NATIONAL OPEN UNIVERSITY OF NIGERIA

COURSE CODE : CIT 834

COURSE TITLE:
OBJECT-ORIENTED PROGRAMMING USING C#

COURSE GUIDE

CIT 834
OBJECT-ORIENTED PROGRAMMING USING C#

Course Developer/Writer	Vivian Nwaocha National Open University of Nigeria Lagos
Programme Leader	Prof Afolabi Adebanjo National Open University of Nigeria Lagos
Course Coordinator	Vivian Nwaocha National Open University of Nigeria, Lagos

NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria
Headquarters
14/16 Ahmadu Bello Way
Victoria Island
Lagos

Abuja Annex
245 Samuel Adesujo Ademulegun Street
Central Business District
Opposite Arewa Suites
Abuja

e-mail: @nou.edu.ng
URL: .nou.edu.ng

First Printed --------

ISBN:

Printed by ……………..
For
National Open University of Nigeria

TABLE OF CONTENTS PAGE

Introduction…………………………………………….. 1
What you will learn in this Course……………………....... 1
Course Aim……………………………………………... 1
Course Objectives………………………………………. 1 – 2
Working through this Course…………………………… 2
Course Materials………………………………………... 2
Study Units……………………………………………… 2 – 3
Recommended Texts……………………………………. 3 – 4
Assignment File………………………………………… 4
Presentation Schedule…………………………………... 4
Assessment……………………………………………… 4
Tutor Marked Assignments (TMAs)………………………. 4 – 5
Final Examination and Grading…………………………… 5
Course Marking Scheme…………………………………… 5
Course Overview……………………………………….. 5 – 6
How to get the most from this course……………………. 6 – 8
Tutors and Tutorials…………………………………….. 8
Summary……….……………………………………… 8

Introduction

CIT 834: Object-Oriented Programming Using C# is a 3 credit course for students studying towards acquiring the Master of Science in Information Technology and related disciplines.

The course is divided into 5 modules and 21 study units. It will introduce the students to concepts of Object-Oriented Programming, .NET framework and C# development. This course also provides information on Simple Class Creation in C#, Methods, C# Constructors, Destructors and Static Behaviour. The last module looks at Polymorphism, Operator Overloading, Indexers and Inheritance.

At the end of this course, it is expected that students should be able to understand, explain and be adequately equipped with basic issues of C# and Object-Oriented Programming in general.

The course guide therefore gives you an overview of what the course: CIT 834 is all about, the textbooks and other course materials to be referenced, what you are expected to know in each unit, and how to work through the course material. It suggests the general strategy to be adopted and also emphasizes the need for self assessment and tutor marked assignment. There are also tutorial classes that are linked to this course and students are advised to attend.

What you will learn in this Course

The overall aim of this course, CIT 834, is to boost the programming expertise of students to enable them develop C# based applications. This course provides extensive hands-on, case study examples, and reference materials designed to enhance your programming skills. In the course of your studies, you will be equipped with definitions of common terms, characteristics and applications of object-oriented programming using C#. You will also learn about .NET framework and C# development. Finally, you will learn about other concepts such as; polymorphism, indexers and inheritance.

Course Aim

This course aims to give students an in-depth understanding of object-oriented programming using C#. It is hoped that the knowledge would enhance the programming expertise of students to enable them develop C# based applications.

Course Objectives

It is pertinent to note that each unit has precise objectives. Students should learn them carefully before proceeding to subsequent units. Therefore, it may be useful to refer to these objectives in the course of your study of the unit to assess your progress. You should always look at the unit objectives after completing a unit. In this way, you can be sure that you have done what is required of you by the end of the unit.

However, below are overall objectives of this course. On successful completion of this course, you should be able to:

- Explain the concept of object-oriented programming (OOP)
- Identify the characteristics of OOP
- Give typical examples of object-oriented programming languages
- Outline setup instructions for C# development
- Describe a C# Console application
- Outline the procedure for creating a windows form
- Identify common variables in C#
- State the general form of a dot operator
- State the class definition syntax
- Outline the procedure for creating a simple C# class
- Identify 2 main methods of adding C# Classes
- Describe how methods are declared
- Give a brief description of constructors
- State the syntax for adding a new constructor to a class
- Give the code for creating a destructor
- Describe the concept of garbage collection
- Outline the steps involved in Calling a static method
- List the properties of Static Constructors
- Explain the concept of Polymorphism
- List 2 techniques of creating overloaded methods
- Identify the procedure for carrying out basic operations on operators
- Give the syntax for declaring one-dimensional and two-dimensional indexers
- Identify the relationship between polymorphism and inheritance

Working through this Course

To complete this course, you are required to study all the units, the recommended text books, and other relevant materials. Each unit contains some self assessment exercises and tutor marked assignments, and at some point in this course, you are required to submit the tutor marked assignments. There is also a final examination at the end of this course. Stated below are the components of this course and what you have to do.

Course Materials

The major components of the course are:

1. Course Guide
2. Study Units
3. Text Books
4. Assignment File
5. Presentation Schedule

Study Units

There are 21 study units and 5 modules in this course. They are:

MODULE 1: FUNDAMENTALS OF OBJECT-ORIENTED PROGRAMMING AND C#.............……… 1

Unit 1 Object-Oriented Programming………….. 1 – 6
Unit 2 .NET Framework and C# Development…… 7 – 12
Unit 3 Getting Started with C#……………………...... 13 – 20
Unit 4 Common Variables in C# ..………………….. 21 – 25

MODULE 2: SIMPLE CLASS CREATION IN C#…………… 26

Unit 1 Class……………………….......................... 26 – 31
Unit 2 Operation on Class……………………….. 32 – 38
Unit 3 Adding Methods …………………………… 39 – 43
Unit 4 C# Class Properties ……………………… 39 – 43

MODULE 3: C# CONSTRUCTORS AND DESTRUCTORS…. 44

Unit 1 Constructors ……………………………... 44 – 50
Unit 2 The Default Constructor……………………. 51 – 57
Unit 3 Destructors ………………………………. 58 – 62
Unit 4 Garbage Collection………………………… 63 – 69

MODULE 4: C# STATIC BEHAVIOUR……………………… 68

Unit 1 Static Behaviours……………..…………… 68 – 71
Unit 2 Creating a Static Method…………………… 72 – 78
Unit 3 Static Properties…………………………… 79 – 82
Unit 4 Static Construction………………………… 83 – 87

MODULE 5: MAIN FEATURES……………………..……… 88

Unit 1 Polymorphism……….…………………….. 88 – 93
Unit 2 Overloaded Method…..…………………... 94 – 100
Unit 3 Operator Overloading…………………… 101 – 105
Unit 4 Indexers……………….………………….... 106 – 110
Unit 5 Events and Inheritance……………………... 106 – 110

Recommended Texts

These texts will be of enormous benefit to you in learning this course:

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278

11. Joyce, F. (2006). Microsoft Visual C#.NET with Visual Studio 2005
12. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
13. Martin, A and Luca, C. (2005). *A Theory of Objects*.
14. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
15. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
16. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
17. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
18. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
19. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
20. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

Assignment File

The assignment file will be given to you in due course. In this file, you will find all the details of the work you must submit to your tutor for marking. The marks you obtain for these assignments will count towards the final mark for the course. Altogether, there are 21 tutor marked assignments for this course.

Presentation Schedule

The presentation schedule included in this course guide provides you with important dates for completion of each tutor marked assignment. You should therefore endeavour to meet the deadlines.

Assessment

There are two aspects to the assessment of this course. First, there are tutor marked assignments; and second, the written examination.

Therefore, you are expected to take note of the facts, information and problem solving gathered during the course. The tutor marked assignments must be submitted to your tutor for formal assessment, in accordance to the deadline given. The work submitted will count for 40% of your total course mark.

At the end of the course, you will need to sit for a final written examination. This examination will account for 60% of your total score.

Tutor Marked Assignments (TMAs)

There are 21 TMAs in this course. You need to submit all the TMAs. The best 4 will therefore be counted. When you have completed each assignment, send them to your tutor as soon as possible and make certain that it gets to your tutor on or before the stipulated deadline. If for any reason you cannot complete your assignment on time, contact your tutor before the assignment is due to discuss the possibility of extension. Extension will not be granted after the deadline, unless on extraordinary cases.

Final Examination and Grading

The final examination for CIT 834 will be of last for a period of 3 hours and have a value of 60% of the total course grade. The examination will consist of questions which reflect the self assessment exercise and tutor marked assignments that you have previously encountered. Furthermore, all areas of the course will be examined. It would be better to use the time between finishing the last unit and sitting for the examination, to revise the entire course. You might find it useful to review your TMAs and comment on them before the examination. The final examination covers information from all parts of the course.

Course marking Scheme

The following table includes the course marking scheme

Table 1 Course Marking Scheme

Assessment	**Marks**
Assignments 1-21	21 assignments, 40% for the best 4 Total = 10% X 4 = 40%
Final Examination	60% of overall course marks
Total	100% of Course Marks

Course Overview

This table indicates the units, the number of weeks required to complete them and the assignments.

Table 2: Course Organizer

Unit	**Title of Work**	**Weeks Activity**	**Assessment (End of Unit)**
	Course Guide	Week 1	
Module 1	**Object-Oriented Programming and C#**		
Unit 1	Object-Oriented Programming	Week 1	Assignment 1
Unit 2	.NET Framework and C# Development	Week 2	Assignment 2
Unit 3	Getting Started with C#	Week 3	Assignment 3
Unit 4	Common Variables in C#.NET	Week 3	Assignment 4
Module 2	**Simple Class Creation in C#**		
Unit 1	Preamble to C# Class	Week 4	Assignment 5
Unit 2	Operations on a Class	Week 4	Assignment 6
Unit 3	C# Methods	Week 5	Assignment 7
Unit 4	C# Class Properties	Week 5	Assignment 8
Module 3	**C# Constructors and Destructors**		
Unit 1	Constructors	Week 6	Assignment 9
Unit 2	The Default Constructor	Week 6	Assignment 10
Unit 3	Destructors	Week 7	Assignment 11
Unit 4	Garbage Collection	Week 7	Assignment 12
Module 4	**C# Static Behaviour**		
Unit 1	Static Behaviours	Week 8	Assignment 13
Unit 2	Creating a Static Behaviour	Week 9	Assignment 14
Unit 3	Static Properties	Week 10	Assignment 15
Unit 4	Static Constructors		Assignment 16
Module 5	**Polymorphism**		
Unit 1	Introduction to Polymorphism	Week 11	Assignment 17
Unit 2	Overloaded Method	Week 12	Assignment 18
Unit 3	C# Operator Overloading	Week 13	Assignment 19
Unit 4	C# Indexers	Week 14	Assignment 20
Unit 5	C# Inheritance and Polymorphism	Week 14	Assignment 21

How to get the most out of this course

In distance learning, the study units replace the university lecturer. This is one of the huge advantages of distance learning mode; you can read and work through specially designed study materials at your own pace and at a time and place that is most convenient. Think of it as reading from the teacher, the study guide indicates what you ought to study, how to study it and the relevant texts to consult. You are provided with exercises at appropriate points, just as a lecturer might give you an in-class exercise.

Each of the study units follows a common format. The first item is an introduction to the subject matter of the unit and how a particular unit is integrated with the other units and the course as a whole. Next to this is a set of learning objectives. These learning objectives are meant to guide your studies. The moment a unit is finished, you must go back and check whether you have achieved the objectives. If this is made a habit, then you will increase your chances of passing the course. The main body of the units also guides you through the required readings from other sources. This will usually be either from a set book or from other sources.

Self assessment exercises are provided throughout the unit, to aid personal studies and answers are provided at the end of the unit. Working through these self tests will help you to achieve the objectives of the unit and also prepare you for tutor marked assignments and examinations. You should attempt each self test as you encounter them in the units.

The following are practical strategies for working through this course

1. Read the course guide thoroughly
2. Organise a study schedule. Refer to the course overview for more details. Note the time you are expected to spend on each unit and how the assignment relates to the units. Important details, e.g. details of your tutorials and the date of the first day of the semester are available. You need to gather together all these information in one place such as a diary, a wall chart calendar or an organizer. Whatever method you choose, you should decide on and write in your own dates for working on each unit.
3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that

they get behind with their course works. If you get into difficulties with your schedule, please let your tutor know before it is too late for help.

4. Turn to Unit 1 and read the introduction and the objectives for the unit.
5. Assemble the study materials. Information about what you need for a unit is given in the table of content at the beginning of each unit. You will almost always need both the study unit you are working on and one of the materials recommended for further readings, on your desk at the same time.
6. Work through the unit, the content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit, you will be encouraged to read from your set books.
7. Keep in mind that you will learn a lot by doing all your assignments carefully. They have been designed to help you meet the objectives of the course and will help you pass the examination.
8. Review the objectives of each study unit to confirm that you have achieved them. If you are not certain about any of the objectives, review the study material and consult your tutor.
9. When you are confident that you have achieved a unit's objectives, you can start on the next unit. Proceed unit by unit through the course and try to pace your study so that you can keep yourself on schedule.
10. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor marked assignment form and also written on the assignment. Consult you tutor as soon as possible if you have any questions or problems.
11. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in this course guide).

Tutors and Tutorials

There are 8 hours of tutorial provided in support of this course. You will be notified of the dates, time and location together with the name and phone number of your tutor as soon as you are allocated a tutorial group.

Your tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties you might encounter and provide assistance to you during the course. You must mail your tutor marked assignment to your tutor well before the due date. At least two working days are required for this purpose. They will be marked by your tutor and returned to you as soon as possible.

Do not hesitate to contact your tutor by telephone, e-mail or discussion board if you need help. The following might be circumstances in which you would find help necessary: contact your tutor if:

- You do not understand any part of the study units or the assigned readings.
- You have difficulty with the self test or exercise.
- You have questions or problems with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

You should try your best to attend the tutorials. This is the only chance to have face-to- face contact with your tutor and ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from the course tutorials, prepare a question list before attending them. You will learn a lot from participating in discussion actively. GOODLUCK!

Course Code	CIT 834
Course Title	Object-Oriented Programming Using C#
Course Developer/Writer	Vivian Nwaocha National Open University of Nigeria Lagos
Programme Leader	Prof Afolabi Adebanjo National Open University of Nigeria Lagos
Course Coordinator	Vivian Nwaocha National Open University of Nigeria Lagos

NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria
Headquarters
14/16 Ahmadu Bello Way
Victoria Island
Lagos

Abuja Annex
245 Samuel Adesujo Ademulegun Street
Central Business District
Opposite Arewa Suites
Abuja

e-mail: @nou.edu.ng
URL: .nou.edu.ng

National Open University of Nigeria 2009

First Printed 2009

ISBN:

All Rights Reserved

Printed by ……………..
For
National Open University of Nigeria

TABLE OF CONTENTS PAGE

Module 1: Object-Oriented Programming and C#.............. 1

Unit 1 Object-Oriented Programming………….. 1 – 6
Unit 2 .NET Framework and C# Development…… 7 – 12
Unit 3 Getting Started with C#…………………...... 13 – 20
Unit 4 Common Variables in C# ..………………….. 21 – 25

Module 2: Simple Class Creation in C#……………… 26

Unit 1 Preamble to C# Class……………………….... 26 – 31
Unit 2 Operations on a Class……………………….. 32 – 38
Unit 3 C# Methods …………………………… 39 – 43
Unit 4 C# Class Properties ……………………… 39 – 43

Module 3: C# Constructors and Destructors……….. 44

Unit 1 Constructors ……………………………... 44 – 50
Unit 2 The Default Constructor……………………. 51 – 57
Unit 3 Destructors ………………………………. 58 – 62
Unit 4 Garbage Collection………………………… 63 – 69

Module 4: C# Static Behaviour……………………… 68

Unit 1 Static Behaviours…………….…………. 68 – 71
Unit 2 Creating a Static Method…………………… 72 – 78
Unit 3 Static Properties…………………………… 79 – 82
Unit 4 Static Constructors………………………… 83 – 87

Module 5: Polymorphism…………………….……… 88

Unit 1 Introduction to Polymorphism…….…………88 – 93
Unit 2 Overloaded Method…..…………………... 94 – 100
Unit 3 C# Operator Overloading……………… 101 – 105
Unit 4 C# Indexers…………….………………… 106 – 110
Unit 5 C# Inheritance and Polymorphism………... 106 – 110

MODULE 1 OBJECT-ORIENTED PROGRAMMING AND C#

Unit 1	Object-Oriented Programming (OOP)……	1 – 6
Unit 2	.NET Framework and C# Development……	7 – 12
Unit 3	Getting Started with C# ………………….......	13 – 20
Unit 4	Common Variables in C#.NET……………….	21 – 25

UNIT 1 OBJECT-ORIENTED PROGRAMMING (OOP)

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
  3.1 Overview
    3.1.1 Object-Oriented Programming Pattern
    3.1.2 Characteristics
    3.1.3 Benefits
    3.1.4 Applications
  3.2 Fundamental Concepts
    3.2.1 Objects
    3.2.2 Classes
    3.2.3 Data Abstraction and Encapsulation
    3.2.4 Inheritance
    3.2.5 Polymorphism
    3.2.6 Message Passing
  3.3 OOP Languages
    3.3.1 Simula
    3.3.2 Smalltalk
    3.3.3 C++
    3.3.4 Java
    3.3.5 .NET
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

The most important thing you would need to learn from this module is the idea that programming in an object-oriented concept or language is much more than just learning new functions, syntax, etc. To this end, object-oriented programming (OOP) is more than learning a new language; it requires *a new way of thinking.* We must no longer think only in terms of data structures - we must think also in terms of *objects*.

This module has thus been designed to boost your programming expertise, assuming the learner has at least completed any programming language. However, some parts of the topic don't even require basic programming idea. Even if you have gained previous programming experience with any conventional programming language, it is recommended that you go through the entire module systematically to gain some insight of the course.

2.0 OBJECTIVES

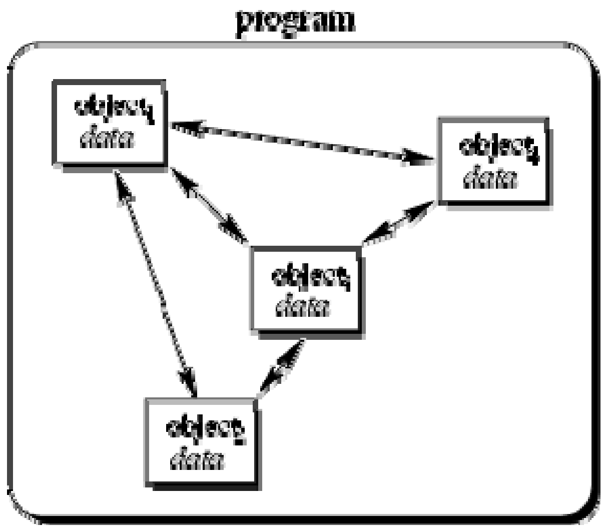
At the end of this unit, you should be able to:

- Define the term object-oriented programming (OOP)
- Identify the characteristics of OOP
- State the benefits of OOP
- Outline the applications of OOP
- Explain the basic concepts of OOP
- Identify what really makes a programming language object-oriented
- Give typical examples of object-oriented programming languages

3.0 MAIN CONTENT

3.1 Overview

Object-orientation or object oriented programming (OOP) was first developed in the 1960s, as a programming concept to help Software Developers build high quality software. Object-orientation is also a concept which makes developing of projects easier. Consequently, object-oriented programming attempts to solve the problems with only one approach; dividing the problems in sub-modules and using different objects. Objects of the program interact by sending messages to each other. The drawing below illustrates this clearly -



To understand the actual concept of object orientation and the OOP, we should first be acquainted with the basic concepts of OOP and ought to know the exclusive terms used in this book later.

Object-oriented programming (OOP), reorganizes the programming problem to allow for a higher level of abstraction. Programming with objects is quite like working with real-world objects. It groups operations and data into modular units called objects. These objects can be combined into structured networks to form a complete program, similar to how the pieces in a puzzle fit together to create a picture.

By breaking down complex software projects into small, self-contained, and modular units, **object** orientation ensures that changes to one part of a software project will not adversely affect other portions of the software. **Object** orientation also aids software reuse. Once functionality is created in one program, it can easily be reused in other programs.

Definition: "Object-oriented programming is a programming approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand."

Thus, an object is considered to be a partitioned area of the computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

3.1.1 Object-Oriented Programming Pattern

The inspiring factor in the invention of object-oriented approach is that some of the flaws encountered in the procedural approach can be eradicated. OOP treats data as a critical element in the program

development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions. OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects.
Figure 1.0 shows the organization of data and functions in object-oriented programs. The data of an object can be accessed only by the functions associated with that object. However, functions of one object can access the functions of other objects.

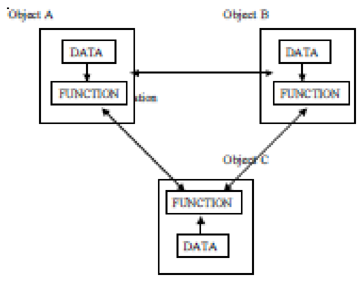


Figure 1.0 Organisation of Data and Functions in OOP

3.1.2 Characteristics of Object-Oriented Programming (OOP)

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.

3.1.3 Benefits of Object-Oriented Programming (OOP)

Object-oriented programming offers several benefits to both the program designer and the user. Object-orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and cheaper cost of maintenance.

The principal advantages are:

- We can eliminate redundant code and extend the use of existing classes through inheritance.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is possible to have multiple instances of an object to coexist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a, project based on objects.
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object-oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer.

There are a number of issues that need to be tackled to reap some of the benefits stated above. For instance, object libraries must be available for reuse. This technology is still developing and, current products may be superseded quickly. Strict controls and protocols need to be developed if reuse is not to be compromised.

However, it is hoped that the object-oriented programming tools would facilitate software development, which otherwise would be quite difficult.

3.1.4 Applications of Object-oriented Programming

Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to

now, has been in the area of user interface design such as windows. Hundreds of windowing systems have been developed, using the OOP techniques.
Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. Object-oriented programming is useful in these types of applications because it can simplify a complex problem. The promising areas for application of object-oriented programming include:

- Real-time systems
- Simulation and modeling
- Object-oriented databases
- Hypertext, hypermedia and expert text
- AI and expert systems
- Neural networks and Parallel programming Decision support and office automation systems
- CIM/CAM/CAD systems

Object-oriented technology is certainly going to change the way software engineers think, analyze, design and implement future systems.

3.2 Fundamental Concepts of Object-Oriented Programming

It is necessary to understand some of the concepts used extensively in object-oriented programming. The fundamental concepts of OOP are as follows:

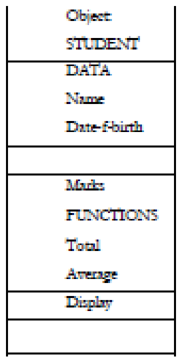
- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

3.2.1 Objects

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists.
Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in C.
When a program is executed, the objects interact by sending messages to one another. For example, if “customer” and “account” are two objects

in a program, then the customer object may send a message to the account object requesting for the bank balance. Each object contains data, and code to manipulate the data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted, and the type of response returned by the objects. However, different authors represent them differently.

Figure 1.1 shows two notations that are popularly used in object-oriented analysis and design.



Two ways of representing an Object

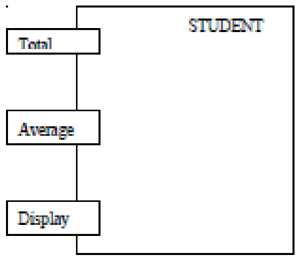


Figure 1.1 Notations used in Object-oriented Analysis and Design

3.2.2 Classes

Objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created.

Definition
A class is a collection of objects of similar type.
For e.g., mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language. The syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement

Fruit mango;
will create an object mango belonging to the class fruit.

3.2.3 Data Abstraction and Encapsulation

Encapsulation, also known as *data hiding*, is an important object-oriented programming concept. It is the act of concealing the functionality of a class so that the internal operations are hidden from the programmer. With correct encapsulation, the developer does not need to understand how the class actually operates in order to communicate with it via its publicly available methods and properties; known as its public *interface*.

Encapsulation is essential for creating maintainable object-oriented programs. When the interaction with an object uses only the publicly available *interface* of methods and properties, the class of the object becomes a correctly isolated unit. The wrapping up of data and functions into a single unit (called class) is known as ***encapsulation***.
Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world, and only those functions, which are wrapped in the class, can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding or information hiding.

Abstraction refers to the act of representing essential features without including the background details or explanations.
Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called data members because they hold information. The functions that operate on these data are sometimes called methods or member functions. Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT)

3.2.4 Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class.
For example, the bird 'robin' is a part of the class "flying bird' which is again a part of the class 'bird'. The principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived as illustrated in Figure 1.2.

The concept of inheritance provides the idea of *reusability*. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side-effects into the rest of the classes.

Note that each sub-class defines only those features that are unique to it. Without the use of classification, each class would have to explicitly include all of its features.

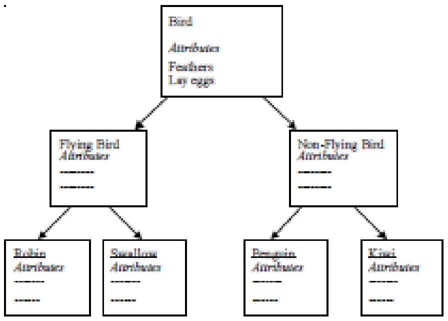


Figure 1.2 Property Inheritance

3.2.5 Polymorphism

Polymorphism refers to the ability to take more than one form.
An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.

For example, consider the operation of addition of two numbers; the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.

Figure 1.3 illustrates that a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having several different meanings depending on the context. Using a single function name to perform different types of tasks is known as **function overloading**.

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ.

Polymorphism is extensively used in implementing inheritance.

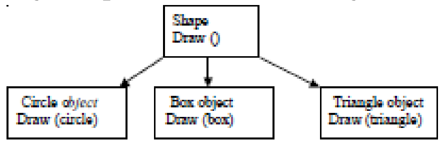


Figure 1.3 Polymorphism

3.2.6 Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

1. Creating classes that define objects and their behavior,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts. A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result.

Thus, **message passing** involves specifying the name of the object, the name of the function (message) and the information to be sent.
Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is a'1ive.

SELF ASSESSMENT EXERCISE

Outline the characteristics of object-oriented programming.

____
____
____
____

3.3 Object-Oriented Programming Language

What really makes a programming language object oriented? A programming language is said to be object-oriented when it allows, to some degree, object-oriented programming techniques, such as encapsulation, inheritance, modularity and polymorphism.

3.3.1 Simula

As the name implies, Simula was created to aid in simulations. It is not a coincidence that simulations typically model real-world systems. Many of these real-world systems contained hundreds, or thousands, of interacting parts.

The initial version of the language, Simula-1, was introduced in 1966. The programming modules defined by Simula were based not on procedures, but on actual physical objects. Simula had a novel way of presenting the object, so that each object has its own behavior and data.

3.3.2 Smalltalk

Many consider that the first truly object-oriented (O-O) language was Smalltalk, developed at the Learning Research Group at Xerox's Palo Alto Research Center in the early 1970s. In Smalltalk, everything is really an object that enforces the O-O paradigm. It is virtually impossible to write a program in Smalltalk that is not O-O. This is not the case for other languages that support objects, such as C++ and Visual Basic (and Java, for that matter).

3.3.3 C++

C++ has its roots in a project to simulate software running on a distributed system. This simulator, actually written in Simula, is where Bjarne Stroustrup conceived of the idea of combining some of the features of Simula with the syntax of C.

While working at Bell, Stroustrup made personal contacts with people such as Brian Kernighan and Dennis Ritchie, who wrote the definitive book on C. When the initial simulator written in Simula failed, Stroustrup decided to rewrite it in a C predecessor called BCPL.

C++ was originally implemented in 1982 under the name C with Classes. As the name suggests, the most important concept of C with Classes was the addition of the class. The class concept provided the encapsulation now requisite with O-O languages.

3.3.4 Java

Java's origins are in consumer electronics. In 1991, Sun Microsystems began to investigate how it might exploit this growing market. Some time later, James Gosling was investigating the possibility of creating a hardware-independent software platform for just this purpose. Initially, he attempted to use C++, but soon abandoned C++ and began the creation of a new language he dubbed Oak.

By fall 1995, Java beta 1 was released, and the Netscape Navigator 2.0 browser incorporated Java. Java 1.0 was officially released in January 1996. Over the past several years, Java has progressed to the current release Java 2 Platform Standard Edition 6.0 as well as other platforms such as an Enterprise Edition (J2EE) for the enterprise/server market and a Micro Edition (J2ME) for mobile and wireless.

3.3.5 .NET

Microsoft initially responded to the popularity of Java by producing a version of Java called Visual J++. However, Microsoft decided on a more comprehensive response. Using many of the groundbreaking concepts that Java implemented, Microsoft developed a language called C# that became the foundation for the company's .NET platform. As with Java, C# relied heavily on the success and failures of earlier languages.

The .NET development environment includes many of the really good characteristics of several other platforms. .NET incorporates many of concepts introduced by the initial Java release. The .NET platform also builds upon many of the powerful features of the VB6 and Visual C++ environments.

Visual Basic 6 was one of the most popular programming languages. The programming environment for VB6 has had a huge impact on state-of-the-art development environments. VB6 has evolved steadily towards the object-oriented model until it finally joined the list of object-oriented languages with the release of Visual Basic .NET. VB6 was not totally object-oriented; it did not implement inheritance completely.

4.0 CONCLUSION

In this unit, we defined some basic concepts of object-oriented programming. We also looked at the areas of application and benefits of object-oriented programming.

5.0 SUMMARY

We hope you enjoyed this unit. This unit provided an overview of object oriented programming: basic definition, characteristics, applications benefits and key concepts. Now, let us attempt the questions below.

6.0 TUTOR MARKED ASSIGNMENT

Give a brief explanation of the following; Data encapsulation, Abstraction, Function Overloading and Message Passing.

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
12. Martin, A and Luca, C. (2005). *A Theory of Objects*.
13. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
14. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
15. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.

UNIT 2 .NET FRAMEWORK AND C# DEVELOPMENT

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
3.1 Introducing the .NET Platform
3.1.1 What is .NET?
3.1.2 The Common Language Runtime (CLR)
3.1.3 The .NET Framework Class Library (FCL)
3.1.4 C# and other .NET Languages
3.1.5 The Common Language Specification (CLS)
3.2 Microsoft C# Development
3.2.1 Requirements
3.2.2 Set up Instructions C# Development
3.2.3 Running the Microsoft.NET(C#)
3.2.4 Troubleshooting Setup Issues
3.3 Hello World
3.3.1 Saving the Hello World File
3.2.2 Initial Hello World Program
3.2.3 Parts of the Hello World Program
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

In unit 1, we gave an overview of ‘object-oriented programming’ as well as its basic concepts and applications. This unit provides information about setting up your development environment for developers using Microsoft C# programming language. It includes requirements, Setup instructions for C# development, Hello world history as well as well as Console vs. Windows vs. Web Application.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Identify and explain characteristics of communication
- Illustrate functions of communication
- Explain purposes of communication

3.0 MAIN CONTENT

3.1 Introducing the .NET Platform

As a C# developer, it is essential that you understand the environment you are building applications on: **Microsoft .NET** (pronounced "Dot Net"). After all, your design and development decisions will often be influenced by code-compilation practicalities, the results of compilation, and the behavior of applications in the runtime environment. The foundation of all .NET development begins here, and so it is important that you take special note of this unit in order to understand concepts that affect the practical implementation of C#.

By learning about the .NET environment, you can gain an understanding of what .NET is and what it means to you. You will equally gain knowledge of the parts of .NET, including the Common Language Runtime (CLR), the .NET Framework Class Library, and how .NET supports multiple languages. Along the way, you will see how the parts of .NET tie together, their relationships, and what they do for you. First, however, you need to know what .NET is, which is explained in the next section.

3.1.1 What is .NET?

Microsoft .NET, which is commonly referred to as just .NET, is a platform for developing "managed" software. The word managed is key here- a concept setting the .NET platform apart from many other development environments.

When referring to other development environments, we would be focusing on the traditional practice of compiling to an executable file that contains machine code and how that file is loaded and executed by the operating system. **Figure 1.1** shows what I mean about the traditional compilation-to-execution process.

