C# - Console Applications

Study Notes

Owner:	Satish Talim
File:	C# - Console Applications
Last saved:	7/6/2001
Email Id.	medunet@vsnl.com
Version	ß3

Notice:

This documentation is an early release of the final Study Notes, which may change substantially prior to final release, and is information of Satish Talim.

This document is provided for informational purposes only and Satish Talim makes no warranties, either express or implied, in this document. Information in this document is subject to change without notice.

The entire risk of the use or the results of the use of this document remains with the user. Complying with all applicable international copyright laws is the responsibility of the user.

Microsoft, Windows, Visual Basic, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names mentioned herein may be the trademarks of their respective owners.

Table of Contents

1. Introduction

1.1 A New Platform?

The crash of the dotcoms and the downturn in the prospects of Java notwithstanding, Microsoft's .NET and C# opens a whole New World of possibilities. There are many people in the industry, which see a great future to the .NET. We now have .NET, a new and exciting technology that finally addresses issues we've been grappling with for years (for example, multi-language development environments, the deployment and versioning problems of large, complex systems, and so on).

1.2 System Requirements

To build and run your C# programs, you will need Windows 2000, IE 5.5, Microsoft .NET SDK (Beta 1), an optional Visual Studio .NET (available for MSDN Universal Subscribers) and an optional free C# IDE – Sharp Develop.

The C# IDE is available from http://www.icsharpcode.net/OpenSource/SD/Download/default.asp . The .NET SDK can be downloaded from http://msdn.microsoft.com/net It's an 80MB download. The C# compiler (csc.exe) currently ships with the .NET beta. Do not get confused by this, because as of today we have the Beta 1 version of .NET, the PDC version and the pre Beta 2 version.

The use of Visual Studio.NET has been intentionally avoided, to focus on the language and runtime environment and not place any restrictions on your particular development environment. Therefore, all the sample codes in these notes will compile and run from the command line.

1.3 Purpose of these Study Notes

To learn the C# language, I had to refer to all available material – books, articles on the net, discussion groups. As I studied the C# language, I kept making notes. I felt the need of a single repository where individuals could go to and know everything about a particular topic in C# – this repository became the Study Notes. Please appreciate, that this is not a book on C#.

1.4 Who can use these Study Notes?

These Study Notes supplement my C# lectures and assume a basic knowledge of Java. However, I am confident that it would prove useful to all those interested in learning the C# language.

1.5 Updates to this document

The latest version of this document in .pdf format is available at the URL:

http://www.pune-csharp.com/downloads/index.htm

All feedback / suggestions about this document can be sent to medunet@vsnl.com

1.6 Recommended Sites on C#

Amit Karmakar (based in Sydney, Australia) and I have launched a site

http://www.pune-csharp.com

which will cater to all the technical needs of the C# professionals.

Here are some other sites related to C# -

http://msdn.microsoft.com/net/

http://www.thecodechannel.com/

http://www.csharphelp.com/

http://www.brinkster.com/Forums/

http://www.c-sharpcorner.com/

http://www.csharpindex.com/

http://www.csharp-station.com/

http://www.c-point.com

http://www.devx.com/dotnet/resources/

http://www.aspwire.com

http://www.asptoday.com

1.7 My Workshops on C#

Here are some brief details of the various workshops on the C# language -

- Console Applications
- ADO.NET
- WinForms
- WebForms
- ASP.NET
- Web Services
- Window / Web Controls and Components

1.8 Satish Talim?

Is a software professional with over 23 years of software consulting experience. He has been working on Java-based software-export projects and corporate training in Java since 1995. His clients include Satyam Ltd., KPIT, IBM ACE, Focus Inc., Kanbay Software Ltd., Global Electronic Commerce Services amongst many others. He is also a certified instructor at the Sun Authorized Java Training center in Pune. He was the founder Director of Infonox Software Pvt. Ltd. (ISPL) based in California, USA. His current area of interest is C#.

1.9 Acknowledgements

These Study Notes would not have been possible without the contribution, support and generous help of many individuals.

I would like to acknowledge the help, right from my Java days, to my student and friend **Amit Karmakar**, who always believes in me and inspires me to go on to greater heights. Amit is based in Sydney, Australia and is a Web Developer for the Department of Education and Training, in New South Wales, Australia.

Sunil Kelkar, who has always stood by me and helped me in going through these notes and suggest changes. Sunil, is an independent consultant and works in Java and C#, in Pune.

I want to thank **Rahul Waghmare,** who made me look at and explore C# ! Rahul, is a graphics artist by profession; works in Pune and likes to experiment with various computer languages – Java, VC++, C#.

To all *my students* and *people* on the Internet who provided me with information/feedback, a big thank-you.

I acknowledge all the authors, whose works I have referred. Here is a partial list.

The *C# Programming* book from Wrox Press was the first book I read on C#. I have been highly influenced by this book and it probably reflects in these Study Notes.

Ben Albahari, author of *C# Essentials* for clarifying my doubt on byte array. The book gives a very precise and to the point description of every element in the language.

Tom Archer, author of *Inside C#* for his valuable tips and help in clarifying many of my doubts in C#. His book helped me understand the internal architecture of C#.

Anders Hejlsberg and **Scott Wiltamuth** for their excellent work, *C# Language Reference.*

Ashish Banerjee, **Willy Denoyette**, **Sunil Gudipati**, **Shafeen Sarangi**, **Chris Maunder**, **Mahesh Chand**, **Sudhakar Jalli**, **Pramod Singh** and **Saurabh Nandu** for their excellent articles on various aspects of C#.

Finally to **Microsoft Corporation** for giving us this C# language.

2. C# Program Elements

C# (pronounced "C sharp") is a simple, modern, object oriented, and type-safe (a reference (when not *null*) is always guaranteed to point to an object that is of the type specified and that has already been allocated on the heap. Also take note of the fact that a reference can be *null*) programming language derived from C and C++. C# aims to combine the high productivity of Visual Basic (C# is a Rapid Application Development Language like Visual Basic. It supports the Drag- Drop and Build features of Visual Basic), the elegance of Java and the raw power of C++.

2.1 Overview of the .NET

.NET shifts the focus in computing from a world in which individual devices and Web sites are simply connected through the Internet to one in which devices, services, and computers work together to provide richer solutions for users.

The .NET infrastructure comprises the .NET Framework, Microsoft Visual Studio.NET, the .NET Enterprise Servers, and Microsoft Windows.NET. The .NET infrastructure refers to all the technologies that make up the new environment for creating and running robust, scalable, distributed applications. The part of .NET that lets us develop these applications is the .NET Framework.

The .NET Framework consists of the:

- Common Language Runtime (CLR). Think of the CLR as the virtual machine / runtime environment in which .NET applications function. The code that runs within the CLR runs in an encapsulated and managed environment, separate from other processes on the machine.
- .NET Framework class libraries sometimes called the Base Class Library (BCL). All .NET languages have the .NET Framework class libraries at their disposal. The .NET Framework class libraries include support for everything from file I/O and database I/O to XML and SOAP. The .NET Framework class libraries are very vast. The BCLs functionality is available to all languages that use the CLR.
- universal type system called the .NET Common Type System (CTS). In addition to defining all types, the (CTS) also stipulates the rules that the CLR follows with regard to applications declaring and using these types. In the world of .NET and C# everything in the CTS is an object. In fact, not only is everything an object but, even more importantly, all objects implicitly derive from a single base class defined as part of the CTS. This base class called *System.Object.*
- NGWS software development kit defines a "Common Language Subset" (CLS), that ensures seamless interoperability between CLS-compliant languages and class libraries. For C# developers, this means that even though C# is a new language, it has complete access to the same rich class libraries that are used by seasoned tools such as Visual Basic and Visual C++. C# itself does not include a class library. Because the same .NET base class library is shared between all programming languages, a developer can take his knowledge of this library with him as he migrates from language to language.

C# is a programming language for developing applications for the Microsoft's .NET development platform. C# is provided as a part of Microsoft .NET. In addition to C#, .NET supports Visual Basic, Visual C++, and the scripting languages VBScript and Jscript (infact 21 languages so far). All of these languages provide access to the Next Generation Windows Services (NGWS) platform (or the new Microsoft .NET platform). A developer picks the .NET language that he/she likes most, writes components in it, and shares the compiled binary versions of his/her components with developers using other .NET languages. This makes the .NET platform language-neutral – i.e. modules written in C# would also be compatible with those written in VC++ and VB. When executed, the components use the .NET runtime component for security and memory management.

The basic premise is quite simple: all .NET programs are compiled to an intermediate language (IL), rather than to native code which can be understood by the computer's processor. This IL code is then compiled to native code either when the application is installed, or when the application is run.

Microsoft has submitted a C# standard for ratification by ECMA (the European Computer Manufacturers' Association). Once this standard is in place, vendors worldwide will be able to use it to develop C# compilers that target their own platforms. C# applications cannot run without the .NET runtime. As present the .NET platform has been released for Win NT/2000. Until the time a separate runtime environment is released for .NET, you will have to install the Full .NET Software Development Kit (SDK) on every machine you can to run your programs.

MS plans to release the .NET runtime for other platforms soon. Now if a .NET runtime has been released for your platform then all the .NET programs will run on your platform. A .NET Platform for Linux is expected soon. As of today, Windows is the only platform for which an IL runtime has been developed so far, but this situation is expected to change. Even if Microsoft Corporation doesn't produce the Macintosh, UNIX, and IBM runtimes, someone else probably will in order to enjoy the reduced development costs associated with platform independence. A threat to Java?

2.2 C# and Java

There's no doubt that C# resembles Java. Both languages, for example, promote the grouping of classes, interfaces and implementation together in one file so that developers can edit the code more easily. Both handle objects in much the same way: via references. There are parallels between Java's JVM and C#'s CLR – both are in charge of interpreting bytecode. Both have garbage collectors. Both are in charge of authenticating code before it's executed. Additionally, some of the classes and namespaces inside the .NET class library are very similar to classes and namespaces in the Java class library.

Some differences though: C# uses operator overloading, type-safe enumeration. The C# compiler has an option to automatically produce XML-formatted code documentation using a special comment syntax.

The one edge that Java has over C# is platform independence. However, as mentioned earlier, it's speculated that Microsoft would release the CLR for platforms other than Windows.

2.3 Our first C# Program – Hello, world

Before you write your first C# application, you need to choose an editor – I suggest Sharp Develop (details of which are mentioned in Section 1.2). We'll now write the "Hello, world" example application to get to know the basic syntax and structure of writing C# applications. Program Hello.cs

```
using System;
class Hello {
  public static void Main(string[] args) {
    Console.WriteLine("Hello, world");
  }
}
```

The default file extension for C# programs is .cs, as in hello.cs. The name of the program can be hello.cs or any name you want. In fact, you can name your program as Hello.txt too! (Java programmers should note this).

An important note by Tom Archer, Author Inside C#:

“Most people would recomend that you use the .cs extension for three reasons:

1. The extension makes it obvious to anyone browsing a particular folder that the file is C# source code.
2. The Visual Studio File Open filter for C# files is set to look for (and therefore, display) files with a .cs extension. Having source code files with any other extension would require you to always change that filter (or update the Registry setting for Visual Studio).
3. When you install Visual Studio.NET, an automatic file association is created for the .cs files such that they appear in the Windows Explorer as "C# Source files" and when you open them, they are automatically opened with Visual Studio. Once again, using a different extension would require you to manually create the association and in the case of a .txt file, that might not be what you want.

With regards to the C# compiler allowing the use of files that don't have the .cs extension, this is simply the compiler team allowing you the freedom to use whatever extension you wish.”

We would be using the C# command-line compiler (csc.exe) throughout these notes. This provides two benefits. First, it means that no matter what environment you're using, the steps for building the programs will always work. Second, learning the different compiler options will help you in the long run in situations where your editor doesn't provide you complete control over this step.

Such a program can be compiled with the command line directive

```
csc hello.cs
```

which produces an executable program named `hello`.exe.

If you haven't made any typos. Something like:

```
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
```

will appear in your command prompt window. The C# compiler produces an executable program named `hello`.exe. C# compiles straight from source code to executable (.exe), with no object files. By simply typing **hello** you will see the output of the program as:

```
Hello, world
```

Close examination of this program is illuminating:

- C# programs (Note: program code only) are case-sensitive and in free format.
- Console (character only and graphics free) applications are normally used for creating driver programs for components that you are testing and a great way to start learning C#. It also means that we are not doing anything specific to Windows development.
- When you define a class in C#, you must define all methods inline—header files do not exist. Why is this a good thing? It enables the writers of classes to create highly mobile code, which is one of the key concepts of the .NET environment. This way when you write a C# class, you end up with a fully encapsulated bundle of functionality that you can easily drop into any other development environment without worrying how that language processes include files or if it has a mechanism for including files within files. Using this "one-stop programming" approach, you can, for instance, take an entire class and drop it into an Active Server Pages (ASP) page and it will function just as it would if it were being compiled into a desktop Windows application!

- The using System; directive references a namespace (similar to a package in Java) called System that is provided by the .NET runtime. System is the root of the .NET base class namespace. using is very similar in concept to Java's import keyword. All it does is to tell the compiler where it can look for unresolved class names. A "using" directive enables unqualified use of the members of a namespace. This namespace contains the Console class referred to in the Main method. Namespaces provide a hierarchical means of organizing the elements of a class library. Unlike Java, in C# you cannot import a single class, but the whole package. However, the components of a namespace name do not have to map onto directories (remember in Java, a package has to physically map to a directory). The "Hello, world" program uses Console.WriteLine as a shorthand for System.Console.WriteLine. What do these identifiers denote? System is a namespace, Console is a class defined in that namespace, and WriteLine is a static method (a static method can access any static member within the class, but it cannot access an instance member) defined in that class.
- In Main() the first character is a capital M.
- The Main function is a static member of the class Hello. Functions and variables are not supported at the global level; such elements are always contained within type declarations (e.g., class and strict declarations). The Main function can have any access modifier and can be written as static void Main() too. Every method (Main method here) must have a return type. In this case, it is "void", which means that "Main" does not return a value. Every method also has a parameter list following its name with zero or more parameters between parenthesis. The Main method is a static member of the class Hello. The static qualifier makes the Main() method a class method, so that it can be invoked on its own, without the creation of an instance of the class.
- Every C# application must have a method named *Main* defined in one of its classes. It doesn't matter which class contains the method—you can have as many classes as you want in a given application—as long as one class has a method named *Main*. It is the entry point of your program, where the program control starts and ends. It is declared inside a class or struct. It must be static. It can either be **void** or return an **int**. The Main method is the place where you create objects and execute other methods. The Main method can be written without parameters or with parameters.
- The Main method can be written without parameters or with parameters. The latter form allows your program to read command-line arguments. There are three ways to declare the Main method: (a) It can return void as in public static void Main() { ... } (b) It can also return an int as in public static int Main() { ...;   return 0; } (c) It can also take arguments as in public static int Main(string[] args) { ...; return 0; }
- The parameter of the Main method is a string array that represents the command-line arguments used to invoke the program. Notice that, unlike C, C++, this array does not include the name of the executable (exe) file.
- If you have zero or more than one Main() in a program, you can expect compiler errors. This is shown in the example ManyMain.cs below:

```
using System;
class GF {
  public GF() {
    Console.WriteLine("In GF class");
  }
  public static void Main(string[] args) {
  }
}
```

```
class F : GF {
  public F() {
    Console.WriteLine("In F class");
  }
  public static void Main(string[] args) {
  }
}
class S : F {
  public S() {
    Console.WriteLine("In S class");
  }
  public static void Main(string[] args) {
    S son = new S();
  }
}
```

This program when compiled gives the error:

```
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]

Copyright (C) Microsoft Corp 2000. All rights reserved.

Test.cs(7,22): error CS0017: Program 'Test.exe' has more than one entry point defined: 'GF.Main(string[])'

Test.cs(16,22): error CS0017: Program 'Test.exe' has more than one entry point defined: 'F.Main(string[])'

Test.cs(26,22): error CS0017: Program 'Test.exe' has more than one entry point defined: 'S.Main(string[])'
```

The designers of C# included a mechanism by which you can define more than one class with a *Main* method. Why would you want to do that? One reason is to place test code in your classes. You can then use the

/main:< className >

switch with the C# compiler to specify which class's *Main* method is to be used.

To compile this application such that the S.Main method is used as the application's entry point, you'd use this switch:

```
csc MultipleMain.cs /main:S
```

Changing the switch to */main:F* would then use the *F.Main* method.

The ***/main*** compiler option specified a class in which to look for a Main method. However, the **Main** method has to be defined as follows:

```
public static void Main(){ }
```

- The “Hello, world” output is produced using a class library. C# does not itself provide a class library. Instead, C# uses a common class library that is also used by other languages such as Visual Basic and Visual C++.
- The program does not contain forward declarations. Forward declarations are never needed in C# programs, as declaration order is not significant.
- C# programs use “.” as a separator in compound names such as Console.WriteLine.
- The program does not use #include to import program text. Dependencies between programs are handled symbolically rather than with program text. This system eliminates barriers between programs written in different languages. For example, the Console class could be written in C# or in some other language.

Note: Observe the following -

a. Save the above Hello, world program as Hello.cs and compile as csc Hello.cs and run the program as Hello

b. Compile the above program as csc hello.cs and run the program as hello or Hello

c. Save the above file as hello.cs and compile as csc Hello.cs or csc hello.cs and run the program as Hello or hello

d. Save the above file as rubbish.cs and compile as csc rubbish.cs and run the program as rubbish

e. Make the class as public and try a, b and c as above

For a to e above we get the same output i.e. Hello, world

Compiler Errors:

Here is what to expect when the compiler encounters syntax errors in your code. First, you'll see the name of the current file being compiled, followed by the line number and column position of the error. Next, you'll see the error code as defined by the compiler—for example, *CS0234*. Finally, after the error code, you'll see a short description of the error. Many times, this description will give you enough to make the error clear. You can also search for the error code in the .NET Framework SDK Documentation (which is installed with the .NET Framework SDK) for a more detailed description.

2.4 Naming Guidelines

Choosing a stable and easily understood naming convention will enable you to write code that's easier to read and, therefore, to maintain. A standard is still evolving. It might end up slightly different from what is presented here, but this will at least give you a place to start.

2.4.1 Namespaces

Use your company or product name, and employ mixed casing with an initial capital letter—for example, *JavaTech*. If you're in the business of selling component software, create a top-level namespace that is the same as your company name. Then for each product, create a nested namespace with its embedded types, which will prevent name collision with other products. An example of this can be found in the .NET Framework SDK: *Microsoft.Win32*. This strategy might result in more long-winded names, but remember that the users of your code need only specify the *using* directive to save typing. If your company is called Javatech, and you sell two products—training and a software development—name your namespaces *JavaTech.Training* and *JavaTech.SoftwareDevelopment*.

2.4.2 Classes

Because objects are supposed to be living, breathing entities that have abilities, name classes by using nouns that describe the class's problem domain. In cases where the class is more generic (that is, less specific to the problem domain) than that—for example, a type that represents an SQL string—use Pascal casing - where the first character is capitalized.

2.4.3 Methods

Use Pascal casing - where the first character is capitalized - on all methods. Methods are meant to act—they carry out work. Therefore, let the names of your methods depict *what* they do. Examples of this are *PrintInvoice* and *OpenDatabase*.

2.4.4 Method Arguments

Use Pascal casing (where the first character is capitalized) on all arguments. Give meaningful names to arguments.

2.4.5 Interfaces

Use Pascal casing (where the first character is capitalized) on all interfaces. It's common to prefix interface names with a capital "I"—for example, *IComparable*. A common technique is naming interfaces with adjectives.

2.4.6 Class members

Use camel casing, in which the first letter is not capitalized. That way if you have a method that takes as an argument something called *Foo*, you can differentiate it from the internal representation of that variable by creating the internal member named *foo*.

2.5 Automatic memory management

Manual memory management requires developers to manage the allocation and de-allocation of blocks of memory. Manual memory management is both time consuming and difficult. C# provides automatic memory management so that developers are freed from this burdensome task. In the vast majority of cases, this automatic memory management increases code quality and enhances developer productivity without negatively impacting either expressiveness or performance.

Once a variable is assigned null, it become eligible for garbage collection. The .NET Garbage Collector (GC) is permitted to clean up immediately, but is not required to do so.

Hence, once you finish with an object, that object no longer has any live references to it (i.e. it won't be assigned to any variables or stored in any arrays). C# has a garbage collector that looks for unused objects and reclaims the memory that those objects are using. You don't have to do any explicit freeing of memory; you just have to make sure that you are not holding onto an object you want to get rid of.

Note:

The Finalize (inherited from Object) method allows an Object to attempt to free resources and perform other cleanup operations before the **Object** is reclaimed by the Garbage Collector (GC). This method may be ignored by the Common Language Runtime; therefore, necessary cleanup operations should be done elsewhere.

2.6 Comments

The first line contains a comment:

// A "Hello World!" program in C#

Text following a comment // up to the end of the line is ignored by the compiler.

// Comments can appear on an independent line or as part of a statement too.

