with time. Suppose we wish to find out how long it takes to initialise a 3 X 3 matrix as we did above. We modify our code as follows:

```
Matrix m = new Matrix(3,3);
Random r = new Random();
int k = 0;
DateTime start = DateTime.Now; //begin recording
while(k < 1000000)
{
for(int i = 0; i < 3; i++)
for(int j = 0; j < 3; j++)
m[i,j] = r.Next(0,10);
k++
}
DateTime end = DateTime.Now; //end recording
TimeSpan diff = end - start;
Double totaltime = diff.TotalSeconds;
Double perinitialisation = totaltime/1000000;
```

This is fairly self explanatory.

Programming Assignment No. 4: Solving systems of linear equations

Implement Matrix and Vector classes (or you could just the implementation from a previous assignment!!) such that they can support the operations required for Gauss-Jacobi and Gauss-Seidel iterations. Create a class which implements both the Gauss-Jacobi and Gauss-Seidel algorithms. Using timing to estimate the relative efficiency of the methods.

Chapter 13
GENERICS

13.1 Introduction

Collections are data structures that are primarily designed to aid the processing of groups of similar objects. We will consider a few of the collections that come for free with C# in a later Chapter. For example, an array is probably the simplest form of collection available in C#. In previous versions of C# more advanced collections (see later on) allowed the processing of different objects in the same collection (it did this by taking advantage of the fact that all objects in C# have a shared ancestor of type `object`). You could add strings, integers, vectors etc. into a single collection. This had several negative consequences. Firstly, there was a performance impact due to boxing (assigning an element and the implicit or explicit associated conversion) and unboxing types (retrieving an element and then casting to the actual type). Secondly, runtime errors are common because the compiler can not know in advance exactly what type was stored in a collection and hence check that a users actions were valid. Generics is an improved mechanism whereby the programmer can specify eactly which data type is to be used in a collection. this eliminates the above two concerns at a stroke. Beneficial aspects of this approach include reuse of code (same code for different types), type safety (compiler knows whats going on) and there is no code bloat, that is C# does not create a whole new ream of code to cope with new generic types. Generics is very much akin to the tried and tested template mechanism of C++. Old style collections are still part of the C# language but generics is the preferred solution now.

13.2 The Stack example

13.2.1 Introduction

One of the most simple data constructs used in programming is the stack. A stack is like a tower of stuck together of lego bricks. There are two permissable operations: 1) put a brick onto the top of the tower (called pushing) or 2) take the top bick off the top of the tower (called popping). Essentially it is a last in - first out (LIFO) queue. Quite clearly it does not matter what the bricks represent (i.e. int, float, Person, Complex, Vector, Matrix etc.), the same operations are carried out. Consider the following "object" based implementation (i.e. given that object is the ultimate

ancestor of all objects in C# then by inheritance we can refer to any object, including the basic data-types, using an object reference):

```
public class Stack
{
readonly int m_Size;
int m_StackPointer = 0;
object[] m_Items;
public Stack():this(100)
{}
public Stack(int size)
{
m_Size = size;
m_Items = new object[m_Size];
}
public void Push(object item)
{
if(m_StackPointer >= m_Size)
throw new StackOverflowException();
m_Items[m_StackPointer] = item;
m_StackPointer++;
}
public object Pop()
{
m_StackPointer--;
if(m_StackPointer >= 0)
{
return m_Items[m_StackPointer];
}
else
{
m_StackPointer = 0;
throw new InvalidOperationException("Cannot pop an empty stack");
}
}
}
```

Now this stack can be used as follows:

```
Stack stack = new Stack();
```

```
stack.Push("1");
string number = (string)stack.Pop();
```

Ths first problem is that with value types (int, float etc.) you have to box them (i.e. convert to a reference type) before pushing them onto the stack object and then unbox them when popping them off the stack. This affects performance. Secondly you loose type-safety as anything can be cast to an object therefore the compiler cannot know in advance what behaviour is possible. For example:

```
Stack stack = new Stack();
stack.Push(1);
string number = (string)stack.Pop();
```

this code compiles but will fail and throw an exception at run-time. One solution to these problems is to write a specific stack for each data type i.e. Intstack, Floatstack. This works but it is tedious, repetitious and error-prone. Also if you discover a fundamental bug in your approach each implementation needs to be individually fixed. Generics to the rescue!

13.2.2 The Generic Stack

Generics solves all of the above problems at a stroke. The code is written once using a generic type parameter (enclosed in angled brackets < >). When an object is declared you provide the actual type to be used. Here is the stack example using generics:

```
public class Stack<T>
{
readonly int m_Size;
int m_StackPointer = 0;
T[] m_Items;
public Stack():this(100)
{}
public Stack(int size)
{
m_Size = size;
m_Items = new T[m_Size];
}
public void Push(T item)
{
if(m_StackPointer >= m_Size)
throw new StackOverflowException();
```

```
m_Items[m_StackPointer] = item;
m_StackPointer++;
}
public T Pop()
{
m_StackPointer--;
if(m_StackPointer >= 0)
{
return m_Items[m_StackPointer];
}
else
{
m_StackPointer = 0;
throw new InvalidOperationException("Cannot pop an empty stack");
}
}
}
```

An example of using the generic Stack class follows:

```
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
int number = stack.Pop();
```

and the compiler will now complain if you try to push something other than an int onto the stack e.g. `stack.Push("1")`. This is the type safety aspect of generics, the compiler can help track errors much earlier in the process (i.e. before run-time).

In this example above, `T` is called the **generic type parameter**, `Stack<T>` is called the **generic type** and the `int` in `Stack<int>` is called the **type argument**. A clear advantage of this mechanism is that the form of the algorithm and data manipulation remain the same even though the actual data type can change.

13.2.3 How do they do that!

On the face of it, generics is identical to the concept of templates in standard C++ and other languages. However there are some important differences. Templates have some disadvantages, for example, some compilers make a copy of the code with specific data type code generated for each data type. The developer does not have to write the code to support the extra data types but the compiler does write it. This leads to code bloat and an increased usage of memory by the executable.

Referring back to section 1.2, remember that in .NET the compiler creates

MSIL (Microsoft Intermediate Language or just plain IL) and that it is only when the code is to be run on a target oeprating system that the JIT (Just In Time) compiler converts to machine readable code. Generics code when compiled to IL only contains parameters (i.e. place holders) for the actual data type. The actual IDE compiler can however enforce type-safety as we have seen by clever programming. So in other words there is no code bloat in the step to IL.

The result of the step from IL to machine code using the JIT depends on whether the type argument is a value or reference type. For value types the JIT compiler creates a separate version of the code for each value-type (some code bloat). However it keeps track of which value types it has created code for already and does not duplicate code production. In the case of reference types, the JIT compiler replaces the generic type parameter with Object and produces machine code once and thus leads to no code bloat.

13.3 Features of Generics

13.3.1 `default()`

Sometimes it is useful to be able to access the default value of the type parameter (for example the default value of `int` is 0 and that of a reference type is `null`). To support this in generics C# provides the `default()` operator. So in our stack example `default(T)` gives the default value of whatever `T` represents at run-time.

13.3.2 Constraints

Sometimes it makes sense when creating a generic class that there are constraints as to what form the type parameter takes. Constraints allow such restrictions to be put in place. The syntax is as follows:

```
where generic_type_parameter :  constraint_type
```

For example:

```
public class Stack<T>
where T: struct
{
//code here
}
```

There are six constraint types. 1) `struct` means type T must be a value type, 2) `class` means type T must be a reference type, 3) `IFoo` means type T is required to implement the `IFoo` interface, 4) `Foo` means type is required to be derived from base class `Foo`. 5) `new()` specifies that a default costructor must be implemented by

type T and finally 6) `T2` specifies that type T must be derived from a generic type `T2`, this is also called a naked type constraint. Multiple constraints can be combined by separating them with a comma.

