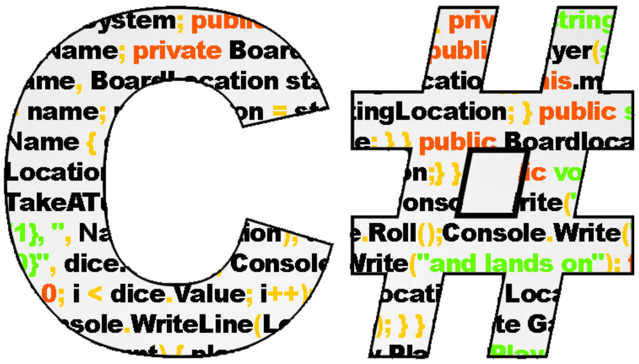
PROGRAMMING I



South Metro TAFE 2017

Version 6.0

Contents

INTRODUCTION

Welcome to the exciting world of programming in C# (pronounced C "sharp"). This book is full of worked examples for students who are starting to learn about computer programming languages. If you have programmed before you should still consider reading the text. It is worth it, just to refresh your understanding and you may actually learn something new.

If you have not programmed before, then this text will explain the basics and demonstrate the concepts with example code. Programming is not difficult, it is thinking in logical steps to solve a problem. The difficulty in learning to program is you will be confronted with a lot of ideas and concepts at the same time, and this can be confusing. The keys to learning programming are:

Practice:

You should code all examples and do a lot of programming that will force you to think about things from a problem solving point of view.

Study:

Look at programs and examples written by other people. You can learn a lot from studying code which other programmers have created. Figuring out how somebody else did the job is a great starting point for your solution. And remember that in many cases there is no best solution, just ones which are better in a particular context, i.e. the fastest, the smallest, the easiest to use etc.

Persistence:

Writing program code is hard work and can be very stressful. The principle reason why most students don't make it as programmers is that they give up. However, don't become too persistent, if you haven't solved a programming problem in 30 minutes you should call time out and seek help. Sometimes it helps to walk away from the problem and come back to it. Staying up all night trying to sort out a problem is not a good plan. It just makes you all irritable in the morning.

Reading the notes

These notes are written to be read straight through, and then referred to afterwards. They contain a number of Programming Points and Programming Activities. These are based on real programming experience and are to be taken seriously. Even if you skip the coding activities you should read these pages as there is important information about each topic.

CHAPTER ONE

In this chapter you will apply the basic C# language syntax and layout. The learning outcomes are to code and execute a simple program using basic data types and simple programming constructs. At the end of the chapter you will have:

- Applied basic language syntax rules,
- Use language data types, operators and expressions to create clear and concise code,

A Bit of History

Microsoft developed C#, a new programming language based on the C and C++ languages. Microsoft describes C# in this way: C# is a simple, modern, object–oriented, and typesafe programming language derived from C and C++. The programming language C# is firmly planted in the C and C++ family tree of languages and will immediately be familiar to C and C++ programmers. C# aims to combine the high productivity of visual basic and raw power of C++.

Anders Hejlsberg, the principal architect of C#, is known for his work with Borland on Turbo Pascal and Delphi (based on object–oriented Pascal). After leaving Borland, Hejlsberg worked at Microsoft on Visual J++.

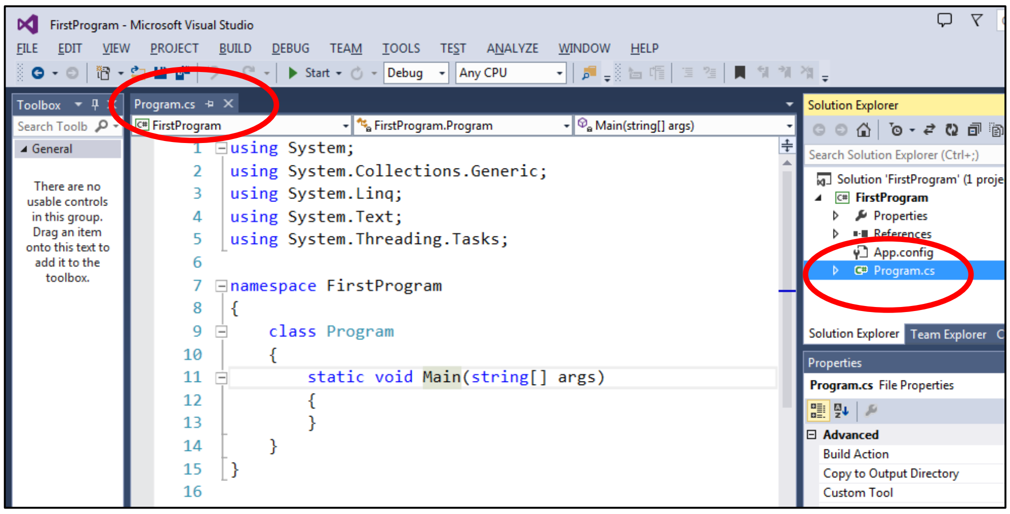
C# is a case sensitive language.

Some aspects of C# will be familiar to those who have programmed in C, C++, or Java. C# incorporates the Smalltalk concept, which means everything is an object. In other words, all types in C# are objects. C# properties are similar to Visual Basic language properties. The Rapid Application Development (RAD) goal in C# is assisted by C#'s use of concepts and keywords, such as class, structure, statement, operator and enumeration. The language also utilizes the concepts contained in the Component Object Model (COM) architecture.

The Basic Program

To get started, open Visual Studio and select File | New | Project. You'll see the New Project window. Your first task is to select the Console Application as the program type from the Visual C# menu option. Then set the program name to FirstProgram and specify a location of your choice for where the project will be created. Other features of the New Project window include the ability to specify the .NET Framework version, sorting options, icon size options, and a search capability.

Once the framework has been created the coding window will open, this is the Program.cs file (Figure 1.). All the console applications in this book will be based on this procedure. However I recommend you create a suitable naming convention and store each program in a new folder. This will help track your work and allow you to refer back to these programs at a later stage.



There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

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

namespace FirstProgram
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Figure 1

Having run the New Project Wizard for a Console application, you'll see a file named Program.cs that contains skeleton code in the editor. Visual Studio will create skeleton code using built-in templates for most project types that you create. You're free to add, remove, or modify this code as you see fit.

Main

The innermost block of the C# code is the static void Main(string[] args) definition, which is called a method. You'll learn later that methods are one way you can group code into logical chunks of functionality. You can think of methods as actions where you, as the method author, tell the computer what to do. The name of this particular method is Main, which is referred to as the entry point of the program, the place where a Console application first starts running. Another way of thinking about Main is that this is the place your computer first transfers control to your program. Therefore, you would want to put code inside of Main to make your program do what you want it to.

In C#, Main must be capitalized. It's also important to remember that C# is case-sensitive, meaning that Main (capitalized) is not the same as main (lowercase). Although Visual Studio capitalizes your code for you if you forget; capitalization is a common mistake, especially for new programmers learning C#.

In C#, methods can return values, such as numbers, text, or other types of values, and the type of thing they can return is specified by you right before the method name. Since Main, in the C# example, does not return a value, the return type is replaced with the keyword “void”. Methods can specify parameters for holding arguments that callers pass to the method. In the case of Main, the parameter is an array of strings, with a variable name of args. The args parameter will hold all of the parameters passed to this program from the command line.

One more part of the C# Main method is the “static” keyword, which is a modifier that says there will only ever be a single instance of this method for the life of the program. To understand instances, consider that methods are members of object types where an object can be anything in the domain of the application you’re writing, such as a Customer, Account, or Vehicle. Think about a company that has multiple customers. Each customer is a separate instance, which also means that each Customer instance contains methods that belong to each instance. If an object such as Customer has methods that belong to each instance, those methods are not static. However, if the Customer object type has a method that is static, then there would only be a single copy of that method that is shared among all Customer objects. For example, what if you wanted to get a discount price for all customers, regardless of who the customer is; you would declare a static method named GetCustomerDiscount. However, if you wanted information that belonged to a specific customer, such as an address, you would create an instance method named GetAddress that would not be modified as static.

In C#, the curly braces define the “begin” and “end” of the Main method. Next, notice that the C# Main method is enclosed inside of a set of braces that belong to something called a class that has been given the name “Program”.

Class

Methods always reside inside of a type declaration. A type could be a class or struct for C#. The term type might be a little foreign to you, but it might be easier if you thought of it as something that contains things. Methods are one of the things that types contain.

The Console application defined the skeleton code class to have the name Program. In reality you can name the class anything you want. Whatever names you choose should make sense for the purpose of the class. For example, it makes sense for a class that works with customers to be named Customer and only contain methods that help you work with customers. You wouldn't add methods for working directly with invoices, products, or anything other than customers because that would make the code in your Customer class confusing. Classes are organized with namespaces, which are discussed next.

Namespace

A namespace helps make your class names unique and therefore unambiguous. They are like adding a middle name and surname to your first name, which makes your whole name more unique. A namespace name, however, precedes the class name, whereas your middle name and surname follow your first or given name. A namespace also helps you organize code and helps you find things in other programmers' code. This organization helps to build libraries of code where programmers have a better chance to find what they need. The .NET platform has a huge class library that is organized into namespaces and assemblies; this will become clearer the more you program. The main .NET namespace is "System", which has multiple sub-namespaces. For example, you can find .NET classes for working with data in the System.Data. While the networking protocols like TCP/IP, FTP, or HTTP are located in the System.Net.

Another benefit of namespaces is to differentiate between classes that have the same name in different libraries. For example, what if you bought a thirdparty library that has a Customer class? Think about what you would do to tell the difference between Customer classes. The solution is namespaces, because if each Customer has its own namespace, you can write code that specifies each Customer by its namespace. Always using namespaces is widely considered to be a best practice.

You can put many classes inside of a namespace, where inside means within the beginning and ending braces for a namespace.

The “using**”** directives at the top of the C# program in Figure 1 are really a shortcut that makes it easier for you to write code. For example, the System namespace contains the Console class. If the using System directive were not present, you would be required to write System.Console.WriteLine instead of just Console.WriteLine. This is a short example, but using directives can help clean up your code and make it more readable.

The C# Language

C# is a modern, general-purpose, object-oriented, high-level programming language. Its syntax is similar to that of C and C++ but many features of those languages are not supported in C# in order to simplify the language, which makes programming easier.

The C# programs consist of one or more files with a .cs extension, which contain definitions of classes and other types. These files are compiled by the C# compiler to executable code and as a result assemblies are created, which are files with the same name but with a different extension (.exe or .dll). For example, if we compile FirstProgam.cs, we will get a file with the name FirstProgram.exe (some additional files will be created as well, but we will not discuss them at the moment).

We can run the compiled code like any other program on our computer (by double clicking it). If we try to execute the compiled C# code (for example FirstProgram.exe) on a computer that does not have the .NET Framework, we will receive an error message.

The C# language uses the following keywords (Table 1) to build its programming constructs. Since the creation of the first version of the C# language, not all keywords are in use. Some of them were added in later versions. The main program elements in C# (which are defined and used with the help of keywords) are classes, methods, operators, expressions, conditional statements, loops, data types, exceptions and few others. In this book, we will review all of these programming constructs and use most of the keywords from Table 1.

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

Table 1

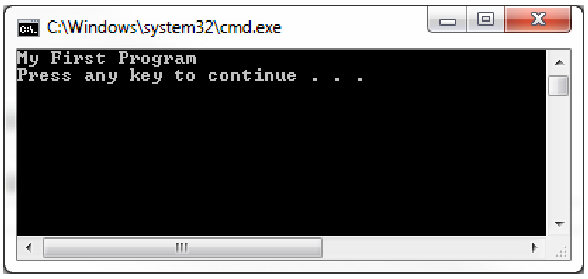
This brief outline of the C# language does not fully demonstrate the complex and powerful tools that are available through the use of .NET platform, .NET Libraries and .NET technologies.

Programming Activity 1

Time for some code; into the Program.cs window add the following code,

```
class Program
{
        static void Main(string[] args)
        {
                Console.WriteLine("My First Program");
        }
}
```

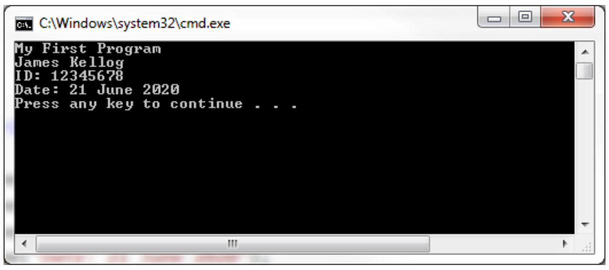
To run this program hold down the CTRL key and press F5, you will see a command window open which displays the output of the program,



```
My First Program
Press any key to continue . . .
```

Close the window and you will return to the programming environment. Add additional lines to the program that will display your name and student ID. The code should be similar to

```
Console.WriteLine("My First Program");
Console.WriteLine("James Kellog");
Console.WriteLine("ID: 12345678");
Console.WriteLine("Date: 21 June 2020");
```



```
My First Program
James Kellog
ID: 12345678
Date: 21 June 2020
Press any key to continue . . .
```

Data Types

In this section we will get familiar with primitive types and variables in C#; what they are and how to work with them. First we will consider the data types; integer, real types with floating-point, Boolean, character, string and object type. We will continue with variables, their characteristics, how to declare them, how they are assigned a value and what variable initialization is.

Basic Data Types

Data types are sets (ranges) of values that have similar characteristics. For instance the byte type specifies the set of integers in the range of [0 … 255].

Data types are characterized by:

Name : for example, int;

Size : (how much memory they use) – for example, 4 bytes;

Default value : for example 0.

Basic data types in C# are distributed into the following types:

Integer types : sbyte, byte, short, ushort, int, uint, long, ulong;

Real floating-point types : float, double;

Real type with decimal precision : decimal;

Boolean type : bool;

Character type : char;

String : string;

Object type : object.

These data types are called primitive (built-in types), because they are embedded in C# language at the lowest level. Table 2 represents all these primitive data types, their default values and range.

Data Types	Default Value	Minimum Value	Maximum Value
sbyte	0	-128	127
byte	0	0	255
short	0	-32768	32767
ushort	0	0	65535
int	0	-2147483648	2147483647
uint	0u	0	4294967295
long	0L	-9223372036854775808	9223372036854775807
ulong	0u	0	18446744073709551615
float	0.0f	$\pm 1.5 \times 10^{-45}$	$\pm 3.4 \times 10^{38}$
double	0.0d	$\pm 5.0 \times 10^{-324}$	$\pm 1.7 \times 10^{308}$
decimal	0.0m	$\pm 1.0 \times 10^{-28}$	$\pm 7.9 \times 10^{28}$
bool	false	Two possible values: **true** and **false**	
char	'\u0000'	'\u0000'	'\uffff'
object	null	-	-
string	null	-	-

Table 2

Programming Activity 2

Let put these data types into action. Add the following code to a console activity and run,

```
// Declare some variables
byte centuries = 20;
ushort years = 2000;
uint days = 730480;
ulong hours = 17531520;
// Print the result on the console
Console.WriteLine(centuries + " centuries are "
        + years + " years, or "
        + days + " days, or "
        + hours + " hours.");
```

This code has comments which are not complied but serve to inform the programmer about the code. The Console.WriteLine is split across several lines to aid readability. Change the year variable from ushort to byte. Why is there an error?

Next, add the following code and run the program again,

```
float floatPI = 3.14f;
Console.WriteLine(floatPI); // 3.14
double doublePI = 3.14;
Console.WriteLine(doublePI); // 3.14
double nan = Double.NaN;
Console.WriteLine(nan); // NaN
double infinity = Double.PositiveInfinity;
Console.WriteLine(infinity); // Infinity
```

The smallest real value of type `double` is the constant `Double.MinValue` = `-1.79769e+308` and the largest is `Double.MaxValue` = `1.79769e+308`. The closest to 0 positive number of type `double` is `Double.Epsilon` = `4.94066e-324`. As with the type `float` the variables of type `double` can take the special values: `Double.PositiveInfinity` (+∞), `Double.NegativeInfinity` (-∞) and `Double.NaN` (invalid number). It is important to use the correct data type to hold your data values, otherwise you will lose data or your program will yield incorrect results.

Now add this code to your editor, and notice the precision of the results.

```
// Declare some variables
float floatPI = 3.141592653589793238f;
double doublePI = 3.141592653589793238;
// Print the results on the console
Console.WriteLine("Float PI is: " + floatPI);
Console.WriteLine("Double PI is: " + doublePI);
// Console output:
// Float PI is: 3.141593
// Double PI is: 3.14159265358979
```

We see that the number π which we declared as a float is rounded to the 7th digit, while the variable we declared as a double is rounded to the 15th digit.

In calculations with real floating-point data types it is possible to observe strange behaviour, because during the representation of a given real number there is a loss of accuracy. For example these numbers do not have an accurate representation in float and double 0.1, 1/3, 2/7. Use the following code to see this effect,

```
float f = 0.1f; Console.WriteLine(f);
// 0.1 (correct due to rounding)
double d = 0.1f; Console.WriteLine(d);
// 0.100000001490116 (incorrect)
float ff = 1.0f / 3; Console.WriteLine(ff);
// 0.3333333 (correct due to rounding)
double dd = ff; Console.WriteLine(dd);
// 0.333333343267441 (incorrect)
```

The reason for the unexpected result in the first example is the fact that the number 0.1 (i.e. 1/10) has no accurate representation in the real floating-point number format IEEE 754 and its approximate value is recorded.

In the second case the number 1/3 has no accurate representation and is rounded to a number very close to 0.3333333. The value of this number is clearly visible when it is written in double type, which preserves more significant digits.

C# supports the decimal floating-point arithmetic, where numbers are represented via the decimal numeral system rather than the binary one. Thus, the decimal floating point-arithmetic type in C# does not lose accuracy when storing and processing floating-point numbers. It has a precision from

28 to 29 decimal places. Its minimal value is -7.9×1028 and its maximum value is +7.9×1028.

The Boolean type is declared with the keyword bool. It has two possible values: true or false. Its default value is false. It is used most often to store the calculation result of logical expressions. Add this code to a new console application and run,

Programming Activity 3

```
// Declare some variables
int a = 1;
int b = 2;
// Which one is greater?
bool greaterAB = (a > b);
// Is 'a' equal to 1?
bool equalA1 = (a == 1);
// Print the results on the console
if (greaterAB)
{
      Console.WriteLine("A > B");
}
else
{
      Console.WriteLine("A <= B");
}
Console.WriteLine("greaterAB = " + greaterAB);
Console.WriteLine("equalA1 = " + equalA1);
// Console output:
// A <= B
// greaterAB = False
// equalA1 = True
```

In this example we declare two variables of type int, compare them and assign the result to the bool variable greaterAB. Similarly, we do the same for the variable equalA1. If the variable greaterAB is true, then A > B is printed on the console, otherwise A <= B is printed. The IF/ELSE selection construct will be explained in detail later in the chapter. Reverse the values of the variables a and b; what happened?

Now we move onto characters and strings. A character type is a single character (16-bit number of a Unicode table character). It is declared in C# with the keyword char. The Unicode table is a technological standard that represents any character (letter, punctuation, etc.) from all human languages as writing systems (all languages and alphabets) with an integer or a sequence of integers. Let’s see an example, add this code to a console application and run,

```
// Declare a variable
char ch = 'a';
// Print the results on the console
Console.WriteLine( "The code of '" + ch + "' is: " + (int)ch);
ch = 'b';
Console.WriteLine( "The code of '" + ch + "' is: " + (int)ch);
ch = 'A';
Console.WriteLine( "The code of '" + ch + "' is: " + (int)ch);
// Console output:
// The code of 'a' is: 97
// The code of 'b' is: 98
// The code of 'A' is: 65
```

In this example we declare one variable of type char, initialize it with value 'a', then 'b', then 'A' and print the Unicode values of these letters to the console. Note that a character variable type is enclosed by single quotes ‘a’.

Strings are sequences of characters. In C# they are declared by the keyword string. Their default value is null. Strings are enclosed in quotation marks, for example “Hello World”.

In the next example we declare several variables of type `string`, initialize them and print their values to the console,

```
// Declare some variables
string firstName = "John";
string lastName = "Smith";
string fullName = firstName + " " + lastName;
// Print the results on the console
Console.WriteLine("Hello, " + firstName + "!");
Console.WriteLine("Your full name is " + fullName + ".");
// Console output:
// Hello, John!
// Your full name is John Smith.
```

Notice that strings can be joined using the + operator, this is known as concatenation.

The Object type is a special type, which is the parent of all other types in the .NET Framework. Declared with the keyword `object`, it can take values from any other type. It is a reference type, which is an index (address) of a memory area which stores the actual value.

In the next example we declare several variables of type `object`, initialize them and print their values on the console,

```
// Declare some variables
object container1 = 5;
object container2 = "Five";
// Print the results on the console
Console.WriteLine("The value of container1 is: " + container1);
Console.WriteLine("The value of container2 is: " + container2);
// Console output:
// The value of container1 is: 5
// The value of container2 is: Five
```

As you can see from the example, we can store the value of any other type in an object type variable. This makes the object type a universal data container.

Variables

If you have been running the code from the previous examples you will have used variables of type number, character and boolean. In order to work effectively with data we should use variables. We have already seen their usage in previous examples, but now let's look at them in more detail. A variable is a container of information, which can change its value. It provides a means for:

Storing information,
Retrieving the stored information,
Modifying the stored information.

In C# programming, you will use variables to store and process information all the time.

The characteristics of Variables are;

Name : for example age,
Type : (of the information preserved in them), for example int,
Value : (stored information), for example 25.

A variable is a named area of memory, which stores a value from a particular data type, so that area of memory is accessible in the program by its name.

Variables can be stored directly in the operational memory of the program (in the stack) or in the dynamic memory in which larger objects are stored (such as character strings and arrays).

Primitive data types (numbers, characters, boolean) are called value types because they store their value directly in the program stack.

Reference data types (such as strings, objects and arrays) are an address, pointing to the dynamic memory where their value is stored. They can be dynamically allocated and released i.e. their size is not fixed in advance contrary to the case of value types.

When we want the compiler to allocate a memory area for some information which is used in our program we must provide a name for it. This works like an identifier and allows reference to the relevant memory area. The name of the variable can be our choice but must follow certain rules defined in the C# language specification:

> Variable names can contain the letters 'a-z', 'A-Z', the digits '0-9' as well as the character '_'.
> Variable names cannot start with a digit.
> Variable names cannot coincide with a keyword of the C# language (refer Table 1).

So, here are some correct variable names,

```
name
first_Name
_name1
```

And here are some incorrect variable names,

```
1 (digit)
if (keyword)
1name (starts with a digit)
```

Now we will provide some recommendations for how you should name your variables. The variable name should be descriptive and explain what the variable is used for. For example, an appropriate name for a variable storing a person's name is personName and inappropriate name is A37. Only Latin characters should be used. Although Cyrillic is allowed by the compiler, it is not a good practice to use it in variable names or in the rest of the identifiers within the program. In C# it is generally accepted that variable names should start with a small letter and include small letters, every new word, however, starts with a capital letter. This is often referred to as camel case. For instance, the variable name firstName is correct and better to use than firstname or first_name. Usage of the character _ in the variable names is considered poor naming style. Variable names should be neither too long nor too short; they just need to clarify the purpose of the variable within its context. As mentioned earlier uppercase and lowercase letters should be used carefully as C# distinguishes them. For instance, 'age' and 'Age' are different variables.

When you declare a variable, you perform the following steps:

1. specify its type (such as int),
2. specify its name (identifier, such as age),
3. optionally specify initial value (such as 25).

Programming Activity 4

Enter this code into a console application to see some common variable declarations and assignments,

```
// String variables
string myName;
myName = "Alan";
string yourName = "Fred";
// Integer variables
int myAge;
myAge = 31;
int yourAge = 21;
// Boolean variable
bool myMarrageStatus;
myMarrageStatus = false;
bool yourMarrageStatus = true;
// Double variables
double myIncome;
myIncome = 59400;
double yourIncome = 78300;
```

```
// Character variables
char myInitial;
myInitial = 'A';
char yourInitial = 'F';
// Display data
Console.WriteLine(myName + " "
                    + myAge + " "
                    + myMarrageStatus + " "
                    + myIncome + " "
                    + myInitial);
Console.WriteLine(yourName + " "
                    + yourAge + " "
                    + yourMarrageStatus + " "
                    + yourIncome + " "
                    + yourInitial);
```

In this example the variable names are easy to read and contain data that is relevant. Assigning a value to a variable is the act of providing a value that must be stored in the variable. This operation is performed by the assignment operator "=". On the left side of the operator we put the variable name and on the right side its new value. This was accomplished with all the variables beginning with “my...”, for example myName = “Alan”.

However, consider the following code, what is the result of running this code?

Programming Activity 5

```
string superMan, maryPoppins = "";
int blackMagic;
int greenLantern = 0;
int batMan =  greenLantern + 1;
Console.Write("Enter your name : ");
superMan = Console.ReadLine();
blackMagic = superMan.Length - batMan;
while (blackMagic >= greenLantern)
{
       maryPoppins = maryPoppins + superMan[blackMagic];
       blackMagic--;
}
Console.WriteLine("Hello " + maryPoppins);
```

The word initialization in programming means specifying an initial value. When setting a value to a variable at the time of their declaration we actually initialize them. In the previous code we initialised the string maryPoppins, and the integers greenLantern and batMan. Despite the strange names these variables can be displayed because they have a value. The complier will raise an error if you try to display and un-initialised variable. The previous code just reverses the string entered by the user.

Operators

In this section we will examine the operators in C# and the actions they can perform when used with the different data types. We will explain which operators have higher priority and analyse the different types of operators.

Operators allow processing of primitive data types and objects. They take an input of one or more operands and return some value as a result. Operators in C# are special characters, for example operators are the signs for adding, subtracting, multiplication and division from math (+, -, *, /).

Operators in C# can be separated in several different categories:

Arithmetic operators are used to perform simple mathematical operations.
Assignment operators allow assigning values to variables.
Comparison operators allow comparison of two literals and/or variables.
Logical operators work with Boolean data types and Boolean expressions.
Binary operators perform operations on the binary representation of numerical data.

Type conversion operators convert data from one type to another.

Below is a table of the operators, separated into categories,

Category	Operators
arithmetic	-, +, *, /, %, ++, --
logical	&&, ||, !, ^
binary	&, |, ^, ~, <<, >>
comparison	==,!=, >, <, >=, <=
assignment	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
string concatenation	+
type conversion	(type), as, is, typeof, sizeof
other	., new, (), [], ?:, ??

Table 3

Some operators in C# perform different operations on the different data types. For example the operator + is used with numeric data types (int, long, float, etc.) to performs mathematical addition. However, when we use it with strings, the operator concatenates (joins together) the content of the two variables/literals and returns the new string.

Programming Activity 6

Let put this to the test, add the following code to a console application and run,

```
int a = 7;
int b = 6;
Console.WriteLine(a + b);
string firstName = "Bruce"; string lastName = "Lee";
// Do not forget the space between them
string fullName = firstName + " " + lastName;
Console.WriteLine(fullName);
```

The compiler and the program will “know” what to do when using the + operator. Now add this code and see the results, can you explain the answer?

```
char c = '7';
char d = '8';
Console.WriteLine(c + d); // 111
```

Hint: What is the Unicode of 7 and 8?

Operator Precedence

Some operators have precedence (priority) over others. For example, in mathematics multiplication has precedence over addition. The operators with a higher precedence are calculated before those with lower precedence. The bracket operator **()** is used to change the precedence and is calculated first.

The information in Table 4 illustrates the precedence of the operators in C#:

Priority	Operators
Highest priority	(,)
	++, -- (as postfix), new, (type), typeof, sizeof
	++, -- (as prefix), +, - (unary), !, ~
	*, /, %
	+ (string concatenation)
	+, -
	<<, >>
	<, >, <=, >=, is, as
	==, !=
	&, ^, |
	&&
	||
	?:, ??
Lowest priority	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

Table 4

The operators located at the top of the table have higher precedence than those below them, and respectively they have an advantage in the calculation of an expression. To change the precedence of an operator we can use brackets.

Programming Activity 7

Add the following code to a console application and run,

```
double x = 10;
double y = 10;
double z = 10;
double result;
// Ambiguous
```

```
result = x + y / z;
Console.WriteLine("result = " + result); // 11
// Unambiguous, recommended
result = (x + y) / z;
Console.WriteLine("result = " + result); // 2
```

If you change the data types of the variables x, y, & z from double to integer will you get the same results? Why not? See below…

Arithmetic Operators

The arithmetical operators perform addition, subtraction and multiplication on numerical values and the result is also a numerical value. The division operator / has a different effect on integer and real numbers. When we divide an integer by an integer (like int, long and sbyte) the returned value is an integer (no rounding, the fractional part is cut). Such division is called an integer division. For example integer division: 7 / 3 = 2.

Integer division by 0 is not allowed and causes a runtime exception `DivideByZeroException`.

The remainder of integer division can be obtained by the operator %. For example, 7 % 3 = 1, and 10 % 2 = 0.

When dividing two real numbers (e.g. float, double, etc.), real division is done (not integer division), and the result is a real number with a fractional part. For example: 5.0 / 2 = 2.5.

Real division by 0.0 is allowed the result is +∞ (`Infinity`), -∞ (`-Infinity`) or NaN (invalid value).

The operator for increasing by one (increment) ++ adds one unit to the value of the variable, respectively the operator -- (decrement) subtracts one unit from the value. When we use the operators ++ and -- as a prefix (when we place them immediately before the variable), the new value is calculated first and then the result is returned. When we use the same operators as post-fix (meaning when we place them immediately after the variable) the original value of the operand is returned first, then the addition or subtraction is performed.

Programming Activity 8

Add the following code to a console application and run,

```
double result;
//real division
double a = 10.0;
double b = 0.0;
result = a / b;
Console.WriteLine("result = " + result); // Infinity
// Integer division
int x = 10;
int y = 0;
result = x / y;
Console.WriteLine("result = " + result); // Error message
```

Division can be very tricky and produce unexpected results. Next, add the following code to view how the increment/decrement operators work,

Programming Activity 9

```
int z = 1;
// Loop five times
for (int i = 0; i < 5; i++)
{
        // z++ display z and then increment;
        Console.WriteLine("z++ = " + (z++));
        // ++z increment then display z;
        Console.WriteLine("++z = " + (++z));
        Console.WriteLine("---------");
}
```

The effects of the increment and decrement are show after the value has been evaluated. Change the increment operator x++ to the decrement operator x--, what happens?