You can also comment a block of text by placing it between the characters /* and */, for example:

/* A "Hello World!" program in C#.

This program displays the string "Hello World!" on the screen. */

The C-style comment can occur within a line and can span more than one line.

// This line is commented out // is extra.

The extra // above, have no effect.

Block comments can't be nested.

Comments are not considered when your program is compiled. They are there to document what your program does in plain English.

In C# you can document the code you write using XML. C# is the only programming language in Visual Studio.NET 7.0 with this feature. In source code files, lines that begin with /// and that precede a user-defined type such as a class, delegate, or interface; member such as a field, event, property, or method; or a namespace declaration can be processed as comments and placed in a file. These will be treated as normal comments by the compiler, unless you use the /doc:filename compiler option to tell it to generate the XML. The compiler also supports a set of documentation tags that you can use within XML comments. The source code file that contains Main is output first into the XML.

The following sample XMLsample.cs program provides a basic overview of a type that has been documented.

```
// XMLDoc\XMLSample.cs
using System;
/// <summary>
/// Class level summary description goes here.</summary>
/// <remarks>
/// Longer comments can be associated with a type or member
/// through the remarks tag</remarks>
public class SomeClass {
  /// <summary>
  /// Store for the name property</summary>
  private string myName = null;
  public SomeClass() {
   // TODO: Add Constructor Logic here
  }
  /// <summary>
  /// Name property </summary>
  /// <value>
  /// A value tag is used to describe the property value</value>
  public string Name {
    get {
      if ( myName == null ) {
        throw new Exception("Name is null");
      }
      return myName;
    }
  }
  /// <summary>
  /// Description for SomeMethod.</summary>
  /// <param name="s"> Parameter description for s goes here</param>
  /// <seealso cref="String">
  /// You can use the cref attribute on any tag to reference
  /// a type or member and the compiler will check that the
  /// reference exists. </seealso>
  public void SomeMethod(string s) {
  }
  /// <summary>
  /// Some other method. </summary>
  /// <returns>
  /// Return results are described through the returns tag.</returns>
  /// <seealso cref="SomeMethod(string)">
  /// Notice the use of the cref attribute to reference a
  /// specific method  </seealso>
  public int SomeOtherMethod() {
    return 0;
  }
  /// <summary>
  /// The entry point for the application.
  /// </summary>
  /// <param name="args"> A list of command line arguments</param>
  public static int Main(string[] args)  {
    // TODO: Add code to start application here
    return 0;
  }
}
```

To compile the sample code, type the following command line:

```
csc XMLsample.cs /doc:XMLsample.xml
```

The /doc option allows you to place documentation comments in an XML file.

This will create the XML file XMLsample.xml. as shown below:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>XMLSample</name>
    </assembly>
    <members>
        <member name="T:SomeClass">
            <summary>
            Class level summary documentation goes here.</summary>
            <remarks>
            Longer comments can be associated with a type or member
            through the remarks tag</remarks>
        </member>
        <member name="F:SomeClass.myName">
            <summary>
            Store for the name property</summary>
        </member>
        <member name="M:SomeClass.SomeMethod(System.String)">
            <summary>
            Description for SomeMethod.</summary>
            <param name="s"> Parameter description for s goes here</param>
            <seealso cref="T:System.String">
            You can use the cref attribute on any tag to reference
            a type or member and the compiler will check that the
            reference exists. </seealso>
        </member>
        <member name="M:SomeClass.SomeOtherMethod">
            <summary>
            Some other method. </summary>
            <returns>
            Return results are described through the returns tag.</returns>
            <seealso cref="M:SomeClass.SomeMethod(System.String)">
            Notice the use of the cref attribute to reference a
            specific method  </seealso>
        </member>
        <member name="M:SomeClass.Main(System.String[])">
            <summary>
            The entry point for the application.
            </summary>
            <param name="args"> A list of command line arguments</param>
        </member>
        <member name="P:SomeClass.Name">
            <summary>
            Name property </summary>
            <value>
            A value tag is used to describe the property value</value>
        </member>
    </members>
</doc>
```

All the members of the assembly are denoted by <member> tags, and you can see how the compiler has added the full name of the member as a name attribute. The T, F and M prefixes denote types, fields and members respectively.

2.7 Blocks

C# code is organised into blocks (or sections). You specify the beginning and end of a block using curly braces.

{ // C# block of code }

Every executable statement in C# will be within one or more blocks. It is a standard C# programming style to identify different blocks with indentation. Every time you enter a new block, you should indent your source code by preferably two spaces. When you leave a block, you should de-indent by two spaces. Blocks define scope (or lifetime and visibility) of program elements.

2.8 Separation

As in C, C# uses the semicolon to indicate the end of a statement.

2.9 Whitespace

C# is a freeform language. You don't have to indent anything to get it to work properly. Whitespace characters are space, tab or newline. Appropriate use of white space makes your programs easier to read and understand.

2.10 Keywords (74)

Because keywords have specific meaning in C#, you can't use them as identifiers for something else, such as variables, constants, class names, and so on. However, they can be used as part of a longer token, for example: public int abstract_int;

Also, because C# is case sensitive, if a programmer is bent on using one of these words as an identifier of some sort, you can use an initial uppercase letter. While this is possible, it is a very bad idea in terms of human readability.

There are numerous Classes defined in the standard packages. While their names are not keywords, the overuse of these names may make your meaning unclear to future people working on your programs.

The keywords are:

```
abstract    base        bool        break       byte
case        catch       char        checked     class
```

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

The break and continue keywords:

These can be used to control iteration. The break statement will exit from the immediately enclosing for, do, while, foreach or switch statement. The continue statement will terminate the current iteration, causing a jump to the end of the block forming the body of the loop. C# does not support labeled break as in Java.

2.11 Constants – const / readonly

A constant is nothing more than a value, in a program, that stays the same throughout the program's execution. Such values as the numeral 2 and the string constant "C#" are sometimes called hard-coded values or literals. Symbolic constants are simply words that represent values in a program, for example:

```
const float SALESTAX = 0.06;
```

const tells C# that this data object is going to be a constant.

```
const int x = 100;
```

```
const int y = x*5;  // OK – compiler knows value of x
```

By default, *const* members are static. A const variable cannot take as its value a variable that is not **const**.

One problem you may encounter is that values of const members are calculated at compile-time, so you can't use const to define a member whose value can't be set in this way. What does a programmer do when the need arises for a field with a value that won't be known until run time and should not be changed once it's been initialized? To get around this, C# has provided a **readonly** modifier, which specifies that the member can have its value set once only, and afterwards is read-only. The readonly fields that we define are instance fields, meaning that the user would have to instantiate the class to use the fields. Hence, when you define a field with the readonly keyword, you have the ability to set that field's value in one place: the constructor. After that point, the field cannot be changed by either the class itself or the class's clients.

In the example below, we are setting the sortcode to the string value passed into the constructor.

```
public class Account {
  public readonly string sortcode;
  public Account(string code) {
    sortcode = code;
  }
}
```

2.12 Variables

Variables are values that can change as much as needed during the execution of a program. One reason you need variables in a program is to hold the results of a calculation. Hence, variables are locations in memory in which values can be stored. They have a name, a type, and a value. C# allows simple and compound variable declarations.

2.13 Naming constants and variables

The rules for choosing constant and variable names (identifiers) are that every C# identifier must begin with one of the characters: A-Z, a-z. Following the first character, the rest of the identifier can use any of these characters: A-Z, a-z, 0-9 An identifier must not clash with a keyword. As a special case, the @ prefix can be used to avoid such a conflict, but the character is not considered part of the identifier that it precedes. For instance, the following two identifiers are equivalent:

```
int

@int
```

The example SI.cs below clarifies this point:

```
using System;
class Tst {
  static void Main(){
    int @int = 20;
    Console.WriteLine("Value is: {0} ", @int);
  }
}
```

A class name or other identifier can be no longer than 512 characters.

C# language uses the Unicode character set. Unicode is a character set definition that not only offers characters in the standard ASCII character set, but also a wide range of other characters for representing most international characters.

Variable definitions can go anywhere in a method definition, although typically variable declarations for a given block are placed immediately after the opening curly brace ({). C# actually have three kinds of variables: instance, class and local variables. C# does not have global variables, as in 'C'.

Instance variables are defined outside the confines of any method declaration. In general, instance variables are available to all methods within the class and are automatically initialised when an object is created. Their initial value depends on the type of variable: null for objects of classes, 0 for numeric variables, '\0' for characters, and false for bool.

Local variables are local to some method, are not automatically initialised and are lost once a method terminates. Your C# program will not compile if you try to use an unassigned local variable. The scope extends to the end of the current block. You can't declare another variable with the same name in the current or any nested blocks.

A static variable belongs to the class itself (not any individual object). No object really owns this variable but any can read or write it. This is useful to provide a means of communication between objects or for storing data that all objects need to reference. No matter how many times a class gets instantiated, only one copy of this variable will exist. In C#, static members must be accessed through a class name. You can't access them via an object, as you can in Java.

2.14 Escape sequences

The \" in a string literal is an example of an escape sequence. The \ character indicates the beginning of an escape sequence and the character or characters that follow are interpreted in a special way. In this example, the escape sequence means that the " mark should be printed and it does not indicate the end of the string literal.

2.15 Statements and Expressions

A C# program is structured into classes, which contain methods, and these in turn are made up of statements. Statements are free format; can run over more than one line without any need for continuation characters; and are terminated with a semicolon. A statement forms a single C# operation. Statements sometimes return values - for example, when you add two numbers together or test to see whether one value is equal to another. Expressions are built by combining variables and operators together into statements. C# statements end with a semicolon. A statement may have a label associated with it. Normally labeled statements are used in switch statements, but in C# they can also be used as the destination for a goto operation.

An **expression statement** evaluates a given expression. The value computed by the expression, if any, is discarded. Not all expressions are permitted as statements. In particular, expressions such as x + y and x == 1 that have no side effects, but merely compute a value (which will be discarded), are not permitted as statements.

The example

```
using System;
class Test {
  static int F() {
    Console.WriteLine("Test.F");
         return 0;
  }
  static void Main() {
         F();
  }
}
```

shows an expression statement. The call to the function F made from Main, constitutes an expression statement. The value that F returns is simply discarded.

2.15.1 Empty statement

An *empty-statement* does nothing.

empty-statement:
;

An empty statement is used when there are no operations to perform in a context where a statement is required. Execution of an empty statement simply transfers control to the end point of the statement. Thus, the end point of an empty statement is reachable if the empty statement is reachable. An empty statement can be used when writing a while statement with a null body:

```
bool ProcessMessage() {...}
void ProcessMessages() {
        while (ProcessMessage());
}
```

Also, an empty statement can be used to declare a label just before the closing “}” of a block:

```
void F() {
        ...
if (done) goto exit;
        ...
exit: ;
}
```

2.16 Types

A C# program is written by building new **types (typically classes)** and leveraging existing types, either those defined in the C# language itself or imported from other libraries. Each type contains a set of data (typically fields) and function members (typically methods), which combine to form the modular units that are the key building blocks of a C# program.

Generally, you must vreate instances of a type to use that type. Those data members and function members that require a type to be instantiated are called instance members. Data members and function members that can be used on the type itself are called static members.

C# supports two major kinds of types (including both predefined types and user-defined types): *value types* and *reference types*. Value types include simple types such as char, int, float etc. **are structs** (simple type actually alias structs found in the System namespace). You can expand the set of simplentypes by defining your own structs and enums. Reference types include class types, interface types, delegate types, and array types.

Value types are allocated on the stack and Reference types are allocated on the heap.

Value types differ from reference types in that variables of the value types directly contain their data, whereas variables of the reference types store references to objects. With reference types, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other. An example SRMEM.cs clarifies this point:

```
// SRMEM.cs
// Reference-type declaration
using System;
class PointR {
  public int x, y;
}
// Value-type declaration
struct PointV {
  public int x, y;
}
class Test {
  static void Main() {
    PointR a; // Local reference-type variable, uses 4 bytes of
              // memory on the stack to hold address
    PointV b; // Local value-type variable, uses 8 bytes of
              // memory on the stack for x and y
    a = new PointR(); // Assigns the reference to address of new
                      // instance of PointR allocated on the
                      // heap. The object on the heap uses 8
                      // bytes of memory for x and y, and an
                      // additional 8 bytes for core object
                      // requirements, such as storing the
                      // object's type & synchronization state
    b = new PointV(); // Calls the value-type's default
                      // constructor.  The default constructor
                      // for both PointR and PointV will set
                      // each field to its default value, which
                      // will be 0 for both x and y.
    a.x = 7;
    b.x = 7;
/*
  }
}
At the end of the method the local variables a and b go out of
scope, but the new instance of a PointR remains in memory until
the garbage collector determines it is no longer referenced
```

```
*/

    // Assignment to a reference type copies an object reference;
    // assignment to a value type copies an object value...
    PointR c = a;
    PointV d = b;
    c.x = 9;
    d.x = 9;
    Console.WriteLine(a.x); // Prints 9
    Console.WriteLine(b.x); // Prints 7
  }
}
```

The process of turning a value type into a reference type is known as *boxing*. Each value type has a corresponding hidden reference type, which is automatically created when it's cast to a reference type.

The process of turning a reference type back into a value type is called *unboxing.*

C# provides a "unified type system", whereby the **object** class is the ultimate base type for both reference and value types. All types – including value types – can be treated like objects. It is possible to call Object methods on any value, even values of "primitive" types such as int. The example

```
using System;
class Test {
  static void Main() {
    Console.WriteLine(3.ToString());
  }
}
```

calls the Object-defined ToString method on a constant value of type int.

The example BUB.cs

```
class BUB {
  static void Main() {
    int i = 123;
    object o = i;       // boxing
    int j = (int) o;    // unboxing
  }
}
```

is more interesting. An int value can be converted to object and back again to int. This example shows both *boxing* and *unboxing*. When a variable of a value type needs to be converted to a reference type, an object *box* is allocated to hold the value, and the value is copied into the box. *Unboxing* is just the opposite. When an object box is cast back to its original value type, the value is copied out of the box and into the appropriate storage location. When unboxing—converting from a reference type to a value type—the cast is needed. This is because in the case of unboxing, an object could be cast to any type. Therefore, the cast is necessary so that the compiler can verify that the cast is valid per the specified variable type.

This type system unification provides value types with the benefits of object-ness, and does so without introducing unnecessary overhead. For programs that don't need int values to act like object, int values are simply 32 bit values. For programs that need int's to behave like objects, this functionality is available on-demand. This ability to treat value types as objects bridges the gap between value types and reference types that exists in most languages.

All types in C# - value and reference – inherit from the Object superclass. The object type is based on **System.Object** in the .NET Framework. You can assign values of any type to variables of type object. All data types, predefined and user-defined, inherit from the **System.Object** class. The object data type is the type to and from which objects are boxed.

Developers can define new value types through enum and struct declarations, and can define new reference types via class, interface, and delegate declarations.

2.17 Predefined types

C# provides a set of predefined types, most of which will be familiar to C and C++ developers.

The table below lists each of the predefined types, and provides examples of each.

Type	Description	Examples
object	The ultimate base type of all other types	Object o = new Stack();
string	String type; a string is a sequence of Unicode characters	String s = "Hello";
sbyte	8-bit signed integral type	Sbyte val = 12;
short	16-bit signed integral type	Short val = 12;
int	32-bit signed integral type	int val = 12;

long	64-bit signed integral type	long val1 = 12;
		long val2 = 34L;

byte	8-bit unsigned integral type	byte val1 = 12;
		byte val2 = 34U;

ushort	16-bit unsigned integral type	Ushort val1 = 12;
		ushort val2 = 34U;

uint	32-bit unsigned integral type	uint val1 = 12;
		uint val2 = 34U;

ulong	64-bit unsigned integral type	ulong val1 = 12;
		ulong val2 = 34U;
		ulong val3 = 56L;
		ulong val4 = 78UL;

float	Single-precision floating point type	float value = 1.23F;
double	Double-precision floating point type	Double val1 = 1.23 double val2 = 4.56D;
bool	Boolean type; a bool value is either true or false	bool value = true;
char	Character type; a char value is a Unicode character	char value = 'h';
decimal	Precise decimal type with 28 significant digits	Decimal value = 1.23M;

C# is a strongly "Typed" language. Thus, all operations on variables are performed with consideration of what the variable's "Type" is. There are rules that define what operations are legal in order to maintain the integrity of the data you put in a variable. This is enforced at compile time.

The conversions between types may be implicit or explicit. Implicit numeric conversions can be performed automatically and are guaranteed to succeed and not lose information. The implicit numeric conversions are:

- From sbyte to short, int, long, float, double, or decimal.
- From byte to short, ushort, int, uint, long, ulong, float, double, or decimal.
- From short to int, long, float, double, or decimal.
- From ushort to int, uint, long, ulong, float, double, or decimal.
- From int to long, float, double, or decimal.
- From uint to long, ulong, float, double, or decimal.
- From long to float, double, or decimal.
- From ulong to float, double, or decimal.
- From char to ushort, int, uint, long, ulong, float, double, or decimal.
- From float to double.

Conversions from int, uint, or long to float and from long to double may cause a loss of precision, but will never cause a loss of magnitude. The other implicit numeric conversions never lose any information.

There are no implicit conversions to the char type. This in particular means that values of the other integral types do not automatically convert to the char type.

Explicit numeric conversions require a cast and runtime circumstances determine whether the conversion succeeds or information is lost during the conversion. An example of this:

```
int x = 123456; // 4bytes
short s = (short)x; //convert to 2 bytes
```

All the types have fixed sizes, and will be the same size on any system. All data types, predefined and user-defined, inherit from the System.Object class.

While the CTS is responsible for defining the types that can be used across .NET languages, most languages choose to implement aliases to those types. There is no advantage to using one technique over the other.

In C#, each of the predefined types is a shorthand (alias) for a system-provided type. For example, the keyword int is shorthand for a struct named System.Int32 by the CTS. The two names can be used interchangeably, though it is considered good style to use the keyword rather than the complete system type name.

The C# type keywords and their aliases are interchangeable. For example, you can declare an integer variable by using either of the following declarations:

```
int x = 123;
System.Int32 x = 123;
```

All the types in the table, except object and string, are referred to as simple types. The predefined value types include signed and unsigned integral types, floating point types, and the types bool, char, and decimal. The signed integral types are sbyte, short, int, and long; the unsigned integral types are byte, ushort, uint, and ulong; and the floating point types are float and double.

The predefined reference types are object and string. The **object** type is based on **System.Object** in the .NET Framework. You can assign values of any type to variables of type **object**. This is in contrast to Java, where the basic types have no object-like behaviour at all, and need to be explicitly converted to and from object wrapper types. In the .NET runtime, a reference-type instance has an eight-byte overhead, which stores the object's type and temporary information such as synchronisation lock state or whether it has been fixed from movement by the garbage collector. Each reference to a reference type instance uses four bytes of storage.

The following sample shows how variables of type **object** can accept values of any data type and how variables of type **object** can use methods on **System.Object** from the .NET Framework.

Example ObjClass.cs

```
using System;
public class MyClass1  {
  public int i = 10;
}
public class MyClass2 {
  public static void Main() {
    object a;
    a = 1;   // an example of boxing
    Console.WriteLine(a);
    Console.WriteLine(a.GetType());
    Console.WriteLine(a.ToString());
    Console.WriteLine();
    a = new MyClass1 ();
    MyClass1 ref_MyClass1;
    ref_MyClass1 = (MyClass1)a;
    Console.WriteLine(ref_MyClass1.i);
  }
}
```

Equals() is a very important method in Object class. The default implementation of **Equals** supports reference equality only, but subclasses can override this method to support value equality instead. In the case of value types, this method returns true if the two types are identical and have the same value. The example ObjEquals.cs clarifies this point.

```
using System;
public class ObjEquals {
  static void Main() {
    Object o  = new Object();
    Object o1 = new Object();
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = "World";
    String s4 = s3;
    int i1 = 1;
    int i2 = 1;
    int i3 = 2;
    if (o == o1)       // false
      Console.WriteLine("Same Object Reference (o and o1)");
    if (o.Equals(o1))  // false
      Console.WriteLine("Same Object Content (o and o1)");
    if (s1 == s2)      // true
      Console.WriteLine("Same Object Reference (s1 and s2)");
    if (s1.Equals(s2)) // true
      Console.WriteLine("Same Object Content (s1 and s2)");
    if (s1 == s3)      // false
      Console.WriteLine("Same Object Reference (s1 and s3)");
    if (s1.Equals(s3)) // false
      Console.WriteLine("Same Object Content (s1 and s3)");
    if (s3 == s4)      // true
      Console.WriteLine("Same Object Reference (s3 and s4)");
    if (s3.Equals(s4)) // true
      Console.WriteLine("Same Object Content (s3 and s4)");
    if (i1.Equals(i2)) // true
      Console.WriteLine("Same Object Content (i1 and i2)");
    if (i1.Equals(i3)) // false
      Console.WriteLine("Same Object Content (i1 and i3)");
  }
}
```

The **string** type represents a string of Unicode characters. string is an alias for **System.String** in the NGWS Framework. The string reference type requires a minimum of 20 bytes of memory. Although string is a reference type, the equality operators (== and !=) are overloaded to compare the values of string objects, not references. This makes testing for string equality more intuitive.