13.3.3 Inheritance

A generic child class can be derived from a generic base class.The only requirement is that the generic type argument is repeated or else the base class generic type argument is given a specific type argument e.g.

```
public class myBase<T>
{
}

public class myChild <T> :  myBase<T>
{//specify the same generic type argument
}

//otherwise
public class myChild<T> :  myBase<string>
{//a specific type argument is given for the base class
}
```

13.3.4 Static data members

Some care must be exercised here as there is a different static member associated with each type argument. For example:

```
public class myBase<T>
{
public static int x;
}

myBase<string>.x = 5; //one static x
myBase<int>.x = 4; // a different static x
Console.WriteLine("x  = {0}", myBase<string>.x);
//outputs 5!!
```

13.3.5 Generic Interfaces

In the chapter on interfaces we saw how to implement the `IComparable` interface in order to use the `Array.Sort()` method. In that example we could also use the generic version of the interface as follows:

```
//interface definition
public interface IComparable<T>
{
int CompareTo(T other);
}

//using the interface
class Person :IComparable<Person>
{
private string firstname;
public string Firstname
{ get { return firstname; } set { firstname = value; } }
public int CompareTo(Person other)
{
//just use the normal string comparison functionality
return this.firstname.CompareTo(other.firstname);
}
}
```

It is worth noting that we can use generic interfaces as constraints on generic classes.

13.3.6 Generic Methods

These are methods where a generic type is included in the signature. For example, it woud make sense if we were writing a swap method. The logic of the method is independent of the data type. Therefore:

```
void Swap<T>(ref T x, ref T y)
{
T temp;
temp = x;
x = y;
y = temp;
}

//call the generic method
int i = 4, j = 5;
Swap<int>(ref i, ref j);
```

Now the compiler can work out the type parameter from the method arguments therefore it could also be called simply as:

```
Swap(ref i, ref j);
```

this is referred to as type inference. In the case of a generic method with no arguments then the compiler cannot figure out the type parameter automatically and it has to be explicitly specified.

13.3.7 Static Methods

Generic static methods can be defined if required, however the type parameter must be provided.

```
public class myClass<T>
{
public static T SomeMethod(T t)
{...}
}
int i = MyClass<int>.SomeMethod(3);
```

You can of course make a static method generic in its own right.

```
public class myClass<T>
{
public static T SomeMethod<X>(T t, X x)
{...}
}
int i = MyClass<int>.SomeMethod<string>(3,"KM");
//or using type inference
int i = MyClass<int>.SomeMethod(3,"KM");
```

Chapter 14
DELEGATES AND EVENTS

14.1 Introduction

Delegates are a new reference data type which allow us to reference methods of a particular signature and return type (all type safe). In line with other reference types, creating a delegate type does nothing, it is not until we assign it to a new delegate object that it becomes useful. Delegates have many uses and enable the passing of methods between objects and methods. For example suppose I wanted to implement a class which can sort a pair of objects. Obviously each pair of objects may sort differently i.e. a pair of ints, as opposed to a pair of strings. One workable but lengthy solution is to create code specific to each object type. With delegates we can pass the ordering responsibility to the objects and the pair class can implement the sort machinery as we will see in our example below.

14.2 Simple Delegates

Just like classes and objects, delegates need to be defined before we can assign an instance of the delegate to it. The syntax for declaring a delegate type is:

```
[access_modifier] delegate return_type identifier ([type param1, type param2,...]);
```

A good way to think about this is that the delegate is giving a name to a particular method signature. This is written as part of the class declaration and they are typically public. In our pair class example:

```
public class Pair
{
//an array of two objects
private object[] data = new object[2];
//the delegate declaration
public delegate bool correctOrder(object o1, object o2);
//constructor
public Pair(object o1, object o2)
{
```

```
data[0]=o1;
data[1]=o2;
}
//this method is expecting to be passed a reference
//to the method which takes care of ordering
public void Sort(correctOrder orderMethod)
{
//if they are not in the correct order then reorder
if(!orderMethod(data[0], data[1]))
{
object tmp = data[0];
data[0] = data[1];
data[1] = tmp;
}
}
public void outPut()
{
Console.WriteLine("{0} and {1}", o1.ToString(), o2.ToString());
}
}
```

Here correctOrder is a delegate type which can reference methods returning a bool (true if they are in the correct order and false otherwise) and taking two objects as parameters. In the Sort method orderMethod is a delegate, i.e. a reference to a method (returning a bool and taking two object parameters), which can be invoked as shown. Next we create a Student class which must implement a suitable ordering method if it is to work with the Pair class.

```
public class Student
{
private string name;
public string Name
{
get { return name; }
set { name = value; }
}
public Student(string name)
{
this.name = name;
}
//this method conforms to our delegate signature
static public bool order(object o1, object o2)
```

```
{
Student s1 = (Student) o1;
Student s2 = (Student) o2;
if(string.Compare(s1,s2) > 0)
return true;
else
return false;
}
public override string ToString()
{
return name;
}
}
```

Finally we write some code to test our arrangement:

```
Student kieran = new Student("Kieran");
Student john = new Student("John");
Pair pairStuds = new Pair(kieran, john);
//create delegate for the student ordering method
Pair.correctOrder c = new Pair.correctOrder(Student.order);
//called the Pair sort method
pairStuds.Sort(c);
//check sorting
Pair.outPut();
```

Delegates consist of (1) a definition (2) some method conforming to the delegate signature (3) an instantiation of a delegate based on a suitable method.

In the above code we needed to create the delegate in the code as follows:

```
//create delegate for the student ordering method
Pair.correctOrder c = new Pair.correctOrder(Student.order);
//called the Pair sort method
students.Sort(c);
```

However, it would be easier if the student class had a pre-prepared, ready to use delegate for us:

```
public class Student
{
//create a ready to go delegate
```

