India Community Initiative

.NET Tutorial for Beginners

Special thanks to the following who have put in sincere efforts to write and bring this tutorial together.

Akila Manian (MVP) | Ajay Varghese (MVP) | Amit Kukreja | Anand M (MVP) | Aravind Corera (MVP) | Arvind Rangan | Balachandran | Bipin Joshi (MVP) | C S Rajagopalan | G Gokulraj | G Arun Prakash | Gurneet Singh (MVP) | Kunal Cheda (MVP) | Manish Mehta (MVP) | Narayana Rao Surapaneni (MVP) | Pradeep | Saurabh Nandu (MVP) | Shankar N.S. | Swati Panhale | Reshmi Nair

Content

1. Getting Ready

Section Owner: Ajay Varghese (MVP)
Content Contributors: Bipin Joshi (MVP)

Welcome friends to the exciting journey of Microsoft .NET. If you are looking for information about what .NET is all about, what it can do for you or how it can help you and your customers, you have come to the right place. This section is intended to tell you about these and many more things. After covering this section you will be ready to delve into details of .NET.

The section is divided into following sub-sections:
1) Tracing the .NET History
2) Flavors of .NET
3) Features of .NET
4) Installing .NET Framework SDK

The first sub-section will introduce you with how .NET evolved and the path of .NET since its Beta releases.

The second sub-section will introduce you with various flavors of...NET and their respective SDKs. It also gives overview of Visual Studio.NET – an excellent IDE for developing .NET applications.

It is necessary to understand the features of .NET that make it robust, programmer friendly, powerful and flexible. The third sub-section is intended just for that. It gives overview of technical features that make .NET shine over traditional programming environments.

The final sub-section tells you how to install .NET framework SDK, what are the system requirements and related topics.

1.1 Tracing the .NET History

Sometime in the July 2000, Microsoft announced a whole new software development framework for Windows called .NET in the Professional Developer Conference (PDC). Microsoft also released PDC version of the software for the developers to test. After initial testing and feedback Beta 1 of .NET was announced. Beta 1 of the .NET itself got lot of attention from the developer community. When Microsoft announced Beta 2, it incorporated many changes suggested by the community and internals into the software. The overall 'Beta' phase lasted for more than 1 ½ years. Finally, in March 2002 Microsoft released final version of the .NET framework.

One thing to be noted here is the change in approach of Microsoft while releasing this new platform. Unlike other software where generally only a handful people are involved in beta testing, .NET was thrown open to community for testing in it's every pre-release version. This is one of the reasons why it created so many waves of excitement within the community and industry as well.

Microsoft has put in great efforts in this new platform. In fact Microsoft says that its future depends on success of .NET. The development of .NET is such an important event that Microsoft considers it equivalent to transition from DOS to Windows. All the future development – including new and version upgrades of existing products – will revolve around .NET. So, if you want to be at the forefront of Microsoft Technologies, you should be knowing .NET!

Now, that we know about brief history of .NET let us see what .NET has to offer.

1.2 Flavors of .NET

Contrary to general belief .NET is not a single technology. Rather it is a set of technologies that work together seamlessly to solve your business problems. The following sections will give you insight into various flavors and tools of .NET and what kind of applications you can develop.

- **What type of applications can I develop?**

  When you hear the name .NET, it gives a feeling that it is something to do only with internet or networked applications. Even though it is true that .NET provides solid foundation for developing such applications it is possible to create many other types of applications. Following list will give you an idea about various types of application that we can develop on .NET.

  1. ASP.NET Web applications: These include dynamic and data driven browser based applications.
  2. Windows Form based applications: These refer to traditional rich client applications.
  3. Console applications: These refer to traditional DOS kind of applications like batch scripts.
  4. Component Libraries: This refers to components that typically encapsulate some business logic.
  5. Windows Custom Controls: As with traditional ActiveX controls, you can develop your own windows controls.
  6. Web Custom Controls: The concept of custom controls can be extended to web applications allowing code reuse and modularization.
  7. Web services: They are "web callable" functionality available via industry standards like HTTP, XML and SOAP.

8. Windows Services: They refer to applications that run as services in the background. They can be configured to start automatically when the system boots up.

As you can clearly see, .NET is not just for creating web application but for almost all kinds of applications that you find under Windows.

- **.NET Framework SDK**

  You can develop such varied types of applications. That's fine. But how? As with most of the programming languages, .NET has a complete Software Development Kit (SDK) - more commonly referred to as **.NET Framework SDK -** that provides classes, interfaces and language compilers necessary to program for .NET. Additionally it contains excellent documentation and Quick Start tutorials that help you learn .NET technologies with ease. Good news is that - .NET Framework SDK is available FREE of cost. You can download it from the MSDN web site. This means that if you have machine with .NET Framework installed and a text editor such as Notepad then you can start developing for .NET right now!

  You can download entire .NET Framework SDK (approx 131 Mb) from MSDN web site at http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/msdn-files/027/000/976/msdncompositedoc.xml

- **Development Tools**

  If you are developing applications that require speedy delivery to your customers and features like integration with some version control software then simple Notepad may not serve your purpose. In such cases you require some Integrated Development Environment (IDE) that allows for Rapid Action Development (RAD). The new Visual Studio.NET is such an IDE. VS.NET is a powerful and flexible IDE that makes developing .NET applications a breeze. Some of the features of VS.NET that make you more productive are:

  - Drag and Drop design
  - IntelliSense features
  - Syntax highlighting and auto-syntax checking
  - Excellent debugging tools
  - Integration with version control software such as Visual Source Safe (VSS)
  - Easy project management

  Note that when you install Visual Studio.NET, .NET Framework is automatically installed on the machine.

- **Visual Studio.NET Editions**

  Visual Studio.NET comes in different editions. You can select edition appropriate for the kind of development you are doing. Following editions of VS.NET are available:

  - Professional
  - Enterprise Developer
  - Enterprise Architect

    Visual Studio .NET Professional edition offers a development tool for creating various types of applications mentioned previously. Developers can use Professional edition to build Internet and Develop applications quickly and create solutions that span any device and integrate with any platform.

    Visual Studio .NET Enterprise Developer (VSED) edition contains all the features of Professional edition plus has additional capabilities for enterprise development. The features include things such as a collaborative team development, Third party tool integration for building XML Web services and built-in project templates with architectural guidelines and spanning comprehensive project life-cycle.

    Visual Studio .NET Enterprise Architect (VSEA) edition contains all the features of Visual Studio .NET Enterprise Developer edition and additionally includes capabilities for designing, specifying, and communicating application architecture and functionality. The additional features include Visual designer for XML Web services, Unified Modeling Language (UML) support and enterprise templates for development guidelines and policies.

    A complete comparison of these editions can be found at http://msdn.microsoft.com/vstudio/howtobuy/choosing.asp

  In addition to these editions, special language specific editions are available. They are:

  - Visual Basic.NET Standard Edition
  - Visual C# Standard Edition
  - Visual C++ .NET Standard (soon to be released)

    These editions are primarily for hobbyist, student, or beginner who wants to try their hands on basic language features.

A complete comparison of these standard editions with professional edition of VS.NET can be found at:

http://msdn.microsoft.com/vcsharp/howtobuy/choosing.asp
http://msdn.microsoft.com/vbasic/howtobuy/choosing.asp

- **.NET Redistributable**

  In order to run application developed using .NET Framework the machine must have certain 'runtime' files installed. They are collectively called as .NET redistributable. This is analogous to traditional Visual Basic applications that required Visual Basic runtime installed on target computers. .NET redistributable provides one redistributable installer that contains the common language runtime (more on that later) and Microsoft .NET Framework components that are necessary to run .NET Framework applications. The redistributable is available as a stand-alone executable and can be installed manually or as a part of your application setup.

  You can download .NET redistributable at http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/msdn-files/027/001/829/msdncompositedoc.xml

  More technical information about .NET redistributable can be found at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetdep/html/dotnetfxref.asp

  Note that if you have installed .NET Framework SDK, there is no need of installing redistributable separately. Also, note that there is difference between .NET Framework SDK and .NET redistributable in terms of purpose and tools and documentation supplied. .NET Framework SDK is intended to 'develop' applications where as .NET redistributable is intended to 'run' .NET applications.

- **.NET and mobile development**

  Now days the use of mobile and wireless devices is ever increasing. PDAs, mobile phones, Smartphones, handheld PCs and HTML pagers are becoming common. As compared to full blown desktop computers, Mobile devices are generally resource-constrained. There are limitations on what they can display and in which form. For example you can easily display graphical menus in desktop applications but the same may not be possible for cell phones.

  Today there are many vendors making CPUs and development tools for mobile devices. However, their standards are much varying. For example devices running Windows CE will have different tools and standards of development than Palm OS. Also, programming model for such devices is an issue of debate. For example, Wireless Application Protocol (WAP) was considered a 'standard' for mobile devices but it introduced disadvantages of its own such as requirement of continuous connectivity, lack in rich user interface and failure to utilize client – side resources effectively.

Mobile devices can be broadly divided into two categories:

1) Mobile Devices that have certain client-side resources like PDAs, Smartphones and Handheld PCs. They can run stand-alone application with rich user interface.
2) Mobile Devices that lack even these client-side resources such as mobile phones. They can not run stand alone applications having rich and more interactive user interface.

In order to encompass all possible devices from above categories Microsoft has developed two distinct technologies namely:

- Microsoft .NET Compact Framework (.NET CF)
- Microsoft Mobile Internet Toolkit (MMIT)

  - **Microsoft .NET Compact Framework**

    .NET compact framework is a sub set of entire .NET framework and is targeted at mobile devices having some client side resources. It provides support for managed code and XML Web services. Currently, .NET Compact Framework is in Beta 1 and is available on devices running the Windows CE or Windows CE .NET operating systems. However, Microsoft has promised support for other platforms in the future. As of now the framework supports Visual Basic.NET and C# as development languages out of the box. Support for other languages is planned in near future.

    Microsoft is creating a set of extensions for Visual Studio .NET called Smart Device Extensions that will allow Visual Studio .NET developers to program for .NET Compact Framework. This means that developers familiar with Visual Studio.NET can start developing for mobile devices almost instantly.