Logical Operators

Logical or Boolean operators take Boolean values and return a Boolean result (true or `false`). The basic Boolean operators are, "AND" (&&), "OR" (||), "exclusive OR" (^) and logical negation (!). Table 5 contains the logical operators in C# and the operations that they perform:

x	y	!x	x && y	x || y	x ^ y
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Table 5

Table 5 demonstrates how two Boolean variables (x and Y) will be evaluated when used with each of the Boolean operators. This will be used in the Programming Constructs section later in the text.

Programming Activity 10

The next example uses a Boolean variable to report on a simple user input question. Add this code to a console application and run,

```
// Declare boolean variable
bool isRaining;
char answer;
Console.WriteLine("Is it raining? y/n");
// Get your answer
if (char.TryParse(Console.ReadLine(), out answer))
{
        // Convert to lower case and evaluate if yes
        if (Char.ToLower(answer) == 'y')
        {
                isRaining = true;
                Console.WriteLine("isRaining is " + isRaining );
        }
        // Convert to lower case and evaluate if no
        if (Char.ToLower(answer) == 'n')
        {
                isRaining = false;
                Console.WriteLine("isRaining is " + isRaining);
        }
}
```

What happens if you respond with a character that is not ‘y’ or ‘n’? This situation will be considered in in the Programming Constructs section of the text.

Expressions

Much of a program’s work is the calculation of expressions. Expressions are sequences of operators, literals and variables that are calculated to a value of some type (number, string, object or other type).

Programming Activity 11

Let's look at some basic expressions and how they can be used to solve a problem. Add the following code to a console application and run,

```
// Declare variable
int radius = 0;
Console.Write("Enter the circle radius :");
// Get user input from keyboard
string keyBoardInput = Console.ReadLine();
// Convert to integer
bool response = int.TryParse(keyBoardInput, out radius);
// If input is valid do the calculations
if (response)
{
      // Expression for calculating the area of the circle
      double area = Math.PI * radius * radius;
      // Expression for calculating the circumference of the circle
      double circumference = 2 * Math.PI * radius;
      Console.WriteLine("Radius " + radius);
      Console.WriteLine("Area " + area);
      Console.WriteLine("Circumference " + circumference);
}
```

This example uses the Math library to get a value for PI; which is used to calculate the area and circumference. In the next example the results are unexpected; add this code to a console application and run. Can you see the difference between integer and real division?

Programming Activity 12

```
// Division on integers
double incorrect = (double)((1 + 2) / 4);
Console.WriteLine("incorrect (1 + 2) / 4 = " + incorrect);
// Division on double
double correct = ((double)(1 + 2)) / 4;
Console.WriteLine("correct (1 + 2) / 4 = " + correct);
// Brackets make a difference
Console.WriteLine("concatenate -> 2 + 3 = " + 2 + 3);   // 2 + 3 = 23
Console.WriteLine("addition -> 2 + 3 = " + (2 + 3));    // 2 + 3 = 5
```

When working with expressions it is important to use brackets whenever there is the slightest doubt about the priorities of the operations.

Optional Activities

For each activity write the appropriate console code.

Act-1: Concatenation or join two strings; using the "+" operator

Solution:

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello C Sharp!\n");
            Console.WriteLine("Please enter your Name : \n");
            string UserName = Console.ReadLine();
            Console.WriteLine("Hello " + UserName);
        }
    }
}
```

Act-2: Using a place marking and pass variable to string output; using {0}

Solution:

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Please enter your Name : \n");
            string UserName = Console.ReadLine();
            Console.WriteLine("Hello  {0}", UserName);
        }
    }
}
```

Act-3: Using a place marking and pass multiple variables to string output; using {0}

Solution:

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Please enter your first name :");
            string FirstName = Console.ReadLine();
            Console.WriteLine("Please enter your last name :");
            string LastName = Console.ReadLine();
```

```
                Console.WriteLine("Hello  {0}, {1}", FirstName, LastName);
                Console.Read();
            }
        }
    }
```

Act-4: Show the difference between Divide and Modulus; using / and %

Solution:

```
    namespace ConsoleApplication1
    {
        class Program
        {
            static void Main(string[] args)
            {
                Console.WriteLine("Divide");
                int Numerator = 10;
                int Denominator = 2;

                int Result1 = Numerator / Denominator;
                int Result2 = Numerator % Denominator;

                Console.WriteLine("Quotent Result = {0}", Result1);
                Console.WriteLine("Remainder Result = {0}", Result2);
            }
        }
    }
```

CHAPTER TWO

In this chapter you will continue to use the basic C# language syntax and layout. The learning outcomes are to code and execute a simple program using the three programming constructs. At the end of the chapter you will have:

- Applied basic language syntax rules,
- Use language data types, operators and expressions to create clear and concise code,
- Use appropriate language syntax for sequence, selection and iteration constructs.

Programming Constructs

All programming languages utilise program constructs. In imperative languages they are used to control the order (flow) in which statements are executed (or not executed). There are three basic programming constructs which can be classified as follows:

Sequence
Selection
Repetition

In this section we will examine each of these constructs and create simple programs that use these constructs to determine the flow and order of code execution.

Sequence

The sequence construct tells the processor which statement is to be executed next. By default, in imperative languages, this is the statement following the current statement (or the first statement in the program). The following code demonstrates a sequence of actions that are performed as a block of code.

Programming Activity 13

Add the following code to a console application and run,

```
string fullName;
string myLocation;
Console.Write("What is your full name : ");
fullName = Console.ReadLine();
Console.Write("What is your location (town or city) : ");
myLocation = Console.ReadLine();
Console.WriteLine("Hello " + fullName + " from " + myLocation);
```

In this example each line of code was executed starting at the top and ending at the bottom. The sequence construct is the simplest programming construct.

Selection

A selection construct provides for selection between multiple alternatives. We can identify two types of selection constructs:

IF / ELSE statements
SWITCH / CASE statements

The basic selection statement involves the IF and ELSE keywords. The keyword IF must be followed by a Boolean expression in parentheses. The statement that follows is executed if the Boolean expression resolves to TRUE. For example;

Programming Activity 14

```
int temperature;
bool keyBoardInput;
Console.Write("What is the temperature outside? ");
keyBoardInput = int.TryParse(Console.ReadLine(), out temperature);
if (keyBoardInput)
{
        if (temperature > 30)
        {
                Console.WriteLine("It is a hot day");
        }
```

In this example the user must input an integer value greater than 30 before the program will report it is a hot day. This program requires more code to ensure all possible outcomes are considered. The extension of the IF statement is the optional ELSE. Add the ELSE statement after the IF statement to improve output,

```
        else
        {
                Console.WriteLine("It is a cool day");
        }
}
```

Now the program will report a second message if the temperature is less than 30. But what if the user enters a non-integer value? Let's add a second ELSE statement to fix that.

Programming Activity 15

Here is the finished IF/ELSE code with comments,

```
int temperature;
bool keyBoardInput;
Console.Write("What is the temperature outside? ");
keyBoardInput = int.TryParse(Console.ReadLine(), out temperature);
if (keyBoardInput)
{
        if (temperature > 30)
        {
                Console.WriteLine("It is a hot day");
        }// IF temperature
        else
        {
                Console.WriteLine("It is a cool day");
        }// ELSE temerpature
}// IF keyBoardInput
else
{
        Console.WriteLine("You did not enter an Integer Value");
}// ELSE keyBoardInput
```

Notice the alignment of the brackets with each IF/ELSE statement, this formatting improves the readability. Placing a second set of IF/ELSE statements within an IF/ELSE statement is referred to as nesting; this provides a method to evaluate several alternative pathways. Replace the inner IF/ELSE code with the following nested statements,

Programming Activity 16

```
if (temperature > 40)
{
        Console.WriteLine("It is a VERY hot day");
}
else if (temperature > 30)
{
        Console.WriteLine("It is a hot day");
}
else if (temperature > 20)
{
        Console.WriteLine("It is a cool day");
}
else
{
        Console.WriteLine("It is a COLD day");
}
```

The series of nested IF/ELSE statements in this example provides a technique of evaluating different temperature levels, BUT could become unwieldy when too many levels are required. The SWITCH/CASE statement is a better alternative.

Programming Activity 17

Replace the inner nested IF/ELSE statements from the previous example with the following code,

```
int range = (temperature / 10) * 10;
switch (range)
{
        case 40 : Console.WriteLine("It is a VERY hot day");
              break;
        case 30: Console.WriteLine("It is a hot day");
              break;
        case 20: Console.WriteLine("It is a cool day");
              break;
        case 10: Console.WriteLine("It is a COLD day");
              break;
        case 0: Console.WriteLine("It is a FREEZING day");
              break;
        // Break goes here
}// end of SWITCH
```

The SWITCH/CASE statement uses discrete values for each alternative; this means we need to modify the input with a simple integer division and multiplication. What about values outside the range? To deal with these values we can add a default option at the end of the CASE statements, this will catch and report these instances. Add the default code after the last case statement.

```
default : Console.WriteLine("That value does not compute!");
      break;
```

This example demonstrates how to use the SWITCH/CASE statement to display a message based on a temperature value input by the program user. However, in order to test this program for each temperature range it must be run several times. This problem can be solved by using a loop, which is the topic in the next section.

Iteration

The iteration construct causes a group of one or more program statements to be invoked repeatedly until some end condition is met. Typically such constructs are used to step through arrays or linked lists. There are three types of iteration constructs:

WHILE statements
DO / WHILE statements
FOR statements

In addition to these three constructs we could add recursion (a routine/method calls itself). Recursion is not used in this text, although it is the principal program construct used to achieve repetition in logic and functional languages, thus we will confine ourselves in the following discussion to fixed and variable count loops.

The WHILE loop is a pre-test loop that will continue looping until an exit condition is satisfied. Therefore the condition must be true before the loop body is executed; add the following code to a console application and run,

Programming Activity 18

```
int stop;
int loopCounter = 0;
Console.WriteLine("Enter 1 to start and 0 to exit");
if ((int.TryParse(Console.ReadLine(), out stop)) && stop != 0)
        stop = 1;
while (stop != 0)
{
        Console.WriteLine("This is loop : " + loopCounter);
        loopCounter++;
        Console.WriteLine("Enter 1 to continue and 0 to exit");
        if ((int.TryParse(Console.ReadLine(), out stop)) && stop != 0)
        stop = 1;
}
```

In the above code example, the condition is any expression that returns a Boolean result – true or false. It determines how long the loop body will be repeated and is called the loop condition. In this example the loop body is the programming code executed after each iteration of the loop, i.e. whenever the input condition is true. This loop can be wrapped around the previous temperature code so the program will loop multiple times.

Programming Activity 19

The code below demonstrates this process. Add the following code to a console application and run,

```
int stop;
int temperature;
bool keyBoardInput;
Console.WriteLine("Enter 1 to start and 0 to exit");
// Compound statement to get input and test if not zero
if ((int.TryParse(Console.ReadLine(), out stop)) && stop != 0)
    stop = 1;
```

```
// Start of WHILE Loop
while (stop != 0)
{
    Console.Write("What is the temperature outside? ");
    keyBoardInput = int.TryParse(Console.ReadLine(), out temperature);
    if (keyBoardInput)
    {
       int range = (temperature / 10) * 10;
       switch (range)
       {
          case 40: Console.WriteLine("It is a VERY hot day");
              break;
          case 30: Console.WriteLine("It is a hot day");
              break;
          case 20: Console.WriteLine("It is a cool day");
              break;
          case 10: Console.WriteLine("It is a COLD day");
              break;
          case 0: Console.WriteLine("It is a FREEZING day");
              break;
          default: Console.WriteLine("That value does not compute!");
              break;
       }
       Console.WriteLine("Temp " + range);
    }
    else
    {
       Console.WriteLine("Please enter an Integer Value");
    }// ELSE keyBoardInput

    // Test if loop is to be repeated
    Console.WriteLine("Enter 1 to continue and 0 to exit");
    if ((int.TryParse(Console.ReadLine(), out stop)) && stop != 0)
    stop = 1;
}// End of WHILE Loop
```

In this example the loop will continue for the input of any non-zero integer value. The program has another major fault, there is no error trapping, this topic will be covered later in the text.

The DO/WHILE statement is a test after loop and is similar to the WHILE loop except the loop body will always execute once. The DO/WHILE loop checks the condition after each execution of its loop body. This type of loop is called a post-test loop.

Programming Activity 20

Add the following code to a console application and run,

```
int menuOption;
// Start of DO/WHILE Loop
do
```

```
{
    Console.WriteLine("Simple MENU");
    Console.WriteLine("1 : Add Record");
    Console.WriteLine("2 : Delete Record");
    Console.WriteLine("3 : Display Record");
    Console.WriteLine("0 : EXIT");
    Console.Write("Enter your choice :");
    if ((int.TryParse(Console.ReadLine(), out menuOption))
                                          && menuOption != 0)
    {
       Console.Clear();
       switch (menuOption)
       {
        case 1: Console.WriteLine("Menu Option ONE");
           break;
        case 2: Console.WriteLine("Menu Option TWO");
           break;
        case 3: Console.WriteLine("Menu Option THREE");
           break;
        default: Console.WriteLine("That value does not compute!");
           break;
       }
    }
    else
    {
       Console.WriteLine("The program will Exit");
       menuOption = 0;
    }
}// end of DO/WHILE
while (menuOption != 0);
```

In this example we use the DO/WHILE loop to display a menu and then read the keyboard input to determine a course of action. The loop is terminated when the user enters an integer value of zero.

The last of the loops is the FOR loop which is referred to as a fixed loop. In the previous example the number of iterations was determined by the user, but in the FOR loop the number of iterations is fixed at run time and cannot be altered. The FOR loop is slightly more complicated than WHILE and DO/WHILE loops but they can solve more complicated tasks with less code. The FOR loop construct has the following elements;

```
for (initialization; condition; update)
{
        loop's body;
}
```

This construct contains an initialization, condition and update for the loop variables.

Programming Activity 21

Add this code to a new console application and run,

```
// Display number from 0 to 9
for (int i = 0; i < 10; i++)
{
        Console.WriteLine("loop number : " + i);
}
```

The initialisation is done with,

```
int i = 0;
```

the termination condition is,

```
i < 10;
```

while the update is,

```
i++
```

The FOR loop is use to traverse an Array or List of data items, but generally it is used where the number of iterations is known and fixed. We will cover Arrays in the next chapter and use the FOR loop extensively to read and write data into Array elements.

Next, add this code to see a number of mathematical series,

This code will find all the prime numbers between 2 and 100. Notice the lack of brackets. When there is only one statement in the FOR body the bracket can be omitted.

Programming Activity 22

```
// Local variable definition
int i, j;
for (i = 2; i < 100; i++)
{
        for (j = 2; j <= (i / j); j++)
                if ((i % j) == 0) break; //if factor found, not prime
        if (j > (i / j))
                Console.WriteLine("{0} is prime ", i);
}
```

The following code will display a table of numbers from 10 to 100

Programming Activity 23

```
// Local variable
int sum;
for (int y = 1; y <= 10; y++)
```

```
{
       for (int x = 1; x <= 10; x++)
       {
              Console.Write((sum = x * y) + "\t");
       }
       Console.WriteLine();
}
```

In this last example we will display all four-digit numbers of the type ABCD, where: A+B = C+D. These are referred to as Lucky Numbers.

Programming Activity 24

```
int AB, CD = 0;
for (int a = 1; a <= 9; a++)
{
 for (int b = 0; b <= 9; b++)
 {
  for (int c = 0; c <= 9; c++)
  {
   for (int d = 0; d <= 9; d++)
   {
    if ((a + b) == (c + d))
    {
     Console.WriteLine("{0}{1}{2}{3}" , a , b, c ,d);
     // place markers {0}(1),etc put the variables a,b,etc into text string.
     AB = a + b;
     CD = c + d;
     Console.WriteLine(AB + " = " + a + " + " + b + " : " + CD +
                                         " = " + c + " + " + d);
     // Same as below.
     //Console.WriteLine("{0} = {1} + {2} : {3} = {4} + {5}",
                                             AB, a, b, CD, c, d);
    }
   }
  }
 }
}
```

We have finished with our examination of the C# code and the various constructs that create basic programs. The previous activities are useful to practice your skills and programming logic.

Optional Activities

For each activity write the appropriate console code. Possible solutions are located in Appendix E.

Act-5: Test if number is equal to a target value: using IF statement

Act-6: Test if number is equal to a target value: using IF-ELSE statement

Act-7: Test if a number is equal to a target value: using the TERNARY operator

ACT-8: Use a sequence of IF statements to test an input value

Act-9: Use a sequence of IF statements to test an input value with compound condition for error trapping, Use AND (&&) operator

Act-10: Re-write Act-9 using IF-ELSE statements

Act-11: Use OR (||) operator in IF statement condition

Act-12: Re-write Act-10 as a SWITCH-CASE statement

Act-13: Modify Act-12 to use "fall throu" CASE option

Act-14: Simple menu using CASE "fall throu" and CHAR input

Act-15: Loop for a user defined number of times

Act-16: Put Act-14 inside a DO-WHILE user defined loop

Assessment Activity AT1.1

You should now attempt Assessment Portfolio AT1.1 which consists of 3 programming questions. Ensure you add the following comments at the top of all your code. The assessment can be downloaded from e-campus.

```
// Your Name
// Student ID
// Date
// Question Number
```

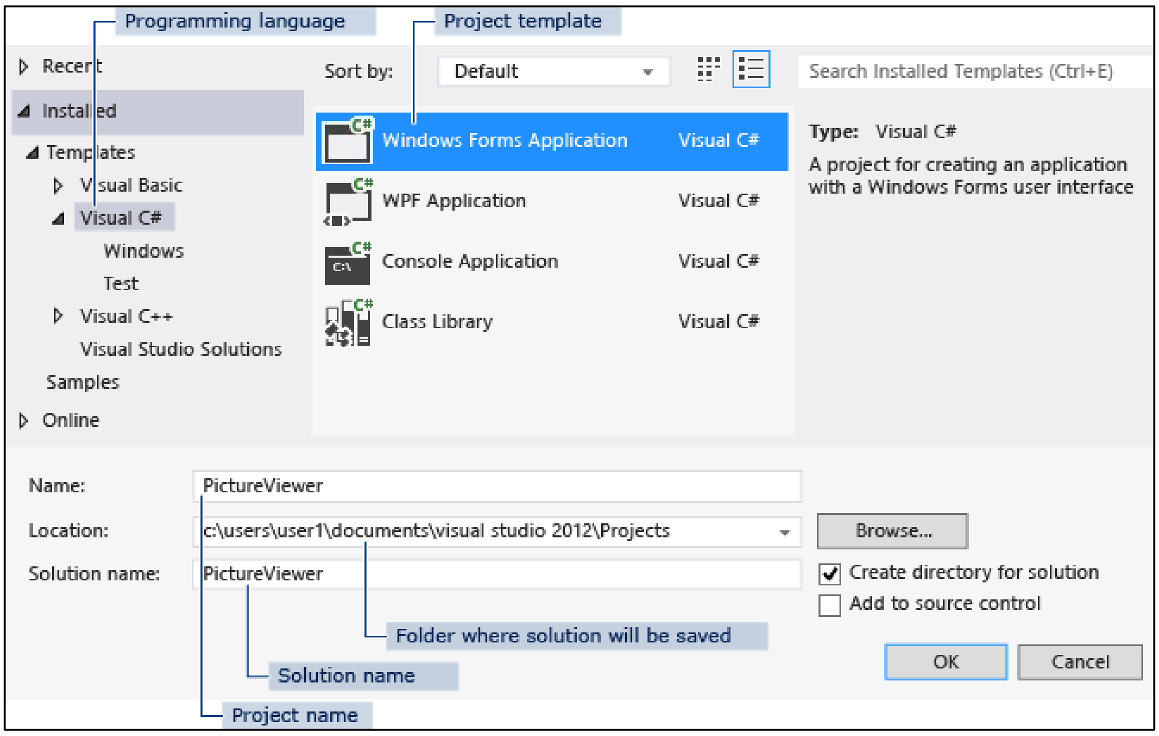
CHAPTER THREE

In this chapter you will use programming constructs within the Widows Forms Applications. You will continue to use the basic C# language syntax and layout from CHAPTER TWO to execute a simple program using the three programming constructs. At the end of the chapter you will have:

- Applied basic language syntax rules,
- Use language data types, operators and expressions to create clear and concise code,
- Use appropriate language syntax for sequence, selection and iteration constructs.

Windows Forms

In this chapter we move from the Console Application to the Windows Forms Application. The underlying code remains the same but our interface will be through a Windows Form with various controls and input fields. To begin, open a new Visual Studio Project, and from the Visual C# menu select the Windows Forms Application. You may need to change the Name, Location and Solution name entries at the bottom of the New Project dialog window (Figure 2).



Type: Visual C#
A project for creating an application with a Windows Forms user interface

Windows Forms Application	Visual C#
WPF Application	Visual C#
Console Application	Visual C#
Class Library	Visual C#

Name: PictureViewer
Location: c:\users\user1\documents\visual studio 2012\Projects
Solution name: PictureViewer
[✓] Create directory for solution
[ ] Add to source control

Figure 2

Visual Studio creates a solution for your program. A solution acts as a container for all of the projects and files needed by your program. Once that has been accomplished, click OK. You should see a blank Windows Form as shown below in Figure 3. If you close or remove any of the windows in Visual Studio you can restore the default window layout by; on the menu bar, choosing Window, Reset Window Layout.

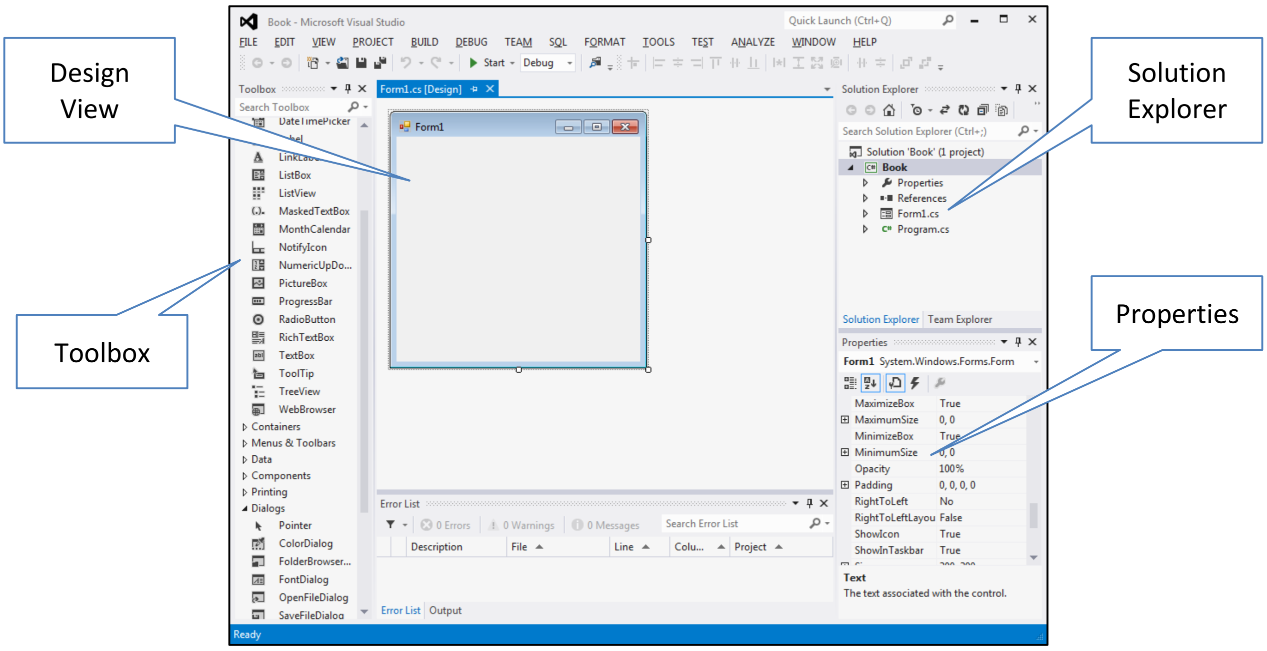
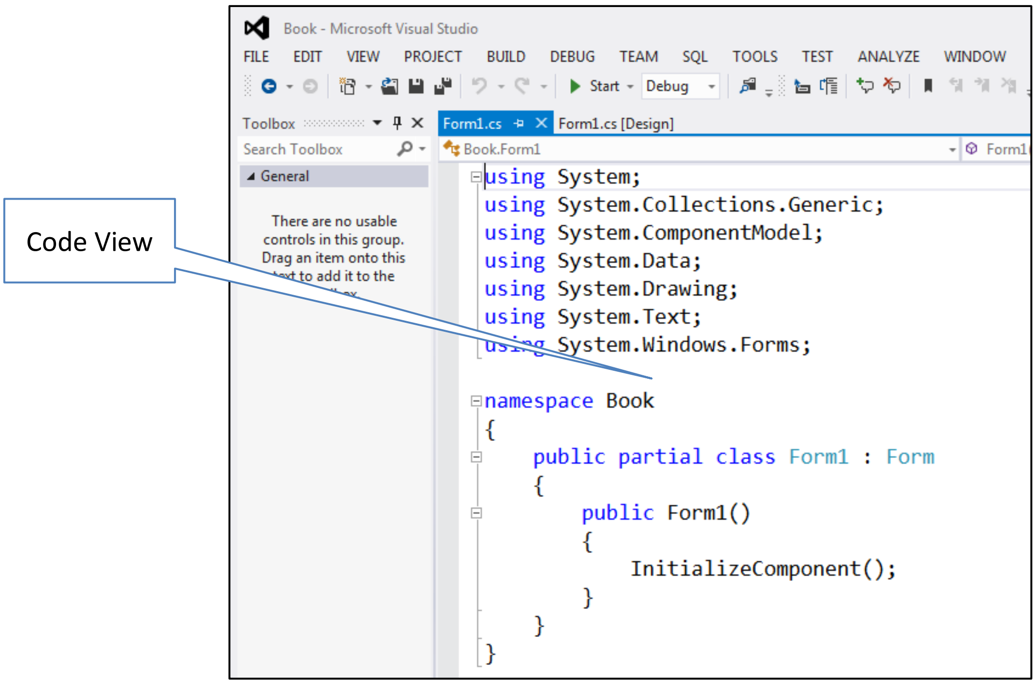


Figure 3

In the Design view you will add various elements from the toolbox located on the left side of the screen. These components include; Label, TextBox, Button and ListBox.

To view the code associated with the form, simply click on the form to select it, and then press F7. A second tab will open that has the C# code associated with the form, as shown below in Figure 4.


Code View
Book - Microsoft Visual Studio
Form1.cs
Form1.cs [Design]
Book.Form1
Toolbox
General
There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

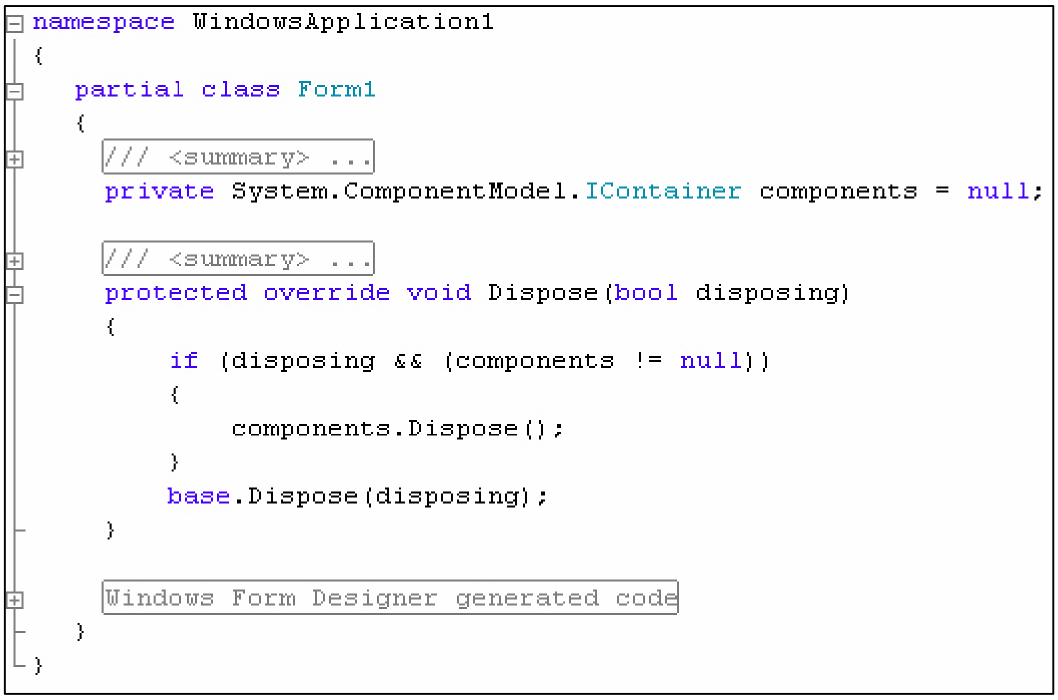
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

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

Figure 4

In the code view you will add the functionality to the Windows Form which connects the design components to real C# programming code. To run the program press the F5 key. This will launch the application and display your form and execute the underling C# code.

Some Background Information

The C# code in the Code View indicates it is a partial class; it's partial because some code is hidden from you. To see the rest of it (which we don't need to alter), click the plus or arrow symbol next to **Form1.cs** in the Solution Explorer and then double click **Form1.Designer.cs** (refer Figure 5)



```
namespace WindowsApplication1
{
    partial class Form1
    {
        /// <summary> ...
        private System.ComponentModel.IContainer components = null;