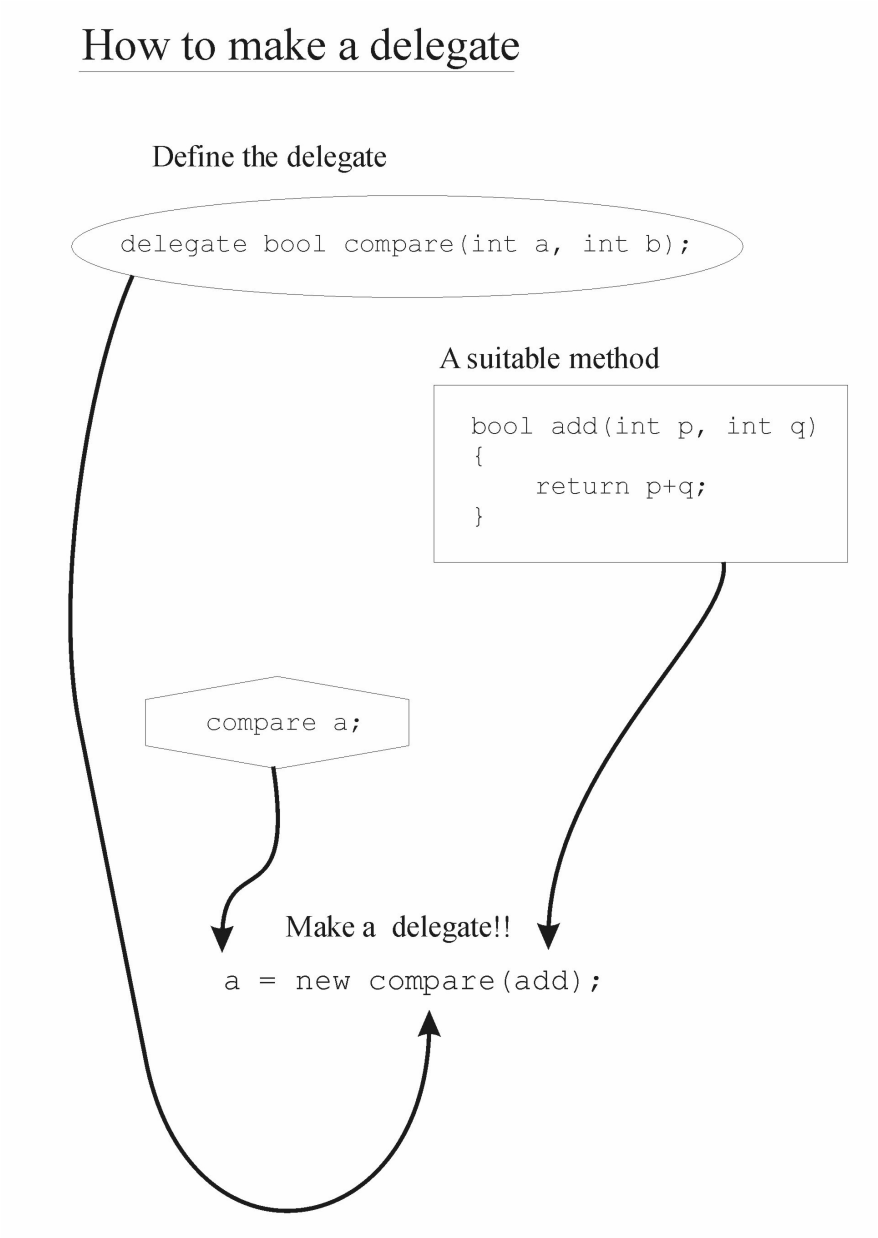

Figure 14.2.1 Making delegates

```
public static Pair.correctOrder orderStudents = new Pair.correctOrder(Student.order);
private string name;
public string Name
{
get { return name; }
set { name = value; }
}
public Student(string name)
{
this.name = name;
}
//this method conforms to our delegate signature
static public bool order(object o1, object o2)
{
Student s1 = (Student) o1;
Student s2 = (Student) o2;
if(string.Compare(s1,s2) > 0)
return true;
else
return false;
}
public override string ToString()
{
return name;
}
}
```

We could then simply call the Sort of Pair as follows (using the pre-prepared delegate):

```
pairStuds.Sort(Student.orderStudents);
```

On the other hand it might be a bit wasteful to always have a delegate created for the Student class when it may not even be used. We can fix this by making the delegate a property (C# style) by replacing:

```
public static Pair.correctOrder orderStudents = new Pair.correctOrder(order);
```

with

```
public static Pair.correctOrder orderStudents
{
```

```
get { return new Pair.correctOrder(order); }
}
```

Now, it is only when a delegate is required that one is created.

Lets look re-examine solving non-linear equations in this context and implement only the bisection method.

```
class NonLinear
{
//should be properties
private double left = 0;
private double right = 1;
private double tol = 0.001;
//declare delegate
public delegate double integrate_func(double x);
public double Solve(integrate_func func)
{
double mid, l, r;
l = left;
r = right;
if (func(l) * func(r) > 0)
//throw exception as bad initial condition
return 0;
while (r - l > tol)
{
mid = (l + r) / 2;
if (func(l) * func(mid) < 0)
r = mid;
else
l = mid;
}
return (l + r) / 2;
}
}
class SoilFunction
{
//ready to use delegate
public NonLinear.integrate_func theFunc
{
//created only when required
get { return new NonLinear.integrate_func(complexFunction); }
}
```

```
public double complexFunction(double x)
{
//solution 0.92459
return x * x * x - Math.Sqrt(Math.Sin(x * x) * Math.Sin(x * x) + 1) + 0.5
* x;
}
}
class EasyFunction
{
//ready to use delegate
public NonLinear.integrate_func theFunc
{
//created only when required
get { return new NonLinear.integrate_func(easyFunction); }
}
public double easyFunction(double x)
{
//solution 0.105573
return x * x - 2 * x + .2;
}
}
```

Here is the associated test code:

```
NonLinear n = new NonLinear();
SoilFunction s = new SoilFunction();
EasyFunction e = new EasyFunction();
Console.WriteLine("The solution is {0}", n.Solve(s.theFunc));
Console.WriteLine("The solution is {0}", n.Solve(e.theFunc));
```

14.3 Multicast Delegates

Sometimes it is useful to be able to invoke a number of methods at the same time. For example, in Graphical User Interfaces, a variety of programs may be interested when the user clicks the mouse. Notification is usually handled by events which amounts to invoking a collection of methods. In C# multicasting is performed using delegates. Up until now each delegate reference has referred to a single delegate instance. However it is possible to refer to more than one delegate instance with a single delegate reference. Invoking the delegate reference then in turn invokes all the methods to which it refers, hence the name multicasting. This is especially useful for

the event construct discussed in the next section.

The syntax for achieving this functionality is quite simple. Suppose we have a delegate reference:

```
Pair.correctOrder orderStudents = new Pair.correctOrder(order);
```

Subsequently add additional delegates using += operator as follows:

```
orderStudents += new Pair.correctOrder(anotherorder);
```

where `anotherorder` is another method conforming to the delegate signature. Then by invoking orderStudents:

```
orderStudent(a,b);
```

will invoke both order and anotherorder with the same arguments.

As an example suppose we have an cat object which occasionally meows and that there are a few dog objects which bark in response to this stimulus. Using multicasting then we have:

```
namespace Delegate
{
public delegate void MakeNoise(string hint);
public class Cat
{
private string name;
//declare delegate
private MakeNoise m;
public void add(MakeNoise p)
{
m+=p;
}
public void remove(MakeNoise p)
{
m-=p;
}
public Cat(string name)
{
this.name = name;
}
public void Meaow()
{
```

```
Console.WriteLine("{0} says MEAOW!!", name);
//invoke anyone listening
m(name);
}
}
public class Dog
{
private string name;
public Dog(string name)
{
this.name = name;
}
public void Bark(string hint)
{
Console.WriteLine("{0} goes WOOF WOOF because of {1}", name, hint);
}
public MakeNoise getDelegate()
{
return new MakeNoise(this.Bark);
}
}
}
```

Some code to test the classes:

```
Cat c = new Cat("Freddie");
Dog d1 = new Dog("Barney");
Dog d2 = new Dog("Spot");
MakeNoise tmp;
c.add( tmp = d1.getDelegate() );
c.add( d2.getDelegate() );
c.Meaow();
c.Meaow();
c.remove(tmp);
c.Meaow();
c.add( d1.getDelegate() );
c.Meaow();
Console.Read();
```

Can you make sense of the output??

14.4 Delegates and Numerical Integration

In an earlier assignment you were asked to implement adaptive quadrature in a base class such that derived classes need only provide a function to be integrated. Delegates are suitable for this purpose as well (probably better suited). Can you modify your previous assignment to produce a generic numerical integration class, whereby the user of the object just needs to provide a reference to the function to be integrated?

14.5 Events

14.5.1 Introduction

Event style programming has become prevalent along with the upsurge in GUI operating systems. A user clicks a button or selects a menu option and something happens. This is implemented using the event model. Events are raised either by user actions or the operating system and when an event occurs code is executed in response to it. The system does nothing until an event occurs. Events can be triggered in any order. There is no limit to the number of objects which may be interested in the occurrence of a particular event.