```
string a = "hello";
string b = "hello";
Console.WriteLine( a == b );  // output: True -- same value
Console.WriteLine( (object)a == b ); //  False -- different objects
```

The + operator concatenates strings:

```
string a = "good " + "morning";
```

The [] operator accesses individual characters of a string:

```
char x = "test"[2];  // x = 's';
```

String literals are of type string and can be written in two forms, quoted and @-quoted. Quoted string literals are enclosed in double quotation marks ("):

```
"good morning"   // a string literal
```

and can contain any character literal, including escape sequences:

```
string a = "\\\u0066\n";  // backslash, letter f, new line
```

@-quoted string literals start with @ and are enclosed in double quotation marks. For example:

```
@"good morning"  // a string literal
```

The advantage of @-quoting is that escape sequences are not processed, which makes it easy to write, for example, a fully qualified file name:

```
@"c:\Docs\Source\a.txt"  // rather than "c:\\Docs\\Source\\a.txt"
```

To include a double quotation mark in an @-quoted string, double it:

```
@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

The C# object class is very similar to Java's Object.

```
int n = 5;
string s = n.ToString(); // call object.ToString()
```

One can declare a string as follows:

```
string str = "Hello";
```

If you declare as follows:

```
string str = new string("Hello");
```

you will get a compiler warning.

The **bool** type is used to represent boolean values: values that are either true or false. The bool keyword is an alias of **System.Boolean**. Although Boolean values require only 1 bit (0 or 1), they occupy 1 byte of storage since this is the minimum chunk addressing on most processor architectures can work with. The inclusion of bool makes it easier for developers to write self-documenting code, and helps eliminate the all-too-common C++ coding error in which a developer mistakenly uses "=" when "==" should have been used. In C#, the example

```
int i = ...;
F(i);
if (i = 0)  // Bug: the test should be (i == 0)
  G();
```

is invalid because the expression `i = 0` is of type `int`, and if statements require an expression of type `bool`. No standard conversions exist between bool and other types. In particular, the `bool` type is distinct and separate from the integral types, and a `bool` value cannot be used in place of an integral value, or vice versa.

The **byte** type is an Unsigned 8-bit integer. The byte keyword is an alias of **System.Byte**.

The **sbyte** type is an Signed 8-bit integer. The sbyte keyword is an alias of **System.SByte**.

The **char** type is used to represent Unicode characters. A variable of type char represents a single 16-bit Unicode character. The char keyword is an alias of **System.Char**. Constants of the char type can be written as character literals, hexadecimal escape sequence, or Unicode representation. You can also cast the integral character codes. All of the following statements declare a char variable and initialize it with the character X:

```
char MyChar = 'X';        // Character literal

char MyChar = '\x0058';   // Hexadecimal

char MyChar = (char)88;   // Cast from integral type

char MyChar = '\u0058';   // Unicode
```

The **decimal** type is appropriate for calculations in which rounding errors are unacceptable. Common examples include financial calculations such as tax computations and currency conversions. The decimal type provides 28 significant digits. The decimal keyword is an alias of **System.Decimal**. It occupies 16 bytes of memory. If you want a numeric real literal to be treated as decimal, use the suffix m or M, for example:

```
decimal myMoney = 300.5m;
```

Without the suffix m, the number is treated as a double, thus generating a compiler error.

The **double** keyword denotes a simple type that stores 64-bit floating-point values. Precision is 15-16 digits. The double keyword is an alias of **System.Double**. By default, a real numeric literal on the right-hand side of the assignment operator is treated as double. However, if you want an integer number to be treated as double, use the suffix d or D, for example:

```
double x = 3D;
```

You can use F to denote single precision (i.e. 12.7F). U can be used to denote unsigned literals, and L longs. The precise decimal type is denoted by an M suffix, as in 12.77M.

The **float** keyword denotes a simple type that stores 32-bit floating-point values. Precision is 7 digits. The float keyword is an alias of **System.Single**. By default, a real numeric literal on the right-hand side of the assignment operator is treated as double. Therefore, to initialize a float variable use the suffix f or F, for example:

```
float x = 3.5F;
```

If you don't use the suffix in the previous declaration, you will get a compilation error because you are attempting to store a double value into a float variable.

The **int** keyword denotes an integral type. It's a Signed 32-bit integer. The int keyword is an alias of **System.Int32**.

The **uint** keyword denotes an integral type. It's an unsigned 32-bit integer. The int keyword is an alias of **System.UInt32**. You can declare and initialize a variable of the type uint like this example:

```
uint myUint = 4294967290;
```

When an integer literal has no suffix, its type is the first of these types in which its value can be represented: int, uint, long, ulong. In this example, it is uint. You can also use the suffix u or U, like this:

```
uint myUint = 123U;
```

When you use the suffix U or u, the literal type is determined to be either uint or ulong according to its size. In this example, it is uint.

The **long** keyword denotes an integral type. It's a signed 64-bit integer. The long keyword is an alias of **System.Int64**. You can declare and initialize a long variable like this example:

```
long myLong = 4294967296;
```

When an integer literal has no suffix, its type is the first of these types in which its value can be represented: int, uint, long, ulong. In the preceding example, it is of the type long because it exceeds the range of uint (see Integral Types Table for the storage sizes of integral types). You can also use the suffix L or l with the long type like this:

```
long myLong = 4294967296L;
```

When you use the suffix L or l, the type of the literal integer is determined to be either long or ulong according to its size. In the case it is long because it less than the range of ulong.

A common use of the suffix is with calling overloaded methods. Consider, for example, the following overloaded methods that use long and int parameters:

```
public static void MyMethod(int i) {}

public static void MyMethod(long l) {}
```

Using the suffix L or l guarantees that the correct type is called, for example:

```
MyMethod(5);    // Calling the method with the int parameter

MyMethod(5L);   // Calling the method with the long parameter
```

You can use the long type with other numeric integral types in the same expression, in which case the expression is evaluated as long (or bool in the case of relational or Boolean expressions). For example, the following expression evaluates as long:

```
898L + 88
```

The **ulong** keyword denotes an integral type. It's an unsigned 64-bit integer. The ulong keyword is an alias of **System.UInt64**.

The **short** keyword denotes an integral type. It's a signed 16-bit integer. The short keyword is an alias of **System.Int16**.

The **ushort** keyword denotes an integral type. It's an unsigned 16-bit integer. The ushort keyword is an alias of **System.UInt16**.

2.18 Operators

Operators are special symbols that are commonly used in expressions. C# has 50 built-in operators. There are four basic kinds of operators: arithmetic, bitwise, relational and logical. In addition, many operators can be overloaded by the user, thus changing their meaning when applied to a user-defined type.

Anyone familiar with the operators of C will have no problem using C#'s. C# does add a few operators of its own. In general, operators with the same precedence level will be evaluated from left to right in the given expression. A student has suggested a mnemonic: "Ulcer Addicts Really Like C A lot" where U is Unary, A is Arithmetic, R is Relational, L is logical, C is Conditional, A lot is Assignment. The Associativity for Assignment, ==, != and ?: is from Right to Left, for all others it's Left to Right.

Expressions are constructed from *operands* and *operators*. The operators of an expression indicate which operations to apply to the operands. Examples of operators include +, -, *, /, and new. Examples of operands include literals, fields, local variables, and expressions.

There are three types of operators:

- Unary operators. The unary operators take one operand and use either prefix notation (such as –x) or postfix notation (such as x++).
- Binary operators. The binary operators take two operands and all use infix notation (such as x + y).
- Ternary operator. Only one ternary operator, ?:, exists. The ternary operator takes three operands and uses infix notation (c? x: y).

The order of evaluation of operators in an expression is determined by the *precedence* and *associativity* of the operators.

Certain operators can be *overloaded*. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type.

The following table summarizes all operators in order of precedence from highest to lowest:

Category	Operators
Primary	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
Unary	+ - ! ~ ++x --x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional AND	&&
Conditional OR	||
Conditional	?:
Assignment	= *= /= %= += -= <<= >>= &= ^= |=

When an operand occurs between two operators with the same precedence, the *associativity* of the operators controls the order in which the operations are performed:

- Except for the assignment operators, all binary operators are *left associative*, meaning that operations are performed from left to right. For example, x + y + z is evaluated as (x + y) + z.
- The assignment operators and the conditional operator (?:) are *right associative*, meaning that operations are performed from right to left. For example, x = y = z is evaluated as x = (y = z).

Precedence and associativity can be controlled using parentheses. For example, x + y * z first multiplies y by z and then adds the result to x, but (x + y) * z first adds x and y and then multiplies the result by z.

An example using % operator – P3.cs

```
// P3.cs
using System;
class P3 {
  public static void Main(String[] args) {
    int number = 548;
    int sum = 0;
    while (true) {
      sum = sum + number % 10;
      number = number / 10;
      if (number == 0) break;
    }
    Console.WriteLine("Sum = " + sum);
  }
}
```

2.18.1 checked and unchecked operators

The arithmetic operators (+, -, *, /) can produce results that are outside the range of possible values for the numeric type involved. In general:

Integer arithmetic overflow either throws an OverflowException or discards the most significant bits of the result (see below). Integer division by zero always throws a DivideByZeroException.

Floating-point arithmetic overflow or division by zero never throws an exception, because floating-point types are based on IEEE 754 and so have provisions for representing infinity and NaN (Not a Number).

Decimal arithmetic overflow always throws an OverflowException. Decimal division by zero always throws a DivideByZeroException.

When integer overflow occurs, what happens depends on the execution context, which can be checked or unchecked. The checked operator tells the runtime to generate an OverflowException if an integral expression exceeds the arithmetic limits of that type. The unchecked operator disables arithmetic checking at compile time, the most significant bits of the result are discarded and execution continues. It is seldom useful. Thus, C# gives you the choice of handling or ignoring overflow.

```
// OverFlowTest.cs - Using checked expressions
// The overflow of non-constant expressions is checked at run time
using System;
class OverFlowTest {
   static short x = 32767;   // Max short value
   static short y = 32767;
   // Using a checked expression
   public static int MyMethodCh() {
      int z =  checked((short)(x + y));
      return z;   // Throws the exception OverflowException
   }
   public static void Main() {
      Console.WriteLine("Checked output value is: {0}", MyMethodCh());
   }
}
```

Output:

When you run the program, it throws the exception OverflowException. You can debug the program or abort execution.

```
// UCS.cs - Using the unchecked statement with constant expressions
// Overflow is checked at compile time
using System;
class UCS {
   const int x = 2147483647;   // Max int
   const int y = 2;
   public int MethodUnCh() {
      // Unchecked statement:
      unchecked {
         int z = x * y;
         return z;
      }   // Returns -2
   }
   public static void Main() {
      UCS myObject = new UCS();
      Console.WriteLine("Unchecked output value: {0}",
         myObject.MethodUnCh());
   }
}
```

Output:

Unchecked output value: -2

In addition to the arithmetic operators, integral-type to integral-type casts can cause overflow (for example, casting a long to an int) and are subject to checked or unchecked execution.

2.19 Operator overloading

All unary and binary operators have predefined implementations that are automatically available in any expression. In addition to the predefined implementations, user-defined implementations can be introduced by including operator declarations in classes and structs. User-defined operator implementations always take precedence over predefined operator implementations: Only when no applicable user-defined operator implementations exist will the predefined operator implementations be considered.

The *overloadable unary operators* are:

```
   +   -   !   ~   ++   --   true   false
```

The *overloadable binary operators* are:

```
   +   -   *   /   %   &   |   ^   <<   >>   ==   !=   >   <   >=   <=
```

Only the operators listed above can be overloaded. In particular, it is not possible to overload member access, method invocation, or the =, &&, ||, ?:, new, typeof, sizeof, and is operators.

When a binary operator is overloaded, the corresponding assignment operator is also implicitly overloaded. For example, an overload of operator * is also an overload of operator *=. Note that the assignment operator itself (=) cannot be overloaded. An assignment always performs a simple bit-wise copy of a value into a variable.

Cast operations, such as (T)x, are overloaded by providing user-defined conversions.

Element access, such as a[x], is not considered an overloadable operator. Instead, user-defined indexing is supported through indexers.

User-defined operator declarations always require at least one of the parameters to be of the class or struct type that contains the operator declaration. Thus, it is not possible for a user-defined operator to have the same signature as a predefined operator.

User-defined operator declarations cannot modify the syntax, precedence, or associativity of an operator. For example, the * operator is always a binary operator, always has the predefined precedence level and is always left associative.

While it is possible for a user-defined operator to perform any computation it pleases, implementations that produce results other than those that are intuitively expected are strongly discouraged. For example, an implementation of operator == should compare the two operands for equality and return an appropriate result.

This example shows how you can use operator overloading to create a complex number class Complex that defines complex addition. The program displays the imaginary and the real parts of the addition result.

Example: Complex.cs

```
// Complex.cs
using System;
public class Complex {
  public int real = 0;
  public int imaginary = 0;
  public Complex(int real, int imaginary) {
    this.real = real;
    this.imaginary = imaginary;
  }
  public static Complex operator +(Complex c1, Complex c2) { //Line 10
    return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
  }
  public static void Main() {
    Complex num1 = new Complex(2,3);
    Complex num2 = new Complex(3,4);
    Complex sum = num1 + num2; // Line 16
    Console.WriteLine("Real: {0}", sum.real);           // Line 26
    Console.WriteLine("Imaginary: {0}", sum.imaginary); // Line 27
  }
}
```

Code Discussion:

- Line 010 declares which operator to overload (+), the types that can be added (two Complex objects), and the return type (Complex).
- Line 016 adds two Complex objects (num1 and num2) through the overloaded plus operator declared on line 010.
- Lines 017 and 018 show on execution that the real and imaginary parts of objects num1 and num2 have been added to produce the sum values.

2.20 Program Control

C# borrows most of its statements directly from C and C++, though there are some noteworthy additions and modifications.

2.20.1 The if statement

An if statement selects a statement for execution based on the value of a boolean expression. An if statement may optionally include an else clause that executes if the boolean expression is false.

The example

```
using System;
class Test {
  static void Main(string[] args) {
    if (args.Length == 0)
      Console.WriteLine("No arguments were provided");
         else
                Console.WriteLine("Arguments were provided");
  }
}
```

shows a program that uses an if statement to write out two different messages depending on whether command-line arguments were provided or not.

2.20.2 The switch statement

A switch statement executes the statements that are associated with the value of a given expression, or default of statements if no match exists.

The example

```
using System;
```

```
class Test {
  static void Main(string[] args) {
    switch (args.Length) {
      case 0:
        Console.WriteLine("No arguments were provided");
        break;
      case 1:
        Console.WriteLine("One arguments was provided");
        break;
      default:
        Console.WriteLine("{0} arguments were provided");
        break;
    }
  }
}
```

switches on the number of arguments provided.

The switch expression must be an integer type (including char) or a string. The case labels have to be constants. Unlike Java, you no longer fall through from case to case if you omit the break **and** there's code in the case. You will get a compiler error instead, and have to use a goto to jump to the next (or previous) case. The end of a case statement must explicitly state where to go next. If you have adjacent case labels, (i.e. No code in the case) then you can fall through.

Example: SwitchSelection.cs

```
using System;
class SwitchSelect {
  public static void Main() {
    string myInput;
    int myInt;
    begin:
      Console.Write("Please enter a number between 1 and 3: ");
      myInput = Console.ReadLine();
      myInt = Int32.Parse(myInput);
      // switch with integer type
      switch (myInt) {
        case 1:
          Console.WriteLine("Your number is {0}.", myInt);
          break;
        case 2:
          Console.WriteLine("Your number is {0}.", myInt);
          break;
        case 3:
          Console.WriteLine("Your number is {0}.", myInt);
          break;
        default:
          Console.WriteLine("Your number {0} is not between 1 and 3.", myInt);
      }
    decide:
      Console.Write("Type \"continue\" to go on or \"quit\" to stop: ");
      myInput = Console.ReadLine();
      // switch with string type
      switch (myInput) {
        case "continue":
          goto begin;
        case "quit":
          Console.WriteLine("Bye.");
          break;
        default:
          Console.WriteLine("Your input {0} is incorrect.", myInput);
          goto decide;
      }
  }
}
```

2.20.3 The while statement

A while statement conditionally executes a statement zero or more times – as long as a boolean test is true.

```
using System;
class Test {
        static int Find(int value, int[] arr) {
                int i = 0;
                while (arr[i] != value) {
                        if (++i > arr.Length)
                                throw new ArgumentException();
                }
                return i;
        }
        static void Main() {
                Console.WriteLine(Find(3, new int[] {5, 4, 3, 2, 1}));
        }
}
```

uses a while statement to find the first occurrence of a value in an array.

2.20.4 The do statement

A do statement conditionally executes a statement one or more times.

The example

```
using System;
class Test {
        static void Main() {
                string s;
                do {
                        s = Console.ReadLine();
                }
                while (s != "Exit");
        }
}
```

reads from the console until the user types “Exit” and presses the enter key.

2.20.5 The for statement

A for statement evaluates a sequence of initialization expressions and then, while a condition is true, repeatedly executes a statement and evaluates a sequence of iteration expressions.

The example

```
using System;
class Test {
  static void Main() {
                for (int i = 0; i < 10; i++)
                        Console.WriteLine(i);
        }
}
```

uses a for statement to write out the integer values 1 through 10.

2.20.6 The foreach statement

A foreach statement lets you iterate over the elements in arrays and collections.

The example FE.cs:

```
using System;
class FE {
  static void Main() {
    int[] arr1 = new int[] {1,2,3,4,5};
    foreach (int i in arr1)
      Console.WriteLine("Value is {0}", i);
  }
}
```

uses a foreach statement to iterate over the elements of an array.

2.21 Console I/O

Console I/O is provided by the System.Console class, which gives you access to the standard input (Console.In), standard output (Console.Out) and standard error (Console.Error) streams.

2.21.1 Console Input

Console.In has two methods for obtaining input. Read() returns a single character as an int, or –1 if no more characters are available. ReadLine() returns a string containing the next line of input, or null if no more lines are available.

Note that Console.In.ReadLine() and Console.ReadLine() are equivalent.

2.21.2 Console Output

Console.Out has two methods for writing output. Write() outputs one or more values without a newline character. WriteLine() does the same but appends a newline.

Write() and WriteLine() have numerous overloads, so that you can easily output many different types of data.

If you want to produce formatted output, you use a version of WriteLine() which takes a string containing a format and a variable number of objects. Formats contain both static text, plus markers that show where items from the argument list are to be substituted, and how they are to be formatted. In its simplest form, a marker is a number in curly brackets, the number showing which argument is to be substituted:

```
"The value is {0}"      // use the first argument
"{0} plus {1} = {2}"    // use the first three arguments
```

Note that Console.Out.WriteLine() and Console.WriteLine() are equivalent.

2.22 Array types

Arrays in C# are objects, accessed through a reference variable. Arrays in C# may be single-dimensional or multi-dimensional.

Single-dimensional arrays are the most common type, so this is a good starting point. The example MyArray.cs

```
using System;
```

```
class MyArray {
  static void Main() {
    int[] arr = new int[5]; // size fixed to 5, can’t be changed
    for (int i = 0; i < arr.Length; i++) // index 0 for arrays
      arr[i] = i * i;
    // we can also do something like this
    // int[] arr = new int[] {1,2,3,4,5};
    // or
    // int[] arr = {1,2,3,4,5};
    // any attempt to access an array outside the bounds
    // will result in a runtime error
    for (int i = 0; i < arr.Length; i++)
      Console.WriteLine("arr[{0}] = {1}", i, arr[i]);
  }
}
```

creates a single-dimensional array of int values, initializes the array elements, and then prints each of them out. The program output is:

```
      arr[0]                                  =                                    0
      arr[1]                                  =                                    1
      arr[2]                                  =                                    4
      arr[3]                                  =                                    9
      arr[4] = 16
```

The type int[] used in the previous example is an array type. Array types are written using a non-array-type followed by one or more rank specifiers.

C# supports two types of multidimensional arrays – *rectangular* and *jagged*. In rectangular arrays, every row is the same length. A jagged array is simply an array of one-dimensional arrays, each of which can be of different length if desired.

The example:

```
class Test {
        static void Main() {
                int[] a1;       // single-dimensional array of int
                int[,] a2;      // 2-dimensional array of int
                int[,,] a3;     // 3-dimensional array of int
                int[][] j2;     // "jagged" array: array of (array of int)
                int[][][] j3;   // array of (array of (array of int))
        }
}
```

shows a variety of local variable declarations that use array types with int as the element type.