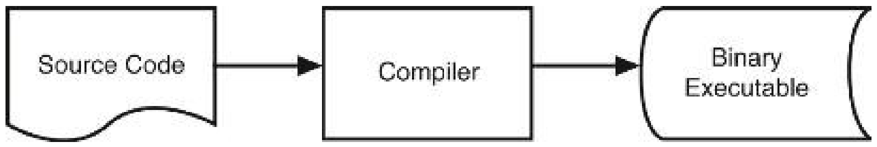


Figure 1.1 Traditional compilation

In the traditional compilation process, the executable file is binary and can be executed by the operating system immediately. However, in the managed environment of .NET, the file produced by the compiler (the C# compiler in our case) is not an executable binary. Instead, it is an assembly, shown in **Figure 1.2**, which contains metadata and intermediate language code.

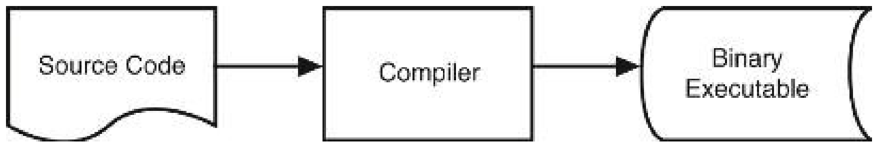


Figure 1.2 Managed compilation.

.NET STANDARDIZATION

As mentioned in the preceding paragraph, an assembly contains intermediate language and metadata rather than binary code. This intermediate language is called Microsoft Intermediate Language (MSIL), which is commonly referred to as IL. IL is a high-level, component-based assembly language. In later sections of this chapter, you learn how IL supports a common type system and multiple languages in the same platform.

.NET has been standardized by both the European Computer Manufacturers Association (ECMA) and the Open Standards Institute (OSI). The standard is referred to as the Common Language Infrastructure (CLI). Similarly, the standardized term for IL is Common Intermediate Language (CIL).

In addition to .NET, there are other implementations of CIL-the two most well known by Microsoft and Novell. Microsoft's implementation is an open source offering for the purposes of research and education called the Shared Source Common Language Infrastructure (SSCLI). The Novell offering is called Mono, which is also open source.

Beyond occasional mention, this book focuses mainly on the Microsoft .NET implementation of the CLI standard.

The other part of an assembly is metadata, which is extra information about the code being used in the assembly. **Figure 1.3** shows the contents of an assembly.

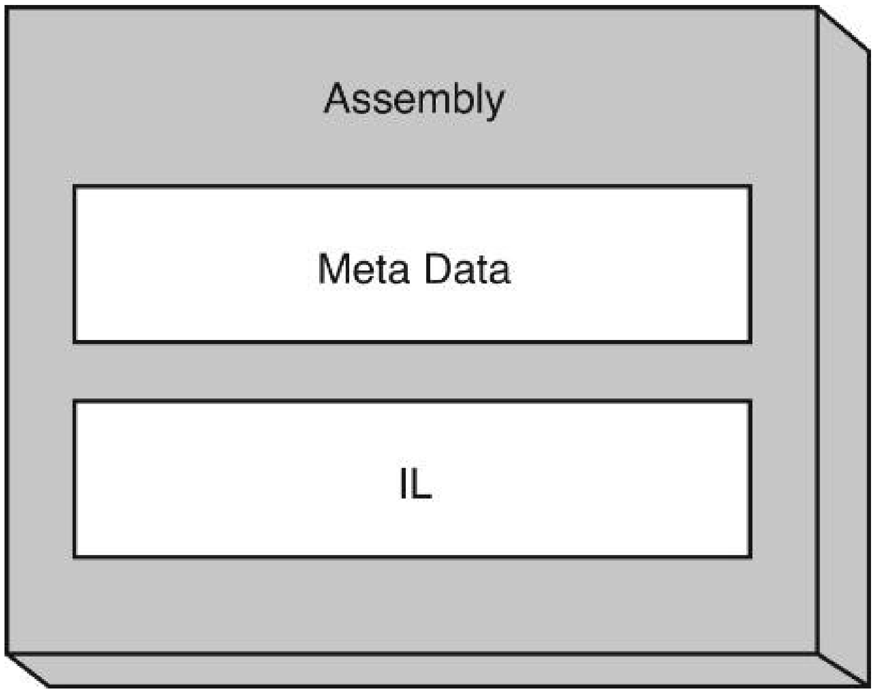


Figure 1.3 Assembly contents.

Figure 1.3 is a simplified version of an assembly, showing only those parts pertaining to the current discussion. Assemblies have other features that illustrate the difference between an assembly and an executable file. Specifically, the role of an assembly is to be a unit of deployment, execution, identity, and security in the managed environment. In Part X, Chapters 43 and 44 explain more about the role of the assembly in deployment, identity, and security. The fact that an assembly contains metadata and IL, instead of only binary code, has a significant advantage, allowing execution in a managed environment.
The next section explains how the CLR uses the features of an assembly to manage code during execution.

3.1.2 The Common Language Runtime (CLR)

As introduced in the preceding unit, C# applications are compiled to IL, which is executed by the CLR. This section highlights several features of the CLR. You'll also see how the CLR manages your application during execution.

Why Is the CLR Important?

In many traditional execution environments of the past, programmers needed to perform a lot of the low-level work (plumbing) that applications needed to support. For example, you had to build custom security systems, implement error handling, and manage memory.

The degree to which these services were supported on different language platforms varied considerably. Visual Basic (VB) programmers had built-in memory management and an error-handling system, but they didn't always have easy access to all the features of COM+, which opened up more sophisticated security and transaction processing. C++ programmers have full access to COM+ and exception handling, but memory management is a totally manual process. In a later section, you learn about how .NET supports multiple languages, but knowing just a little about a couple of popular languages and a couple of the many challenges they must overcome can help you to understand why the CLR is such a benefit for a C# developer.

The CLR solves many problems of the past by offering a feature-rich set of plumbing services that all languages can use. The features described in the next section further highlight the value of the CLR.

CLR Features

This section describes, more specifically, what the CLR does for you. **Table 1.1** summarizes CLR features with the corresponding descriptions.

In addition to the descriptions provided in Table 1.1, the following sections expand upon a few of the CLR features. These features are included in the CLR execution process.

The CLR Execution Process

Beyond just executing code, parts of the execution process directly affect your application design and how a program behaves at runtime. Many of these subjects are handled throughout this book, but this section highlights specific additional items you should know about.

From the time you or another process selects a .NET application for execution, the CLR executes a special process to run your application, shown in **Figure 1.4**.

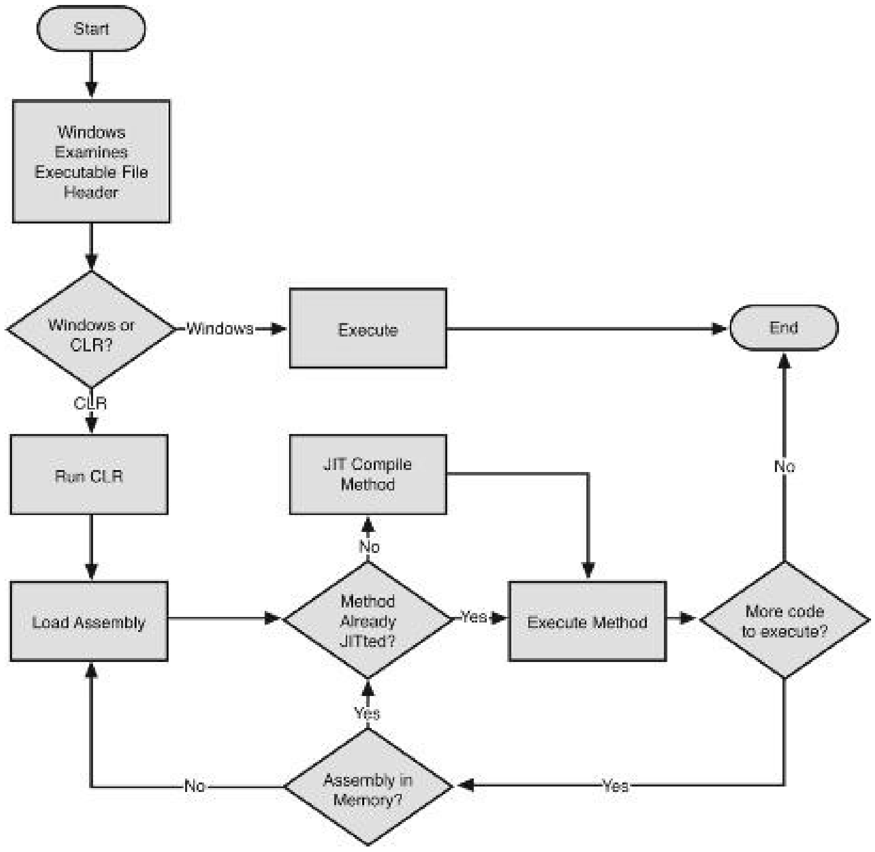


Figure 1.4 The CLR execution process (summarized).

As illustrated in **Figure 1.4**, Windows (the OS) will be running at Start; the CLR won't begin execution until Windows starts it. When an application executes, OS inspects the file to see whether it has a special header to indicate that it is a .NET application. If not, Windows continues to run the application.

If an application is for .NET, Windows starts up the CLR and passes the application to the CLR for execution. The CLR loads the executable assembly, finds the entry point, and begins its execution process.

The executable assembly could reference other assemblies, such as dynamic link libraries (DLLs), so the CLR will load those. However, this is on an as-needed basis. An assembly won't be loaded until the CLR needs access to the assembly's code. It's possible that the code in some assemblies won't be executed, so there isn't a need to use resources unless absolutely necessary.

As mentioned previously, the C# compiler produces IL as part of an assembly's output. To execute the code, the CLR must translate the IL to binary code that the operating system understands. This is the responsibility of the JIT compiler.

As its name implies, the JIT compiler only compiles code before the first time that it executes. After the IL is compiled to machine code by the JIT compiler, the CLR holds the compiled code in a working set. The next time that the code must execute, the CLR checks its working set and runs the code directly if it is already compiled. It is possible that the working set could be paged out of memory during program execution, for various reasons that are necessary for efficient operation of the CLR on the particular machine it is running on. If more memory is available than the size of the working set, the CLR can hold on to the code. Additionally, in the case of Web applications where scalability is an issue, the working set can be swapped out due to periodic recycling or heavier load on the server, resulting in additional load time for subsequent requests.

The JIT compiler operates at the method level. If you aren't familiar with the term method, it is essentially the same as a function or procedure in other languages. Therefore, when the CLR begins execution, the JIT compiler compiles the entry point (the Main method in C#). Each subsequent method is JIT compiled just before execution. If a method being JIT compiled contains calls to methods in another assembly, the CLR loads that assembly (if not already loaded).

This process of checking the working set, JIT compilation, assembly loading, and execution continues until the program ends. Some other detail you might be concerned with is application performance. As described earlier, code is loaded and JIT compiled. Another DLL adds load time, which may or may not make a difference to you, but it is certainly something to be aware of. By the way, after code has been JIT compiled, it executes as fast as any other binary code in memory.

One of the CLR features listed in Table 1.1 is .NET Framework Class Library (FCL) support. The next section goes beyond FCL support for the CLR and gives an overview of what else the FCL includes.

3.1.3 The .NET Framework Class Library (FCL)

.NET has an extensive library, offering literally thousands of reusable types. Organized into namespaces, the FCL contains code supporting all the .NET technologies, such as Windows Forms, Windows Presentation Foundation, ASP.NET, ADO.NET, Windows Workflow, and Windows Communication Foundation. In addition, the FCL has numerous crosslanguage technologies, including file I/O, networking, text management, and diagnostics. As mentioned earlier, the FCL has CLR support in the areas of built-in types, exception handling, security, and threading. **Table 1.2** shows some common FCL libraries.

What Is a Type?

Types are used to define the meaning of variables in your code. They could be **built-in types** such as int, double, or string. You can also have **custom types** such as Customer, Employee, or Bank Account. Each type has optional data/behavior associated with it.

The namespaces in **Table 1.2** are a sampling from the many available in the .NET Framework. They're representative of the types they contain. For example, you can find Windows Presentation Foundation (WPF) libraries in the System.Windows namespace, Windows Communication Foundation (WCF) is in the System.ServiceModel namespace, and Language Integrated Query (LINQ) types can be found in the System.Linq namespace.

Another aspect of **Table 1.2** is that only two levels are included in the namespace hierarchy, System.*. In fact, there are multiple namespace levels, depending on which technology you view. For example, if you want to write code using the Windows Workflow (WF) runtime, you look in the System.Workflow.Runtime namespace. Generally, you can find the more common types at the higher namespace levels.

One of the benefits you should remember about the FCL is the amount of code reuse it offers. As you read through this book, you'll see many examples of how the FCL forms the basis for code you can write. The FCL was built and intended for reuse, and you can often be much more productive by using FCL types rather than building your own from scratch.

Another important feature of the FCL is language neutrality. Just like the CLR, it doesn't matter which .NET language you program in-the FCL is reusable by all .NET programming languages, which are discussed in the next section.

3.1.4 C# and Other .NET Languages

.NET supports multiple programming languages, which are assisted by both the CLR and the FCL. Literally dozens of languages target the .NET CLR as a platform. **Table 1.3** lists some of these languages.

Table 1.3 is not a comprehensive list because there are new languages being created for .NET on a regular basis. An assumption one could make from this growing list is that .NET is a successful multi-language platform.

The C# compiler emits IL. However, the C# compiler is not alone-all compilers for languages in **Table 1.2** emit IL, too. By having a CLR that consumes IL, anyone can build a compiler that emits IL and join the .NET family of languages.

In the next section, you learn how the CLR supports multiple languages via a Common Type System (CTS), the relationship of the languages via a Common Language Specification (CLS), and how languages are supported via the FCL.

3.1.5 The Common Language Specification (CLS)

Although the CLR understands all types in the CTS, each language targeting the CLR will not implement all types. Languages must often be true to their origins and will not lose their features or add new features that aren't compatible with how they are used.

However, one of the benefits of having a CLR with a CTS that understands IL, and an FCL that supports all languages, is the ability to write code in one language that is consumable by other languages. Imagine you are a third-party component vendor and your language of choice is C#. It would be desirable that programmers in any .NET language (for example, IronRuby or Delphi) would be able to purchase and use your components.

For programming languages to communicate effectively, targeting IL is not enough. There must be a common set of standards to which every .NET language must adhere. This common set of language features is called the Common Language Specification (CLS).

Most .NET compilers can produce both CLS-compliant and non-CLS-compliant code, and it is up to the developer to choose which language features to use. For example, C# supports unsigned types, which are non-CLS compliant. For CLS compliance, you can still use unsigned types within your code so long as you don't expose them in the public interface of your code, where code written in other languages can see.

Table 1.1 CLR Features with the Corresponding Descriptions

Feature	***Description***
.NET Framework Class Library support	Contains built-in types and libraries to manag assemblies, memory, security, threading, an other runtime system support
Debugging	Facilities for making it easier to debug code.
Exception management	Allows you to write code to create and handle exceptions.
Execution management	Manages the execution of code
Garbage collection	Automatic memory management and garbage collection (Chapter 15)
Interop	Backward-compatibility with COM and Win32 code.
Just-In-Time (JIT) compilation	An efficiency feature for ensuring that the CLR only compiles code just before it executes
Security	Traditional role-based security support, in addition to Code Access Security (CAS)
Thread management	Allows you to run multiple threads of execution
Type loading	Finds and loads assemblies and types
Type safety	Ensures references match compatible types, which is very useful for reliable and secure code

Table 1.2 Common FCL Libraries

Feature	***Description***
.NET Framework Class Library support	Contains built-in types and libraries to manag assemblies, memory, security, threading, an other runtime system support
Debugging	Facilities for making it easier to debug code.
Exception management	Allows you to write code to create and handle exceptions.
Execution management	Manages the execution of code
Garbage collection	Automatic memory management and garbage collection.
Interop	Backward-compatibility with COM and Win32 code.
Just-In-Time (JIT) compilation	An efficiency feature for ensuring that the CLR only compiles code just before it executes
Security	Traditional role-based security support, in

	addition to Code Access Security (CAS)
Thread management	Allows you to run multiple threads of execution
Type loading	Finds and loads assemblies and types
Type safety	Ensures references match compatible types, which is very useful for reliable and secure code.

Table 1.3 .NET Languages

A#	Fortran	Phalanger (PHP)
APL	IronPython	Python
C++	IronRuby	RPG
C#	J#	Silverfrost FTN95
COBOL	Jscript	Scheme
Component Pascal	LSharp	SmallScript
Delphi	Mercury	Smalltalk
Delta Forth	Mondrian	TMT Pascal
Eiffel.NET	Oberon	VB.NET
F#	Perl	Zonnon

SELF ASSESSMENT EXERCISE

What is a type?

____
____
____
____

3.2 Microsoft C# Development

The most common way of programming with C# is by using the Visual Studio.NET developers integrated development environment (IDE). Prior to programming, you should install the latest version of the .NET

Framework to your PC running Windows. If you like you can install Visual Studio.NET which installs the .NET Framework automatically.

Once you have the .NET Framework installed, you also have the Framework SDK, which includes the command-line compiler for C#. The C# compiler is called csc.exe and exists in the following directory.

C:\WINDOWS\Microsoft.NET\Framework\v1.0.2914
The final directory named v1.0.2914 indicates the version of the .NET Framework that you have installed.

This directory is set to a path, using the advanced tab of the *System* control panel applet. (The Visual Studio .NET installation optionally sets my path for me). This lets me type csc from the command line in any directory in my file-system. (I find that I use Visual Studio for my big projects, but I use the command line compiler for my little tests and scripts.)

This is all you need to use C#. You can use any editor you like to create C# source code modules. You should give your C# models a .cs extension, and you are good to go.

3.2.1 Requirements

The VI SDK includes C# (.cs) source files and Microsoft Visual Studio project files (solutions or .sln) for both Microsoft Visual Studio 2003 and Microsoft Visual Studio 2005. Web services client application development for C# requires:

- Development environment for C#, such as Microsoft Visual C#, Microsoft Visual Studio 2003, or Microsoft Visual Studio 2005.

- Microsoft .NET Framework, specifically Microsoft .NET Framework 2.0 (which is included with Microsoft Visual Studio 2005).

- Microsoft .NET Framework 2.0 Software Development Kit. Depending on the specific version of the Microsoft Visual Studio and Microsoft .NET Framework that you use, you may need to also specifically install the Microsoft .NET Framework 2.0 software development kit, which includes the command-line C# compiler (csc.exe).

3.2.2 Setup Instructions for C# Development

Specific setup instructions will vary, depending on whether your development workstation already meets some or all of the requirements, and whether you plan to use the provided samples.

To set up a development workstation to use C#

1. Install the Microsoft Visual programming environment, such as Microsoft Visual Studio 2005, Microsoft Visual Studio .NET 2003, or Microsoft Visual C#.
   VMware recommends using Microsoft Visual Studio 2005, which includes the required .NET Framework 2.0 and improved versions of Web-services-client tools.

2. Obtain the Microsoft .NET Framework 2.0 or Microsoft .NET Framework 1.1. If you have been using Microsoft development tools for any length of time, it's likely you already have what's needed. If not, you can obtain Microsoft .NET Framework from Microsoft, at: http://msdn.microsoft.com

3. Obtain the VMware Infrastructure SDK (VI SDK) package from VMware Web site: http://www.vmware.com/download/sdk/

4. Unpack the various components into appropriate sub-directories. Use Microsoft defaults for Microsoft Visual Studio, Microsoft .NET Framework, and Microsoft .NET Framework SDK. To ensure that all paths to all tools (wsdl.exe, csc.exe) are set correctly, it's best to execute the command script from the Microsoft .NET 2.0 SDK command prompt, available from the .NET SDK menu.

To build the C# samples

1. Open the Microsoft .NET Framework 2.0 SDK command prompt (from the Windows Start menu, select **Programs > Microsoft .NET Framework SDK v2.0 > SDK Command Prompt**. The command prompt launches.

2. Navigate to the sub-directory containing the Build2005.cmd and other files:
   cd %SDKHOME%\samples\DotNet

3. Run the Build2005.cmd to generate new stubs and compile into an assembly. Simply enter the name of the script at the command prompt: build2005

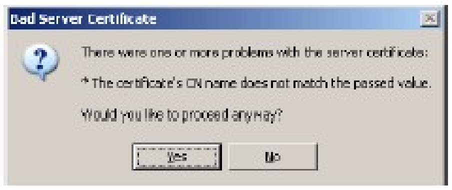
4. Test your development workstation.

3.2.3 Running the Microsoft.NET (C#)

These instructions assume that you have followed all setup instructions detailed in “Setup Instructions for C# Development” of this unit. For all intents and purposes, using Visual Studio 2003 is discouraged.

To run the SimpleClient application

1. Navigate to the sub-directory where the compiled object code is located. From the top-level directory of the SDK download, the directory is as follows:
   %SDKHOME%\samples\DotNet\cs\SimpleClient\bin\Debug
2. Run the application, passing to the command line the server URL and logon credentials. Note that, as coded, the SimpleClient application (SimpleClient.cs) requires that credentials (user account and password) be passed to the executable—even if the server has been configured to support HTTP. You can change the source code yourself and recompile. If you don’t provide all three arguments as shown here—server address, account name, and password—the executable generates an error message and comes to a halt.
   simpleclient https://sdkpubslab-01.eng.vmware.com/sdk *useraccount password*
   The application connects to the server. A “Bad Server Certificate” message displays:


Bad Server Certificate

There were one or more problems with the server certificate:

* The certificate's CN name does not match the passed value.

Would you like to proceed anyway?

Figure

3. Click **Yes** to dismiss this message and proceed to the server. Soon, the output from server should display in the console (command prompt), as shown in Example 4-2:

Example 1.0 Sample Output of a Successful Run of SimpleClient

```
Object Type : Datacenter
Reference Value : ha-datacenter
Property Name : name
Property Value : ha-datacenter
Object Type : Folder
Reference Value : ha-folder-root
Property Name : name
Property Value : ha-folder-root
```

3.2.4 Troubleshooting Setup Issues

If you are unable to successfully run the SimpleClient, check your environment settings and all other setup tasks. The table Table 4-3 lists some common error messages and provides steps that you can take to resolve.

Table 4-3. Microsoft C# Runtime Error Messages

Symptom	**Possible Solution**
SimpleClient.exe https://<management-server>/sdk <user> <pass> Caught Exception : Name : WebException Message : Unable to connect to the remote server Trace : at System.Net.HttpWebRequest.GetRequestStream() at System.Web.Services.Protocols.SoapHttpClientProtocol.Invoke(String methodName, Object[] parameters) at VimApi.VimService.RetrieveServiceContent(ManagedObject Reference _this) ... Exception disconnecting. Caught Exception : Name : NullReferenceException Message : Object reference not set to an instance of an object. Trace: ...	Cannot connect to the web service from Microsoft .NET client sample through proxy server. Try a different server on the same subnet as the client.

3.3 Hello World

After understanding the fundamentals of .NET and its Structure, let us now move on to look at a classic Hello C# Program. Essentially you can use any text editor depending on your convenience.

The coding for a simple Hello C# Program is as follows:

```
using System ;
Class Hello
{
Public static void Main ()
{
Console.writeLine ("Hello C#");
}
} //end of the main
```

3.3.1 Saving the Hello World File

The file above is saved as Hello.cs. Where ".cs" is an extension to indicate C-sharp, just like .java for a Java Source File. You have to supply this extension while saving your file, otherwise the code will not compile correctly. The saved file will be of .cs.txt extension.

Compile the above Code by giving the following Command at the Command Prompt **csc Hello.cs**
If there are compile errors then you will be prompted accordingly, otherwise you will be getting a com

3.3.2 Initial Hello World Program

If you run C# for the first time it will take slightly more time as it configures the environment for the first time. Once C# starts running, adopt the following steps:

- Select File -> New project
- From the project dialog, select the Console application
- This is the most basic application type on a Windows system
- Click Ok
- Visual C# Express creates a new project for you, including a file called Program.cs.

It should look somewhat like this:

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Now write the following two lines within the two inner braces

```
Console.WriteLine("Hello, world!");
Console.ReadLine();
```

The whole thing now should look as follows

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

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
           Console.WriteLine("Hello, world!");
           Console.ReadLine();
      }
   }
}
```

Hit F5 to run the above code. If you have not made any typographical error the program will actually run and display a black window displaying Hello World.
`
In the two lines that we wrote the first line '**Writes a line in the console window**'. The second line 'Reads a line in the window'. The second line is required because, without it the program will run and come out of it. So before you could observe the output on the console window, the program finishes the task and the console window is closed. Try running the code with only the first line and see what happens.

You have completed the installation of your first c# program. In the next unit we will develop detailed C# concepts.

3.3.3 Parts of the Hello World Program

Let us take a look at the first four lines of the code.

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

"using" is a keyword. A keyword is higlighted in blue in Microdoft IDE editor. This keyword ***using*** imports a namespace. A namespace is a collection of classes. We will get into the details of classes later on. For now you can assume the classes as a set of variables, properties and classes. In our program we have imported four namespaces. If you notice, we have our own namespace ConsoleApplication1.

```
namespace ConsoleApplication1
```

Next we define our own class. The C# is an Object Oriented language. Every line of code that actually does something, is wrapped inside a class. In this case, the class is called Program.

```
class Program
```

A class can contain several variables, properties and methods, concepts. It is important that you note that our current class only contains one method and nothing else. It's declared like this:

```
static void Main(string[] args)
```

We will now explain the above line. The first word static is a keyword. The static keyword tells us that this method should be accessible without creating an instance of the class. The next keyword is void, and tells us what this method should return. The void means it returns nothing. For instance, int could be an integer or a string. The next word is Main, this is the name of our method. This method is the so-called entry-point of our application, that is, the first piece of code to be executed, and in our example, the only piece to be executed. Now, after the name of a method, a set of arguments can be specified within a set of parentheses. In our example, our method takes only one argument, called args. The type of the argument is a string, or to be more precise, an array of strings. Windows applications can always be called with a set of arguments. These arguments will be passed as text strings to our main method.

4.0 CONCLUSION

From our studies in this unit, it is vital to remember that .NET supports multiple programming languages. It is equally worth noting that some specific setup instructions are required for C# development.

5.0 SUMMARY

In this unit, we looked at .NET and the programming languages it supports. We equally considered the setup instructions for C# development as well as the 'Hello World' program. We hope you found the unit enlightening. To assess your comprehension, attempt the questions below.

6.0 TUTOR MARKED ASSIGNMENT

What is the relevance of the Common Language Runtime (CLR)?
Outline the procedure for running a Simpleclient application

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.

10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Kay, Alan. *The Early History of Smalltalk.* http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
12. Martin, A and Luca, C. (2005). *A Theory of Objects*.
13. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
14. Michael Lee Scott (2006). *Programming language pragmatics,* (2nd Edition) p. 470
15. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
16. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
17. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
18. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
19. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 3 GETTING STARTED WITH C#

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
    3.1 C# Console Application
        3.1.1 Saving in C#.NET
    3.2 C# Code
    3.3 Running C# Programmes
    3.4 C# Windows Applications
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

The initial task we have in this unit is to create a very simple programme, so that you can see what makes up a C# .NET project. You might be tempted to think you're never going to get to grips with the project. But don't worry - after a few lessons, things will start to feel familiar, and you will gain more confidence.

2.0 OBJECTIVES

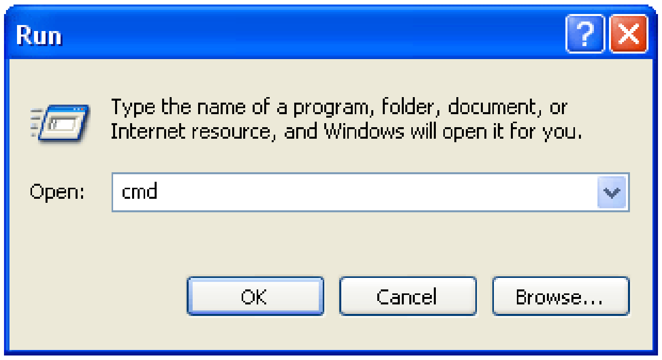
By the end of this unit, you'll have learnt the following:

- How to create new projects
- What the Solution Explorer is
- The various files that make up of a C# .NET project
- How to save your work
- How to run programmes

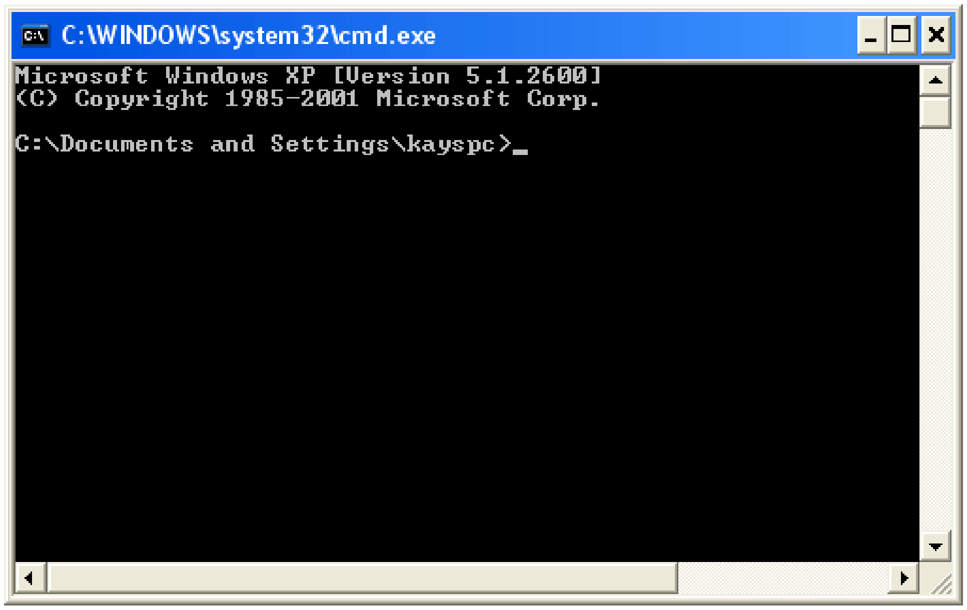
3.0 MAIN CONTENT

3.1 C# Console Application

A **Console** Application is an application that takes input and displays output at the console. This console looks like a DOS window. If you need to see what the DOS window looks like, click your **Start** menu in the bottom left of your screen. Click on **Run**. From the dialogue box that appears, type **cmd**:


Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open:
cmd
OK
Cancel
Browse...

Click OK and you'll see a black screen, like this one:

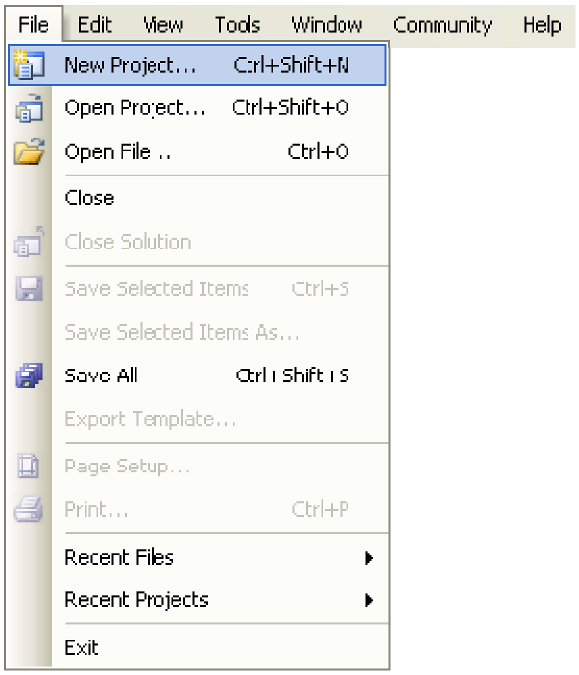