Events are supported as part of the language in C#, whereas in C and C++ this was supported by messages and a big switch statement to decide what to do when a particular message was received. C# uses the publish and subscribe model for events. A class can publish a list of events that it may raise. Other classes then subscribe to one or more events. When an event occurs then all subscribing classes are informed about the occurrence. In simpler terms the publishing class says "here are the events that I make happen" and the subscribing class says "Ok, let me know when event X occurs". Recall the multicasting ability of delegates, so it is no surprise that delegates enable events to be implemented.

14.5.2 An Example with Delegates

In this section we will implement a simple example using only delegates. Suppose that as part of complex queuing model we have waiting area object which generates an event whenever someone (i.e. a customer object) arrives. Suppose that there is log object which is interested in these things and wants to record the time each customer arrives in the waiting area. For the sake of adding some level complexity to the example suppose that there is a receptionist object who is also interesed in this information. Let's use delegates to implement an event mechanism. Here is the solution with some guiding comments.

```
      class WaitingArea
{//Publishes the event i.e.  this event is available
//define the delegate
public delegate void ArrivalEvent(object o, EventArgs e);
//create delegate instance
public ArrivalEvent avh;
//cause an event to happen
```

```
public void Arrive()
{
EventArgs t = new EventArgs();
//create event
if(avh != null) //if someone is listening
avh(this, t);
}
}
class Log
{//subscribes to the event
public void subscribe(WaitingArea w)
{
//hook up the waiting area delegate and the event code here
//it makes sense to work this way as the current object knows
//the method to be called in repsonse to the event
w.avh += new WaitingArea.ArrivalEvent(LogArrivalEvent);
//alternative syntax
w.avh += LogArrivalEvent;
}
public void LogArrivalEvent(object o, EventArgs a)
{
Console.WriteLine("Heard Event in Log Object");
}
}
class Receptionist
{//subscribes to the event
public void subscribe(WaitingArea w)
{
//hook up the waiting area delegate and the event code here
w.avh += new WaitingArea.ArrivalEvent(RecepArrivalEvent);
//alternative syntax
w.avh += RecepArrivalEvent;
}
public void RecepArrivalEvent(object o, EventArgs a)
{
Console.WriteLine("Heard Event in Receptionist Object");
}
}
```

Glue code:

```
WaitingArea p = new WaitingArea();
```

```
Receptionist r = new Receptionist();
Log l = new Log();
r.subscribe(p);
l.subscribe(p);
p.Arrive();
p.Arrive();
```

As you can see events really are multicast delegates in disguise. Anyhow, C# recognises that events are used in a different context to the normal use of delegates and therefore provides a distinct implementation. Also there is a potential problem here. What if the log object above put the line:

```
w.avh = new WaitingArea.ArrivalEventHandler(ArrivalEvent);
```

into its subscription method? This would essentially delete any other multicast references and as you may appreciate this is a very simple error to make. Another problem with this approach is that I can make an event happen as follows:

```
//make an event directly
EventArgs t = new EventArgs();
//create event
p.avh(p, t);
```

Usually the designer of the class would like to control directly where and when events occur. Any behaviour (which the programmer is free to implement) may lead to erroneous answers. One could make the delegate private but then you can't wire it up to listen objects. What's needed is special kind of delegate which can be subscribed/unsubscribed to but not invoked.

14.5.3 The `event` Keyword

The problems identified in the previous section are solved by the `event` keyword. We only need to change one line:

```
//delegate instance
public ArrivalEvent avh;
```

becomes

```
//delegate instance
public event ArrivalEvent avh;
```

This ensures that Arrival Event cannot be directly called and can only be subscribed to using += and unsubcribed using -=. Make the change in the above example and

see what the compiler outputs.

14.5.4 Making Events Happen

There are a number of steps required to implement events:

1. Publisher Responsibilities
   (a) Create a suitable delegate definition (there is a convention for what these should look like).
   (b) Define the event the publisher will raise.
   (c) Make events happen.
2. Subscriber responsibilities
   (a) Create a method which is to be invoked when an event occurs (this must conform to the delegate specification).
   (b) Connect this method to the publisher event i.e. the actual subscription.

The delegate used in event-style programming is by convention a method which returns void and takes a C# object as its first parameter and an EventArgs object (or derivative) as its second parameter. The first argument is meant to the be a reference to the object creating the event and the second argument is for passing any relevant data. This is such a common process that the .NET framework has a suitable delegate defined in the System namespace, it is called EventHandler and is defined as follows:

```
public delegate void EventHandler(Object sender, EventArgs e);
```

Typically we need to forward specific information with the event (i.e. mouse position or anything you care to think of!), to this end, deriving a child class from EventArgs enables the passing of any extra information required. There is nothing special about the EventHandler delegate and we could just as easily create our own delegate with any set of arguments (although we would be flying in the face of convention).

Consider the following example:

```
// the publisher class
public class Metronome
{
//(1) create the delegate definition
public delegate void TickHandler(Metronome m, EventArgs e);
//(2) define the event
public event TickHandler Tick;
public EventArgs e = null;
```

```
public void Start()
{
while (true)
{
System.Threading.Thread.Sleep(3000);
if (Tick != null)
{
//(3) make events happen
Tick(this, e);
}
}
}
}
//the subscriber class
public class Listener
{
//(1) define a suitable method
private void HeardIt(Metronome m, EventArgs e)
{
System.Console.WriteLine("HEARD IT");
}
//(2) enable the subscription
public void Subscribe(Metronome m)
{
m.Tick += new Metronome.TickHandler(HeardIt);
//alternative syntax
//m.Tick += HeardIt;
}
}

//glue code
Metronome m = new Metronome();
Listener l = new Listener();
l.Subscribe(m);
m.Start();
```

Here is the same example except there is extra information being passed to the event using a class derived from EventArgs:

```
//modified EventArgs for extra information
public class TimeOfTick :  EventArgs
```

```
{
private DateTime TimeNow;
public DateTime Time
{
set { TimeNow = value; }
get { return this.TimeNow; }
}
}
// the publisher class
public class Metronome
{
//(1) create the delegate definition
public delegate void TickHandler(Metronome m, TimeOfTick e);
//(2) define the event
public event TickHandler Tick;
public void Start()
{
while (true)
{
System.Threading.Thread.Sleep(3000);
if (Tick != null)
{
//create specific information
TimeOfTick TOT = new TimeOfTick();
TOT.Time = DateTime.Now;
//(3) make events happen
Tick(this, TOT);
}
}
}
}
//the subscriber class
public class Listener
{
//(1) define a suitable method
private void HeardIt(Metronome m, TimeOfTick e)
{
System.Console.WriteLine("HEARD IT AT {0}", e.Time);
}
//(2) enable the subscription
public void Subscribe(Metronome m)
{
```

```
m.Tick += new Metronome.TickHandler(HeardIt);
//alternative syntax
//m.Tick += HeardIt;
}
}

//glue code
Metronome m = new Metronome();
Listener l = new Listener();
l.Subscribe(m);
m.Start();
```

14.6 An Event Example - The Queue Model

Can you develop the queue model using the event mechanism?

14.7 Anonymous Methods and Lambda Expressions

14.7.1 Anonymous Methods

These are a fairly new feature of the language. Recall that making use of delegates involves (1) defining the delegate (2) creating a suitable method and (3) create a delegate instance. For example:

```
//define delegate
public delete void myDelegate(int x);
//a suitable method
public void Output(int n)
{
    for(int i = 0; i < n; i++)
        Console.WriteLine("Out");
}
//make a delegate instance
myDelegate p = new Delegate(Output);
//use the delegate or maybe pass it around
p(10);
```

Thus up until now, a method needs to be created before using a delegate. Anonymous methods (i.e. methods without a name) can be used to create a suitable method and a delegate instance all in the one go. For example:

```
//still define a delegate
int y = 10;
public delete void myDelegate(int x);
myDelegate p = delegate(int x)
{
```

```
            for(int i = 0; i < x; i++)
                Console.WriteLine("Out, I can see y it has value {0}",y);
        }
```

The anonymous method can only be called using the delegate i.e. it doesn't exist in the same way as other methods. You can access variables in local scope inside the anonymous methods.