        /// <summary> ...
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        Windows Form Designer generated code
    }
}
```

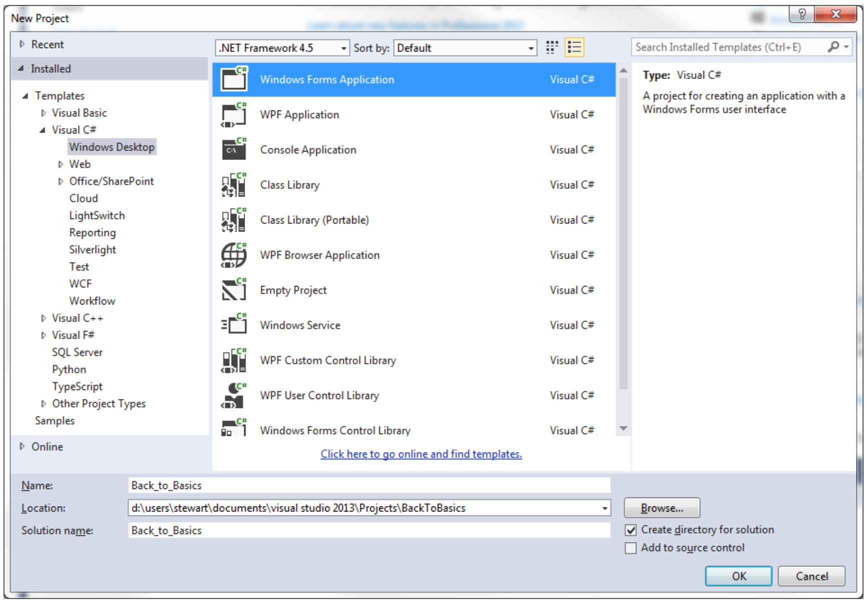
Figure 5

Click on the + symbol to expand the code, the InitializeComponent is code (a Method) that is automatically generated for you when you create a new Windows Application project. As you add things like buttons and text boxes to your form, more code will be added here for you. You don't need to do anything in this window, so you can close this tab.

Programming Activity 25

In this activity we will revisit the three constructs from the previous chapter but the input and output of information will be via the Windows Form.

Open a new project and fill in the Name, Location and Solution name as shown in Figure 6 (you can to use your own naming conversion but don't use the defaults).


New Project
Recent
Installed
Templates
Visual Basic
Visual C#
Windows Desktop
Web
Office/SharePoint
Cloud
LightSwitch
Reporting
Silverlight
Test
WCF
Workflow
Visual C++
Visual F#
SQL Server
Python
TypeScript
Other Project Types
Samples
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Class Library Visual C#
Class Library (Portable) Visual C#
WPF Browser Application Visual C#
Empty Project Visual C#
Windows Service Visual C#
WPF Custom Control Library Visual C#
WPF User Control Library Visual C#
Windows Forms Control Library Visual C#
Click here to go online and find templates.
Browse...
OK
Cancel

Type: Visual C#

A project for creating an application with a Windows Forms user interface

Name: Back_to_Basics

Location: d:\users\stewart\documents\visual studio 2013\Projects\BackToBasics

Solution name: Back_to_Basics

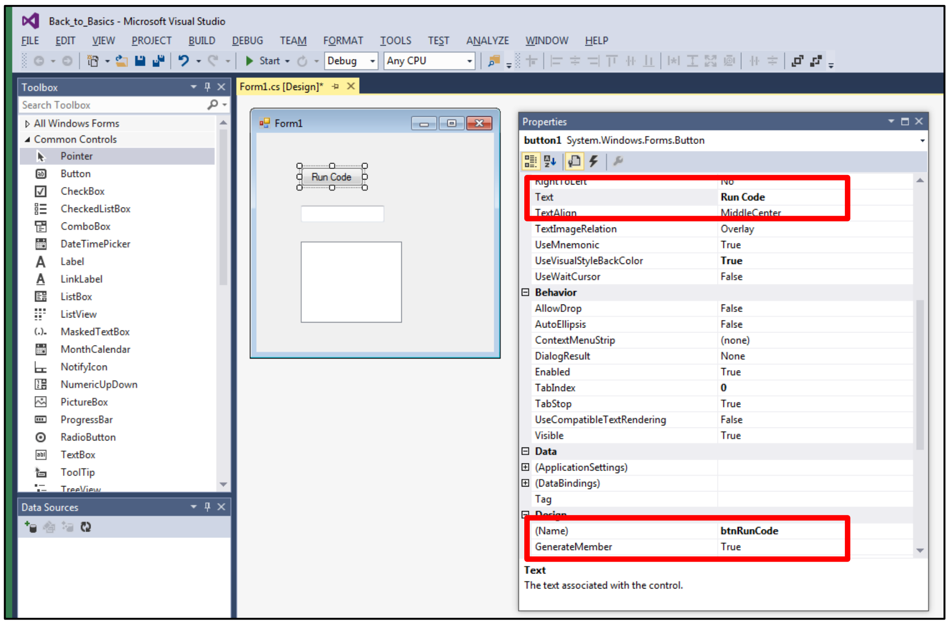
☑ Create directory for solution

☐ Add to source control

Figure 6

From the toolbox add a Button, TextBox and ListBox, then in the Properties widow change the following (refer Figure 7);

Text : Run Code
(Name) : btnRunCode



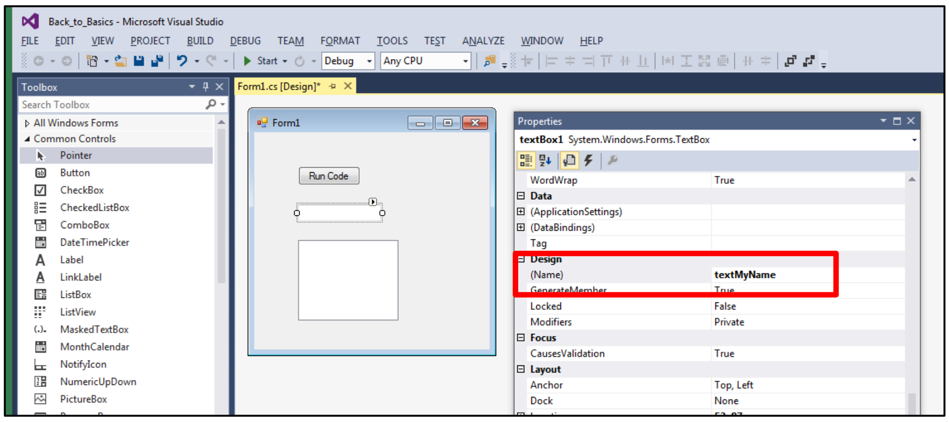
button1 System.Windows.Forms.Button

RightToLeft	No
Text	Run Code
TextAlign	MiddleCenter
TextImageRelation	Overlay
UseMnemonic	True
UseVisualStyleBackColor	True
UseWaitCursor	False
Behavior	
AllowDrop	False
AutoEllipsis	False
ContextMenuStrip	(none)
DialogResult	None
Enabled	True
TabIndex	0
TabStop	True
UseCompatibleTextRendering	False
Visible	True
Data	
(ApplicationSettings)	
(DataBindings)	
Tag	
Design	
(Name)	btnRunCode
GenerateMember	True

Text
The text associated with the control.

Figure 7

Now change the properties of the TextBox as shown in Figure 8,

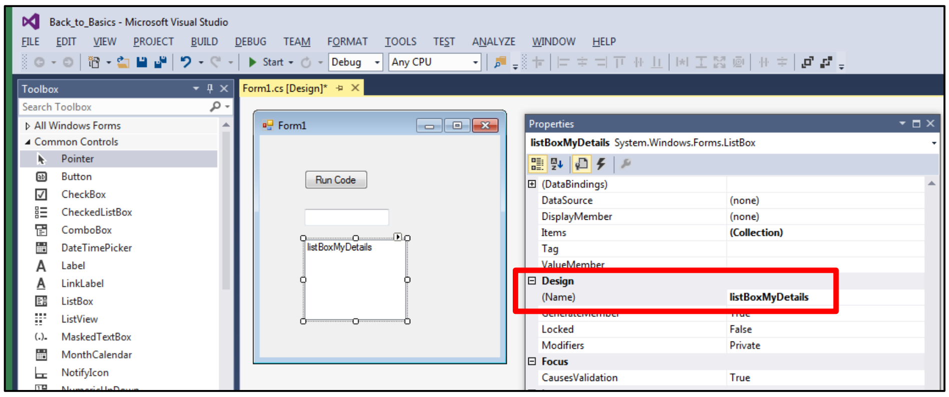


textBox1 System.Windows.Forms.TextBox

WordWrap	True
Data	
(ApplicationSettings)	
(DataBindings)	
Tag	
Design	
(Name)	textMyName
GenerateMember	True
Locked	False
Modifiers	Private
Focus	
CausesValidation	True
Layout	
Anchor	Top, Left
Dock	None

Figure 8

Finally, change the (name) of the ListBox to listBoxMyDetails as shown in Figure 9,

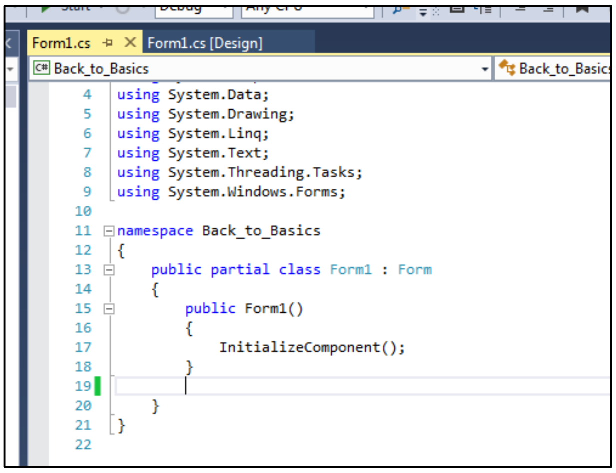


listBoxMyDetails System.Windows.Forms.ListBox

(DataBindings)	
DataSource	(none)
DisplayMember	(none)
Items	(Collection)
Tag	
ValueMember	
Design	
(Name)	listBoxMyDetails
GenerateMember	True
Locked	False
Modifiers	Private
Focus	
CausesValidation	True

Figure 9

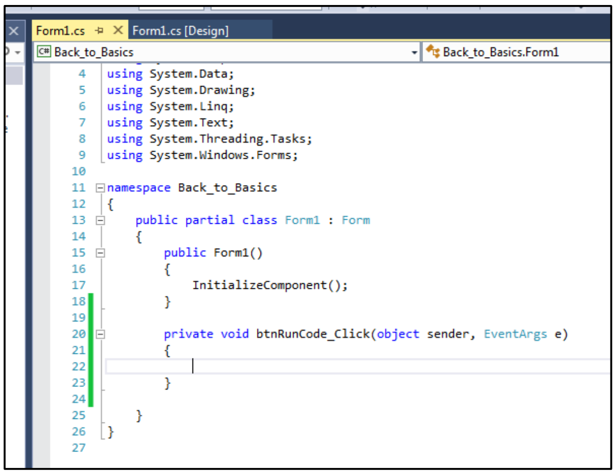
Open the Code View and look at the code, you will only see the basic form as shown in Figure 10,



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

namespace Back_to_Basics
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

    }
}
```

Figure 10

To add an event to the button we must tab back to the Design View. When you have the Design View open double click the Button, this will add a click event in the Code View (refer Figure 11).



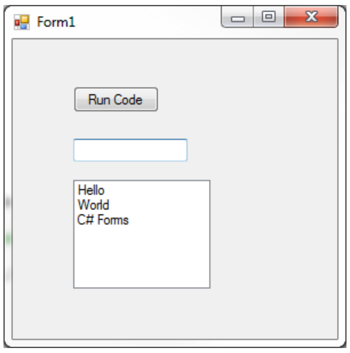
```
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Back_to_Basics
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnRunCode_Click(object sender, EventArgs e)
        {

        }

    }
}
```

Figure 11

We will get the input from the TextBox and display this in the ListBox. The following code will accomplish this task; notice how we use the names of the form objects (Button, TextBox and ListBox) to link to the C# code. Add this code inside the btnRunCode_Click method,

```
private void btnRunCode_Click(object sender, EventArgs e)
{
        // Get the text from the TextBox and add it
        // to the ListBox
        listBoxMyDetails.Items.Add(textMyName.Text);
        // Clear the TextBox for more data
        textMyName.Clear();
}
```

Run this program (F5) and add text into the TextBox and click the Button.



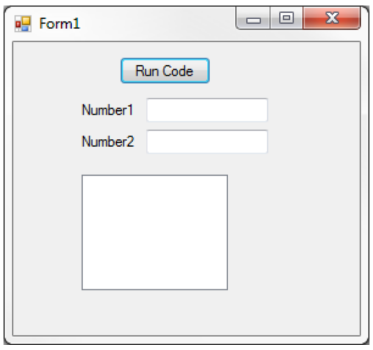
Naming conventions are important to ensure your code is easy to read and understand. This will also help you to identify errors in your code and logic.

The Selection Construct using Forms

As we learnt in Chapter One the selection construct will evaluate a condition and execute the body code when true. Let's create a simple form that will take two integer inputs and perform some basic mathematics.

Programming Activity 26

Create a Windows Forms Application and then add the following components to the form,



Button: btnRunCode
Label: Number1
Label: Number2
TextBox: textNumber1
TextBox: textNumber2
ListBox: listOutput

We will declare several variables to hold the text input and then convert these into integers. Once this is accomplished we will perform various maths based operations and display these results in the ListBox. Add the following code the btnRunCode_Click method and then run the application,

```
// Clear the ListBox for output
listOutput.Items.Clear();
// Declare several variables
int num1 = 0;
int num2 = 0;
int sum = 0;
int product = 0;
// Test if the both TextBoxes have integer values
if (!((int.TryParse(textNumber1.Text, out num1)) &&
(int.TryParse(textNumber2.Text, out num2))))
{
       // Report error message
       MessageBox.Show("Thats not an integer");
}
else
{
       // Compare the two integers and output the greater
       if (num1 > num2)
       {
              listOutput.Items.Add("Number 1 is greater");
       }
       else
       {
```

```
                listOutput.Items.Add("Number 2 is greater");
        }
        // Calculate the sum and product
        sum = num1 + num2;
        product = num1 * num2;
        // Output the sum and product
        listOutput.Items.Add("The sum is " + sum);
        listOutput.Items.Add("The product is " + product);
}
// Clear the TextBoxes
textNumber1.Clear();
textNumber2.Clear();
```

In this example the user can enter two integers and the Button Click method will run the code and display the results. The MessageBox is an appropriate way to report error messages, this object can be customised with additional parameters, a full list can be found on the Visual Studio web site.

Programming Activity 27

In the next example we will create a simple form with a series of radio buttons, so that when the code is run a mathematical operation will be performed on two integers.

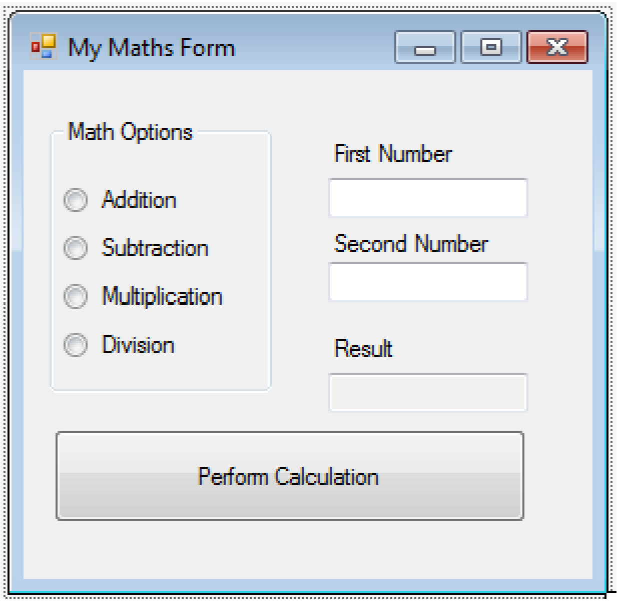


Figure 12

Create a new Windows Forms Application and add the following components to a blank form and adjust their properties;

Form Text : My maths Form
GroupBox Text : Math Options (in Containers)

GroupBox (Name) : grpMathOptions
RadioButton Text : Addition RadioButton (Name) : rdoAddition
RadioButton Text : Subtraction RadioButton (Name) : rdoSubtraction
RadioButton Text : Multiplication RadioButton (Name) : rdoMultiplication
RadioButton Text : Division RadioButton (Name) : rdoDivison
Label Text : First Number TextBox : textNumber1
Label Text : Second Number TextBox : textNumber2
Label Text : Result TextBox : textResult TextBox ReadOnly : True
Button Text : Perform Calculation Button (Name) : btnCalculate

Now create the Click method for the Button and add the following code inside,

```
private void btnCalculate_Click(object sender, EventArgs e)
{
    int num1, num2 = 0;
    string calculationType = "";
    if (!((Int32.TryParse(textNumber1.Text, out num1)) &&
               (Int32.TryParse(textNumber2.Text, out num2))))
    {
        MessageBox.Show("You must enter an integer, please try again");
        return;
    }

    foreach (RadioButton rdo in grpMathOptions.Controls)
    {
        if (rdo.Checked == true)
            calculationType = rdo.Text;
    }

    switch(calculationType)
    {
        case "Addition": textResult.Text = (num1 + num2).ToString();
            break;
        case "Subtraction": textResult.Text = (num1 - num2).ToString();
            break;
        case "Multiplication": textResult.Text = (num1 * num2).ToString();
            break;
        case "Division": textResult.Text =
                                  ((Double)num1/(Double)num2).ToString();
            break;
    }
}// end of btnCalculate_Click
```

In this example the SWITCH/CASE does not require a default entry because we have filtered the input by using the four RadioButton options. The second point to observe is the CASE options; they are four strings that are the same as the RadioButton Text. The division option uses the (Double) key word to perform real division on the two integer values. If this key word is removed the program will run, but the results of the integer division will not yield correct results.

The Iteration Construct using Forms

In this section we will examine the WHILE loop and the fixed iterative FOR loop. In both cases we will create a simple Windows Form to demonstrate how these constructs work.

Programming Activity 28

A while loop will start looping and continue until some condition has changed, to demonstrate this, create a new Windows Form Application and add the following components,

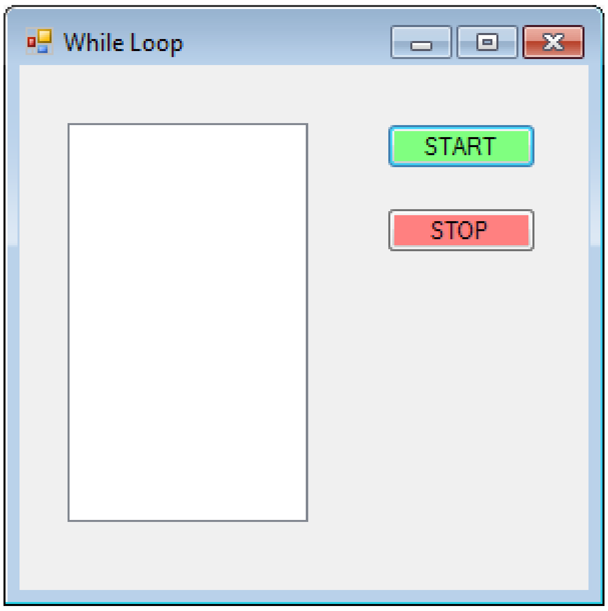


Figure 13

Form Text : While Loop
ListBox (Name) : listOutput
Button Text : START Button (Name) : btnStart Button BackColor : Custom - Green

Button Text : STOP
Button (Name) : btnStop
Button BackColor : Custom - Red

The code to run this form is controlled by the two buttons; double click each button to create the Click methods. Open the Code View and insert the following code,

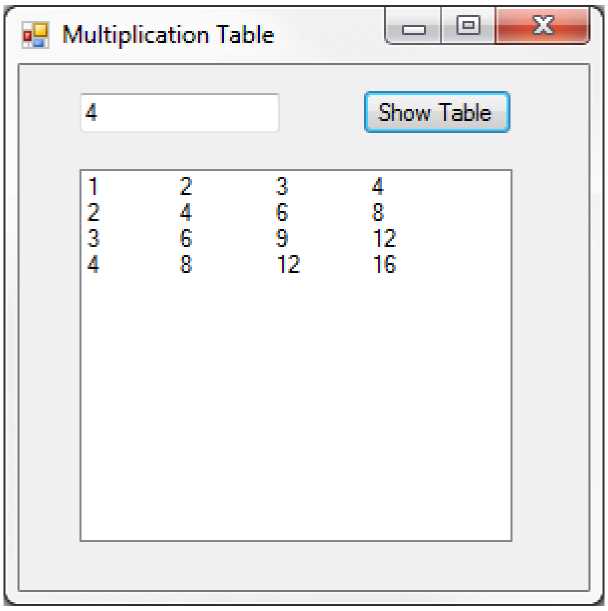
```
// Global variable to end loop
bool endLoop = true;

private void btnStart_Click(object sender, EventArgs e)
{
        int counter = 0;
        listOutput.Items.Clear();

        while (endLoop)
        {
                counter++;
                listOutput.Items.Insert(0, "Loop number " + counter);
                Application.DoEvents();
                // requires System.Threading libraries
                Thread.Sleep(500);
        }
        endLoop = true;
}// end of btnStart_Click

private void btnStop_Click(object sender, EventArgs e)
{
        endLoop = false;
        listOutput.Items.Insert(0, "Loop stopped");
}// end of btnStop_Click
```

Add using
{ } System.Threading

This application will run the loop body when the Start Button is clicked, and pause for 500 milliseconds during each loop. To stop the program you will need to click the stop button (this may take several attempts as the stop event can only be executed after the Thread is awake). This is a crude method of pausing the program BUT it is used to demonstrate the WHILE loop construct. The global variable is declared outside the two Button methods and set to TRUE so the program will be primed to run once the Start Button is clicked.

We will not be examining the DO/WHILE loop as it is simply the reverse of the WHILE loop and is not generally used.

The FOR loop has many applications and is used extensively to traverse Arrays, Lists and Collections of data. In the following example we will demonstrate the FOR loop to create a simple multiplication table.



```
1       2       3       4
2       4       6       8
3       6       9       12
4       8       12      16
```

In this application the user enters an integer value and clicks the Show Table button to display the results in the ListBox. The multiplication table is created by nesting two FOR loops, one to traverse the columns and one to list the rows. Create a new Windows Forms Application and configure the following components,

Programming Activity 29

Form Text : Mulitiplication Table
TextBox (Name) : textInput
Button Text : Show Table Button (Name) : btnShowTable
ListBox (Name) : listOutput

Now add the following code,

```
private void btnShowTable_Click(object sender, EventArgs e)
{
    // Variables for input and sum
    int num1, sum;
    // Variable to hold a row of values
    string oneRow = "";
    // Test if input is an integer
    if (!(Int32.TryParse(textInput.Text, out num1)))
    {
        MessageBox.Show("You must enter a number, please try again");
        return;
    }// end of IF
    // Clear the ListBox
    listOutput.Items.Clear();
    // Outer loop for each ROW
    for (int row = 1; row <= num1; row++)
    {
        // Inner loop for each column
        for (int col = 1; col <= num1; col++)
        {
            // Calculate the sum
            sum = row * col;
            // Add the value to end of the sting variable
            oneRow = oneRow + sum + "\t";
        }// end of Inner FOR loop
        // Display one row of values
        listOutput.Items.Add(oneRow);
        // Clear the string for the next row
        oneRow = "";
    }// end of Outer FOR loop
}// end of btnShowTable_Click
```

In this example the calculation could be changed to addition, subtraction or any other data processing operation. The limits of the table are confined to the size of the ListBox, which could be altered by changing the properties and adding scroll bars or auto resize.

Assessment Activity AT1.2

You should now attempt Assessment Portfolio AT1.2 which consists of 2 programming questions using Windows Forms Application. Ensure you add comments at the top of all your code and for each method block. The assessment can be downloaded from e-campus.

CHAPTER FOUR

In this chapter you will use data structures to collect, process and display data. The learning outcomes are to code and execute a program using an array to manipulate data and define data structures that allow a user to input values that the computer will process and then display the results. At the end of the chapter you will have:

- Demonstrated an understanding of data structures,
- Written code to create and manipulate arrays,
- Design, define and use data structures.

Arrays

In this section we will examine Arrays and how to work with a sequence of elements of the same data type. We will explain what Arrays are, how they are declared, created and instantiated. The examples demonstrated will include both one-dimensional and multidimensional Arrays. Finally, we will use FOR loops to iterate through the array, to read and write data.

Arrays are vital for most programming languages. They are collections of variables, which are stored in cells we call elements:

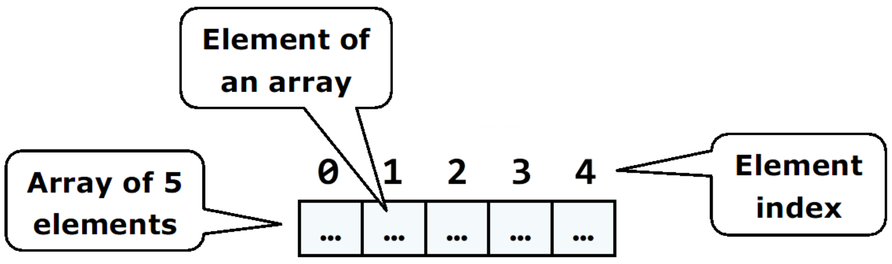


Figure 14

Array elements in C# are numbered with 0, 1, 2, … N-1. Those numbers are called indices. The first index of all Arrays is number 0; the total number of elements in a given Array is called the length of an Array. The Array in Figure 14 has 5 elements or size =5.

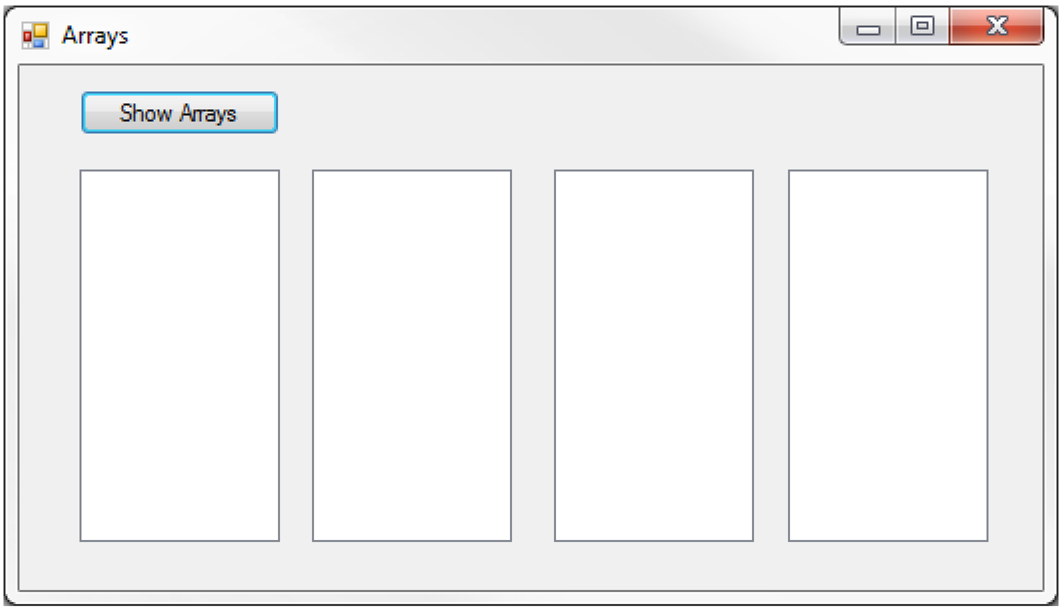
All elements of a given Array are of the same type, no matter whether they are primitive or reference types. This allows us to represent a group of similar elements as an ordered sequence or work on them as a whole.

Arrays can be in different dimensions, but the most common are the one-dimensional and the two-dimensional Arrays. One-dimensional Arrays are also called vectors and two-dimensional Arrays are also known as matrices.

In C# the Arrays have fixed length, which is set at the time of their instantiation and determines the total number of elements. Once the length of an Array is set it cannot be changed. An Array is created with the help of the keyword new, which is used to allocate memory. In C# all variables, including the elements of Arrays have a default initial value.

Programming Activity 30

Create a new Windows Forms Application and add the following components, four ListBoxes and one Button.



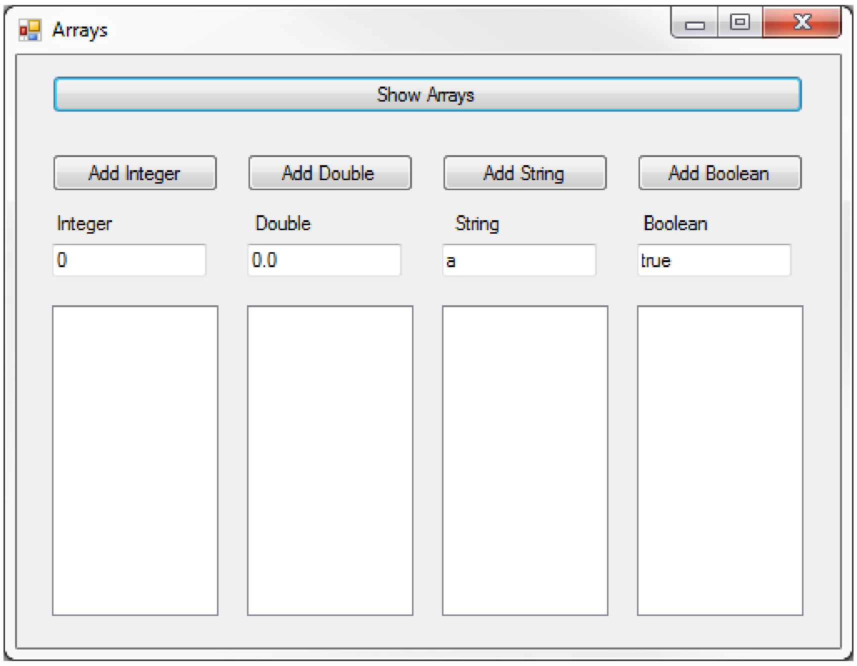
Change the properties as indicated below,

Form Text : Arrays
ListBox (Name) : listInteger
ListBox (Name) : listDouble
ListBox (Name) : listString
ListBox (Name) : listBoolean
Button Text : Show Arrays Button (Name) : btnShow Array

Now add this code to the Button Click method and run.

```
// Maximum number of Array elements
int max = 5;
// Array of Ints, Double, String and Boolean
int[] myIntArray = new int[max];
double[] myDoubleArray = new double[max];
string[] myStringArray = new string[max];
bool[] myBooleanArray = new bool[max];
// Loop through each element and display the default value
for (int i = 0; i < max; i++)
{
    listInteger.Items.Add("index " + i + " Integer " + myIntArray[i]);
    listDouble.Items.Add("Double " + myDoubleArray[i]);
    listString.Items.Add("String " + myStringArray[i]);
    listBoolean.Items.Add("Boolean " + myBooleanArray[i]);
}
```

You will notice that the string value is null or empty while the other data types have a value. Now extend this activity by adding four Labels, four TextBoxes and four Buttons.



Change the properties on each of the components as follows, and add default values to the TextBoxes by entering a value in the Text property.