Arrays are reference types, and so the declaration of an array variable merely sets aside space for the reference to the array. Array instances are actually created via array initializers and array creation expressions. The example

```
class Test {
        static void Main() {
                int[] a1 = new int[] {1, 2, 3};
                int[,] a2 = new int[,] {{1, 2, 3}, {4, 5, 6}};
                int[,,] a3 = new int[10, 20, 30];
                int[][] j2 = new int[3][];
                j2[0] = new int[] {1, 2, 3};
                j2[1] = new int[] {1, 2, 3, 4, 5, 6};
                j2[2] = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9};
        }
}
```

shows a variety of array creation expressions. The variables a1, a2 and a3 denote *rectangular arrays*, and the variable j2 denotes a *jagged array*. It should be no surprise that these terms are based on the shapes of the arrays. Rectangular arrays always have a rectangular shape. Given the length of each dimension of the array, its rectangular shape is clear. For example, the length of a3's three dimensions are 10, 20, and 30 respectively, and it is easy to see that this array contains 10*20*30 elements.

In contrast, the variable j2 denotes a "jagged" array, or an "array of arrays". Specifically, j2 denotes an array of an array of int, or a single-dimensional array of type int[]. Each of these int[] variables can be initialized individually, and this allows the array to take on a jagged shape. The example gives each of the int[] arrays a different length. Specifically, the length of j2[0] is 3, the length of j2[1] is 6, and the length of j2[2] is 9.

It is important to note that the element type and number of dimensions are part of an array's type, but that the length of each dimension is not part of the array's type. This split is made clear in the language syntax, as the length of each dimension is specified in the array creation expression rather than in the array type. For instance the declaration

```
      int[,,] a3 = new int[10, 20, 30];
```

has an array type of int[,,] and an array creation expression of new int[10, 20, 30].

For local variable and field declarations, a shorthand form is permitted so that it is not necessary to re-state the array type. For instance, the example

```
      int[] a1 = new int[] {1, 2, 3};
```

can be shortened to

```
      int[] a1 = {1, 2, 3};
```

without any change in program semantics.

It is important to note that the context in which an array initializer such as {1, 2, 3} is used determines the type of the array being initialized. The example

```
class Test {
        static void Main() {
                short[] a = {1, 2, 3};
                int[]   b = {1, 2, 3};
                long[] c = {1, 2, 3};
        }
}
```

shows that the same array initializer can be used for several different array types. Because context is required to determine the type of an array initializer, it is not possible to use an array initializer in an expression context. The example

```
class Test {
        static void F(int[] arr) {}
        static void Main() {
                F({1, 2, 3});
        }
}
```

is not valid because the array initializer {1, 2, 3} is not a valid expression. The example can be rewritten to explicitly specify the type of array being created, as in

```
class Test {
        static void F(int[] arr) {}
        static void Main() {
                F(new int[] {1, 2, 3});
        }
}
```

Example RA.cs

```
// RA.cs
using System;
class RA {
  public static int Main() {
    int[,] arr2;          // 2D rectangular array of ints
    arr2 = new int[5,5];  // create 5 by 5 array
    for (int i=0; i<5; i++)
      for (int j=0; j<5; j++)
        arr2[i,j] = i*j;
      // create a 2D array with implied size of 3 by 2
      int[,] arr3 = new int[,] {
        {1,2},   // first row
        {4,5},   // second row
        {7,8}    // third row
      };
      // write out some elements
      Console.WriteLine("{0}", arr2[2,2]);
      Console.WriteLine("{0}", arr3[1,1]);
      return 0;
  }
}
```

Example JA.cs

```
//JA.cs
using System;
class JA {
  public static void Main() {
    int[][] arr4;           // 2D jagged array of ints
    arr4 = new int[4][];    // four rows in this array
    arr4[0] = new int[5];   // first row has 5 elements
    arr4[1] = new int[3];   // second row has 3 elements
    arr4[2] = new int[4];   // third row has 4 elements
    arr4[3] = new int[10];  // last row has 10 elements
    arr4[1][1] = 3;         // assign an element
  }
}
```

In C#, arrays are actually objects. **System.Array** is the base type of all array types. You can use the properties, and other class members that **System.Array** has. An example of this would be using the **Length** property to get the length of an array. The following code assigns the length of the numbers array, which is 5, to a variable called LengthOfNumbers:

```
int[] numbers = {1, 2, 3, 4, 5};
int LengthOfNumbers = numbers.Length;
```

The **System.Array** class provides many other useful methods/properties, such as methods for sorting, searching, and copying arrays.

Getting Command-Line Input:

Example: NamedWelcome.cs

```
// Namespace Declaration
using System;
// Program start class
class NamedWelcome {
  // Main begins program execution.
  public static void Main(string[] args) {
    // Write to console
    Console.WriteLine("Hello, {0}!", args[0]);
    Console.WriteLine("Welcome to the CSharp Seminar!");
  }
}
```

Remember to add your name to the command-line, i.e. "NamedWelcome Satish". If you don't, your program will crash. You'll notice an entry in the "Main" method's parameter list. The parameter name is "args". It's what you use to refer to the parameter later in your program. The "string[]" expression defines the Type of parameter that "args" is. The "string" Type holds characters. These characters could form a single word, or multiple words. The "[]", square brackets denote an Array, which is like a list. Therefore, the Type of the "args" parameter, is a list of words from the command-line.

You'll also notice an additional "Console.WriteLine(...)" statement within the "Main" method. The argument list within this statement is different from before. It has a formatted string with a "{0}" parameter embedded in it. The first parameter in a formatted string begins at number 0, the second is 1, and so on. The "{0}" parameter means that the next argument following the end quote will determine what goes in that position.

The next argument following the end quote is the "args[0]" argument, which refers to the first string in the "args" array. The first element of an Array is number 0, the second is number 1, and so on. For example, if we write "NamedWelcome Satish" on the command-line, the value of "args[0]" would be "Satish".

Now we'll get back to the embedded "{0}" parameter in the formatted string. Since "args[0]" is the first argument, after the formatted string, of the "Console.WriteLine()" statement, it's value will be placed into the first embedded parameter of the formatted string. When this command is executed, the value of "args[0]", which is "Satish" will replace "{0}" in the formatted string. Upon execution of the command-line with "NamedWelcome Satish", the output will be as follows:

```
Hello, Satish!

Welcome to the CSharp Seminar!
```

Another way to provide input to a program (InteractiveWelcome.cs) is via the console. The next program shows how to obtain interactive input from the user.

```
//InteractiveWelcome.cs
// Namespace Declaration
using System;
// Program start class
class InteractiveWelcome {
  // Main begins program execution.
  public static void Main() {
    // Write to console/get input
    Console.Write("What is your name?: ");
    Console.Write("Hello, {0}! ", Console.ReadLine());
    Console.WriteLine("Welcome to the CSharp Seminar!");
  }
}
```

This time, the "Main" method doesn't have any parameters. However, there are now three statements and the first two are different from the third. They are "Console.Write(...)" instead of "Console.WriteLine(...)". The difference is that the "Console.Write(...)" statement writes to the console and stops on the same line, but the "Console.WriteLine(...)" goes to the next line after writing to the console.

The first statement simply writes "What is your name?: " to the console.

The second statement doesn't write anything until its arguments are properly evaluated. The first argument after the formatted string is "Console.ReadLine()". This causes the program to wait for user input at the console, followed by a Return or Enter. The return value from this method replaces the "{0}" parameter of the formatted string and is written to the console.

The last statement writes to the console as described earlier. Upon execution of the command-line with "InteractiveWelcome", the output will be as follows:

```
What is your Name?  <type your name here>

Hello, <your name here>!  Welcome to the CSharp Seminar!
```

2.23 Calling methods – ref / out

Methods are extremely useful because they allow you to separate your logic into different units. Defining and calling methods in C# closely follows the Java model.

The structure of a method is as follows:

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

We defer discussion of attributes and modifiers. The return-type can be any C# type, if the method does not return a value, its return type is void. It can be assigned to a variable for use later in the program or the return value can be ignored. 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. If there are no parameters, the method name is followed by empty parentheses. They are surrounded by parenthesis. Statements within the curly braces carry out the functionality of the method. As in Java, you are free to ignore the return value of a method call.

What if you want to pass a value type into a method and modify it? C# provides two ways of doing this using the **ref** and **out** keywords.

The **ref** keyword tells the C# compiler that the arguments being passed point to the same memory as the variables in the calling code. That way, if the called method modifies these values and then returns, the calling code's variables will have been modified. When you use the *ref* keyword, you must initialize the passed arguments before calling the method. See the example RefTest.cs below.

```
using System;
public class RefTest {
  int p = 3;
  public RefTest() {
    refMethod(ref p);
  }
  public static void Main(string[] args) {
    RefTest t = new RefTest();
    Console.WriteLine(t.p);
  }
  void refMethod(ref int n) {
    n += 3;
  }
}
```

The only difference between the **ref** keyword and the **out** keyword is that the *out* keyword doesn't require the calling code to initialize the passed arguments first. See the example OutTest.cs below.

```
using System;
public class OutTest {
  int p;
  public OutTest() {
    outMethod(out p);
  }
  public static void Main(string[] args) {
    OutTest t = new OutTest();
    Console.WriteLine(t.p);
  }
  void outMethod(out int n) {
    n = 3;
  }
}
```

We will now use the ref keyword on a string variable and modify the previous RefTest.cs program and call it RefTest2.cs

```
using System;
public class RefTest2 {
  string s = "Satish";
  public RefTest2() {
    refMethod(ref s);
  }
  public static void Main(String[] args) {
    RefTest2 t = new RefTest2();
    Console.WriteLine(t.s);
  }
  void refMethod(ref string s) {
    s = "Talim";
  }
}
```

The output is Talim, a expected. If we now remove the ref keyword in the above program and call the new program RefTest3.cs

```
using System;
public class RefTest3 {
  string s = "Satish";
  public RefTest3() {
    refMethod(s);
  }
  public static void Main(String[] args) {
    RefTest3 t = new RefTest3();
    Console.WriteLine(t.s);
  }
  void refMethod(string s) {
    s = "Talim";
  }
}
```

The output is Satish. Therefore, if a parameter is declared for a method without **ref** or **out**, the parameter can have a value associated with it. That value can be changed in the method, but the changed value will not be retained when control passes back to the calling procedure. By using a method parameter keyword (ref, out), you can change this behavior.

2.23.1 Method Overloading

C# allows you to declare more than one method with the same name. This is called overloading. Overloaded methods must differ in the number and/or type of arguments they take. The return type does not play any part in the overload resolution, since it's always possible to call a method without using the return value.

```
using System;
class Log {
  public Log(string fileName) {
    // Open fileName and seek to end.
  }
  public void WriteEntry(string entry) {
    Console.WriteLine(entry);
  }
  public void WriteEntry(int resourceId) {
    Console.WriteLine
      ("Retrieve string using resource id and write to log");
  }
}
```

2.23.2 Variable Method Parameters - params

You can specify a variable number of method parameters by using the **params** keyword and by specifying an array in the method's argument list.

```
// VarArgsApp.cs
using System;
class MyPoint {
  public int x;
  public int y;
  public MyPoint(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
class Chart {
  public void DrawLine(params MyPoint[] p) {
    Console.WriteLine("\nThis method would print a line " +
                      "along the following points:");
    for (int i = 0; i < p.GetLength(0); i++) {
      Console.WriteLine("{0}, {1}", p[i].x, p[i].y);
    }
  }
}
class VarArgsApp {
  public static void Main() {
    MyPoint p1 = new MyPoint(5,10);
    MyPoint p2 = new MyPoint(5, 15);
    MyPoint p3 = new MyPoint(5, 20);
    Chart chart = new Chart();
    chart.DrawLine(p1, p2, p3);
  }
}
```

GetLength(0) returns the number of elements in the first dimension of the **Array**. The *DrawLine* method tells the C# compiler that it can take a variable number of *Point* objects. At run time, the method then uses a simple *for* loop to iterate through the *Point* objects that are passed, printing each one.

2.24 Handling Exceptions

C# supports exceptions in very much the same way as Java. In C#, what you throw has to be a System.Exception object, or something derived from System.Exception. Code that may give rise to exceptions is enclosed in a try block, which is followed by one or more catch blocks.

Beta 1 of the .NET framework includes built-in exception handlers for common exception types such as divide-by-zero exception. Try this code, in the source file DBZ.cs

```
using System;
public class DBZ {
  public static void Main(string[] args) {
    double a = 15.0;
    double b = 0.0;
    double result = a/b;
    Console.WriteLine("Result is {0}", result);
  }
}
```

You will get the output:

```
Result is Infinity
```

Modifying the program to DBZ2.cs as shown below, makes no difference to the output:

```
using System;
public class DBZ2 {
  public static void Main(string[] args) {
    double a = 15.0;
    double b = 0.0;
    double result = a/b;
    try {
      Console.WriteLine("Result is {0}", result);
    } catch (Exception e) {
        Console.WriteLine("Exception occured: " + e);
    }
  }
}
```

A catch block which will catch any exception is called a general catch clause. These blocks don't specify an exception variable, and can be written like this:

```
try {
  // code which may fail
} catch {
    // handle error
}
```

A try block can also have a finally block associated with it. If a finally block exists, it will be executed before the try block is completed, and after any possible exceptions are caught in the catch clause. This will happen no matter how control leaves the try, whether it is due to normal termination, to an exception occurring, a break or continue (or goto) statement, or a return. A finally block can occur with or without catch blocks. It is an error to transfer control out of a finally block using break, continue, return or goto.

The point to note for Java programmers is: Not only is the argument to catch optional, but the entire catch clause itself is optional. Also, C# has no **throws** keyword. By not requiring explicit exception declarations in method signatures, C# values short-term programmer convenience over program safety and correctness.

```
public void ReadFile() {
  try {
     // code which may fail
  } finally {
       // close the file
  }
}
```

CoreException is C#'s equivalent of Error in Java and you are not expected to handle it. You can make up your own exception classes by deriving from one of the various exception classes existing. See MyException.cs below:

```
using System;
class VowelException : Exception {}
class BlankException : Exception {}
class ExitException  : Exception {}

class MyException {
  public static void Main(String[] args) {
    bool finished = false;

    do {
      try {
        processUserInput();
      } catch (VowelException x) {
          Console.WriteLine("VowelException occured: " + x);
      } catch (BlankException y) {
          Console.WriteLine("BlankException occured: " + y);
      } catch (ExitException z) {
          Console.WriteLine("ExitException occured: " + z);
          // Using StackTrace property of Exception class
          Console.WriteLine("Trace: " + z.StackTrace);
          finished = true;
      } finally {
          Console.WriteLine("This is the finally clause.");
      }
    } while(!finished);
  }

  // Note unlike Java, we do not declare that the method
  // processUserInput() throws … exceptions
  static void processUserInput()  {
    Console.Write("Enter a character: ");
    String s;
    s = Console.ReadLine();
    char c = s[0];
    switch (c) {
      case 'A':
      case 'a':
      case 'E':
      case 'e':
      case 'I':
      case 'i':
      case 'O':
      case 'o':
      case 'U':
      case 'u':
        throw (new VowelException());
      case ' ':
        throw (new BlankException());
      case 'X':
        throw (new ExitException());
    }
  }
}
```

2.25 Namespaces

Namespaces are a great way of categorizing your types and classes to avoid name collisions.

When a language develops, many third party components are available. All these parties try to give meaningful names to their classes like the Math class, or InterestCalculator etc. We as developers end up in misery due to this. Just consider an example, I am developer who is writing up a shopping cart class. I am using third party components. I purchase two such components, one calculates discount rates for retail customers and other calculates discount rates for wholesale customers. The problem arises when both these components have a class called ***Discount***. Now in my shopping cart class how do I use both these classes unambiguously? eg.

```
Discount d1 = new Discount();
```

//Which component does the compiler here refer to? Reseller discount or Wholesaler Discount?

int discount = d1.Cal(45.78) ;

Some compilers will complain others might compile and use any of the components as they wish!

This problem has been identified and solved by Microsoft on the .NET Platform with the use of Namespaces. The relationship between Namespaces and classes can directly be compared to the relationship between Files and Folders. Files actually contain data, while Folders are used to just manage and logically arrange Files. Folders can also contain many files and sub folders within them.
In the same way, Classes actually contain data and Namespaces are used to logically arrange classes. Namespaces can also contain many other Namespaces and classes. This concept may seem similar to the Package - Class relationship used in Java. However, one point of caution, you DO NOT need to create folders to store classes into a Namespace. Just defining the Namespace keyword above the class definition is enough.

Again you might say that there are two companies whose abbreviated names might be XYZ and if both companies develop the Discount class then, even if they used namespaces their full names would be XYZ.Discount class? Which will lead to the problem that we faced earlier! This is very true since Namespaces only help to extend the name of a class to make it unique. Hence, Microsoft is encouraging companies to use their full names to develop components and not abbreviations, also if possible, the department names as well as the team names should be used . Therefore, a company like American Business Company should use its full name. AmericanBusinessCompany.It.DotNet.Discount instead of using ABC.Discount. This will help to solve the naming conflict.

C# programs are organized using namespaces. Namespaces provide a way to group classes, by providing an extra level of naming beyond the class name. To use namespaces all you have to do is place the namespace definition above your class definition. The Dot '.' is used to denote and access classes within the namespace.

Earlier, we presented a “Hello, world” program. We’ll now rewrite this program (as HelloMsg.cs) in two pieces: a HelloMessage component that provides messages and a console application that displays messages.

First, we’ll provide a HelloMessage class in a namespace. What should we call this namespace? By convention, developers put all of their classes in a namespace that represents their company or organization. We’ll put our class in a namespace named JavaTech.CSharp.Introduction.

```
namespace JavaTech.CSharp.Introduction {
  public class HelloMessage {
    public string GetMessage() {
      return "Hello, world";
    }
  }
}
```

Namespaces are hierarchical, and the name JavaTech.CSharp.Introduction is actually shorthand for defining a namespace named JavaTech that contains a namespace named CSharp that itself contains a namespace named Introduction, as in:

```
namespace JavaTech {
        namespace Csharp {
                namespace Introduction
                {....}
        }
}
```

Next, we'll write a console application that uses the HelloMessage class. We could just use the fully qualified name for the class – JavaTech.CSharp.Introduction.HelloMessage – but this name is quite long and unwieldy. An easier way is to use a "using" directive, which makes it possible to use all of the types in a namespace without qualification.

```
using JavaTech.CSharp.Introduction;
class Hello {
  static void Main() {
    HelloMessage m = new HelloMessage();
    System.Console.WriteLine(m.GetMessage());
  }
}
```

Note that the two occurrences of HelloMessage are shorthand for JavaTech.CSharp.Introduction.HelloMessage.

C# also enables the definition and use of aliases. Such aliases can be useful in situation in which name collisions occur between two libraries, or when a small number of types from a much larger namespace are being used. Our example can be rewritten using aliases as:

```
using MessageSource = JavaTech.CSharp.Introduction.HelloMessage;
class Hello {
        static void Main() {
                MessageSource m = new MessageSource();
                System.Console.WriteLine(m.GetMessage());
        }
}
```

Namespaces are often related to assemblies (which we will cover later).

Namespace elements cannot be explicitly declared as private or protected. Only public members are allowed in a namespace. Internal is the default. The public keyword must be explicitly specified.

The following sample generates error CS1527:

```
namespace bad {
  private class foo1 {}   // CS1527
  protected class foo2 {} // CS1527
  class foo3 {            // allowed. This is internal
    static void Main() {}
  }
}
```

Inside a namespace, the compiler only accepts classes, structs, unions, enums, interfaces, and delegates.

The following sample generates error CS0116:

```
namespace x {
   int xx;   // CS0116
}
```

Even if you do not explicitly declare a namespace, a default namespace is created. This unnamed namespace, sometimes called the global namespace, is present in every file. Any identifier in the global namespace is available for use in a named namespace. Namespaces implicitly have public access and this is not modifiable. An example of this will make it clear. Create a program, NSP.cs, compile and run:

```
// NSP.cs
using System;
namespace Test {
  public class Tst {
    public static void Main(string[] args) {
      GTst t = new GTst();
    }
  }
}
class GTst { // this is in the global namespace
  public GTst() {
    Console.WriteLine("GTst called...");
  }
}
```

The **using** keyword has two uses:

- Create an alias for a namespace (a **using** alias).
- Permit the use of types in a namespace, such that, you do not have to qualify the use of a type in that namespace (a **using** directive).

Java programmers should note that, we could use namespace first, followed by using or vice-versa.

The only purpose of the using command in this context is to save you typing and make your code simpler. It does not, for example, cause any other code or libraries to be added to your project. If your code uses base classes, you need to ensure separately that the compiler knows which assemblies to look in for the classes (/r switch in the compiler).

Although you can't specify a class in a *using* directive, the following variant of the *using* directive does enable you to create aliases for classes:

using *alias* = *class*

Using this form of the *using* directive, you can write code like the following:

```
using output = System.Console;
class HelloWorld {
  public static void Main() {
    output.WriteLine("Hello, World");
  }
}
```

This gives you the flexibility to apply meaningful aliases to classes that are nested several layers deep in the .NET hierarchy, thus making your code a little easier to both write and maintain.

The using keyword cannot be declared inside a class.