Typically anonymous methods are used with events. The idea is that you don't have to go through the rigmarole of creating a special method and thereby save on the amount of code to be written. However, if the code is useful in other contexts then you should create a normal method. In the metronome example earlier we could code the subscriber class as follows

```
//the subscriber class
public class Listener
{
public void Subscribe(Metronome m)
{
m.Tick += delegate(Metronome m, TimeOfTick e)
          {
              System.Console.WriteLine("HEARD IT AT {0}", e.Time);
}
      }
}
```

14.7.2 Lambda Expressions

Lambda expressions extend the concept of anonymous methods and are more powerful and flexible. The syntax is:

```
(input parameters) => {expression or statement block};
```

The => operator is the lambda operator and should be read as "goes to". The return value of a lambda expression is a method. The return type and input parameters are those of the delegate.

```
       //define delegate
       public delegate double doOp(double x);
       static void Main(string[] args)
       {
       //create delegate instance and suitable method
      //long verion
      doOp d = (double val) => { val * 2};
      //short version
       doOp d = val => val * 2;
       //create delegate instance and suitable method
       doOp c = val => val * val;
```

```
Console.WriteLine(d(2));
Console.WriteLine(d(4));
Console.WriteLine(c(2));
Console.WriteLine(c(4));
}
```

The short version can aboce can be used when there is one parameter and of course the compiler can figure out the type from the delegate. The curly brackets can be removed if there is only one statement in the method body.

14.8 Generic Delegates and Events

It should come as no surprise that generic delegates and events are supported by C#, if required.

Chapter 15
GENERIC COLLECTION CLASSES

15.1 Introduction

Lets firm up on Generics by taking a look at a few Generic Collection Classes which implement some fundamental data structures. These classes are invaluable and can save a huge amount of coding. There also exists a set of non-generic collection classes (from the time before generics), however, the generic versions are the preferred solution at present.

15.2 List<T>

We will use the following class as an example:

```
class Person :IComparable<Person>, IFormattable
{
private string firstname;
public string Firstname
{ get { return firstname; } set { firstname = value; } }
private string lastname;
public string Lastname
{ get { return lastname; } set { lastname = value; } }
public int CompareTo(Person obj)
{
int compare = this.lastname.CompareTo(obj.lastname);
if(compare == 0) //lastnames equal
return this.firstname.CompareTo(obj.firstname);
return compare;
}
public string ToString(string format, IFormatProvider fp)
{//this is satisfy the IFormattable interface
if (format == null)
return ToString();
string fu = format.ToLower();
```

```
switch (fu)
{
case "g":
return string.Format("{0}, {1}", firstname, lastname);
case "f":
return string.Format("{0}", firstname);
case "l":
return string.Format("{0}", lastname);
default:
return string.Format("{0}, {1}", firstname, lastname);
}
}
}
```

You may want to look the IFormattable interface to understand the ToString() implementation.

List<T> is the equivalent of the non-generic ArrayList class. Using a list offers much more flexibility than a straight array because 1) its size can change dynamically, 2) you add, insert and remove elements (all non-trivial in an array), 3) you can search and sort a list. However, it should be remembered that the hidden implementation of List<T> is simply an encapsulation of an array of type T (i.e. T [] _items). This has performance implications. For example, appending items to the end of an array is fast provided a resize is not required whereas inserting an element into the middle of an array gets slower and slower as the number of item in the array increases.

15.2.1 Creating a List<T>

Lists are created as normal and the default constructor creates an empty list however you can also specify the initial size.

```
List<person> pList = new List<person>();//default
List<int> iList = new List<int>(20);//specify size
```

As soon as you add an element to the list created with default constructor space is allocated for 4 elements. If a fifth element is needed then the capacity doubles to 8. In general the capacity doubles for every resize. The internal mechanism is as you'd expect: a new array is created of the new size and the elements of the old array are copied over using Array.Copy(). The capacity of a list is a get/set property, e.g. pList.Capacity, whereas the Size property indicates how may elements are in the List<T> object.

A List<T> object can be initialised with a list of objects just like the sysntax for an array:

```
List<int> iList = new List<int>(20) {1,2,3};
```

15.2.2 Adding, Inserting, Accessing and Removing Elements

The Add() method allows elements to be added on to the end of the List<T> object. AddRange() enables you to add several objects in the one go.

```
List<Person> pList = new List<Person>();
pList.Add(new Person("Kieran", "Mulchrone"));
pList.Add(new Person("Jimmy", "Jazz"));
pList.AddRange(new Person[] { new Person("J", "Iscariot"), new Person("F",
"Trump") });
```

Elements may be accessed using an index from 0 to the Count of elements less one. Note that List<T> implements IEnumerable so it can be used in a foreach loop.

```
int i;
for (i = 0; i < pList.Count; i++)
//note f means print only the firstname
Console.WriteLine("{0:f}", pList[i]);
foreach (Person p in pList)
//note l means print only the lastname
Console.WriteLine("{0:l}", p);
```

Elements can be inserted into a List<T> using the insert method which takes as its first argument the position where the insert is to occur and as its second argument the object to be inserted (which must be of type T). For example:

```
pList.Sort();
//to maintain order where would you put JF Kennedy?
Person tmp = new Person("JF","Kennedy");
//BinarySearch only works when list is sorted!
int i = pList.BinarySearch(tmp);
if(i < 0) //doesnt exist
i=~i; //gives position where it should be
pList.Insert(i, tmp);
```

Elements can be removed either by index (fast) or by object (slower as it must be found first). Following on from the last example:

```
p.RemoveAt(0); //index
p.Remove(tmp); //object
```

15.2.3 Sorting and Searching

Sorting is performed by the Sort method which has a number of overloads. Calling Sort() without any arguments requires that the elements in the List<T> implement the IComparable interface (as does the Person class).

Another overload expects an object that implements the IComparer<T> interface. For example, create such an object for the Person class:

```
class PersonComparer :  IComparer<Person>
{
public enum CompareType
{
Firstname,
Lastname
}
private CompareType comptype;
public CompareType CompType
{
get { return comptype;}
set{comptype = value;}
}
public PersonComparer(CompareType ct)
{
comptype = ct;
}
public int Compare(Person p1, Person p2)
{
switch (comptype)
{
case CompareType.Firstname:
return p1.Firstname.CompareTo(p2.Firstname);
case CompareType.Lastname:
return p1.Lastname.CompareTo(p2.Lastname);
}
}
}
```

The use it to do the comparison:

```
//sort by firstname
pList.Sort(new PersonComparer(PersonComparer.CompareType.Firstname));
//Sort by lastname
```

```
pList.Sort(new PersonComparer(PersonComparer.CompareType.Lastname));
```

It is also possible to sort only a range of the data using the IComparer<T> implementing object i.e.:

```
//Sort a subrange
//start at position 1, for the next 2 elements
pList.Sort(1, 2, new PersonComparer(PersonComparer.CompareType.Firstname));
```

Finally there is a generic delegate with Comparison<T> which has the definition:

```
public delegate int Comparison<T>(T x1, T x2);
```

It is also possible to sort by using the delegate Comparison<T>, which takes two objects of type T as arguments and returns an integer value. If the arguments are equal then return zero if the first is bigger than the second return a positive number, otherwise return a negative number (this is all much like the situation for IComparer). An anonymous delegate is probably most handy:

```
List<person> lp = new List<person>();
lp.Add(new person("P", "Tomlinson"));
Person p = new Person("K", "Mulchrone");
lp.Add(p);
lp.Add(new Person("P", "Mulchrone"));
lp.Add(new Person("M", "Mulchrone"));
lp.Add(new Person("J", "Edward"));
lp.Add(new Person("A", "Farnley"));
lp.Sort(
delegate(person a, person b)
{
    //sort by Firstname then Lastname
    int i = a.Firstname.CompareTo(b.Firstname);
    if(i == 0)
        return a.Lastname.CompareTo(b.Lastname);
    else
        return i;
}
);
```

Of course we could have used a lambda expression as follow:

```
lp.Sort((a,b) => a.Lastname.CompareTo(b.Lastname));
```

which compares by lastname only.