Form Text : Arrays
ListBox (Name) : listInteger
ListBox (Name) : listDouble
ListBox (Name) : listString
ListBox (Name) : listBoolean
Button Text : Show Arrays Button (Name) : btnShow Array
Label Text : Integer
Label Text : Double
Label Text : String
Label Text : Boolean
TextBox (Name) : textInteger TextBox Text : 0
TextBox (Name) : textDouble TextBox Text : 0.0
TextBox (Name) : textString TextBox Text : A
TextBox (Name) : textBoolean TextBox Text : true

Button Text : Add Integer Button (Name) : btnAddInteger
Button Text : Add Double Button (Name) : btnAddDouble
Button Text : Add String Button (Name) : btnAddString
Button Text : Add Boolean Button (Name) : btnAddBoolean

Because we are using several button methods to manipulate the four arrays the declarations will need to be moved out of the Show Button method. Beware this code does **not** have error trapping and incorrect input will cause the program to fault. This is a simple program that demonstrates the basic Array types. Modify and/or add the following code.

```
// Maximum number of Array elements
static int max = 5;
// Array of Ints, Double, String and Boolean
int[] myIntArray = new int[max];
double[] myDoubleArray = new double[max];
string[] myStringArray = new string[max];
bool[] myBooleanArray = new bool[max];


private void btnShowArray_Click(object sender, EventArgs e)
{
    listInteger.Items.Clear();
    listDouble.Items.Clear();
    listString.Items.Clear();
    listBoolean.Items.Clear();
    // Loop through each element and display the values
    for (int i = 0; i < max; i++)
    {
        listInteger.Items.Add("index " + i + " Integer " + myIntArray[i]);
        listDouble.Items.Add("Double " + myDoubleArray[i]);
        listString.Items.Add("String " + myStringArray[i]);
        listBoolean.Items.Add("Boolean " + myBooleanArray[i]);
    }
}// end of btnShowArray_Click
private void btnAddInteger_Click(object sender, EventArgs e)
{
    for (int i = 0; i < max; i++)
    {
        if (myIntArray[i] == 0)
        {
            myIntArray[i] = int.Parse(textInteger.Text);
            return;
        }
    }
}// end of btnAddInteger_Click
private void btnAddDouble_Click(object sender, EventArgs e)
{
    for (int i = 0; i < max; i++)
```

```
        {
            if (myDoubleArray[i] == 0)
            {
                myDoubleArray[i] = double.Parse(textDouble.Text);
                return;
            }
        }
    }// end of btnAddDouble_Click
    private void btnAddString_Click(object sender, EventArgs e)
    {
        for (int i = 0; i < max; i++)
        {
            if (myStringArray[i] == null)
            {
                myStringArray[i] = textString.Text;
                return;
            }
        }
    }// end of btnAddString_Click
    private void btnAddBoolean_Click(object sender, EventArgs e)
    {
        for (int i = 0; i < max; i++)
        {
            if (myBooleanArray[i] == false)
            {
                myBooleanArray[i] = bool.Parse(textBoolean.Text);
                return;
            }
        }
    }// end of btnAddBoolean_Click
```

In this example the user can input new values into each of the Arrays separately and display the contents. The Array elements are tested if they contain the default value before a new value is added to the element. Once the Array is full no more values can be added. Take a moment to study this code as future programming tasks will rely of a sound knowledge of Arrays.

Data Structures

In this section we are going to examine some of the basic presentations of data in programming, lists and data structures. Very often in order to solve a given problem we need to work with a sequence of elements. Depending on the task, we have to apply different operations on this set of data. Data structures are a set of data organized on the basis of logical and mathematical laws. Very often the choice of the right data structure makes the program much more efficient.

Basic Data Structures

We can differentiate several groups of data structures,

Linear : these include lists, stacks and queues
Tree : binary trees, B-trees and balanced trees
Dictionaries : key-value pairs organized in hash tables
Sets : unordered bunches of unique elements
Others : multi-sets, bags, multi-bags, priority queues, graphs, …

Next we are going to consider the structures "stack" and "queue", as well as their applications. We are going to implementation a queue structure using an Array.

Programming Activity 31

In this example we will simulate a queue of airplanes lining up to take off at a busy airport. As each airplane gets ready to leave, it will radio to the control tower and the air traffic controller will place the airplane in the queue behind other airplanes waiting for their turn to take off. It is a simple concept and is often referred to as First In First Out (FIFO), the queue has a front and back as shown in Figure 15.

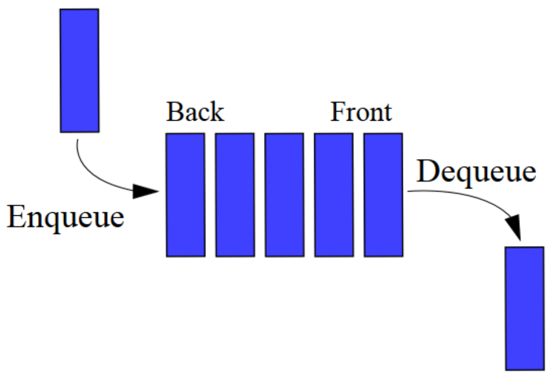
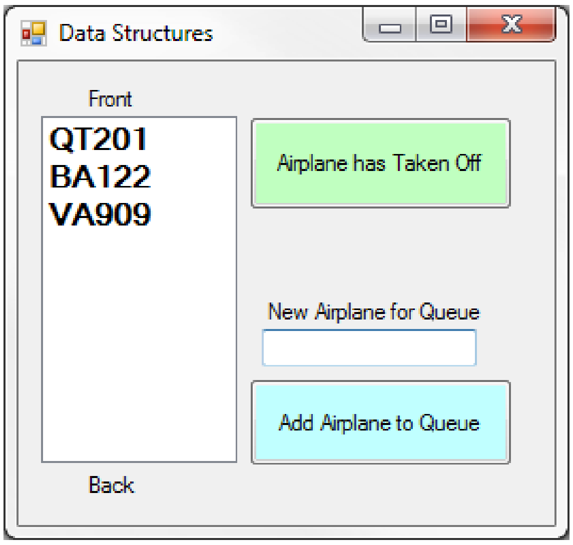


Figure 15

Create a new Windows Forms Application and add the following components, one ListBox, one TextBox and two Buttons. Add the appropriate labels to identify each component.



Form Text : Data Structures
ListBox (Name) : listDisplayQueue ListBox Font Size : 12pt
TextBox (Name) : textAddAirplane
Button Text : Airplane has Taken Off Button (Name) : btnDequeue
Button Text : Add Airplane to Queue Button (Name) : btnEnqueue

Now add the following code,

```
// Max size of array (you can change this)
const int max = 4;
// Array to hold airplane flight numbers
string[] airplaneQueue = new string[max];

private void btnEnqueue_Click(object sender, EventArgs e)
{
    // Loop through the array
    for (int i = 0;i < max; i++)
    {
        // Test if array element has data
        if (String.IsNullOrEmpty(airplaneQueue[i]))
        {
            // Add data from TextBox to array
            airplaneQueue[i] = textAddAirplane.Text;
            // Clear the TextBox
            textAddAirplane.Clear();
            // Display the array
            DisplayQueue();
            return;
        }
    }// end of FOR loop
}// end of btnEnqueue_Click

private void btnDequeue_Click(object sender, EventArgs e)
{
    // Loop through the array
    for (int i = 0; i < max; i++ )
    {
        // Move array element down Array
        if (i != (max - 1))
            airplaneQueue[i] = airplaneQueue[i + 1];
        else
            // Fill last element with null value
            airplaneQueue[i] = null;
    }// end of FOR loop
    DisplayQueue();
}// end of btnDequeue_Click

// Method to display the queue data structure
private void DisplayQueue()
{
    // Clear the ListBox
    listDisplayQueue.Items.Clear();
    // Display each element in the queue
    for (int i = 0; i < max; i++)
    {
        if(airplaneQueue[i] != null)
            listDisplayQueue.Items.Add(airplaneQueue[i]);
    }
}// end of DisplayQueue
```

In this example we declare a constant value for the Array size and then add flight numbers to the Array via the TextBox. When an airplane takes off it is removed from the front of the queue and the remaining airplanes are moved along the queue. It is not necessary to delete the front airplane; simply write the second airplane into this space. When we reach the last space in the queue we write a null value into this space so future values can be added. This program simulates a queue structure; a similar construct could be used to model a queue of numbers or other objects. Take a moment to examine this code as this idea will be used in latter parts of the course.

The other major data structure that will be discussed is the stack, which can also be implemented using an Array. The stack data structure is used to push a series of elements into an Array and is often referred to as Last In First Out (LIFO) as shown in Figure 16.

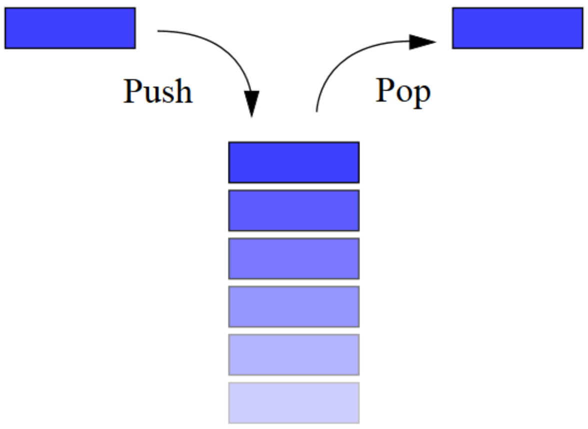


Figure 16

In order to read the data from a single element in the middle of the stack the values on top must be removed or popped. The stack is used in various computer environments and high level mathematics. We will not implement a stack at this point as this structure will be an activity.

Abstract Data Structures

In this section we will introduce the concept of abstract data types (ADT) and will explain how a certain ADT can have multiple different implementations. In the previous activity the program acted as a simple queue (using an Array), with new airplanes being added at the back and airplanes leaving at the front. This represents a basic abstract data structure, and could be implemented using a Queue method from the Collections namespace (refer Appendix D).

In general, abstract data types (ADT) give us a definition (abstraction) of the specific structure, i.e. defines the allowed operations and properties, without being interested in the specific implementation. This allows an abstract data type to have several different implementations and respectively different efficiency. In the next activity we will create a simple List<> of employees which can be managed using several buttons.

Programming Activity 32

The form layout uses two TextBoxes and a DropDown list which are grouped into the employees details (refer Figure 17). There are 5 Buttons and a ListBox inside an Employees group.

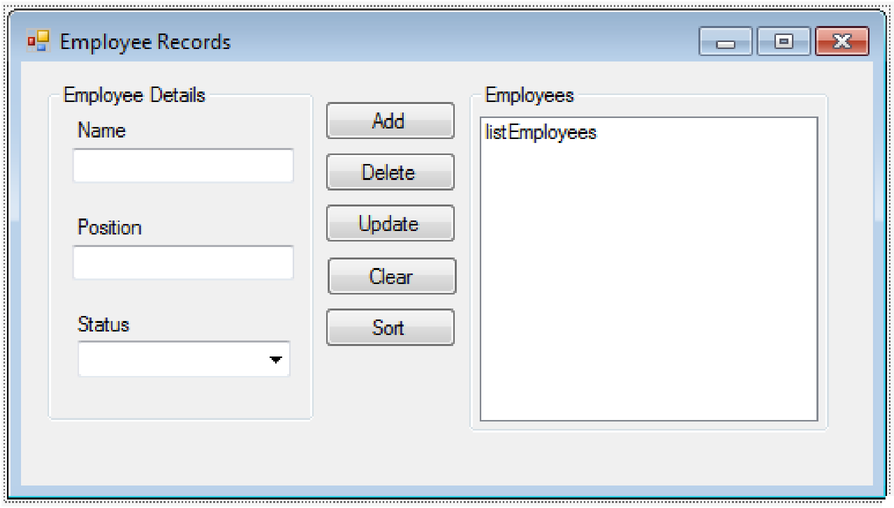
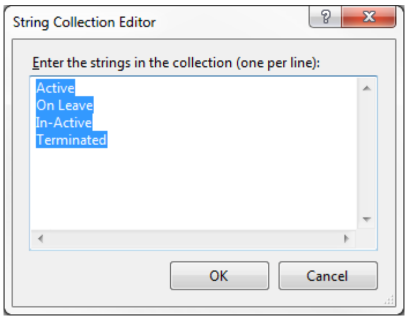


Figure 17

GroupBox grpDetails; Label lblName; TextBox textName; Label lblPosition; TextBox textPosition; Label lblStatus; ComboBox cmbStatus;	Button btnAdd; Button btnDelete; Button btnUpdate; Button btnClear; Button btnSort;	GroupBox grpEmployees; ListBox listEmployees;

The combo box has several items which can be modified in the properties of the ComboBox (select ***Items*** in the properties).



Enter the strings in the collection (one per line):

```
Active
On Leave
In-Active
Terminated
```

At the top of the code inside the class definition declare a new List<>

```
public partial class EmployeeRecordsForm : Form
{
    public EmployeeRecordsForm()
    {
        InitializeComponent();
    }

    // Create an acme_emplyee list of employees
    List<Employee> acme_Employee = new List<Employee>();
```

Button Method to add a new List<> item

```
// Method to add employee data to acme_employee list
// test if each field has data
private void btnAdd_Click(object sender, EventArgs e)
{
    // Local variable to check if text box has data
    bool hasData = true;
    // New employee
    Employee newEmplyee = new Employee();
    // Get employee data from text box
    newEmplyee.fullName = textName.Text;
    newEmplyee.position = textPosition.Text;
    newEmplyee.status = cmbStatus.Text;
    // Check if text box has data
    // If null or empty set hasData to false
    if (String.IsNullOrEmpty(textName.Text))
    {
        MessageBox.Show("Please enter full name");
        hasData = false;
        return;
    }
    if (String.IsNullOrEmpty(textPosition.Text))
    {
        MessageBox.Show("Please enter Position");
        hasData = false;
        return;
    }
    if (String.IsNullOrEmpty(cmbStatus.Text))
    {
        MessageBox.Show("Please enter Status");
        hasData = false;
        return;
    }
    // If all data is valid and there are no duplicates
    // then add the record to the list
    bool duplicateFound = acme_Employee.Exists(x => x.fullName ==
                                                        textName.Text);
    if (hasData && !duplicateFound)
    {
        // Add record to acme employee List<>
        acme_Employee.Add(newEmplyee);
        // Empty the fields for the next record
        ResetDetails();
        // Display the records in the list box
        DisplayRecords();
    }
    else
    {
        MessageBox.Show("thats never going to work");
    }
}
```

Button Method to delete a List<> item

The delete method checks to see which record in the ListBox has been selected and the removes it from the List<>.

```
// Check if a record has been selected from the list box
// then remove that record
private void btnDelete_Click(object sender, EventArgs e)
{
    if (listEmployees.SelectedIndex == -1)
    {
        MessageBox.Show("Select a record from the List Box");
    }
    else
    {
        string curItem = listEmployees.SelectedItem.ToString();
        int indx = listEmployees.FindString(curItem);
        acme_Employee.RemoveAt(indx);
        DisplayRecords();
    }
}
```

Button Method to update/edit the details of a List<> item

```
// Update a single field in a record
private void btnUpdate_Click(object sender, EventArgs e)
{
    Employee updatedEmployee = new Employee();
    updatedEmployee.fullName = textName.Text;
    updatedEmployee.position = textPosition.Text;
    updatedEmployee.status = cmbStatus.Text;
    if (listEmployees.SelectedIndex == -1)
    {
        MessageBox.Show("Select a record from the List Box");
    }
    else
    {
        string curItem = listEmployees.SelectedItem.ToString();
        int indx = listEmployees.FindString(curItem);
        acme_Employee[indx].fullName = updatedEmployee.fullName;
        acme_Employee[indx].position = updatedEmployee.position;
        acme_Employee[indx].status = updatedEmployee.status;
        ResetDetails();
        DisplayRecords();
    }
}
```

Button Method to clear the Textbox fields

```
// Button method to clear all the TextBoxes
private void btnClear_Click(object sender, EventArgs e)
{
    ResetDetails();
}
```

Button Method to sort the List<> by name field

```
// Method that sorts the List<> by fullName uses the
// IComparable<employee> in the employee class
```

```
        // and the CompareTo method
        private void btnSort_Click(object sender, EventArgs e)
        {
            acme_Employee.Sort();
            DisplayRecords();
        }
```

MouseClick Method to populate the TextBox fields when an item in the ListBox is selected

```
        // Method to obtain a selected record from the ListBox
        private void listEmployees_MouseClick(object sender, MouseEventArgs e)
        {
            if (listEmployees.SelectedIndex == -1)
            {
                MessageBox.Show("Select a record from the List Box");
            }
            else
            {
                string curItem = listEmployees.SelectedItem.ToString();
                int indx = listEmployees.FindString(curItem);
                listEmployees.SetSelected(indx, true);
                textName.Text = acme_Employee[indx].fullName;
                textPosition.Text = acme_Employee[indx].position;
                cmbStatus.Text = acme_Employee[indx].status;
            }
        }
```

After the code for each button is created there are several methods that are used throughout the program that assist in a smooth user experience.

Method to display fields in the Form Listbox

```
        // Show all the records in the List<>
        public void DisplayRecords()
        {
            // Clear the list before displaying the records
            listEmployees.Items.Clear();
            // Loop through the List<> and show the fullName and Position
            foreach (var emp in acme_Employee)
            {
                listEmployees.Items.Add(emp.fullName + "\t:  " + emp.position);
            }
        }
```

Method to Reset the Form Textboxes

```
        // Reset the field to empty for the next data input
        private void ResetDetails()
        {
            textName.Text = "";
            textPosition.Text = "";
            // set default value for status
            cmbStatus.Text = "Active";
        }
    }// END OF THE CLASS EmployeeRecordsForm
```

A new class is created to hold the fields associated with each List<> item, the IComparable<employee> provides a mechanism for the Sort method. This code must be outside the partial form class BUT inside the namespace.

Class to hold List item structure

```
// Class of employee fields
    // IComparable<employee> is used for the sort method.
    public class Employee : IComparable<Employee>
    {
        [XmlElement("fullName")]
        public string fullName
        {
            get;
            set;
        }
        [XmlElement("position")]
        public string position
        {
            get;
            set;
        }
        [XmlElement("status")]
        public string status
        {
            get;
            set;
        }

        // Simple compare method to sort by fullName
        public int CompareTo(Employee other)
        {
            return this.fullName.CompareTo(other.fullName);
        }
    }
}// END OF THE NAMESPACE
```

Add using System.Xml.Serialization

Once each of these methods has been implemented and tested you will notice the additional functionality that was added to demonstrate the various methods. This could be removed and hidden from the user; for example the Sort and Clear buttons could be removed and these methods called from within other methods. When a new record is added the display method could sort the list before displaying while the textboxes can be cleared for the next record. You may wish to add code to display the Status in the listbox.

In the next chapter we will investigate how to find a particular item in a large collection of records.

Optional Activities

For each activity write the appropriate console code. Possible solutions are located in Appendix E.

Act-17: Make an array of three element which are pre-filled

Act-18: Use three loops to traverse the array from Act-16; use FOREACH, FOR, WHILE.

Act-19: Create two arrays using two different pre-filled options. Traverse using FOR loop

Act-20: Create an array and fill using FOR loop, display using a second FOR loop

Act 21: Modify Act-19 to provide more input/output details

Act-22: Swap two numbers if first > second using a method call

Act-23: Calculate rectangle using class and methods

Act-24: Use methods to modularise code

Act-25: Use nested loops to display all primes from 1..100

Act-26: Validate integer input using TRY-CATCH

CHAPTER FIVE

In this chapter you will use standard algorithms to search and locate specific items from a collection of organised data. The learning outcomes are to code and execute a simple program using an Array of basic data types. At the end of the chapter you will have:

- Create code for a linear (sequential) search, binary search, insertion and deletion algorithms to operate on arrays
- Code standard sequential access algorithms and random access algorithms

Searching and Sorting

One very common application for computers is storing and retrieving information. For example, the Government stores information about our taxes while Education Departments collect information about students (courses, pass/fail, attendance, etc). As the amount of information to be stored and accessed becomes very large, the computer proves to be a useful tool to assist in this task. Over the years, as computers have been applied to these types of tasks, many techniques and algorithms have been developed to efficiently maintain and process information.

The processes of “looking up” a particular data record in a database is called searching. We will look at two different search algorithms; one very easy to implement, but inefficient, the other much more efficient. As we will see, in order to do an efficient search in a database, the records must be maintained in some order. The process of ordering the records in a database is called sorting. We will use a simple bubble sorting algorithm to demonstrate how records are sorted. Sorting and searching together constitute a major area of study in computer programming.

Linear Search

A linear search does not require the list/database to be sorted as the algorithm will test each element in the list. The linear search, also sometimes referred as brute-force search or sequential search is one of the simplest search algorithms. It starts by checking if the first element of a list matches the target and continues on for each element of the list till a match is found or the end of the list is reached. For a linear search, a data structure like an Array or linked list is the general construct.

Therefore, if the list has 1001 items the worst case scenario will be when the search item is located in the last element. For example if we wanted to search for the value KK in the Array shown in Figure 18 we would need to test all 1001 element before it was found in the last element. However, if we searched for the value AB we would find it located in the first element; this would be the best case scenario.

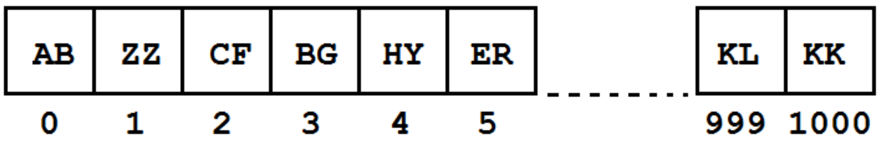


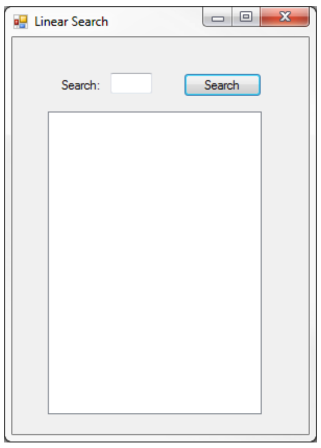
Figure 18

So, given that the worst case performance characteristics of linear search, why would you want to use it? What are some of the scenarios where it is acceptable or even necessary to use this brute force algorithm?

- If your collection size is relatively small and you would be performing this search relatively few times, then this might an option.
- If you have constant insertion performance (like using a linked list) and the search frequency is less.
- A linear search places very few restrictions on the complex data types; when it is only necessary to match one value.

Programming Activity 33

In this activity we will use a random number method to fill an Array of integers. Then we can search the Array to find a match or report an error message if not found. Create a new Windows Form Application and add one TextBox, one Button and one ListBox.



Change the properties of the form components as shown below,

Form Text : Linear Search
ListBox (Name) : listResults
TextBox (Name) : textSearch
Button Text : Search Button (Name) : btnSearch

Now add the following code, but notice the addition of code in the public Form1 method. This code will initialise the Array and fill the elements with random integers.

```
public Form1()
{
    InitializeComponent();
    // Call method to fill Array at start up
    FillArray();
}

// Array of integers
static int max = 50;
int[] myArray = new int[max];

private void btnSearch_Click(object sender, EventArgs e)
{
    int target;
    bool found = false;
    if (!(Int32.TryParse(textSearch.Text, out target)))
    {
        MessageBox.Show("You must enter an integer");
        return;
    }
    listResults.Items.Clear();
    for (int x = 0; x < max; x++)
    {
        listResults.Items.Add(x + " [ " + myArray[x] + " ]");
        // Test if the array element matches the target
        if (myArray[x] == target)
        {
            listResults.Items.Add("Found at index " + x);
            found = true;
            // return;
        }
    }// end of FOR loop
    if (!found)
    {
        MessageBox.Show("Not Found, try again.");
    }
}// end of btnSearch_Click
```

```
// Method to fill Array with random numbers
private void FillArray()
{
    // Create a random number
    Random rand = new Random();
    for (int i = 0; i < max; i++)
    {
        // Add random number 0..100
        myArray[i] = rand.Next(100);
    }
}// end of method FillArray
```

The Array will contain duplicates as the Random method does not check to see if a number has been used before. Furthermore, the search algorithm does not stop searching until it reaches the end of the Array. So you may get more than one result. To fix this problem and report the first instance of the search target you will need to stop the search once you have found the target. Add a return statement after the “found = true;” in the Button method (shown as a comment).

Whilst this example is very simplistic and the number of elements is very small the real performance problems begin to appear when the number of elements is in the 100’s of thousands.

Binary Search

The Binary search adds a small bit of overhead for a large performance gain. The basic criteria for a binary search are; the collection is kept in a sorted manner before the search happens. The concept of a binary search is simple: In a sorted list where each element supports comparability, search for the target element in the middle. If the target value is less, discard the right half of the list and if the target is more, discard the left half. Continue this process with the remaining half till the element is found or the search element does not exist. Whilst the Linear search is a brute force approach the Binary search is more in the category of Divide and Conquer. With each iteration of the search the size of array is halved.

Most modern programming languages have pre-defined methods for sorting and searching, but it is important to understand the underlying algorithms.

Therefore, we will implement a simple Binary search to demonstrate this construct.

Programming Activity 34

In this example the code will create an Array of 20 elements and fill it with random integers between 0 and 100. Use the previous activity as a template and modify the code. There are two methods; the first will fill the Array and second will display the Array. The Binary search is run each time the Button is clicked.

```
        public Form1()
        {
            InitializeComponent();
            // Call method to fill Array at start up
            FillArray();
        }

        // Array of random integers
        static int max = 20;
        int[] myArray = new int[max];

        private void btnSearch_Click(object sender, EventArgs e)
        {
            int mid;
            int lowBound = 0;
            int highBound = max;
            int target;
            if (!(Int32.TryParse(textSearch.Text, out target)))
            {
                MessageBox.Show("You must enter an integer");
                return;
            }
            while (lowBound <= highBound) // Check “<” or “<=”
            {
                // Display list
                ShowArray(lowBound, highBound);
                // Find the mid-point
                mid = (lowBound + highBound) / 2;
                // Pause with a messagebox
                MessageBox.Show("Low:" + lowBound + " Mid:" + mid + " High:" +
highBound);
                if (myArray[mid] == target)
                {
                    // Target has been found
                    listResults.Items.Add("Found at index " + mid);
                    return;
                }
                else if (myArray[mid] >= target)
                {
                    highBound = mid - 1;
                }
                else
                {
                    lowBound = mid + 1;
                }
            }
            MessageBox.Show("Not Found, try again.");
        }// end of btnSearch_Click

        // Method to display Array
        private void ShowArray(int low, int high)
        {
            listResults.Items.Clear();
            for (int i = low; i < high; i++)
            {
                listResults.Items.Add(myArray[i]);
            }
        }// end of method ShowArray

        // Method to fill Array with random numbers
        private void FillArray()
        {
            // Create a random number
            Random rand = new Random();
            for (int i = 0; i < max; i++)
            {
                // Random number 0..100
                myArray[i] = rand.Next(100);
            }
            // Use the build in sort method
            Array.Sort(myArray);
        }// end of method FillArray
    }
```

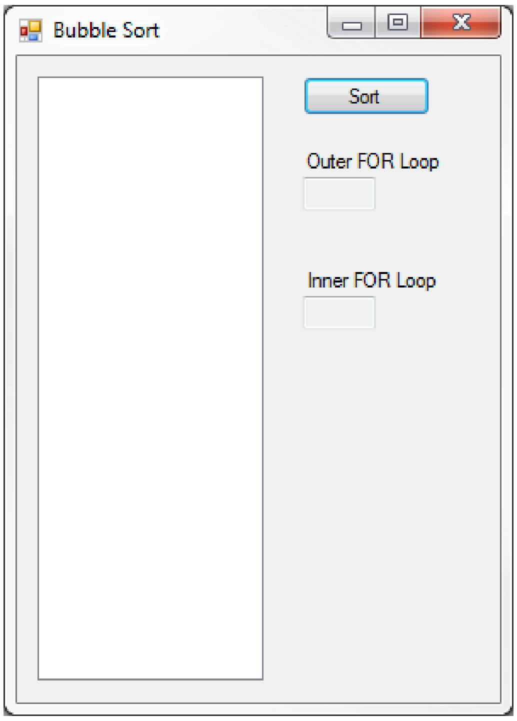
There are some issues with this code; for example if you search for a value outside the range then the program will fault. These types of problems can be fixed by testing the upper and lower values in the List and then filtering the user input. The purpose is to demonstrate how each iteration of the search halves the array. This example also uses a built in method to sort the Array, we will examine a simple sort algorithm in the next section.

Bubble Sort

The simplest sorting algorithm is the bubble sort. The bubble sort works by iterating down a list to be sorted from the first element to the last, comparing each pair of elements and switching their positions if necessary. This process is repeated as many times as necessary, until the list is sorted. Since the worst case scenario is that the list is in reverse order, and that the first element in sorted list is the last element in the starting list, the most exchanges that will be necessary is equal to the length of the list.

Programming Activity 35

In this activity we create a simple bubble sort using some of the code from the previous search activities. Create a new Windows Forms Application and modify the properties as shown. There are additional components added to the form for demonstration purposes and a pause has been used to slow down the process.



Form Text : Bubble Sort
ListBox (Name) : listResults
TextBox (Name) : textOuterFOR
TextBox ReadOnly : True
TextBox (Name) :textInnerFOR
TextBox ReadOnly : True
Button Text : Sort Button (Name) : btnSort

In this example the two TextBoxes have been made read only, this setting is optional. Add the following code to the application in the Code View,

```
public Form1()
{
    InitializeComponent();
    // Call method to fill Array at start up
    FillArray();
}
```

```
// Array of random integers
static int max = 20;
int[] myArray = new int[max];

private void btnSort_Click(object sender, EventArgs e)
{
    int temp = 0;
    for (int outer = 0; outer < max; outer++)
    {
        for (int inner = 0; inner < max - 1; inner++)
        {
            if (myArray[inner] > myArray[inner + 1])
            {
                // Swap routine
                temp = myArray[inner + 1];
                myArray[inner + 1] = myArray[inner];
                myArray[inner] = temp;
            }
            // Code to demonstrate the bubble sort
            ShowArray();
            Application.DoEvents();
            Thread.Sleep(100);
            textInnerFOR.Text = inner.ToString();
            textOuterFOR.Text = outer.ToString();
        }// Inner FOR Loop
    }// Outer FOR Loop
}// end of btnSort_Click

// Method to display array
private void ShowArray()
{
    listResults.Items.Clear();
    for (int i = 0; i < max; i++)
    {
        listResults.Items.Add(myArray[i]);
    }
}// end of method ShowArray

// Method to fill Array with random numbers
private void FillArray()
{
    // Create a random number
    Random rand = new Random();
    for (int i = 0; i < max; i++)
    {
        // Random number 0..100
        myArray[i] = rand.Next(100);
    }
}// end of method FillArray
```

Add using
{ } System.Threading

The key to this sorting algorithm is the swap routine (refer Figure 19) which puts the higher value element into a temporary variable space (1) whilst the lower value element is moved (2). Finally the higher value element is placed back into the array (3).