    More information about .NET Compact Framework can be obtained at http://msdn.microsoft.com/vstudio/device/compact.asp

  - **Microsoft Mobile Internet Toolkit**

    Microsoft Mobile Internet Toolkit (MMIT) is designed to develop server side applications for mobile devices such as cell phones, PDAs, and pagers. It is different than .NET compact Framework in that it is a server side technology. It is ideal for devices that can not run stand alone applications.

    MMIT mainly uses ASP.NET as a technology for delivering markup to a wide variety of mobile devices. As we know that each mobile device has its own set of underlying standards and markup. MMIT shields these details from the developer and allows ‘uniform code’ for any target device. Based on the capabilities of target device the output is rendered.

More information about MMIT can be obtained from http://msdn.microsoft.com/vstudio/device/mitdefault.asp

1.3 Features of .NET

Now that we know some basics of .NET, let us see what makes .NET a wonderful platform for developing modern applications.

- **Rich Functionality out of the box**

  .NET framework provides a rich set of functionality out of the box. It contains hundreds of classes that provide variety of functionality ready to use in your applications. This means that as a developer you need not go into low level details of many operations such as file IO, network communication and so on.

- **Easy development of web applications**

  ASP.NET is a technology available on .NET platform for developing dynamic and data driven web applications. ASP.NET provides an event driven programming model (similar to Visual Basic 6 that simplify development of web pages (now called as web forms) with complex user interface. ASP.NET server controls provide advanced user interface elements (like calendar and grids) that save lot of coding from programmer’s side.

- **OOPs Support**

  The advantages of Object Oriented programming are well known. .NET provides a fully object oriented environment. The philosophy of .NET is – “Object is mother of all.” Languages like Visual Basic.NET now support many of the OO features that were lacking traditionally. Even primitive types like integer and characters can be treated as objects – something not available even in OO languages like C++.

- **Multi-Language Support**

  Generally enterprises have varying skill sets. For example, a company might have people with skills in Visual Basic, C++, and Java etc. It is an experience that whenever a new language or environment is invented existing skills are outdated. This naturally increases cost of training and learning curve. .NET provides something attractive in this area. It supports multiple languages. This means that if you have skills in C++, you need not throw them but just mould them to suit .NET environment. Currently four languages are available right out of the box namely – Visual Basic.NET, C# (pronounced as C-sharp), Jscript.NET and

Managed C++ (a dialect of Visual C++). There are many vendors that are working on developing language compilers for other languages (20+ language compilers are already available). The beauty of multi language support lies in the fact that even though the syntax of each language is different, the basic capabilities of each language remain at par with one another.

- **Multi-Device Support**

  Modern lift style is increasingly embracing mobile and wireless devices such as PDAs, mobiles and handheld PCs. . . .NET provides promising platform for programming such devices. .NET Compact Framework and Mobile Internet Toolkit are step ahead in this direction.

- **Automatic memory management**

  While developing applications developers had to develop an eye on system resources like memory. Memory leaks were major reason in failure of applications. .NET takes this worry away from developer by handling memory on its own. The garbage collector takes care of freeing unused objects at appropriate intervals.

- **Compatibility with COM and COM+**

  Before the introduction of .NET, COM was the de-facto standard for componentized software development. Companies have invested lot of money and efforts in developing COM components and controls. The good news is – you can still use COM components and ActiveX controls under .NET. This allows you to use your existing investment in .NET applications. .NET still relies on COM+ for features like transaction management and object pooling. In fact it provides enhanced declarative support for configuring COM+ application right from your source code. Your COM+ knowledge still remains as a valuable asset.

- **No more DLL Hell**

  If you have worked with COM components, you probably are aware of "DLL hell". DLL conflicts are a common fact in COM world. The main reason behind this was the philosophy of COM – "one version of component across machine". Also, COM components require registration in the system registry. .NET ends this DLL hell by allowing applications to use their own copy of dependent DLLs. Also, .NET components do not require any kind of registration in system registry.

- **Strong XML support**

  Now days it is hard to find a programmer who is unaware of XML. XML has gained such a strong industry support that almost all the vendors have released some kind of upgrades or patches to their existing software to make it "XML compatible". Currently, .NET is the only platform that has built with XML right into the core framework. .NET tries to harness power of XML in every possible way. In addition to providing support for manipulating and transforming XML documents, .NET provides XML web services that are based on standards like HTTP, XML and SOAP.

- **Ease of deployment and configuration**

  Deploying windows applications especially that used COM components were always been a tedious task. Since .NET does not require any registration as such, much of the deployment is simplified. This makes XCOPY deployment viable. Configuration is another area where .NET – especially ASP.NET – shines over traditional languages. The configuration is done via special files having special XML vocabulary. Since, most of the configuration is done via configuration files, there is no need to sit in front of actual machine and configure the application manually. This is more important for web applications; simply FTPing new configuration file makes necessary changes.

- **Security**

  Windows platform was always criticized for poor security mechanisms. Microsoft has taken great efforts to make .NET platform safe and secure for enterprise applications. Features such as type safety, code access security and role based authentication make overall application more robust and secure.

1.4 Installing the .NET Framework SDK

Now that you have fare idea of what .NET I and what it can do for you, it is time to install .NET framework SDK on your machine. Following sections will tell you everything you need to know for installing .NET framework.

- **Hardware Requirements**

  In order to install .NET framework SDK following hardware is required:

  - Computer/Processor : Intel Pentium class, 133 megahertz (MHz) or higher
  - Minimum RAM Requirements : 128 megabytes (MB) (256 MB or higher recommended)
  - Hard Disk :
    - Hard disk space required to install: 600 MB
    - Hard disk space required: 370 MB
  - Display : Video: 800x600, 256 colors
  - Input Device : Microsoft mouse or compatible pointing device

- **Software Requirements**
  - Microsoft Internet Explorer 5.01 or later is required
  - Microsoft Data Access Components 2.6 is also required (Microsoft Data Access Components 2.7 is recommended)
  - Operating System :
    - Microsoft Windows® 2000, with the latest Windows service pack and critical updates available from the Microsoft Security Web page

- Microsoft Windows XP – (Microsoft Windows XP Professional if you want to run ASP.NET)
- Microsoft Windows NT® 4.0

Note: If you want to simply run .NET applications then you can also run them on Microsoft Windows XP Home edition, Windows Millennium Edition (Windows ME) and Windows 98.

Here are some URLs that you will find handy in making your system up-to-date for above software requirements.

Internet Explorer 6 can be downloaded from http://www.microsoft.com/windows/ie/downloads/ie6/default.asp

Microsoft Data Access Components 2.7 can be downloaded from http://www.microsoft.com/data/download_270RTM.htm

Various Windows service packs and patches can be obtained from http://www.microsoft.com/downloads/search.asp

- **Where to get .NET Framework SDK**

  As mentioned earlier .NET framework SDK is freely downloadable from MSDN site. Visit http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/msdn-files/027/000/976/msdncompositedoc.xml and download it now.

  The total download size is 137,363,456 bytes (approximately 131 Mb). For your convenience Microsoft has provided multi-part version of the entire download. If you are unable to download the SDK from MSDN web site, check out popular PC magazines around. Many of them contain .NET Framework SDK on their companion CD.

- **Starting the installation**

  Note: If you already have a previous version of .NET installed on the machine then it must first be uninstalled. Refer ReadMe files that ship with .NET framework SDK. These files contain valuable information related to installation, system requirements and trouble shooting.

  In order to start the installation, you need to run the setup program that is available with the download mentioned above. A wizard will guide you with necessary process. It will also allow you to select various components of the framework.

After the installation is complete it is a good idea to apply .NET framework Service pack 1. The service pack fixes some of the bugs. It can be downloaded from:
http://msdn.microsoft.com/netframework/downloads/sp1/default.asp

- **Installing Samples and Quick Start Tutorials**

  .NET framework comes with an excellent set of tutorials that help you learn various technologies such as ASP.NET and windows forms. In order to configure the tutorials follow Start menu -> Program -> Microsoft .NET Framework SDK -> Samples and Quick Start Tutorials. This will open up a HTML document that will guide you through the process of configuring the samples and tutorials.

- **Installing MSDE**

  .NET framework samples and quick start tutorials require a Microsoft SQL Server Desktop Engine (MSDE). MSDE is scaled down version of SQL Server. The samples use databases from the MSDE. In order to work with the samples make sure you have started an instance of MSDE. You can use this MSDE for creating your own databases for testing applications.

Coming Next…

By the time you must have got idea about what .NET is and what it can do for you. You probably will have installed .NET on your machine waiting eagerly to try hands on it. However, before you go into the code level details, it is essential that you firmly understand certain fundamentals. In the next section we will demystify some intrinsic concepts and features of .NET framework.

2. Introduction to the .NET Initiative and the .NET Platform

Section Owner: Saurabh Nandu (MVP)
Content Contributors: Balachandran, Pradeep

The Microsoft .NET initiative is a very wide initiative and it spans multiple Microsoft Products ranging from the Windows OS to the Developer Tools to the Enterprise Servers. The definition of .NET differs from context to context, and it becomes very difficult for you to interpret the .NET strategy. This section aims at demystifying the various terminologies behind .NET from a developer's perspective. It will also highlight the need for using this new .NET Platform in your applications and how .NET improves over its previous technologies.

2.1 Understanding the Existing Development Scenario

Windows DNA is a concept for building distributed applications using the Microsoft Windows operating system and related software products.
First we will understand about the 2- tier, 3- tier and then move on to N- tier Windows DNA.

Why to divide an application into logical layers?

Factoring an application into logical parts is useful. Breaking a large piece of software into smaller pieces can make it easier to build, easier to reuse and easier to modify. It can also be helpful in accommodating different technologies or different business organizations.