```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\kayspc>_
```

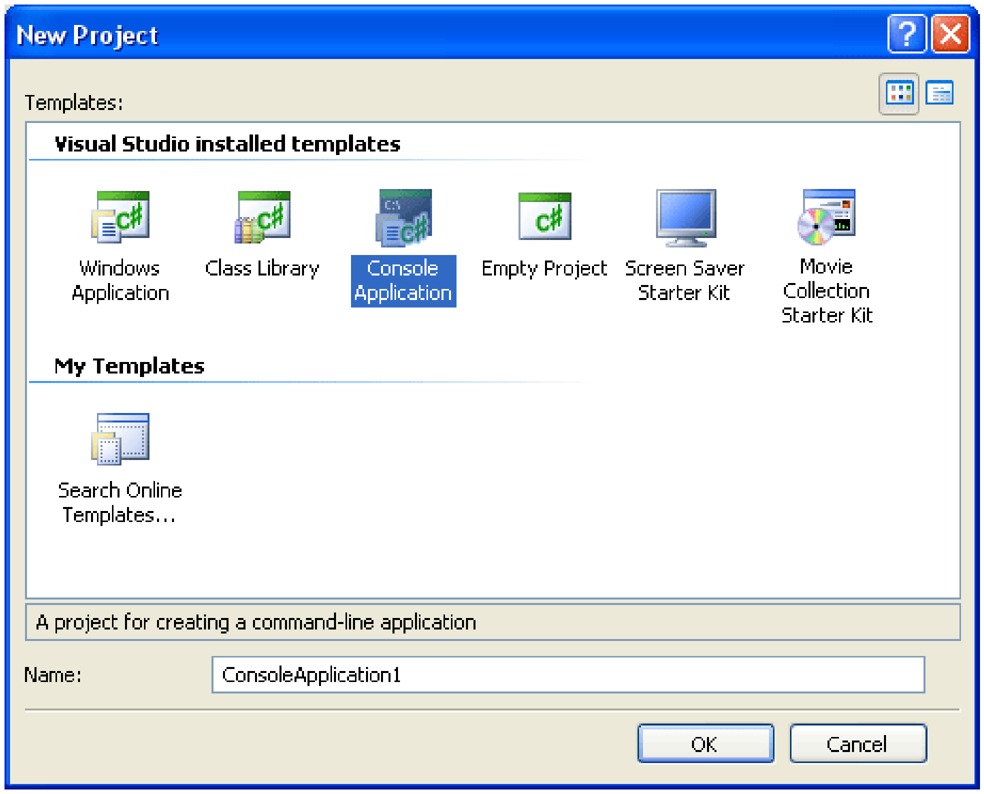
This is the type of window you'll see for Console Applications. These applications are ideal for learning C# development because the user interface is so simple. Console applications are also very useful for utility programs that require little or no user interaction.

On the other hand, when you create your Windows forms, there's a whole lot of code to get used to. But Console Applications start off fairly simple, and you can see which part of the programme is the most important.

So with Visual C# Express open, click **File** from the menu bar at the top. From the File menu, select **New Project**:



When you click on New Project, you'll see the following dialogue box appear:

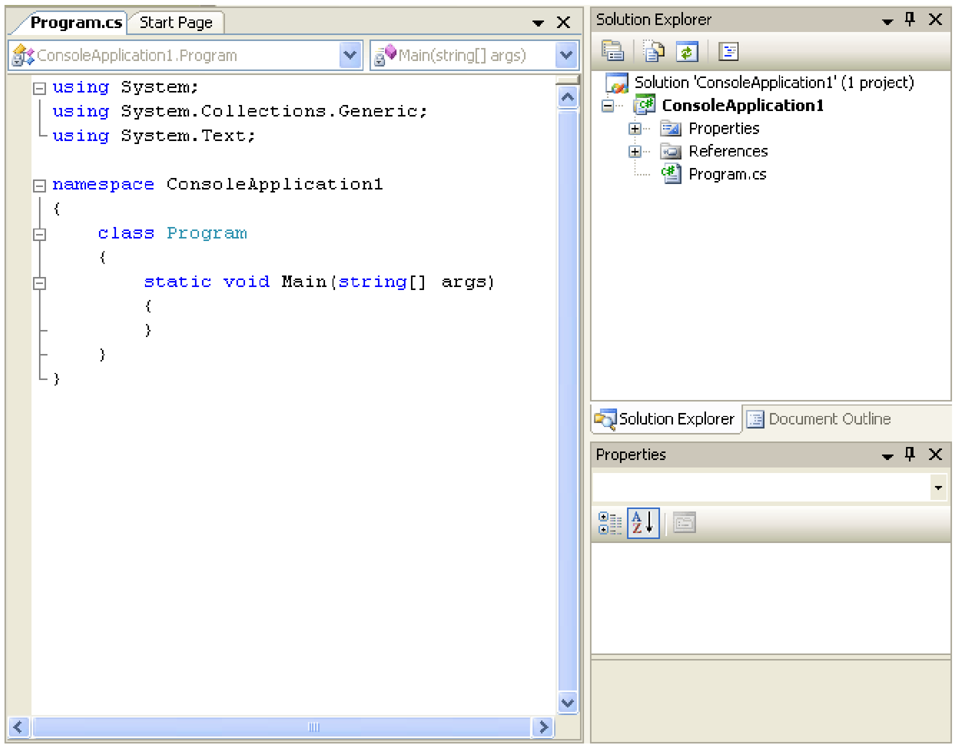


A project for creating a command-line application

Name: ConsoleApplication1

This is where you select the type of project you want to create. If you only have the Express edition of Visual C#, the options are limited. For the rest of this book, we'll be creating Windows Applications. For now, select **Console Application**. Then click OK.

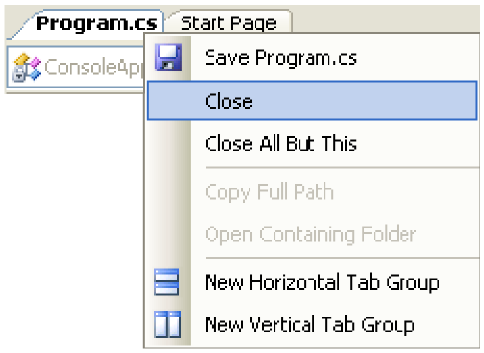
When you click OK, a new Console Application project will be created for you. Some code should be displayed:



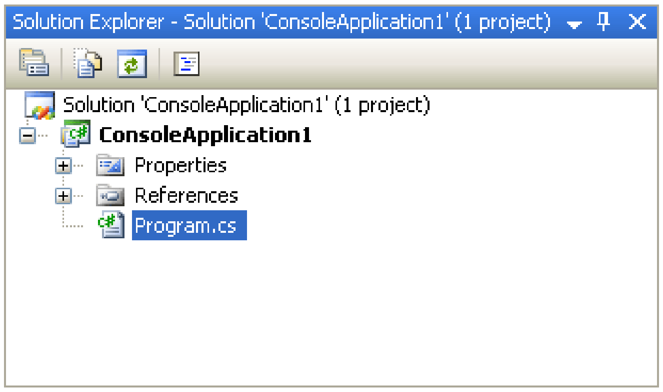
```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

As well as the code, have a look on the right hand side and you'll see the Solution Explorer. This is where all the files for your project are. (If you can't see the Solution Explorer, click **View** from the C# menu bar at the top. From the View menu, click **Solution Explorer**.)

The code itself will look very complicated, if you're new to programming. We'll get to it shortly. For now, right click the **Program.cs** tab at the top, and click **Close** from the menu that appears:

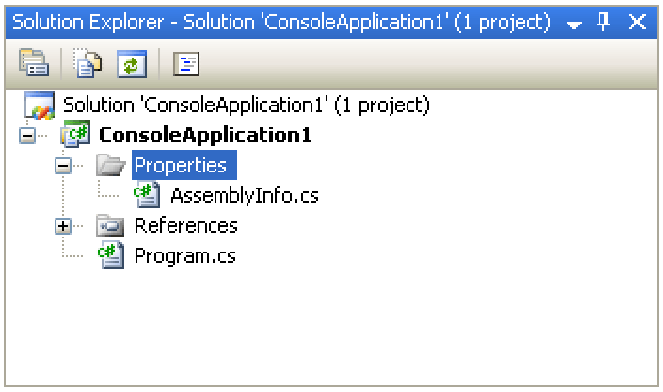


Now double click the Program.cs file in the **Solution Explorer**:


Solution Explorer - Solution 'ConsoleApplication1' (1 project)
Solution 'ConsoleApplication1' (1 project)
ConsoleApplication1
Properties
References
Program.cs

When you double click Program.cs, you should see the code reappear. So this code is the programme that will run when anyone starts your application.

Now click the plus symbol next to Properties in the Solution Explorer above. You'll see the following:

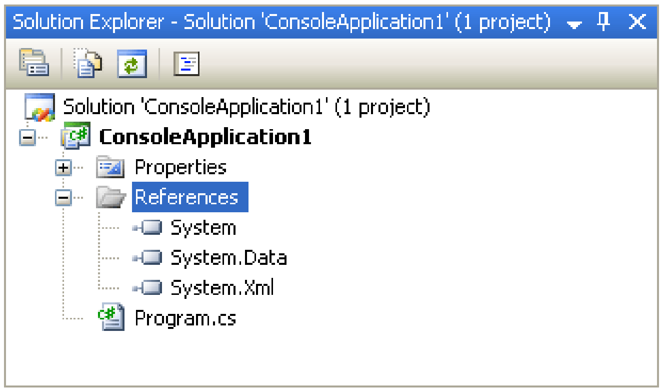

Solution Explorer - Solution 'ConsoleApplication1' (1 project)
Solution 'ConsoleApplication1' (1 project)
ConsoleApplication1
Properties
AssemblyInfo.cs
References
Program.cs

The file called **AssemblyInfo.cs** contains information about your programme. Double click this file to open it up and see the code. Here's just some of it:

```
[assembly: AssemblyTitle("ConsoleApplication1")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("zzz")]
[assembly: AssemblyProduct("ConsoleApplication1")]
[assembly: AssemblyCopyright("Copyright © zzz 2007")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

The reddish colour text is something you can change. You can add a Title, Description, Copyright, Trademark, etc.

But right click the AssemblyInfo.cs tab at the top, and click **Close** from the menu. Now, in the Solution Explorer, click the plus symbol next to References:


Solution Explorer - Solution 'ConsoleApplication1' (1 project)
Solution 'ConsoleApplication1' (1 project)
ConsoleApplication1
Properties
References
System
System.Data
System.Xml
Program.cs

These are references to code built in to C#. Much later, you'll see how to add your own files to this section.

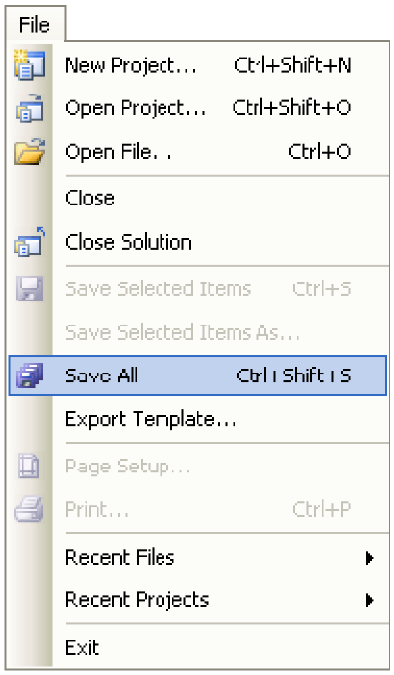
Before we add some code, let's save the project. We'll do that in the next part below.

3.1.1 Saving in C#.NET

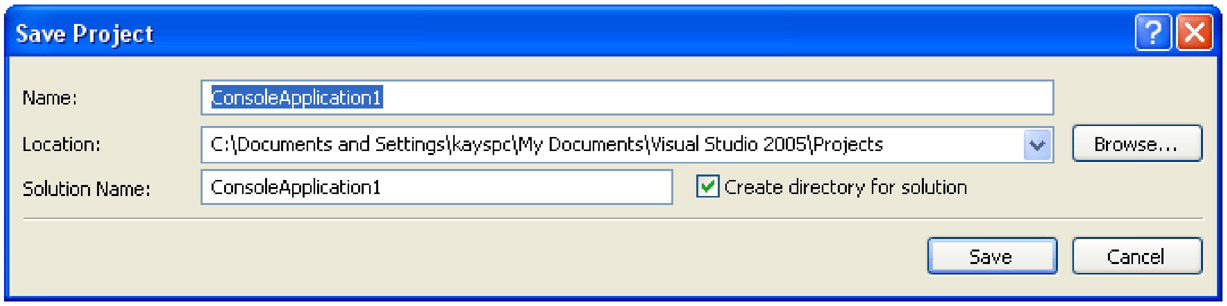
When you save your work, C# will create quite a few folders and files for you.

The procedure for saving in C#.NET is as follows:

- Click **File** from the menu bar at the top of the Visual C# Express software, then **Save All**:



- When you click Save All, you'll see the following dialogue box appear:


Save Project

Name: ConsoleApplication1

Location: C:\Documents and Settings\kayspc\My Documents\Visual Studio 2005\Projects

Solution Name: ConsoleApplication1

[✓] Create directory for solution

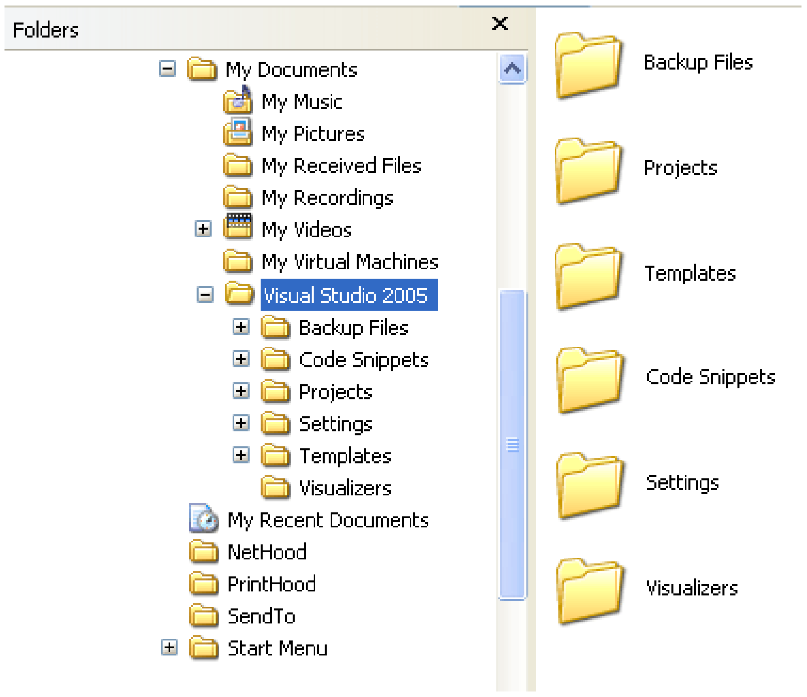
- You can type any name you like for your project. The default Name is ConsoleApplication1. Have a look at the location of the project, though:

C:\Documents and Settings\kayspc\My Documents\Visual Studio 2005\Projects

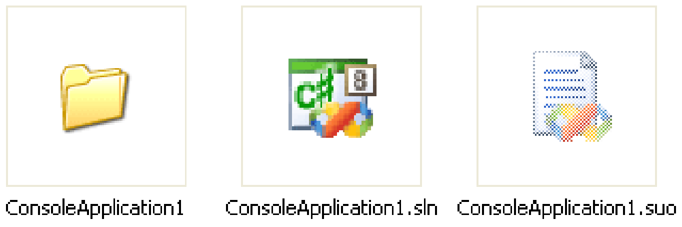
So it's going to be saved to the "My Documents" folder of this computer (XP). In the "My Documents" folder you'll find another one called **Visual Studio 2005**. In this folder there will be one called **Projects**.

- Before clicking the Save button, ensure that there is a tick in the box for "Create directory for solution". Then click **Save**.

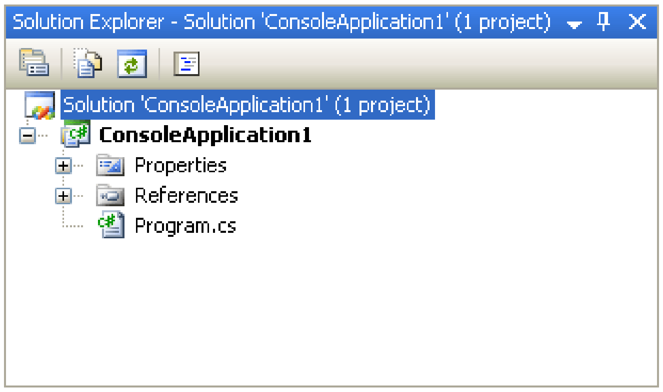
- Now open up Windows Explorer (Hold down the Windows key on your keyboard, then press the letter "e"), or just double click the "My Computer" icon on your desktop. Navigate to the folder location above. In the image below, we've used Windows Explorer to navigate to the Visual Studio 2005 folder (the 2008 version should be the same):



- Double click the **Projects** folder to see inside of it. You should see a folder called **ConsoleApplication1**. Double click this folder and you'll see the following:

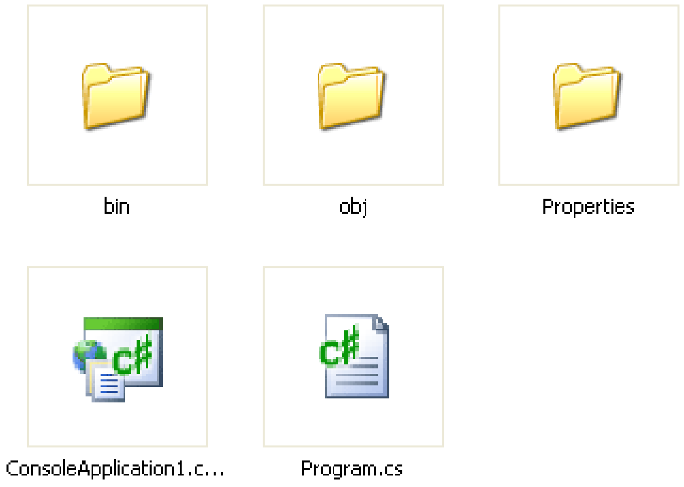


So there's another folder called ConsoleApplication1. There are two other files: one that ends in **sln**, and another that ends in **suo**. The **sln** file is the entire solution. Have a look at the Solution Explorer again:

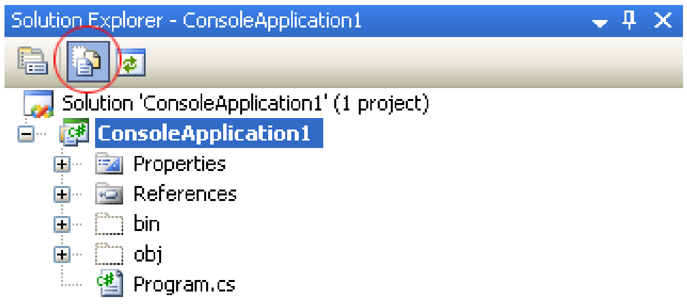

Solution Explorer - Solution 'ConsoleApplication1' (1 project)
Solution 'ConsoleApplication1' (1 project)
ConsoleApplication1
Properties
References
Program.cs

The one highlighted in blue at the top refers to the **sln** file. The **suo** file contains information about the Visual Studio environment - whether the plus symbols are expanded in the Solution Explorer, what other files you have open in this project, and a whole host of other settings. (If you can't see this file, click **Tools** > **Folder Option** in Windows Explorer. Click the **View** tab, and select the option for "Show hidden files and folders".)

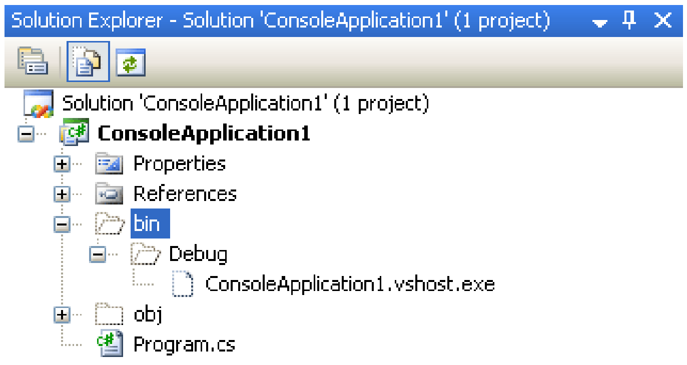
Double click your ConsoleApplication1 folder, though, to see inside of it:



Now we have three more folders and two files. You can see the **bin** and **obj** folders in the Solution Explorer:


Solution Explorer - ConsoleApplication1
Solution 'ConsoleApplication1' (1 project)
ConsoleApplication1
Properties
References
bin
obj
Program.cs

Click ConsoleApplication1, second from the top. Then click the icon for **Show all Files**, circled in red in the image above. The **bin** and **obj** folders will appear. Click the plus arrows to see what's inside of these folders:


Solution Explorer - Solution 'ConsoleApplication1' (1 project)
Solution 'ConsoleApplication1' (1 project)
ConsoleApplication1
Properties
References
bin
Debug
ConsoleApplication1.vshost.exe
obj
Program.cs

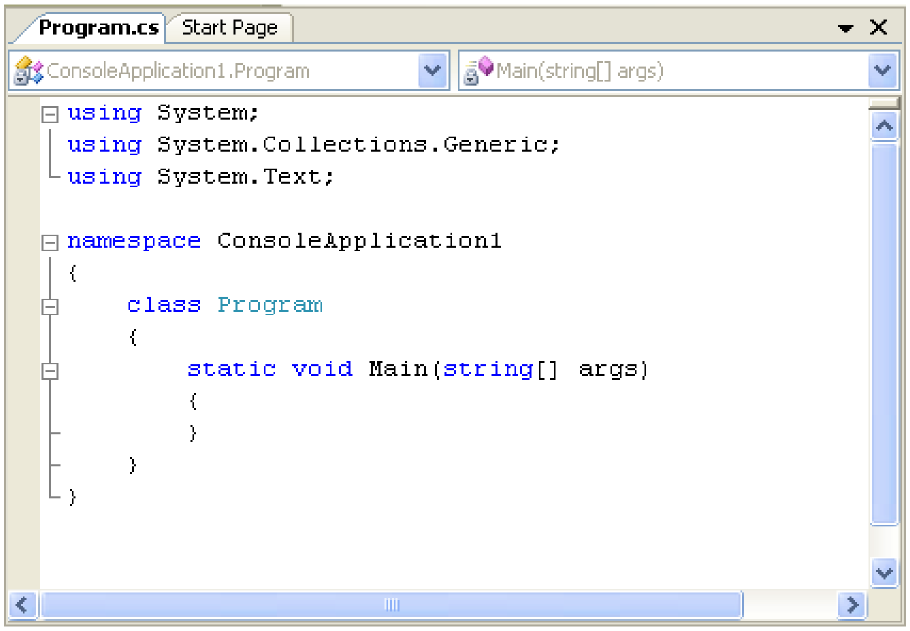
The important one for us is the Debug folder under **bin**. You'll see why in a moment. However, it's time to write some code!

SELF ASSESSMENT EXERCISE

What sort of information is found in the suo file?

3.2 C# Code

The only thing we'll do with the code is to write some text to the screen. But here's the code that Visual C# prepares for you when you first create a Console Application:



```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

The 3 lines that start with **using** add references to in-built code. The **namespace** line includes the name of your application. A namespace is a way to group related code together.

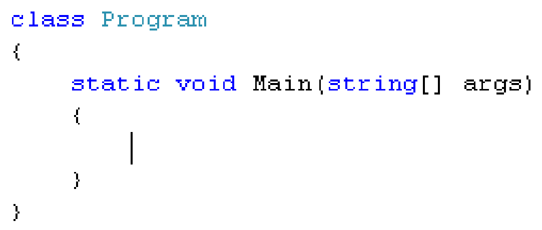
You'll need to take note of the word **class**. All your code will be written in classes. This one is called **Program** (you can call them anything you like, as long as C# hasn't taken the word for itself). But think of a class as a segment of code that you give a name to.

Inside of the class called Program there is this code:

```
static void Main(string[] args)
{
}
```

This piece of code is known as a **Method**. The name of the Method above is **Main**. When you run your programme, C# looks for a Method called Main. It uses the Main Method as the starting point for your programmes. It then executes any code between those two curly brackets. The blue words above are all special words referred to as **keywords**.

But position your cursor after the first curly bracket, and then hit the enter key on your keyboard:



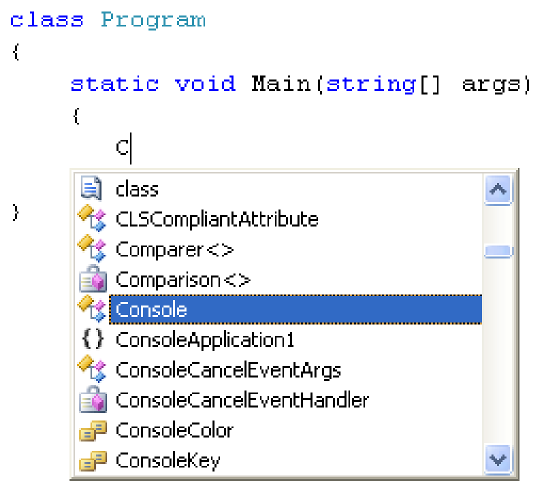
```
class Program
{
    static void Main(string[] args)
    {
        
    }
}
```

The cursor automatically indents for you, ready to type something. Note where the curly brackets are, though in the code above you have a pair for **class Program**, and a pair for the **Main method**. However, if you miss one out and you'll get error messages.

The single line of code we'll write is this (but don't write it yet):

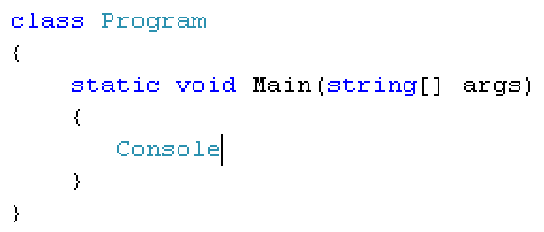
Console.WriteLine("Hello C Sharp!");

First, type the letter "C". You'll see a popup menu. This popup menu is called the IntelliSense menu. It tries to guess what you want, and allows you to quickly add the item from the list. But it should look like this, after you have typed a capital letter "C":



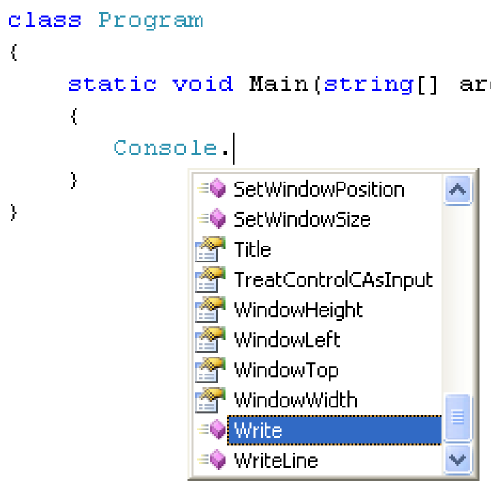
```
class Program
{
    static void Main(string[] args)
    {
        C
}
```

The icon to the left of the word **Console** on the list above means that it is a Class. But press the Enter key on your keyboard. The word will be added to your code:



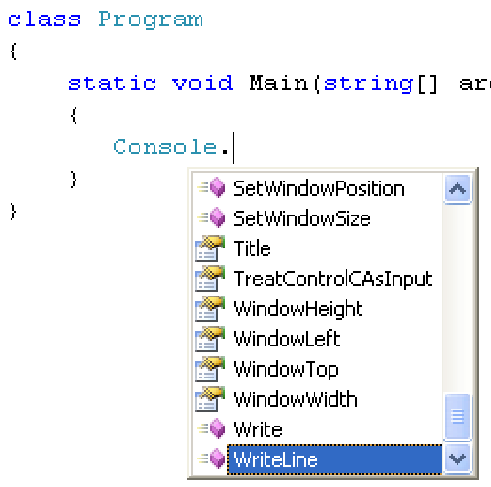
```
class Program
{
    static void Main(string[] args)
    {
        Console
    }
}
```

Now type a full stop (period) immediately after the word Console. The IntelliSense menu appears again:



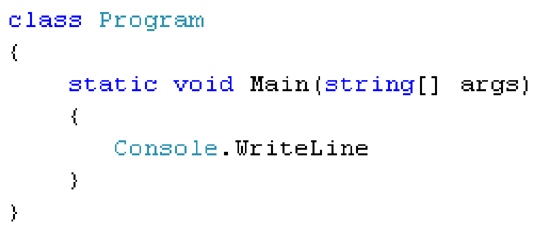
```
class Program
{
    static void Main(string[] args)
    {
        Console.
    }
}
```

You can use the arrow keys on your keyboard to move up or down the list. But if you type **Write** and then the letter **L** of **Line**, IntelliSense will automatically move down and select it for you:



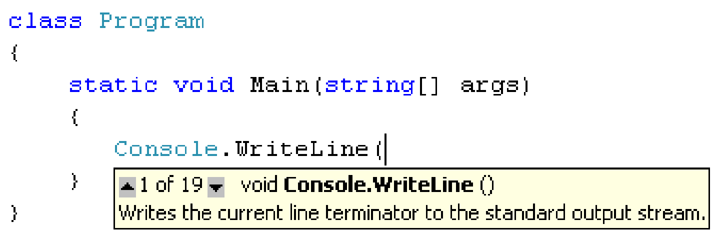
```
class Program
{
    static void Main(string[] args)
    {
        Console.
    }
}
```

Press the Enter key to add the word **WriteLine** to your code:



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

Now type a left round bracket. As soon as you type the round bracket, you'll see this:

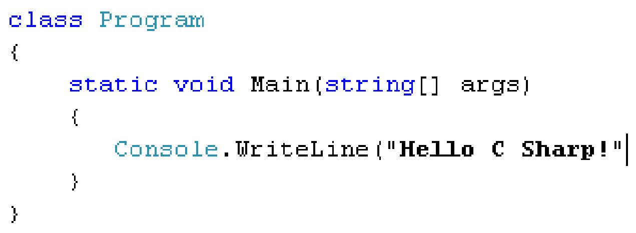


```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(
    }    ▲1 of 19 ▼ void Console.WriteLine ()
}        Writes the current line terminator to the standard output stream.
```

WriteLine is another Method (A Method is just some code that does a particular job). But the yellow box is telling you that there are 19 different versions of this Method. You could click the small arrows to move up and down the list. Instead, type the following:

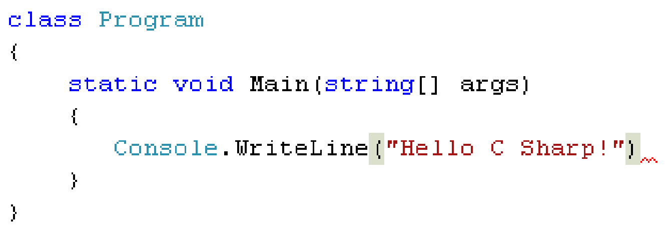
"Hello C Sharp!"

Don't forget the double quotes at the start and end. These tell C# that you want text. Your code will look like this:



```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello C Sharp!"
    }
}
```

Now type a right round bracket:



```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello C Sharp!")
    }
}
```

Notice the red wiggly line at the end. This is the coding environment's way of telling you that you've missed something out.