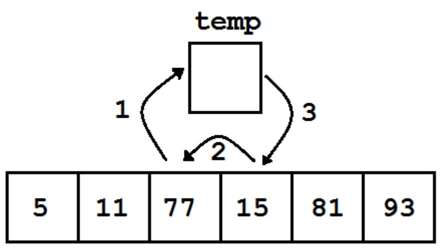


Figure 19

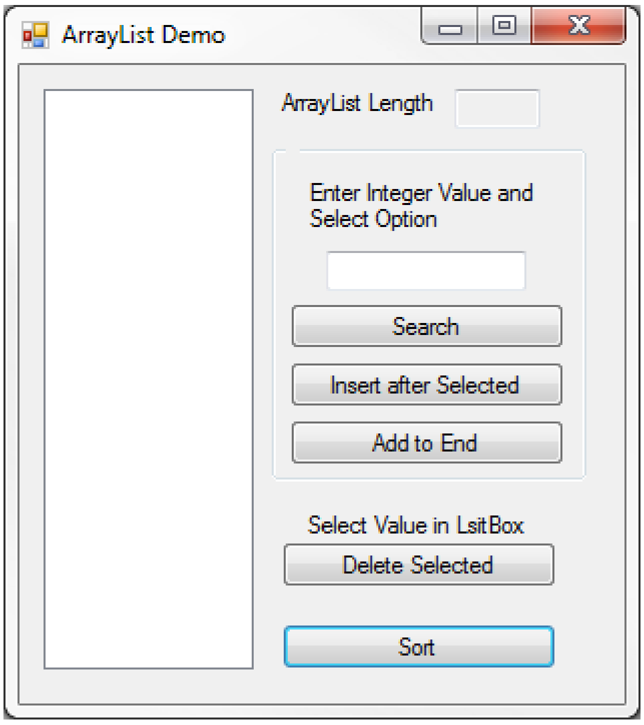
Insertion and Deletion

The nature of Arrays is that their length is immutable, that is fixed at the time of execution. You can't add or delete any of the Array elements because the maximum size of the Array is declared before the Array is created. This fixed Array construct is a serious problem for real world scenarios. The ArrayList is one of the most flexible data structures from C# Collections. The ArrayList contains a simple list of values and implements the IList interface so we can easily add, insert, delete, view, search, and sort. It is very flexible because it will grow and shrink dynamically as elements are modified in the ArrayList.

In the next activity we will create a simple ArrayList and add sufficient buttons to demonstrate the versatility of this construct.

Programming Activity 36

In this activity create a new Windows Forms Application and add the following components. Change the properties of these components as shown in the table below.



Form Text : ArrayList Demo
ListBox (Name) : listResults
TextBox (Name) : textArrayListLength
TextBox ReadOnly : True
TextBox (Name) : textInput
Button Text : Binary Search Button (Name) : btnSearch
Button Text : Insert after Selection Button (Name) : btnInsert
Button Text : Add to End Button (Name) : btnAdd
Button Text : Delete Selected Button (Name) : btnDelete
Button Text : Sort Button (Name) : btnSort

Add the following code and then run.

Add using
{ } System.Collections

```
// Create an ArrayList
ArrayList myList = new ArrayList();

// Button Method to insert new item
private void btnInsert_Click(object sender, EventArgs e)
{
    if (!(String.IsNullOrEmpty(textInput.Text)))
    {
        if (!(listResults.SelectedIndex == -1))
        {
            string curIndex = listResults.SelectedItem.ToString();
            int indx = listResults.FindString(curIndex);
            myList.Insert(indx + 1,textInput.Text);
            ShowArrayList();
        }
        else
        {
            MessageBox.Show("Select an insertion point in the ListBox");
        }// if/else
    }// IF
}// end of btnInsert_Click

// Button Method to search ArrayList
private void btnSearch_Click(object sender, EventArgs e)
{
    myList.Sort();
    int indx = -1;
    if (!(String.IsNullOrEmpty(textInput.Text)))
    {
        indx = myList.BinarySearch(textInput.Text);
        if (indx > -1)
        {
            MessageBox.Show("Found at index " + indx);
        }
        else
        {
            MessageBox.Show("Search did not find " + textInput.Text);
        }
    }
}// end of btnSearch_Click

// Button Method to delete selected item from ArrayList
private void btnDelete_Click(object sender, EventArgs e)
{
    if (!(listResults.SelectedIndex == -1))
    {
        string curIndex = listResults.SelectedItem.ToString();
        int indx = listResults.FindString(curIndex);
        myList.Remove(curIndex);
        ShowArrayList();
    }
    else
    {
        MessageBox.Show("Select an item for deletion from the ListBox");
    }// else
}// end of btnDelete_Click

// Button Method to add item to end of ArrayList
private void btnAdd_Click(object sender, EventArgs e)
{
    myList.Add(textInput.Text);
    textInput.Clear();
    ShowArrayList();
}// end of btnAdd_Click

// Button Method to sort ArrayList
private void btnSort_Click(object sender, EventArgs e)
{
    myList.Sort();
    ShowArrayList();
}// end of btnSort_Click

// Method to display ArrayList
private void ShowArrayList()
{
    listResults.Items.Clear();
    for (int i = 0; i < myList.Count; i++)
    {
        listResults.Items.Add(myList[i]);
    }
    textArrayListLength.Text = myList.Count.ToString();
}// endof ShowArrayList
```

Enter several integer/text items and then use the buttons to manipulate the data. This activity demonstrates the basics of inserting and deleting items from an ArrayList. You will notice that the ArrayList does not have a data type and will accept all alpha numeric data.

Sequential and Random Access

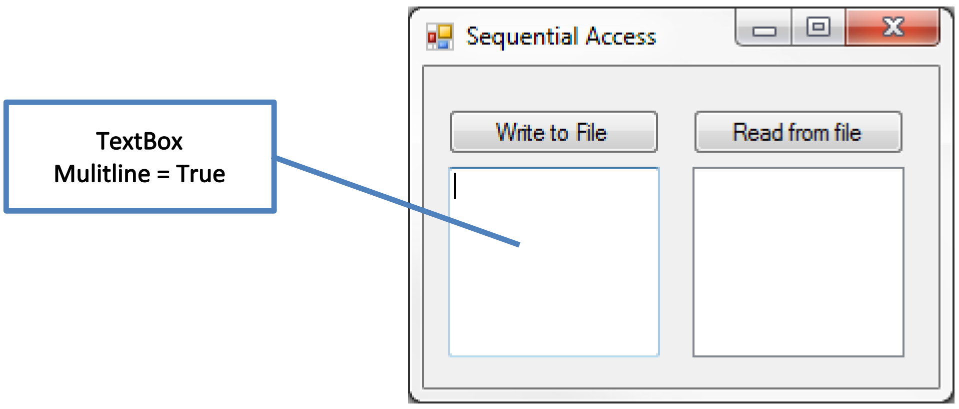
In programming, sequential access means that a group of elements (such as data in an Array) is accessed in an ordered sequence. Sequential access is sometimes the only way of accessing the data, for example if it is on a tape or a file. To read or write data in sequential order, means reading one record after the other. For example to read record #10, you would first need to read records 1 through 9. This differs from random access, in which you can read and write records in any order.

With random access (more precisely and more generally called direct access) there is the ability to access an item of data at any given point in the Array of elements. As a rule the assumption is that each element can be accessed roughly as easily and efficiently as any other, no matter how many elements may be in the Array. Data might be stored notionally in a single sequence like a row, in two dimensions like rows and columns (table), or in multiple dimensions.

There is no difference in the files, only in the way they are accessed. In the next activity we will read and write to a simple text file; the data will be stored on the local hard disk which can be read by notepad or other text editor.

Programming Activity 37

Create a new Windows Forms Application and modify the properties of the components as shown below.



Form Text : Sequential Access
ListBox (Name) : listOutput
TextBox (Name) : textInput
Button Text : Write to File Button (Name) : btnWrite
Button Text : Read from File Button (Name) : btnRead

Now add the following code to the Button methods; ensure the directory c:\temp is created before you run this program.

```
private void btnWrite_Click(object sender, EventArgs e)
{
    // Create a writer and open the file
    TextWriter tw = new StreamWriter("c:\\temp\\data.txt");

    // Write each line of text into the file
    string line = textInput.Text;
    tw.WriteLine(line);

    // Close the stream
    tw.Close();
    textInput.Clear();
}// end of btnWrite_Click

private void btnRead_Click(object sender, EventArgs e)
{
    listOutput.Items.Clear();
    // Create reader & open file
    TextReader tr = new StreamReader("c:\\temp\\data.txt");

    //Read each line of text
    string line;
    while ((line = tr.ReadLine()) != null)
        listOutput.Items.Add(line);

    // Close the stream
    tr.Close();
}// end of btnRead_Click
```

Add using
{ } System.IO

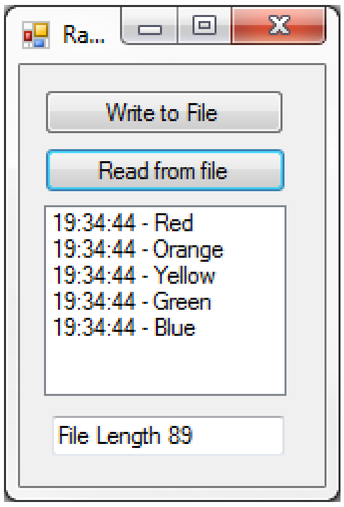
The location of the data file can be changed but it is important that the Directory structure exists before you run this program. To test this program, enter a series of names (press Enter to put each name on a new Line). Once you click the Write Button this text will be written sequentially to a text file. Now click the Read Button to read the text from the file into the ListBox. If you locate the data.txt file in the C:\TEMP directory you can open this with Notepad to examine the contents, it will be a list of the names you entered from the application. Reading and writing sequential data is very simple and usually done in tape backup.

Binary streams, as their name suggests, work with binary (raw) data. This makes them universal so they can be used to read information from all sorts of files (music and multimedia files, text files etc.). Binary files are useful for saving application configurations for example in a settings file. They can also save and load data entered by the user during runtime.

In the next activity we create a set of simple data and write these values to a binary file. Then we will read these values into a ListBox.

Programming Activity 38

Create a new Windows Forms Application and following components. Change the properties of these components as shown in the table below.



```
19:34:44 - Red
19:34:44 - Orange
19:34:44 - Yellow
19:34:44 - Green
19:34:44 - Blue
```

File Length 89

Form Text : Random Access
ListBox (Name) : listOutput
TextBox (Name) : textMessage
Button Text : Write to File Button (Name) : btnWrite
Button Text : Read from File Button (Name) : btnRead

Now add the following code to the Button methods; ensure the directory c:\temp is created before you run this program.

```
static string fileName = "C:\\temp\\data.bin";

private void btnWrite_Click(object sender, EventArgs e)
{
    string[] colors = { "Red", "Orange", "Yellow", "Green", "Blue" };
```

```
     if (!(File.Exists(fileName)))
     {
         using (BinaryWriter bw = new BinaryWriter(File.Open(fileName,
                                                           FileMode.Create)))
         {
             for (int x = 0; x < 5; x++)
             {
                 bw.Write(DateTime.Now.Hour);
                 bw.Write(DateTime.Now.Minute);
                 bw.Write(DateTime.Now.Second);
                 bw.Write(colors[x]);
             }
             textMessage.Text = "Binary File created";
         }
     }
     else
     {
         textMessage.Text = "File not created";
     }

 }// end of btnWrite_Click

private void btnRead_Click(object sender, EventArgs e)
{
    string data = null;
    listOutput.Items.Clear();
    if (File.Exists(fileName))
    {
        using (BinaryReader br = new BinaryReader(File.Open(fileName,
                                                           FileMode.Open)))
        {
            for (int x = 0; x < 5; x++)
            {
                data = br.ReadInt32() + ":" + br.ReadInt32() + ":"
                    + br.ReadInt32() + " - " + br.ReadString();
                listOutput.Items.Add(data);
            }
             int fileLength = (int)br.BaseStream.Length;
             textMessage.Text = "File Length " + fileLength;
        }
    }
    else
    {
        textMessage.Text = "File not created";
    }
}// end of btnRead_Click
```

Add using
{ } System.IO

Click the “*Write to File”* button to start the program. This example is very simplistic and lacks the functionality of a full application; however, the basic concepts are demonstrated. Each data type must be written and read in the correct order. If you open the Binary file with Notepad you will see the difference between the Text file and the Binary file. This construct supports a range of methods which allow the file reading/writing position to be moved within the file. These concepts are outside the scope of this text.

Optional Activities

For each activity write the appropriate console code. Possible solutions are located in Appendix E.

Act-27: Simple array search using FOR and IF statements

Act-28: Sort a table of values using a 2D array

Assessment Activity AT1.3

You should now attempt Assessment Portfolio AT1.3 which consists of a sorting programming question using Windows Forms Application. Ensure you add comments at the top of all your code and for each method block. The assessment can be downloaded from e-campus.

CHAPTER SIX

In this chapter you will use the Visual Studio integrated development environment to debug a series of algorithms. The learning outcomes are to code and execute a simple program using the debug tools to trace through the code whilst examining the variables. At the end of the chapter you will have:

- Used the debugging tools provided by the Visual Studio integrated development environment (IDE) to debug code.
- Used a debugger to trace code execution and examine variable contents to detect and correct errors.

Debugging

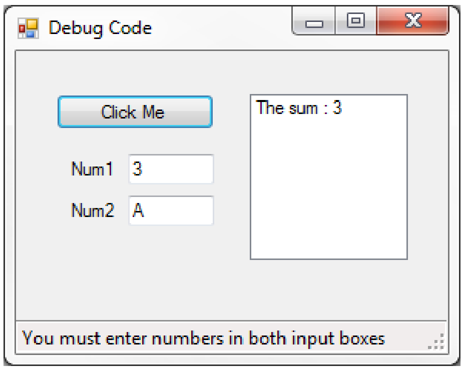
The errors your program will encounter can be classified in three categories: runtime, syntax, and logic errors. Once you have created your application and resolved the syntax errors you must now correct those logic errors that keep your application running correctly. You can do this with the development environment's integrated debugging functions. These allow you to stop at procedure locations, inspect memory and register values, change variables, observe message traffic, and get a close look at what your code does.

In Visual Studio the debug tools can let you; Start, Break, Step, Run through Code, and Stop Debugging. Generally bugs occur when the logic of the algorithm has not been tested and unforeseen problems occur. For example when you iterate though an Array of data using a loop, the code must not overrun the end of the Array. When reading data from the keyboard you must ensure input is suitable for the variable data type (not entering a character into an integer variable). There are numerous logical errors that occur in real world programming applications that are overlooked by programmer and software developers (ref: the division error in the movie Office Space 1999).

In the next activity you will use the debug tools to examine a simple program and view the change in the values of the variables as their scope changes. The scope of a variable is the area of code in which the variable is visible.

Programming Activity 39

Create a new Windows Forms Application and add the following components,



The sum : 3

You must enter numbers in both input boxes

Form Text : Debug Code
Button Text : Click Me Button (Name) : btnStart
TextBox (Name) : textNum1
TextBox (Name) : textNum2
ListBox (Name) : listResults
StarusStrip (Name) : statusMsg

In this code we segment the code and use several methods to perform difference activities. The Comments have been removed so you will have an opportunity to discover how this program works. Once the code is error free, use the data in the test table to test the program and record your results.

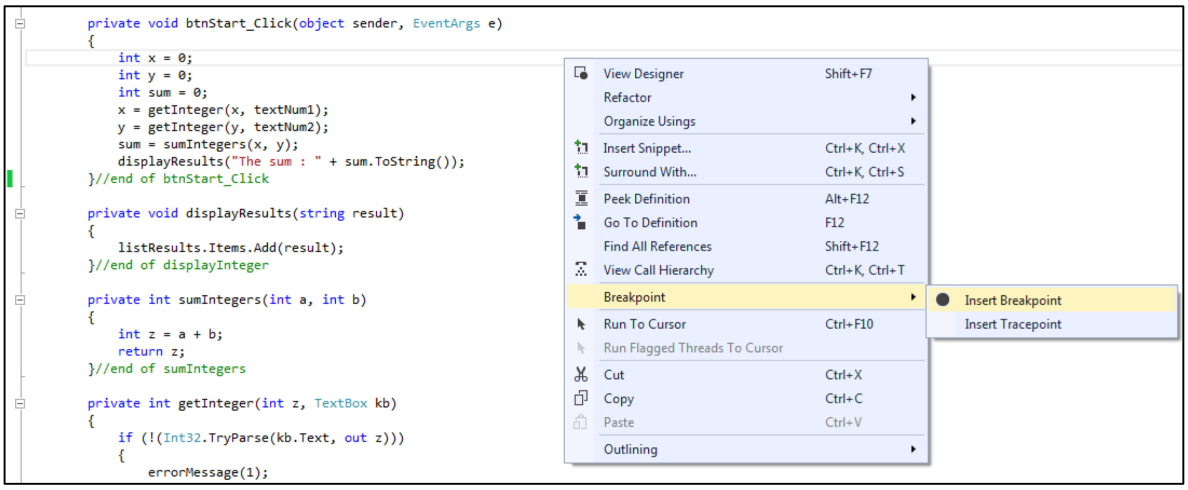
```
private void btnStart_Click(object sender, EventArgs e)
{
    int x = 0;
    int y = 0;
    int sum = 0;
    x = GetInteger(x, textNum1);
    y = GetInteger(y, textNum2);
    sum = SumIntegers(x, y);
    DisplayResults("The sum : " + sum.ToString());
}// end of btnStart_Click

private void DisplayResults(string result)
{
    listResults.Items.Add(result);
}// end of DisplayInteger

private int SumIntegers(int a, int b)
{
    int z = a + b;
    return z;
}// end of SumIntegers

private int GetInteger(int z, TextBox kb)
{
    if (!(Int32.TryParse(kb.Text, out z)))
    {
        ErrorMessage(1);
        return 0;
    }
    else
    {
        ErrorMessage(2);
        return z;
    }
}// end of method GetInteger

private void ErrorMessage(int error)
{
    statusMsg.Items.Clear();
    switch(error)
    {
        case 1 : statusMsg.Items.Add("Enter Integers into the input boxes");
            break;
        case 2 : statusMsg.Items.Add("Both numbers are OK");
            break;
    }
}// end of ErrorMessage
```

The code from this example does not flag any errors but once you begin to test the program you will find some unusual results. These are logical faults that don’t always stop the program but return “rubbish” results.

Input		Expected Output	Actual Results
Num1 :	Num2 :		
4	5	Sum : 9	Correct
6	A	No Result, cannot add integer to alphabet	Sum : 6
B	Z	No Result, cannot add two alphabet characters	Sum : 0
2.3	8	Sum : 10.3	Sum : 8
5.5	4.5	Sum : 10	Sum : 0
Null	Null	No Result	Sum : 0

The obvious errors can be fixed by changing the data type to a Double or Float. But now add a break point to the code so you can step through the logic and see how each method is called and executed.

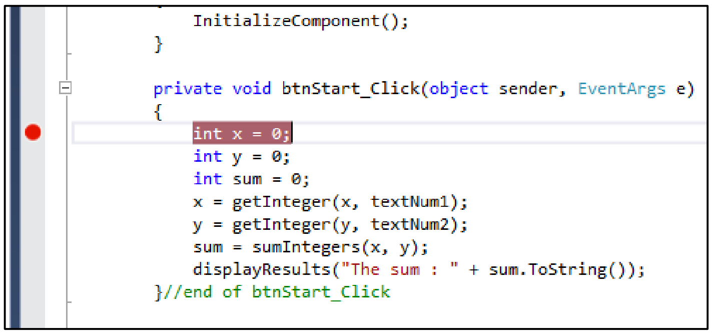
On the line of code “`int x = 0;`” right click to bring up the context menu, then select Breakpoint > Insert Breakpoint (ref Figure 20).



```
        private void btnStart_Click(object sender, EventArgs e)
        {
            int x = 0;
            int y = 0;
            int sum = 0;
            x = getInteger(x, textNum1);
            y = getInteger(y, textNum2);
            sum = sumIntegers(x, y);
            displayResults("The sum : " + sum.ToString());
        }//end of btnStart_Click

        private void displayResults(string result)
        {
            listResults.Items.Add(result);
        }//end of displayInteger

        private int sumIntegers(int a, int b)
        {
            int z = a + b;
            return z;
        }//end of sumIntegers

        private int getInteger(int z, TextBox kb)
        {
            if (!(Int32.TryParse(kb.Text, out z)))
            {
                errorMessage(1);
```

Figure 20

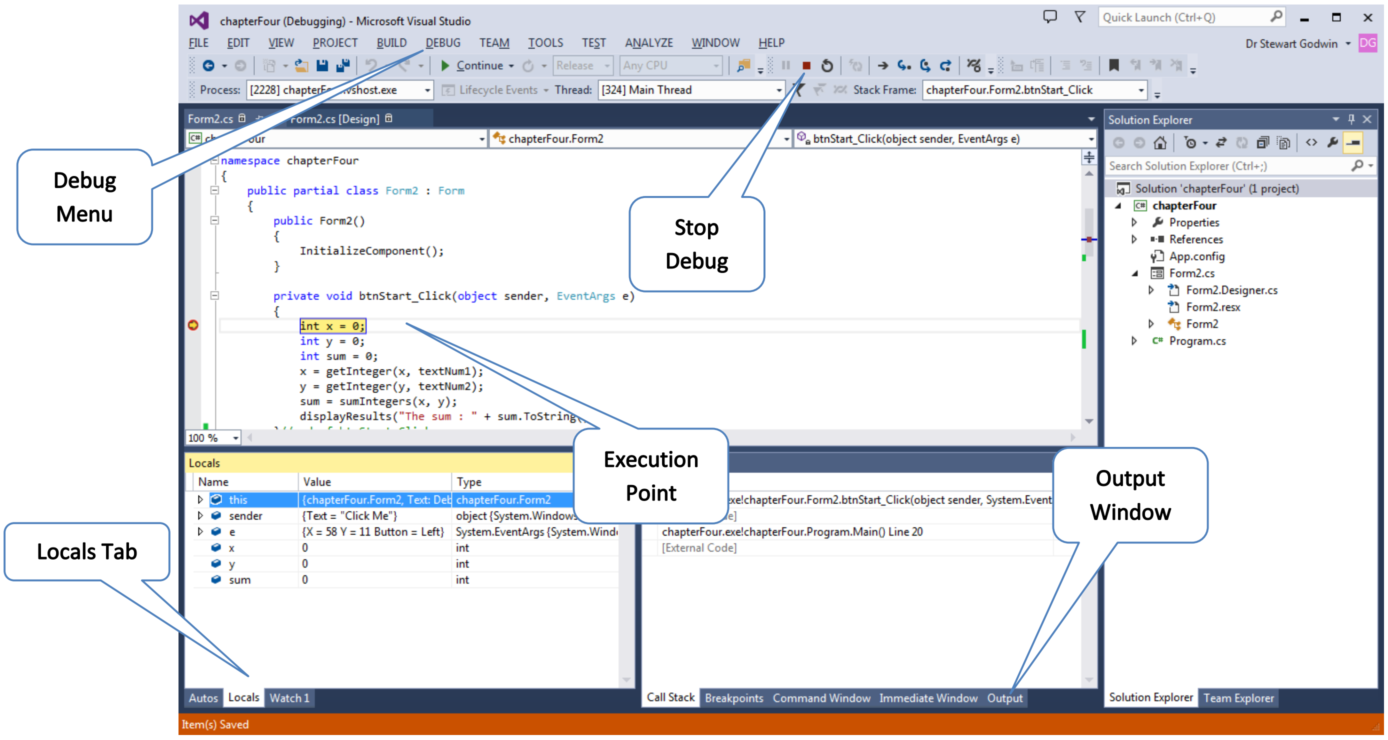
You will now have a line of code that is highlighted as shown in Figure 21.



```
            InitializeComponent();
        }

        private void btnStart_Click(object sender, EventArgs e)
        {
            int x = 0;
            int y = 0;
            int sum = 0;
            x = getInteger(x, textNum1);
            y = getInteger(y, textNum2);
            sum = sumIntegers(x, y);
            displayResults("The sum : " + sum.ToString());
        }//end of btnStart_Click
```

Figure 21

Run the program by pressing the F5 key and enter two integers into the TextBoxes, then click the Button. You will notice the program stops at the first line where the break point was inserted.


Debug Menu
Stop Debug
Execution Point
Output Window
Locals Tab
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP
Continue
Autos Locals Watch 1
Call Stack Breakpoints Command Window Immediate Window Output
Solution Explorer Team Explorer

```
namespace chapterFour
{
    public partial class Form2 : Form
    {
        public Form2()
        {
            InitializeComponent();
        }

        private void btnStart_Click(object sender, EventArgs e)
        {
            int x = 0;
            int y = 0;
            int sum = 0;
            x = getInteger(x, textNum1);
            y = getInteger(y, textNum2);
            sum = sumIntegers(x, y);
            displayResults("The sum : " + sum.ToString(
```

Locals

Name	Value	Type
this	{chapterFour.Form2, Text: Deb	chapterFour.Form2
sender	{Text = "Click Me"}	object {System.Windows
e	{X = 58 Y = 11 Button = Left}	System.EventArgs {System.Wind
x	0	int
y	0	int
sum	0	int

```
xe!chapterFour.Form2.btnStart_Click(object sender, System.Event
e]
chapterFour.exe!chapterFour.Program.Main() Line 20
[External Code]
```

Figure 22

You are in the debug mode, so you will be able to move through the code one line at a time (press F11). In the lower windows select the Locals tab to display the variable in your code (ref Figure 22). Each time you press the F11 key you will advance the program by one line of code. This will demonstrate how the program works and how each method in called. Once you enter the getInteger method the local variable x and y will disappear from the Locals tabs because they are out of scope. You will notice the variable z has come into scope. In this code the variable names x, y and z do not provide must help, unlike the variable sum which is self-explanatory. Without the comments it is not obvious what each method does, also using suitably descriptive variable names would clarify what is happening.

Once the program if finished, you can right click the break point line of code and delete this break point using the context menu. All the debug actions can be performed using the Debug Menu.

You should now change all the integer variables into Double data types and run the test data again. The final logical error is in the getInteger method, this will require re-coding so that the sum method will not execute if an alphabet character is entered into the TextBoxes.

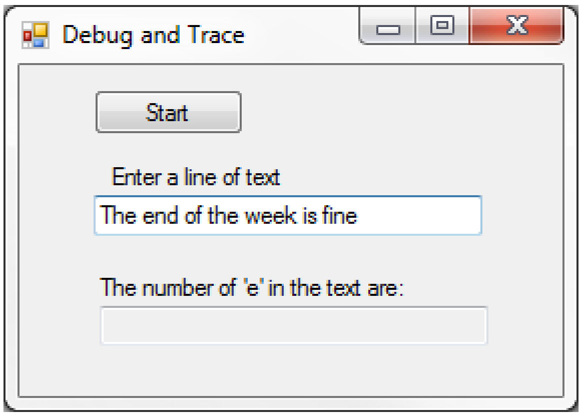
Error Tracing

A logic error occurs when the program (the code) is written without syntax errors but the result it produces is not reliable. With logic errors, the Code Editor does not see anything wrong in the code and therefore cannot point out a problem. Using a trace can help identify certain errors like “Out By One Error” which is usually associated with reading and writing data into arrays and collections.

Debugging allows us to observe and correct programming errors. Tracing is a form of Debugging that allows us to keep track of the health and sanitary conditions of our applications when they are in production.

Programming Activity 40

In this activity you will create a simple program to count the number of ‘e’ in a sentence. Create a new Windows Forms Application and add the following components, the code will use a new method called Substring. The Substring method grabs characters from the text. The first 1 in the brackets means start at letter 1 in the text; the second 1 grab 1 character.


Debug and Trace
Start
Enter a line of text
The end of the week is fine
The number of 'e' in the text are:

Form Text : Debug and Trace
Button Text : Start Button (Name) : btnStart
TextBox (Name) : textInput
TextBox (Name) : textOutput

Add the following code and run the program. Use the debug break point to view the variables.

```
private void btnStart_Click(object sender, EventArgs e)
{
    int letterCount = 0;
    string inputString = textInput.Text;
    string oneLetter;
    for (int i = 0; i < inputString.Length; i++)
    {
        oneLetter = inputString.Substring(1,1);
        if (oneLetter == "e")
            letterCount++;
    }// FOR loop
    textOutput.Text = letterCount.ToString();
}// end of btnStart_Click
```

The first 1 in the brackets means start at letter 1 in the text; the second 1 grab 1 character

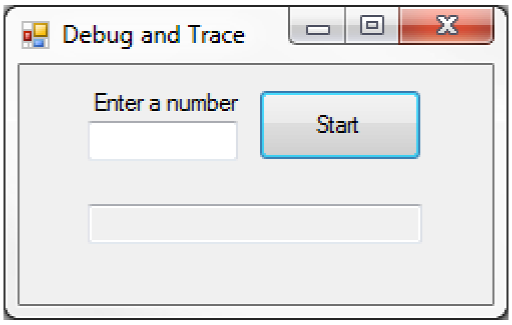
The problem is obvious when you discover the variable oneLetter is not moving. To fix this change the code Substring(1, 1) to Substring(i, 1).

The C# Framework contains a mechanism called a Trace Listener. This Listener is an object which directs the trace output to an appropriate target, such as an error log for example. In our next activity we will use the default target which is the Output Window in Visual Studio (ref Figure 22).

You must exercise care when placing your trace statements for use during run time. You must consider what tracing information is likely to be needed in a deployed application, so that all likely tracing scenarios are adequately covered.

Programming Activity 41

In this activity we create a simple program that will test if a number is a Prime, that is does not have any factors (1, 2, 3, 5, 7, 11 …). In this program the Button will be activated by pressing the Enter key and the input cursor will stay focused on the TextBox.



The code has been fully commented so each section of code can be understood without further explanation. There is full error trapping which will prevent invalid input. Finally the Tracing code has been added so that when the program is run you will see the log file entries in the Output Window.