2-Tier: Client Server

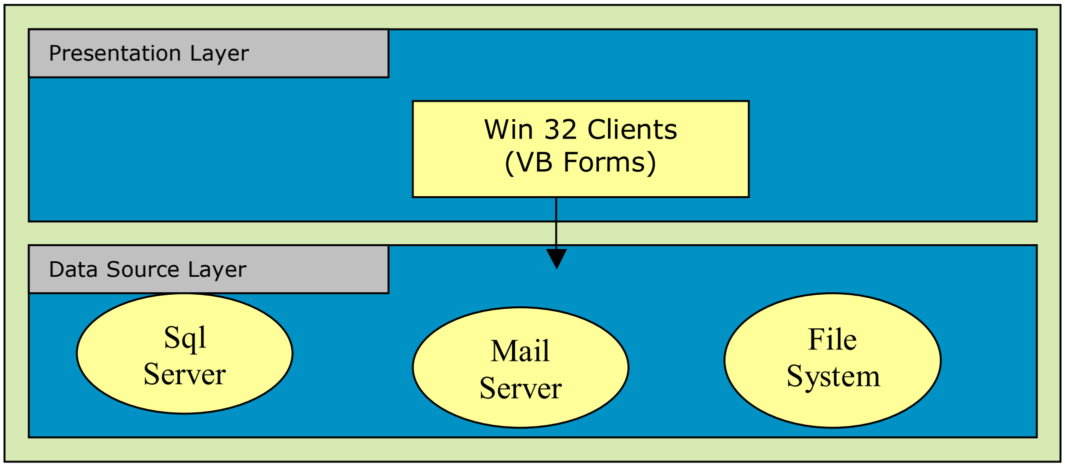


Fig Showing 2 – Tier Client Server Model

Through the appearance of Local-Area-Networks, PCs came out of their isolation, and were soon not only being connected mutually but also to servers. Client/Server-computing was born. *A two-tiered application is an application whose functionality can only be segmented into two logical tiers, presentation services and data services.* The presentation services of a two-tiered application are responsible for gathering information from the user, interacting with the data services to perform the application's business operations, and presenting the results of those operations to the user. The *Presentation services* are also called the presentation layer because it presents information to the user. Things you might find in a *presentation layer* include a Web browser, a terminal, a custom-designed GUI, or even a character-based user interface. Client-Server architecture was a major buzzword in the early 90's, taking initially dumb terminal applications and giving them a fancy windows-like front end, using PCs with terminal emulators which presented pretty GUIs (Graphical user interface) or later Visual Basic etc front-ends. A web browser talking to a web server is an example of a client talking to a server. Here there is presentation logic (presentation tier) happening at the client, and data/file access (data access tier) and logic happening at the server. One reason why the 2-tier model is so widespread is because of the quality of the tools and middleware that have been most commonly used since the 90’s: Remote-SQL, ODBC, relatively inexpensive and well-integrated PC-tools (like Visual Basic, Power-Builder, MS Access, 4-GL-Tools by the DBMS manufactures). In comparison the server side uses relatively expensive tools. In addition the PC-based tools show good Rapid-Application-Development (RAD) qualities i.e. simpler applications can be produced in a comparatively short time. The 2-tier model is the logical consequence of the RAD-tools’ popularity.

3 – Tier: Client Server

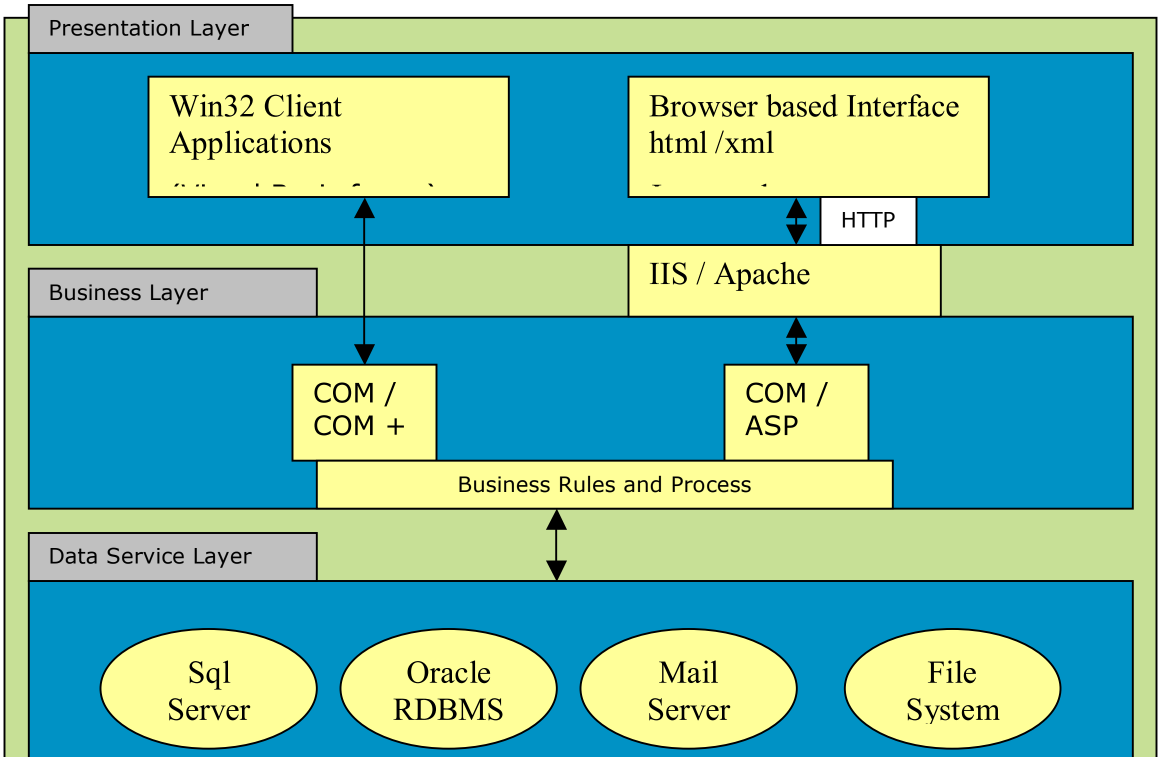


Fig Showing 3 – Tier or N- Tier Client Server Model

In a three-tiered application, the presentation services are responsible for gathering information from the user, sending the user information to the business services for processing, receiving the results of the business services processing, and presenting those results to the user. The most popular architecture on the web currently, mostly taking the form of web browser processing client side presentation in the form of HTML/DHTML, etc, the web server using some scripting language (ASP) and the database server (SQL Server for example) serving up the data.

The basic functionalities of 3 – Tier or N-Tier follows are

The presentation services tier is responsible for:

- Gathering information from the user
- Sending the user information to the business services for processing
- Receiving the results of the business services processing
- Presenting those results to the user

The business services tier is responsible for:

- Receiving input from the presentation tier.
- Interacting with the data services to perform the business operations.
- Sending the processed results to the presentation tier.

The data services tier is responsible for the:

- Storage of data.
- Retrieval of data.
- Maintenance of data.
- Integrity of data.

In Windows DNA applications commonly implement their business logic using one or more of three implementation options.

- Asp Pages
- COM components
- Stored procedures running in the DBMS

Writing much business logic in ASP pages is a bad idea. Since simple languages are used, such as Microsoft Visual Basic Script, and the code is interpreted each time it is executed, which hurts the performance. Code in ASP pages is also hard to maintain, largely because business logic is commonly intermixed with presentation code that creates the user interface.

One recommended approach for writing middle-tier business logic is to implement that logic as COM objects. This approach is a bit more complex than writing a pure ASP application. Wrapping business logic in COM objects also cleanly separates this code from the presentation code contained in ASP pages, making the application easier to maintain.

The Third option for writing business logic is to create some of that code as stored procedures running in the database management system (DBMS). Although a primary reason for using stored procedures is to isolate the details of database schema from business logic to simplify code management and security, having code in such a close proximity to data can also help optimize performance.

2.2 Challenges faced by developers

In Windows DNA, there are two major choices of user interfaces - Win32 clients and browser based clients. During the Internet revolution of the late 90s we saw the emergence of the browser and the Web Server. With the introduction of Internet, information started being available but with limited functionality. With the development of the Windows Distributed Internet Architecture, we started to see Web sites that allowed simple transactions to occur. Clients on browsers could access Web sites that had COM components available to them that allowed them to retrieve information from the database. So now we gained the capability to simulate the environment of the Win32 platform. The client software – the browser – can access information on a server. But as with the Win32 environment, we are limited in the way in which the information is presented to us. Customization is neither widespread nor broadly developed.

Let us look into limitations of these technologies.

Limitations in Win32 Clients

In a client-server environment visual tool such as Visual Basic, are often used to create a rich user interface. The drawbacks is that such client software is difficult to deploy and maintain, requiring and install on every client and a change to every client when an upgrade is needed.

DLL conflicts on the client are frequent because of variations in the version of the operating system and other software installed on the client.

Visual Basic is the most common language used to write middle-tier components. This requires high level of expertise in COM. Since these middle-tire components are implemented using Microsoft Transaction Server on Windows NT or COM+ services on Windows 2000. These components use stateless designs, which can look very different from the stateful designs often used in client-based components.

COM components, in the middle tier must work together, Versioning all the components properly so that they understand each other's interfaces can be a challenge. This requires a highly sophisticated skill level and a well - controlled deployment process.

COM works well on Microsoft platforms. But it suffers from lack of interoperability with other platforms. One of the most important ways functionality can be reused is for a software component to inherit another component, But COM does not support inheritance.

Visual Basic is the most popular language for developing applications with the DNA model, this is used in two major roles - forms based VB Clients and COM components. This VB6 language has its own limitations it doesn't have the capability of multithreading, lack of OOPS concepts, Poor error handling ability and poor integration with other languages. Hence it makes it unsuitable for development of object-based frameworks.

Today's applications need to use the Win32 API for a variety of purposes like monitor widows messages, manipulate controls, reading and writing to INI files and socket programming etc. But these widows API are hard to program for variety of reasons, like it is not object oriented and complex calls to the functions with long lists of arguments, since Win32 API is written in C++ language, getting calling conventions right on data types is messy.

Limitations in DNA-Based Internet Development or Browser based clients

With DNA - based software development, creating software that is accessed by a user locally is done very differently from development for the Internet. The Visual Basic forms for client-server user interfaces versus the use of Active Server Pages for Internet user interfaces. Even though both situations involve designing and implementing GUI based user interfaces the tools and programming techniques used are quite different.