The thing we've missed out is a semicolon. All complete lines of code in C# must end with a semicolon. Miss one out and you'll get error messages. Type the semicolon at the end and the red wiggly line will go away. Your code should now look like this:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello C Sharp!");
    }
}
```

Note all the different colours. Visual C# colour-codes the different parts of your code. The reddish colour between double quotes means that they are texts; the green colour means it's a Class; blue words are words that C# reserves for itself.

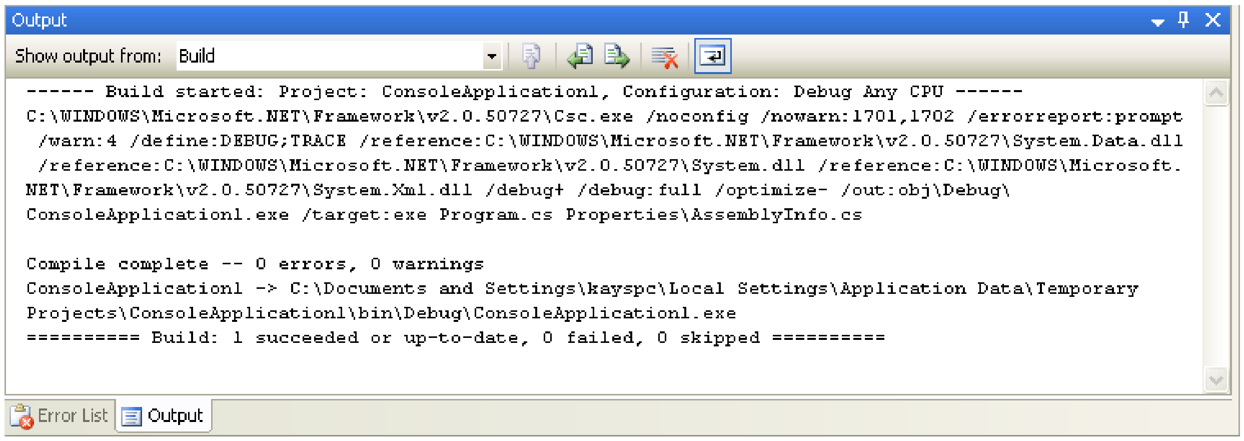
However, you can change these colours. From the menu bar at the top, click **Tools** > **Options**. Under **Environment**, click **Fonts and Colors**.)

It's time to Build and Run your code!

3.3 Running C# Programmes

You can test your programme a number of ways. This is after it has been built. This is when everything is checked to see if there are any errors. Try this:

- From the **View** menu at the top of Visual C# Express, click **Output**. You'll see a window appear at the bottom
- From the **Build** menu at the top of Visual C# Express, click **Build Solution**
- You should see the following report:



```
------ Build started: Project: ConsoleApplication1, Configuration: Debug Any CPU ------
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\Csc.exe /noconfig /nowarn:1701,1702 /errorreport:prompt
 /warn:4 /define:DEBUG;TRACE /reference:C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Data.dll
 /reference:C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.dll /reference:C:\WINDOWS\Microsoft.
NET\Framework\v2.0.50727\System.Xml.dll /debug+ /debug:full /optimize- /out:obj\Debug\
ConsoleApplication1.exe /target:exe Program.cs Properties\AssemblyInfo.cs

Compile complete -- 0 errors, 0 warnings
ConsoleApplication1 -> C:\Documents and Settings\kayspc\Local Settings\Application Data\Temporary
Projects\ConsoleApplication1\bin\Debug\ConsoleApplication1.exe
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

The final line is this:

Build: 1 succeeded or up-to-date, 0 failed, 0 skipped

That's telling you that everything is OK.

Now try this:

- Delete the semicolon from the end of your line of code
- Click **Build** > **Build Solution** again
- Examine the output window

This time, you should see these two lines at the end of the report:

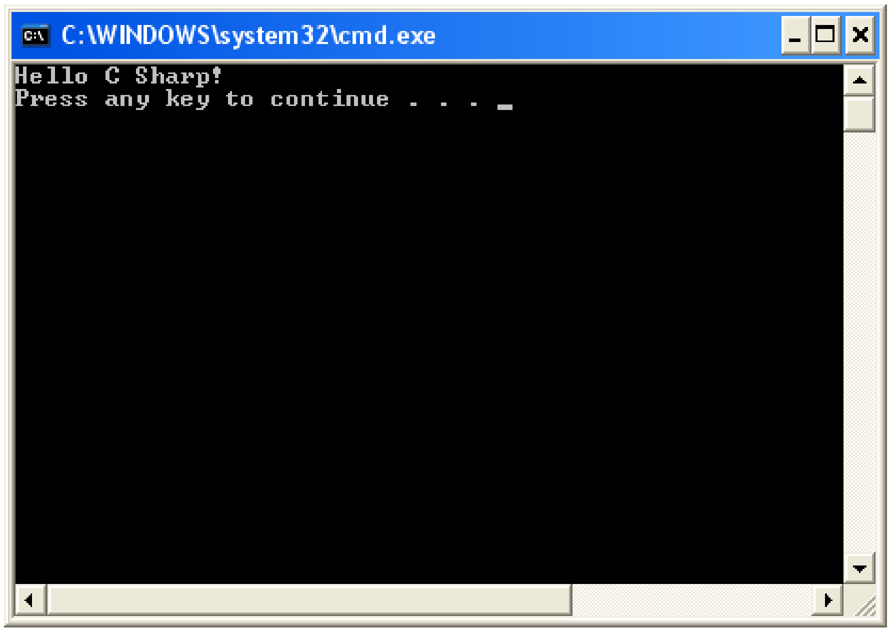
Compile complete -- 1 errors, 0 warnings
Build: 0 succeeded or up-to-date, 1 failed, 0 skipped

This indicates that it couldn't build your solution because there was 1 error.

Put the semicolon back at the end of the line. Now click **Debug** from the menu at the top of Visual C# Express. From the Debug menu, select **Start Debugging**.

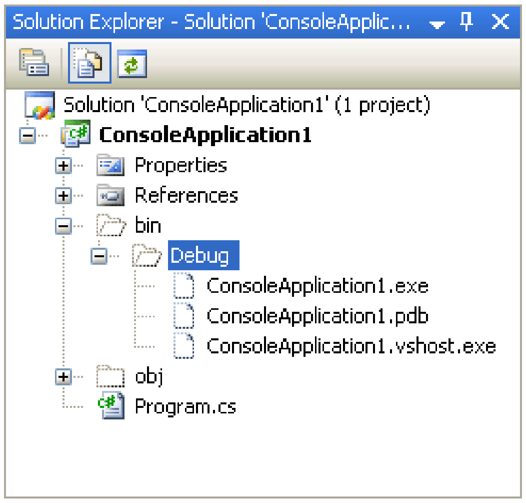
You should see a black DOS window appear and then disappear. Your programme has run successfully!

To actually see your line of text, click **Debug** > **Start Without Debugging**. You should now see this:

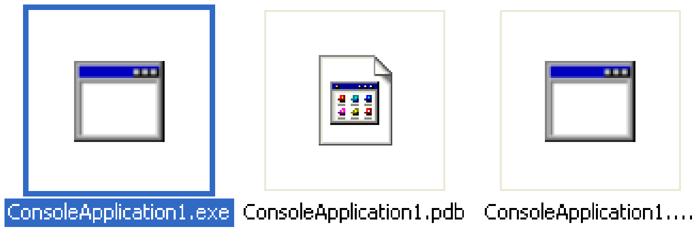


```
Hello C Sharp!
Press any key to continue . . . _
```

And that's your programme! Have a look at the Solution Explorer on the right. Because the project has been built, you'll see two more files under **Debug**:


Solution Explorer - Solution 'ConsoleApplic...
Solution 'ConsoleApplication1' (1 project)
ConsoleApplication1
Properties
References
bin
Debug
ConsoleApplication1.exe
ConsoleApplication1.pdb
ConsoleApplication1.vshost.exe
obj
Program.cs

We now have a ConsoleApplication1.exe and ConsoleApplication1.pdb. The **exe** file is an executable programme, and it appears in the **bin/debug** folder. Switch back to Windows Explorer, if you still have it open. You'll see the exe file there:



You could, if you wanted, create a desktop shortcut to this **exe** file. When you double click the desktop shortcut, the programme will run.

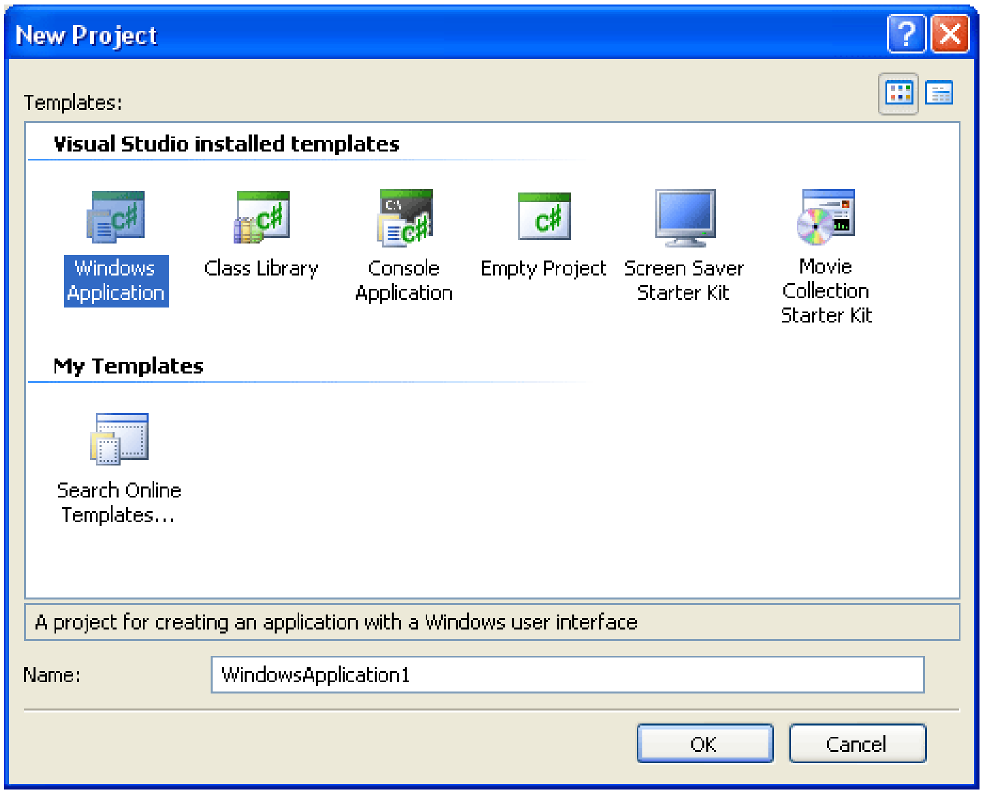
But that's enough of Console Applications - we'll move on to creating Windows Applications.

3.4 C# Windows Applications

From now on, we're going to be creating Windows Applications, rather than Console Applications. Windows Applications make use of something called a **Form**. The Form is blank at first. You then add control to your form, things like buttons, text boxes, menus, check boxes, radio buttons, etc. To get your first look at a Windows Form, do the following.

If you still have your Console Application open from the previous section, click **File** from the menu bar at the top of Visual C# Express. From the File menu, click on **Close Solution**.

To create your first Windows form project, click the File menu again. This time, select **New Project** from the menu. When you do, you'll see the New Project dialogue box again:

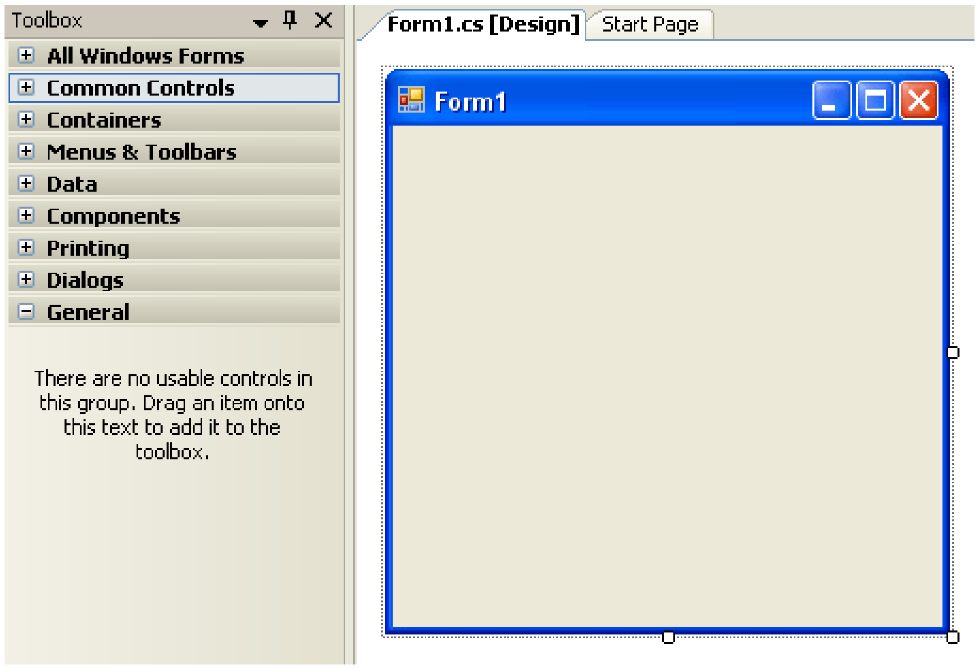


A project for creating an application with a Windows user interface

Name: WindowsApplication1

Select **Windows Application** from the available templates. Keep the **Name** on the default of **WindowsApplication1** and then click OK.

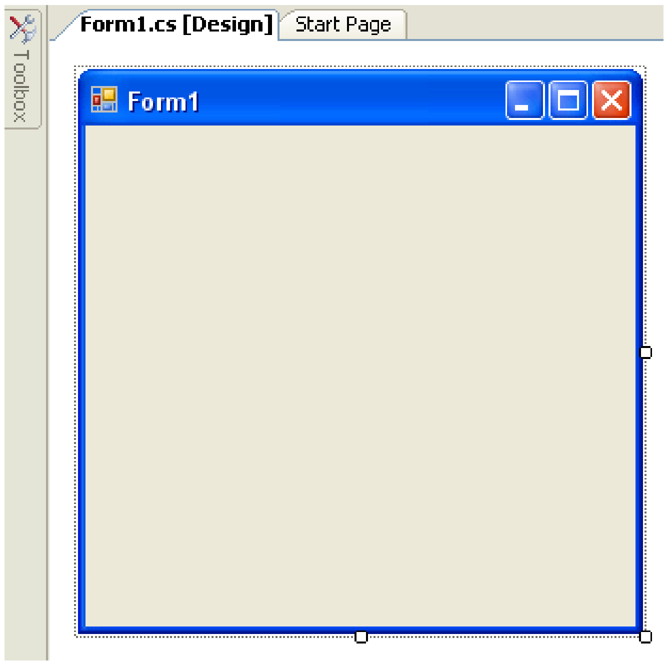
When you click OK, a new Windows Application project will be created for you:



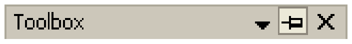
There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

The obvious difference from the Console Application you created in the previous section is the blank Form in the main window. Notice the **Toolbox**, though, on the left hand side. We'll be adding controls from the Toolbox to that blank **Form1** you can see in the image above.

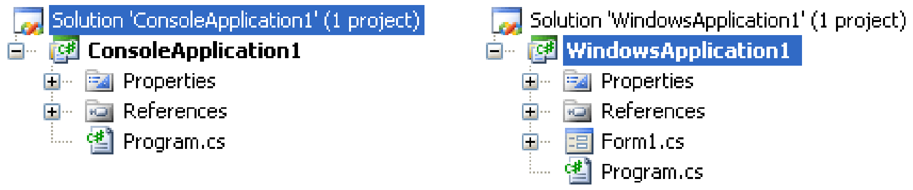
If you can't see the Toolbox, you may just see the Tab, as in the following image:



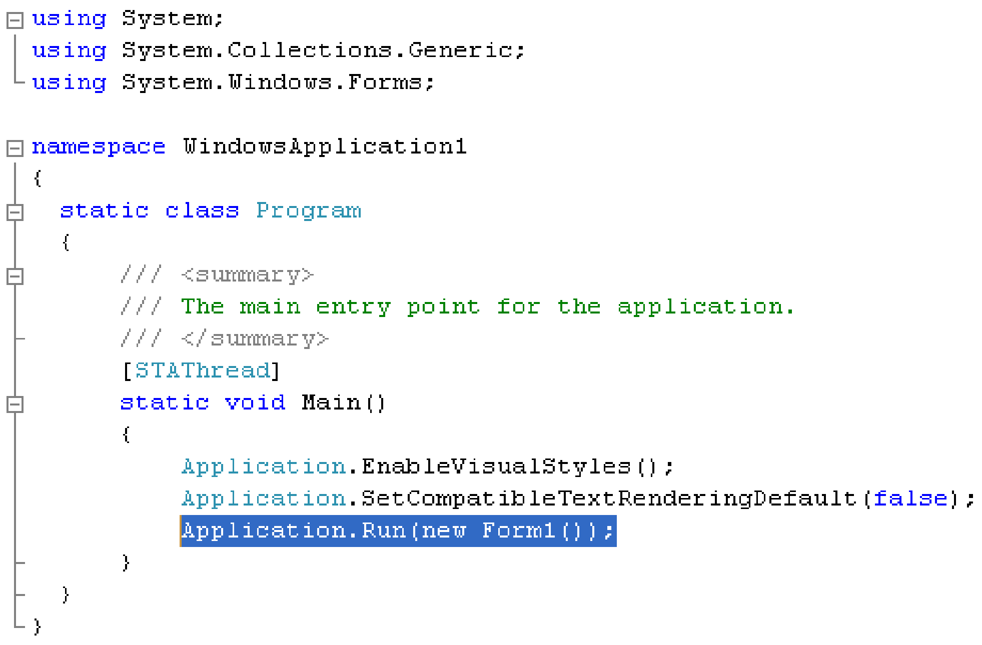
If your screen looks like the one above, move your mouse over to the Toolbox tab. It will expand to look like the first one. If you want to permanently display the Toolbox, click on the pin symbol:



Notice the Solution Explorer on the right side of your screen. (If you can't see the Solution Explorer, click its entry on the View menu at the top of Visual C# Express.) If you compare it with the Solution Explorer when you created your Console Application, you'll see there's only one difference - the Form.



We still have the Properties, the References and the Program.cs file. Double click the Program.cs file to open it, and you'll see some familiar code:



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

namespace WindowsApplication1
{
  static class Program
  {
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main()
      {
          Application.EnableVisualStyles();
          Application.SetCompatibleTextRenderingDefault(false);
          Application.Run(new Form1());
      }
  }
}
```

And here's the code from the Console Application:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

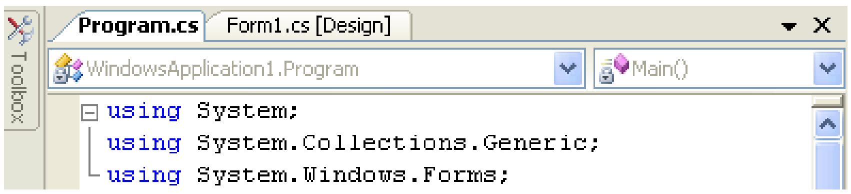
Both have the same using lines, a namespace, a class called Program, and a **Main** Method.

The Main Method is the entry point for your programme. The code between the curly brackets of Main will get executed when the programme first starts. The last line in the **WindowsApplication1** code above is the one that Runs Form1 when the Application starts.

You can do other things here. For example, suppose you had a programme that connects to a server. If it finds a connection then it loads some information from a database. In the Main Method, you could check that the server connection is OK. If it's not, display a second form; if it's OK, then display the first form.

But don't worry if all that code has you scratching your head. The thing to bear in mind here is that a method called **Main** starts your programme. And **Program.cs** in the Solution Explorer on the right is where the code for Main lives.

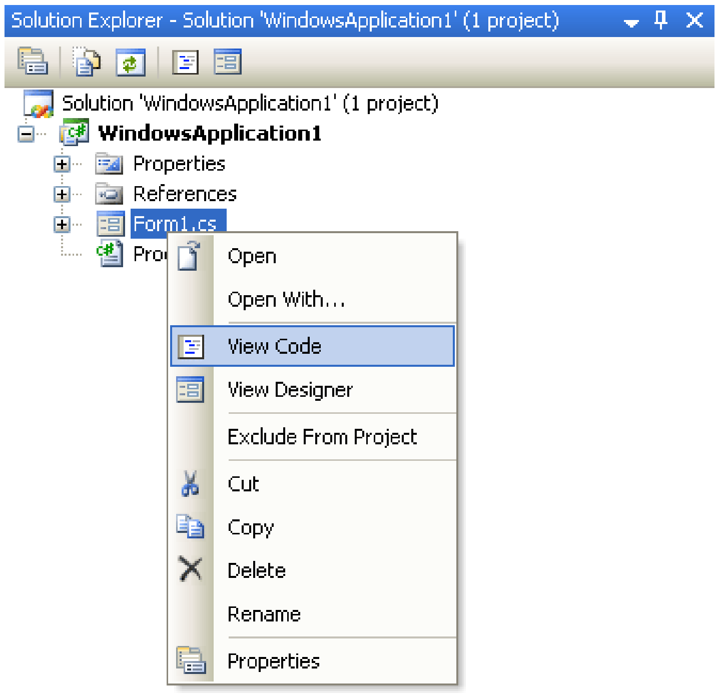
But we won't be writing code in the Program.cs file, so we can close it. Have a look near the top of the coding window, and you'll see some tabs:



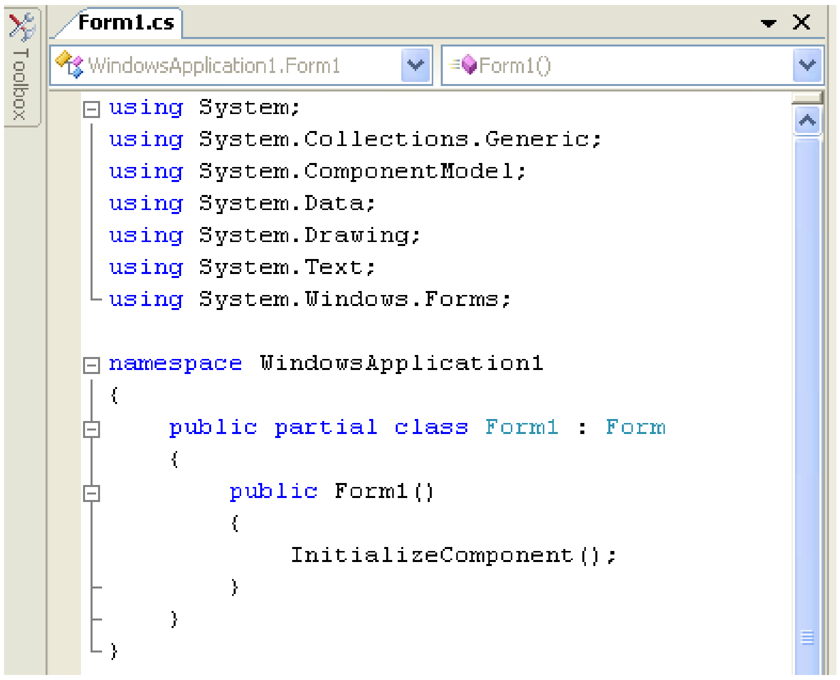
```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
```

Right click the Program.cs tab and select **Close** from the menu that appears. You should now see your form again (you may have a Start tab as well. You can close this, if you want).

To see the window where you'll write most of your code, right click **Form1.cs** in the Solution Explorer:

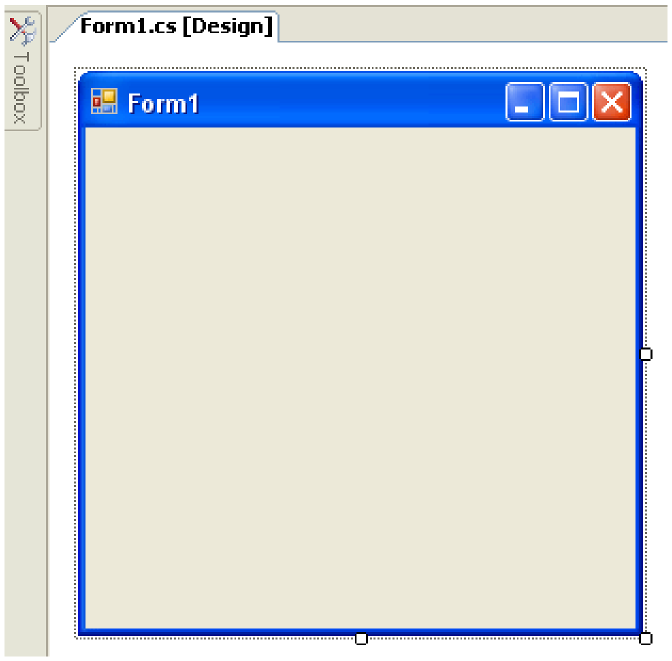


The menu has options for **View Code** and **View Designer**. The Designer is the Form you can see at the moment. Click **View Code** from the menu to see the following window appear (you can also press the F7 key on your keyboard):



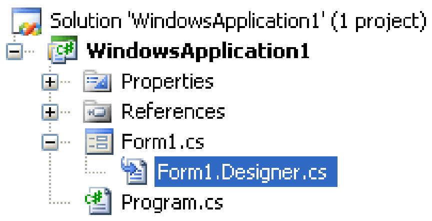
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

This is the code for the Form itself. This Form:

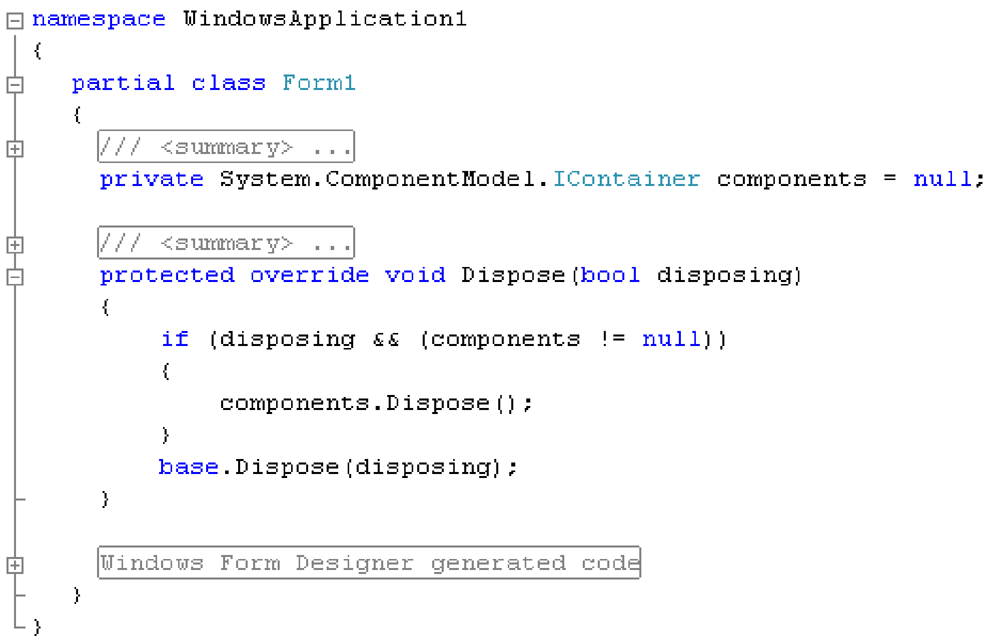


The code has a lot more **using** statements than before. Don't worry about these for now. They just mean "using some code that's already been written".

The code also says partial class Form1. 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 symbol next to Form1.cs in the Solution Explorer:

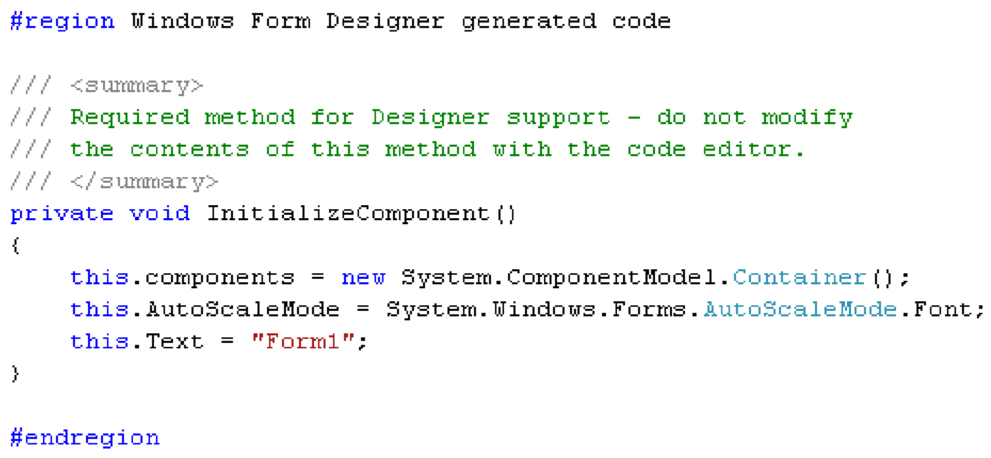


Now double click **Form1.Designer.cs**. You'll see the following code:



```
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
    }
}
```

Again, you see partial class Form1, which is the rest of the code. Click the plus symbol next to **Windows Form Designer generated code**. You'll see the following:

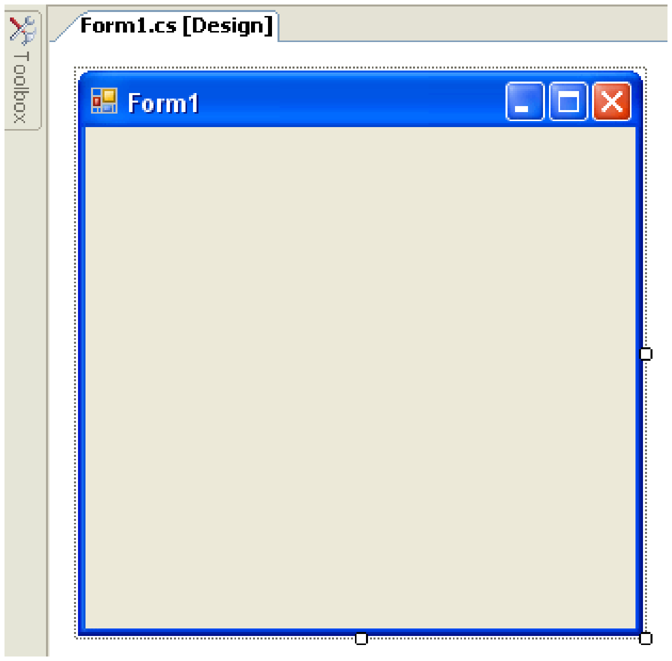