There are six methods available for searching, four which retrieve indices and two which retrieve actual objects. IndexOf() takes an argument of type T and returns

the index of the object sought in the List<T>. If the object is not found it returns -1. For example:

```
int ind = lp.IndexOf(p);
```

LastIndexOf() operates in much the same way but returns the index of the last occurance of the sought object. FindIndex() allows you to search object indices with a particular characteristic. It takes a delegate argument of type Predicate<T> which takes an argument of type T and return a bool. If a match is made it returns true and the search stops otherwise it returns false the search continues. Check out the following two examples using anonymous methods and lambda expressions.

```
ind = lp.FindIndex(delegate(Person p1)
 {
 return p1.Firstname.CompareTo("K")==0;
 });
ind = lp.FindIndex(r => r.Firstname.CompareTo("K") == 0);
```

In this context lambda expressions make life very easy. FindLastIndex() works in exactly the same way, but returns the last occurrence of the match.

Find() also takes a Predicate<T> argument and returns the first object matching. Lambda expressions simplify the search code considerably. For example:

```
Person tmp = lp.Find(delegate(Person p1)
 {
 return p1.Firstname[0] > 'M';
 });
List<Person> lt = lp.FindAll(r => r.Firstname[0] > 'H');
```

The method FindAll() returns a List<T> object of all objects matching the search criteria. Naturally we can make the criteria as complicated as we can imagine.

15.3 Other Collections

There are many other collection objects available free in .NET. A few will only be briefly mentioned here but if necessary you can look up more details on the internet.

15.3.1 Queues

These encapsulate a first-in, first-out (FIFO) queue (unlike the stack which was LIFO). They are generic Queue<T> and supportseveral important methods Enqueue(T), add an object to the queue and T Dequeue() removes an object from the queue and returns it. T Peek() allows you to see whats next in the queue without removing it. Count gives the number of items in a queue. CopyTo() allows the objects in the Queue to be copied to an existing array, whereas ToArray() copies the objects to a new array. The bool Contains(T) method checks to see if an object is in a Queue.

15.3.2 Stacks

We have written our version of these earlier and they implement a LIFO arrangement. They support the methods Pop(), Push(), Peek(), Contains(), CopyTo() and

ToArray() and are implemented as you would expect.

15.3.3 Linked Lists

Linked lists are an important construct in computer science. When inserting or removing elements from a standard array there can be a large amount of operations involved. For example, suppose you wish to insert an item at position 1 in a 1000 element array. Then you need to extend the size of the array (possibly get new memory, copy etc.), then shuffle all elements after position 1 along to make room and finally insert the element. A linked list reduces this to a few operations. However, there is a price to pay. Arrays are fasted for random access (e.g. get at the element at position 53), whereas in a linked list you must start at the start and move along one by one until you get to the specified position. In a linked list (doubly linked list) each element keeps a reference to the next (and previous) element in the list.

A LinkedList<T> is a collection of LinkedListNode<T> objects. Each LinkedListNode<T> has the following properties (i) List – the LinkedList<T> object the node is part of, (ii) Next - the next LinkedListNode<T> element, (iii) Previous – this should be obvious (iv) Value the object of type T associated with the node.

The LinkedList <T> object has the following properties: (i) Count - the number of nodes in the list (ii) First – the first node in the list (we can iterate forwards through all the nodes using this reference) and (iii) Last – the last node. There are a group of fairly obvious methods which take either a LinkListNode<T> or T object as an argument (they are overloaded) AddBefore(), AddAfter(), AddFirst(), AddLast(), Remove(). RemoveFirst() and RemoveLast() simply remove the first or last node on the list and thus don't need an argument. Clear() removes all nodes. Contains() checks to see if an object of type T is in the list. Find() (FindLast()) finds the first (last) occurrence of a specified object.

Chapter 16
EXCEPTIONS AND FILE I/O

16.1 Introduction

There are three classes of difficulties which may occur in software (i) a bug is a programmer error, which should be fixed by testing prior to software delivery (ii) an error is caused by user action, e.g. entering a letter where a number is expected. Errors should be anticipated and preventative code should be written to deal with these situations (iii) finally there are exceptions, these are the results of predictable, but unpreventable problems i.e. running out of memory, loss of network connection etc. This chapter provides a brief introduction to exception handling C#'s way of dealing with predictable but unpreventable problems. We also consider the unrelated topic of reading and writing data from/to file.

16.2 Throwing and catching

An exception is an object which contains information about the problem. We can create our own exceptions and throw them (also termed raising an exception) when problematic situations arise in our objects (for example in the Fraction class we could raise an exception if we try to divide by zero). Most often, objects from the .NET framework will raise exceptions and we should catch these (i.e. decide what to do), dealing with exceptions usually entails closing the program gracefully and providing a message about the problem.

It is considered best practice to handle (i.e. catch) an exception as close to the source of the problem as possible. For example suppose in a some program Method A calls Method B which then calls Method C where a problem occurs and an exception is thrown. First Method C is checked for an exception handler, if none exists then Method B is checked and finally Method A (this is termed unwinding the call stack). If the exception were handled in Method C then perhaps Method B and A can finish their work and some graceful solution is found. However if the exception has to handled in Method A then there is no garauntee that the work of Methods B and C can be rescued. In fact is no exception handler exists then the .NET framework just shuts the program down and outputs a technical (i.e. scary to the user) message.

16.2.1 How to throw an exception

All exception objects are of type System.Exception or derived from it. We can derive our own exception classes if needs be, or else just use some of the existing exceptions. For example, ArgumentNullException, InvalidCastException, OverflowException, are all exceptions we can create and throw.

Exceptions are thrown using the `throw` keyword. The procedure is to create an exception object (using the `new` keyword) and then throw it as follows:

```
ArgumentNullException e = new ArgumentNullException("parameter x", "passed
as null to evaluation procedure");
throw e;
```

or doing it all in the one go:

```
throw new ArgumentNullException("parameter x", "passed as null to evaluation
procedure");
```

That's all there is to it. It is up to some other programmer to decide what to do when things go wrong. The exception is thrown at a logical point, i.e. in the above example we might have some logic in a procedure checking for null arguments, and if it finds a null argument then the exception is thrown.