ASP lacks in state management between post backs. Every time a page is rendered, the programmer must make sure that all the visual controls like text boxes, dropdowns have their information loaded. It is the programmer's responsibility to manage the state in the user interface and to transfer state information between pages. This causes developers to have to write a lot of code for the internet user interfaces that is not relevant to business problem being solved.

If the Internet application is going to run on a group of Web Servers, then considerable additional work is necessary to design a state management system that is independent of particular server.

Browser based clients are somewhat more difficult to create, and offer a more limited user interface with fewer controls and less control over layout of the screen and handling of screen events. It is possible to create rich user interfaces using DHTML, but it requires lot of coding and also browser compatibility issues rises, for which a separate coding or two version of the same page have to be maintained, keeping in mind, the browser we are targeting.

The Internet has caused server-based applications to become much more popular than ever before and has made the connectionless request/response programming model common. But communicating between servers—especially among those running on different platforms—is difficult, and because most substantial Internet applications are Database-Centric, the ability to access a wide variety of data sources easily is more important than ever.

As we move on to handheld devices or wireless devices, kiosks or other type of systems, many of which run a different processors and do not use standard operating system. So sharing the data between these devices and communication varies which is not uniform, becomes difficult.

2.3 NET Philosophy / Where does .NET fit in?

The driving force behind Microsoft® .NET is a shift in focus from individual Web sites or devices to new constellations of computers, devices, and services that work together to deliver broader, richer solutions.

The platform, technology that people use is changing. Since 1992, the client/server environment has been in place, with people running the applications they need on the Win32 platform, for example. Information is supplied by the databases on the servers, and programs that are installed on the client machine determine how that information is presented and processed.

One of the things people are looking for is a one-sentence definition of ".NET". What is it? Why should I care? .NET is Microsoft's strategy for software that empowers people any time, any place, and on any device.

Many of the goals Microsoft had in mind when designing .NET reflect the limitations we previously discussed for development with previous tools and technologies.

Microsoft.NET solutions

- ***Single Programming Model*** A related goal is to have development for the internet environment look very much like development for other types of software. Likewise, developing user interfaces in Windows Forms is very similar to developing them in Web Forms. There are commonly used controls, such as Labels and Text Boxes, in both, with similar sets of properties and method. The amount of commonality makes it easy to transition between the two types of development, and easier for traditional VB developers to start using Web Forms.

- ***Distributed Systems*** The Vision of Microsoft.NET is globally distributed systems, using XML as the universal glue to allow functions running on different computers across an organization or across the world to come together in a single application. In this vision, systems from servers to Wireless Palmtops, with everything in between, will share the same general platform, with versions of .NET available for all of them, and with each of them able to integrate transparently with the others.

- ***Richer User Interface*** Web Forms are a giant step towards much richer web-based user interfaces. Their built-in intelligence allows rich, browser-independent screens to be developed quickly, and to be easily integrated with compiled code. Microsoft has announced an initiative for the future called the ***Universal Canvas*** which builds upon the XML standards to transform the internet from a Read only environment into a read/write platform, enabling users to interactively create, browse, edit and analyze information. The universal canvas can bring together multiple sources of information anywhere in the world to enable seamless data access and use.(The universal canvas will log on to the Ms System of servers whenever the new device is turned on) Centrally controlled OS, Office and Visual Studio.

- ***Easy Deployment*** Executable modules in .NET are self-describing. Once the Common Language Runtime (CLR is explained in next sections) knows where a module resides, it can find out everything else it needs to know to run the module, such as the module's object interface and security requirements, from the module itself. That means a module can just be copied to a new environment and immediately executed.

- ***Support for Multiple Languages*** The CLR executes binary code called MSIL (Microsoft intermediate language), and that code looks the same regardless of the original source language. All .NET –enabled languages use the same data types and the same interfacing conventions. This makes possible for all .NET language to interoperate transparently. One language can call another easily, and languages can even inherit classes written in another language and extend them current platform has anywhere near this level of language interoperability.

- ***Extendibility*** The completely object based approach of .NET is designed to allow base functionality to be extended through inheritance (unlike COM) and the platform's functionality is appropriately partitioned to allow various parts(such as the just-in-time compilers discussed in the next section) to be replaced as new versions are needed. It is likely that, in the future, new ways of interfacing to the outside world will be added to the current trio of windows Form, Web Forms, and Web Services such as universal Canvas.

- ***Portability of compiled Applications*** .NET allows the future possibility of moving software to other hardware and operating system platforms. The ultimate goal is that compiled code produced on one implementation of .NET (such as Windows) could be moved to another implementation of .NET on a different operating system merely by copying the compiled code over and running it.

- ***Integrity with COM*** .NET integrates very will with COM-based software. Any COM component can be treated as a .NET component by other .NET components. The .NET Framework wraps COM components and exposes an interface that .NET components can work with. This is absolutely essential to the quick acceptance of .NET, because it makes .NET interoperable with a tremendous amount of older COM-based software.

Other benefits of using .NET architecture

- The Microsoft .NET platform's reliance on XML for data exchange—an open standard managed by the World Wide Web Consortium (W3C)—and modular XML Web services removes barriers to data sharing and software integration.
- The .NET platform, through the .NET Framework's common language runtime, enables XML Web services to interoperate whatever their source language. Developers can build reusable XML Web services instead of monolithic applications. By making it easy to offer your XML Web services to others.

- The ability to easily find available XML Web services means you can buy pieces of your applications rather than build everything from scratch, focusing your time and money where it makes the most sense.

- Easier to build sophisticated development tools – debuggers and profilers can target the Common Language Runtime, and thus become accessible to all .NET-enabled languages.

- Potentially better performance in system level code for memory management, garbage collection, and the like have yielded an architecture that should meet or exceed performance of typical COM-based applications today.

- Fewer bugs, as whole classes of bugs should be unknown in .NET. With the CLR handling memory management, garbage collection.

- Faster development using development tool like visual studio.net

N-tier architecture with .NET

Applications developed in the .NET Framework will still, in, many cases, use a DNA model to design the appropriate tiers. However, the tiers will be a lot easier to produce in .NET. The presentation tier will benefit from the new interface technologies and especially Web Forms for Internet development. The middle tier will require far less COM-related headaches to develop and implement. And richer, more distributed middle tier designs will be possible by using Web Services.

Let us look into how .Net fit into n – tier architecture. When you talk about a true distributed n-tier type of application, you are talking about separating the components of the different tiers on different machines as well as in separate components. Figure 1 shows a typical example of an n-tier application with multiple components on each machine.

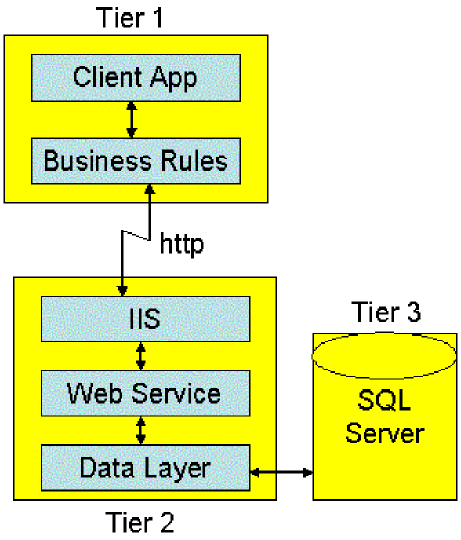
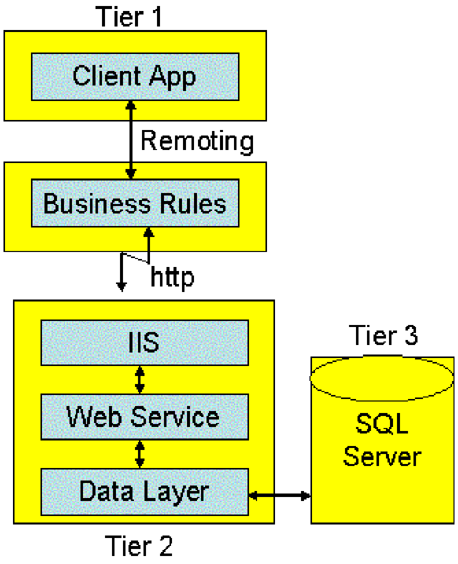


Figure 1. A distributed n-tier application has three physical tiers with one or more logical tiers on each machine

There are many different ways you could configure an n-tier application. For example, the business rules may go on a separate machine and you might use .NET Remoting to talk from the client application to the business rule tier as shown in Figure 2.



We may also have a data input validation rule component on the client to check simple rules such as required fields and formatting. These are rules that you do not want to make a trip across the network just to check. You may then also add a business rule layer on the same tier as the data layer component to check complicated business rules that compare the data from one table to another.

These are just a few different configurations that you may utilize. Of course, you could come up with something unique that fits your specific situation. Regardless of how you structure the physical implementation of the components, make sure that the logical structure of the program is broken up into components as shown in the above figures.