```
#region Windows Form Designer generated code

/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
}

#endregion
```

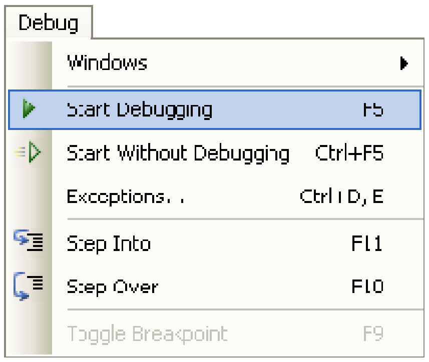
InitializeComponent is a code (a Method) that is automatically generated for you when you create a new Windows Application project. As you add things like buttons and text boxes to your form, more code will be added here for you.

But you don't need to do anything in this window, so you can right click the **Form1.Designer.cs** tab at the top, and click **Close** from the menu.

Click back on the **Form1.cs** tab at the top to see you form again. If the tab is not there, right click Form1.cs in the Solution Explorer on the right. From the menu, select **View Designer**. Here's what you should be looking at:



It's in Designer view that we'll be adding things like buttons and text boxes to our form. But you can run this programme as it is. From the **Debug** menu at the top, click **Start Debugging** (Or you can just press the F5 key on your keyboard.):



When you click Start Debugging, Visual C# will Build the programme first, and then run it, if it can. If it can't run your programme you'll see error messages.

But you should see your form running on top of Visual Studio. It will have its own Red X and its own minimize and maximize buttons. Click the Red X to close your programme, and to return to Visual C# Express.

From now on, when we say Run your programme, this is what we mean: either press F5, or click **Debug** > **Start Debugging**.

4.0 CONCLUSION

Two main applications in C# are console and windows applications. Double quotes are used to indicate texts. A code is said to be partial if some codes are hidden. Debugging is used to fix error in C# programs.

5.0 SUMMARY

In this unit, we considered the console and windows applications in C#. We equally looked at the procedure for these two applications as well as debugging. Hoping that you understood the topics discussed, you may now attempt the questions below.

6.0 TUTOR MARKED ASSIGNMENT

What is the function of the double quote in C#? 47
Outline the procedure for debugging a C# program. 49
What makes a code partial? 56

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.

8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.

UNIT 4 COMMON VARIABLES IN C#.NET

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
3.1 String Variables
3.2 Setting up String variables
3.2.1 Characters for Variables
3.3 Assigning Texts to Strings
3.4 C# Comments
3.5 C# Number Variables
3.5.1 C# Integers
3.5 Double and Float Variables
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

Programmes work by manipulating data stored in memory. These storage areas come under the general heading of **Variables**. In this unit, we'll be looking at how to set up and use variables. You'll see how to set up both text and number variables. Setting up both text and number variables likewise form part of this unit.

2.0 OBJECTIVES

By the end of this unit, you'll have learnt the following:

- Identify the common variables in C#
- Explain how texts are assigned to strings in C#
- State how comments are incorporated in C#

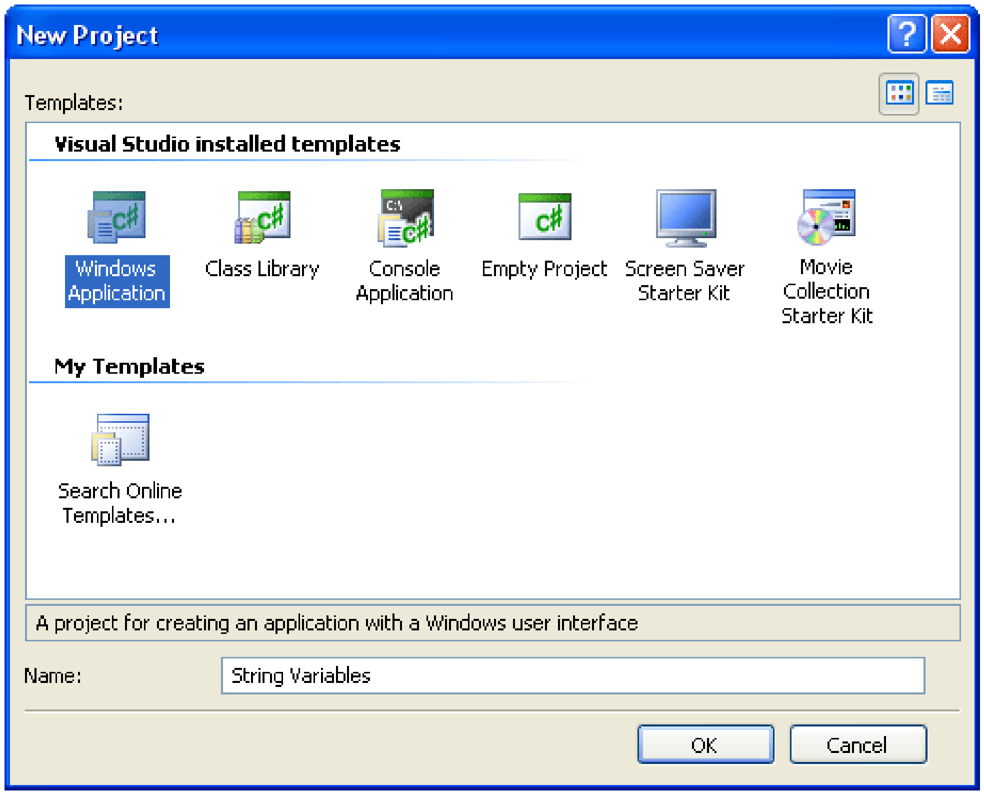
3.0 MAIN CONTENT

3.1 String Variables

The first type of variable we'll take a look at is called a **String**. String variables are always text. We'll write a little programme that takes text from a text box, store the text in a variable, and then display the text in a message box.

But bear in mind that a variable is just a storage area for holding things that you'll need later. Think of them like boxes in a room. The boxes are empty until you put something in them. You can also place a sticker on the box, so that you'll know what's in it. Let's look at a programming example.

If you've got your project open from the previous section, click **File** from the menu bar at the top of Visual C#. From the File menu, click **Close Solution**. Start a new project by clicking **File** again, then **New Project**. From the New Project dialogue box, click on Windows Application. For the Name, type String Variables, as in the image below:



A project for creating an application with a Windows user interface

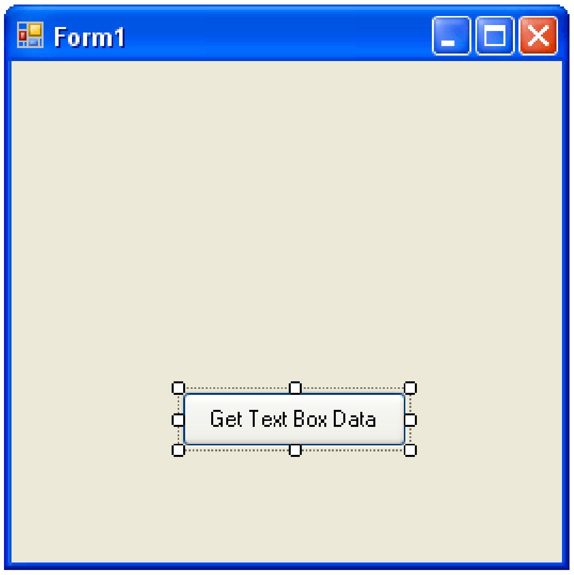
Name: String Variables

Click OK, and you'll see a new form appear. Add a button to the form, just like you did in the previous section. Click on the button to select it (it will have the white squares around it), and then look for the

Properties Window in the bottom right of Visual Studio. Set the following Properties for your new button:

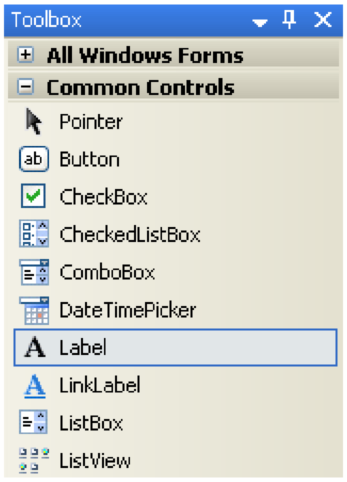
Name: btnStrings
Location: 90, 175
Size: 120, 30
Text: Get Text Box Data

Your form should then look like this:

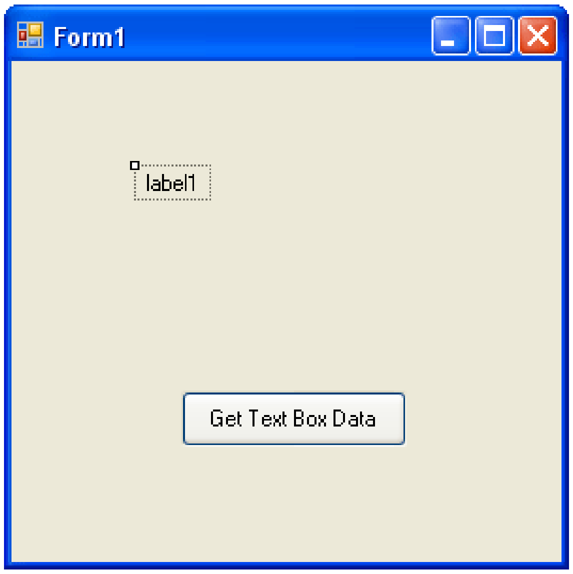


We can add two more controls to the form, a Label and a Text Box. When the button is clicked, we'll get the text from the text box and display whatever was entered in a message box.

A Label is just that: a means of letting your users know what something is, or what it is for. To add a Label to the form, move your mouse over to the Toolbox on the left. Click the **Label** item under **Common Controls**:



Now click once on your form. A new label will be added:

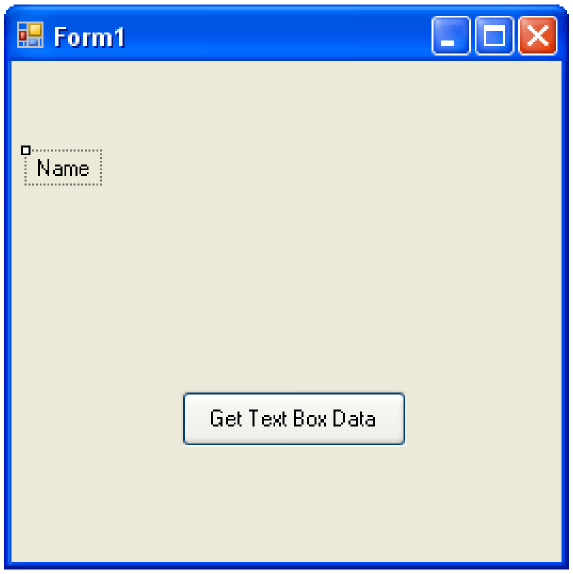


The Label has the default text of label1. When your label is selected, it will have just the one white square in the top left. When it is selected, the Properties Window will have changed. Notice that the properties for a label are very similar to the properties for a button - most of them are the same!

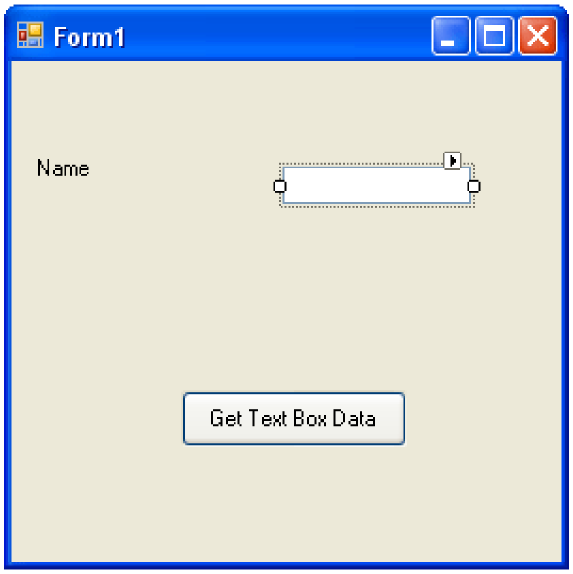
Change the following properties of your label, just like you did for the button:

Location: 10, 50
Text: Name

You don't really need to set a size, because Visual C# will automatically resize your label to fit your text. But your Form should look like this:

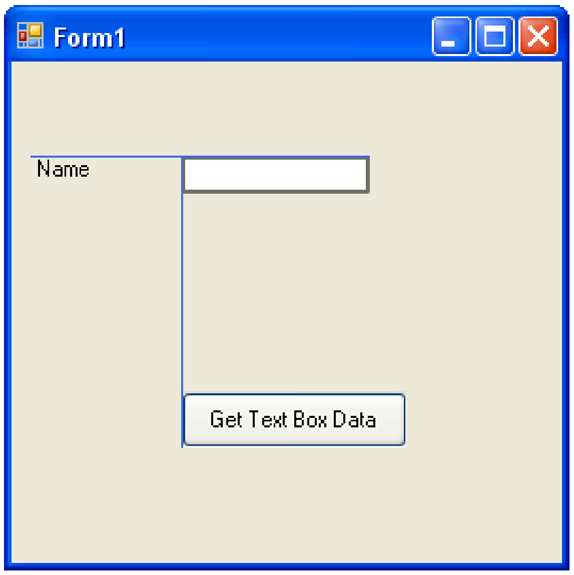


Move your mouse back over to the Toolbox. Click on the **TextBox** entry. Then click on your form. A new Text Box will be added, as in the following image:



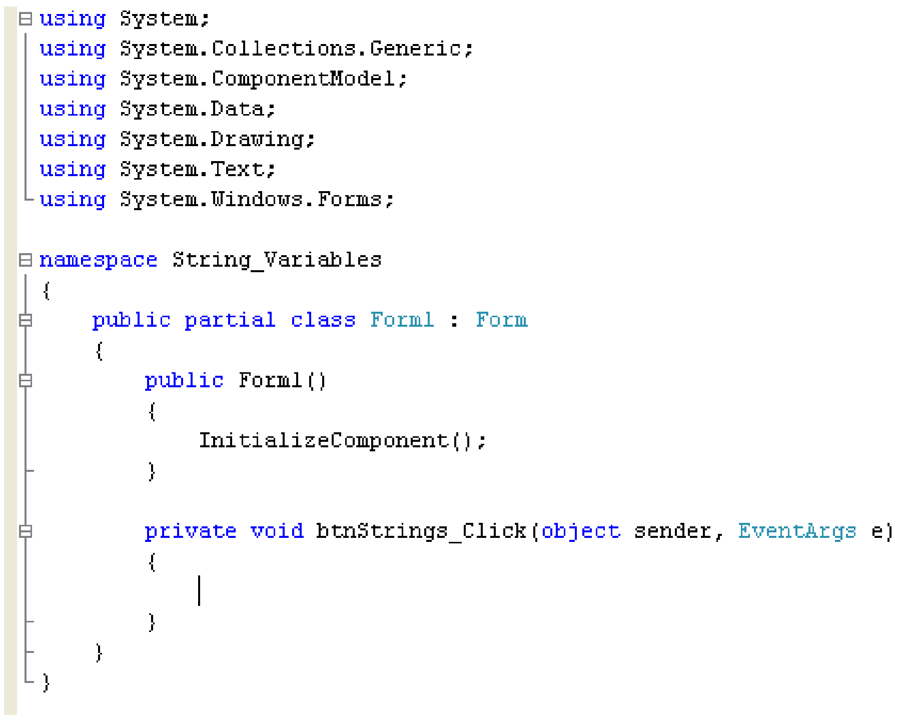
Instead of setting a location for your text box, simply click it with your left mouse button. Hold your left mouse button down, and the drag it just to the right of the Label.

Notice that when you drag your text box around, lines appear on the form. These are so that you can align your text box with other controls on the form. In the image below, we've aligned the text box with the left edge of the button and the top of the Label.



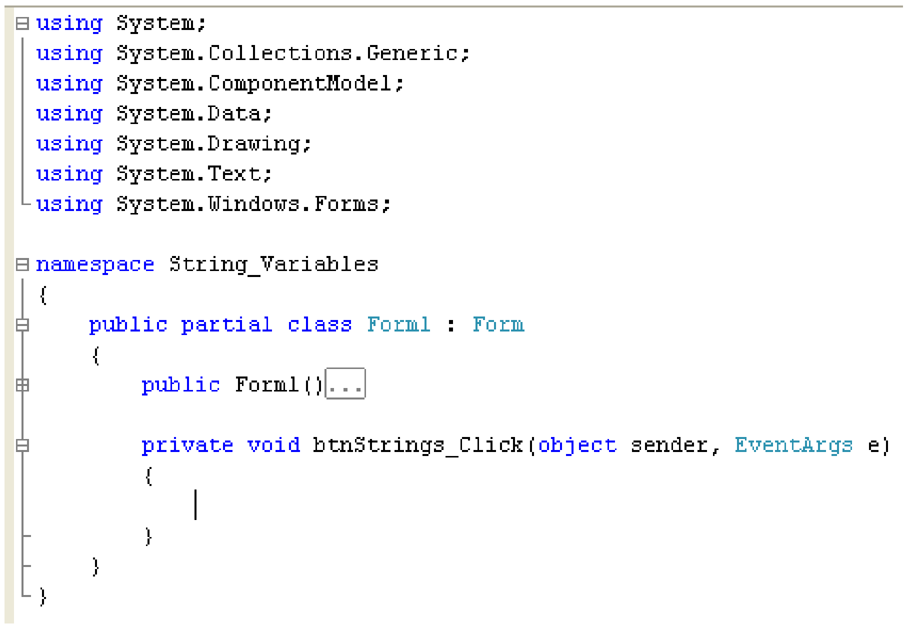
OK, time to add some code. Before you do, click **File > Save All** from the menu bar at the top of Visual C#. You can also run your programme to see what it looks like. Type some text in your text box, just to see if it works. Nothing will happen when you click your button, because we haven't written any code yet. Let's do that now. Click the red X on your form to halt the programme, and you'll be returned to Visual C#.

Double click your button to open up the coding window. Your cursor will be flashing inside of the curly brackets for the button code:



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

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

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

Notice all the minus symbols on the left hand side. You can click these, and it will hide code for you. Click the minus symbol next to public **Form1( )**. It will turn into a plus symbol, and the code for just this Method will be hidden:



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

namespace String_Variables
{
    public partial class Form1 : Form
    {
        public Form1()...

        private void btnStrings_Click(object sender, EventArgs e)
        {

        }
    }
}
```

Hiding code like this makes the rest of the coding window easier to read.

SELF ASSESSMENT EXERCISE

How would you access the coding window?

3.2 Setting up String Variables

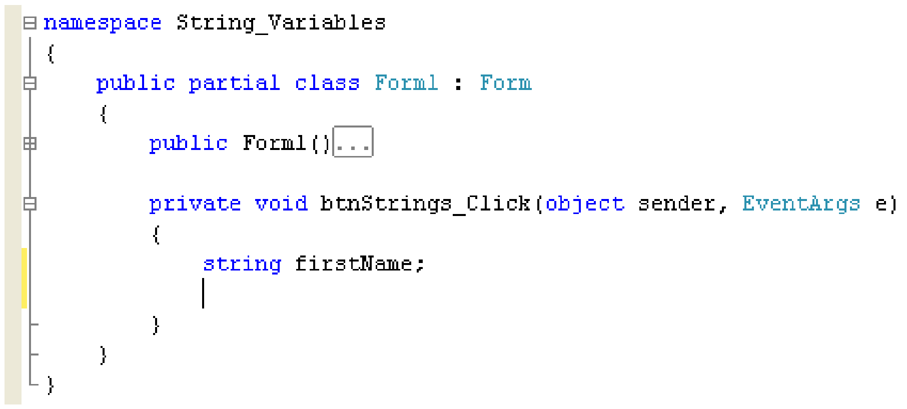
We're going to set up a string variable. To do this, you require two things:

- ❖ The Type of variable you want, and
- ❖ A name for your variable.

Click inside the two curly brackets of the button code, and add the following:

string **firstName**;

After the semi-colon, press the enter key on your keyboard to start a new line. Your coding window will then look like this:



```
namespace String_Variables
{
    public partial class Form1 : Form
    {
        public Form1()...

        private void btnStrings_Click(object sender, EventArgs e)
        {
            string firstName;

        }
    }
}
```

What you have done is to set up a variable called **firstName**. The Type of variable is a string. Note that the coding editor will turn the word "string" blue. Blue denotes the variable type - a string, in this case. (Other variable types are int, float, and double. These are all number variables that you'll meet shortly.)

After you have told C# which type of variable you want, you then need to come up with a name for your variable. This is like the sticker on an empty box. The empty box is the variable type. Think of these empty boxes as being of different sizes and different materials. A big, cardboard box is totally different from a small wooden one! But what you are really doing here is telling C# to set aside some memory, and that this storage area will hold strings of text. You give it a unique name so as to tell it apart from other items in memory. After all, would you be able to find the correct box, if they were all the same size, the same shape, the same colour, and had no stickers on them?

The name you pick for your variables, **firstName** in our case, can be almost anything you want - it's entirely up to you what you call them. But you should pick something that is descriptive, and gives you a clue as to what might be in your variable.

We say you can call your variables almost anything. But there are some rules, and some words that C# bags for itself. The words that C# reserves for itself are called Keywords. There are about 80 of these words, things like using, for, new, and public. If the name you have chosen for your variable turns blue in the coding window, then it's a reserved word, and you should pick something else.

3.2.1 Characters for Variables

The only characters that you can use in your variable names are letters, numbers, and the underscore character (_). And you must start the variable name with a letter, or underscore. You'll get an error message if you start your variable names with a number. So these are OK:

firstName
first_Name
firstName2

But these are not:

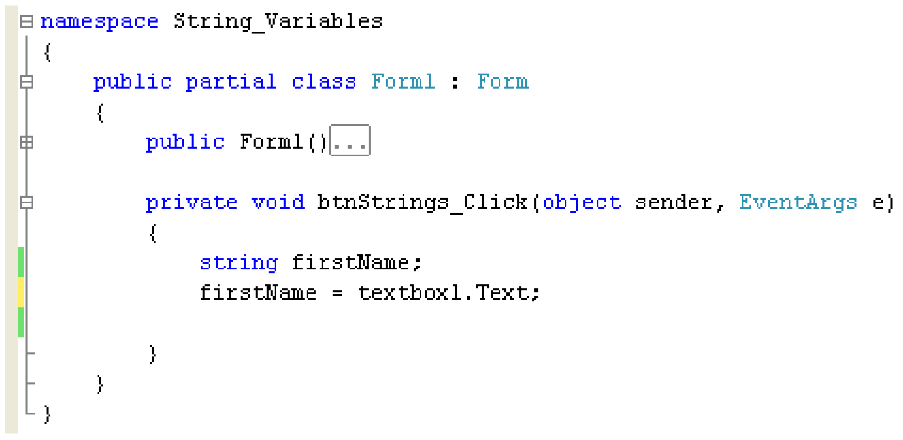
1firstName (Starts with a number)
first_Name& (Ends with an illegal character)
first Name (Two words, with a space in between)

Notice that all the variable names above start with a lowercase letter. Because we're using two words joined together, the second word starts with an uppercase letter. It's recommended that you use this format for your variables (called camelCase notation.) So firstName, and not Firstname.

After setting up your variable (telling C# to set aside some memory for you), and giving it a name, the next thing to do is to store something in the variable. Add the following line to your code (don't forget the semi-colon on the end):

firstName = textbox1.Text;

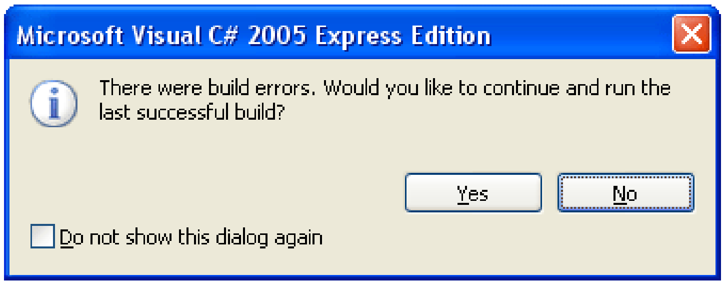
Your coding window will then look like this:



```
namespace String_Variables
{
    public partial class Form1 : Form
    {
        public Form1()...

        private void btnStrings_Click(object sender, EventArgs e)
        {
            string firstName;
            firstName = textbox1.Text;

        }
    }
}
```

To store something in a variable, the name of your variable goes on the left hand side of an **equals sign**. After an equals sign, you type what it is you want to store in the variable. For us, this is the **Text** from **textbox1**.

Except, there's a slight problem. Try to run your code. You should see an error message like this one:


Microsoft Visual C# 2005 Express Edition

There were build errors. Would you like to continue and run the last successful build?

Click **No**, and have a look at your code:

```
private void btnStrings_Click(object sender, EventArgs e)
{
    string firstName;
    firstName = textbox1.Text;

}
```

There is a blue wiggly line under textbox1. Hold your mouse over this and Visual Studio will tell you that:

The name 'textbox1' does not exist in the current context.

If you see an error like this, which is quite common, it means that Visual C# cannot find anything with the name you've just typed. So it thinks we don't have a textbox called textbox1. And we don't! It's called textBox1. We've typed a lowercase "b" when it should be an uppercase "B". So it's important to remember that C# is case sensitive. This variable name:

firstName

Is different to this variable name:

FirstName

The first one starts with a lowercase "f" and the second one starts with an uppercase "F".

Delete the lowercase "b" from your code and type an uppercase "B" instead. Run your programme again and you won't see the error message. Now stop your programme and return to the coding window. The blue wiggly line will have disappeared.

What have so far, then, is the following:

string **firstName;**
firstName = textBox1.Text;

The first line sets up the variable, and tells C# to set aside some memory that will hold a string of text. The name of this storage area will be **firstName**.

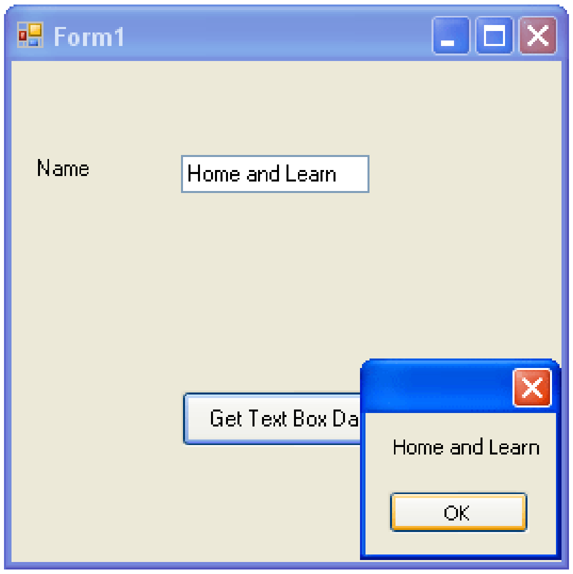
The second line is the one that actually stores something in the variable - the Text from a text box called **textBox1**.

Now that we have stored the text from the text box, we can do something with it. In our case, this will be to display it in a message box. Add this line to your code:

MessageBox.Show(firstName);

The MessageBox.Show( ) Method is one you've just used. In between the round brackets, you can either type text surrounded by double quotes, or you can type the name of a string variable. If you're typing the name of a variable, you leave the double quotes off. You can do this because C# knows what is in your variable (you have just told it on the second line of your code.)

Run your programme again. Type something in your text box, and then click the button. You should see the text you typed:



Home and Learn

Halt your programme and return to the coding window.

3.3 Assigning Text to a String Variable

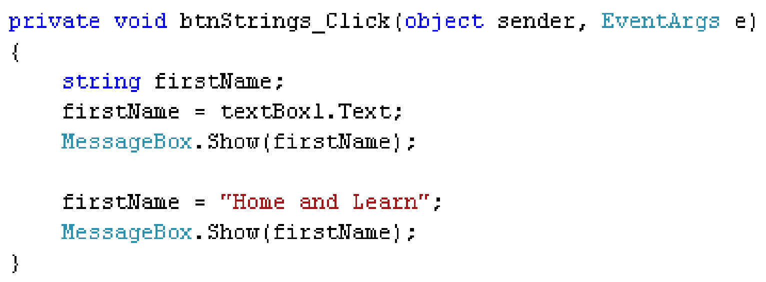
As well as assigning text from a text box to your variable, you can assign text like this:

firstName = "Home and Learn";

On the right hand side of the equals sign, we now have some direct text surrounded by double quotes. This then gets stored into the variable on the left hand side of the equals sign. To try this out, add the following two lines just below your MesageBox line:

firstName = "Home and Learn";
MessageBox.Show(firstName);

Your coding window will then look like this:



```
private void btnStrings_Click(object sender, EventArgs e)
{
    string firstName;
    firstName = textBox1.Text;
    MessageBox.Show(firstName);

    firstName = "Home and Learn";
    MessageBox.Show(firstName);
}
```

Run your programme again. Type something in the text box, your own first name. Then click the button. You should see two message boxes, one after the other. The first one will display your first name. But the second will display "Home and Learn".

We're using the same variable name, here: **firstName**. The first time we used it, we got the text directly from the text box. We then displayed it in the Message Box. With the two new lines, we're typing some text directly in the code, "Home and Learn", and then assigning that text to the **firstName** variable. We've then added a second MessageBox.Show( ) method to display whatever is in the variable.

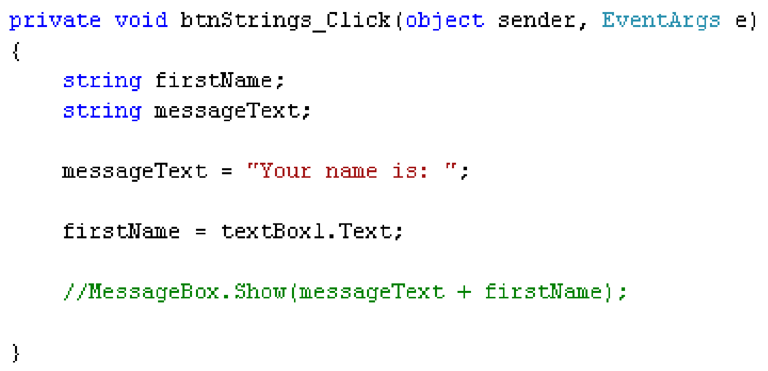
(If you want, you can change these colours. From the menu bar at the top, click **Tools > Options**. Under **Environment**, click **Fonts and Colors**.)

Time now to Build and Run your code!

3.4 C# Comments

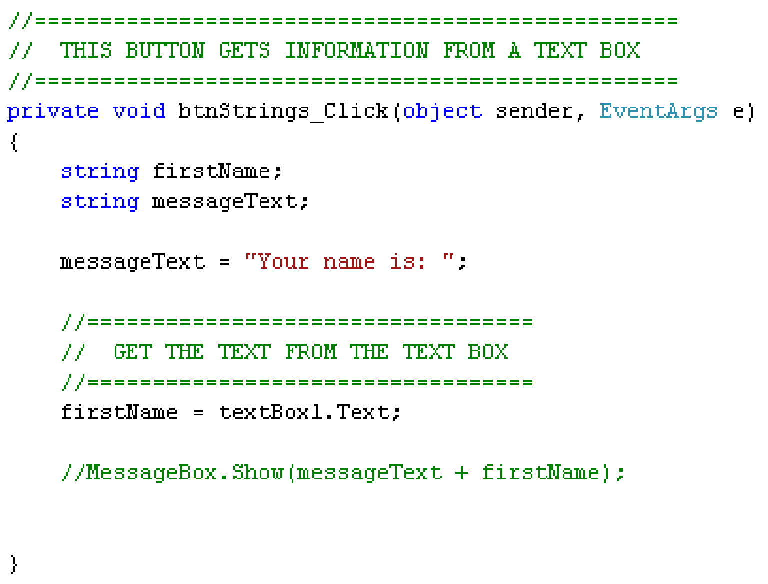
Comments in C# are incorporated by adding slashes. Thus to integrate a C# comment the following procedure is adopted:

1 Return to your coding window, and add two forward slashes to the start of your MessageBox.Show( ) line. The line should turn green, as in the following image:



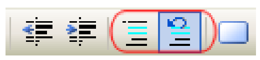
```
private void btnStrings_Click(object sender, EventArgs e)
{
    string firstName;
    string messageText;

    messageText = "Your name is: ";

    firstName = textBox1.Text;

    //MessageBox.Show(messageText + firstName);

}
```

The reason it turns green is that two forward slashes are the characters you use to add a comment. C# then ignores theses lines when running the programme. Comments are a very useful way to remind yourself what the programme does, or what a particular part of your code is for. Here's our coding window with some comments added:



```
//==================================================
//  THIS BUTTON GETS INFORMATION FROM A TEXT BOX
//==================================================
private void btnStrings_Click(object sender, EventArgs e)
{
    string firstName;
    string messageText;

    messageText = "Your name is: ";

    //===================================
    //  GET THE TEXT FROM THE TEXT BOX
    //===================================
    firstName = textBox1.Text;

    //MessageBox.Show(messageText + firstName);


}
```

You can also use the menu bar, or the toolbar, to add comments. Highlight any line of text in your code. From the menu bar at the top of Visual C#, select **Edit > Advanced > Comment Selection**. Two forward slashes will be added to the start of the line. You can quickly add or remove comments by using the toolbar. Locate the following icons on the toolbars at the top of Visual C#:

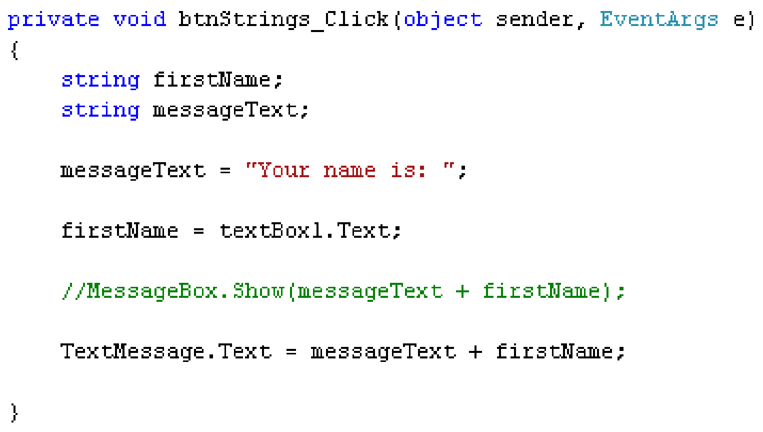


The comment icons are circled in red, in the image above. The first one adds a comment, and the second one removes a comment. (If you can't see the above icons anywhere on your toolbars, click **View** > **Toolbars** > **Text Editor**.)