Consider the following example code where an exception is thrown. Run the code yourself and see what happens.

```
      class Tester
{

static void Main()
{
Console.WriteLine("Enter Main...");
Tester t = new Tester();
t.Run();
Console.WriteLine("Exit Main...");
}

public void Run()
{
Console.WriteLine("Enter Run...");
Func1();
Console.WriteLine("Exit Run...");
}

public void Func1()
```

```
{
Console.WriteLine("Enter Func1...");
Func2();
Console.WriteLine("Exit Func1...");
}

public void Func2()
{
Console.WriteLine("Enter Func2...");
throw new System.Exception();
Console.WriteLine("Exit Func2...");
}
}
```

16.2.2 Catching exceptions with the try and catch statements

This is where the real action happens. Suspect code, i.e. code which may be problematic (opening a file), should be wrapped in a try block, followed by a catch block which is executed only if an exception is raised in the try block. Here is the typical structure of exception handling code

```
try
{
//potentially hazardous code here
}
catch
{
//exception handler code goes here
}
```

Consider the following example:

```
class Tester
{

static void Main()
{
Console.WriteLine("Enter Main...");
Tester t = new Tester();
t.Run();
Console.WriteLine("Exit Main...");
}
```

```
public void Run()
{
Console.WriteLine("Enter Run...");
Func1();
Console.WriteLine("Exit Run...");
}

public void Func1()
{
Console.WriteLine("Enter Func1...");
Func2();
Console.WriteLine("Exit Func1...");
}

public void Func2()
{
Console.WriteLine("Enter Func2...");
try //possibly dangerous code
{
Console.WriteLine("Entering try block...");
throw new System.Exception();
Console.WriteLine("Exiting try block...");
}
catch //if exception occurs handle it here
{
Console.WriteLine("Exception caught and handled!");
}
Console.WriteLine("Exit Func2...");
}
}
```

Try the above code with and without the try and catch blocks. In a real situation you should try to fix the problem (i.e. restore internet or database connectivity etc.). Notice that all exit statement are output as after the exception is handled the code resumes executation after the catch block. Try the following slightly modified version of the above code:

```
class Tester
{

static void Main()
```

```
{
Console.WriteLine("Enter Main...");
Tester t = new Tester();
t.Run();
Console.WriteLine("Exit Main...");
}

public void Run()
{
Console.WriteLine("Enter Run...");
Func1();
Console.WriteLine("Exit Run...");
}

public void Func1()
{
Console.WriteLine("Enter Func1...");
try
{
Console.WriteLine("Entering try block...");
Func2();
Console.WriteLine("Exiting try block...");
}
catch
{
Console.Writeline("Exception caught and handled");
}
Console.WriteLine("Exit Func1...");
}

public void Func2()
{
Console.WriteLine("Enter Func2...");
throw new System.Exception();
Console.WriteLine("Exit Func2...");
}
}
```

What the differences in the output between the two versions?

16.2.3 Dedicated catching

In the above example the catch block is a generic catch-all and no provision is made for specific error situations that are thought to be likely. For example, if you are dividing two numbers it is possible that a divide by zero error occurs. Hence consider the following code:

```
try
{
int a = 5;
int b = 0;
Console.WriteLine("{0}/{1} = {2}", a, b, a / b);
}
catch (DivideByZeroException e)
{//dedicated handler for divide by zero
Console.WriteLine("Divide by zero error");
Console.WriteLine("Message = {0}", e.Message);
}
catch (ArithmeticException e)
{ //dedicated handler for this exception
Console.WriteLine("Arithmetic error");
Console.WriteLine("Message = {0}", e.Message);
}
catch(Exception e)
{ //handler for unanticipated errors
Console.WriteLine("Unknown error caught");
}
```

The order of the catch statements is important, i.e. most specific followed by more general, because more specific errors would match up to more general exceptions, hence the order gives more chance of finding the particular type of error. Notice that we can access the exception object (and this is the one thrown by either the system or the code) and use its properties and methods to find out more about the error and in fact give the user more information as well, if that is deemed to be useful.

16.2.4 The finally block

This is used for specific cases where some resource is opened for use (i.e. a file, a database connection, a graphical resource etc.) which ought to be released back into the system no matter what happens, i.e. error or no error. The typical use is as follows:

```
try
```

```
{
int a = 5;
int b = 0;
Console.WriteLine("Open File A");
Console.WriteLine("{0}/{1} = {2}", a, b, a / b);
}
catch (DivideByZeroException e)
{//dedicated handler for divide by zero
Console.WriteLine("Divide by zero error");
Console.WriteLine("Message = {0}", e.Message);
}
catch (ArithmeticException e)
{ //dedicated handler for this exception
Console.WriteLine("Arithmetic error");
Console.WriteLine("Message = {0}", e.Message);
}
catch(Exception e)
{ //handler for unanticipated errors
Console.WriteLine("Unknown error caught");
}
finally
{
Console.WriteLine("Close File A");
}
```

In this case, it doesnt matter what exception is is thrown or not thrown the file A will always be closed. This saves repetitious code in the try block and in each catch block and also means that the required action is only stated in one place. This is a neater solution and being human we may forget to take the action somewhere along the line.

16.3 File I/O

Files and folders are a key part of the day to day use of computers. Elementary users of computer systems are used to storing files and retrieving files using software applications (i.e. programs) and creating directory or folder structures to orgaise their data in a sensible fashion. It make sense then that we can create and manipulate both files and folders inside our software applications. Typically files are used to store data, so that between closing and opening a software application we can get back to where we were working.

16.3.1 FileInfo and DirectoryInfo Objects

These objects are primarily designed to be used in an informative capacity although they can be used for some file operations. Typically objects are constructed by passing a string representing the file or folder to be represented by the object. By default if the file or folder does not exist an exception is not thrown until the first time the object is used in a way that requires the file or folder to be valid. Therefore is it best practice to test for existence using the Exists property of each object to check that it actually exists before using the object and throw an exception if it does not exist. The following code gives an idea of some of the properties available for these objects.

```
FileInfo f = new FileInfo(@"c:\685.pdf");
try
{
if (!f.Exists)
throw new Exception("File does not exist");
Console.WriteLine("{0}", f.Exists.ToString());
Console.WriteLine("{0}", f.CreationTime.ToString());
Console.WriteLine("{0}", f.DirectoryName.ToString());
Console.WriteLine("{0}", f.Extension.ToString());
Console.WriteLine("{0}", f.FullName.ToString());
Console.WriteLine("{0}", f.LastAccessTime.ToString());
Console.WriteLine("{0}", f.LastWriteTime.ToString());
Console.WriteLine("{0}", f.Name.ToString());
Console.WriteLine("{0}", f.Length.ToString());
}
catch (Exception e)
{
Console.WriteLine("{0}, cannot proceed!", e.Message);
}
DirectoryInfo d = new DirectoryInfo(@"c:\Program Files\adobe");
try
{
if (!d.Exists)
throw new Exception("File does not exist");
Console.WriteLine("{0}", d.Exists.ToString());
Console.WriteLine("{0}", d.CreationTime.ToString());
Console.WriteLine("{0}", d.Parent.ToString());
Console.WriteLine("{0}", d.Extension.ToString());
Console.WriteLine("{0}", d.FullName.ToString());
Console.WriteLine("{0}", d.LastAccessTime.ToString());
Console.WriteLine("{0}", d.LastWriteTime.ToString());
Console.WriteLine("{0}", d.Name.ToString());
```

```
Console.WriteLine("{0}", d.Root.ToString());
}
catch (Exception e)
{
Console.WriteLine("{0}, cannot proceed!", e.Message);
}
```

Some of the more useful methods are briefly mentioned here. `Create()` creates and empty file or folder with the name specified in FullName and in the case of FileInfo a stream object for the file is returned (see next section). `Delete()` deletes the file of folder (in Directory info this can be made recursive). `MoveTo(string destFileName)` moves a file or folder. Check out the following in the online help: `CopyTo(), GetDirectories(), GetFiles() and GetFileSystemInfos()`. Perhaps a better implementation of many of these methods are provided by the `File` and `Directory` classes, where more intuitive static methods are exposed. For example, you should investigate the following: `File.Copy(), File.Move, File.Delete(), Directory.CreateDirectory()` `Directory.Move(), Directory.Copy()`.

It is also worth having a look at the Path class which exposes a variety of helpful static methods for manipulating folders and files without having to worry about parsing and formats etc.