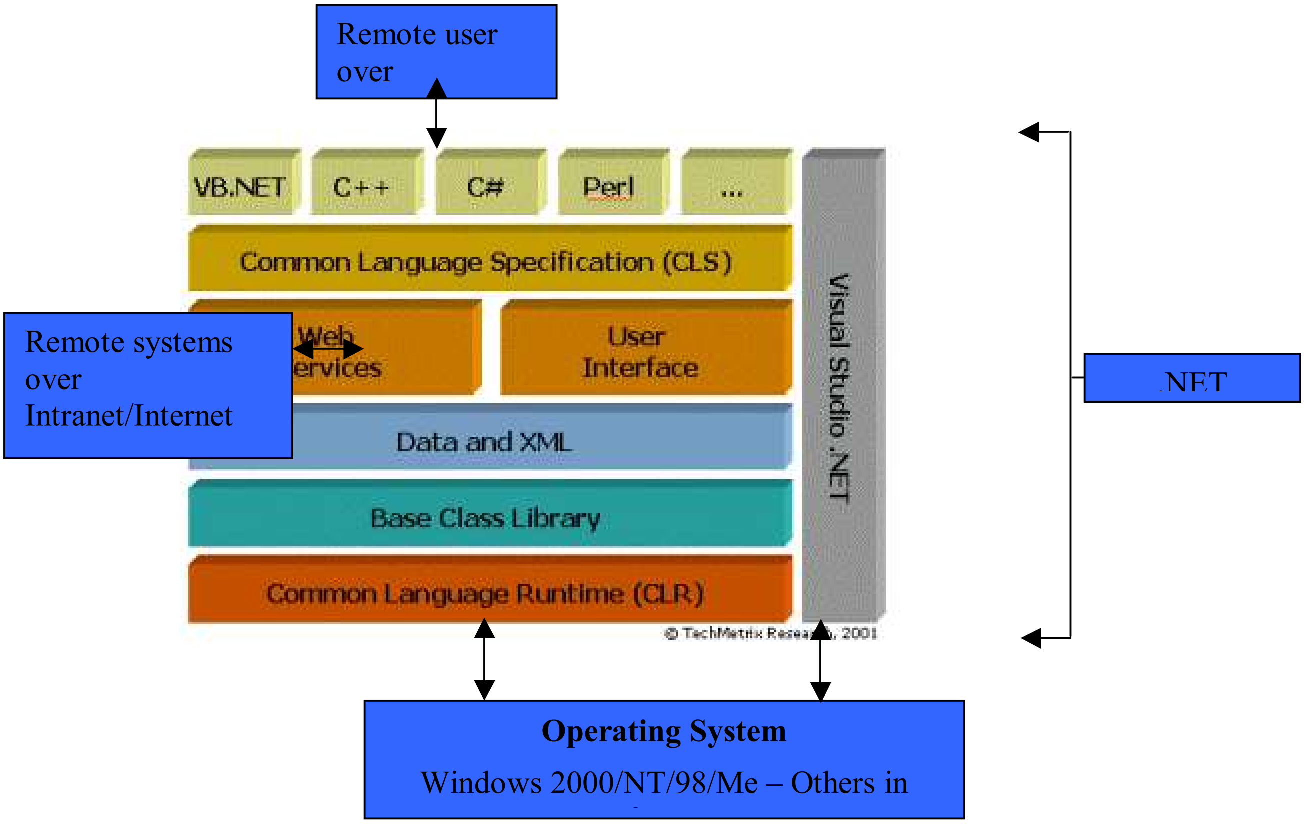
2.4 Understanding the .NET Platform and its layers

Here in this section we will be covering what the .NET Platform is made up of and we will define its layers. To start, .NET is a framework that covers all the layers of software development above the Operating System. It provides the richest level of integration among presentation technologies, component technologies, and data technologies ever seen on Microsoft, or perhaps any, platform. Secondly, the entire architecture has been created to make it easy to develop Internet applications, as it is to develop for the desktop.

Constituents of .NET Platform

The .NET consists of the following three main parts

- .NET Framework – a completely re-engineered development environment.
- .NET Products – applications from MS based on the .NET platform, including Office and Visual Studio.
- .NET Services – facilitates 3rd party developers to create services on the .NET Platform.



.NET Platform Architecture

The above diagram gives you an overview of the .NET architecture. At the bottom of the diagram is your Operating System above that sits the .NET framework that acts as an interface to it. The .NET *wraps* the operating system, insulating software developed with .NET from most operating system specifics such as file handling and memory allocation.

The Common Language Runtime (CLR)

At the base is the CLR. It is considered as the heart of the .NET framework. .NET applications are compiled to a common language known as Microsoft Intermediate Language or "IL". The CLR, then, handles the compiling the IL to machine language, at which point the program is executed.

The CLR environment is also referred to as a managed environment, in which common services, such as garbage collection and security, are automatically provided.

More information on CLR is available at http://msdn.microsoft.com/library/en-us/cpguide/html/cpconthecommonlanguageruntime.asp

The .NET Class Framework

The next layer up in the framework is called the .NET Class Framework also referred as .NET base class library. The .NET Class Framework consists of several thousand type definitions, where each type exposes some functionality. All in all, the CLR and the .NET Class Framework allow developers to build the following kinds of applications:

- Web Services. Components that can be accessed over the Internet very easily.
- Web Forms. HTML based applications (Web Sites).
- Windows Forms. Rich Windows GUI applications. Windows form applications can take advantage of controls, mouse and keyboard events and can talk directly to the underlying OS.
- Windows Console Applications. Compilers, utilities and tools are typically implemented as console applications.
- Windows Services. It is possible to build service applications controllable via the Windows Service Control Manager (SCM) using the .NET Framework.
- Component Library. .NET Framework allows you to build stand-alone components (types) that may be easily incorporated into any of the above mentioned application types.

ADO.NET: Data and XML

ADO.NET is the next generation of Microsoft ActiveX Data Object (ADO) technology. ADO.NET is heavily dependent on XML for representation of data. It also provides an improved support for the disconnected programming model.

ADO.NET's DataSet object, is the core component of the disconnected architecture of ADO.NET. The DataSet can also be populated with data from an XML source, whether it is a file or an XML stream.

For more details on ADO.NET, check out
http://msdn.microsoft.com/library/en-us/cpguide/html/cpconaccessingdatawithadonet.asp

User Interface

The next layer consists of the user and programming interface that allows .NET to interact with the outside world. The following are the types of interaction interfaces that are supported by the .NET framework:

- Web Forms
- Windows Forms
- Web Services

Now let me tell you about Windows Forms and ASP.NET. WinForms (Windows Forms) is simply the name used to describe the creation of a standard Win32 kind of GUI applications.

The Active Server Pages web development framework has undergone extensive changes in ASP.NET. The programming language of choice is now full-blown VB.NET or C# (or any supported .NET language for that matter). Other changes include:

- New support for HTML Server Controls (session state supported on the server).
- It is now possible for the server to process client-side events.

- New control families including enhanced Intrinsics, Rich controls, List controls, DataGrid control, Repeater control, Data list control, and validation controls.
- New support for developing Web Services—application logic programmatically accessible via the Internet that can be integrated into .NET applications using the Simple Object Access Protocol (SOAP).

Languages

The CLR allows objects created in one language be treated as equal citizens by code written in a completely different language. To make this possible, Microsoft has defined a *Common Language Specification (CLS)* that details for compiler vendors the minimum set of features that their compilers must support if they are to target the runtime.

Any language that conforms to the CLS can run on the CLR. In the .NET framework, Microsoft provides Visual Basic, Visual C++, Visual C#, and JScript support.

.NET Products

Microsoft Visual Studio .NET
Microsoft Visual Studio .NET represents the best development environment for the .NET platform.

Integrations is the key in the new VS.NET IDE, thus a single IDE can be used to program in a variety of managed languages from VB.NET to Visual C++ with Managed extensions. Advance features in VS.NET truly propel development in to the highest gear.

.NET Services:

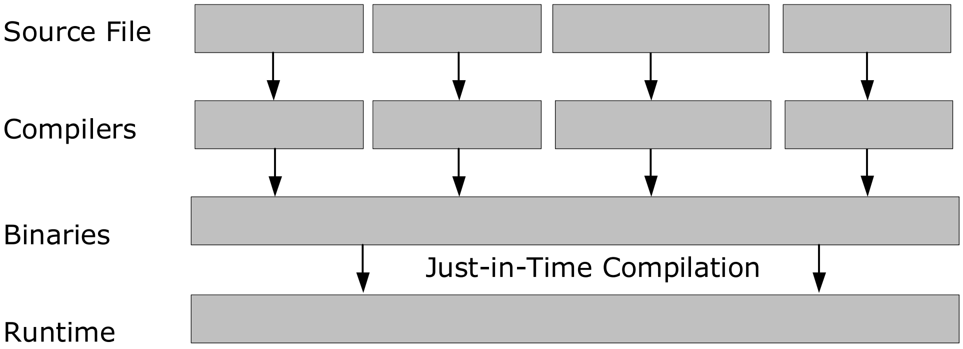
XML Web Services
XML is turning the way we build and use software inside out. The Web revolutionized how users talk to applications. XML is revolutionizing how applications talk to other applications—or more broadly, how computers talk to other computers—by providing a universal data format that lets data be easily adapted or transformed:

- XML Web services allow applications to share data.
- XML Web services are discrete units of code; each handles a limited set of tasks.
- They are based on XML, the universal language of Internet data exchange, and can be called across platforms and operating systems, regardless of programming language.
- .NET is a set of Microsoft software technologies for connecting your world of information, people, systems, and devices through the use of XML Web services.

For more details refer:
http://msdn.microsoft.com/nhp/default.asp?contentid=28000442

.NET Runtime:

Let’s now discuss about the .NET Runtime.



The .NET Framework provides a run-time environment called the Common Language Runtime, which manages the execution of code and provides services that make the development process easier. Compilers and tools expose the runtime's functionality and enable you to write code that benefits from this managed execution environment. Code developed with a language compiler that targets the runtime is called **managed code**.

To enable the runtime to provide services to managed code, language compilers must emit metadata, which the runtime uses to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set run-time context boundaries.
The runtime automatically handles objects, releasing them when they are no longer being used. Objects whose lifetimes are managed in this way are called managed data. Automatic memory management eliminates memory leaks as well as many other common programming errors.

The CLR makes it easy to design components and applications whose objects interact across languages. For example, you can define a class and then use a different language to derive a class from your original class, or call a method on the original class. You can also pass an instance of a class to a method on a class written in a different language. This cross-language integration is possible because of the common type system defined by the runtime, and they follow the runtime's rules for defining new types, as well as for creating, using, persisting, and binding to types.
Language compilers and tools expose the runtime's functionality in ways that are intended to be useful and intuitive to their developers. This means that some features of the runtime might be more noticeable in one environment than in another. How you experience the runtime depends on which language compilers or tools you use. The following benefits of the runtime might be particularly interesting to you:

- Performance improvements.
- The ability to easily use components developed in other languages.
- Extensible types provided by a class library.
- A broad set of language features.