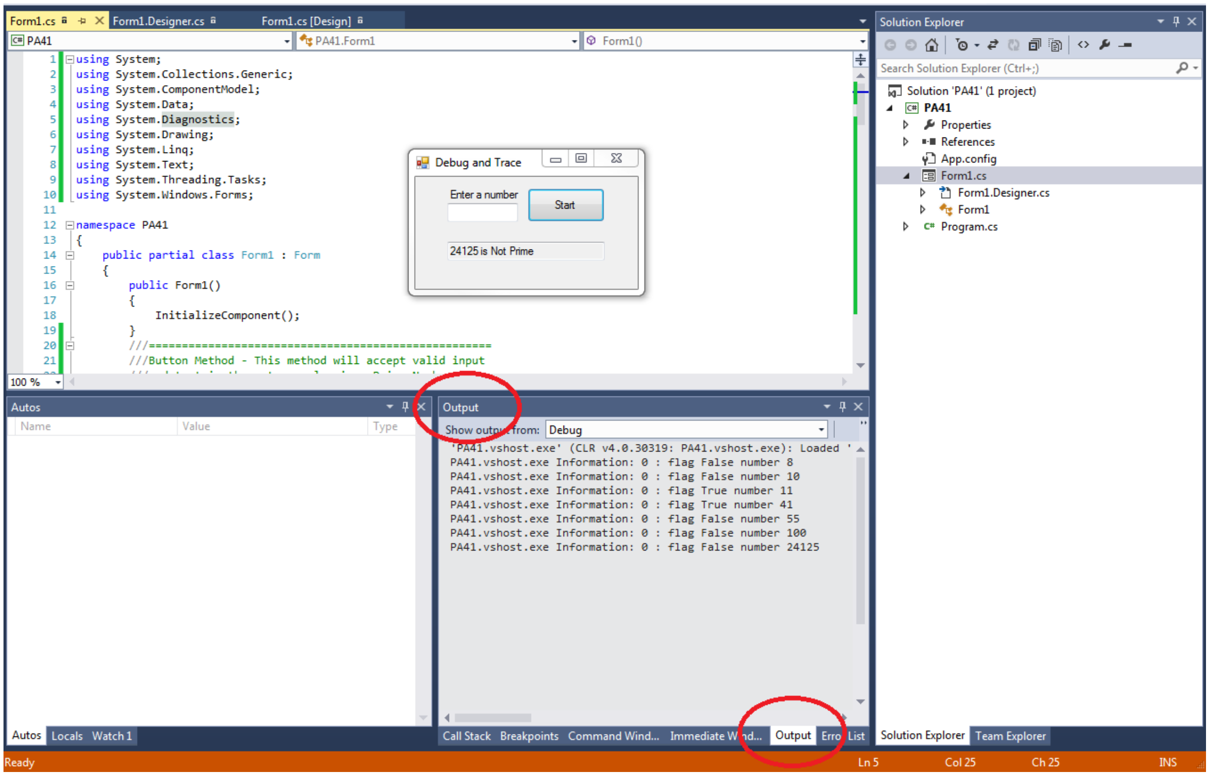
```
// Button Method - This method will accept valid input
// and test is the entry value is a Prime Number
private void btnStart_Click(object sender, EventArgs e)
{
    // Clear the output box from previous values
    textOutput.Clear();
    // Declare and initialise input variable
    int myInt = 0;
    // Set a boolean flag for Prime
    bool isPrime = true;
    // Test if input value is valid
    try
    {
        // Parse the TextBox for an Integer
        myInt = int.Parse(textInput.Text);
        // Call method to clear and rest TextBox
        ClearReset();
        // Filter negative input values
        if (myInt < 0)
        {
            // Show error message
            textOutput.Text = "Enter Number > 1";
            // Exit the method
```

```
                return;
            }// IF
        }
        catch (FormatException)
        {
            // Show error message
            textOutput.Text = "Well that didnt work!";
            // Call method to clear and rest TextBox
            ClearReset();
            // Exit the method
            return;
        }// end of TRY/CATCH

        // Divide the input to find factors
        // Use modulus to determine if number is Prime
        for (int i = 2; i <= myInt - 1; i++)
        {
            // Test if number is Prime
            if ((myInt % i) == 0)
            {
                // Set flag when number has a factor
                isPrime = false;
            }//IF
        }// FOR
        // Call method to display result, send the
        // input number and isPrime Flag as parameters
        ShowResult(myInt, isPrime);
    }// end of btnStart_Click

    // Method ShowResult - This method take two arguments
    // and display the input value and if the number is a Prime Number
    private void ShowResult(int x, bool flag)
    {
        // If number is prime display value and text message
        if (flag)
        {
            textOutput.Text = x + " is Prime";
            // Output Trace information
            Trace.TraceInformation("flag " + flag + " number " + x);
        }
        // Else number is not a Prime
        else
        {
            textOutput.Text = x + " is Not Prime";
            // Output Trace information
            Trace.TraceInformation("flag " + flag + " number " + x);
        }
        // Clear the TextBox for the next entry
        textInput.Clear();
        // Put the cursor into the TextBox
        textInput.Focus();
    }// end of ShowResults

    // Method ClearReset - Method to clear the input
    // TextBox and put curse into TextBox
    private void ClearReset()
    {
        // Clear the TextBox for the next entry
        textInput.Clear();
        // Put the cursor into the TextBox
        textInput.Focus();
    }// end of ClearReset
```

Add using
{ } System.Diagnostics

```
}// end of Class Debug_Trace
```

Run the program and view the Output Window in Visual Studio, you will notice each time you click the Start Button new entries will appear. This information could be sent to a log file on a hard disk. The end user is unaware of the logging activity until the program crashes. To demonstrate this point, enter a very large number that is outside the maximum value for an integer (greater than 2147483647).



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

namespace PA41
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        ///===================================================
        ///Button Method - This method will accept valid input
```

Enter a number

24125 is Not Prime

Show output from: Debug

```
'PA41.vshost.exe' (CLR v4.0.30319: PA41.vshost.exe): Loaded '
PA41.vshost.exe Information: 0 : flag False number 8
PA41.vshost.exe Information: 0 : flag False number 10
PA41.vshost.exe Information: 0 : flag True number 11
PA41.vshost.exe Information: 0 : flag True number 41
PA41.vshost.exe Information: 0 : flag False number 55
PA41.vshost.exe Information: 0 : flag False number 100
PA41.vshost.exe Information: 0 : flag False number 24125
```

Optional Activities

For each question research and write the appropriate definition or explanation.

a. Define Debug,

b. Define Trace,

c. Explain the difference between debug and trace,

d. Finally provide an example where each could be used.

Provide a suitable definition for;

e. Runtime,

f. Syntax,

g. Logic errors.

CHAPTER SEVEN

In this chapter you will perform documentation activities by adding comments to programming code using the basic C# language syntax. Then you will generate the XML file associated with the code and view the output. At the end of the chapter you will have:

- Follow organisational guidelines for developing maintainable code and adhere to the provided coding standard when documenting activities.
- Apply internal documentation suitable for use by peers to all code created and use documentation tools available in the target language when documenting activities.

Coding Standards

Coding standards are important so programmers and software developers can read and understand the code during the development stages and later when the application is in production. Most applications will require maintenance and updates, which occur on a regular basis, these changes are implemented based on the original code. We use code conventions as a set of guidelines which relate to each language and recommend the style, practices and methods for each aspect of how this language is written.

These conventions usually cover indentation, comments, declarations, statements, naming conventions and best practices. Throughout this text we have been using the conventions suggested by the Microsoft Developers Network. However, as a lecturer in computer programming I have altered these basic conventions to help students identify various variables and components. In a real world programming environment a programmer will be required to adopt the organisation conventions of their employer.

The following conventions are presented here as a guide and reflect the suggestions from Microsoft.

General Naming Conventions

The following table describes the capitalization and naming rules for several basic types of identifiers. A full list can be obtained from the relevant website. You should always use meaningful names for all functions, methods, variables, constructs and types. Avoid shortening or contracting names as this can lead to ambiguity when the code is examined by other programmers. As a generally rule, write your code so a layperson might understand.

Link: [All-In-One Code Framework]()

Capitalization Naming Rules for Identifiers

Identifier	Casing	Naming Structure	Example

Class, Structure	PascalCasing	Noun	`public class ComplexNumber {...}`
			`public struct ComplextStruct {...}`

Namespace	PascalCasing	Noun	`namespace Microsoft.Sample.Windows7`
☒ **Do not** use the same name for a namespace and a type in that namespace.			

Method	PascalCasing	Verb or Verb phrase	`public void Print() {...}`
			`public void ProcessItem() {...}`

Public Property	PascalCasing	Noun or Adjective	`public string CustomerName`
			`public ItemCollection Items`
			`public bool CanRead`

☑ **Do** name collection proprieties with a plural phrase describing the items in the collection, as opposed to a singular phrase followed by “List” or “Collection”. ☑ **Do** name Boolean proprieties with an affirmative phrase (CanSeek instead of CantSeek). Optionally, you can also prefix Boolean properties with “Is,” “Can,” or “Has” but only where it adds value.			

Non-public Field	camelCasing or _camelCasing	Noun or Adjective.	`private string name;`
			`private string _name;`

☑ **Do** be consistent in a code sample when you use the '_' prefix.			

Constant	PascalCasing or camelCasing	Noun	`public const string MessageText = "A";`
			`private const string messageText = "B";`
			`public const double PI = 3.14159...;`

PascalCasing for publicly visible; camelCasing for internally visible; All capital only for abbreviation of one or two chars long.			
Parameter, Variable	camelCasing	Noun	`int customerID;`

UI Control Naming Conventions

The other area that requires consideration is the components on the Windows Forms. These require a prefix so they can be differentiated from the C# code in the Code View. The primary purpose is to make the code more readable. The following table has the standard prefixes for the controls used throughout this text,

Control Type	Prefix
Button	btn
CheckBox	chk
ComboBox	cmb
DateTimePicker	dtp
Form	suffix: XXXXForm
GroupBox	grp
Label	lb
ListBox	lst
Menu	mnu
RadioButton	rad
StatusBar	sts
TextBox	tb

The use of prefixes in this text has been modified for educational purposes. However, in future activities you should begin to adopt these conventions as a personal standard.

Documentation

Software documentation exists in two forms, external and internal. External documentation is maintained outside of the source code, such as specifications, help files, and design documents. Internal documentation is composed of comments that developers write within the source code at development time.

We will be dealing with internal documentation in this section, and examine the general rules for comments and how they can be auto generated by the development environment.

The following rules for comments are designed to assist in this process,

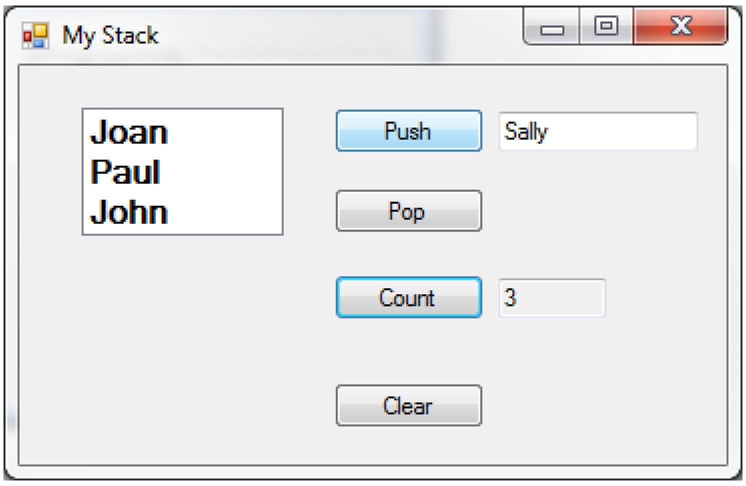
Comment Rules

1. When modifying code, always update the comments.
2. At the beginning of every method, provide a standard, boilerplate comment, indicating the routine's purpose, assumptions, and limitations.
3. Avoid adding comments at the end of a line of code.
4. Avoid surrounding a block comment with a typographical frame.
5. If you need comments to explain a complex section of code, examine the code to determine if you should rewrite it. If at all possible, do not document bad code—rewrite it.
6. Use complete sentences when writing comments. Comments should clarify the code, not add ambiguity.
7. Use comments to explain the intent of the code. They should not serve as inline translations of the code.
8. Comment anything that is not readily obvious in the code.
9. Use comments on code that consists of loops and logic branches. These are key areas that will assist the reader when reading source code.

In the next activity we will create a simple program and add suitable comments so the XML documentation can be produced. When creating XML documentation the comments can be categorised by XML elements, we will only use a few of these tags to demonstrate the concept.

Programming Activity 42

Create a new Windows Forms Application and modify the properties of the components as shown below. This program will implement the ADT Stack, which behaves like a vertical column of items (i.e. a stack of books). The explanation of a stack was introduced in Chapter Two.



Form Text : My Stack
Button Text : Push Button (Name) : btnPush
Button Text : Pop Button (Name) : btnPop
Button Text : Count Button (Name) : btnCount
Button Text : Clear Button (Name) : btnClear
TextBox (Name) : tbPush
TextBox (Name) : tbCount TextBox ReadOnly : True
ListBox : lbStack ListBox SelectionMode : None ListBox : 12pt Bold

Add the following code to the Code View and ensure the comments are included.

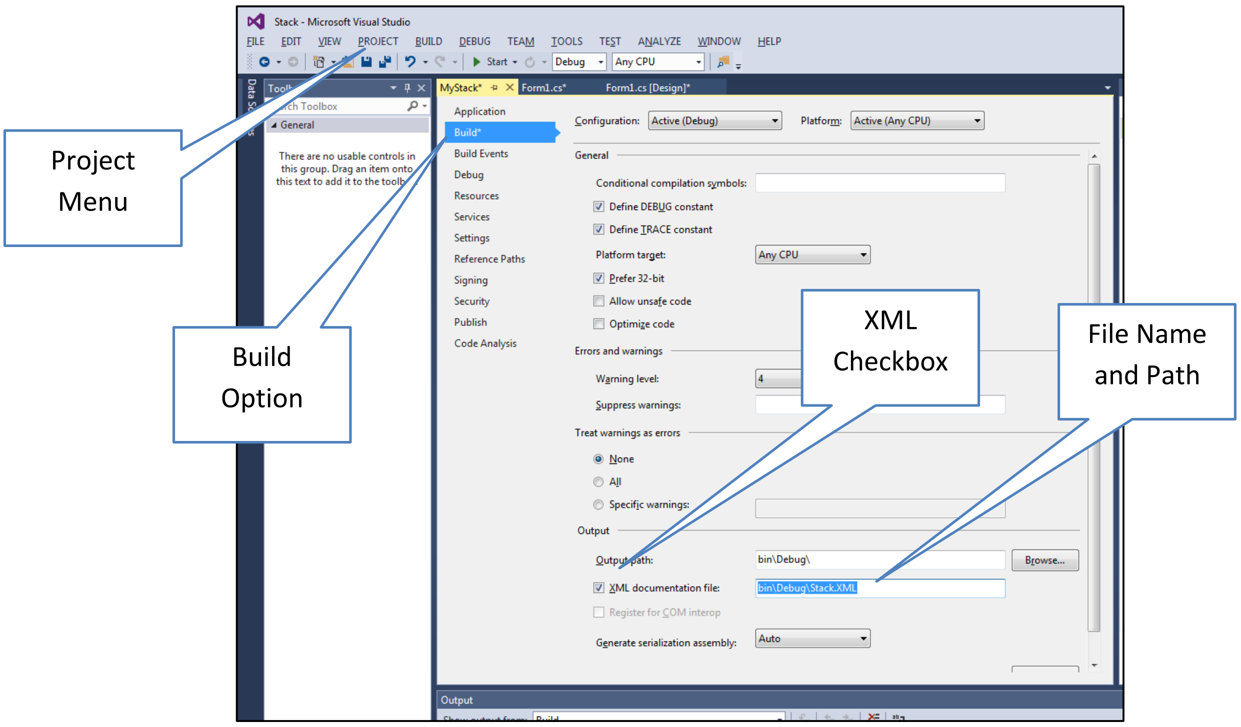
```
/// <summary>
/// This program will implement the ADT Stack.
/// Author: Harry Windsor
/// Date: 23 Jan 2020
/// Version 1.00
/// </summary>
Stack MyStack = new Stack();
```

Add using
{ } System.Collections

```
/// <summary>
/// The Button method will push new data
/// from the TextBox onto the top of the
/// stack. The TextBox will be cleared and
/// the Focus set for the next data item.
/// The ShowStack method will display all
/// the data items in the stack.
/// </summary>
private void btnPush_Click(object sender, EventArgs e)
{
    if (!(String.IsNullOrEmpty(tbPush.Text)))
    {
        MyStack.Push(tbPush.Text);
        tbPush.Clear();
        tbPush.Focus();
        ShowStack();
    }
    else
    {
        MessageBox.Show("nothing to push");
    }
}

/// <summary>
/// The Button method will pop the top data item
/// from the stack. Then call the ShowStack method
/// to display all the data items.
/// </summary>
private void btnPop_Click(object sender, EventArgs e)
{
    try
    {
        MyStack.Pop();
        ShowStack();
    }
    catch
    {
        MessageBox.Show("nothing to pop");
    }

}

/// <summary>
/// The Button method will return the number of
/// data items in the stack.
/// </summary>
private void btnCount_Click(object sender, EventArgs e)
{
    int numStackItems = 0;
    numStackItems = MyStack.Count;
    tbCount.Text = numStackItems.ToString();
}

/// <summary>
/// The Button method will clear the stack
/// of all data items and reset the stack
/// count to 0. Finally call the ShowStack method
/// to display all the data items in the stack.
/// </summary>
private void btnClear_Click(object sender, EventArgs e)
```

```
{
    MyStack.Clear();
    ShowStack();
}

/// <summary>
/// The method will display all the data items in
/// the stack by using the foreach loop. The ListBox
/// will be cleared before the data items are displayed.
/// </summary>
private void ShowStack()
{
    /// <remarks>
    /// The lbHeight variable will be used
    /// to increment the ListBox size as new
    /// data items are added.
    /// </remarks>
    int lbHeight = 20;
    lbStack.Items.Clear();
    foreach (Object obj in MyStack)
    {
        lbHeight = lbHeight + 20;
        lbStack.Height = lbHeight;
        lbStack.Items.Add(obj);
    }
}
```

Once you have an error free program you will need to change the properties of the project in order to generate the XML documentation. From the Menu bar select Project, then MyStack Properties.



Configuration: Active (Debug)　Platform: Active (Any CPU)

General
- Conditional compilation symbols:
- [✓] Define DEBUG constant
- [✓] Define TRACE constant
- Platform target: Any CPU
- [✓] Prefer 32-bit
- [ ] Allow unsafe code
- [ ] Optimize code

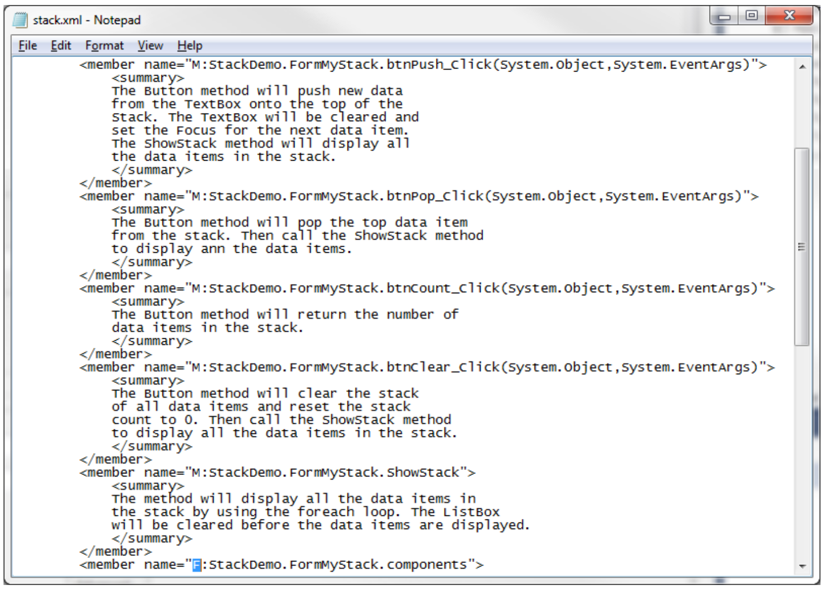
Errors and warnings
- Warning level: 4
- Suppress warnings:

Treat warnings as errors
- [✓ None]
- All
- Specific warnings:

Output
- Output path: bin\Debug\
- [✓] XML documentation file: bin\Debug\Stack.XML
- [ ] Register for COM interop
- Generate serialization assembly: Auto

Figure 23

Once the Properties tab has opened select the Build option and then check the XML Documentation Checkbox. Finally ensure the file name and path are suitable. Once this have been completed you can build the project by using the short cut keys Ctlrl + Shift + B. You will need to locate the XML file from is build location and open it to see the documented comments.



```
        <member name="M:StackDemo.FormMyStack.btnPush_Click(System.Object,System.EventArgs)">
            <summary>
            The Button method will push new data
            from the TextBox onto the top of the
            Stack. The TextBox will be cleared and
            set the Focus for the next data item.
            The ShowStack method will display all
            the data items in the stack.
            </summary>
        </member>
        <member name="M:StackDemo.FormMyStack.btnPop_Click(System.Object,System.EventArgs)">
            <summary>
            The Button method will pop the top data item
            from the stack. Then call the ShowStack method
            to display ann the data items.
            </summary>
        </member>
        <member name="M:StackDemo.FormMyStack.btnCount_Click(System.Object,System.EventArgs)">
            <summary>
            The Button method will return the number of
            data items in the stack.
            </summary>
        </member>
        <member name="M:StackDemo.FormMyStack.btnClear_Click(System.Object,System.EventArgs)">
            <summary>
            The Button method will clear the stack
            of all data items and reset the stack
            count to 0. Then call the ShowStack method
            to display all the data items in the stack.
            </summary>
        </member>
        <member name="M:StackDemo.FormMyStack.ShowStack">
            <summary>
            The method will display all the data items in
            the stack by using the foreach loop. The ListBox
            will be cleared before the data items are displayed.
            </summary>
        </member>
        <member name="F:StackDemo.FormMyStack.components">
```

The source code for any project can be obtained from the Solution Explorer or printed as a PDF file. Generally a wiki or cloud based repository is used to maintain the documentation of the project. This central location enables all members of a development team access to the application resources and when used with proper versioning can assist project management of the project.

Optional Activities

For each question research and write the appropriate definition or explanation.

a. What is a main **static** method?

b. What is a method **modifier**?

c. What is the difference between **class** and **namespace**?

CHAPTER EIGHT

In this chapter you will design and conduct a series of tests on simple programs. The learning outcomes from these activities will be reports that highlight errors and suggest improvements. At the end of this chapter you will have:

- Designed and documented tests,
- Captured and recorded test results.

Testing

Program testing is a very important stage of the software development life cycle. Its purpose is to make sure that all the requirements are strictly followed and covered. This process can be implemented manually, but the preferred method is automated testing. These tests are small programs, which automate the trials as much as possible. However, there are parts of the programming code which are difficult to automate, so a manual testing method is applied.

The testing process is implemented by quality assurance engineers (QAs) who work closely with programmers to find and correct errors (bugs) in the software. Once defects and errors are found the program is sent back to the implementation stage where the code is examined and re-written. In this section we will review several major testing approaches and then conduct a test on a program with several obscure problems.

Black-White Box Testing

The box approach to testing is divided into two areas; White Box and Black Box. White box testing refers to testing of the internal structure of the code; this involves designing test cases for all decision paths and iterative conditions. The tester has full access and knowledge of the programming constructs and can see the code. Black box testing is concerned with the functionality of the program and has no knowledge of the underlying program code. This type of testing is usually associated with end user testing, because the tests are conducted through the User Interface.

Unit Testing

Unit testing means to write a program that tests a certain method or class. A typical unit test executes the method that should be tested, passes sample data via parameters and checks whether the method's result is correct. Using the unit test approach a single method is tested by several unit tests, each implementing a different testing scenario. First, the typical case is checked, and then the border cases are checked.

Test Driven Development

This approach to software development relies on the repetition of very short development cycles, where the programmer writes a test case that defines a problems specification and requirements. Then the programmer writes some code that will pass the test case. Once the code has passed it is included into larger software application project.

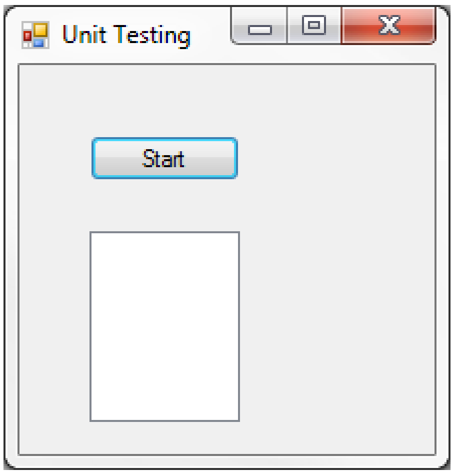
Fault Reporting

A fault is something which the end user sees as a result of an error in your program. If a program fails as part of formal testing, then the process can be repeated and a solution can be implemented (software update). However, if a user reports a fault then the evidence may well be anecdotal. You may not be supplied with a sequence of steps which caused the fault to appear, and you could be told "There's a bug in the print routine". There is a strong argument for ignoring a fault report from a user; however, this approach will not make you popular. Without substantive evidence it may be impossible to track down a bug/fault. A better course of action is to log the events and monitor the program (Tracing Logs).

In the next activity we will examine a snippet of code and design suitable test cases that will determine the quality of the programming logic.

Programming Activity 43

Create a new Windows Forms Application and add the following components. We will be testing this program with data from a test table that ensures all possible alternatives have been considered.



Form Text : Unit Testing
Button Text : Start Button (Name) : btnStart
ListBox (Name) : lbOutput

In the following code we are testing the Sum method, so we will pass an Array of integers to the method and return the sum of all the elements. Once you have error free code you can begin the test by adding the test code in Button click method.

```
static int Sum(int[] numbers)
{
    int sum = numbers[0];
    for (int i = 1; i < numbers.Length; i++)
    {
        sum += numbers[i];
    }
    return sum;
}

private void btnStart_Click(object sender, EventArgs e)
{
    int[] myArray = {1};
    lbOutput.Items.Add(Sum(myArray));
}
```

The default case is where the Array has one element with an integer value equal to 1. A test table can have many columns, which depend on the complexity of the test; in this activity we will use a table of four columns. Modify the integer Array for each test case and run the program, then observe the results.

```
int[] myArray = {1}; // change this assignment for each test case.
```

You should get the same results as the completed table below (Figure 24).

Input Int[ ] myArray =	Expected Output	Actual Results	Comment
{1};	1	1	OK
{1,2,3};	6	6	OK
{0};	0	0	OK
{ }; //Empty	0	Exception	IndexOutOfRangeException
{-1, -2};	-3	-3	OK
{2000000000, 2000000000};	4000000000	-294967296	Overflow (number too big)
null;	0	Exception	NullReferenceException

Figure 24

We now have some test data to use when fixing our bugs, the first error is Index Out of Range. This is associated with reading an Array beyond it limits. This can be fixed by changing the start value of the FOR loop from 1 to 0 and modify the initialisation of the sum variable.

```
//Original Code
int sum = numbers[0];
for (int i = 1; i < numbers.Length; i++)
//New Code
int sum = 0;
for (int i = 0; i < numbers.Length; i++)
```

Re-test the empty Array to ensure your changes have worked.

The next error is the overflow where the retuned value was larger than the maximum size of an integer. This can be fixed by increasing the size of the method return and the sum variable. Modify the code as follows,

```
//Original Code
static int Sum(int[] numbers)
int sum = 0;
//New Code
static long Sum(int[] numbers)
long sum = 0;
```

Re-test the Array with the large integers to ensure the code returns the correct result.

The last error is more difficult to fix and requires more re-coding. A method should return a valid result and NOT throw an exception; therefore any null Array should not crash the program, but report the problem. Modify your program by adding a TRY/CATCH around the body of the FOR loop.