16.3.2 Working with Files

Files are a fundamental part of computing and are an invaluable form of persisent storage whether on hard disk, cd or usb key. We will focus on the stream concept and associated objects here, although there are other objects in the .NET framework which support file activities. Streaming is a general concept associated with the transfer of data. For example, when you use Console.Writeline() you are streaming data from the program to the screen. When you accept input from the keyboard you are streaming data from the keyboard into your program. In these examples the Console object was automatically connected to the screen or keyboard so you you probably didn't think twice about. When data is streamed from some source outside the program, into your program this is called reading. When data is stream from your program to some external source this is called writing. When working with files the source of data or the destination of data is a file. However, unlike the screen or keyboard it does not make sense to automatically connect to every file ont he computer and have then ready to either sedn fo recieve data. Therefore before we read from or write to a file we must establish a connection. We must also disconnect fromthe file when we are finished as there are a limited number of files that can be held open on a computer at any one time.

A file is essentially a long string of bits of information. The operating system uses filenames to identify different sections of bits. When a section of bits are read into a program then it is up to the program to understand how to interpret the bits in a sensible manner. Likewise when we write a sequence of bits to a file then

they only make sense if we know how they were written and what was the overall design. Wihtout knowing the design or purpose then the file is meaningless. One method of interpreting the bits in a file is that every 8 bits makes up a char. This interprets every file as a sequence of characters and are commonly referred to as text files. For example these can be open, created and modified in notepad. Another way of interpreting a file is that there is adesign which specifies what each sub-sequence of bits means. For example, the first 32 bits are an integer number, the next 8 bits represent a char, the next 400 bits represent 400 floats side by side etc. This approach is of called reading/writing binary files. In .NET there are different objects to support each activity.

Binary files

Binary file access is encapsulated by the FileStream class. An object is constructed by passing the full name of the file and specifying the mode, access and sharing type which can be made up of from the corresponding enumerations `FileMode`, `FileAccess` and `FileShare`. The mode can be one of `Append, Create, CreateNew, Open, OpenOrCreate,` or `Truncate`. Append means open an existing file and go to the end of the file, if the file does not exist it will be created. Create will create a new file and if a file of the same name already exists then it will be overwritten (i.e. complete loss of information contained in the file). CreateNew specifies that a new file be created and an exception is thrown if the file already exists. Open requires that an existing file be opened and an exception is thrown if it does not exist. OpenOrCreate specifies that the operating system should open a file if it exists; otherwise, a new file should be created. Truncate specifies that the operating system should open an existing file. Once opened, the file should be truncated so that its size is zero bytes. FileAccess can be one of Read, Write or ReadWrite with the obvious meaning implied. FileShare specifies the kind of concurrent access other FileStream objects can have and the key options are None, Read, Write and ReadWrite. There are many constructors available for FileStream objects and at the very least you must always specify the name and mode of opening.

In order to read and write data to the FileStream object we need to use objects of type BinaryWriter and BinaryReader. Lets look at an example:

```
FileStream f = new FileStream(@"c:\tmp.txt", FileMode.Truncate,FileAccess.Write);
BinaryWriter bw = new BinaryWriter(f);
Random r = new Random();
int i, num;
double val;
Console.WriteLine("Start Writing");
bw.Write(10);//design is:  first int specifys number of double to follow
for (i = 0; i < 10; i++)
```

```
{
val = r.NextDouble();
bw.Write(val);
Console.WriteLine(val);
}
f.Close();
Console.WriteLine("End Writing");
//end of writing
//start of reading
f = new FileStream(@"c:\tmp.txt", FileMode.Open, FileAccess.Read);
BinaryReader br = new BinaryReader(f);
Console.WriteLine("Start Reading");
num = br.ReadInt32();
for (i = 0; i < num; i++)
{
val = br.ReadDouble();
Console.WriteLine(val);
}
f.Close();
Console.WriteLine("End Reading");
```

Of course unless you know that the file format is an int (say n) followed by n doubles, then the file is just plain gibberish. In fact run the code and take a look at the output in notepad. Don't forget notepad interprets each byte as a character and thats why we get rubbish on the screen. However, the beauty of binary files is that an int (for example) is stored in 4 bytes both in memory and in the file. Hence the number 2000000000, takes up 10 chars, which maybe up to 20 bytes, whereas in binary an int is stored in 4 bytes. Write is overloaded for all the basic datatypes and there is a version of read for each basic type as well.

Text files

Text files are easy to work with because typically you read and write strings which are usually human readable. These files should also be understandable when opened in notepad. Two classes encapsulate maniipulating these files: StreamWriter and StreamReader. The following example should suffice:

```
StreamWriter sw = new StreamWriter(@"c:\temp2.txt", false);
sw.WriteLine("Happy Birthday");
sw.WriteLine("You are 46");
sw.WriteLine("65,76,32,12");
```

```
Random r = new Random();
sw.WriteLine("Random number = {0}", r.NextDouble());
sw.Close();
StreamReader sr = new StreamReader(@"c:\temp2.txt");
string tmp = "";
do
{
tmp = sr.ReadLine();
Console.WriteLine(tmp);
}
while (tmp != null);
sr.Close();
```

The StreamWriter constructor takes the filename and a bool which specifies whether you wish to append lines of text (true) or just start at the beginning (false). The WriteLine() method works just like the Console version. In the StreamReader the method ReadLine() reads lines of text into a string and returns the null value when you get to the end.

Programming Assignment No. 5.: Polynomial Parser

The basic requirement of this assignment is to write a C# program which can resolve a simple polynomial into a vector of its components and to make use of this form to implement differentiation of the equation. The equation parser should be implemented as a function that is passed an ASCII character string, which is used to represent a polynomial of degree n, which has the general form:

$$a_{-n}x^{-n} + a_{-n+1}x^{-n+1} + \ldots + a_{-1}x^{-1} + a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1} + a_nx^n$$

where x is the dependant variable, and a_n are the coefficients of the polynomial (which may be limited to signed integer values).

The character string representation may be of any degree, n, and the components may be in any random order and may even be repeated. Use the '^' character to represent "to the power of". The character string may also include white space characters for readability, namely the space character, which do not convey any information but which nonetheless must be processed being part of the string. Some examples of valid equations are given below where '_' represents the space character.

```
1 + x + 2x^2 + 5x^4
2x^4 + 3x -12
-6 + 2x^3 - 2x + 4 - x
-4 x^-2 + x^-1 + 2x^3
```

```
x_^_2_+_2_x_^_3
```

The equation parser must resolve the polynomial into a vector representation, i.e. it must establish a value for the coefficient of each component of the polynomial. This information or vector should be stored in a single dimension integer array where the index into the array represents a specific component of the vector. Vector[0] represents the coefficient of x^0, Vector[1] represents the coefficient of x^1, and so on up to the degree of the polynomial.

The size of the vector array should be just sufficient to store the required vector, memory being allocated dynamically once the degree of the polynomial has been established. For example a polynomial of degree 2 would require an array of three elements to store all the component coefficients.

For testing purposes your program should first of all take in a polynomial in the form of a character string as input, resolve this into a vector using the parser function, and then pass this vector to a function which should differentiate the vector (or equation) and store the result in a different vector of appropriate dimension.

The results of each operation should be displayed in tabular form as illustrated in the following example where the equation input was

```
5x^3 + 3x^2 - x + 4
 Coefficients  x^0  x^1  x^2  x^3
 Equation      4    -1   3    5
 Equation'     -1   6    15
```

The results of the operation should also be stored to a user specified text file in the same tabular form as above. In the case where the file specified already exists and has a number of valid entries, the new entry to be added should be separated by a blank line from the others for readability.