2.26 Namespaces and Assemblies

The basic element of packaging in .NET is the *assembly*. An assembly consists of Intermediate Language code, metadata that describes what is in the assembly, and any other files that the application needs to run, such as graphics and sound files (assemblies are similar to jar files in Java). On Windows systems, assemblies can be an executable (.exe) file, a DLL, or some combination of these. It may also contain resource files. One assembly can be split across more than one physical file.

There is often a correspondence between namespaces and assemblies, so that the classes in the say Finance.Bank namespace would be built into an assembly called Finance.Banl.dll. This is not mandatory.

In order to create an assembly with a particular name, compile as follows:

```
csc Bankstuff.cs /t:library /out:Finance.Bank.dll
```

During compilation how can the compiler come to know which assemblies you have used and how can the compiler map the functions and class names?

To enable the compiler to know where it can find the assembly which contains all the Namespaces and classes you have used, you have to reference those assemblies during compilation with the /Reference or /r compiler switch. If you fail to reference all the assemblies you have used, the compiler cannot map the classes and Namespaces and a CS0234 Error "The type or Namespace "XXX" does not exist in the class or Namespace "XXX" is raised.

So how do you know which assemblies should you reference? A simple way would be to look at the *using* keyword, check all the Namespaces you have aliased and in which assembly (library dll) does it exist. To find this out start the Reference documentation from "Start -> Program Files -> Microsoft.NET -> Reference Documentation". Go to the ".NET Framework Reference" section, then check out which Namespaces you have used. The reference documentation contains information about the assemblies/ libraries that contain the Namespace. Once you have figured out the list of assemblies that you have to reference use the CSC C# compiler and while compiling add the reference to all the assemblies.

Example: For a simple Win Form application the compilation string is :

```
csc
/r:System.dll;System.WinForms.dll;Microsoft.Win32.Interop.dll;System.Drawing.dll
yourcodefile.cs
```

The library file mscorlib.dll is automatically referenced by the C# compiler.

If you're packaging several classes in a given namespace, you can define each of these classes in its own source code file. A programmer employing your classes can get access to all the classes within a namespace through the using keyword. Let’s see this in more detail:

```
// NA.cs
using System;
namespace A {
  public class NA {
    static void Main(String[] args) {
    }
    public NA() {
      Console.WriteLine("Using A.NA");
    }
  }
}


// NB.cs
```

```
using System;
namespace A {
  public class NB {
    public NB() {
      Console.WriteLine("Using A.NB class");
    }
    static void Main() {
    }
  }
}
```

```
// Test.cs
using System;
using A;
public class Test {
  public static void Main(string[] args) {
    NA o1 = new NA();
    NB o2 = new NB();
  }
}
```

Both NA.cs and NB.cs compile. However, when I compile Test.cs, I get an error:

Test.cs(3,7): error CS0234: The type or namespace name 'A' does not exist in the class or namespace ''

To overcome this error use the .NET BETA 1 instructions for DLLs -

```
csc                    /t:library                    na.cs
csc                    /t:library                    nb.cs
csc        test.cs            /r:na.dll            /r:nb.dll
```

The program Test.exe will now run.

2.27 Summary of Key Concepts

- To build and run your C# programs, you will need Windows 2000, IE 5.5, Microsoft .NET SDK (Beta 1), an optional Visual Studio .NET and an optional free C# IDE – Sharp Develop.
- The .NET Framework consists of the:
  1. Common Language Runtime (CLR): The code that runs within the CLR runs in an encapsulated and managed environment, separate from other processes on the machine.
  2. Base Class Library (BCL): C# itself does not include a class library. The BCL is shared between all programming languages on the .NET.
  3. Common Type System (CTS): In the world of .NET and C# everything in the CTS is an object.
  4. All .NET programs are compiled to an intermediate language (IL), rather than to native code which can be understood by the computer's processor. This IL code is then compiled to native code either when the application is installed, or when the application is run.
- The name of the Csharp program can be anything you want.
- C# compiles straight from source code to executable (.exe), with no object files.
- C# program code is case sensitive and in free format.

- A **namespace** in C# is somewhat similar to a package statement in Java. However, the namespace name does not have to map onto phyical directories, as is the case with the package statement in Java.
- **using** is very similar in concept to Java's import keyword. Unlike Java, in C# you cannot import a single class, but the whole package.
- In Main() the first character is a capital M.
- Main() is declared inside a class or struct.
- Every C# application must have a method named *Main* defined in one of its classes. It doesn't matter which class contains the method—you can have as many classes as you want in a given application—as long as one class has a method named *Main*. If you have zero or more than one Main() in a program, you can expect compiler errors.
- You can compile a program containing multiple Main() by using the */main:*< className > switch with the C# compiler to specify which class's *Main* method is to be used. The **/main** compiler option specifies a class in which to look for a Main method. However, the **Main** method has to be defined as follows: public static void Main(){ }
- The **Main**() method can be written as follows:
  1. static void Main()
  2. public static void Main()
  3. static int Main() – it can return an int only.
  4. static void Main(string[] args) – can take command line arguments.
- For **Compiler Errors** - You can search for the error code in the .NET Framework SDK Documentation (which is installed with the .NET Framework SDK) for a more detailed description.
- **Naming Guidelines**: First alphabet of each word (eg. MyCounter) for namespaces, classes, methods, method arguments and interfaces (interfaces should start with I). Class members (variables), the first alphabet is lower-case and the start of each word is upper-case.
- C# supports three types of **comments**: //, /* and ///. The later will be treated as normal comments by the compiler, unless you use the /doc:filename compiler option to tell it to generate the XML. Observe the generated .xml file - All the members of the assembly are denoted by <member> tags, the compiler adds the full name of the member as a name attribute. The T, F and M prefixes denote types, fields and members respectively.
- **const** members are calculated at compile-time. By default, *const* members are static.
- A **readonly** modifier, specifies that the member can have its value set once only, and afterwards is read-only. The readonly fields that we define are instance fields, meaning that the user would have to instantiate the class to use the fields.
- In C#, static members must be accessed through a class name. You can't access them via an object, as you can in Java.
- C# supports two kinds of types: *value types* and *reference types*. Value types are allocated on the stack and Reference types are allocated on the heap.
- The process of turning a value type into a reference type is known as ***boxing****.* The process of turning a reference type back into a value type is called ***unboxing****.* In the case of unboxing, an object could be cast to any type. Therefore, the cast is necessary so that the compiler can verify that the cast is valid per the specified variable type.

- All types – including value types – can be treated like objects and derive from Object. It's possible to call Object methods on any value, even values of "primitive" types such as `int`.
- All the types have fixed sizes, and will be the same size on any system.
- In C#, each of the predefined types is a shorthand (alias) for a system-provided type.
- The **`decimal`** type is appropriate for calculations in which rounding errors are unacceptable. Common examples include financial calculations such as tax computations and currency conversions. The `decimal` type provides 28 significant digits.
- `Equals()` is a very important method in `Object` class. The default implementation of **Equals** supports reference equality only, but subclasses can override this method to support value equality instead. In the case of value types, this method returns true if the two types are identical and have the same value.
- **string** is a reference type; the equality operators (== and !=) are overloaded to compare the values of string objects, not references.
- The + operator concatenates strings. The [] operator accesses individual characters of a string
- String literals are of type string and can be written in two forms, quoted and @-quoted.
- If you declare a string as `string str = new string("Hello");` you will get a compiler warning.
- In C#, many operators can be overloaded by the user.
- Integer arithmetic overflow either throws an OverflowException or discards the most significant bits of the result. Integer division by zero always throws a DivideByZeroException.
- Floating-point arithmetic overflow or division by zero never throws an exception, because floating-point types are based on IEEE 754 and so have provisions for representing infinity and NaN (Not a Number).
- Decimal arithmetic overflow always throws an OverflowException. Decimal division by zero always throws a DivideByZeroException.
- When integer overflow occurs, what happens depends on the execution context, which can be checked or unchecked. In a **checked** context, an OverflowException is thrown. In an **unchecked** context, the most significant bits of the result are discarded and execution continues.
- The switch expression must be an integer type (including char) or a `string`. The case labels have to be constants. Unlike Java, you no longer fall through from case to case if you omit the break **and** there's code in the case. You will get a compiler error instead, and have to use a goto to jump to the next (or previous) case. If you have adjacent case labels, (i.e. No code in the case) then you can fall through.
- A **`foreach`** statement lets you iterate over the elements in arrays and collections.
- C# supports two types of multidimensional arrays – ***rectangular* and *jagged***. In rectangular arrays, every row is the same length. A jagged array is simply an array of one-dimensional arrays, each of which can be of different length if desired.
- C# provides two ways of passing a value type into a method and modify it using the **ref** and **out** keywords. The **ref** keyword tells the C# compiler that the arguments being passed point to the same memory as the variables in the calling code. That way, if the called method modifies these values and then returns, the calling code's variables will have been modified. When you use the *ref* keyword, you must initialize the passed arguments before calling the method.

- The only difference between the **ref** keyword and the **out** keyword is that the *out* keyword doesn't require the calling code to initialize the passed arguments first.
- You can specify a variable number of method parameters by using the **params** keyword and by specifying an array in the method's argument list.
- In C#, what you throw has to be a System.Exception object, or something derived from System.Exception. Beta 1 of the .NET framework includes built-in exception handlers for common exception types such as divide-by-zero exception.
- A catch block which will catch any exception is called a general catch clause. These blocks don’t specify an exception variable.
- If a finally block exists, it will be executed before the try block is completed, and after any possible exceptions are caught in the catch clause. A finally block can occur with or without catch blocks.
- The argument to catch is optional, but the entire catch clause itself is optional. Also, C# has no **throws** keyword.
- Namespace elements cannot be explicitly declared as private or protected. Only public members are allowed in a namespace (even internal and default are allowed).
- Even if you do not explicitly declare a namespace, a default namespace is created. This unnamed namespace, sometimes called the global namespace, is present in every file. Any identifier in the global namespace is available for use in a named namespace. Namespaces implicitly have public access and this is not modifiable.
- Java programmers should note that, we could use namespace first, followed by using or vice-versa.
- The **using** keyword has two uses - Create an alias for a namespace or permit the use of types in a namespace, such that, you do not have to qualify the use of a type in that namespace.
- We are not allowed to use the same **using** directive more than once in our program ie. we cannot say:

  ```
  using System;
  using System;
  ```
- You can think of assemblies are something similar to jar files in Java.

3. Object Oriented Concepts

Object-oriented technologies provide many benefits to software developers and their products. Object-oriented concepts can be difficult to grasp and one's understanding of the concepts often slowly evolve over time.

If there's a downside, it is the expense of the learning curve. Thinking in objects is a dramatic departure from thinking procedurally, and the process of designing objects is much more challenging than procedural design, especially if you're trying to create reusable objects.

The significance of object-oriented technology is that it enables programmers to design software in much the same way that they perceive the real world.

Here's an example. You can walk into a computer store, and with a little background and often some help, assemble an entire PC computer system from various components: a motherboard, a CPU chip, a video card, a hard-disk, a keyboard, and so on. Ideally, when you finish assembling all the various self-contained units, you have a system in which all the units work together to create a larger system with which you can solve the problems you bought the computer in the first place.

Internally, each of those components may be vastly complicated and engineered by different companies with different methods of design. However, you don't need to know how the component works, what every chip on the board does, or how, when you press the A key, an "A" is sent to your computer. As the assembler of the overall system, each component you use is a self-contained unit, and all you are interested in is how the unit interacts with each other. Will this video card fit into the slots on the motherboard and will this monitor work with this video card? Will each particular component speak the right commands to the other components it interacts with so that each part of the computer is understood by every other part? Once you know what the interactions are between the components and can match the interactions, putting together the overall system is easy.

Object-Oriented Programming works in exactly the same way. Using object-oriented programming, your overall program is made up of lots of different self-contained components (objects), each of which has a specific role in the program and all of which can talk to each other in predefined ways.

3.1 What is an Object?

Objects are software bundles of data and related procedures that act on that data. The procedures are also known as methods (function in C). Software objects are often used to model real-world objects you find in everyday life.

As the name implies, objects are the key concept to understanding object-oriented technology. You can look around you now and see many examples of real-world objects: your dog, your desk, your television set, and your bicycle.

These real-world objects share two characteristics: they all have attributes and they all have behaviour. For example, Bicycles have attributes (current gear, current pedal cadence, two wheels, number of gears) and behaviour (braking, accelerating, slowing down, changing gears).

Attributes are the individual things that differentiate one object from another and determine the appearance, state or other qualities of that object. Attributes of an object can also include information about its state; for example, you could have features for engine condition (off or on). Attributes are defined by variables.

Software objects are modeled after real world objects in that, they too, have attributes and behaviour. A software object maintains its attributes in variables and implements its behaviour with methods.

Definition: An object is a software bundle of variables and related methods.

You can represent real-world objects in programs using software objects. You might want to represent a real-world bicycle as a software object within an electronic exercise bike. However, you can also use software objects to "objectify" abstract concepts. For example, "event" is a common object used in GUI window systems to represent the event when a user presses a mouse button or types a key on the keyboard.

Everything that the software object knows (attribute) and can do (behaviour) is expressed by the variables and methods within that object. A software object that modeled your real-world bicycle would have variables that indicated the bicycle's current state: its speed is 10 mph, its pedal cadence is 90 rpm, and its current gear is the 5th gear. These variables and methods are formally known as instance variables and instance methods to distinguish them from class variables and class methods. The software bicycle would also have methods to brake, change the pedal cadence and change gears. (The bike would not have a method for changing the speed of the bicycle as the bike's speed is really just a side-effect of what gear it's in, how fast the rider is pedaling and how steep the hill is).

Anything that an object does not know or cannot do is excluded from the object. For example, your bicycle (probably) doesn't have a name, and it can't run, bark or fetch. Thus, there are no variables or methods for those states and behaviours.

As you can visualise, the object's variables make up the center or nucleus of the object and the methods surround and hide the object's nucleus from other objects in the program. Packaging an object's variables within the protective custody of its methods is called encapsulation. Typically, encapsulation is used to hide unimportant implementation details from other objects. When you want to change gears on your bicycle, you don't need to know how the gear mechanism works; you just need to know which lever to move. Thus, the implementation can change at any time without changing other parts of the program.

3.2 The Benefit of Encapsulation

Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two primary benefits to software developers:

- Modularity - the source code for an object can be written and maintained independently of the source code for other objects. In addition, an object can be easily passed around in the system. You can give your bicycle to someone else and it will still work.
- Information hiding - an object has a public interface which other objects can use to communicate with it. However, the object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it. You don't need to understand the gear mechanism on your bike in order to use it.

3.3 What are Messages?

Software objects interact and communicate with each other via messages.

A single object alone is generally not very useful and usually appears as a single component of a larger program or application that contains many other objects. It is through the interaction of these objects that programmers achieve higher order functionality and more complex behaviour. Your bicycle hanging from a hook in the garage is just a bunch of titanium alloy and rubber; by itself, the bicycle is incapable of any activity. The bicycle is useful only when another object (you) interact with it (start pedaling).

Software objects interact and communicate with each other by sending messages to each other. When object A wants object B to perform one of its methods, object A sends a message to object B.

Sometimes the receiving object needs more information so that it knows exactly what to do - for example, when you want to change gears on your bicycle, you have to indicate which gear you want. This information is passed along with the message as parameters. The message parameters are actually the parameters to a method. The method's return type is the object's response to the message.

Three components comprise a message:

- the object to whom the message is addressed (bicycle)
- the name of the method to perform (change gears)
- any parameters needed by the method (to a higher gear)

These three components are enough information for the receiving object to perform the desired method. No other information or context is required.

The Benefit of Messages:

Everything an object can do is expressed through its methods, so message passing supports all possible interactions between objects.

Objects don't need to be in the same process or even on the same machine to send and receive messages back and forth to each other.

3.4 What are Classes?

A class is a blueprint, prototype or description that defines the variables and the methods common to all objects of a certain kind.

In the real world, you often have many objects of the same kind. For example, your bicycle is just one of many bicycles in the world. Using object-oriented terminology, we say that your bicycle object is an instance of the class of objects known as bicycles. All bicycles have some attributes (current gear, current cadence, two wheels) and behaviour (change gears, brake) in common. However, each bicycle's attribute is independent of and can be different from other bicycles.

When building bicycles, manufacturers take advantage of the fact that bicycles share characteristics and they build many bicycles from the same blueprint - it would be very inefficient to produce a new blueprint for every individual bicycle they manufactured.

In object-oriented software, it's also possible to have many objects of the same kind that share characteristics: rectangles, employee records, video clips and so on. Like the bicycle manufacturers, you can take advantage of the fact that objects of the same kind are similar and you can create a blueprint for those objects. Software "blueprints" for objects are called classes.

Hence, a class is a data structure that contains data members (constants, fields, and events), function members (methods, properties, indexers, operators, constructors, and destructors), and nested types. Class types support inheritance, a mechanism whereby derived classes can extend and specialize base classes.

Definition: A class is a blueprint or prototype that defines a group of data items called fields or variables and the associated functions or methods that perform operations on this data. Hence, classes are software blueprints for objects.

The Benefit of Classes:

Objects provide the benefit of modularity and information hiding. Classes provide the benefit of reusability. Bicycle manufacturers reuse the same blueprint repeatedly to build many bicycles. Software programmers use the same class repeatedly to create many objects.

When you write a C# program, you design and construct a set of classes. Then, when your program runs, instances of those classes are created and discarded as needed. Your task, as a C# programmer, is to create the right set of classes to accomplish what your program needs to accomplish.

The C# environment comes with a library of classes that implement a lot of the basic behaviour you need - not only for basic programming tasks (classes to provide basic math functions, arrays, strings, and so on), but also for graphics and networking behaviour. A class library is a set of classes.

Because each instance of a class can have different values for its variables, each variable is called an instance variable.

Class declarations are used to define new reference types. C# supports single inheritance only, but a class may implement multiple interfaces.

Class members can include constants, fields, methods, properties, indexers, events, operators, constructors, destructors, and nested type declaration.

The differentiation between classes and objects is often the source of some confusion. In the real world, it's obvious that classes are not themselves the objects that they describe - a blueprint of a bicycle is not a bicycle. However, it's a little more difficult to differentiate classes and objects in software. This is partially because software objects are electronic models of real-world objects or abstract concepts in the first place. People use the term "object" inconsistently and use it to refer to both classes and instances.

So far, a class's methods and variables don't exist yet. You must create an instance from the class before you can call the methods and before the variables can have any values. In comparison, an object's methods and variables actually exist and you can use it. You can send the object a message and it will respond by performing the method and perhaps modifying the values of the variables.

An instance of a class is another word for an actual object. If the class is the general representation of an object, an instance is its concrete representation.

3.5 What is Inheritance?

Classes inherit variables and methods from their superclass. Inheritance provides a powerful and natural mechanism for organising and structuring software programs.

Generally speaking, objects are defined in terms of classes. You know a lot about an object by knowing its class. Even if you don't know what a penny-farthing is, if I told you it was a bicycle, you would know that it had two wheels, handle bars and pedals.

Object-oriented systems take this a step further and allow classes to be defined in terms of other classes. For example, mountain bikes, racing bikes and tandems are all different kinds of bicycles. In object-oriented terminology, mountain bikes, racing bikes and tandems are all subclasses of the bicycle class. Similarly, the bicycle class is the superclass of mountain bikes, racing bikes and tandems.

Each subclass inherits attribute (in the form of variable declarations) from the superclass. Mountain bikes, racing bikes and tandems share some attributes: cadence, speed and the like. In addition, each subclass inherits methods from the superclass. Mountain bikes, racing bikes and tandems share some methods: braking and changing pedaling speed.

However, subclasses are not limited to the state and behaviours provided to them by their superclass. Subclasses can add variables and methods to the ones they inherited from the superclass. Tandem bicycles have two seats and two sets of handle bars; some mountain bikes have an extra set of gears with a lower gear ratio.

Subclasses can also override inherited methods and provide specialised implementations for those methods. For example, if you had a mountain bike with an extra set of gears, you would override the "change gears" method so that the rider could actually use those new gears.

You are not limited to just one layer of inheritance - the inheritance tree, or class hierarchy, can be as deep as needed. Methods and variables are inherited down through the levels. The further down in the hierarchy a class appears, the more specialised its behaviour.

At the top of the C# class hierarchy is the class Object; all classes inherit from this one superclass. Object is the most general class in the hierarchy; it defines methods inherited by all the classes in the C# class hierarchy.

You can think of a class hierarchy as defining very abstract concepts at the top of the hierarchy and those ideas becoming more concrete the farther down the chain of superclasses you go.

What if your class defines entirely new behaviour, and isn't really a subclass of another class? Your class can also inherit directly from Object, which still allows it to fit neatly into the C# class hierarchy. In fact, if you create a class definition that doesn't indicate its superclass in the first line, C# automatically assumes that you are inheriting from Object. The MotorCycle class you created, inherited from Object.

Each C# class can have only one superclass i.e. single inheritance.