```
      //New Code
static long Sum(int[] numbers)
{
    long sum = 0;
    try
    {
        for (int i = 0; i < numbers.Length; i++)
        {
            sum += numbers[i];
        }
        return sum;
    }
    catch (NullReferenceException)
    {
        MessageBox.Show("That didnt work");
        return 0;
    }
}
```

Re-test the Program to ensure this error has been fixed. A sound testing program will fix most bugs but only when the program is in production will all the error be uncovered. This activity has demonstrated what data needs to be used in a White Box test, as the programmer you are able to see the code and determine the boundary points and unusual conditions.

Once a test table has been created the results can be recorded via observation or to a log file. When logging the results of a testing session to a simple text file you must open and output the data directly to this file.

In the following assessment activity you will be required to code a simple program and implement a simple testing session. You will design and record the output to a table, before correcting the errors to the program.

White Box Testing

White-box testing is a verification technique software engineers can use to examine if their code works as expected. White-box testing is also known as structural testing, clear box testing, and glass box testing (Beizer, 1995). The connotations of “clear box” and “glass box” appropriately indicate that you have full visibility of the internal workings of the software product, specifically, the logic and the structure of the code.

With white-box testing, you must run the code with predetermined inputs and check to make sure that the code produces predetermined outputs.

Another way to devise a good set of white-box test cases is to consider the control flow of the program. The control flow of the sample program is represented in a flow diagram, as shown below in Figure 25.

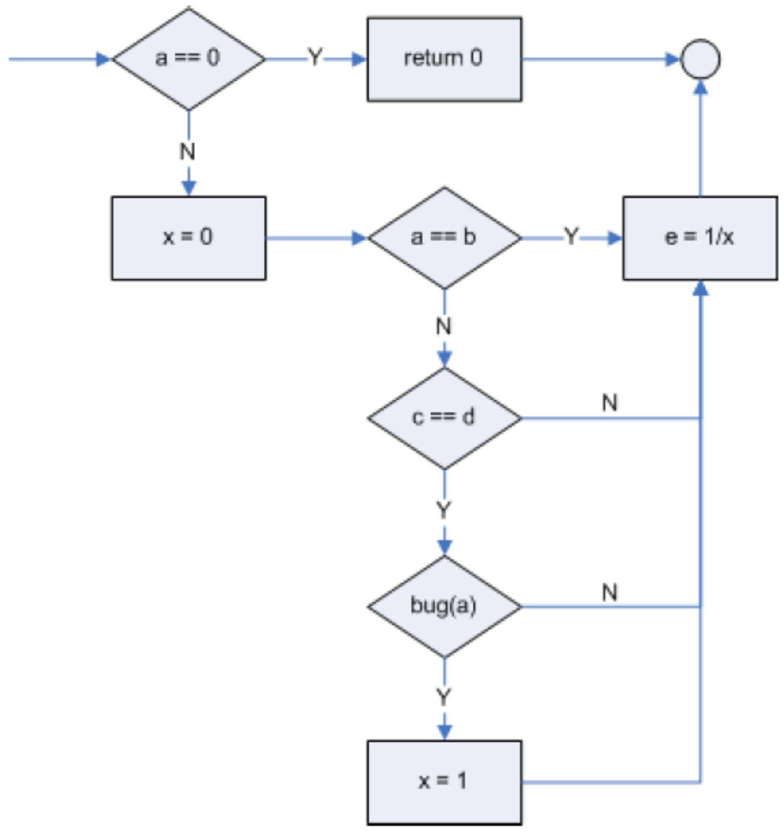


Figure 25

When you encounter a compound predicate, you must break the expression up so that each Boolean sub-expression is evaluated on its own.

The sample code from the flow diagram is shown below;

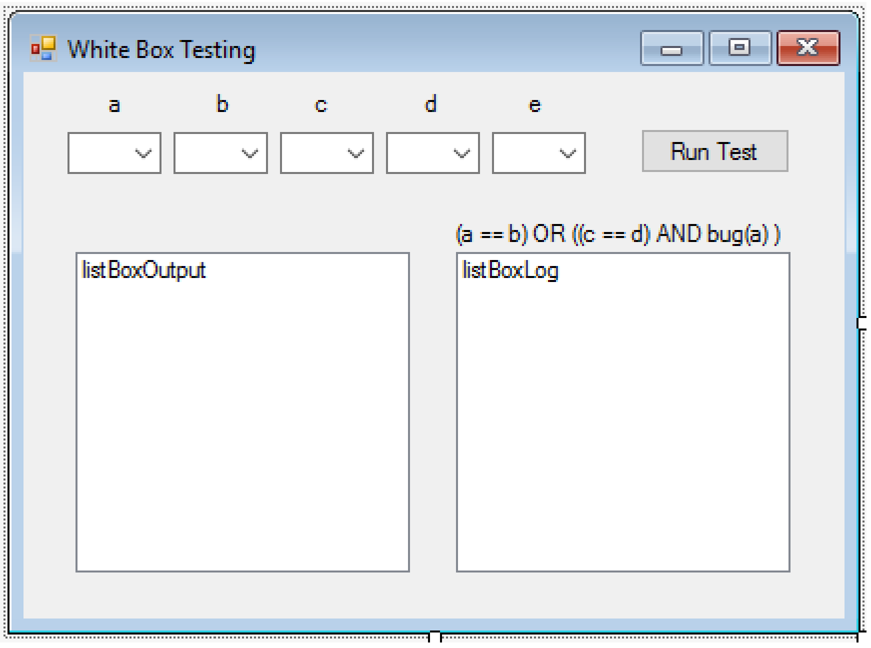
```
Line #      Code

1           int runCalculation (int a, int b, int c, int d, int e) {
2           if (a == 0) {
3                 return 0;
            }
4           int x = 0;
5           if ((a==b) OR ((c == d) AND bug(a) )) {
6                 x=1;
            }
7           e = 1/x;
8           return e;
            }
```

We need to ensure that each of the compound predicates are tested as both true and false. When we look at the code to calculate an expected return value, we realize that some test cases uncover a previously undetected division-by-zero problem. We can then immediately go to the code and protect from such an error. This illustrates the value of test planning to achieve 100% branch coverage through the flow diagram.

Programming Activity 44

In this activity you will implement the code to test the sample program shown in figure 25. Create a new Windows Forms Application and add the components as indicated below. Test the program using the test table input data and record your results and correct the code as required. The test data is only a sample, you will need to add additional values to fully test the program and all the compound conditions.

The input is via 5 combo boxes, while the output is through a list box. The additional list box will output the boolean result for the compound statement, which is the focus of this test. The test data is limited to values between 0 and 3 for each of the variables.



Form Text : White Box Testing
Button Text: Run Test Button (Name) : btnTest
ComboBox (Name) : comboBox_a ComboBox Items, Collections : 0, 1, 2, 3. (repeat this for each of the five combo boxes)
ListBox (Name) : listBoxOutput listBox (Name) : listBoxLog

In the following code we are testing the runCalculation method, so we will pass five integers to the method and return an integer value. Once you have error free code you can begin the test by adding the test code and then click the Run Button.

```
public Form1()
{
    InitializeComponent();
}
// true of false conditions for the IF statement
// compound conditions from Ln 5
bool cond1, cond2, cond3;

private void btnTest_Click(object sender, EventArgs e)
{
    // get integer values from the form
    int val_a = int.Parse(comboBox_a.Text);
    int val_b = int.Parse(comboBox_b.Text);
    int val_c = int.Parse(comboBox_c.Text);
    int val_d = int.Parse(comboBox_d.Text);
    int val_e = int.Parse(comboBox_e.Text);
```

```
        // declare output and call function
        int output = runCalculation(val_a, val_b, val_c, val_d, val_e);

        // display the input and return values
        listBoxOutput.Items.Add("Input ( " + val_a + ", " + val_b + ", " +
              val_c + ", " + val_d + ", " + val_e + " )" + " return " + output);

        // display the IF statements conditions
        listBoxLog.Items.Add("  " + cond1 + " OR ( " + cond2 + " AND " +
                                                           cond3 + " )");
    }



    private int runCalculation(int a, int b, int c, int d, int e)   // Ln 1
    {
        if (a == 0)                                        // Ln 2
        {
            return 0;                                      // Ln 3
        }
        int x = 0;                                         // Ln 4
        if ( (a == b) || ( (c == d) && bug(a) ))           // Ln 5
        {

            // evaluate the three boolean conditions
            cond1 = (a == b);
            cond2 = (c == d);
            cond3 = (bug(a));

            x = 1;                                         // Ln 6
        }
        e = 1 / x;                                         // Ln 7
        return e;                                          // Ln 8
    }

    private bool bug(int A)
    {
        if (A < 1)
            return true;
        else
            return false;
    }
```

Testing

In the first Test Case we enter runCalculation(0, 0, 0, 0, 0) and expect a return value of 0. If you look at the code you will see that if ‘a’ has a value of 0 it doesn’t matter what the other values are for ‘b’, ‘c’, ‘d’ or ‘e’. In this test we use 0 for the remaining values of ‘b’, ‘c’, ‘d’ and ‘e’.

Next we test for non-zero values of ‘a’. Below is the truth table for Ln 5 of the code. If you need to see the AND and OR truth tables they are located at the end of Appendix B. From the code and the OR truth table the following Test Case will always return 1: runCalculation(1, 1, x, x, 1).

Input	A== B OR	C==D && Bug(A)		Result of C==D && Bug(A)	Final A== B OR Result
0,0,0,0,0	T	NA	NA	NA	NA
1,1,1,1,1	T	T	F	F	T
1,2,1,1,1	F	T	F	F	F
1,2,1,2,1	F	F	F	F	F
3,2,1,1,1	F	T	F	F	F

We need condition coverage testing, to ensure that each of these sub-expressions has independently been tested as both true and false.

We write the following Test Case to address test (c==d) as true: runCalculation(1, 2, 1, 1, 1), expected return value 1. However, when we actually run the test case, the function bug(a) actually returns false, which causes our actual return value (division by zero) to not match our expected return value. This allows us to detect an error in the bug method. Without the addition of condition coverage, this error may not have been revealed.

Examine the code and the bug(a) information, which informs us that the bug method should return a false value if fed any integer greater than 1. So we create Test Case 5, runCalculation (3, 2, 1, 1, 1), expected return value “division by error”.

To correct this program you will need to reverse the return values form the bug() method.

```
private bool bug(int A)
{
        if (A >= 1)
                return true;
        else
                return false;

}
```

Run the Test Code and record your results, remember test case runCalculation(0, 0, 0, 0, 0) will always return 0.

Input	A== B OR	C==D AND Bug(A)		Results from AND	Final compound result
0,0,0,0,0	T	T	F	F	T
1,1,1,1,1	T	T	T	T	T
1,2,1,1,1					
1,2,1,2,1					
3,2,1,1,1					

Finally, fix the "division by zero' error for the runCalculation(1,2, 1, 2, 1) test case, this can be done in several ways; with a simple try/catch around Ln 7 or changing the AND operator to an OR operator `( (c == d) || bug(a) )`. Examine the code and decide how you might fix this logic error.

Assessment Activity AT1.4

You should now attempt Assessment Portfolio AT1.4 which consists of a programming logic task using Windows Forms Application. The assessment can be downloaded from e-campus.

REFERENCES

This text was complied with extracts from the following resources;

Brown, E. (2002). Windows Forms Programming with C#.

Carney, K. (2012). Visual C# .NET

Ferguson, J., Patterson, B. & Beres, J. (2002). C# Bible.

Mayo, J. (2010). Microsoft Visual Studio 2010 A Beginners Guide.

Mehra, P.S. (2011). Programming C# for Beginners

Microsoft MSDN (2002). MSDN Training Programming with C#

Miles, R. (2014). C# Programming

Nakov, S. & Kolev, V. (2013). FUNDAMENTALS OF COMPUTER PROGRAMMING WITH C#

Petzol, C. (2007). .NET Book Zero

Powers, L. & Snell, M. (2008) Microsoft Visual Studio 2008 UNLEASHED.

Pseudocode Basics (2014).

Stellman, A. & Greene, J. (2013). Head First C#, 3rd Ed.

Web Resources

http://csharp.net-informations.com/gui/cs_forms.htm

http://msdn.microsoft.com/en-us/library/vstudio/dd492132.aspx

http://www.c-sharpcorner.com/1/76/windows-forms-C-Sharp.aspx

http://msdn.microsoft.com/en-us/library/aa260844%28v=vs.60%29.aspx#cfr_names

http://en.wikipedia.org/wiki/Coding_conventions

http://msdn.microsoft.com/en-us/library/ff926074.aspx

APPENDIX A

Pseudo Code

Flowcharts were the first design tool to be widely used, but unfortunately they do not reflect some of the concepts of structured programming. Pseudo code, on the other hand, is a newer tool and has features that make it more reflective of the structured concepts.

Rules for Pseudo Code

1. Write only one statement per line

Each statement in your pseudo code should express just one action for the computer. If the task list is properly drawn, then in most cases each task will correspond to one line of pseudo code.

EX: TASK LIST:

```
Read name, hourly rate, hours worked, deduction rate
Perform calculations
     gross = hourlyRate * hoursWorked
     deduction = grossPay * deductionRate
     net pay = grossPay – deduction
Write name, gross, deduction, net pay
```

PSEUDOCODE:

```
READ name, hourlyRate, hoursWorked, deductionRate
grossPay = hourlyRate * hoursWorked
deduction = grossPay * deductionRate
netPay = grossPay – deduction
WRITE name, grossPay, deduction, netPay
```

2. Capitalize initial keyword

In the example above, **READ** and **WRITE** are in caps. There are just a few keywords we will use: **READ, WRITE, IF, ELSE, ENDIF, WHILE, ENDWHILE, REPEAT, UNTIL**

3. Indent to show hierarchy

We use a particular indentation pattern in each of the design structures:

SEQUENCE: keep statements that are "stacked" in sequence all starting in the same column.

SELECTION: indent the statements that fall inside the selection structure, but not the keywords that form the selection

LOOPING: indent the statements that fall inside the loop, but not the keywords that form the loop

EX: In the example below, employees whose *grossPay* is less than 100 do not have any deduction.

TASK LIST:

```
Read name, hourly rate, hours worked, deduction rate
Compute gross, deduction, net pay
     Is gross >= 100?
          YES: calculate deduction
          NO: no deduction
Write name, gross, deduction, net pay
```

PSEUDOCODE:

```
READ name, hourlyRate, hoursWorked
grossPay = hourlyRate * hoursWorked
IF grossPay >= 100
     deduction = grossPay * deductionRate
ELSE
     deduction = 0
ENDIF
netPay = grossPay - deduction
WRITE name, grossPay, deduction, netPay
```

4. End multiline structures

See how the IF/ELSE/ENDIF is constructed above. The ENDIF (or END whatever) always is in line with the IF (or whatever starts the structure).

5. Keep statements language independent

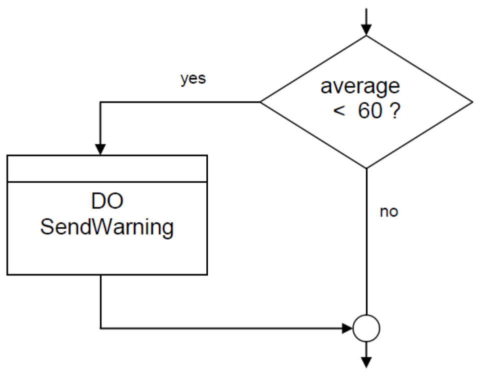
Resist the urge to write in whatever language you are most comfortable with. In the long run, you will save time! There may be special features available in the language you plan to eventually write the program in; if you are SURE it will be written in that language, then you can use the features. If not, then avoid using the special features.

Programming Structures

There are two basic selection constructs; the IF THEN ELSE and the SWITCH / CASE. These two constructs have several variations but the basic idea is still the same, to offer alternative pathways through a program.

Selection Structure

The basic selection structure has a simple outcome, "do something" or "do nothing". The pseudo code for this is:



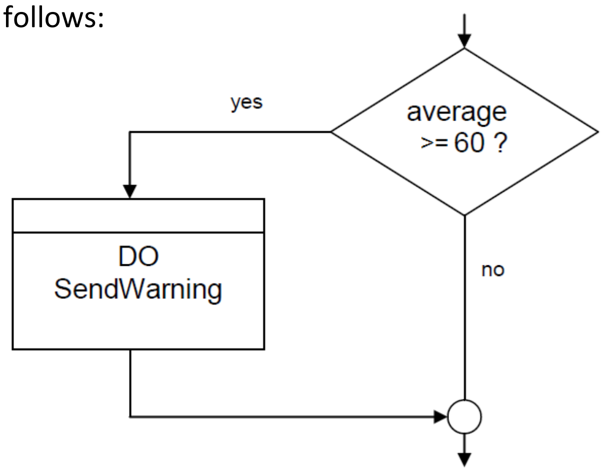
```
IF average < 60
     DO SendWarning
ENDIF;
```

It is considered **poor programming style** to have a 1-sided IF statement where the action is on the "no" or ELSE side. Consider this code:

```
IF average < 60
     NULL
ELSE
     DO GivePassingGrade
ENDIF
```

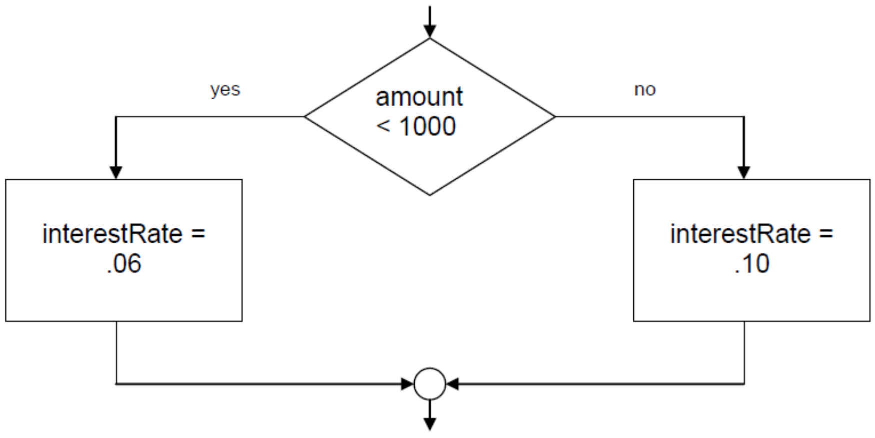
This **should** be rewritten to eliminate the NULL “yes” part. To do that, we change the < to its opposite: >= as follows:

```
IF average >= 60
     DO GivePassingGrade
ENDIF
```



The typical selection statement has two alternatives, one for the Yes (True) scenario and one for a No (False) scenario. The **pseudo code** for this would be:

```
IF amount < 1000
     interestRate = .06  //the “yes” or “true” action
ELSE
     interestRate = .10  //the “no” or “false” action
ENDIF
```



Nesting IF Statements

When more than two alternatives are required a nested IF statement will provide a solution

What if we wanted to put a little menu up on the screen,

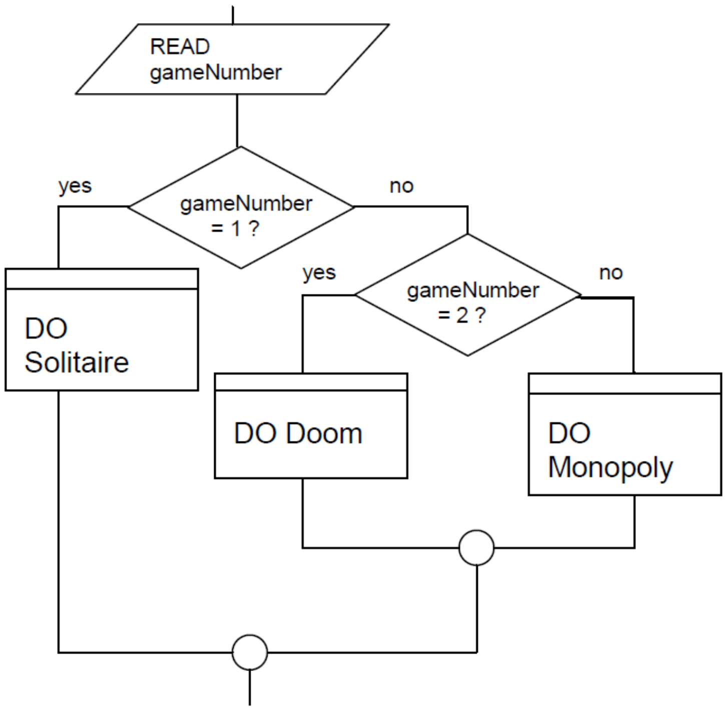
1. Solitaire

2. Doom

3. Monopoly

and have the user select which game to play. How would we activate the correct game?

```
READ gameNumber
IF gameNumber = 1
     DO Solitaire
ELSE
     IF gameNumber = 2
          DO Doom
     ELSE
          DO Monopoly
     ENDIF
ENDIF
```

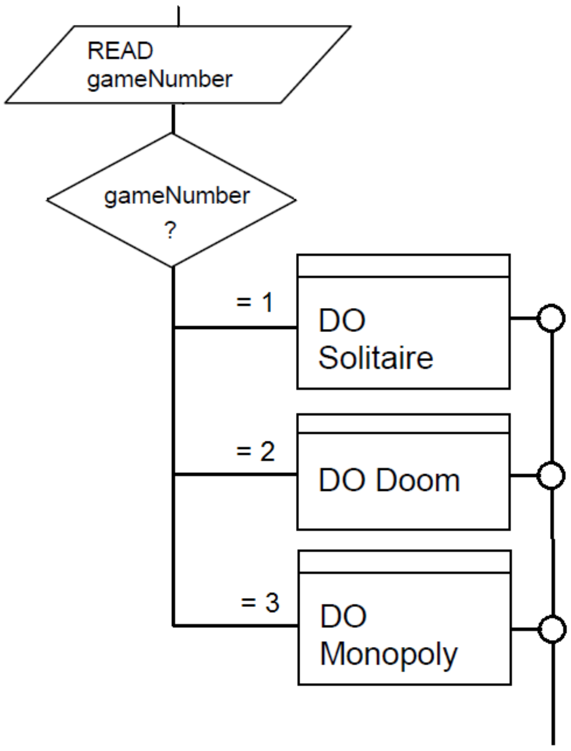


We must pay particular attention to where the IFs end. The nested IF must be completely contained in either the IF or the ELSE part of the containing IF. Watch for and line up the matching ENDIF.

The alternative to using a nested IF statement is the SWITCH / CASE statement. In this construct the variable is evaluated once and the appropriate action is selected. The default option is the “catch all” when an incorrect option is selected for evaluation.

The **pseudo code** for the Switch / Case construct would be:

```
READ gameNumber
SWITCH (gameNumber)
     CASE 1 : DO Solitaire
     CASE 2 : DO Doom
     CASE 3 : DO Monopoly
     DEFAULT : Error Message
END SWITCH
```



Looping Structures

One of the most confusing things for students first seeing a flowchart is telling the loops apart from the selections. This is because both use the diamond shape as their control symbol. In pseudo code this confusion is eliminated. To mark our loops we will use these pairs:

WHILE / ENDWHILE,

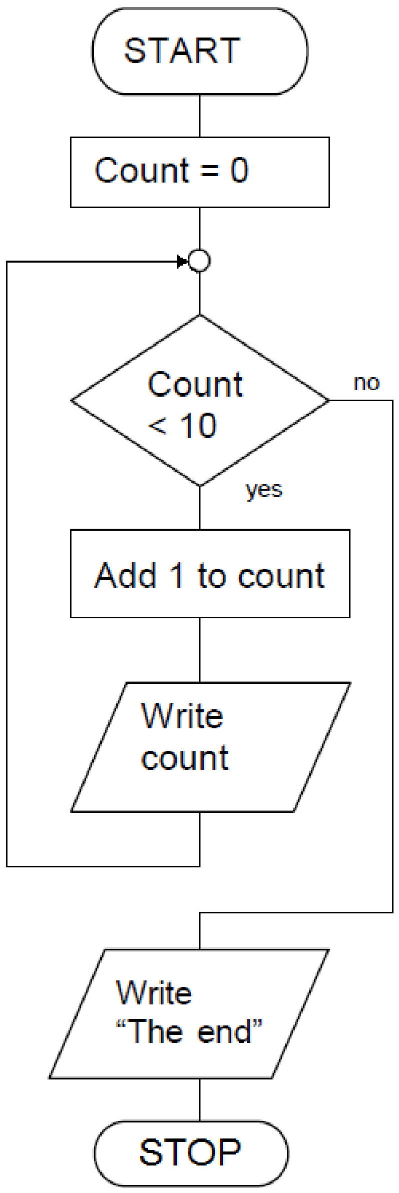
REPEAT / UNTIL,

FOR / ENDFOR

While Loop

The WHILE loop will have the following **pseudo code**:

```
count = 0
WHILE count < 10
      ADD 1 to count
      WRITE count
ENDWHILE
WRITE "The end"
```



Notice that the connector and test at the top of the loop in the flowchart become the WHILE statement in pseudo code. The end of the loop is marked by ENDWHILE.

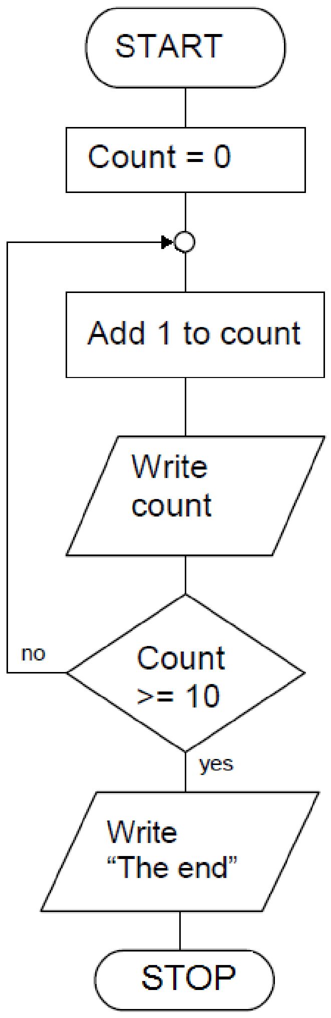
Sometimes it is desirable to put the steps that are inside the loop into a separate method. Then the **pseudo code** might be this:

```
//We often use this name for the first module.
Mainline
//Initialization comes first
count = 0
WHILE count < 10
        //The processing loop uses this module
        DO Process
ENDWHILE
//Termination does clean-up
WRITE "The end"

//Go through these steps and then return
//to the method that sent you here (Mainline)
Process
ADD 1 to count
WRITE count
```

Repeat Loop

This time we will see how to write pseudo code for an REPEAT loop:

```
count = 0
REPEAT
        ADD 1 to count
        WRITE count
UNTIL count >= 10
WRITE "The end"
```

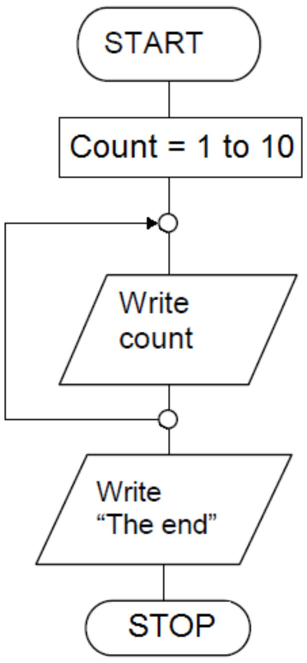


Notice how the connector at the top of the loop corresponds to the REPEAT keyword, while the test at the bottom corresponds the UNTIL statement. When the loop is over, control goes to the statement following the UNTIL.

For Loop

The FOR loop is often referred to as a fixed loop. This is because the loop will have a fixed number of iterations so the actions will be performed a fixed number of times. The FOR loop will have the following **pseudo code**:

```
FOR count = 1 to 10
     Write count
ENDFOR
WRITE "The end"
```



Notice the connector at the bottom of the loop is not tested; since this is a fixed loop the number of iterations has been determined at the top of the loop. Once the loop has finish, control goes to the statement following the ENDFOR.

Advantages and Disadvantages

Pseudo code Disadvantages

- It’s not visual
- There is no accepted standard, so it varies widely from company to company

Pseudo code Advantages

- Can be done easily on a word processor
- Easily modified
- Implements structured concepts well

Flowchart Disadvantages

- Hard to modify
- Need special software
- Structured design elements not all implemented

Flowchart Advantages

- Standardized: most symbols and their meaning have an agreed meaning
- Visual (but this does bring some limitations)

How we Store and Access Data

What happens on a computer when we execute a READ statement? For example when we READ firstName, hoursWorked, hourlyRate, deductionRate, the computer stores all program data into memory locations. It knows these locations are by their hexadecimal memory addresses. These memory addresses are not known to the programmer and would be difficult to remember. For example the firstName might be located at memory address #FF00234A. Programmers use descriptive names for their code and let the computer decide which memory location to use. Every language has its own (different) set of rules about how these names are formed**. We will use a simple style**:

- variable names will start with a lowercase letter
- they will have no spaces in them
- additional words in them will start with a capital
- names must be unique within the program
- consistent use of names

The READ statement tells the computer to get a value from the input device (keyboard, file, …) and store it in the names memory location. When we need to compute a value in a program (like *grossPay*) we will use an assignment statement. variable = expression. Be careful to understand the difference between these two statements:

```
num1 = num2
num2 = num1
```

The WRITE statement is used to display information on the default output device (screen). To display words, enclose them in quotes. A variable's value will be displayed. So if the variable firstN*ame* currently contains John Smith, then the statement

```
WRITE "Employee name: ", firstName
```

will output like this:

```
Employee name: John Smith
```

Calculation Symbols

We will often have to represent an expression like the one that computes *grossPay*. To symbolize the arithmetic operators we use these symbols

grouping ( )

exponent ** or ^

multiply * divide /

add + subtract -

Order of Execution

There is a precedence or hierarchy implied in these symbols.

( ) equations in parenthesis

** exponentiation

/ * division and multiplication

+ - addition and subtraction

Note: when operators of equal value are to be executed, the order of execution is left to right.

APPENDIX B

Boolean Symbols

When we have to make a choice between actions, we almost always base that choice on a test. The test uses phrases like "is less than" or "is equal to". There is a universally accepted set of symbols used to represent these phrases:

> (greater than)

< (less than)

>= (greater than or equal to)

<= (less than or

= (equal to)

<> (not equal to)

It is interesting to notice that these can be paired up:

SYMBOL	IS OPPOSITE TO
>	<=
<	>=
=	<>

Logical Operators: And, Or, Not

AND: if any of the conditions are false, the whole expression is false.

```
IF day = "Saturday" AND weather = "sunny"
     WRITE "Go to the beach!"
ENDIF
```

OR: if any of the conditions are true, the whole expression is true

```
IF month = "June" OR month = "July" OR month = "August"
     WRITE "Winter time!"
ENDIF
```

NOT: reverses the outcome of the expression; true becomes false, false becomes true.