Now that you have commented out the MessageBox line, it won't get executed when your code runs. Instead, add the following like to the end of your code:

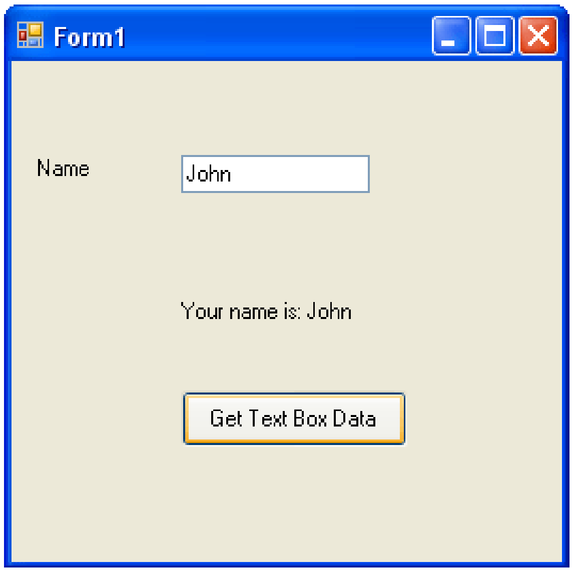
TextMessage.Text = messageText + firstName;

Your coding window should then look like this:



```
private void btnStrings_Click(object sender, EventArgs e)
{
    string firstName;
    string messageText;

    messageText = "Your name is: ";

    firstName = textBox1.Text;

    //MessageBox.Show(messageText + firstName);

    TextMessage.Text = messageText + firstName;

}
```

Run your programme again. Type your name in the text box, and then click your button. The message should now appear on your label, instead of in a Message Box:



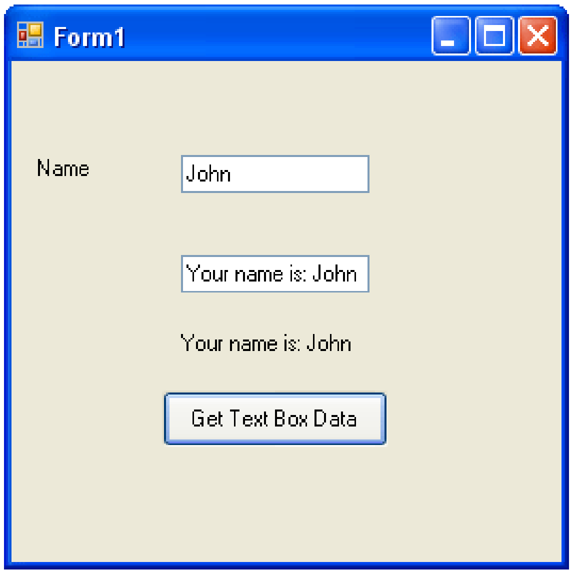
```
Your name is: John
```

The reason is does so is because you're now setting the Text property of the Label with code. Previously, you changed the Label's Text Property from the Properties Window. The name of our label is **TextMessage**. To the right of the equals sign, we have the same code that was in between the round brackets of the Show( ) method of the MessageBox.

OK, time for an exercise.

Exercise
Add a second text box to your form. Display your message in the text box as well as on the label. So if your name is John, your second text box should have: "Your name is: John" in it after the button is clicked.

When you complete this exercise, your form should look like this, when the button is clicked:



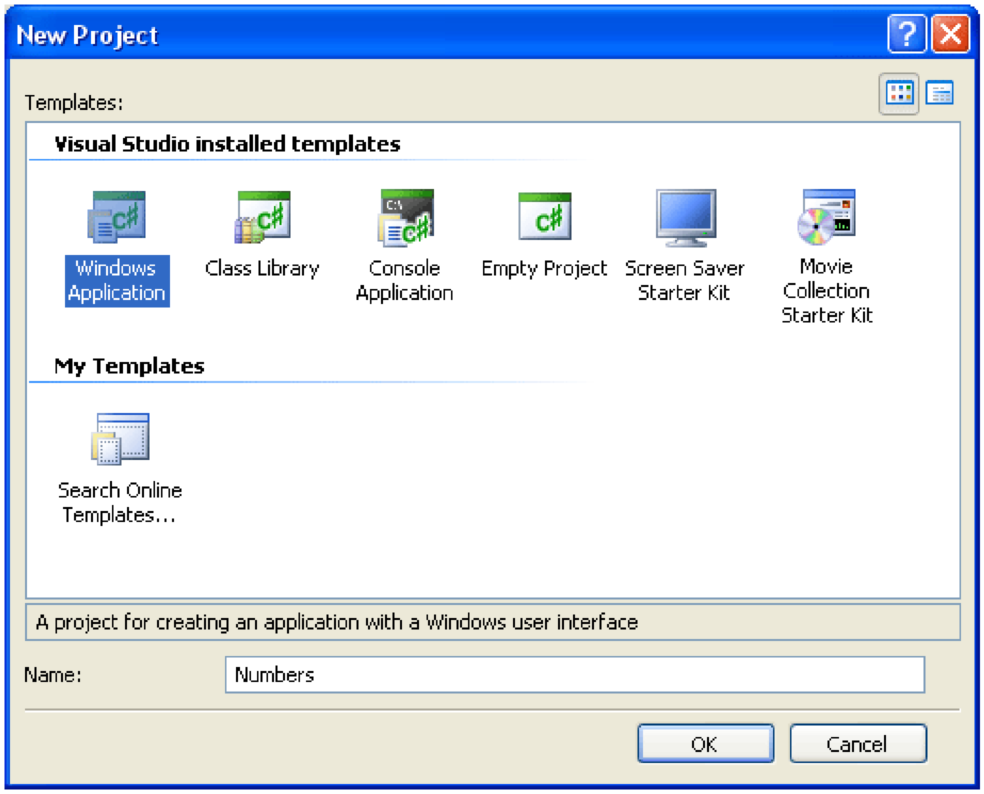
We're now going to move away from string variables and on to number variables. The same principles you've just learnt still apply, though:

- Set up a variable, and give it a name
- Store something in the variable
- Use code to manipulate what you have stored

3.5 C# Number Variables

As well as storing text in memory you can, of course, store numbers. There are a number of ways to store numbers, and the ones you'll learn about now are called Integer, Double and Float. First up, though, are Integer variables.

First, close any solution you have open by clicking **File > Close Solution** from the menu bar at the top of Visual Studio. Start a new project by clicking **File > New Project**. From the New Project dialogue box, select **Windows Application** from the available templates. Type a Name for your project. In the image below, we've chosen the name **Numbers**:



A project for creating an application with a Windows user interface

Name: Numbers

Click OK, and you'll have a new form to work with.

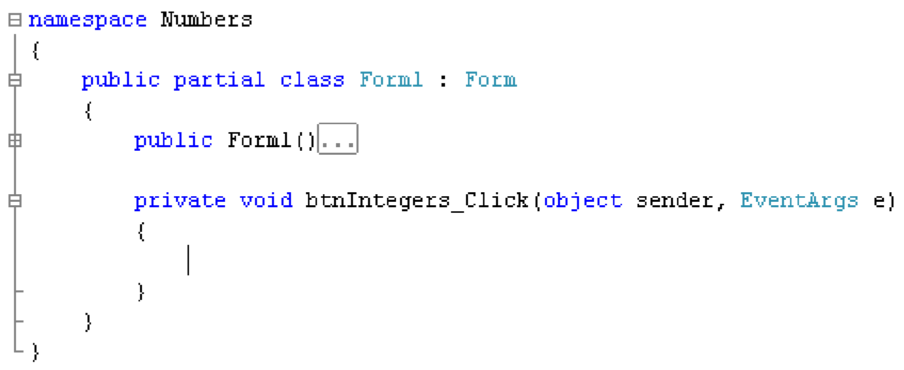
3.5.1 C# Integers

An integer is a whole number. It's the 6 of 6.5, for example. In programming, you'll work with integers a lot. But they are just variables that you store in memory and want to manipulate. You'll now see how to set up and use Integer variables.

Add a button to your form, and set the following properties for it in the Properties Window:

Name: btnIntegers
Text: Integers
Location: 110, 20

Now double click your button to get at the code:



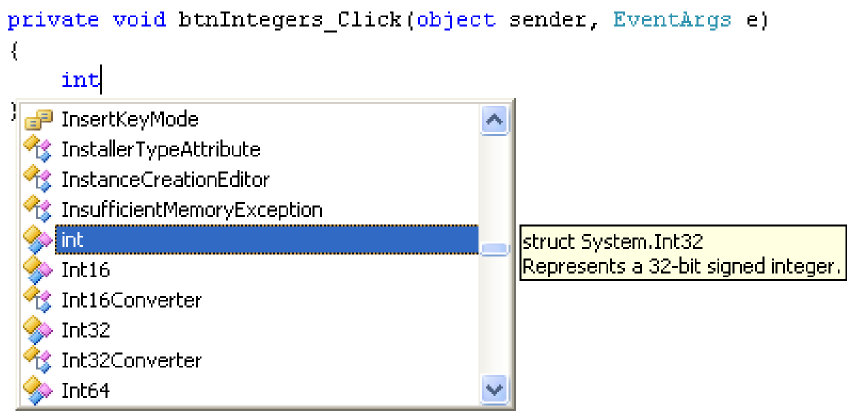
```
namespace Numbers
{
    public partial class Form1 : Form
    {
        public Form1()...

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

In the previous section, you saw that to set up a string variable you just did this:

string myText;

You set up an integer variable in the same way. Except, instead of typing the word string, you type the word **int** (short for integer).

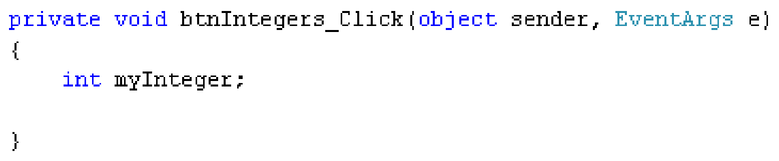
So, in between the curly brackets of your button code, type **int**. You should see the word turn blue, and the IntelliSense list appear:



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

struct System.Int32
Represents a 32-bit signed integer.

Either press the enter key on your keyboard, or just hit the spacebar. Then type a name for your new variable. Call it **myInteger**. Add the semi-colon at the end of your line of code, and hit the enter key. Your coding window will then look like this:



```
private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

}
```

Notice the text in the yellow box, in the image one up from the one above. It says:

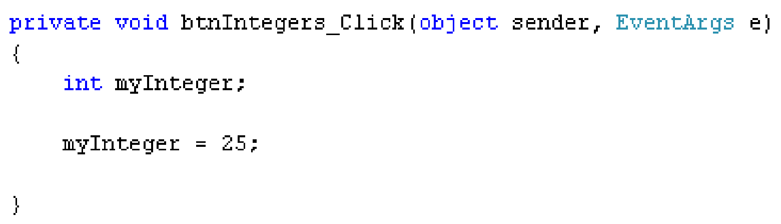
Represents a 32-bit signed integer

A signed integer is one that can have negative values, like -5, -6, etc. (The opposite, no negative numbers, is called an unsigned integer.) The 32-bit part is referring to the range of numbers that an integer can hold. The maximum value that you can store in an integer is: 2,147,483,648. The minimum value is the same, but with a minus sign on the front: -2,147,483,648.

To store an integer number in your variable, you do the same as you did for string: type the name of your variable, then an equals sign (=), then the number you want to store. So add this line to your code (don't forget the semi-colon on the end):

myInteger = 25;

Your coding window should look like this:



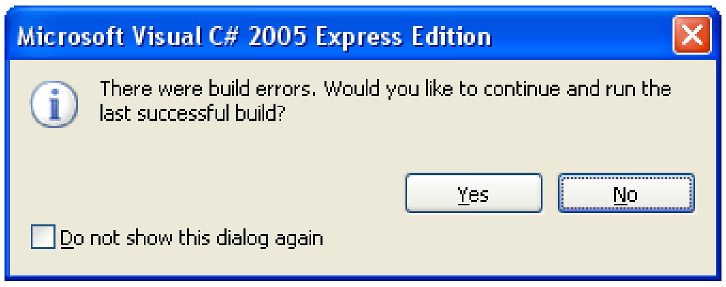
```
private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

    myInteger = 25;

}
```

So we've set up an integer variable called myInteger. On the second line, we're storing a value of 25 inside of the variable.

We'll use a message box to display the result when the button is clicked. So add this line of code for line three:

MessageBox.Show(**myInteger**);

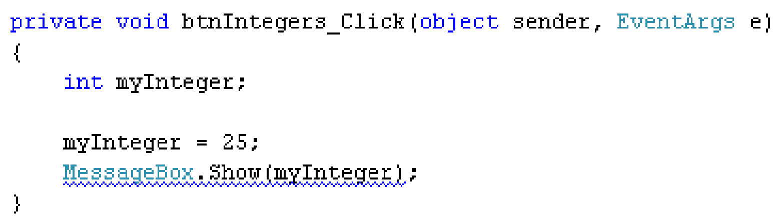
Now try to run your code. You'll get the following error message:



Microsoft Visual C# 2005 Express Edition

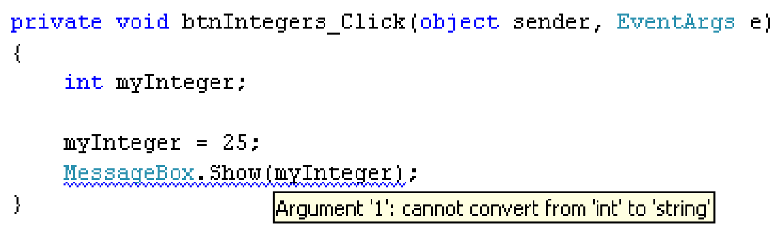
There were build errors. Would you like to continue and run the last successful build?

You should see a blue wiggly line under your MessageBox code:



```
private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

    myInteger = 25;
    MessageBox.Show(myInteger);
}
```

Hold your mouse over **myInteger**, between the round brackets of **Show**( ). You should see the following yellow box:

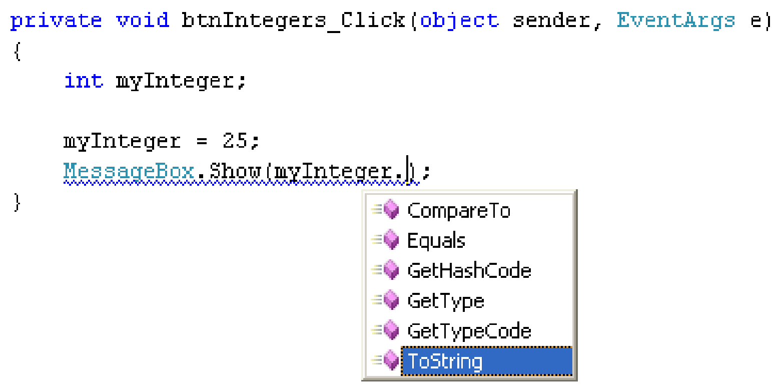


```
private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

    myInteger = 25;
    MessageBox.Show(myInteger);
}
```

Argument '1': cannot convert from 'int' to 'string'

The error is: "Cannot convert from int to string". The reason you get this error is because **myInteger** holds a number. But the MessageBox only displays text. C# does not convert the number to text for you. It doesn't do this because C# is a programming language known as "strongly typed". What this means is that you have to declare the type of variable you are using (string, integer, double). C# will then check to make sure that there are no numbers trying to pass themselves off as strings, or any text trying to pass itself off as a number. In our code above, we're trying to pass **myInteger** off as a string. And C# has spotted it!

What you have to do is to convert one type of variable to another. You can convert a number into a string quite easily. Type a full stop (period) after the "r" of myInteger. You'll see the IntelliSense list appear:



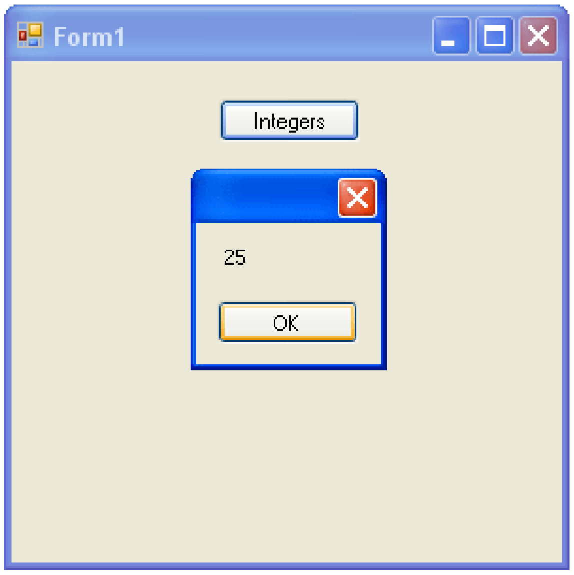
```
private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

    myInteger = 25;
    MessageBox.Show(myInteger.);
}
```

Select **ToString** from the list. Because **ToString** is a method, you need to type a pair of round brackets after the "g" of ToString. Your code will then look like this (we've highlighted the new addition):

```
private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

    myInteger = 25;
    MessageBox.Show(myInteger.ToString());
}
```

The ToString method, as its name suggests, converts something to a string of text. The thing we are converting is an integer.

Start your programme again. Because you've converted an integer to a string, you should find that it runs OK now. Click your button and you should see the message box appear:



In the next lesson, we'll take a look at double variables, and float variables.

When you complete this exercise, your form should look like this, when the button is clicked:

3.6 Double and Float Variables

Integers, as was mentioned, are whole numbers. They can't store the point something, like **.7, .42, and .007**. If you need to store numbers that are not whole numbers, you need a different type of variable. You can use the **double** type, or the **float** type. You set these types of variables up in exactly the same way: instead of using the word int, you type double, or float. Like this:

float myFloat;
double myDouble;

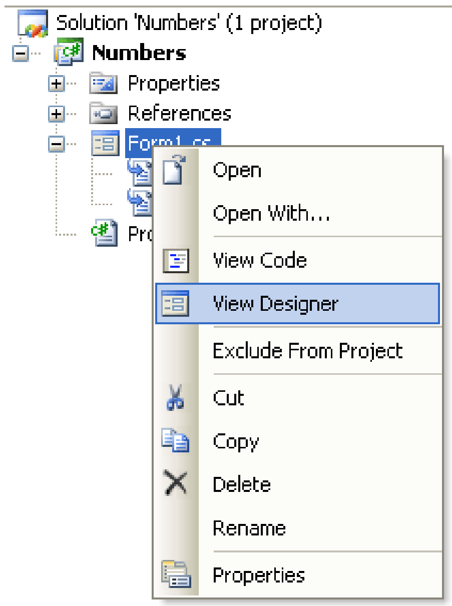
(Float is short for "floating point", and just means a number with a point something on the end.)

The difference between the two is in the size of the numbers that they can hold. For float, you can have up to 7 digits in your number. For doubles, you can have up to 16 digits. To be more precise, here's the official size:

float: 1.5 × 10-45 to 3.4 × 1038
double: 5.0 × 10-324 to 1.7 × 10308

Float is a 32-bit number and double is a 64-bit number.

To get some practice using floats and doubles, return to your form. If you can't see the **Form1.cs [Design]** tab at the top, right click Form1.cs in the Solution Explorer on the right hand side. (If you can't see the Solution Explorer, click View > Solution Explorer from the menu bar at the top.)



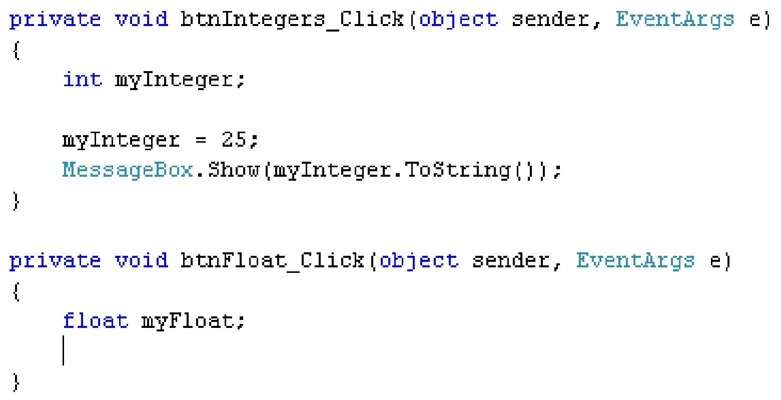
Add a new button to your form. Set the following properties for it in the Properties Window:

Name btnFloat
Location: 110, 75
Text: Float

Double click your new button, and add the following line to the button code:

float myFloat;

Your coding window will then look like this:



```
private void btnIntegers_Click(object sender, EventArgs e)
{
    int myInteger;

    myInteger = 25;
    MessageBox.Show(myInteger.ToString());
}

private void btnFloat_Click(object sender, EventArgs e)
{
    float myFloat;
    |
}
```

To store something inside of your new variable, add the following line:

myFloat = 0.42F;

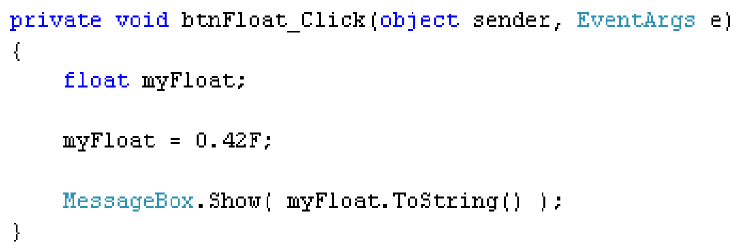
The capital letter F on the end means Float. You can leave it off, but C# then treats it like a double. Because you've set the variable up as a float, you'll get errors if you try to assign a double to a float variable.

Add a third line of code to display your floating point number in a message box:

MessageBox.Show(**myFloat.ToString( )**);

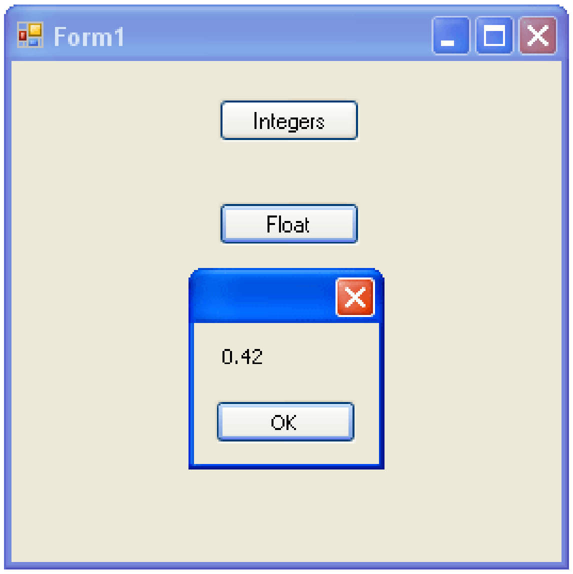
Again, we have to use ToString( ) in order to convert the number to a string of text, so that the message box can display it.

But your coding window should look like ours below:



```
private void btnFloat_Click(object sender, EventArgs e)
{
    float myFloat;

    myFloat = 0.42F;

    MessageBox.Show( myFloat.ToString() );
}
```

Run your programme and click your Float button. You should see a form like this:



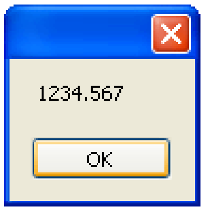
```
0.42
```

Halt the programme and return to your coding window. Now delete the capital letter F from 0.42. The line will then be:

myFloat = 0.42;

Try to run your programme again. You'll get an error message, and a blue wiggly line under your code. Because you've missed the F out, C# has defaulted to using a double value for your number. A float variable can't hold a double value, confirming that C# is a strongly typed language. (The opposite is a weakly typed language. PHP, and JavaScript are examples of weakly typed languages - you can store any kind of values in the variables you set up.)

Another thing to be careful of when using float variables is rounding up or down. As an example, change the number from 0.42F to 1234.567F. Now run your programme, and click your float button. The message box will be this:

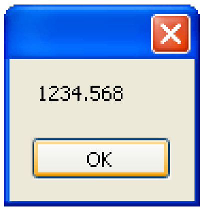


```
1234.567
```

Halt the programme and return to your code. Now add an 8 before the F and after the 7, so that your line of code reads:

myFloat = 1234.5678F;

Now run your programme again. When you click the button, your message box will be this:



```
1234.568
```

It's missed the 7 out! The reason for this is that float variables can only hold 7 numbers in total. If there's more than this, C# will round up or down. A number that ends in 5 or more will be rounded up. A number ends in 5 or less will be rounded down:

1234.5678 (eight numbers ending in 8 - round up)
1234.5674 (eight numbers ending in 4 - round down)

The number of digits that a variable can hold is known as **precision**. For float variable, C# is precise to 7 digits: anything more and the number is rounded off.

4.0 CONCLUSION

Note that, some common variables in C# are Strings, Numbers, Integers, Double and Float variables. Comments are incorporated in C# by adding slashes.

5.0 SUMMARY

This unit provided us with information on common variables and comments in C#. We hope you found this unit remarkable and simple. Let us attempt the question below.

6.0 TUTOR MARKED ASSIGNMENT

- List the common variables in C#
- Outline the procedure for assigning texts to strings in C#
- State how comments are incorporated in C#

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Kay, Alan. *The Early History of Smalltalk.* http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.

MODULE 2 C# CLASS

Unit 1 Preamble to C# Class
Unit 2 Operation on C# Class
Unit 3 Adding Methods
Unit 4 C# Class Properties

UNIT 1 PREAMBLE TO C# CLASS

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
  3.1 Class Fundamentals
    3.1.1 Class Description
    3.1.2 General Form of a Class
    3.1.3 General Form of an Instance Variable
    3.1.4 General Form of a Dot Operator
    3.1.5 Class Definition Syntax
    3.1.6 Simple C# Class
  3.2 Creating a C# Class
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

Classes are at the heart of every object-oriented language. With classes, as with a lot of the language's features, C# borrows a little from C++ and Java and adds some ingenuity to create elegant solutions to old problems. In this unit, we'll equally give a brief description of classes in C#.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Describe a class
- Give the general form of a class
- Declare an instance variable
- State the general form of a dot operator
- State the class definition syntax
- Outline the procedure for creating a simple C# class

3.0 MAIN CONTENT

3.2 Class Fundamentals

The class is the basic building block of code when creating object-oriented software. It describes in abstract all of the characteristics and behaviour of a type of object. Once *instantiated*, an object is generated that has all of the methods, properties and other behaviour defined within the class.

Consequently a class should not be confused with an object. The class is the abstract concept for an object that is created at design-time by the programmer. While objects based upon the class are the concrete instances of the class that occur at run-time. For example, the *Car* class will define that all cars have a make, model and colour. Only at run-time will an object be instantiated as a Red.

3.1.1 Class Description

A class is can simply be described as a template that defines the form of an object. It specifies both the data and the code that will operate on that data. Methods and variables that constitute a class are called members of the class.

3.1.2 General Form of a Class

A class is created by use of the keyword **class**.
The general form of a class definition that contains only instance variables and

methods is given as:

```
class classname {
   // declare instance variables
   access type var1;
   access type var2;
   // ...
```

```
    access type varN;

    // declare methods
    access ret-type method1(parameters) {
        // body of method
    }
    access ret-type method2(parameters) {
        // body of method
    }
    // ...
    access ret-type methodN(parameters) {
        // body of method
    }
}
```

3.1.3 Declaring Instance Variable

The general form for declaring an instance variable is shown here:

access type var-name;

An instance variable consists of the following:

1. access which specifies the access,
2. type which specifies the type of variable, and
3. var-name which is the variable's name.

In order to access these variables, you will use the dot (.) operator. The dot operator links the name of an object with the name of a member.

3.1.4 General Form of a Dot Operator

The general form of the dot operator is shown here:

Object.member

The dot operator is used to access both instance variables and methods.

3.1.5 Class Definition Syntax

Class definition syntax is given as:

```
class MyClass
{
   int simpleValue = 0;
}
```

3.1.6 Simple C# Class

A simple C# class is given as:

```
using System;

class MainClass {
   MainClass() {
      Console.WriteLine("MainClass Constructor Called");
   }

   ~MainClass() {
      Console.WriteLine("MainClass Destructor Called");
   }
   void PrintAMessage(string msg) {
      Console.WriteLine("PrintAMessage: {0}", msg);
   }
   void Dispose() {
      GC.SuppressFinalize(this);
   }

   static void Main() {
      Console.WriteLine("Top of function main");
      MainClass app = new MainClass();
      app.PrintAMessage("Hello from class");
      Console.WriteLine("Bottom of function main");
      app.Dispose();
   }
}
```

SELF ASSESSMENT EXERCISE

State the class definition syntax

3.2 Creating a Class

Now that we have had an overview of classes, we would consider the syntax for the creation of a new class. The basic syntax for the creation of a new class is very simple. The keyword '**class**' followed by the name of the new class is simply added to the program. This is then followed by a code block surrounded by brace characters {} to which the class' code will be added.

class *class-name* {}

4.0 CONCLUSION

We discovered that the class is the basic building block of code when creating object-oriented software. It describes in abstract all of the characteristics and behaviour of a type of object. We equally saw the syntax for a simple class creation as well as the general form of a dot operator.

5.0 SUMMARY

In this unit, we learnt about the general form of a class, instance variable declaration general form of a dot operator as well as the procedure for creating a simple C# class. Be assured that the facts gathered from this unit will be valuable for building C# applications. OK! Let us attempt the questions below.

6.0 TUTOR MARKED ASSIGNMENT

- Give a brief description of a class
- State the general form of a dot operator

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Kay, Alan. *The Early History of Smalltalk.* http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
12. Martin, A and Luca, C. (2005). *A Theory of Objects*.
13. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
14. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
15. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
16. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
17. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
18. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
19. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 2 OPERATIONS ON A CLASS

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
  3.1 Adding a C# Class
    3.1.1 Adding a C# Class using a Solution Explorer
    3.1.2 Adding a C# Class in Class View
  3.2 Instantiating a Class
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

C# Classes can be added by two main methods, these are by using:
(1) The Solution Explorer or (2) The Class View. This unit sheds more light on this.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Identify 2 main methods of adding C# Classes
- Enumerate the steps involved in adding a C# Class using Solution Explorer
- Outline the procedure for adding a C# Class in Class View
- Describe how to instantiate a Class

3.0 MAIN CONTENT

3.1 Adding a C# Class

We can add C# Classes to applications in two main ways, these are by:

a. Adding a C# Class using Solution Explorer
b. Adding a C# Class in Class View

We would look at the procedure for accomplishing these tasks in the proceeding units.

3.1.1 Adding a C# Class Using Solution Explorer

You can build a generic class that is automatically declared under the current default namespace. For example, classes added to a project named WindowsApplication1 are added to the namespace WindowsApplication1.

Adding a class in Solution Explorer

1. In Solution Explorer, right-click the project name and click **Add**, and then click **Add Class**.

   The **Add New Item** dialog box appears with the **C# Class** icon already selected.

2. In the dialog box, enter a class name in the **Name** field and click **Open**.

   The new class is added to your project.

3.1.2 Adding a C# Class Using Class View

When you add a class from Class View, you have maximum control defining elements while using C# Class Wizard.

To add a class in Class View

- In Class View, right-click the project name and click **Add**
- Then click **Add Class**
- The C# Class Wizard appears

SELF ASSESSMENT EXERCISE

Outline the procedure for adding a C# Class using Class View

3.2 Instantiating a Class

Although we have not explicitly added any functionality to the class, it can now be instantiated to create objects. These objects will have the standard behaviour of all classes. To demonstrate this, return to the program code file containing the Main method. In this method we will create a new vehicle object and run its *ToString* method to see the results. As we have not yet defined how ToString should work, this will simply show the fully qualified name.