The Benefit of Inheritance:

- Subclasses provide specialised behaviours from the basis of common elements provided by the superclass. With inheritance, programmers can reuse the code in the superclass many times.
- Programmers can implement superclasses that define "generic" behaviours (called abstract classes). The essence of the superclass is defined and may be partially implemented but much of the class is left undefined and non- implemented. Other programmers fill in the details with specialised subclasses.

4. Class and Object

4.1 Class Declaration

The use of a class tells the compiler that it's a reference type. All declarations of data members and methods take place between the opening and closing curly brackets. As in Java, the classes are defined and implemented in the same place. Class methods are invoked by using the dot notation. The data members in C# classes are set to certain default values, 0 for numeric types, false for Boolean, '\0' for chars and null for references.

A *class-declaration* is a *type-declaration* that declares a new class.

class-declaration:
attributes$_{opt}$ *class-modifiers*$_{opt}$ `class` *identifier* *class-base*$_{opt}$ *class-body* ;$_{opt}$

A *class-declaration* consists of an optional set of *attributes*, followed by an optional set of *class-modifiers*, followed by the keyword `class` and an *identifier* that names the class, followed by an optional *class-base* specification, followed by a *class-body*, optionally followed by a semicolon.

Some example classes:

```
// MotorCycle.cs
using System;
public class MotorCycle {
  string colour, make;
  bool engineState;
  public MotorCycle() {
    Console.WriteLine("Constructor called");
  }
  public void startEngine() {
    if (engineState)
      Console.WriteLine("Engine is Already ON");
    else {
      engineState = true;
      Console.WriteLine("Engine is now ON");
    }
  }
  public void showAttr() {
    Console.WriteLine("Make   is = " + make);
    Console.WriteLine("Colour is = " + colour);
    if (engineState)
      Console.WriteLine("Engine is Running");
    else
      Console.WriteLine("Engine is Idle");
  }
  public static void Main(String[] args) {
    MotorCycle m = new MotorCycle();
    m.make = "Yamaha";
    m.colour = "black";
    m.startEngine();
    m.showAttr();
    m.startEngine();
  }
}


//Account.cs
using System;
```

```
class Account {
  private double balance; // data member
  public Account(){
    balance = 0.0;
  }
  public Account(double amt){
    balance = amt;
  }
  public bool Deposit(double amt){
    //deposit cash
    if (amt <= 0.0)
      return false;
    else {
      balance += amt;
      return true;
    }
  }
  public bool Withdraw(double amt){
    //withdraw cash
    if ((balance-amt) < 0.0)
      return false;
    else
      balance -=amt;
    return true;
  }
  public double QueryBalance(){
    return balance;
  }
}
public class Test {
  public static void Main(String[] args){
    Account a1 = new Account();
    Account a2 = new Account(3000.00);
  }
}
```

4.1.1 Class modifiers

A *class-declaration* may optionally include a sequence of class modifiers:

class-modifier:

```
new
public
protected
internal
private
abstract
sealed
```

It is an error for the same modifier to appear multiple times in a class declaration.

The new modifier is only permitted on nested classes. It specifies that the class hides an inherited member by the same name.

The public, protected, internal, and private modifiers control the accessibility of the class. Depending on the context in which the class declaration occurs, some of these modifiers may not be permitted.

The abstract and sealed modifiers are discussed in the following sections.

4.1.1.1 Abstract classes

The abstract modifier is used to indicate that a class is incomplete and intended only to be a base class of other classes. An abstract class differs from a non-abstract class in the following ways:

- An abstract class cannot be instantiated, and it is an error to use the new operator on an abstract class. While it is possible to have variables and values whose compile-time types are abstract, such variables and values will necessarily either be null or contain references to instances of non-abstract classes derived from the abstract types.
- An abstract class is permitted (but not required) to contain abstract methods and accessors.
- An abstract class cannot be sealed.

When a non-abstract class is derived from an abstract class, the non-abstract class must include actual implementations of all inherited abstract methods and accessors. Such implementations are provided by overriding the abstract methods and accessors. In the example

```
abstract class A {
        public abstract void F();
}
abstract class B: A {
        public void G() {}
}
class C: B {
        public override void F() {
                // actual implementation of F
        }
}
```

the abstract class A introduces an abstract method F. Class B introduces an additional method G, but doesn't provide an implementation of F. B must therefore also be declared abstract. Class C overrides F and provides an actual implementation. Since there are no outstanding abstract methods or accessors in C, C is permitted (but not required) to be non-abstract. You should note that the presence of an abstract method in a class doesn't implicitly make the class abstract. You have to use the abstract modifier on the class as well. The following example illustrates this:

```
public class Account {
  public abstract bool withdraw(double amt);
}
```

When we compile this we get an error:

Test.cs(2,24): error CS0513: 'Account.withdraw(double)' is abstract but it is cntained in nonabstract class 'Account'

The correct way to write is as follows:

```
public abstract class Account {

  public abstract bool withdraw(double amt);

}
```

Implementing classes could be SavingsAccount, CurrentAccount.

4.1.1.2 Sealed classes

The sealed modifier (similar to final in Java) is used to prevent derivation from a class. An error occurs if a sealed class is specified as the base class of another class.

A sealed class cannot also be an abstract class.

The sealed modifier is primarily used to prevent unintended derivation, but it also enables certain run-time optimizations.

The sealed modifier cannot be used on methods.

Example:

```
using System;
sealed class MyPoint {
  public MyPoint(int x,int y) {
    this.x =x;
    this.y =y;
  }
  private int X;
  public int x {
    get {
      return this.X;
    }
    set {
      this.X =value;
    }
  }
  private int Y;
  public int y {
    get {
      return this.Y;
    }
    set {
      this.Y = value;
    }
  }
}
class SealedApp {
  public static void Main() {
    MyPoint pt = new MyPoint(6,16);
    Console.WriteLine("x = {0}, y = {1}", pt.x, pt.y);
  }
}
```

Note that we used the *private* access modifier on the internal class members *X* and *Y*. Using the *protected* modifier would result in a warning from the compiler because of the fact that protected members are visible to derived classes and, as you now know, sealed classes don't have any derived classes.

4.1.1.3 Inner Classes

We can have inner classes as shown in the program IC.cs

```
using System;
public class A {
  public class B {
    public class C {
      public C() {
        Console.WriteLine("In C");
      }
    }
  }
  static void Main() {
    A.B.C c = new A.B.C();
  }
}
```

4.1.2 Class base specification

A class declaration may include a *class-base* specification, which defines the direct base class of the class and the interfaces implemented by the class.

```
class-base:
    :                                                    class-type
    :                                           interface-type-list
    :  class-type  ,  interface-type-list

interface-type-list:
    interface-type
    interface-type-list  ,  interface-type
```

4.1.2.1 Base classes

When a *class-type* is included in the *class-base*, it specifies the direct base class of the class being declared. If a class declaration has no *class-base*, or if the *class-base* lists only interface types, the direct base class is assumed to be object. A class inherits members from its direct base class.

In the example:

```
class A {}
class B: A {}
```

class A is said to be the direct base class of B, and B is said to be derived from A. Since A does not explicitly specify a direct base class, its direct base class is implicitly object.

The direct base class of a class type must be at least as accessible as the class type itself. For example, it is an error for a public class to derive from a private or internal class (see example Assembly2.cs later on).

The base classes of a class are the direct base class and its base classes. In other words, the set of base classes is the transitive closure of the direct base class relationship. Referring to the example above, the base classes of B are A and object.

Except for class object, every class has exactly one direct base class. The object class has no direct base class and is the ultimate base class of all other classes.

When a class B derives from a class A, it is an error for A to depend on B. A class *directly depends on* its direct base class (if any) and *directly depends on* the class within which it is immediately nested (if any). Given this definition, the complete set of classes upon which a class depends is the transitive closure of the *directly depends on* relationship.

The example:

```
class A: B {}
class B: C {}
class C: A {}
```

is in error because the classes circularly depend on themselves. Likewise, the example

```
class A: B.C {}
class                      B:                      A                     {
    public                   class                   C                  {}
}
```

is in error because A depends on B.C (its direct base class), which depends on B (its immediately enclosing class), which circularly depends on A.

Note that a class does not depend on the classes that are nested within it. In the example

```
      class                               A                                  {
         class                     B:                     A                {}
      }
```

B depends on A (because A is both its direct base class and its immediately enclosing class), but A does not depend on B (since B is neither a base class nor an enclosing class of A). Thus, the example is valid.

It is not possible to derive from a sealed class. In the example

```
      sealed class A {}
      class B: A {}        // Error, cannot derive from a sealed class
```

class B is in error because it attempts to derive from the sealed class A.

4.1.2.2 Internal class

Internal members are accessible only within files in the same assembly. A common use of internal access is in component-based development because it enables a group of components to cooperate in a private manner without being exposed to the rest of the application code. For example, a framework for building graphical user interfaces could provide Control and Form classes that cooperate using members with internal access. Since these members are internal, they are not exposed to code that is using the framework.

It is an error to reference a member with internal access outside the assembly within which it was defined.

This example contains two files, Assembly1.cs and Assembly2.cs. The first file contains an internal base class, BaseClass. In the second file, an attempt to access the member of the base class will produce an error.

File Assembly1.cs:

```
// compile with /target:library
internal class BaseClass {
  public static int IntM = 0;
}
```

File Assembly2.cs:

```
// compile with /target:exe /reference:Assembly1.dll
public class TestAccess {
  public static void Main() {
    // error, BaseClass not visible outside assembly
    BaseClass myBase = new BaseClass();
  }
}
```

4.1.2.3 Interface implementations

A *class-base* specification may include a list of interface types, in which case the class is said to implement the given interface types.

4.1.3 Class body

The *class-body* of a class defines the members of the class.

class-body:
{ *class-member-declarations$_{opt}$* }

4.1.3.1 Class members

The members of a class consist of the members introduced by its *class-member-declaration*s and the members inherited from the direct base class.

class-member-declarations:
class-member-declaration
class-member-declarations class-member-declaration

class-member-declaration:
constant-declaration
field-declaration
method-declaration
property-declaration
event-declaration
indexer-declaration
operator-declaration
constructor-declaration
destructor-declaration
static-constructor-declaration
type-declaration

The members of a class are divided into the following categories:

- Constants, which represent constant values associated with the class.
- Fields, which are the variables of the class.
- Methods, which implement the computations and actions that, can be performed by the class.
- Properties, which define named attributes and the actions associated with reading and writing those attributes.
- Events, which define notifications that, are generated by the class.
- Indexers, which permit instances of the class to be indexed in the same way as arrays.
- Operators, which define the expression operators that, can be applied to instances of the class.
- Instance constructors, which implement the actions, required to initialize instances of the class.
- Destructors, which implement the actions to perform before instances of the class, are permanently discarded.
- Static constructors, which implement the actions, required to initialize the class itself.
- Types, which represent the types that, are local to the class.

Members that contain executable code are collectively known as the *function members* of the class. The function members of a class are the methods, properties, indexers, operators, constructors, and destructors of the class.

A *class-declaration* creates a new declaration space, and the *class-member-declarations* immediately contained by the *class-declaration* introduce new members into this declaration space. The following rules apply to *class-member-declaration*s:

- Constructors and destructors must have the same name as the immediately enclosing class. All other members must have names that differ from the name of the immediately enclosing class.
- The name of a constant, field, property, event, or type must differ from the names of all other members declared in the same class.
- The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the *signature* of a method must differ from the signatures of all other methods declared in the same class.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same class.
- The signature of an operator must differ from the signatures of all other operators declared in the same class.

The inherited members of a class are specifically not part of the declaration space of a class. Thus, a derived class is allowed to declare a member with the same name or signature as an inherited member (which in effect hides the inherited member).

4.1.3.2 Signature

Methods, constructors, indexers, and operators are characterized by their signatures:

The signature of a method consists of the name of the method and the number, modifiers, and types of its formal parameters. The signature of a method specifically does not include the return type.

The signature of a constructor consists of the number, modifiers, and types of its formal parameters.

The signature of an indexer consists of the number and types of its formal parameters. The signature of an indexer specifically does not include the element type.

The signature of an operator consists of the name of the operator and the number and types of its formal parameters. The signature of an operator specifically does not include the result type.

4.1.3.3 Constructors

A constructor is a method, which is called when an object of a class type is constructed, and is usually used for initialisation. A constructor method has several characteristics:

- It has the same name as the class name
- It has no return type
- It does not return any value.

Java Programmers should note that: If you attempt to prefix a constructor with a type, the compiler will emit an error stating that you cannot define members with the same names as the enclosing type.

In the example VarTest.cs:

```
class VarTest {
  public VarTest() {

  }
  static void Main() {
    VarTest t;
  }
}
```

If you compile the above program, the C# compiler will warn you that the variable t has been declared but is never used in the application.

Constructor Initializers:

All C# object constructors—with the exception of the *System.Object* constructors—include an invocation of the base class's constructor immediately before the execution of the first line of the constructor. These constructor initializers enable you to specify which class and which constructor you want called. This takes two forms:

- An initializer of the form *base(.)* enables the current class's base class constructor—that is, the specific constructor implied by the form of the constructor called—to be called.
- An initializer taking the form *this(.)* enables the current class to call another constructor defined within itself. This is useful when you have overloaded multiple constructors and want to make sure that a default constructor is always called.

```
using System;
class A {
  public A() {
    Console.WriteLine("A");
  }
}
class B : A {
  public B() : base() {
    Console.WriteLine("B");
  }
}
class BaseDefaultInitializerApp {
  public static void Main() {
    B b = new B();
  }
}
```

Another example, is when I have two classes: *A* and *B*. This time, class *A* has two constructors, one that takes no arguments and one that takes an *int*. Class *B* has one constructor that takes an *int*. The problem arises in the construction of class *B*. How do I ensure that the desired class *A* constructor will be called? By explicitly telling the compiler which constructor I want called in the initalizer list, as below:

```
using System;
class A {
  public A() {
    Console.WriteLine("A");
  }
  public A(int foo) {
    Console.WriteLine("A = {0}", foo);
  }
}
class B : A {
  public B(int foo) : base(foo) {
    Console.WriteLine("B = {0}", foo);
  }
}
class DerivedInitializer2App {
  public static void Main() {
    B b = new B(42);
  }
}
```

An example of this(.) is now provided through the example ThisEx.cs

```
// ThisEx.cs
using System;
class Tmp {
  public Tmp() {
    Console.WriteLine("In Tmp()");
  }
  public Tmp(int i) : this() {
    Console.WriteLine("In Tmp with int");
  }
  static void Main() {
    Tmp t = new Tmp(10);
  }
}
```

You can use overloading to create several constructors for a class; which one will get called depends on the arguments you give to new.

It's to be noted that:

- Constructors are not inherited. Thus, a class has no other constructors than those that are actually declared in the class. If a class contains no constructor declarations, a default constructor is automatically provided. The default constructor simply invokes the parameterless constructor of the direct base class. If the direct base class does not have an accessible parameterless constructor, an error occurs.
- If you write a class with only one constructor with parameters and use it while creating an object, then a default constructor need not be given.
- If you write a class with only one constructor with parameters and try to create an object using a default constructor (not given by you) then the program does not compile. As in Java, it's advisable to write a default constructor for every class you write.
- Constructors are usually public, but can also be private or protected. When a class declares only private constructors, it is not possible for other classes to derive from the class or create instances of the class (an exception being classes nested within the class). Private constructors are commonly used in classes that contain only static members. For example:

```
public class Trig {
        private Trig() {}               // Prevent instantiation
        public const double PI = 3.14159265358979323846;
        public static double Sin(double x) {...}
        public static double Cos(double x) {...}
        public static double Tan(double x) {...}
}
```

The Trig class provides a grouping of related methods and constants, but is not intended to be instantiated. It therefore declares a single private constructor. Note that at least one private constructor must be declared to suppress the automatic generation of a default constructor (which always has public access).

In the Tree.cs program below – Java programmers should note that the name of the program can be anything you want (in any case too). All classes can be public or otherwise. Also note that in the program there can be only one and only one Main() method.

```
using System;
public class GF {
  public GF() {
    Console.WriteLine("In GF");
  }
}
public class F : GF { // : is similar to extends in Java
  public F() {
    Console.WriteLine("In F");
  }
}
public class S : F {
  public S() {
    Console.WriteLine("In S");
  }
  public static void Main(String[] args){
    S son = new S();
  }
}
```

4.1.3.4 Calling Base Class Constructors

```
public class CheckAccount : Account {
  public CheckAccount(double amt) : base(amt) {
  }
}
```

Here, if a CheckAccount is created with a double being passed in to the constructor, that double is passed on to the base class constructor. The compiler sees base(amt) as meaning, “Pass amt through to the base class constructor which takes one double as an argument. In this case, Account has a suitable constructor, so the compiler can find and use it.

4.1.3.5 Static Constructors

Static constructors implement the actions required to initialize a class i.e the constructor is called when the class is loaded. Static constructors are declared using *static-constructor-declarations*:

> *static-constructor-declaration:*
> *attributes*$_{opt}$ static *identifier* ( ) *block*

A *static-constructor-declaration* may include set of *attributes*.

The *identifier* of a *static-constructor-declarator* must name the class in which the static constructor is declared. If any other name is specified, an error occurs.

The *block* of a static constructor declaration specifies the statements to execute in order to initialize the class. This corresponds exactly to the *block* of a static method with a void return type.

Static constructors are not inherited.

Static constructors are invoked automatically, and cannot be invoked explicitly. The exact timing and ordering of static constructor execution is not defined, though several guarantees are provided:

- The static constructor for a class is executed before any instance of the class is created.
- The static constructor for a class is executed before any static member of the class is referenced.
- The static constructor for a class is executed before the static constructor of any of its derived classes are executed.
- The static constructor for a class never executes more than once.

- The static constructor cannot have parameters.
- There’s no such thing as a static destructor.

The example SC1.sc

```
using System;
class SC1 {
  static void Main() {
    A.F();
    B.F();
  }
}
class A {
  static A() {
    Console.WriteLine("Init A");
  }
  public static void F() {
    Console.WriteLine("A.F");
  }
}
class B {
  static B() {
    Console.WriteLine("Init B");
  }
  public static void F() {
    Console.WriteLine("B.F");
  }
}
```

could produce either the output:

```
Init                                                                        A
A.F
Init                                                                        B
B.F
```

or the output:

```
Init                                                                        B
Init                                                                        A
A.F
B.F
```

because the exact ordering of static constructor execution is not defined.

The example SC2.cs

```
using System;
class SC2 {
  static void Main() {
    Console.WriteLine("1");
    B.G();
    Console.WriteLine("2");
  }
}
class A {
  static A() {
    Console.WriteLine("Init A");
  }
}
class B: A {
  static B() {
    Console.WriteLine("Init B");
  }
  public static void G() {
    Console.WriteLine("B.G");
  }
}
```

is guaranteed to produce the output:

```
1
Init A
Init B
B.G
2
```

because the static constructor for the class A must execute before the static constructor of the class B, which derives from it.

In the example SRO.cs below, the screen resolution fields are static and read-only, and there's a static constructor.

```
using System;

class GraphicsPackage {
  public static readonly int ScreenWidth;
  public static readonly int ScreenHeight;
  static GraphicsPackage() (
    // Code would be here to
    // calculate resolution.
    ScreenWidth = 1024;
    ScreenHeight = 768;
  }
}
class ReadOnlyApp {
  public static void Main() {
    Console.WriteLine("Width = {0}, Height = {1}",
                      GraphicsPackage.ScreenWidth,
                      GraphicsPackage.ScreenHeight);
  }
}
```

4.1.3.6 Destructors

Destructors are also called a finalizer. C# destructors are very similar to Java's finalize() method, and have all the same disadvantages.

Destructors implement the actions required to destruct instances of a class. Destructors are declared using *destructor-declaration*s:

> *destructor-declaration:*
> *attributes*$_{opt}$ ~ *identifier* ( ) *block*

A *destructor-declaration* may include set of *attributes*.

The *identifier* of a *destructor-declarator* must name the class in which the destructor is declared. If any other name is specified, an error occurs.

The *block* of a destructor declaration specifies the statements to execute in order to initialize a new instance of the class. This corresponds exactly to the *block* of an instance method with a void return type.

Destructors are not inherited. Thus, a class has no other destructors than those that are actually declared in the class.

Destructors are invoked automatically, and cannot be invoked explicitly. An instance becomes eligible for destruction when it is no longer possible for any code to use the instance. Execution of the destructor or destructors for the instance may occur at any time after the instance becomes eligible for destruction. When an instance is destructed, the destructors in an inheritance chain are called in order, from most derived to least derived.

The name of a class destructor is the class name preceded by a tilde (~). They are always public and have no return value. They do not take any arguments, so there can only ever be one for a class.