```
IF day <> "Saturday" AND day <> "Sunday"
     WRITE "This is a work day"
ENDIF
```

Truth Tables for AND and OR

The AND truth table

p	***q***	***p* AND *q***
T	T	T
T	F	F
F	F	F

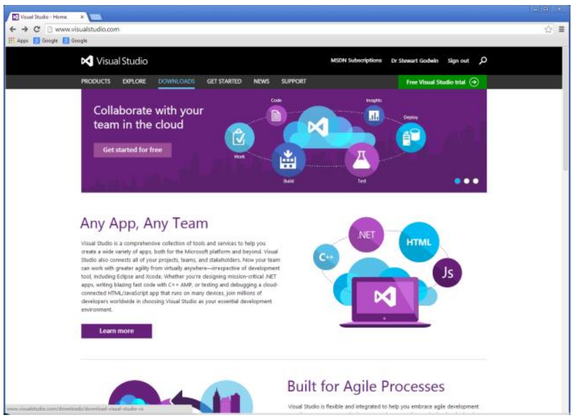
The OR truth table

p	***q***	***p* OR *q***
T	T	T
F	T	T
F	F	F

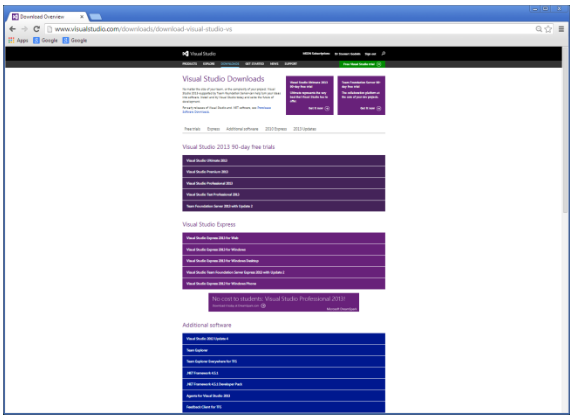
APPENDIX C

Getting Started

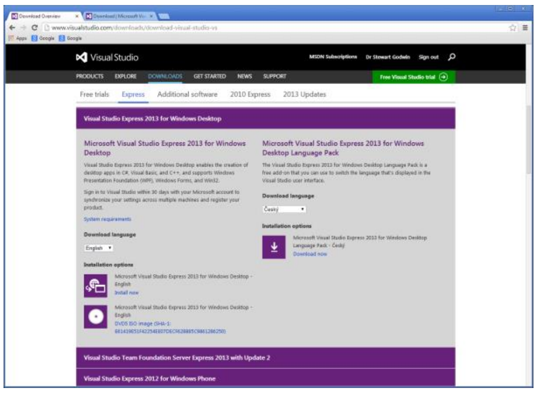
Download a suitable version of Visual Studio. Visit the Microsoft Visual Studio webs site and click on the DOWNLOADS menu option.

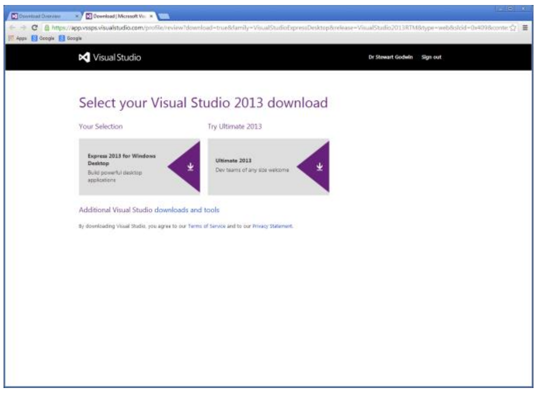

Visual Studio
PRODUCTS EXPLORE DOWNLOADS GET STARTED NEWS SUPPORT
Collaborate with your team in the cloud
Get started for free
Any App, Any Team
Learn more
Built for Agile Processes

There are several different versions and you will need to select one that suits your projects. For this course we suggest the Visual Studio Express2013 for Windows Desktop.

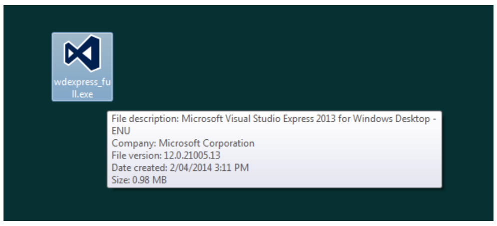


Click on the version your require, (don’t forget to choose your language) this will open up an new browser window. If you don’t have an account I suggest you create one as it may be useful in the future.


Visual Studio Express 2013 for Windows Desktop
Microsoft Visual Studio Express 2013 for Windows Desktop
Microsoft Visual Studio Express 2013 for Windows Desktop Language Pack
Download language
Installation options
Install now
Download now


Select your Visual Studio 2013 download
Your Selection
Try Ultimate 2013
Express 2013 for Windows Desktop
Ultimate 2013
Additional Visual Studio downloads and tools

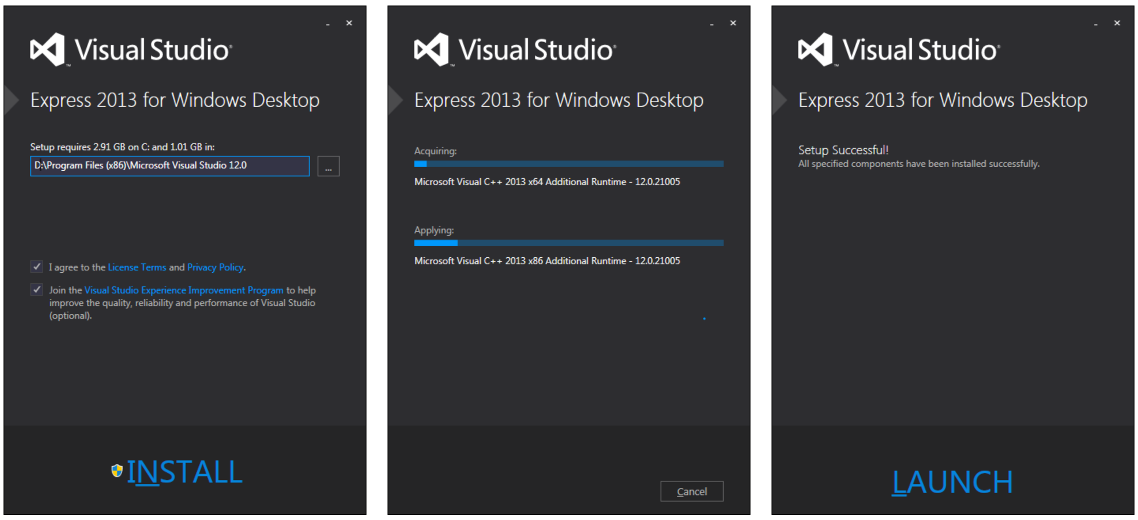
Now click your download option and save the file to the desktop.



File description: Microsoft Visual Studio Express 2013 for Windows Desktop - ENU
Company: Microsoft Corporation
File version: 12.0.21005.13
Date created: 2/04/2014 3:11 PM
Size: 0.98 MB

Installing the Software

Launch the installer and follow the instructions. You may require administrator rights to install this software. Check the install destination and check the "I agree to the..." check box. Now click the Install area at the bottom of the dialog box.



Express 2013 for Windows Desktop

Setup requires 2.91 GB on C: and 1.01 GB in:
D:\Program Files (x86)\Microsoft Visual Studio 12.0

✓ I agree to the License Terms and Privacy Policy.
✓ Join the Visual Studio Experience Improvement Program to help improve the quality, reliability and performance of Visual Studio (optional).

Express 2013 for Windows Desktop

Acquiring:
Microsoft Visual C++ 2013 x64 Additional Runtime - 12.0.21005

Applying:
Microsoft Visual C++ 2013 x86 Additional Runtime - 12.0.21005

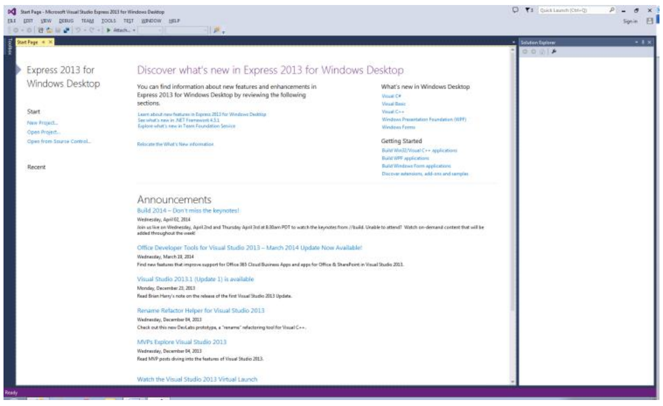
Express 2013 for Windows Desktop

Setup Successful!
All specified components have been installed successfully.

The installation may take a few minutes as the files are downloaded and installed onto your hard drive. Finally, click the LAUNCH area to start the program.

Using Visual Studio

You can sign in but that is optional. Once the program has opened you will have several option to create and build desktop apps. During this course you will use the console and the windows options to create simple apps that demonstrate the basic programming constructs.


Start Page - Microsoft Visual Studio Express 2013 for Windows Desktop
FILE EDIT VIEW DEBUG TEAM TOOLS TEST WINDOW HELP
Attach...
Quick Launch (Ctrl+Q)
Sign in
Toolbox
Start Page
Solution Explorer
Ready

Express 2013 for Windows Desktop

Start
New Project...
Open Project...
Open from Source Control...

Recent

Discover what's new in Express 2013 for Windows Desktop

You can find information about new features and enhancements in Express 2013 for Windows Desktop by reviewing the following sections.

Learn about new features in Express 2013 for Windows Desktop
See what's new in .NET Framework 4.5.1
Explore what's new in Team Foundation Service

Relocate the What's New information

What's new in Windows Desktop
Visual C#
Visual Basic
Visual C++
Windows Presentation Foundation (WPF)
Windows Forms

Getting Started
Build Win32/Visual C++ applications
Build WPF applications
Build Windows Form applications
Discover extensions, add-ons and samples

Announcements

Build 2014 – Don't miss the keynotes!
Wednesday, April 02, 2014
Join us live on Wednesday, April 2nd and Thursday April 3rd at 8:30am PDT to watch the keynotes from //build. Unable to attend? Watch on-demand content that will be added throughout the week!

Office Developer Tools for Visual Studio 2013 – March 2014 Update Now Available!
Wednesday, March 19, 2014
Find new features that improve support for Office 365 Cloud Business Apps and apps for Office & SharePoint in Visual Studio 2013.

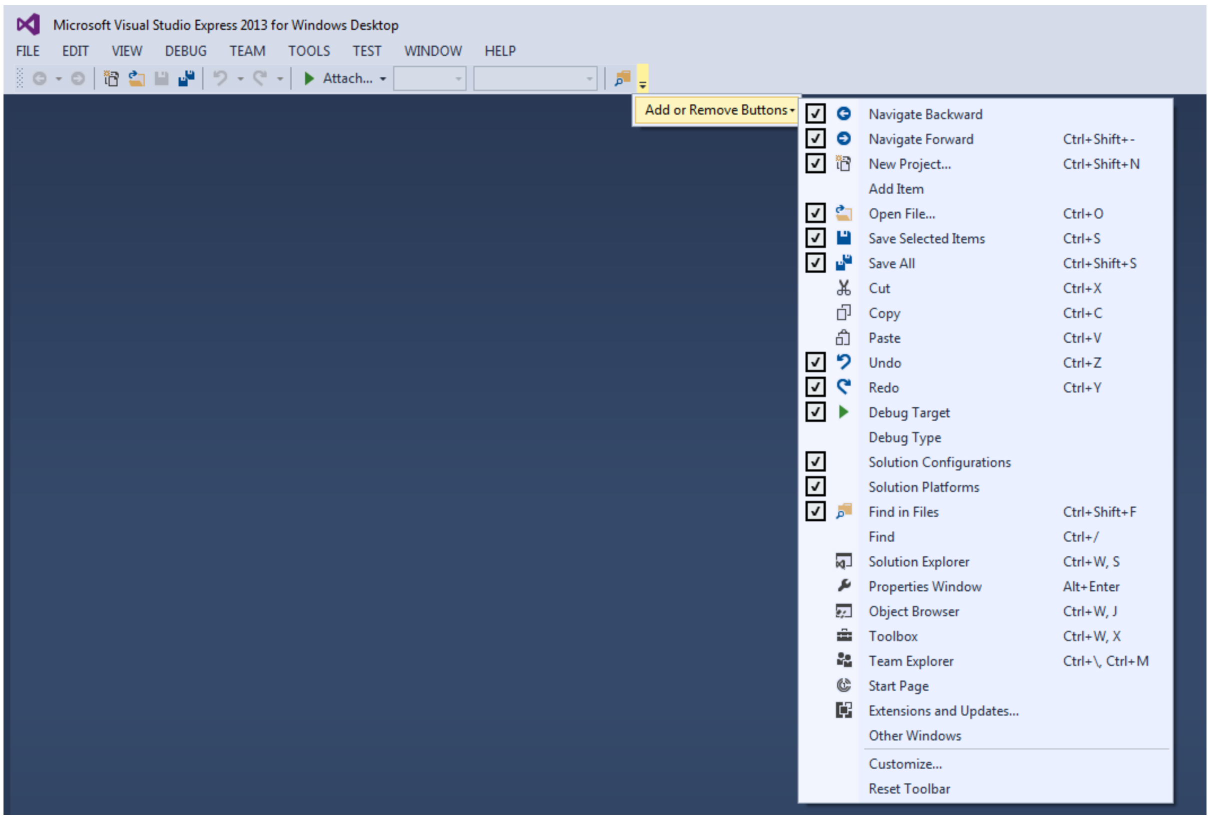
Visual Studio 2013.1 (Update 1) is available
Monday, December 23, 2013
Read Brian Harry's note on the release of the first Visual Studio 2013 Update.

Rename Refactor Helper for Visual Studio 2013
Wednesday, December 04, 2013
Check out this new DevLabs prototype, a "rename" refactoring tool for Visual C++.

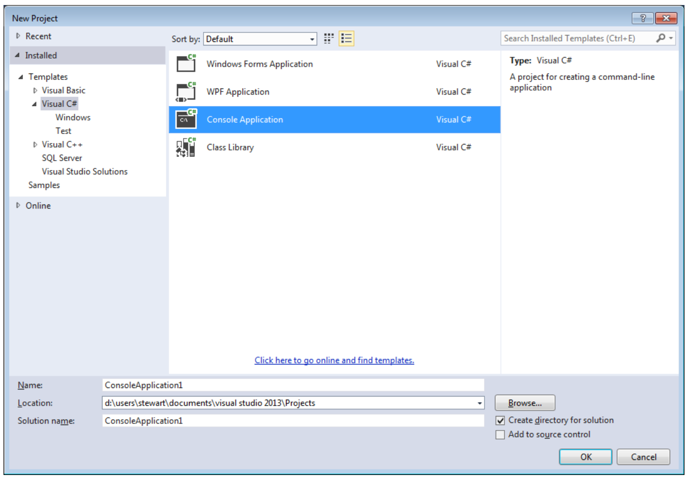
MVPs Explore Visual Studio 2013
Wednesday, December 04, 2013
Read MVP posts diving into the features of Visual Studio 2013.

Watch the Visual Studio 2013 Virtual Launch

You can close the Start Page and then add or remove different options as shown below.


Microsoft Visual Studio Express 2013 for Windows Desktop
FILE EDIT VIEW DEBUG TEAM TOOLS TEST WINDOW HELP
Attach...
Add or Remove Buttons
Navigate Backward
Navigate Forward Ctrl+Shift+-
New Project... Ctrl+Shift+N
Add Item
Open File... Ctrl+O
Save Selected Items Ctrl+S
Save All Ctrl+Shift+S
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Undo Ctrl+Z
Redo Ctrl+Y
Debug Target
Debug Type
Solution Configurations
Solution Platforms
Find in Files Ctrl+Shift+F
Find Ctrl+/
Solution Explorer Ctrl+W, S
Properties Window Alt+Enter
Object Browser Ctrl+W, J
Toolbox Ctrl+W, X
Team Explorer Ctrl+\, Ctrl+M
Start Page
Extensions and Updates...
Other Windows
Customize...
Reset Toolbar

To start a new project click FILE > NEW PROJECT and then select TEMPLATES > VISUAL C#. During this course you will be using both the WINDOWS FORMS APPLICATIONS and CONSOLE APPLICATIONS. Select the CONSOLE APPLICATIONS and then rename the project in the NAME: Textbox, then click the OK button.



Type: Visual C#
A project for creating a command-line application

Name: ConsoleApplication1
Location: d:\users\stewart\documents\visual studio 2013\Projects
Solution name: ConsoleApplication1
[✓] Create directory for solution
[ ] Add to source control

You can now start working through the activities and assessment projects.

APPENDIX D

Implementing a Queue

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

namespace book
{
    public partial class Form5 : Form
    {
        public Form5()
        {
            InitializeComponent();
        }
        //create a queue of items
        Queue myTasks = new Queue();

        private void btnAddItem_Click(object sender, EventArgs e)
        {
            if (!(String.IsNullOrEmpty(textNewItem.Text)))
            {
                myTasks.Enqueue(textNewItem.Text);
                textNewItem.Clear();
                displayItems();
            }
            else
            {
                MessageBox.Show("Nothing to add");
            }
        }//end of btnAddItem_Click

        private void btnRemoveItem_Click(object sender, EventArgs e)
        {
            if (myTasks.Count > 0)
            {
                myTasks.Dequeue();
                displayItems();
            }
            else
            {
                MessageBox.Show("Nothing to remove");
            }
        }//end of btnRemoveItem_Click

        private void displayItems()
        {
            listItems.Items.Clear();
            foreach (Object obj in myTasks)
            {
                listItems.Items.Add(obj);
            }
        }//end of method displayItems

    }
}
```

APPENDIX E

Optional Activity Solutions

Act-5: Test if number is equal to a target value: using IF statement

```
static void Main(string[] args)
{
    Console.WriteLine("is equal");
    int Number = int.Parse(Console.ReadLine());
    if (Number == 10)
    {
        Console.WriteLine("the variable Number is equal to 10");
    }

}
```

Act-6: Test if number is equal to a target value: using IF-ELSE statement

```
static void Main(string[] args)
{
    Console.WriteLine("The If condition");
    int Number = 11;
    bool IsNumber10;

    if (Number == 10)
    {
        IsNumber10 = true;
    }
    else
    {
        IsNumber10 = false;
    }

    Console.WriteLine("Number == 10 is {0}", IsNumber10);

}
```

Act-7: Test if a number is equal to a target value: using the TERNARY operator

```
static void Main(string[] args)
{
    Console.WriteLine("The If condition");
    int Number = 10;

    bool IsNumber10 = Number == 10 ? true : false;

    Console.WriteLine("Number == 10 is {0}", IsNumber10);

}
```

ACT-8: Use a sequence of IF statements to test an input value

```
static void Main(string[] args)
{
    Console.WriteLine("The If statement\n");
    Console.WriteLine("Please enter a number");

    int UserNumber = int.Parse(Console.ReadLine());

    if (UserNumber == 1)
    {
        Console.WriteLine("The number is  One >> {0}", UserNumber);
    }
    if (UserNumber == 2)
    {
        Console.WriteLine("The number is  Two >> {0}", UserNumber);
    }
    if (UserNumber == 3)
    {
        Console.WriteLine("The number is  Three >> {0}", UserNumber);
    }

}
```

Act-9: Use a sequence of IF statements to test an input value with compound condition for error trapping, Use AND (&&) operator

```
    static void Main(string[] args)
    {
        Console.WriteLine("The If statement\n");
        Console.WriteLine("Please enter a number");

        int UserNumber = int.Parse(Console.ReadLine());

        if (UserNumber == 1)
        {
            Console.WriteLine("The number is  One >> {0}", UserNumber);
        }
        if (UserNumber == 2)
        {
            Console.WriteLine("The number is  Two >> {0}", UserNumber);
        }
        if (UserNumber == 3)
        {
            Console.WriteLine("The number is  Three >> {0}", UserNumber);
        }
        if (UserNumber != 1 && UserNumber != 2 && UserNumber != 3)
        {
            Console.WriteLine("your number was Not 1, 2 or 3");
        }
    }
```

Act-10: Re-write Act-9 using IF-ELSE statements

```
    static void Main(string[] args)
    {
        Console.WriteLine("The If statement\n");
        Console.WriteLine("Please enter a number");

        int UserNumber = int.Parse(Console.ReadLine());

        if (UserNumber == 1)
        {
            Console.WriteLine("The number is  One >> {0}", UserNumber);
        }
            else if (UserNumber == 2)
            {
                Console.WriteLine("The number is  Two >> {0}", UserNumber);
            }
                else if (UserNumber == 3)
                {
                    Console.WriteLine("The number is  Three >> {0}", UserNumber);
                }
        else
        {
            Console.WriteLine("your number was Not 1, 2 or 3");
        }
    }
```

Act-11: Use OR (||) operator in IF statement condition

```
    static void Main(string[] args)
    {
        Console.WriteLine("The If statement\n");
        Console.WriteLine("Please enter a number");
```

```
            int UserNumber = int.Parse(Console.ReadLine());

            if (UserNumber == 10 || UserNumber == 20)
            {
               Console.WriteLine("The number is  10 or 20 >> {0}", UserNumber);
            }
            else
            {
               Console.WriteLine("The number is Not 10 or 20 it was {0}", UserNumber);
            }

        }
```

Act-12: Re-write Act-10 as a SWITCH-CASE statement

```
        static void Main(string[] args)
        {
            Console.WriteLine("The CASE or SWITCH statement\n");
            Console.WriteLine("Please enter a number");

            int UserNumber = int.Parse(Console.ReadLine());

            switch (UserNumber)
            { //start of switch
                case 10: Console.WriteLine("Your number is 10");
                    break;
                case 20: Console.WriteLine("Your number is 20");
                    break;
                case 30: Console.WriteLine("Your number is 30");
                    break;
                default: Console.WriteLine("Your number is not 10, 20 or 30");
                    break;
            }
        }
```

Act-13: Modify Act-12 to use "fall throu" CASE option

```
        static void Main(string[] args)
        {
            Console.WriteLine("The If statement\n");
            Console.WriteLine("Please enter a number");

            int UserNumber = int.Parse(Console.ReadLine());

            switch (UserNumber)
            {
                case 10:
                case 20:
                case 30: Console.WriteLine("Your number is {0}", UserNumber);
                    break;
                default: Console.WriteLine("Your number is not 10, 20 or 30");
                    break;
            }
        }
```

Act-14: Simple menu using CASE "fall throu" and CHAR input

```
class Program
{
    public static void Main()
    {
        string myChoice;
        Program om = new Program();
        do
       {
            myChoice = om.getChoice();
            // Make a decision based on the user's choice
            switch(myChoice)
            {
                case "A":
                case "a":
                    Console.WriteLine("You wish to add an address.");
                    break;
                case "D":
                case "d":
                    Console.WriteLine("You wish to delete an address.");
                    break;
                case "M":
                case "m":
                    Console.WriteLine("You wish to modify an address.");
                    break;
                case "V":
                case "v":
                    Console.WriteLine("You wish to view the address list.");
                    break;
                case "Q":
                case "q":
                    Console.WriteLine("Bye.");
                    break;
                default:
                    Console.WriteLine("{0} is not a valid choice", myChoice);
                    break;
            }
            // Pause to allow the user to see the results
            Console.WriteLine();
            Console.Write("press Enter key to continue...");
            Console.ReadLine();
            Console.WriteLine();
        } while (myChoice != "Q" && myChoice != "q"); // Keep going until the user wants to quit
    }

    string getChoice()
    {
        string myChoice;

        // Print A Menu
        Console.WriteLine("My Address Book\n");
        Console.WriteLine("A - Add New Address");
        Console.WriteLine("D - Delete Address");
        Console.WriteLine("M - Modify Address");
        Console.WriteLine("V - View Addresses");
        Console.WriteLine("Q - Quit\n");
        Console.Write("Choice (A,D,M,V,or Q): ");
        // Retrieve the user's choice
        myChoice = Console.ReadLine();
        Console.WriteLine();
        return myChoice;
    }
}
```

Act-15: Loop for a user defined number of times

```
static void Main(string[] args)
        {
            Console.WriteLine("The While construct\n");
            Console.WriteLine("Please enter a target number");
            int UserTargert = int.Parse(Console.ReadLine());
            int Start = 0;
            while (Start <= UserTargert)
            {
                //Console.Write("{0}",Start, " ");
                //Console.Write("{0} ",Start);
                Console.Write(Start + " ");
                Start = Start + 3;
            }
        }
```

Act-16: Put Act-14 inside a DO-WHILE user defined loop

```
        static void Main(string[] args)
        {
            Console.WriteLine("The Do While construct\n");

            string UserChoice = string.Empty;
            //Declare loop terminator and set to empty

            do
            {
                Console.WriteLine("Please enter a target number");
                int UserTargert = int.Parse(Console.ReadLine());
                int Start = 0;

                while (Start <= UserTargert)
                {
                    Console.Write(Start + " ");
                    //Console.Write(Start + " ");
                    Start = Start + 2;
                }

                do
                {
                    Console.WriteLine("Do you want to contiue? Yes or No");
                    UserChoice = Console.ReadLine();
                    if (UserChoice != "yes" && UserChoice != "no")
                    {
                        Console.WriteLine("Invalid Choice, Enter yes or no");
                    }
                } while (UserChoice != "yes" && UserChoice != "no");
            } while (UserChoice == "yes");
        }
```

Act-17: Make an array of three element which are pre-filled

```
        static void Main(string[] args)
        {
            int[] Numbers = new int[3];
            Numbers[0] = 101;
            Numbers[1] = 102;
            Numbers[2] = 103;
            int i = 0;
            while (i < Numbers.Length)
            {
                Console.WriteLine(Numbers[i]);
                i++;
            }

        }
```

Act-18: Use three loops to traverse the array from Act-16; use FOREACH, FOR, WHILE.

```
        static void Main(string[] args)
        {
            int[] Numbers = new int[3];
            Numbers[0] = 101;
            Numbers[1] = 102;
            Numbers[2] = 103;
            Console.WriteLine("The foreach loop");
            foreach (int k in Numbers)
            {
                Console.WriteLine(k);
            }
            Console.WriteLine("\nThe for loop");
            for (int j = 0; j < Numbers.Length; j++)
            {
                Console.WriteLine(Numbers[j]);
            }
            Console.WriteLine("\nThe while loop");
            int i = 0;
            while (i < Numbers.Length)
            {
                Console.WriteLine(Numbers[i]);
                i++;
            }
        }
```

Act-19: Create two arrays using two different pre-filled options. Traverse using FOR loop

```
        static void Main(string[] args)
        {
            int[] EvenNumbers = new int[3];
            int[] OddNumbers = {3, 5, 7};

            EvenNumbers[0] = 2;
            EvenNumbers[1] = 4;
            EvenNumbers[2] = 6;

            for (int j = 0; j < EvenNumbers.Length; j++)
            {
                Console.WriteLine(EvenNumbers[j]);
            }

            for (int k = 0; k < EvenNumbers.Length; k++)
            {
                Console.WriteLine(OddNumbers[k]);
            }
        }
```

Act-20: Create an array and fill using FOR loop, display using a second FOR loop

```
        static void Main(string[] args)
        {
            int[] EvenNumbers = { 0, 0, 0 };

            for (int j = 0; j < EvenNumbers.Length; j++)
            {
                Console.WriteLine("Please enter a number");
                EvenNumbers[j] = int.Parse(Console.ReadLine());
            }

            for (int k = 0; k < EvenNumbers.Length; k++)
            {
                Console.WriteLine(EvenNumbers[k]);
            }
        }
```

Act 21: Modify Act-19 to provide more input/output details

```
        static void Main(string[] args)
        {
            int[] EvenNumbers = { 0, 0, 0 };
            Console.WriteLine("Data entry for Array of size three (3) \n");
            for (int j = 0; j < EvenNumbers.Length; j++)
            {
                Console.WriteLine("Please enter a number for element {0}", j + 1);
                EvenNumbers[j] = int.Parse(Console.ReadLine());
            }
            Console.WriteLine("\nDisplay contents of Array\n");
            for (int k = 0; k < EvenNumbers.Length; k++)
            {
                Console.WriteLine("The value in element {0} is {1}", k +1 , EvenNumbers[k]);
            }
        }
```

Act-22: Swap two numbers if first > second using a method call

```
    class Program
    {
        void swap(ref int a, ref int b)
        {
            int temp = a;
            a = b;
            b = temp;
        } //swap

        static void Main(string[] args)
        {
            int first = 0;
            int second = 0;
            Program bigger = new Program();

            Console.WriteLine("Please enter first number:");
            first = int.Parse(Console.ReadLine());
            Console.WriteLine("Please enter second number:");
            second = int.Parse(Console.ReadLine());
            Console.WriteLine("first {0} second {1}", first, second);

            if (first > second)
                bigger.swap(ref first, ref second);
            Console.WriteLine("first {0} second {1}", first, second);
        }//main
    }
```

Act-23: Calculate rectangle using class and methods

```
class Rectangle
    {
        // member variables
        double length;
        double width;
        public void Acceptdetails()
        {
            length = 4.5;
            width = 3.5;
        }
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }

    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.Acceptdetails();
            r.Display();
            Console.ReadLine();
        }
    }
```

Act-24: Use methods to modularise code

```
    class Program
    {

        public int GetSpeed()
        {
            int userinput;
            Console.WriteLine("What is the top speed?");
            userinput = int.Parse(Console.ReadLine());
            return userinput;
        }

        public string GetColor()
        {
            string color = string.Empty;
            Console.WriteLine("What is the color");
            color = Console.ReadLine();
            return color;
        }

        static void Main(string[] args)
        {
            int speed = 0;
            string color = string.Empty;

            Program Vspeed = new Program();
            Program Vcolor = new Program();

            speed = Vspeed.GetSpeed();
            color = Vcolor.GetColor();

            Console.WriteLine("The speed is " + speed + color);
        } // end of Main
    }
```

Act-25: Use nested loops to display all primes from 1..100

```
      static void Main(string[] args)
      {
         /* local variable definition */
         int i, j;

         for (i = 2; i < 100; i++)
         {
            for (j = 2; j <= (i / j); j++)
               if ((i % j) == 0) break; // if factor found, not prime
            if (j > (i / j))
               Console.WriteLine("{0} is prime", i);
         }
      }
```

Act-26: Validate integer input using TRY-CATCH

```
static void Main(string[] args)
{
  int target = 0;
  bool validInt = false;
  //get integer from keyboard
  do
  {
   Console.WriteLine("Please enter in an Integer between 1 and 9");
     try
     {
        target = int.Parse(Console.ReadLine());
        if (target > 0 && target < 10)
           validInt = true;
        else
           validInt = false;
      }//try
      catch
      {
         Console.WriteLine("That was wrong");
      }
   } while (!validInt); //DO Loop

  Console.WriteLine("Yes {0} is an INTEGER", target);
}
```

Act-27: Simple array search using FOR and IF statements

```
        static void Main(string[] args)
        {
            int[] arry = {12, 3, 5, 14, 6, 8, 9, 2};
            //get integer from keyboard
            Console.WriteLine("Please enter in an Ingteger");
            int target = int.Parse(Console.ReadLine());
            //search for target integer
            for(int i = 0; i < arry.Length; i++)
            {
                if (arry[i] == target)
                {
                    Console.WriteLine("found {0} at element {1}", target, i);
                    break; //stop searching and exit FOR loop
                }
                else
                {
                    Console.Write("..");
                }
            }  // end of for Loop
        }
```

Act-28: Sort a table of values using a 2D array

```
        static void Main(string[] args)
        {
            int[,] arr = { { 20, 9, 11 }, { 30, 5, 6 } };
            Console.WriteLine("before");
            for (int i = 0; i < arr.GetLength(0); i++)
            {
                for (int j = 0; j < arr.GetLength(1); j++)
                {
                    Console.Write("{0,3}", arr[i, j]);
                }
                Console.WriteLine();
            }
            Console.WriteLine("After");

            for (int i = 0; i < arr.GetLength(0); i++) // Array Sorting
            {
                for (int j = arr.GetLength(1) - 1; j > 0; j--)
                {

                    for (int k = 0; k < j; k++)
                    {
                        if (arr[i, k] > arr[i, k + 1])
                        {
                            int temp = arr[i, k];
                            arr[i, k] = arr[i, k + 1];
                            arr[i, k + 1] = temp;
                        }
                    }
                }
                Console.WriteLine();
            }

            for (int i = 0; i < arr.GetLength(0); i++)
            {
                for (int j = 0; j < arr.GetLength(1); j++)
                {
                    Console.Write("{0,3}", arr[i, j]);
                }
                Console.WriteLine();
            }
        }
```