```
static void Main(string[] args)
{
   Vehicle car = new Vehicle();
   Console.WriteLine(car.ToString()); // Outputs "ClassTest.Vehicle"
}
```

NB: The prefix of ClassTest is simply the name of the namespace of the Vehicle class.

4.0 CONCLUSION

Classes are added in C# either by employing the class view or solution explorer.

5.0 SUMMARY

In summary, this unit looked at two main methods of adding C# Classes. We equally identified the procedure for instantiating Classes. We can now attempt the questions below.

6.0 TUTOR MARKED ASSIGNMENT

Outline the procedure for accomplishing the following tasks:

1. Adding a C# class in solution explorer
2. Adding a C# class in class view
3. Instantiating a C# class

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Kay, Alan. *The Early History of Smalltalk.* http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
12. Martin, A and Luca, C. (2005). *A Theory of Objects*.
13. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.

14. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
15. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
16. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
17. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
18. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
19. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 3 C# METHODS

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
3.1 Methods
3.1.1 Declaring Methods
3.1.2 Calling a Method
3.2 Method Parameters
3.3 Creating a Reference Type
3.4 Return Values
3.5 Public Methods
3.6 Private Methods
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

In C# every executed instruction is done in the context of a ***method***. We'll be considering methods, how they are called and declared as well the two common methods.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Describe a Method
- Identify how Methods are declared
- Explain the concept of Method Parameters
- State the syntax for declaring a Reference type

3.0 MAIN CONTENT

3.1 Methods

A ***method*** is a code block containing a series of statements. In C#, every executed instruction is done in the context of a *method*.

3.1.1 Declaring Methods

Methods are declared within a class or struct by specifying the access level, the return value, the name of the method, and any method parameters. Method parameters are surrounded by parentheses, and separated by commas. Empty parentheses indicate that the method requires no parameters. This class contains three methods:

```
class Motorcycle
{
   public void StartEngine() { }
   public void AddGas(int gallons) { }
   public int Drive(int miles, int speed) { return 0; }
}
```

3.1.2 Calling a Method

Calling a method on an object is similar to accessing a field. After the object name; add a period, the name of the method, and parentheses. Arguments are listed within the parentheses, and separated by commas. The methods of the Motorcycle class can therefore be called like this:

```
Motorcycle moto = new Motorcycle();

moto.StartEngine();
moto.AddGas(15);
moto.Drive(5, 20);
```

SELF ASSESSMENT EXERCISE

How are Methods declared?

3.2 Method Parameters

Passing arguments to a method is simply a matter of providing them in the parentheses when calling a method. To the method being called, the incoming arguments are called *parameters*.

The parameters a method receives are also provided in a set of parentheses, but the type and a name for each parameter must be specified. The name does not have to be the same as the argument. For example:

```
public static void PassesInteger()
{
   int fortyFour = 44;
   TakesInteger(fortyFour);
}
static void TakesInteger(int i)
{
   i = 33;
}
```

Here a method called PassesInteger passes an argument to a method called TakesInteger. Within PassesInteger, the argument is named fortyFour, but in TakeInteger, this is a parameter named i. This parameter exists only within the TakesInteger method. Any number of other variables can be named i, and they can be of any type, so long as they are not parameters or variables declared inside this method.

Notice that TakesInteger assigns a new value to the provided argument. One might expect this change to be reflected in the PassesInteger method once TakeInteger returns, but in fact the value in the variable fortyFour remains unchanged. This is because **int** is *a value type*. By default, when a value type is passed to a method, a copy is passed instead of the object itself. Because they are copies, any changes made to the parameter have no effect within the calling method. Value types get their name from the fact that a copy of the object is passed instead of the object itself. The value is passed, but not the same object.

This differs from *reference types*, which are passed by reference. When an object based on a reference type is passed to a method, no copy of the object is made. Instead, a reference to the object being used as a method argument is made and passed. Changes made through this reference will therefore be reflected in the calling method.

3.3 Creating a Reference Type

A reference type is created with the **class** keyword, like this:

```
public class SampleRefType
{
   public int value;
}
```

Now, if an object based on this type is passed to a method, it will be passed by reference. For example:

```
public static void TestRefType()
{
   SampleRefType rt = new SampleRefType();
   rt.value = 44; ModifyObject(rt);
   System.Console.WriteLine(rt.value);
}
static void ModifyObject(SampleRefType obj)
{
   obj.value = 33;
}
```

This example essentially does the same thing as the previous example. But, because a reference type is used, the modification made by ModifyObject is made to the object created in the TestRefType method. The TestRefType method will therefore display the value 33.

3.4 Return Values

Methods can return a value to the caller. If the return type, the type listed before the method name, is not **void**, then the method can return the value using the **return** keyword. A statement with the keyword return followed by a value that matches the return type will return that value to the method caller. The **return** keyword also stops the execution of the method. If the return type is **void**, a **return** statement with no value is still useful to stop the execution of the method. Without the **return** keyword, the method will stop executing when it reaches the end of the code block. Methods with a non-void return type are required to use the **return** keyword to return a value. For example, these two methods use the **return** keyword to return integers:

```
class SimpleMath
{
   public int AddTwoNumbers(int number1, int number2)
   {
      return number1 + number2;
   }

   public int SquareANumber(int number)
   {
      return number * number;
   }
}
```

To use a value returned from a method, the calling method can use the method call itself anywhere a value of the same type would suffice. You can also assign the return value to a variable. For example, the following two code examples accomplish the same goal:

```
int result = obj.AddTwoNumbers(1, 2);
obj.SquareANumber(result);

obj.SquareANumber(obj.AddTwoNumbers(1, 2));
```

Using an intermediate variable, in this case, result, to store a value is optional. It may help the readability of the code, or it may be necessary if the value is going to be used more than once.

3.5 Public Methods

Public methods are part of the class' public interface. Essentially, they are the methods that can be called by other objects.

The syntax for creating methods described previously must be modified slightly to make the methods visible to external objects. To achieve this, the *public* keyword is used as a prefix. The following code added to the vehicle class provides a new method for pressing a vehicle's horn. Make sure that you add the code within the class' code block.

```
public void PressHorn()
{
   Console.WriteLine("Toot toot!");
}
```

To use the new method, change the code within the Main method as follows:

```
static void Main(string[] args)
{
   Vehicle car = new Vehicle();
   car.PressHorn();                // Outputs "Toot toot!"
}
```

3.6 Private Methods

To provide for encapsulation, where the internal functionality of the class is hidden, some methods will be defined as ***Private***. Methods with a private *protection level* are completely invisible to external classes. This makes it safe for the code to be modified to change functionality, improve performance, etc. without the need to update classes that use the public interface. To define a method as private, the *private* keyword can be used as a prefix to the method. Alternatively, using no prefix at all implies that the method is private by default.

The following method of the car class is a part of the internal implementation not the public interface so is defined as being private.

```
private void MonitorOilTemperature()
{
   // Internal oil temperature monitoring code...;
}
```

To demonstrate that this method is unavailable to external classes, try the following code in the Main method of the program. When you attempt to compile or execute the program, an error occurs indicating that the MonitorOilTemperature method cannot be called due to its protection level.

```
static void Main(string[] args)
{
   Vehicle car = new Vehicle();
   car.MonitorOilTemperature();
```

4.0 CONCLUSION

In conclusion, ***Methods*** are codes of blocks containing series of statements. They are declared within a class or struct by specifying the access level, the return value, the name of the method, and any method parameters. Methods can either be ***Public*** or ***Private.***

5.0 SUMMARY

In this unit, we described methods and identified the syntax for declaring and calling methods. We equally considered the concept of method parameters and reference types. Hope you grasped the key points. Now, let us attempt the questions below.

6.0 TUTOR MARKED ASSIGNMENT

- Explain how methods are declared
- State the syntax for declaring a Reference Type

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.

8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
12. Martin, A and Luca, C. (2005). *A Theory of Objects*.
13. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
14. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
15. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
16. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
17. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
18. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
19. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 4 C# CLASS PROPERTIES

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
  3.1 Adding State
  3.2 Defining a Class-Scoped Variable
  3.3 Accessing Properties of Instantiated Objects
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

This unit expands upon the previous creation of simple classes by introducing class properties. Properties of a class allow instantiated objects to have a state with each object controlling its own data. Enjoy your studies.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Define a class-scoped variable
- State the syntax for adding two integer variables to the Rectangle class to hold the height and width
- Explain how to access properties of Instantiated Objects

3.0 MAIN CONTENT

3.1 Adding State

In the previous unit, you learnt about the basic syntax and process of creating a class that has methods as a part of its public interface. This allows the generation of simple classes with behaviour but without *state*.

However, the **state** of an object is the property which describes its individual data or unique configuration. In the example of a car object, methods allow us to accelerate or decelerate but the state, described in *properties*, allows us to determine the actual speed and to describe one car object as red and another as blue. This information creates a real differentiation between the two objects.

SELF ASSESSMENT EXERCISE

Which property describes the unique configuration of an object?

____

____

____

3.2 Defining a Class-Scoped Variable

In many cases, the information that is made public via a property is held directly within the object as a variable. This variable has a scope that makes it visible to the entire class. This is not always the case however, as the information may be held in a database or other external source or may be calculated rather than stored. In the class example in this unit, two of the property values will be held in variables and two will be calculated.

To define a class-scoped variable, the declaration is made within the class' code block but outside of any methods or properties. Although not required, it is useful to precede the variable declaration with the private keyword to make the code clear and easy to read. Using this syntax, we can add two integer variables to the Rectangle class to hold the height and width.

```
class Rectangle
{
    private int _width;
    private int _height;
}
```

NB: The use of lower camel case and an underscore (_) prefix is one naming standard for class-level variables. It is a useful, though not essential, convention.

3.3 Accessing Properties of Instantiated Objects

Properties of instantiated objects are accessed using the object name followed by the member access operator (.) and the property name. The property can be read from and written to using similar syntax as for a standard variable. To demonstrate this, consider the following example code, added to the Main method of the program. It creates two objects based upon the Rectangle class and assigns and reads their properties individually. When trying to assign an invalid height to a rectangle, an exception is thrown by the validation code.

```
static void Main(string[] args)
{
    Rectangle rect = new Rectangle();
    rect.Width = 50;
    rect.Height = 25;

    Rectangle square = new Rectangle();
    square.Height = square.Width = 40;

    Console.WriteLine(rect.Height);         // Outputs "25"
    Console.WriteLine(square.Width);        // Outputs "40"

    rect.Height = 25;                       // Throws the validation exception.
}
```

NB: The sample code demonstrates both encapsulation and state. The state of the two rectangles is held internally and independently and the implementation details of the properties are hidden from the program. If, in the future, we need to move the data from the private variables into an XML file or database, the Main method will not need to be changed.

4.0 CONCLUSION

To wrap up, recall that the **state** of an object is the property which describes its individual data or unique configuration. A class-scoped variable is defined by making a declaration within the class' code block but outside of any methods or properties. Properties of instantiated objects are accessed using the object name followed by the member access operator (.) and the property name.

5.0 SUMMARY

This unit provided an overview of class properties, illustrating how to define class-scoped variables and access properties of instantiated objects. However, to assess your level of assimilation , you would need to attempt the questions below.

6.0 TUTOR MARKED ASSIGNMENT

- Define a class-scoped variable
- Describe how to access properties of instantiated objects

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.

8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages,* Cambridge University Press, 2003, p.278
11. Kay, Alan. *The Early History of Smalltalk.* http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
12. Martin, A and Luca, C. (2005). *A Theory of Objects*.
13. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
14. Michael Lee Scott (2006). *Programming language pragmatics,* (2nd Edition) p. 470
15. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
16. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
17. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
18. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
19. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

MODULE 3 C# CONSTRUCTORS AND DESTRUCTORS

Unit 1 Constructors
Unit 2 Default Constructor
Unit 3 Destructors
Unit 4 Garbage Collection

UNIT 1 CONSTRUCTORS

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
 3.1 Significance of Constructors
 3.2 What is a Constructor?
 3.3 Creating a New Class
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

This unit examines constructors. These special methods allow objects to be initialised on instantiation and to perform final actions before they are removed from memory. Enjoy your studies.

2.0 OBJECTIVES

What you would study in this unit, would enable you:

- Identify the main significance of constructors
- Give a brief description of constructors
- Create a new class

3.0 MAIN CONTENT

3.1 Significance of Constructors

To set properties in Classes, an object is instantiated and the property values are assigned individually. This gives the desired result but is not ideal as it is possible for a property to be forgotten and left undefined, possibly leaving the entire object in an invalid state. This problem is solved with the use of constructors by initialising all the public and private state of the object.

SELF ASSESSMENT EXERCISE

What is the main function of a constructor?

____
____
____
____

3.2 What is a Constructor?

A constructor is a special class member that is executed when a new object is created. The constructor's job is to initialise all of the public and private state of the new object and to perform any other tasks that the programmer requires before the object is used.

3.3 Creating a New Class

In this unit, we will create a new class to represent a triangular shape. This class will define three properties: the triangle's height, base-length and area. To begin, create a new console application and add a class named "Triangle". Copy and paste the following code into the class to create the three properties that are required.

```
public class Triangle
{
   private int _height;
```

```
    private int _baseLength;

    public int Height
    {
        get
        {
            return _height;
        }
        set
        {
            if (value < 1 || value > 100)
            {
                throw new OverflowException();
            }

            _height = value;
        }
    }

    public int BaseLength
    {
        get
        {
            return _baseLength;
        }
        set
        {
            if (value < 1 || value > 100)
            {
                throw new OverflowException();
            }

            _baseLength = value;
        }
    }

    public double Area
    {
        get
        {
            return _height * _baseLength * 0.5;
        }
    }
}
```

4.0 CONCLUSION

Winding up, we can go over the key points of this unit. A ***constructor*** is a special class member that is executed when a new object is created. The main function of the constructor is to initialise all of the public and private state of the new object and to perform any other tasks that the programmer requires before the object is used. To create a new class to represent a triangular shape that will define three properties: the triangle's height, base-length and area. First, create a new console application and add a class named "Triangle". Then key in the required appropriate code into the class to create the three properties that are required.

5.0 SUMMARY

This unit provided an overview of constructors, their key role and procedure for creating a class of a new object to define some properties. We hope you have found this unit interesting.

6.0 TUTOR MARKED ASSIGNMENT

- What is a constructor?
- Outline the procedure involved in creating a new class to represent a rectangular shape that will define the rectangle’s length, width and area

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)

5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
12. Martin, A and Luca, C. (2005). *A Theory of Objects*.
13. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
14. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
15. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
16. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
17. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
18. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
19. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 2 THE DEFAULT CONSTRUCTOR

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
  3.1 What is a Default Constructor?
  3.2 Replacing the Default Constructor
    3.2.1 Syntax for Adding a New Constructor to a Class
    3.2.2 Maintaining the Valid Range of the Sides of a Triangle
  3.3 Executing the Constructor
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

This unit gives a general idea of the default constructor. It equally states the syntax for adding a new constructor to a class and gives the description of how the constructor is executed. Enjoy your studies.

2.0 OBJECTIVES

What you would study in this unit, would equip you to do the following:

- Describe the default constructor
- State the Syntax for adding a new constructor to a class
- Give the code for maintaining the valid range of the sides of a triangle (***class***)
- Describe how to execute the constructor

3.1 What is a Default Constructor?

A default constructor is simply the constructor that is applied in a class if no other constructor is explicitly declared by the developer. This constructor causes all of the value type properties and variables to be set to zero and all reference types to be set to null. If these are invalid values for an object, as in the Triangle class above, the default constructor will always create an object that is invalid. In this case, the default constructor should be replaced.

SELF ASSESSMENT EXERCISE

When is a default constructors applied in a class?

____
____
____

3.2 Replacing the Default Constructor

3.2.1 Syntax for Adding a New Constructor to a Class

The syntax to add a new constructor to a class is similar to that of adding a method. However, the constructor has the same name as the class and does not include a return type. The declaration for the Triangle's new constructor is therefore simply:

```
public Triangle()
```

3.2.2 Maintaining the Valid Range of the Sides of a Triangle

To ensure that all triangles will have a height and base-length within the valid range, we will make the class' constructor set both of these properties to one unit for all new Triangle objects. This is achieved by simply setting the underlying private variables in the constructor's code block. Add the following code within the class' code block to add the constructor.

```
public Triangle()
{
   Console.WriteLine("Triangle constructor executed");

   _height = _baseLength = 1;
}
```

NB: The Console.WriteLine command is added to show that the constructor has been executed in the examples. This would generally not be added to a real class definition.

3.3 Executing the Constructor

The newly added constructor replaces the default constructor and is executed automatically when a new Triangle is instantiated. This can be tested by adding some code to the console application's Main method. Add the following code and execute the program to test the results and to see that the height and base-length are set correctly.

```
static void Main(string[] args)
{
   Triangle triangle = new Triangle();

   Console.WriteLine("Height:\t{0}", triangle.Height);
   Console.WriteLine("Base:\t{0}", triangle.BaseLength);
   Console.WriteLine("Area:\t{0}", triangle.Area);
}

/* OUTPUT

Triangle constructor executed
Height: 1
Base:   1
Area:   0.5

*/
```

4.0 CONCLUSION

To end, a default constructor is simply the constructor that is applied in a class if no other constructor is explicitly declared by the developer. We also identified. Ensure that all classes, say triangles, have a height and base-length within the valid range, by simply setting the underlying private variables in the constructor's code block.

5.0 SUMMARY

This unit provided an overview of the default constructor. It equally stated the syntax for adding a new constructor to a class and explained how the constructor is executed. We hope you found this unit enlightening.

6.0 TUTOR MARKED ASSIGNMENT

- Give the code for maintaining the valid range of the sides of a triangle (***class***)
- Describe how to execute the constructor

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages,* Cambridge University Press, 2003, p.278
11. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
12. Martin, A and Luca, C. (2005). *A Theory of Objects*.
13. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
14. Michael Lee Scott (2006). *Programming language pragmatics,* (2nd Edition) p. 470
15. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.

16. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
17. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
18. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
19. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 3 DESTRUCTORS

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
    3.1 Destructors
        3.1.1 Application of Destructors
        3.2.1 Calling the Finalizer of a Base Class Object
    3.2 Creating a Destructor
    3.3 Executing the Destructor
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

In the previous unit we examined default constructors. Here, we will be looking at destructors. These special methods allow an object to be initialised on instantiation and to perform final actions before it is removed from memory. Do make the most of your studies.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Define a destructor
- Discover the application of destructors
- Call the function of a base class object
- Give the syntax for creating a destructor
- State the condition for automatically executing destructors

3.0 MAIN CONTENT

3.1 Destructors

A ***destructor*** is a special member that can be added to a class. It is called automatically when an object is no longer required and is being removed from memory.

3.1.1 Application of Destructors

The destructor can be useful because it can ensure that all objects of a particular class terminate cleanly. For example, if a class maintains a database connection, the destructor can be used to ensure that any database transaction is rolled back if it has not been committed and that the database connection is closed when the object is cleaned up.

3.1.2 Calling the Finalizer of a Base Class Object

Destructors are sometimes known as finalizers. In other .NET languages, the object class' Finalize method can be overridden to provide the clean-up code. In C#, this is not permitted so the destructor syntax must be used. This forces the calling of the finalizer of the base class of the object too, by implicitly converting the destructor statements into the Finalize code below:

```
protected override void Finalize()
{
   try
   {
      // Destructor code
   }
   finally
   {
      base.Finalize();
   }
}
```

SELF ASSESSMENT EXERCISE

How would you call the finalizer of a base class object?
____
____
____

3.2 Creating a Destructor

The syntax to create a destructor is very simple. The class name is prefixed with a tilde character (~). No parameters are permitted as the destructor cannot be called manually. To add a destructor to the Triangle class, add the following code:

```
~Triangle()
{
   Console.WriteLine("Triangle destructor executed");
}
```

NB: The destructor outputs a message only as there is no clean up required for Triangle objects.

3.3 Executing the Destructor

The destructor is executed automatically when the object is being removed from memory. This can be demonstrated by running the console application. The output should be as follows:

```
Triangle constructor executed
Height: 5
Base:   8
Area:   20
Triangle destructor executed
```

NB: Classes that require finalizers should also implement the interface.

4.0 CONCLUSION

In conclusion, a ***destructor*** also known as a ***finalizer*** is a special member that can be added to a class. It is executed automatically when the object is being removed from memory.

5.0 SUMMARY

We considered destructors, their application, and the syntax for creating them as well as the condition for automatically executing them. To test your knowledge, let us attempt the exercise below.

6.0 TUTOR MARKED ASSIGNMENT

- Explain how destructors are called
- Give the code for adding a destructor to the Triangle class

ANSWER TO SELF ASSESSMENT EXERCISE

The finalizer of the base class of the object is called by implicitly converting the destructor statements into the Finalize code below:

```
protected override void Finalize()
{
   try
   {
      // Destructor code
   }
   finally
   {
      base.Finalize();
   }
}
```

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.

2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
12. Martin, A and Luca, C. (2005). *A Theory of Objects*.
13. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
14. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
15. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
16. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
17. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
18. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
19. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 4 GARBAGE COLLECTION

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
    3.1 Garbage Collection
    3.2 Role of a Garbage Collector
    3.3 Garbage Collection and Destructors
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

In this unit we will be learning about the concept of garbage collection. We would also discover the role of garbage collection and its relationship with destructors. Hope you would be able to grasp the key points.

2.0 OBJECTIVES

What you would study in this unit, would enable you to:

- Describe the concept of garbage collection
- Discover the role of a garbage collector
- Identify the relationship between garbage collection and destructors

3.0 MAIN CONTENT

3.1 Garbage Collection

When objects are created they occupy some memory. As memory is a limited resource, it is important that once the objects are no longer required, they are cleaned up and their memory is reclaimed. The .NET framework uses a system of *garbage collection* to perform this activity automatically so that the developer does not need to control this directly.

SELF ASSESSMENT EXERCISE

Why is garbage collection essential in the process of object creation?

3.2 Role of the Garbage Collector

- The garbage collector periodically checks for objects in memory that are no longer in use and that are referenced by no other objects.
- It also detects groups of objects that reference each other but as a group have no other remaining references. When detected, the objects' destructors are executed and the memory utilised is returned to the pool of available memory.

3.3 Garbage Collection and Destructors

The garbage collection system is completely automated and requires little consideration by the C# developer. However, it is important to understand that an object's destructor is not called immediately, that it falls out of scope. There can be a delay until the garbage collector decides to reclaim the object's memory and it is only at this point that the destructor is called.

4.0 CONCLUSION

In this unit, we saw that garbage collectors periodically check for objects in memory that are no longer in use and that are referenced by no other objects. They also detect groups of objects that reference each other but as a group have no other remaining references. The destructor is called when the garbage collector decides to reclaim the object's memory.

5.0 SUMMARY

This unit introduced the concept of garbage collection. We equally identified the key roles of garbage collectors as well as their relationship with destructors. We hope you enjoyed your studies.

6.0 TUTOR MARKED ASSIGNMENT

What are the key roles of the garbage collector?

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Kay, Alan. *The Early History of Smalltalk.* http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
12. Martin, A and Luca, C. (2005). *A Theory of Objects*.
13. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
14. Michael Lee Scott (2006). *Programming language pragmatics,* (2nd Edition) p. 470

15. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
16. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
17. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
18. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
19. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

MODULE 4 C# STATIC BEHAVIOUR

Unit 1 Methods
Unit 2 Creating a Static Method
Unit 3 Static Properties
Unit 4 Static Constructors

UNIT 1 C# METHODS

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
  3.1 Relevance of Methods
  3.2 Syntax required for creating a Method
  3.3 One Simple Method: OneMethod.cs
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

In the previous modules, we discussed classes, methods and constructors as well as destructors. This unit describes the concept of static behaviours. These behaviours are accessible from classes without the requirement to instantiate new objects.

2.0 OBJECTIVES

What you would study in this unit, would enable you to:

- Discover the importance of methods
- State the syntax required for creating a method
- Identify the

3.0 MAIN CONTENT

3.1 Relevance of Methods

Methods help you separate your code into modules that perform specific tasks. They are extremely useful because they allow you to separate your logic into different units. You can pass information to methods, have it perform one or more statements, and retrieve a return value. The capability to pass parameters and return a value is optional and depends on what you want the method to do.

SELF ASSESSMENT EXERCISE

Why are methods extremely useful?

3.2 Syntax Required for Creating a Method

Here's a description of the syntax required for creating a method:

```
 attributes modifiers return-type method-name(parameters )
{
   statements
}
```

The return-type can be any C# type. It can be assigned to a variable for use later in the program. The method name is a unique identifier for what you wish to call a method. To promote understanding of your code, a method name should be meaningful and associated with the task the method performs. Parameters allow you to pass information to and from a method. They are surrounded by parenthesis. Statements within the curly braces carry out the functionality of the method.

3.3 One Simple Method: OneMethod.cs

The program below illustrates one simple method:

```
using System;

class OneMethod
{
    public static void Main()
    {
        string myChoice;

        OneMethod om = new OneMethod();

        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;
    }
}
```

The program above accepts input in the *Main()* method, this functionality has been moved to a new method called *getChoice()*. The return type is a *string*. This *string* is used in the *switch* statement in *Main()*. The method name "getChoice" describes what happens when it is invoked. Since the parentheses are empty, no information will be transferred to the *getChoice()* method.

Within the method block we first declare the variable *myChoice*. Although this is the same name and type as the *myChoice* variable in *Main()*, they are both unique variables. They are local variables and they are visible only in the block they are declared. In other words, the *myChoice* in *getChoice()* knows nothing about the existence of the *myChoice* in *Main()*, and vice versa.

The *getChoice()* method prints a menu to the console and gets the user's input. The *return* statement sends the data from the *myChoice* variable back to the caller, *Main()*, of *getChoice()*. Notice that the type returned by the *return* statement must be the same as the return-type in the function declaration. In this case it is a *string*.

In the *Main()* method we must instantiate a new *OneMethod* object before we can use *getChoice()*. This is because of the way *getChoice()* is declared. Since we did not specify a *static* modifier, as for *Main()*, *getChoice()* becomes an instance method. The difference between instance methods and *static* methods is that multiple instances of a class can be created (or instantiated) and each instance has its own separate *getChoice()* method. However, when a method is *static*, there are no instances of that method, and you can invoke only that one definition of the *static* method.

So, as stated, *getChoice()* is not *static* and therefore, we must instantiate a new object to use it. This is done with the declaration *OneMethod om = new OneMethod()*. On the left hand side of the declaration is the object reference *om* which is of type *OneMethod*. The distinction of *om* being a reference is important. It is not an object itself, but it is a variable that can refer (or point) to an object of type *OneMethod*. On the right hand side of the declaration is an assignment of a new *OneMethod* object to the reference *om*. The keyword *new* is a C# operator that creates a new instance of an object on the heap. What is happening here is that a new *OneMethod* instance is being created on the heap and then being assigned to the *om* reference. Now that we have an instance of the *OneMethod* class referenced by *om*, we can manipulate that instance through the *om* reference.

Methods, fields, and other class members can be accessed, identified, or manipulated through the "." (dot) operator. Since we want to call *getChoice()*, we do so by using the dot operator through the *om* reference: *om.getChoice()*. The program then executes the statements in the *getChoice()* block and returns. To capture the value *getChoice()* returns, we use the "=" (assignment) operator. The returned *string* is placed into *Main()'s* local *myChoice* variable. From there, the rest of the program executes as expected, using concepts from previous units.

4.0 CONCLUSION

In this unit, we learnt that methods are valuable because they allow you to separate your logic into different units. You can pass information to methods, have it perform one or more statements, and retrieve a return value. We equally identified the syntax for creating a method.

5.0 SUMMARY

In this unit, we considered methods and illustrated one simple method. We hope you enjoyed your studies. Let us attempt the question below.

6.0 TUTOR MARKED ASSIGNMENT

State the syntax for creating a method

7.0 REFERENCES/FURTHER READINGS

21. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
22. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
23. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
24. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
25. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
26. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
27. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
28. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
29. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.

30. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
31. Joyce, F. (2006). Microsoft Visual C#.NET with Visual Studio 2005
32. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
33. Martin, A and Luca, C. (2005). *A Theory of Objects*.
34. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
35. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
36. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
37. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
38. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
39. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
40. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 2 CREATING A STATIC METHOD

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
    3.1 Declaring a Static Method in C#
    3.2 Calling a Static Method in C#
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, you'll gain knowledge of how to declare and call static methods in C#. Enjoy your studies.

2.0 OBJECTIVES

What you would study in this unit, would equip you to do the following:

- Declare a static method
- Add a method to a new class
- Call a static method

3.0 MAIN CONTENT

3.1 Declaring a Static Method in C#

A static method is declared using a similar syntax to a class method.

The 'static' keyword is used in the declaration to indicate the modified

behaviour. To add the mass calculation method to the new class,

insert the following code:

```
public static int CalculateMass(int density, int volume)
{
   return density * volume;
}
```

SELF ASSESSMENT EXERCISE

State the code for adding a mass calculation method to a new class

3.2 Calling a Static Method in C#

Static methods are called without reference to a specific instance of a class. Instead of supplying an object name followed by the method name, the class name and method name are used, separated by a full stop (or period). We can demonstrate this using the Main method of the console application as follows:

```
static void Main(string[] args)
{
   int density = 50;
   int volume = 100;
   int mass = MassCalculator.CalculateMass(density, volume);

   Console.WriteLine("Mass: {0}", mass);          // Outputs "Mass:
5000"
}
```

It is important to understand that static methods may not directly use non-static members. It is invalid for one static method to directly call a non-static method or property without first instantiating an object. Similarly, private variables that are not marked as static cannot be utilised by a static method. The reverse of this is not true of course, as a non-static member can call a static method.

4.0 CONCLUSION

To wrap up, we discovered that the 'static' keyword is used to modify behaviour. We also learnt that ***Static methods*** are called without reference to a specific instance of a class.

5.0 SUMMARY

In this unit, we learnt how to declare and call static methods in C#. Let us now attempt the questions below.

6.0 TUTOR MARKED ASSIGNMENT

What is the function of the ‘static’ keyword in a C# declaration?

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.

4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages,* Cambridge University Press, 2003, p.278
11. Joyce, F. (2006). Microsoft Visual C#.NET with Visual Studio 2005
12. Kay, Alan. *The Early History of Smalltalk.* http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
13. Martin, A and Luca, C. (2005). *A Theory of Objects*.
14. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
15. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
16. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
17. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
18. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
19. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
20. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 3 STATIC PROPERTIES

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
    3.1 Types of Properties
    3.2 Creating a Static Private Variable
    3.3 Exposing a Static Property
    3.4 Using a Static property
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, you have a chance to learn another aspect of object-oriented programming using C#. We will study the types of static properties and the procedure for using them. We'll equally look at the condition for adding a static property to a class.

2.0 OBJECTIVES

What you would study in this unit, would enable you do the following:

- List the common types of static properties
- Explain how to create static private variable
- State the condition for adding a static property to a class
- Outline the procedure for using a static property

3.0 MAIN CONTENT

3.1 Types of Properties

Static properties provide the functionality of standard properties, except that the property is not linked to any instance of the class. Static properties can be ***read/write, read-only or write-only*** as with standard properties. As such, they can essentially be thought of as global variables.

SELF ASSESSMENT EXERCISE

List 3 types of static properties

3.2 Creating a Static Private Variable

In the earlier unit describing class properties we created a class-level private variable to hold the data behind a property. As mentioned above, private variables may not be accessed by static methods, and this also applies in the case of static properties. Instead, we must create a static private variable if the property value is to be held rather than calculated.

In the MassCalculator class we will implement a call counter that is incremented every time the mass calculation method is used. The current value for the call counter is stored in a static private variable that may now be added within the class' code block:

```
private static int _callCount;
```

To maintain the counter, adjust the CalculateMass method so that it increments the variable on every call:

```
public static int CalculateMass(int density, int volume)
{
   _callCount++;
   return density * volume;
}
```

3.3 Exposing a Static Property

As you may expect, adding a static property to a class requires only that the declaration includes the 'static' keyword to modify the property's behaviour. We can now add the read-only call count property to the class.

```
public static int CallCount
{
   get
   {
      return _callCount;
   }
}
```

3.4 Using a Static Property

To use the static property, the property name is preceded by the class name and the member access operator (.). To demonstrate, adjust the Main method to perform two mass calculations and output the call count property as follows:

```
static void Main(string[] args)
{
   int density = 50; int
   volume = 100; int
   volume2 = 180;

   int mass1 = MassCalculator.CalculateMass(density, volume);
   int mass2 = MassCalculator.CalculateMass(density, volume2);
   int calls = MassCalculator.CallCount;

   Console.WriteLine("Mass1: {0}", mass1);        // Outputs "Mass1:
5000"
   Console.WriteLine("Mass2: {0}", mass2);        // Outputs "Mass2:
9000"
   Console.WriteLine("Calls: {0}", calls);        // Outputs "Calls: 2"
}
```

4.0 CONCLUSION

In conclusion, we have seen that static properties are often thought of as global variables, they can be ***read/write, read-only or write-only.*** The only condition for adding a static property to a class is that the declaration includes the 'static' keyword to modify the property's behaviour.

5.0 SUMMARY

In sum, we discovered the common types of static properties, the condition for adding a static property to a class as well as the procedure for using a static property. You can now attempt the questions below.

6.0 TUTOR MARKED ASSIGNMENT

Outline the procedure for using a static property

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages,* Cambridge University Press, 2003, p.278
11. Joyce, F. (2006). Microsoft Visual C#.NET with Visual Studio 2005
12. Kay, Alan. *The Early History of Smalltalk.* http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.

13. Martin, A and Luca, C. (2005). *A Theory of Objects*.
14. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
15. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
16. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
17. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
18. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
19. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
20. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 4 STATIC CONSTRUCTORS

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
3.1 Overview of Static Constructors
3.2 Properties of Static Constructors
3.3 Adding a Static Constructor
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

This unit gives a general idea of static constructors and their specific properties. You'll actually find this aspect simple and interesting. Enjoy your studies!

2.0 OBJECTIVES

What you would study in this unit, would equip you to do the following:

- Explain the concept of static constructors
- Identify the properties of static constructors
- Outline the procedure involved in adding a static constructor

3.0 MAIN CONTENT

3.1 Overview of Static Constructors

A static constructor is used to initialise the static state of a class when it is first used. A static constructor is always declared as private and as such may not be directly called by a program. A static constructor therefore has no facility to add parameters. It is also not possible to include a static destructor.

SELF ASSESSMENT EXERCISE

How are static constructors declared?

3.2 Properties of Static Constructors

Static constructors have the following properties:

- A static constructor does not take access modifiers or have parameters.
- A static constructor is called automatically to initialize the before the first instance is created or any static members are referenced.
- A static constructor cannot be called directly.
- The user has no control on when the static constructor is executed in the program.
- A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.
- Static constructors are also useful when creating wrapper classes for unmanaged code, when the constructor can call the **LoadLibrary** method.
- If a static constructor throws an exception, the runtime will not invoke it a second time, and the type will remain uninitialized for the lifetime of the application domain in which your program is running.

3.3 Adding a Static Constructor

To add a static constructor, create a private constructor with the static keyword as a prefix. The following code could be added to the MassCalculator class if the appropriate static methods were available to retrieve the previously saved call count from a file or other storage.