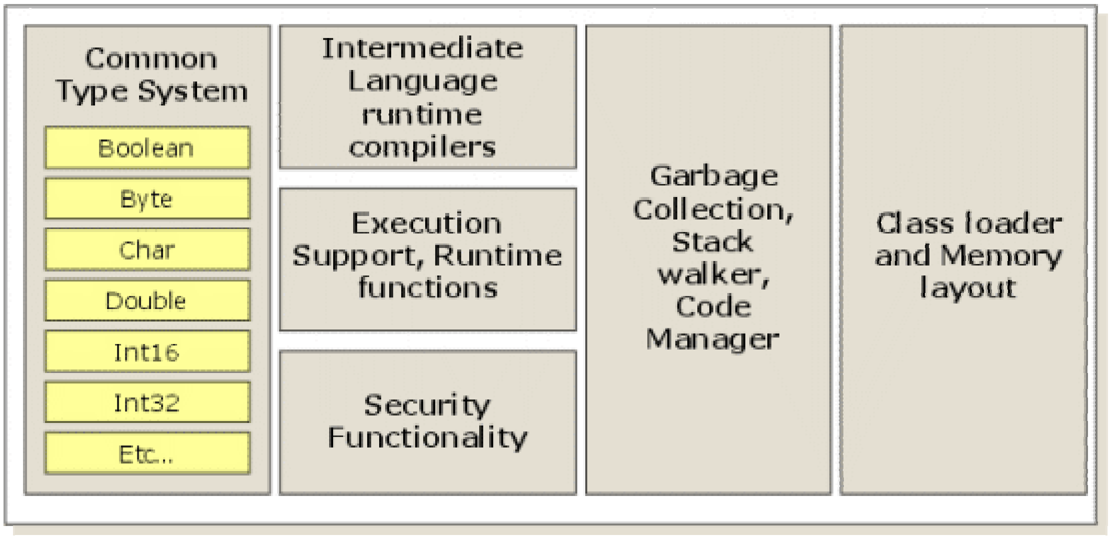
2.5 Understanding the various components of the .NET Platform and the functions performed by them

Now we will go in detail about the various components that build the .NET framework and its functionalities.

<u>Common Language Runtime</u>

At the core of the .NET platform is the Common Language Runtime (CLR). The CLR simplifies application development, provides a robust and secure execution environment, supports multiple languages and simplifies application deployment and management.

The diagram below provides more details on the CLR's features:


Common Type System
Boolean
Byte
Char
Double
Int16
Int32
Etc..
Intermediate Language runtime compilers
Execution Support, Runtime functions
Security Functionality
Garbage Collection, Stack walker, Code Manager
Class loader and Memory layout

In this section we will cover some of the more significant features provided to .NET applications by the CLR. These include:

- Memory Management
- Common Type System

Before moving further let us discuss briefly about Common Language Infrastructure(CLI) according to Standardizing Information and Communication Systems(ECMA) specifications. The Microsoft Shared Source CLI Implementation is a file archive containing working source code for the ECMA-334 (C#) and ECMA-335 (Common Language Infrastructure, or CLI) standards. In addition to the CLI implementation and the C# compiler, the Shared Source CLI Implementation from Microsoft called ROTOR contains tools, utilities, additional Framework classes, and samples.

For the benefit of existing codebases, the CLI standard also takes pains to describe in detail how unmanaged software can co-exist safely with managed components, enabling seamless sharing of computing resources and responsibilities.

Like the C runtime, the CLI has been designed to exploit the power of diverse platforms, as well as to complement existing tools, languages, and runtimes. Let's look at a few of the likely ways that the Shared Source CLI Implementation might interest you:

- There are significant differences in implementation between this code and the code for Microsoft's commercial CLR implementation, both to facilitate portability and to make the code base more approachable. If you are a developer who is interested in knowing how JIT compilers and garbage collectors work, or of how Microsoft Visual Studio works on your behalf under the covers, this distribution will definitely hold your attention!
- The distribution will help you in creating courseware around interesting topics that can be illustrated by this codebase.
- The distribution will help you in implementing your own version of the CLI and it also helps you in understanding the way the compilers and tools target the CLI.

Automatic Memory Management

Now let us discuss about an important feature of the CLR called Automatic Memory Management. A major feature of .NET framework CLR is that the runtime automatically handles the allocation and release of an object's memory resources. Automatic memory management enhances code quality and developer productivity without negatively impacting expressiveness or performance.

The Garbage Collector (GC) is responsible for collecting the objects no longer referenced by the application. The GC may automatically be invoked by the CLR or the application may explicitly invoke the GC by calling *GC.Collect*. Objects are not released from memory until the GC is invoked and setting an object reference to *Nothing* does not invoke the GC, a period of time often elapses between when the object is no longer referenced by the application and when the GC collects it.

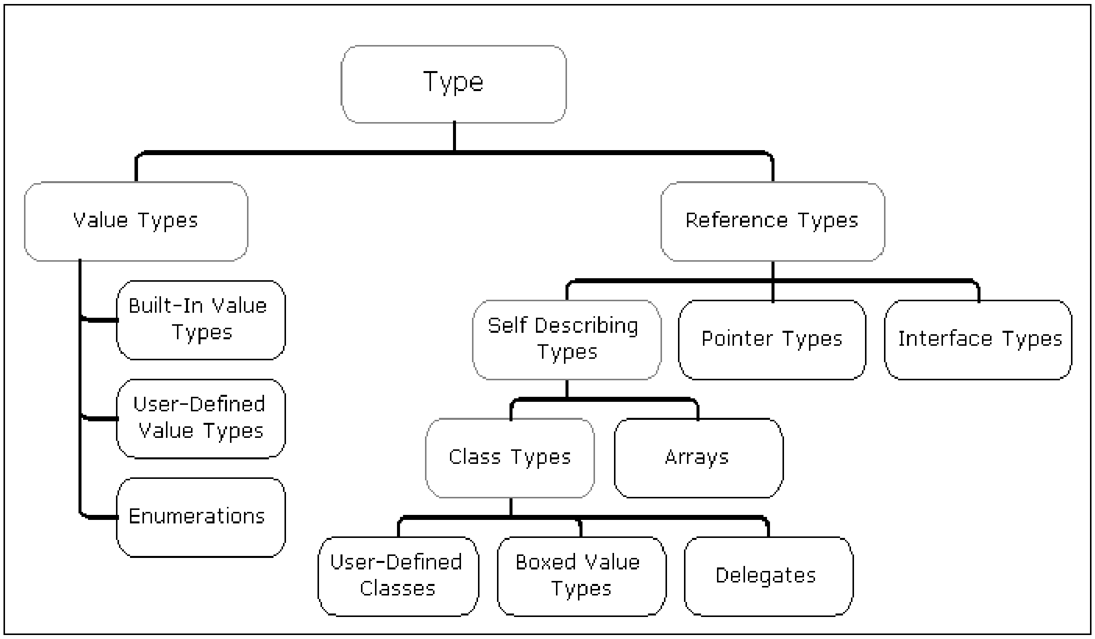
Common Type System

The Common Type System defines how data types are declared, used, and managed in the runtime, and is also an important part of the runtime's support for the Cross-Language Integration. The common type system performs the following functions:

- Establishes a framework that enables cross-language integration, type safety, and high performance code execution.
- Provides an object-oriented model that supports the complete implementation of many programming languages.
- Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.

The Common Type System can be divided into two general categories of types, Reference type and Value type each of which is further divided into subcategories.

Common Type System Architecture



The .NET type system has two different kinds of types namely Value types and Reference types.

Value types directly contain the data, and instances of value types are either allocated on the stack or allocated inline in a structure. Value types can be built-in (implemented by the runtime), user-defined, or enumerations.
The core value types supported by the .NET platform reside within the root of the *System* namespace. There types are often referred to as the .NET "Primitive Types".

They include:

- Boolean
- Byte
- Char
- DateTime
- Decimal
- Double
- Guid
- Int16
- Int32
- Int64
- SByte
- Single
- Timespan

Reference types store a reference to the value's memory address, and are allocated on the heap. Reference types can be self-describing types, pointer types, or interface types. The type of a reference type can be determined from values of self-describing types. Self-describing types are further split into arrays and class types.

Value Type vs. Reference Type

The primary difference between reference and value types is how instances of the two types are treated by the CLR. One difference is that the GC collects instances of reference types that are no longer referenced by the application. Instances of value types are automatically cleaned up when the variable goes out of scope. Let's take a look at an example in VB.NET:

```
Sub Test()
        Dim myInteger as Integer
        Dim myObject as Object
End Sub

‘myInteger a Value type is automatically cleaned up when the Sub ends.
‘But myObject a Reference type is not cleaned up until the GC is run.
```

Another difference is when one variable is set equal to another or passed as a parameter to a method call. When a variable of a *reference type (A)* is set equal to another variable of the same type *(B)*, variable *A* is assigned a reference to *B.* Both variables reference the same object. When a variable of *value type (A)* is set equal to another variable of the same type *(B)*, variable *A* receives a copy of the contents of *B.* Each variable will have its own individual copy of the data.

Yet another difference between the behaviors of value types versus reference types is how equality is determined. Two variables of a given reference type are determined to be equal if both the variables refer to the same object. Two variables of a given value type are determined to be equal if the state of the two variables are equal.

The final difference between the two is the way the instances of a type are initialized. In a reference type, the variable is initialized with a default value of *Null.* The variable will not reference an object until explicitly done by the object. Whereas a variable declared as a value type will always reference a valid object.

Custom Types

A Custom Type is a set of data and related behavior that is defined by the developer. A developer can define both custom reference type and custom value types.

In vb.net we can define custom types by using the *Structure* keyword. Let's look at an example wherein we define a custom value type.

```
Module Module1
        Public Structure Test
                Public myString as String
                Public myInteger as Integer
        End Structure

        Public Sub Main()
                ‘Notice that both declarations are equivalent
                ‘Both x and y are instance of type test

                Dim x as New Test()
                Dim y as Test

                x.myInteger = 4
                y.myString = “Test”

                ‘Reference to x is assigned to y
                y = x


                y.myInteger = 1
                y.myString = “Changed”

                Console.WriteKine(String.Format(“x :  myInt = {0}  and  String = {1} ”,  _
                                x.myInteger, x.myString))

                Console.WriteKine(String.Format(“y :  myInt = {0}  and  String = {1} ”,  _
                                y.myInteger, y.myString))
        End Sub
```

x and then set *y* equal to *x.* Since *x* and *y* are both instances of value types, *y* is set equal to the value of *x.* After changing the fields in *y* write the value of the fields in both *x* and *y* to the Console. The output of the program is:

```
x: myInt = 4 and myString = Test
y: myInt = 1 and myString = Changed
```

Notice that even after changing the value of fields in *y* it did not affect *x.* This is exactly the behavior required for primitive types.