An example:

```
public class Test {
  private int x;
  public Test() {
    x = 0;
  }
  ~Test() {
    // tidy up
  }
}
```

4.1.3.7 Inheritance - Single

A class *inherits* (use: instead of extends as in Java) the members of its direct **base** class. Inheritance means that a class implicitly contains all members of its direct base class, except for the constructors and destructors of the base class. Some important aspects of inheritance are:

- Inheritance is transitive. If C is derived from B, and B is derived from A, then C inherits the members declared in B as well as the members declared in A.
- A derived class *extends* its direct base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member.
- Constructors and destructors are not inherited, but all other members are, regardless of their declared accessibility. However, depending on their declared accessibility, inherited members may not be accessible in a derived class.
- A derived class can *hide* inherited members by declaring new members with the same name or signature. Note however that hiding an inherited member does not remove the member—it merely makes the member inaccessible in the derived class.
- An instance of a class contains a copy of all instance fields declared in the class and its base classes, and an implicit conversion exists from a derived class type to any of its base class types. Thus, a reference to a derived class instance can be treated as a reference to a base class instance.
- A class can declare virtual methods, properties, and indexers, and derived classes can override the implementation of these function members. This enables classes to exhibit polymorphic behavior wherein the actions performed by a function member invocation vary depending on the run-time type of the instance through which the function member is invoked.

4.1.3.8 Accessing Base Class Members

You may want to call a method in a class' base class from time to time. You can do this using the **base** keyword. base is in many ways the equivalent of Java's super keyword.

4.1.3.9 The this Reference

A class method is always called in the context of some object. Suppose you wanted to know which object has called a method – you can get at this via the keyword **this**, which is a reference to the object that called the method.

4.1.3.10 The new modifier

A *class-member-declaration* is permitted to declare a member with the same name or signature as an inherited member. When this occurs, the derived class member is said to *hide* the base class member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived class member can include a new modifier to indicate that the derived member is intended to hide the base member.

If a new modifier is included in a declaration that doesn't hide an inherited member, a warning is issued to that effect. This warning is suppressed by removing the new modifier.

It is an error to use the new and override modifiers in the same declaration.

Use the **new** modifier to explicitly hide a member inherited from a base class. To hide an inherited member, declare it in the derived class using the same name, and modify it with the **new** modifier.

Consider the following class:

```
public class MyBaseC  {
  public int x;
  public void Invoke();
}
```

Declaring a member with the name M in a derived class, for example:

```
public class MyDerivedC : MyBaseC {
  new public void Invoke();
}
```

will hide the method Invoke() in the base class. However, the field x will not be affected because it is not hidden by a similar name.

In the example below (MyDerivedC.cs) a base class, MyBaseC, and a derived class, MyDerivedC, use the same field name x, thus hiding the value of the inherited field. The example demonstrates the use of the **new** modifier. It also demonstrates how to access the hidden members of the base class by using the fully qualified names.

```
// The new modifier
using System;
public class MyBaseC {
  public static int x = 55;
  public static int y = 22;
}
public class MyDerivedC : MyBaseC {
  new public static int x = 100;   // Name hiding
  public static void Main() {
    // Display the overlapping value of x:
    Console.WriteLine(x);
    // Access the hidden value of x:
    Console.WriteLine(MyBaseC.x);
    // Display the unhidden member y:
    Console.WriteLine(y);
  }
}
```

Output

```
100
55
22
```

If you remove the **new** modifier, the program will still compile and run, but you will get the warning:

SC0108: The keyword new is required on 'MyDerivedC.x' because it hides inherited member 'MyBaseC.x'.

4.1.3.11 Casting between Types

Assuming a base class named *Employee* and a derived class named *ContractEmployee*, the following code works because there's always an implied upcast from a derived class to its base class:

```
class Employee { }
class ContractEmployee : Employee { }
class CastExample1 {
  public static void Main () {
    Employee e = new ContractEmployee();
  }
}
```

However, the following is illegal, because the compiler cannot provide an implicit downcast:

```
class Employee { }
class ContractEmployee : Employee { }
class CastExample2 {
  public static void Main () {
    ContractEmployee ce = new Employee(); // Won't compile.
  }
}
```

Remember: a derived class can be used in place of its base class.

In the next example, the program compiles but throws a throws a `System.InvalidCastException` at run-time.

```
class Employee { }
class ContractEmployee : Employee { }
class CastExample3 {
  public static void Main () {
    //Downcast will fail.
    ContractEmployee ce = (ContractEmployee)new Employee();
  }
}
```

There is one other way of casting objects: using the ***as*** keyword. The advantage to using this keyword instead of a cast is that if the cast is invalid, you don't have to worry about an exception being thrown. What will happen instead is that the result will be *null*. Here's an example:

```
using System;
class Employee { }
class ContractEmployee : Employee { }
class CastExample5 {
  public static void Main () {
    Employee e = new Employee();
    Console.WriteLine("e = {0}",
                   e == null ? "null" : e.ToString());
    ContractEmployee c  = e as ContractEmployee;
    Console.WriteLine("c = {0}",
                       c == null ? "null" : c.ToString());
  }
}
```

If you run this example, you'll see the following result:

```
c:>CastExample5
e = Employee
c = null
```

4.1.3.12 Access modifiers

A ***class-member-declaration*** can have any one of the five possible types of declared accessibility: public, protected internal, protected, internal, or private. Except for the protected internal combination, it is an error to specify more than one access modifier. When a ***class-member-declaration*** does not include any access modifiers, the declaration defaults to private declared accessibility.

Each member of a class has a form of accessibility, as under:

- `public` members are available to all code;
- `protected` members are accessible only from derived classes;
- `internal` members are accessible only from within the same assembly;
- `protected internal` members are accessible only from derived classes within the same assembly;
- `private` members are accessible only from the class itself.

The point to remember is that the access modifiers are associated with members or types (classes etc.) and are not allowed on namespaces.

The examples below should illustrate the above:

```
// Am.cs
using System;
public class Dad {
  public bool diamond;
  protected bool gold;
  protected internal bool silver;
  internal bool car;
  bool house;
}
class Son : Dad {
  public static void Main(String[] args) {
    Son s = new Son();
    s.diamond = true;
    s.gold    = true;
    s.silver  = true;
    s.car     = true;
    s.house   = true;  // Here private is inaccessible
    Console.WriteLine("diamond = {0}, gold = {1}", s.diamond, s.gold);
    Console.WriteLine("silver  = {0}, car  = {1}", s.silver,  s.car);
    Console.WriteLine("house = {0}", s.house);
  }
}
```

```
// AM2.cs
using System;
public class Dad {
  public bool diamond;
  protected bool gold;
  protected internal bool silver;
  internal bool car;
  bool house;
}
class Friend {
  public static void Main(String[] args) {
    Dad d = new Dad();
    d.diamond = true;
    d.gold    = true;  // protected inaccessible
    d.silver  = true;
    d.car     = true;
    d.house   = true;  // private inaccessible
    Console.WriteLine("diamond = {0}, gold = {1}", d.diamond, d.gold);
    Console.WriteLine("silver  = {0}, car  = {1}", d.silver,  d.car);
    Console.WriteLine("house = {0}", d.house);
  }
}
```

4.1.3.13 Restrictions on Using Accessibility Levels

When you declare a type, it is essential to see if that type has to be *at least as accessible as* another member or type. For example, the direct base class must be at least as accessible as the derived class.

The following declarations will result in a compiler error, because the base class BaseClass is less accessible than MyClass:

```
class BaseClass {...}
public class MyClass: BaseClass {...} // Error
```

4.1.3.14 Accessibility Domain

The example AM3.cs contains a top-level type, T1, and two nested classes, M1 and M2. The classes contain fields with different declared accessibilities. In the Main method, a comment follows each statement to indicate the accessibility domain of each member. Notice that the statements that attempt to reference the inaccessible members are commented out. If you want to see the compiler errors caused by referencing an inaccessible member, remove the comments one at a time.

```
using System;
namespace MyNameSpace  {
  public class T1 {
    public static int myPublicInt;
    internal static int myInternalInt;
    private static int myPrivateInt = 0;
    public class M1 {
      public static int myPublicInt;
      internal static int myInternalInt;
      private static int myPrivateInt = 0;
    }
    private class M2 {
      public static int myPublicInt = 0;
      internal static int myInternalInt = 0;
      private static int myPrivateInt = 0;
    }
  }
  public class MainClass {
    public static int Main() {
       // Access to T1 fields:
       T1.myPublicInt = 1;      // Access is unlimited
       T1.myInternalInt = 2;    // Accessible only in current project
       // T1.myPrivateInt = 3;  // Error: inaccessible outside T1

       // Access to the M1 fields:
       T1.M1.myPublicInt = 1;     // Access is unlimited
       T1.M1.myInternalInt = 2;   // Accessible only in current project
       // T1.M1.myPrivateInt = 3; // Error: inaccessible outside M1

       // Access to the M2 fields:
       // T1.M2.myPublicInt = 1;   // Error: inaccessible outside T1
       // T1.M2.myInternalInt = 2; // Error: inaccessible outside T1
       // T1.M2.myPrivateInt = 3;  // Error: inaccessible outside M2
       return 0;
    }
  }
}
```

4.1.3.15 Virtual methods

One of the properties of inheritance is that we can use a derived class object through a base class reference, like this:

```
// Create a savings account

Account s1 = new CheckAccount(); //CheckAccount : Account

// Deposit and Withdraw

s1.Deposit(15000.00);

s1. Withdraw(2000.00);
```

As I've already mentioned, this works because a CheckAccount is an Account, so we can do everything to a CheckAccount that we can to the base class. In this case, we are simply calling the Deposit() and Withdraw() methods that CheckAccount inherits from Account.

Suppose that we want to provide CheckAccount with its own version of Withdraw(), because CheckAccounts can have an overdraft limit. We can code one up quite easily:

```
public class CheckAccount : Account {
  private double overdraftLimit = 500.00;
  public bool Withdraw(double amt) {
    if (balanceamt >= overdraftLimit) {
      base.Withdraw(amt);
      return true;
    }
    else
      return false;
  }
}
```

There's a problem with this, if we try to use our new version through an Account reference, as the compiler will insist on calling the one from Account, rather than from CheckAccount. This problem arises because the compiler has to decide which method to call, so it looks at the reference s1, sees that it is of type Account, and so calls the Account.Withdraw() method.

A **virtual method** is one where the decision on exactly which method to call is delayed until run-time, allowing the dynamic type of the reference to be used. You declare a method as virtual by using the virtual modifier in the base class:

```
public class Account {
  Virtual public bool Withdraw(double amt) {
  }
}
```

4.1.3.16 Override methods

When you override a virtual method in a derived class, you can use the **override** keyword to signal that you are overriding a virtual method.

```
public class CheckAcount : Account {
  private double overdraftLimit = 500.00;
  public override bool Withdraw(double amt) {
    if (balanceamt >= overdraftLimit) {
      base.Withdraw(amt);
      return true;
    }
    else
      return false;
  }
}
```

Let us wrap up this discussion with a consolidated example – VO.cs

```
// VO.cs
using System;
public class Account {
  public void Withdraw(double amt) {
    Console.WriteLine("Account.Withdraw method called...");
  }
}
public class CheckAccount : Account {
  public void Withdraw(double amt) {
    Console.WriteLine("CheckAccount.Withdraw method called...");
  }
  public static void Main(string[] args) {
    Account s1 = new CheckAccount();
    s1.Withdraw(5000.00);
  }
}
```

The output of the above is:

```
Account.Withdraw method called...
```

As mentioned earlier, the compiler has to decide which method to call, so it looks at the reference s1, sees that it is of type Account, and so calls the Account.Withdraw() method.

In the example VO1.cs, we use the virtual and override keywords to give a different ouput.

```
// VO1.cs
using System;
public class Account {
  virtual public void Withdraw(double amt) {
    Console.WriteLine("Account.Withdraw method called...");
  }
}
public class CheckAccount : Account {
  public override void Withdraw(double amt) {
    Console.WriteLine("CheckAccount.Withdraw method called...");
  }
  public static void Main(string[] args) {
    Account s1 = new CheckAccount();
    s1.Withdraw(5000.00);
  }
}
```

The output is:

```
CheckAccount.Withdraw method called...
```

Remember that the *virtual* keyword must be used on the base class's method, and the *override* keyword is used on the derived class's implementation of the method.

This example VO2.cs has the same method overridden multiple times.

```
using System;
class A {
  public virtual void MtdA() {
  Console.WriteLine("In A");
  }
}
class B : A {
  public override void MtdA() {
  Console.WriteLine("In B");
  }
}
class C : B {
  public override void MtdA() {
    Console.WriteLine("In C");
  }
}
class Tmp {
  static void Main() {
    A c = new C();
    c.MtdA();
  }
}
```

4.2 Interfaces

The **interface** keyword declares a reference type that has abstract members.

Interfaces are used to define a contract; a class or struct that implements the interface must adhere to this contract.

Interfaces can contain methods, properties, indexers, and events as members. They can't contain constants, fields (private data members), constructors and destructors or any type of static member.

All the members of an interface are public by definition, and the compiler will give you an error if you try to specify any other modifiers on interface members. The static and public modifiers are not permitted on interface methods.

C# interfaces are very much similar to Java interfaces. In C# we say : instead of implements, as in Java.

The example

```
interface Iexample {
        string this[int index] { get; set; }
        event EventHandler E;
        void F(int value);
        string P { get; set; }
}
public delegate void EventHandler(object sender, Event e);
```

shows an interface that contains an indexer, an event E, a method F, and a property P.

Interfaces may employ multiple inheritance. In the example below, the interface IComboBox inherits from both ITextBox and IListBox.

```
interface Icontrol {
        void Paint();
}
interface ITextBox: Icontrol {
        void SetText(string text);
}
interface IListBox: Icontrol {
        void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}
```

Classes and structs can implement multiple interfaces. In the example below, the class EditBox derives from the class Control and implements both IControl and IDataBound.

```
interface IdataBound {
        void Bind(Binder b);
}
public class EditBox: Control, IControl, IdataBound {
        public void Paint();
        public void Bind(Binder b) {...}
}
```

In the example above, the Paint method from the IControl interface and the Bind method from IDataBound interface are implemented using public members on the EditBox class. C# provides an alternative way of implementing these methods that allows the implementing class to avoid having these members be public. Interface members can be implemented by using a qualified name. For example, the EditBox class could instead be implemented by providing IControl.Paint and IDataBound.Bind methods.

```
public class EditBox: IControl, IdataBound {
        void IControl.Paint();
        void IDataBound.Bind(Binder b) {...}
}
```

Interface members implemented in this way are called “explicit interface member implementations” because each method explicitly designates the interface method being implemented.

Explicit interface methods can only be called via the interface. For example, the EditBox’s implementation of the Paint method can be called only by casting to the IControl interface.

```
class Test {
  static void Main() {
    EditBox editbox = new EditBox();
    editbox.Paint();    // error: EditBox does not have a Paint method
    IControl control = editbox;
    control.Paint();    // calls EditBox’s implementation of Paint
  }
}
```

4.3 Structs

A struct in C# is simply a composite data type, consisting of a number of elements (or members) of other types. The variables which make up a struct are called its members (fields in C#), and can be accessed using a simple dot notation. It's quite possible to nest references to structs, and you can use the dot notation to access all levels within the nested structure. You can also declare structs nested inside other structs. The list of similarities between classes and structs is long – structs can implement interfaces, and can have the same kinds of members as classes. Structs differ from classes in several important ways, however: structs are value types rather than reference types, and inheritance is not supported for structs. Struct values either are stored "on the stack" or "in-line". Careful programmers can enhance performance through judicious use of structs.

For example, the use of a struct rather than a class for a Point can make a large difference in the number of allocations. The program below creates and initializes an array of 100 points. With Point implemented as a class, the program instantiates 101 separate objects – one for the array and one each for the 100 elements.

```
class Point {
        public int x, y;
        public Point() {
                x = 0;
                y = 0;
        }
        public Point(int x, int y) {
                this.x = x;
                this.y = y;
        }
}
class Test {
        static void Main() {
                Point[] points = new Point[100];
                for (int i = 0; i < 100; i++)
                        points[i] = new Point(i, i*i);
        }
}
```

If Point is instead implemented as a struct, as in

```
struct Point {
        public int x, y;
        public Point(int x, int y) {
                this.x = x;
                this.y = y;
        }
}
```

then the test program instantiates just one object, for the array. The Point instances are allocated in-line within the array. Of course, this optimization can be mis-used. Using structs instead of classes can also make your programs fatter and slower, as the overhead of passing a struct instance by value is slower than passing an object instance by reference. There is no substitute for careful data structure and algorithm design.

If you pass a struct to a function, by default, the entire structure is copied onto the stack.

4.4 Enums

An enum type declaration defines a type name for a related group of symbolic constants (named integer constants). Enums are typically used when for "multiple choice" scenarios, in which a runtime decision is made from a number of options that are known at compile-time. Each of the named constants has a value, which by default starts at zero and increases by one for each succeeding member. You can give explicit values to any or all of the constants; any that you don’t specify get a value one more than the proceeding constant. The default type of the constants is int; you can declare enum which use other integral types like – byte, sbyte, ushort, short, uint, long, ulong.

```
enum Weekday : short (Mon, Tue, Wed);
```

The other example

```
enum Color {
        Red,
        Blue,
        Green
}
class Shape {
        public void Fill(Color color) {
                switch(color) {
                        case Color.Red:
                                ...
                                break;
                        case Color.Blue:
                                ...
                                break;
                        case Color.Green:
                                ...
                                break;
                        default:
                                break;
                }
        }
}
```

shows a Color enum and a method that uses this enum. The signature of the Fill method makes it clear that the shape can be filled with one of the given colors.

The use of enums is superior to the use of integer constants – as is common in languages without enums – because the use of enums makes the code more readable and self-documenting. The self-documenting nature of the code also makes it possible for the development tool to assist with code writing and other “designer” activities. For example, the use of Color rather than int for a parameter type enables smart code editors to suggest Color values.

4.5 Properties

In OO languages, the data members are normally private and access to them is through get and set (or accessor) methods. The drawback here is that one has to code the accessor methods for each data member and users have to remember to use them to access these data members.

C# has this idea of accessing data members through get and set code built into the language, in the form of **properties.** The difference between using get/set methods and properties is that to a user, using a property looks like they are getting direct access to the data, whereas in fact the compiler is mapping the call onto the get/set methods.

The success of rapid application development tools like Visual Basic can, to some extent, be attributed to the inclusion of properties as a first-class element. VB developers can think of a property as being field-like, and this allows them to focus on their own application logic rather than on the details of a component they happen to be using. On the face of it, this difference might not seem like a big deal, but modern component-oriented programs tend to be chockfull of property reads and writes. Languages with method-like usage of properties (e.g., o.SetValue(o.GetValue() + 1);) are clearly at a disadvantage compared to languages that feature field-like usage of properties (e.g., o.Value++;).

Properties are defined in C# using property declaration syntax. The first part of the syntax looks quite similar to a field declaration. The second part includes a get accessor and/or a set accessor. In the example below, the Button class defines a Caption property.

```
public class Button: Control {
  private string caption;
  public string Caption {
    get {
     return caption;
    }
    set {
      caption = value;
      Repaint();
    }
  }
}
```

Properties that can be both read and written, like the Caption property, include both get and set accessors. The get accessor is called when the property's value is read; the set accessor is called when the property's value is written. In a set accessor, the new value for the property is given in an implicit value parameter (the value represents the value passed in from the user).

Declaration of properties is relatively straightforward, but the true value of properties shows itself is in their usage rather than in their declaration. The Caption property can be read and written in the same way that fields can be read and written:

```
Button b = new Button();
b.Caption = "ABC";      // set
string s = b.Caption;   // get
b.Caption += "DEF";     // get & set
```

You can omit either the set or get clause. You don't have to return the value of a variable in a get clause, but can use any code you like to calculate or obtain the value of the property. This means that properties don't have to be tied to a data member but can represent dynamic data. Properties can be inherited and you can use the abstract and virtual modifiers with them, so that derived classes can be required to implement their own versions of property methods. In addition, the static modifier can be used to create properties that belong to classes as opposed to individual objects.

4.6 Assignment

Use a generic, higher level **IAccount** interface and an abstract class **BankAccount** to collect all features common to all bank accounts. The BankAccount class has account number and balance as instance variables. It has the methods for deposit and withdrawal. The **SavingsAccount** and **BonusSaverAccount** are both savings accounts through which you can make deposits and withdrawals. However, the BonusSaverAccount is designed with incentives to encourage faster savings accumulation. Both types of accounts earn interest, but the BonusSaverAccount earns more. Furthermore, there is a financial penalty every time a withdrawal is made from a BonusSaverAccount.

The SavingsAccount and CurrentAccount classes are derived from BankAccount and BonusSaverAccount is derived from SavingsAccount.