```
static MassCalculator()
{
   _callCount = InitialiseCallCount();
}
```

4.0 CONCLUSION

A static constructor is used to initialise the static state of a class when it is first used. It is constantly declared as ***private*** and as such may not be directly called by a program. A static constructor is added by creating a private constructor with the static keyword as a prefix.

5.0 SUMMARY

In this unit, we learnt about static constructors, their properties as well as the procedure for adding a static constructor. Ok! We can now attempt the questions below.

6.0 TUTOR MARKED ASSIGNMENT

State at least 4 properties of static constructors
Give a brief description of how to add a static constructor

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128.

http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.

3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Joyce, F. (2006). Microsoft Visual C#.NET with Visual Studio 2005
12. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
13. Martin, A and Luca, C. (2005). *A Theory of Objects*.
14. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
15. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
16. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
17. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
18. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
19. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
20. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

MODULE 5 POLYMORPHISM

Unit 1 Introduction to Polymorphism
Unit 2 Overloaded Method
Unit 3 C# Basic Operator Overloading
Unit 4 C# Indexers
Unit 5 C# Inheritance and Polymorphism

UNIT 1 INTRODUCTION TO POLYMORPHISM

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
  3.1 Polymorphism versus Object-oriented Programming
  3.2 Method Overloading
  3.3 Advantages of Method Overloading
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

This unit is a preamble to polymorphism. It describes the concept of method overloading stating their advantages. Do take note of the key points.

2.0 OBJECTIVES

What you would study in this unit, would equip you to:

- Give a brief description of the term "polymorphism"
- Identify the relationship between polymorphism and object-oriented programming
- Describe the concept of "Method Overloading"
- Explain the expression "Signature of the Method"
- Outline the advantages of method overloading

3.0 MAIN CONTENT

3.1 Polymorphism versus Object-oriented Programming

One of the key features of object-oriented programming is ***polymorphism***. Polymorphism permits objects to behave in different ways according to the manner in which they are used. One part of polymorphism is the ability for a method to behave differently according to the types and number of parameters that are passed to it. This is achieved through *method overloading*.

SELF ASSESSMENT EXERCISE

State the relationship between polymorphism and object-oriented programming

____
____
____

3.2 Method Overloading

Method overloading is a technique which allows the programmer to define many methods with the same name but with a different set of parameters. Each combination of parameter types is known as a ***signature of the method***. When a call is made to one of these overloaded methods, the compiler automatically determines which of the methods should be used according to the arguments used in the call and the available method signatures.

3.3 Advantages of Method Overloading

One of the greatest advantages of method overloading is the improvement that it provides to code readability and maintainability. In languages that do not support this technique, or that of optional operands, a new method must be created for every possible combination of parameters. For example, in the ANSI C programming language to truncate a value you would use *trunc*, *truncf* or *truncl* according to the data type being rounded.

In C#, method overloading allows you to always call *Math.Truncate*. This becomes even more useful when a change in the requirements of the program means that data types change. Unlike with the older languages, the C# truncate method would require no code modification.

When using method overloading, each version of a method should perform the same general function using different data types or numbers of parameters. Although it is possible to create two methods with the same name that perform completely different tasks, this just reduces the quality of your code.

4.0 CONCLUSION

To end, we learnt that polymorphism is one of the key features of object-oriented programming and that method overloading allows the programmer to define many methods with the same name but with a different set of parameters.

5.0 SUMMARY

In this unit, we looked at: the relationship between object-oriented programming and polymorphism as well as the concept of method overloading and their advantages. You may now proceed to the tutor marked assignment below.

6.0 TUTOR MARKED ASSIGNMENT

Explain the expression “Signature of the Method”

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)

6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Joyce, F. (2006). Microsoft Visual C#.NET with Visual Studio 2005
12. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
13. Martin, A and Luca, C. (2005). *A Theory of Objects*.
14. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
15. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
16. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
17. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
18. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
19. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
20. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 2 OVERLOADED METHOD

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
    3.1 Creating an Overloaded Method
    3.2 Automatic Type Conversion
    3.3 Return Type Limitation
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

In this unit, we'll learn about the procedure for creating a method as well as adding a new method. We’ll equally discuss the relevance of having different signatures in a method declaration.

2.0 OBJECTIVES

What you would study in this unit, would enable you do the following:

- List 2 techniques of creating overloaded methods
- Identify the procedure for creating a method
- Explain how to add a new method
- State the condition for “Automatic Type Conversion”

3.0 MAIN CONTENT

3.1 Creating an Overloaded Method

Creating an overloaded method is achieved by simply adding two or more methods of the same name to a class. The methods can be normal or static. As long as the method signatures differ, the code will compile correctly. In the following example, we will create a method that calculates the square of a number using different data types. To begin, create a new console application and add a new class file named "Calculate". Add the following code to the new class:

```
class Calculate
{
   public static int Square(int number)
   {
      Console.WriteLine("Integer Square calculated");
      return number * number;
   }
}
```

The new method calculates the square of an integer value. The Console.WriteLine command is included so that we can easily see the flow of execution. To test the calculation, modify the Main method of the program as follows and run the program to see the results.

```
static void Main(string[] args)
{
   int squareMe = 5;
   Console.WriteLine(Calculate.Square(squareMe));
}
```

```
/* OUTPUT

Integer Square calculated
25

*/
```

The program takes the integer value and squares it using the static Square method of the Calculate class, giving the correct result of twenty-five. However, if the data type of the value to be squared is changed, the result can be different. If you were to change the Main method so that the squared variable is a *double* the code will no longer compile because the double data type may not be implicitly cast to an integer.

```
static void Main(string[] args)
{
   double squareMe = 5;                    // Does not compile
   Console.WriteLine(Calculate.Square(squareMe));
}
```

In order to support the double data type we can add a second variation of the method to the Calculate class. This *overloaded* method will accept and return doubles rather than integers. Add the new method as follows:

```
public static double Square(double number)
{
   Console.WriteLine("Double Square calculated");
   return number * number;
}
```

Now that the Calculate class can square integers and doubles, change the Main method as follows and execute the program. You can see that the compiler correctly determines which of the overloaded methods to execute for each call to Calculate.Square.

```
static void Main(string[] args)
{
   double squareMe = 5; int squareMeToo = 5;
   Console.WriteLine(Calculate.Square(squareMe));
   Console.WriteLine(Calculate.Square(squareMeToo));
}

/* OUTPUT

Double Square calculated
25
Integer Square calculated
25

*/
```

SELF ASSESSMENT EXERCISE

Give a brief description of how overloaded methods are created

3.2 Automatic Type Conversion

In the example above, a second variant of the method was created because we wanted to square a double and this could not be implicitly cast to an integer. However, where an implicit cast is possible, the compiler will perform this conversion automatically. In the following example, the Main method is updated to use a float. As you can see, because no overloaded method exists specifically for floats, the double variation is used instead.

```
static void Main(string[] args)
{
   float squareMe = 5;
   Console.WriteLine(Calculate.Square(squareMe));
}

/* OUTPUT

Double Square calculated
25

*/
```

3.3 Return Type Limitations

The signature defines the name and the set of parameters for the method. In order to use overloaded methods, the signature must differ for each method declaration. This means that every overloaded method in a class must have either a different number of parameters or a different set of argument data types to every other method with the same name. However, the return type of the method is not included in this signature. This means that two methods that differ only in return type cannot be created in the same class. For this reason, the following code is invalid and so will not compile:

```
class Calculate
{
   public static int Square(double number)
   {
      return (int)(number * number);
```

```
    }

    public static double Square(double number)
    {
        return number * number;
    }
}
```

4.0 CONCLUSION

To wrap up, we learnt that: an overloaded method is created by simply adding two or more methods of the same name to a class, the compiler carries out automatic conversion where an implicit cast is possible, and that two methods that differ only in return type cannot be created in the same class.

5.0 SUMMARY

In this unit, we identified the techniques of creating overloaded methods and the condition for automatic type conversion. You may now proceed to the tutor marked assignment. Good luck!

6.0 TUTOR MARKED ASSIGNMENT

State the condition for the compiler to carry out automatic conversion.

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)

6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Joyce, F. (2006). Microsoft Visual C#.NET with Visual Studio 2005
12. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
13. Martin, A and Luca, C. (2005). *A Theory of Objects*.
14. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
15. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
16. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
17. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
18. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
19. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
20. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 3 C# OPERATOR OVERLOADING

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
  3.1 Operator Overloading
    3.1.1 Creating a Two-Dimensional Vector Class
  3.2 Binary Operator Overloading
    3.1.1 Declaring a Binary Operator
    3.1.2 Parts of a Declared Binary Operator
  3.3 Creating the Addition (+) Operator
  3.4 Creating the Subtraction (-) Operator
  3.5 Creating the Multiplication (*) Operator
  3.6 Unary Operators
  3.7 Creating the Increment and Decrement Operators
  3.8 Creating the Negation Operator
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

In this unit you learn about the process of declaring operator overloading. You will equally learn about the rules for carrying out basic operations on operators.

OBJECTIVES

At the end of this unit, you should be able to:

- Describe the process of operator overloading
- State the syntax declaring binary operators
- Describe the parts of a declared binary operator
- Identify the procedure for carrying out basic operations on operators
- State the syntax for declaring common operators

3.0 MAIN CONTENT

3.1 Operator Overloading

Operator overloading is simply the process of adding operator functionality to a class. This allows you to define exactly how the operator behaves when used with your class and other data types. This can be standard uses such as the ability to add the values of two vectors, more complex mathematics for multiplying matrices or non-arithmetic functions such as using the + operator to add a new item to a collection or combine the contents of two arrays. Multiple overloaded versions of operators may also be created to provide different functionality according to the data types being processed, in a similar manner to the varying signatures of method overloading.

3.1.1 Creating a Two-Dimensional Vector Class

The first step in creating a new class to represent a two-dimensional vector with X and Y properties is to create a new console application named "VectorDemo" and add a new class file named "Vector".

Add the following code to the new class to create the properties and a basic constructor:

```
private int _x, _y;

public Vector(int x, int y) { _x = x; _y = y; }

public int X
{
   get { return _x; }
   set { _x = value; }
}

public int Y
{
   get { return _y; }
   set { _y = value; }
}
```

SELF ASSESSMENT EXERCISE

What is the procedure for creating a two-dimensional vector class?

3.2 Binary Operator Overloading

The ***binary operator*** is a type of operator that requires ***two*** values to work with. These include the simple arithmetic operators such as +, -, *, / and %.

3.2.1 Declaring a Binary Operator

To declare a binary operator, the following syntax is used:

```
public static result-type operator binary-operator (
    op-type operand,
    op-type2 operand2
)
```

3.2.2 Parts of a Declared Binary Operator

This initially appears to be a rather complex declaration but in fact is quite simple. The declaration starts with ***public static*** as all operators must be declared as such. Other scopes are not permitted and neither are non-static operators.

The *result-type* defines the data type or class that is returned as the result of using the operator. Usually this will be the same type as the class that it is being defined within. However, that need not be the case and it is perfectly valid to return data of a different type.

The *operator* keyword is added to tell the compiler that the following *binary-operator* symbol is an operator rather than a normal method. This operator will then process the two *operand* parameters, each prefixed with its data type *(op-type* and *op-type2*). As least one of these operands must be the same type as the containing class.

3.3 Creating the Addition (+) Operator

The syntax for binary operators can now be used to create a new addition operator for the Vector class. This operator will simply add the X and Y elements of two vectors together and return a new vector containing the result. Add the following to the Vector class to provide this functionality. Note that a new Vector is created rather than adjusting one of the operands. This is because the operands are *reference-types* and the original values should not be updated in this case.

```
public static Vector operator +(Vector v1, Vector v2)
{
   return new Vector(v1.X + v2.X, v1.Y + v2.Y);
}
```

We can now test the Vector's new operator by modifying the program's main method. The following program instantiates two Vector objects, adds them together and outputs the values of the resultant Vector's X and Y properties.

```
static void Main(string[] args)
{
   Vector v1 = new Vector(4, 11);
   Vector v2 = new Vector(0, 8);

   Vector v3 = v1 + v2;

   Console.WriteLine("({0},{1})", v3.X, v3.Y);     // Outputs "(4,19)"
}
```

3.4 Creating the Subtraction (-) Operator

Addition is a ***commutative*** operation. This means the order of the two operands can be swapped without affecting the outcome. However, this is not the case in subtraction, thus it is important to remember that the first operand in the declaration represents the value to the left of the operator and the second operand represents the value to the right. If these are used incorrectly, the resultant value will be incorrect. Using this knowledge we can add a subtraction operator to the Vector class:

```
public static Vector operator -(Vector v1, Vector v2)
{
   return new Vector(v1.X - v2.X, v1.Y - v2.Y);
}
```

To test the new operator, modify the Main method as follows and execute the program.

```
static void Main(string[] args)
{
   Vector v1 = new Vector(4, 11);
   Vector v2 = new Vector(0, 8);

   Vector v3 = v1 - v2;

   Console.WriteLine("({0},{1})", v3.X, v3.Y);     // Outputs "(4,3)"
}
```

3.5 Creating the Multiplication (*) Operator

The last binary operator that will be added to Vector class is for multiplication. This operator will be used to scale the vector by multiplying the X and Y properties by the same integer value. This demonstrates the use of operands of a different type to the class they are defined within.

```
public static Vector operator *(Vector v1, int scale)
{
   return new Vector(v1.X * scale, v1.Y * scale);
}
```

To test the multiplication operator, adjust the Main method again:

```
static void Main(string[] args)
{
   Vector v1 = new Vector(4, 11);

   Vector v2 = v1 * 3;

   Console.WriteLine("({0},{1})", v2.X, v2.Y);     // Outputs "(12,33)"
}
```

In the operator code for the multiplication operator, the Vector is the first operand and the integer the second. This means that the order used in the multiplication statement must have the Vector at the left of the operator and the integer value to the right. Changing the order of the operands in the Main method will cause a compiler error.

```
static void Main(string[] args)
{
   Vector v1 = new Vector(4, 11);
```

```
    Vector v2 = 3 * v1;

    Console.WriteLine("({0},{1})", v2.X, v2.Y);     // Does not compile
}
```

If the class must support both variations of multiplication, both must be declared in the code. This provides the benefit of allowing the order of operands change the underlying function. To provide the second variation of multiplication, add the following code to the Vector class. Afterwards, the program will execute correctly.

```
public static Vector operator *(int scale, Vector v1)
{
    return new Vector(v1.X * scale, v1.Y * scale);
}
```

3.6 Unary Operators

Unary operators are operators that require a single operand. These include the simple increment (++) and decrement (--) operators. To declare a unary operator, the following syntax is used:

public static *result-type* operator *unary-operator* (*op-type operand*)

This syntax is almost identical to that used for binary operators. The difference is that only one operand is declared. The operand type must be the same as the class in which the operator is declared.

3.7 Creating the Increment and Decrement Operators

Using the syntax defined above, we can now add the increment and decrement operators to the Vector class. Note that there is only a single definition for each. There is no way to differentiate between prefix and postfix versions of the operator so both provide the same underlying functionality.

To declare the two operators, add the following code to the Vector class. Each increments or decrements both the X and Y properties for Vector objects.

```
public static Vector operator ++(Vector v)
```

```
{
   v.X++;
   v.Y++;
   return v;
}

public static Vector operator --(Vector v)
{
   v.X--;
   v.Y--;
   return v;
}
```

To test these operators, update and execute the Main method:

```
static void Main(string[] args)
{
   Vector v1 = new Vector(4, 11);

   v1++;
   Console.WriteLine("({0},{1})", v1.X, v1.Y);     // Outputs "(5,12)"

   v1--;
   Console.WriteLine("({0},{1})", v1.X, v1.Y);     // Outputs "(4,11)"
}
```

3.8 Creating the Negation Operator

The last arithmetic unary operator to be considered in this unit is the *negation operator*. This is the unary version of subtraction used to identify a negative version of a value. We can add this operator using the following code:

```
public static Vector operator -(Vector v)
{
   return new Vector(-v.X, -v.Y);
}
```

To test the negation operator, update the Main method and run the program.

```
static void Main(string[] args)
{
   Vector v1 = new Vector(4, 11);
```

```
    Vector v2 = -v1;
    Console.WriteLine("({0},{1})", v2.X, v2.Y);      // Outputs "(-4,-11)"
}
```

4.0 CONCLUSION

In this unit, we were made to understand that the ***binary operator*** is a type of operator that requires ***two*** values to work with while ***unary*** operators are operators that require a single operand. We also spotted the various parts of a declared binary operator as well as the procedure for carrying out basic operations on the common C# operators.

5.0 SUMMARY

In this unit, we learnt the following: the process of operator overloading, the syntax for declaring common operators as well as the procedure for carrying out basic operations on C# operators. Let us now attempt the questions below.

6.0 TUTOR MARKED ASSIGNMENT

- Define a binary operator?
- State the syntax for declaring a unary operator

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)

6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Joyce, F. (2006). Microsoft Visual C#.NET with Visual Studio 2005
12. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
13. Martin, A and Luca, C. (2005). *A Theory of Objects*.
14. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
15. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
16. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
17. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
18. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
19. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
20. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 4 C# INDEXERS

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
    3.1 C# Indexer
    3.2 One-dimensional Indexer
    3.3 Creating a New Array-Like Class
    3.4 Adding Class Variables
    3.5 Adding the Constructor
    3.6 Adding the Indexer
        3.6.1 Testing MyArray Class
        3.6.2 Creating a Multi-dimensional Indexer
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

This unit describes indexers in C#, stating the syntax for declaring the common types of indexers. You would require these syntax in C# programming, so do take note of them.

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Define the term “C# Indexer”
- Describe the common types of indexers
- State the syntax for declaring one-dimensional indexers
- Identify the roles of the parts of the syntax for declaring indexers
- State the syntax for declaring two-dimensional indexers

3.0 MAIN CONTENT

3.1 C# Indexers

Defining a C# indexer is much like defining properties. We can say that a C# ***indexer*** is a member that enables an object to be indexed in the same way as an array.

```
 <modifier> <return type> this [argument list]
 {
  get
  {
   // Get codes goes here
  }
  set
  {
   // Set codes goes here
  }
 }
```

where the modifier can be ***private***, ***public***, ***protected*** or ***internal***. The return type can be any valid C# types. The 'this' is a special keyword in C# to indicate the object of the current class. The formal-argument-list specifies the parameters of the indexer. The formal parameter list of an indexer corresponds to that of a method, except that at least one parameter must be specified, and that the ref and out parameter modifiers are not permitted. Remember that indexers in C# must have at least one parameter. Other wise the compiler will generate a compilation error.

The following program shows a C# indexer in action

```
using System;
using System.Collections;

class MyClass
{
 private string []data = new string[5];
 public string this [int index]
 {
  get
  {
   return data[index];
  }
  set
  {
   data[index] = value;
  }
 }
}


class MyClient
{
 public static void Main()
 {
  MyClass mc = new MyClass();
  mc[0] = "Ayomide";
  mc[1] = "A3-126";
  mc[2] = "Epe";
  mc[3] = "Ikeja";
  mc[4] = "Badagry";

Console.WriteLine("{0},{1},{2},{3},{4}",mc[0],mc[1],mc[2],mc[3],m
c[4]);
 }
}
```

SELF ASSESSMENT EXERCISE

Define the term “C# Indexer”

____
____
____
____

3.2 One-dimensional Indexer

The simplest version of an indexer is the one-dimensional type. A one-dimensional indexer accepts a single value between the square brackets when used. The standard syntax used to declare the indexer is similar to that used to define the get and set accessors of a property. However, instead of defining a property name, the accessors are declared for *this[]* as follows:

```
public data-type this[index-type index-name]
{
   get {}
   set {}
}
```

In the syntax definition, *data-type* determines the type of information that will be returned when the indexer is queried and the type that will be required when setting a value. *Index-type* specifies the data type of the indexer itself. This permits declaration of indexers that are not based upon integer values, allowing similar functionality to that of a Hashtable for example. The *index-name* is the variable containing the index value that can be used during processing of the get and set accessors.

The get accessor is required for an indexer and must return a value of the type *data-type*. The set accessor is defined for writeable indexers and is omitted if a read-only variant is desired.

3.3 Creating a New Array-Like Class

To demonstrate the use of an indexer, in this unit we will create a new class that behaves like a simple array of string variables. Unlike a standard array that only permits zero-based indexing, the new class will provide an integer-based array for which the programmer can specify the upper and lower boundaries using a constructor during instantiation.

To start, create a new console application named "IndexerDemo" and add a new class file named "MyArray".

3.4 Adding the Class Variables

The new array-like class requires three private variables. Two integer values will hold the upper and lower boundaries. An array of strings will also be required to store the items added to the MyArray class. This will be a zero-based array with the same length as the created MyArray object. The indexer will interpret the index number supplied and map it against this underlying array for get and set operations.

To add the class level variables, add the following code to the MyArray class' code block:

```
int  _lowerBound;
int  _upperBound;
string[] _items;
```

3.5 Adding the Constructor

The constructor for the new class will accept two integer parameters that define the upper and lower boundaries. These values will be stored in the two associated class variables. Using these boundaries the length of the underlying array can be calculated and the _items array can be initialised accordingly.

To create the constructor, add the following code to the class:

```
public MyArray(int lowerBound, int upperBound)
{
   _lowerBound = lowerBound;
   _upperBound = upperBound;
   _items = new string[1 + upperBound - lowerBound];
}
```

3.6 Adding the Indexer

Now that the preparation work is complete we can add the indexer to the class. For this simple array-like class the indexer accepts a single integer parameter containing the index of the string that is being read from or written to. This index needs to be adjusted to correctly map to the underlying data before returning the value from the array or writing the new value into the array.

The code to add the indexer is shown below. Note that, as with property declarations, the set accessor uses the 'value' variable to determine the value that has been assigned by the calling function:

```
public string this[int index]
{
   get
   {
      return _items[index - _lowerBound];
   }
   set
   {
      _items[index - _lowerBound] = value;
```

```
    }
}
```

3.6.1 Testing the MyArray Class

The new class can be tested using the Main method of the program. Open the Program class and add the following code to create a new instance of MyArray and to populate it with values.

```
static void Main(string[] args)
{
    MyArray fruit = new MyArray(-2, 1);
    fruit[-2] = "Apple";
    fruit[-1] = "Orange";
    fruit[0] = "Banana";
    fruit[1] = "Blackcurrant";
    Console.WriteLine(fruit[-1]);       // Outputs "Orange"
    Console.WriteLine(fruit[0]);        // Outputs "Banana"
}
```

3.6.2 Creating a Multidimensional Indexer

Indexers are not limited to a single dimension. By including more than one index variable in the square brackets of the indexer declaration, multiple dimensions may be added. For example, to declare a two-dimensional indexer the syntax is as follows:

```
public data-type this[index-type1 index-name1, index-type2 index-
name2]
{
    get {}
    set {}
}
```

4.0 CONCLUSION

In this unit, we were made to understand the following: that a C# ***indexer*** is a member that enables an object to be indexed in the same way as an array, a one-dimensional indexer accepts a single value between the square brackets. We also learnt that by including more than one index variable in the square brackets of the indexer declaration, multiple dimensions may be added.

5.0 SUMMARY

In this unit, we defined C# indexer and described the common types of indexers. We equally discovered the syntax for declaring one-dimensional and two-dimensional indexers. We hope you enjoyed your studies. Let us attempt the questions below.

6.0 TUTOR MARKED ASSIGNMENT

Outline the procedure for creating a multidimensional indexer

7.0 REFERENCES/FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.
7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Joyce, F. (2006). Microsoft Visual C#.NET with Visual Studio 2005
12. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.

13. Martin, A and Luca, C. (2005). *A Theory of Objects*.
14. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
15. Michael Lee Scott (2006). *Programming language pragmatics*, (2nd Edition) p. 470
16. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
17. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
18. Schreiner, A. (1993). *Object oriented programming with ANSI-C*.
19. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
20. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, Vol. 3, issue 3, p.14.

UNIT 5 C# INHERITANCE AND POLYMORPHISM

CONTENTS

1.0 Introduction
2.0 Objectives
3.0 Main Content
3.1 Inheritance
3.2 Types of Inheritance
3.2.1 Single Inheritance
3.2.2 Multiple Inheritance
3.3 Demonstrating Inheritance
3.3.1 Procedure for Demonstrating Inheritance
3.3.2 Instantiating and Testing a New Class
3.4 Polymorphism
3.5 Polymorphism and Inheritance
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignment
7.0 References/Further Readings

1.0 INTRODUCTION

This last unit provides an overview of two object-oriented programming concepts. It begins with the discussion of inheritance, its' uses and types. It then proceeds to explain the concept of polymorphism and its' relationship with inheritance. Enjoy your studies!

2.0 OBJECTIVES

At the end of this unit, you should be able to:

- Explain the concept of inheritance
- List and describe the 2 types of inheritance
- Give the procedure for demonstrating inheritance
- Explain the concept of polymorphism
- Identify the relationship between polymorphism and inheritance

3.0 MAIN CONTENT

3.1 Inheritance

Inheritance is a powerful object-oriented concept that permits the creation of hierarchical groups of classes that share common functionality.

Using inheritance, one class can be derived from another. The derived class, also known as the *child* class or *subclass*, inherits functionality from the *base class,* also known as the *parent* class or *superclass*. The subclass can add methods and properties or modify the functionality that it has inherited to provide a more specialised version of the base class.

Using inheritance, classes become grouped together in a hierarchical tree structure. The more generalised classes appear at the root of the hierarchy with the more specific classes appearing on the tree's branches. This categorisation of classes into related types is why inheritance is sometimes referred to as *generalisation* (or generalization).

When using inheritance, all of the base class' methods, properties, events, indexers, operators and some internal variables are all automatically provided to every subclass. As this functionality is automatically available in the child class, there is no need to duplicate code. This reduces maintenance problems, as changes need only to be made in one class, rather than throughout a series of related classes.

SELF ASSESSMENT EXERCISE

Inheritance can also be referred to as generalisation. True or False? Give reasons for your answer.

3.2 Types of Inheritance

There are two types of inheritance used in modern programming languages, they are:

- **Single Inheritance**
- **Multiple Inheritance**

3.2.1 Single Inheritance

C# and other .NET languages use *single inheritance*. This means that a subclass may only inherit functionality from a single base class.

3.2.2 Multiple Inheritance

Multiple inheritance permits a subclass to have two or more superclasses. In this situation the derived class inherits functionality from several base classes. Multiple inheritance is not supported by C# or the .NET framework. However, this does not stop a class from providing many public interfaces as will be seen in a later unit.

3.3 Demonstrating Inheritance

To demonstrate the use of inheritance with real-world objects, during the rest of this unit we will create an example project based around the Vehicle class. This general class will provide methods and properties that all vehicles provide. We will then create subclasses of Vehicle with more specialised functionality.

3.3.1 Procedure for Demonstrating Inheritance

To begin, create a new console application named "InheritanceDemo". Create a class file named "Vehicle" and add the following code to the new class to provide a Speed property and Accelerate and Decelerate methods to all vehicles:

```
private int _speed;     // Miles per hour

public int Speed
{
    get
    {
        return _speed;
    }
}

public void Accelerate(int mph)
{
    _speed += mph;
}
ssss
public void Decelerate(int mph)
{
    _speed -= mph;
}
```

3.3.2 Instantiating and Testing a New Class

The new Vehicle class can be instantiated and tested in its own right. To ensure the class is working correctly, we can modify the Main method of the program to test the methods and the property:

```
static void Main(string[] args)
{
    Vehicle v = new Vehicle();
    Console.WriteLine("Speed: {0}mph", v.Speed);    // Outputs "Speed
0mph"
    v.Accelerate(25);
    Console.WriteLine("Speed: {0}mph", v.Speed);    // Outputs "Speed
25mph"
    v.Decelerate(15);
    Console.WriteLine("Speed: {0}mph", v.Speed);    // Outputs "Speed
10mph"
}
```

3.4 Polymorphism

Another key concept of object-oriented programming is *polymorphism*. This is the ability for an object to change its behaviour according to how it is being used. This is an important part of inheritance and means that a subclass can be used as if it were an instance of its base class. If, for example, you were to create a class with generic functionality for processing the control of vehicles, and you created a more specialised subclass for cars, the Car objects could be passed to methods that expected a Vehicle object. The method would operate on the using the public interface of the Vehicle class. However, the underlying functionality of the Car class would be used. The same routine could be used to process bicycles, buses and other vehicle types.

This type of polymorphism relies heavily on the *encapsulation* principle. As the method using the Vehicle object is aware only of the Vehicle class' public interface and not its internal workings, it is easy to substitute a Car object because cars are just a special type of vehicle.

3.5 Polymorphism and Inheritance

Polymorphism is an important object-oriented programming concept. Polymorphism is the ability for an object to change its public interface according to how it is being used. When using inheritance, polymorphism is achieved when an object of a derived class is substituted where an instance of its parent class is expected. This uses a process known as *upcasting*. With upcasting, the object of the more specialised class is implicitly cast to the base class type required.

4.0 CONCLUSION

To wrap up, recall the following: ***inheritance*** is a powerful object-oriented concept that permits the creation of hierarchical groups of classes that share common functionality; the object of the more specialised class is implicitly cast to the base class type required through the process of ***upcasting***; polymorphism is achieved when an object of a derived class is substituted where an instance of its parent class is expected. Remember that with more practise, you will acquire skills for advanced object-oriented programming using C#. All the best!

5.0 SUMMARY

In this unit we discovered the concept of inheritance and polymorphism. We also spotted the relationship between the two concepts. Hope you enjoyed this course. Let us now attempt the questions below.

6.0 TUTOR MARKED ASSIGNMENT

List and explain the two types of inheritance employed in modern programming.

7.0 REFERENCES.FURTHER READINGS

1. Abelson, H and Gerald J. S. (1997). *Structure and Interpretation of Computer Programs*. The MIT Press.
2. Armstrong, Deborah J. (2006). "The Quarks of Object-Oriented Development". *Communications of the ACM* **49** (2): 123–128. http://portal.acm.org/citation.cfm?id=1113040. Retrieved 2006-08-08.
3. Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley.
4. Date, C. J and Hugh, D. (2006). Foundation for Future Database Systems: The Third Manifesto (2nd Edition)
5. Date, C. J and Hugh, D. (2007). Introduction to Database Systems: The Sixth Manifesto (6th Edition)
6. Eeles, P and Oliver, S. (1998). *Building Business Objects*. John Wiley & Sons.

7. Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
8. Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons.
9. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley.
10. John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, p.278
11. Joyce, F. (2006). Microsoft Visual C#.NET with Visual Studio 2005
12. Kay, Alan. *The Early History of Smalltalk*. http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.
13. Martin, A and Luca, C. (2005). *A Theory of Objects*.
14. Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall.
15. Michael Lee Scott (2006). *Programming language pragmatics,* (2nd Edition) p. 470
16. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press.
17. Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall.
18. Schreiner, A. (1993). *Object oriented programming with ANSI-C.*
19. Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons.
20. Trofimov, M. (1993) *OOOP - The Third "O" Solution: Open OOP.* First Class, OMG, Vol. 3, issue 3, p.14.