Boxing and Unboxing Value Types

Sometimes it is required to treat an instance of a value type as if it were an instance of a reference type. An example of this is when a value type is passed *ByRef* as a parameter of a method. This is where the concept of Boxing becomes important.

Boxing occurs when an instance of a value type is converted to a reference type. An instance of a value type can be converted either to a *System.Object* or to any other interface type implemented by the value type.

```
Module Module1

        Public Function Add(ByVal x As Object, ByVal y As Object) As Object
                Add = x + y
        End Function

        Public Sub Main
                Dim x As Integer = 2
                Dim y As Integer = 3
                Dim sum As Integer

                Sum = Add(x , y)

                Console.WriteLine(“ {0} + {1} = {2} ”, x, y, sum)

        End Sub

End Module
```

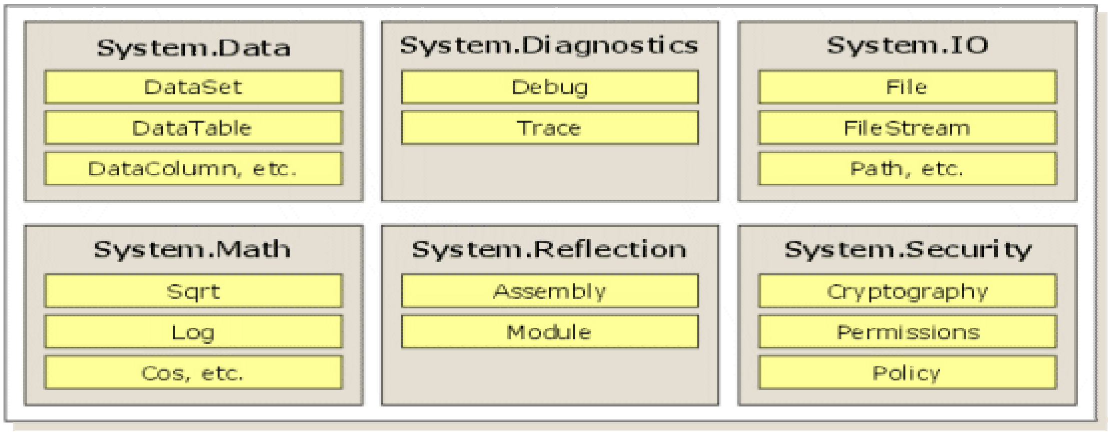
In the above example both *x* and *y* are boxed before they are passed to *Add.*

Then *x,y* and *Sum* are boxed before they are passed to *WriteLine.*
Unboxing involves the conversion of an instance of a reference type back to its original value type. In Vb.net it is done using the helper functions in the *Microsoft.VisualBasic.Helpers* namespace. For example in the above example, *IntegerType.FromObject* is called to unbox the return parameter of type *object* back to Integer.

More information about Common Type System can be obtained from
http://msdn.microsoft.com/library/en-us/cpguide/html/cpconcommontypesystemoverview.asp

The .NET Class Framework

We will now discuss about the .NET Class Framework. In conjunction with the CLR, the Microsoft has developed a comprehensive set of framework classes, several of which are shown below:



Since the .NET Class Framework contains literally thousands of types, a set of related types is presented to the developer within a single *namespace.* For example, the *System* namespace (which you should be most familiar with) contains the Object base type, from which all other types ultimately derive. In addition the *System* namespace contains types of integers, characters, strings, exception handling, and console I/O's as well as a bunch of utility types that convert safely between data types, format data types, generate random numbers, and perform various math functions. All applications use types from *System* namespace.

To access any platform feature, you need to know which namespace contains the type that exposes the functionality you want. If you want to customize the behavior of any type, you can simply derive your own type from the desired .NET framework type. The .NET Framework relies on the object-oriented nature of the platform to present a consistent programming paradigm to software developers. It also enables you to create your own namespaces containing their own types, which merge seamlessly into the programming paradigm. This greatly simplifies the Software Development.

The table below lists some of the general namespaces, with a brief description of what the classes in that namespace is used for:

Namespace	**Purpose of Class**
System	All the basic types used by every application.
System.Collections	Managing collections of objects. Includes the popular collection types such as Stacks, Queues, HashTables etc.
System.Diagnostics	Instrumenting and Debugging your application.
System.Drawing	Manipulating 2D graphics. Typically used for Windows Forms applications and for creating Images that are to appear in a web form.
System.EnterpriseServices	Managing Transactions, queued components, object pooling, just-in-time activation, security and other features to make use of managed code more efficient on the server.
System.Globalization	National Language Support(NLS), such as string compares, formatting and calendars.
System.IO	Doing Stream I/O, walking directories and files.
System.Management	Managing other computers in the enterprise via WMI.
System.Net	Network Communications.
System.Reflection	Inspecting metadata and late binding of types and their members.
System.Resources	Manipulating external data resources.
System.Runtime.InteropServices	Enabling managed code to access unmanaged OS platform facilities, such as COM components and functions in Win32 DLLs.
System.Runtime.Remoting	Accessing types remotely.
System.Runtime.Serilization	Enabling instances of objects to be persisted and regenerated from a stream.
System.Security	Protecting data and resources.
System.Text	Working with Text in different encodings, like ASCII or Unicode.
System.Threading	Performing asynchronous operations and synchronizing access to resources.
System.Xml	Processing XML Schemas and data.

In addition to the general namespace the .Net Class Framework offers namespaces whose types are used for building specific application types. The table below lists some of the application specific namespaces:

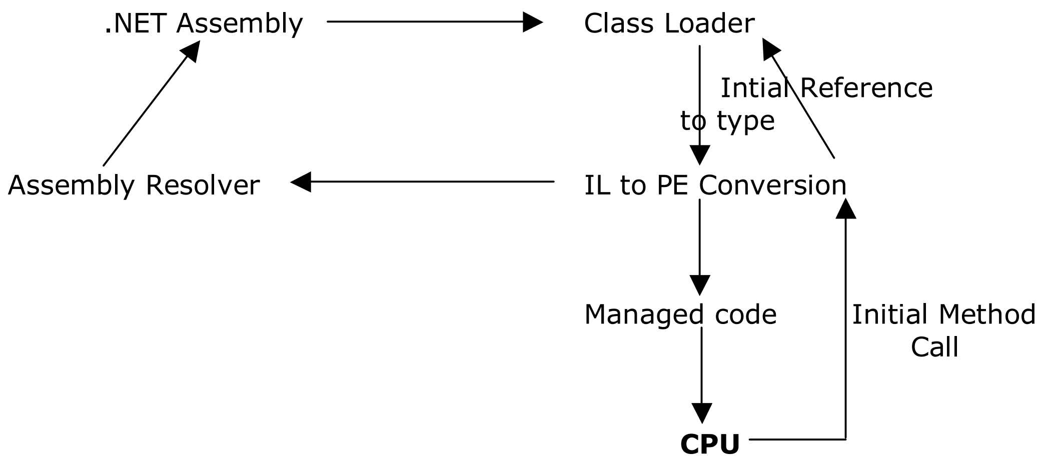
Namespace	**Purpose of Types**
System.Web.Services	Building web services
System.Web.UI	Building web forms.
System.Windows.Forms	Building Windows GUI applications.
System.ServiceProcess	Building a windows service controllable by Service Control Manager.

Refer the following link for .NET framework class library.
http://msdn.microsoft.com/library/en-us/cpguide/html/cpconthenetframeworkclasslibrary.asp

Just-In-Time Compilation (JIT)

The MSIL is the language that all of the .NET languages compile down to. After they are in this intermediate language, a process called Just-In-Time (JIT) compilation occurs when resources are used from your application at runtime. JIT allows "parts" of your application to execute when they are needed, which means that if something is never needed, it will never compile down to the native code. By using the JIT, the CLR can cache code that is used more than once and reuse it for subsequent calls, without going through the compilation process again.

The figure below shows the JIT Process:



JIT Compilation Process

The JIT process enables a secure environment by making certain assumptions:

- Type references are compatible with the type being referenced.
- Operations are invoked on an object only if they are within the execution parameters for that object.
- Identities within the application are accurate.

By following these rules, the managed execution can guarantee that code being executed is type safe; the execution will only take place in memory that it is allowed to access. This is possible by the verification process that occurs when the MSIL is converted into CPU-specific code. During this verification, the code is examined to ensure that it is not corrupt, it is type safe, and the code does not interfere with existing security policies that are in place on the system.

2.6 Structure of a .NET Application

DLL Hell

DLLs gave developers the ability to create function libraries and programs that could be shared with more than one application. Windows itself was based on DLLs. While the advantages of shared code modules expanded developer opportunities, it also introduced the problem of updates, revisions, and usage. If one program relied on a specific version of a DLL, and another program upgraded that same DLL, the first program quite often stopped working.

Microsoft added to the problem with upgrades of some system DLLs, like comctl.dll, the library used to get file, font, color and printing dialog boxes. If things weren't bad enough with version clashes, if you wanted to uninstall an application, you could easily delete a DLL that was still being used by another program.

Recognizing the problem, Microsoft incorporated the ability to track usage of DLLs with the Registry starting formally with Windows 95, and allowed only one version of a DLL to run in memory at a time. Adding yet another complication, when a new application was installed that used an existing DLL, it would increment a usage counter. On uninstall, the counter would be decremented and if no application was using the DLL, it could be deleted.

That was, in theory. Over the history of Windows, the method of tracking of DLL usage was changed by Microsoft several times, as well as the problem of rogue installations that didn't play by the rules--the result was called "DLL HELL", and the user was the victim.