The CurrentAccount has an overridden withdrawal method that makes use of overdraft protection.

```
// Assignment.cs
namespace BankSystem {
using System;
public interface IAccount {
  // Two abstract methods:
  void    deposit(double amount);
  bool withdrawal(double amount);
}
public abstract class BankAccount : IAccount {
  // Fields:
  private int    account;
  private double balance;
  // Constructor:
  public BankAccount(int accountNum, double initialBal) {
    account = accountNum;
    balance = initialBal;
  }
  // Property implementation:
  public int Account {
    get {
      return account;
    }
    set {
      account = value;
    }
  }
  public double Balance {
    get {
      return balance;
    }
    set {
      balance = value;
    }
  }
  public void deposit(double amount) {
    balance += amount;
    Console.WriteLine("Deposit into account " + account);
    Console.WriteLine("Amount: " + amount);
    Console.WriteLine("New Balance: " + balance);
  }
  public virtual bool withdrawal(double amount) {
    bool result = false;
    Console.WriteLine("Withdrawal from Account " + account);
    Console.WriteLine("Amount: " + amount);
    if (amount > balance)
      Console.WriteLine("Insufficient funds");
    else {
      balance -= amount;
      Console.WriteLine("New Balance: " + balance);
      result = true;
    }
    return result;
  }
}
class SavingsAccount : BankAccount {
  protected double rate;
  public SavingsAccount (int accountNum, double initialBal,
                       double interestRate) : base(accountNum, initialBal) {
    rate = interestRate;
  }
  public virtual void addInterest() {
    // using the Property names: Account and Balance
    Balance += Balance * rate;
    Console.WriteLine("Interest added to account: " + Account);
    Console.WriteLine("New balance: " + Balance);
  }
}
class BonusSaverAccount : SavingsAccount {
  private const int    PENALTY   = 25;
  private const double BONUSRATE =  0.02;
  public BonusSaverAccount (int accountNum, double initialBal,
                            double interestRate)
                          : base(accountNum, initialBal, interestRate){
  }
  public override bool withdrawal(double amount) {
    Console.WriteLine("Penalty incurred: " + PENALTY);
    return base.withdrawal(amount+PENALTY);
  }
  public override void addInterest() {
    Balance += Balance * (rate + BONUSRATE);
    Console.WriteLine("Interest added to account: " + Account);
    Console.WriteLine("New balance: " + Balance);
  }
}
class CurrentAccount : BankAccount {
  private SavingsAccount overdraft;
  public CurrentAccount(int accountNum, double initialBal,
                      SavingsAccount protection) : base(accountNum, initialBal){
   overdraft = protection;
  }
  public override bool withdrawal(double amount) {
    bool result = false;
    if (! base.withdrawal(amount)) {
      Console.WriteLine("Using overdraft...");
      if (! overdraft.withdrawal(amount - Balance))
        Console.WriteLine("Overdraft source insufficient...");
      else {
        Balance = 0;
        Console.WriteLine("New Balance on account " + Account + ": " + Balance);
        result = true;
      }
    }
    return result;
  }
}
class Assignment {
  public static void Main(String[] args) {
    SavingsAccount savings =
      new SavingsAccount(101, 10000.00, 0.04);
    BonusSaverAccount bigSavings =
      new BonusSaverAccount(201, 10000.00, 0.02);
    CurrentAccount checking =
      new CurrentAccount(301, 500.00, savings);
    savings.deposit(1000.00);
    bigSavings.deposit(1000.00);
    savings.withdrawal(500.00);
    bigSavings.withdrawal(500.00);
    checking.withdrawal(501.00);
  }
}
}
```

4.7 Summary of Key Concepts

- An `abstract` class indicates that the class is incomplete and intended only to be a base class of other classes. An abstract class cannot be instantiated and cannot be sealed.
- The `sealed` modifier (similar to `final` in Java) is used to prevent derivation from a class. The `sealed` modifier cannot be used on methods.
- The direct base class of a class type must be at least as accessible as the class type itself.
- Internal members are accessible only within files in the same assembly.
- Constructors and destructors must have the same name as the immediately enclosing class. All other members must have names that differ from the name of the immediately enclosing class.
- If you attempt to prefix a constructor with a type, the compiler will emit an error stating that you cannot define members with the same names as the enclosing type.
- A constructor initializer of the form *base(.)* enables the current class's base class constructor—that is, the specific constructor implied by the form of the constructor called—to be called.
- An initializer taking the form *this(.)* enables the current class to call another constructor defined within itself. This is useful when you have overloaded multiple constructors and want to make sure that a default constructor is always called.
- Constructors are not inherited.
- It's advisable to write a default constructor for every class you write.
- Constructors are usually public, but can also be private or protected.
- Static constructors implement the actions required to initialize a class i.e. the constructor is called when the class is loaded.
- Static constructors are not inherited.
- Static constructors are invoked automatically, and cannot be invoked explicitly. The exact timing and ordering of static constructor execution is not defined, though several guarantees are provided:
    1. The static constructor for a class is executed before any instance of the class is created.
    2. The static constructor for a class is executed before any static member of the class is referenced.
    3. The static constructor for a class is executed before the static constructor of any of its derived classes are executed.
    4. The static constructor for a class never executes more than once.
    5. The static constructor cannot have parameters.
    6. There’s no such thing as a static destructor.

- Destructors are invoked automatically, and cannot be invoked explicitly. The name of a class destructor is the class name preceded by a tilde (~). They are always public and have no return value. They do not take any arguments, so there can only ever be one for a class.
- A derived class can *hide* inherited members by declaring new members with the same name or signature. Note however that hiding an inherited member does not remove the member—it merely makes the member inaccessible in the derived class.
- The **base** keyword is in many ways the equivalent of Java's super keyword.
- Use the **new** modifier to explicitly hide a member inherited from a base class.
- A derived class can be used in place of its base class.
- The ***as*** keyword can be used instead of a cast. The advantage over a cast is that if the cast is invalid, you don't have to worry about an exception being thrown. What will happen instead is that the result will be *null*.
- When a ***class-member-declaration*** does not include any access modifiers, the declaration defaults to `private` declared accessibility.
- The access modifiers are associated with members or types (classes etc.) and are not allowed on namespaces.
- A **virtual method** is one where the decision on exactly which method to call is delayed until run-time, allowing the dynamic type of the reference to be used. You declare a method as virtual by using the `virtual` modifier in the base class.
- When you override a virtual method in a derived class, you can use the **`override`** keyword to signal that you are overriding a virtual method.
- Remember that the *virtual* keyword must be used on the base class's method, and the *override* keyword is used on the derived class's implementation of the method.
- The **interface** keyword declares a reference type that has abstract members. Interfaces can contain methods, properties, indexers, and events as members. They can't contain constants, fields (private data members), constructors and destructors or any type of static member. All the members of an interface are public by definition, and the compiler will give you an error if you try to specify any other modifiers on interface members. The static and public modifiers are not permitted on interface methods.
- Structs are value types rather than reference types, and inheritance is not supported for structs.
- An enum type declaration defines a type name for a related group of symbolic constants (named integer constants). Each of the named constants has a value, which by default starts at zero and increases by one for each succeeding member. You can give explicit values to any or all of the constants; any that you don't specify get a value one more than the proceeding constant. The default type of the constants is `int`**.**
- The difference between using get/set methods and properties is that to a user, using a property looks like they are getting direct access to the data, whereas in fact the compiler is mapping the call onto the get/set methods.
- In a property set accessor, the new value for the property is given in an implicit value parameter (the value represents the value passed in from the user).
- You can omit either the set or get clause in case of a property.

5. Using .NET Base Classes

Microsoft has supplied a large number of base classes as part of the .NET framework. The base classes are accessible from C#, VB, C++ or any other .NET compliant language.

5.1 The WinCV Tool

WinCV is a class viewer tool, which can be used to examine the classes available in shared assemblies – including all the base classes. For each class, it lists the methods, properties, events and fields using a C# like syntax.

At the command prompt type **wincv**. You then type in a search string that is contained in the class you are looking for in the **Searching For** box. WinCv will then display a list of classes that contain the string in question in their names. Scan through the names to find the one that looks most suitable. The definition of the selected class is then displayed in the right hand pane. The left hand pane also indicates the namespace the class is contained in – you'll need this in order to be able to use that class in your C# code. The right hand pane of WinCV tells us which assembly the class is defined in. This information is useful to indicate if we need to link in any assemblies when we compile, in order to have access to that class.

5.2 StringBuilder class

This class is used to represent *mutable* strings. It is convenient for situations in which it is desirable to modify a string, perhaps by removing, replacing, or inserting characters, without creating a new string subsequent to each modification. It is used in conjunction with the String class to carry out modifications upon strings. The methods contained within this class do not return a new StringBuilder object unless specified otherwise. This class is defined in the namespace System.Text. It starts at a predefined size (16 characters by default) and grows dynamically as more string data is added.

```
using System;
using System.Text;
public class SBTest {
  public string SqrtIntFaster(int i) {
    StringBuilder buf = new StringBuilder(50);
    buf.Append("sqrt(").Append(i).Append(')');
    buf.Append(" = ").Append(Math.Sqrt(i));
    return buf.ToString();
  }
  static void Main() {
   SBTest sb = new SBTest();
   Console.WriteLine(sb.SqrtIntFaster(25));
  }
}
```

In the above program SBTest.cs, the StringBuilder constructor has the suggested starting size of 50. This size can increase as more characters are added. The Append() method appends a typed object to the end of the current StringBuilder. The ToString() method is overloaded and it converts a StringBuilder to a String. We have also used Sqrt() method of the Math class.

5.3 File and Folder operations

The .NET base classes include a number of classes that provide a rich set of functionality to access the file system, to read and write to files, to move or copy files around, or to explore folders to check what files are there. These classes are contained in the System.IO namespace.

5.3.1 Finding out information about a File

The program IBF.cs will demonstrate how to get information about a file, using some of the properties of the File class.

```
using System;
using System.IO;
public class IBF {
  public static void Main(String[] args) {
    File f = new File(@"C:\WINDOWS\BOOTLOG.TXT");
    Console.WriteLine("Connected to file..." + f.Name);
    Console.WriteLine("In Folder.........." + f.Directory);
    Console.WriteLine("Full Path.........." + f.FullName);
    Console.WriteLine("Is Directory......." + f.IsDirectory.ToString());
    Console.WriteLine("Is File............" + f.IsFile.ToString());
    Console.WriteLine("Last Write Time....." + f.LastWriteTime.ToString());
    Console.WriteLine("Size in bytes......" + f.Length);
  }
}
```

In general you can use a File object to connect to either a file or a folder, although if you connect to a folder then attempting to access those properties that don't make sense for a folder (such as Length or LastWriteTime) will raise an exception.

5.3.2 Listing Files in a Folder

To explore the contents of a folder, we need another base class – the Directory class, also in the System.IO namespace. Note that the .NET base classes generally refer to folders as *directories* in class and method names.

The example FIF.cs connects to the folder H:\pune-csharp and separately lists the files and folders in it.

```
using System;
using System.IO;
public class FIF {
  public static void Main(String[] args) {
    Directory d1 = new Directory(@"H:\pune-csharp");
    Console.WriteLine("Connected to folder..." + d1.Name);
    Console.WriteLine("Full Path............" + d1.FullName);
    Console.WriteLine("Is Directory........." + d1.IsDirectory.ToString());
    Console.WriteLine("Is File.............." + d1.IsFile.ToString());
    Console.WriteLine("Files contained in this folder:");
    File[] childfiles = d1.GetFiles();
    foreach (File childfile in childfiles)
      Console.WriteLine(childfile.Name);
    Console.WriteLine("Subfolders contained in this folder:");
    Directory[] childfolders = d1.GetDirectories();
    foreach (Directory childfolder in childfolders)
      Console.WriteLine(childfolder.Name);
  }
}
```

5.3.3 Copying and Deleting Files

We start off by binding to the H:\pune-csharp folder, and we create both a new empty file and a new empty subfolder there, as in example CFF.cs:

```
using System;
using System.IO;
public class CFF {
  public static void Main(String[] args) {
    Directory d1 = new Directory(@"H:\pune-csharp");
    d1.CreateSubdirectory("Talim");
    d1.CreateFile("Satish");
  }
}
```

Next, we bind to one of the files in the H:\pune-csharp folder, rename it and copy it, as in example RC.cs:

```
using System;
using System.IO;
public class RC {
  public static void Main(String[] args) {
    File f1 = new File(@"H:\pune-csharp\Satish");
    f1.CopyTo(@"H:\pune-csharp\Talim\NewSatish");
  }
}
```

Next we delete the file NewSatish and the folder Talim, as in the example DFF.cs:

```
using System;
using System.IO;
public class DFF {
  public static void Main(String[] args) {
    File f1 = new File(@"H:\pune-csharp\Satish");
    f1.Delete();
    Directory d1 = new Directory(@"H:\pune-csharp\Talim");
    d1.Delete(true);
  }
}
```

Here Delete(true) is used to remove directories, subdirectories, and contents; otherwise **false**.

5.3.4 Reading Text Files

The available classes are all derived from the class `System.IO.Stream`, which can represent any stream. To read text files we use the class `StreamReader`. The example RTF.cs demonstrates this.

```
using System;
using System.IO;
public class RTF {
  public static void Main(String[] args) {
    File f1 = new File(@"H:\CSHARPTEMP\Hello.cs");
    StreamReader sr = f1.OpenText();
    // continue reading until end of file
    string sLine;
    while ((sLine = sr.ReadLine()) != null)
      Console.WriteLine(sLine);
    sr.Close();
  }
}
```

5.3.5 Writing Text Files

For writing text files we use the StreamWriter class. The example WTF.cs explains this:

```
using System;
using System.IO;
public class WTF {
  public static void Main(String[] args) {
    StreamWriter sw = new StreamWriter(@"H:\CSHARPTEMP\TEMP.cs", false);
    sw.WriteLine("How do you find the C# Workshop?");
    sw.WriteLine("We now write some numbers and bool...");
    sw.WriteLine(6);
    sw.WriteLine(65.34);
    sw.WriteLine(true);
    sw.Close();
  }
}
```

The StreamWriter constructor takes two parameters: the full name of the file and a boolean that indicates whether data should be appended to the file. If this is false then the contents of the file will be overwritten by the StreamWriter. In either case, the file will be opened if it already exists or created if it does not.

5.3.6 Reading Binary Files

Here we use either the Stream or FileStream class. The program RBFS.cs demonstrates how to use the Stream class to read data. It opens a file and reads it, a byte at a time, each time displaying the numeric value of the byte read.

```
using System;
using System.IO;
public class RBFS {
  public static void Main(String[] args) {
    File f1 = new File(@"H:\CSHARPTEMP\Hello.cs");
    Stream s = f1.OpenRead();
    int iNext;
    while ((iNext = s.ReadByte()) != -1)
      Console.WriteLine(iNext.ToString());
    s.Close();
  }
}
```

5.3.7 Writing Binary Files

The program WBFFS.cs writes out a short text file that contains the letters FGHIJK followed by a carriage return-line feed combination.

```
using System;
using System.IO;
public class WBFFS {
  public static void Main(String[] args) {
    byte[] bytes = {70, 71, 72, 73, 74, 75, 13, 10};
    FileStream fs = new FileStream(@"H:\CSHARPTEMP\WBFFS.txt",
                               FileMode.OpenOrCreate, FileAccess.Write);
    foreach (byte bNext in bytes)
      fs.WriteByte(bNext);
    fs.Close();
  }
}
```

The constructor we use takes three parameters: the full pathname of the file, the mode we are using to open it and the access required. The mode and access are enumerated values respectively taken from two further classes in the `System.IO` namespace: `FileMode` and `FileAccess`. The possible values for mode is `Append`, `Create`, `CreateNew`, `Open`, `OpenOrCreate` and `Truncate`. For the access, they are `Read`, `ReadWrite` and `Write`.

5.4 Networking

High level access is performed using a set of types that implement a generic request/response architecture that is extensible to support new protocols. The implementation of this architecture in the BCL also includes HTTP-specific extensions to make interacting with web servers easy.

Should the application require lower-level access to the network, types exist to support TCP and UDP. Finally, in situations where direct transport-level access is required, there are types that provide raw socket access.

5.4.1 HTTP

The HTTP protocol accounts for a large share of all traffic on the Internet; and the .NET frameworks provide robust support for the HTTP protocol with the **HttpWebRequest** and **HttpWebResponse** classes. These classes are the **WebRequest** and **WebResponse** derived classes returned whenever a URI beginning with "http" or "https" is presented to the **Create** method on the **WebRequestFactory**. In most cases, the **WebRequest** and **WebResponse** classes will provide all that is necessary to make the request, but when access to HTTP-specific features is required, the request or response can be typecast to **HttpWebRequest** or **HttpWebResponse**.

The **HttpWebRequest** and **HttpWebResponse** classes encapsulate a standard HTTP request and response transaction, and provide access to common HTTP headers through properties. These classes also support most of the HTTP 1.1 protocol features, including pipelining, chunking, authentication, pre-authentication, encryption, proxy support, server certificate validation, connection management, and HTTP extensions. Custom headers and headers not provided through properties can be accessed by storing them in the **Headers** property.

The following sample shows how to access HTTP specific properties, in this case turning off the HTTP Keep-alive behavior and getting the protocol version number from the Web server:

```
HttpWebRequest HttpWReq =
(HttpWebRequest)WebRequestFactory.Create("http://www.pune-csharp.com");
//   Turn off connection keep-alives
HttpWReq.KeepAlive = false;

HttpWebResponse HttpWResp = (HttpWebResponse)HttpWReq.GetResponse();

// Look at the HTTP protocol version number returned by the server
String ver = HttpWResp.Version.ToString();
```

HttpWebRequest is the default class used by **WebRequestFactory** and does not need to be registered before passing an HTTP Uniform Resource Identifier (URI) to the **WebRequestFactory.Create** method.

Your application can automatically follow HTTP redirects by setting the **AllowAutoRedirect** property **true**. When the request is redirected, the **ResponseURI** property of the **HttpWebResponse** will contain the actual Web resource that responded to the request. When **AllowAutoRedirect** is **false**, your application must be prepared to handle redirects as an HTTP protocol error.

Applications receive HTTP protocol errors by catching a **WebException** with the **Status** set to **WebStatus.ProtocolError**. The **Response** property contains the **WebResponse** sent by the server and it can be examined to find the actual HTTP error encountered.

5.4.2 Generic Request/Response Architecture

This is based on Uniform Resource Indicator (URI) and stream I/O, follows the factory design pattern, and makes good use of abstract types and interfaces.

A uniform resource locator is a compact representation of a resource available to your application via the Internet. The **URI** class defines the properties and methods for handling URIs, including parsing, comparing, and combining.

The **URI** class stores only absolute URIs, relative URIs must be expanded with respect to a base URI so that they are absolute.

The **URI** is stored as a canonical URI in "escaped" format, with all characters with an ASCII value greater than 127 replaced with a hexidecimal representation. To put the URI in canonical form, the URI constructor:

- Converts the URI scheme to lower case.
- Converts the host name to lower case.
- Removes default and empty port numbers.
- Simplyfies the URI by removing superfluous segments such as "/" and "/test" segments.

The **URI** class can be transformed from an escaped URI reference to a readable **URI** reference with the **ToString** method.

The following example creates an instance of the **URI** class (derived from **Object**), which can further be used to create a **WebRequest**:

```
URI siteURI = new URI("http://www.pune-csharp.com/");
```

The **WebRequestFactory** class (derived from **Object**) is a static class that returns an instance of an object derived from **WebRequest**. The specific class of object returned is based on the URI scheme passed to the **Create** method.

The following example creates a **WebRequest** instance for an HTTP request. Since the URI indicates an HTTP request, the actual instance returned is an instance of **HttpWebRequest**

```
WebRequest wr = WebRequestFactory.Create(siteURI);
```

The **WebRequest** is an abstract class derived from **Object**. Applications should never create **WebRequest** objects directly. The **GetResponse** method in this class, when overridden in a derived class, returns the response to an Internet request.

The **WebResponse** class is an abstract base class (derived from **Object**) from which protocol-specific classes are derived. The **WebResponse** class can be used to access any resource on the network that is addressable with a URI. Client applications should never create **WebResponse** objects directly. The **GetResponseStream** method in this class, when overridden in a derived class, returns the **Stream** object used for reading data from the resource referenced in the **WebRequest** object.

The **Stream** class provides a way to write and read bytes to and from a backing store. This class is abstract. The **StreamReader** class implements a **TextReader** (represents a reader that can read a sequential stream of characters. This class is abstract) that reads characters from a byte stream in a particular encoding. The character encoding is set by **Encoding** class and the default buffer size is used. The **ReadToEnd** method of the **StreamReader** class reads the stream from the current position to the end of the stream. The **Close** method closes the **StreamReader** and releases any system resources associated with the reader.

The example Snarf.cs below uses the WebRequest and WebResponse classes to retrieve the contents of a URI and display them to the console.

```
// Snarf.cs
// Compile with /r:System.Net.dll
// Run Snarf.exe <http-url> to retrieve a web page
using System;
using System.IO;
using System.Net;
using System.Text;
class Snarf {
  public static void Main(string[] args) {
    URI siteURI = new URI(args[0]);
    WebRequest req = WebRequestFactory.Create(siteURI);
    WebResponse res = req.GetResponse();
    Stream s = res.GetResponseStream();
    StreamReader sr = new StreamReader(s, Encoding.ASCII);
    string doc = sr.ReadToEnd();
    Console.WriteLine(doc);
    sr.Close();
  }
}
```