Solving DLL hell is one thing that the .NET Framework and the CLR targeted. Under the .NET Framework, you can now have multiple versions of a DLL running concurrently. This allows developers to ship a version that works with their program and not worry about stepping on another program. The way .NET does this is to discontinue using the registry to tie DLLs to applications and by introducing the concept of an assembly.

On the .NET Platform, if you want to install an application in the clients place all you have to do is use *XCopy* which copies all the necessary program files to a directory on the client’s computer. And while uninstalling all you have to do is just delete the directory containing the application and your application is uninstalled.

Metadata

An Assembly is a logical DLL and consists of one or more scripts, DLLs, or executables, and a manifest (a collection of metadata in XML format describing how assembly elements relate). Metadata stored within the Assembly, is Microsoft's solution to the registry problem. On the .NET Platform programs are compiled into .NET PE (Portable Executable) files. The header section of every .NET PE file contains a special new section for Metadata (This means Metadata for every PE files is contained within the PE file itself thus abolishing the need for any separate registry entries). Metadata is nothing but a description of every namespace, class, method, property etc. contained within the PE file. Through Metadata you can discover all the classes and their members contained within the PE file.

Metadata describes every type and member defined in your code in a Multilanguage form. Metadata stores the following information:

- Description of the assembly
  - Identity (name, version, culture, public key).
  - The types that are exported.
  - Other assemblies that this assembly depends on.

    - Security permissions needed to run

- Description of types
    - Name, visibility, base class, and interfaces implemented.
    - Members (methods, fields, properties, events, nested types)

- Attributes
    - Additional descriptive elements that modify types and members

Advantages of Metadata:

Now let us see the advantages of Metadata:

Self describing files:
CLR modules and assemblies are self-describing. Module's metadata contains everything needed to interact with another module. Metadata automatically provides the functionality of Interface Definition Language (IDL) in COM, allowing you to use one file for both definition and implementation. Runtime modules and assemblies do not even require registration with the operating system. As a result, the descriptions used by the runtime always reflect the actual code in your compiled file, which increases application reliability.

Language Interoperability and easier component-based design:
Metadata provides all the information required about compiled code for you to inherit a class from a PE file written in a different language. You can create an instance of any class written in any managed language (any language that targets the Common Language Runtime) without worrying about explicit marshaling or using custom interoperability code.

Attributes:
The .NET Framework allows you to declare specific kinds of metadata, called attributes, in your compiled file. Attributes can be found throughout the .NET Framework and are used to control in more detail how your program behaves at run time. Additionally, you can emit your own custom metadata into .NET Framework files through user-defined custom attributes.

Assembly

Assemblies are the building blocks of .NET Framework applications; they form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. An assembly provides the common language runtime with the information it needs to be aware of type implementations. To the runtime, a type does not exist outside the context of an assembly.

An assembly does the following functions:

- It contains the code that the runtime executes.
- It forms a security boundary. An assembly is the unit at which permissions are requested and granted.

- It forms a type boundary. Every type's identity includes the name of the assembly at which it resides.
- It forms a reference scope boundary. The assembly's manifest contains assembly metadata that is used for resolving types and satisfying resource requests. It specifies the types and resources that are exposed outside the assembly.
- It forms a version boundary. The assembly is the smallest version able unit in the common language runtime; all types and resources in the same assembly are versioned as a unit.
- It forms a deployment unit. When an application starts, only the assemblies the application initially calls must be present. Other assemblies, such as localization resources or assemblies containing utility classes, can be retrieved on demand. This allows applications to be kept simple and thin when first downloaded.
- It is a unit where side-by-side execution is supported.

Contents of an Assembly

- Assembly Manifest
- Assembly Name
- Version Information
- Types
- Locale
- Cryptographic Hash
- Security Permissions

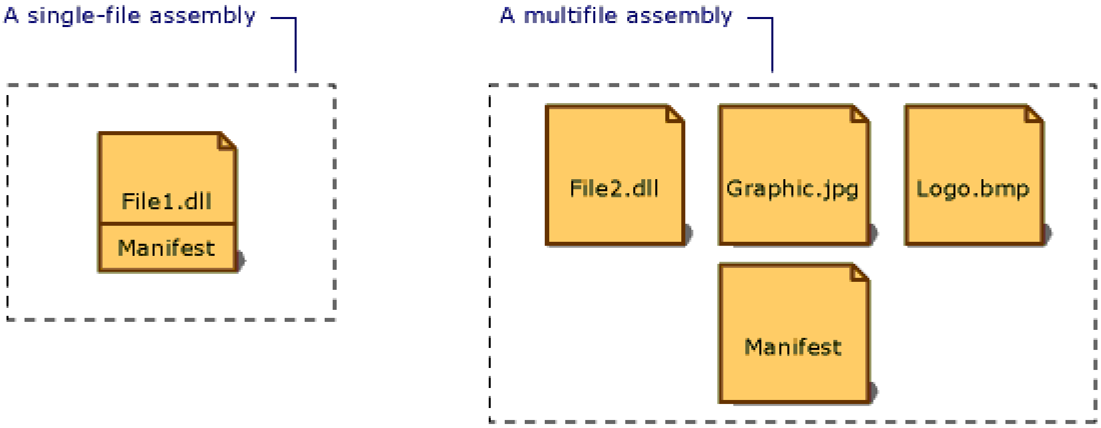
Assembly Manifest

Every assembly, whether static or dynamic, contains a collection of data that describes how the elements in the assembly relate to each other. The assembly manifest contains this assembly metadata. An assembly manifest contains the following details:

- Identity. An assembly's identity consists of three parts: a name, a version number, and an optional culture.
- File list. A manifest includes a list of all files that make up the assembly.
- Referenced assemblies. Dependencies between assemblies are stored in the calling assembly's manifest. The dependency information includes a version number, which is used at run time to ensure that the correct version of the dependency is loaded.
- Exported types and resources. The visibility options available to types and resources include "visible only within my assembly" and "visible to callers outside my assembly."
- Permission requests. The permission requests for an assembly are grouped into three sets: 1) those required for the assembly to run, 2) those that are desired but the assembly will still have some functionality even if they aren't granted, and 3) those that the author never wants the assembly to be granted.

In general, if you have an application comprising of an assembly named Assem.exe and a module named Mod.dll. Then the assembly manifest stored within the PE Assem.exe will not only contain metadata about the classes, methods etc. contained within the Assem.exe file but it will also contain references to the classes, methods etc, exported in the Mod.dll file. While the module Mod.dll will only contain metadata describing itself.

The following diagram shows the different ways the manifest can be stored:



For an assembly with one associated file, the manifest is incorporated into the PE file to form a single-file assembly. You can create a multifile assembly with a standalone manifest file or with the manifest incorporated into one of the PE files in the assembly.

The Assembly Manifest performs the following functions:

- Enumerates the files that make up the assembly.
- Governs how references to the assembly's types and resources map to the files that contain their declarations and implementations.
- Enumerates other assemblies on which the assembly depends.
- Provides a level of indirection between consumers of the assembly and the assembly's implementation details.
- Renders the assembly self-describing.

For more information on Assemblies refer:
http://msdn.microsoft.com/library/en-us/cpguide/html/cpconassemblies.asp

Modules

Modules are also PE files (always with the extension .netmodule) which contain Metadata but they do not contain the assembly manifest. And hence in order to use a module, you have to create a PE file with the necessary assembly manifest.
In C#, you can create a module using the /t:module compiler switch.

There are a few ways to incorporate a module into an Assembly. You can either use /addmodule switch to add module/s to your assembly, or you can directly use the /t:exe, /t:winexe and /t:library switches to convert the module into an assembly.

Difference between Module and Assembly

A module is an .exe or .dll file. An assembly is a set of one or more modules that together make up an application. If the application is fully contained in an .exe file, fine—that's a one-module assembly. If the .exe is always deployed with two .dll files and one thinks of all three files as comprising an inseparable unit, then the three modules together form an assembly, but none of them does so by itself. If the product is a class library that exists in a .dll file, then that single .dll file is an assembly. To put it in Microsoft's terms, the assembly is the unit of deployment in .NET.

An assembly is more than just an abstract way to think about sets of modules. When an assembly is deployed, one (and only one) of the modules in the assembly must contain the assembly manifest, which contains information about the assembly as a whole, including the list of modules contained in the assembly, the version of the assembly, its culture, etc.

Microsoft Intermediate Language (MSIL)

When compiling to managed code, the compiler translates your source code into Microsoft intermediate language (MSIL), which is a CPU-independent set of instructions that can be efficiently converted to native code. MSIL includes instructions for loading, storing, initializing, and calling methods on objects, as well as instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and other operations. Before code can be executed, MSIL must be converted to CPU-specific code by a just in time (JIT) compiler. Because the runtime supplies one or more JIT compilers, for each computer architecture it supports, the same set of MSIL can be JIT-compiled and executed on any supported architecture.

When a compiler produces MSIL, it also produces metadata. The MSIL and metadata are contained in a portable executable (PE file) that is based on and extends the published Microsoft PE and Common Object File Format (COFF) used historically for executable content. This file format, which accommodates MSIL or native code as well as metadata, enables the operating system to recognize common language runtime images. The presence of metadata in the file along with the MSIL enables your code to describe itself, which means that there is no need for type libraries or Interface Definition Language (IDL). The runtime locates and extracts the metadata from the file as needed during execution.

3. Code Management

Section Owner: Gurneet Singh (MVP)
Content Contributors: Anand M (MVP), C S Rajagopalan, G Gokulraj, G Arun Prakash

3.1 Introduction

We all know that there have been disparities between different languages such as VB, VC++ and developers, who code program through these languages. The disparity lies in terms of language features, performance, and flexibility in developing any piece of program. Well it's a known fact that, at the end, what matters is how efficiently your programs run on the client machine, no matter what language you use. Earlier this was driven by compilers, which were used to compile the code written using these languages to make it native code (processor specific).

With the release .NET framework Microsoft has driven out the disparities in such a way that no matter whatever .NET language you use to develop .NET applications, still the end result will be determined by .NET framework runtime and not by the language compilers as it was happening earlier. In this tutorial we will identify some of the key elements of the .NET framework through a simple program and concentrate on how .NET framework Runtime addresses platform or processor specific code issues to produce optimized code, which is native to the processor and to know how the framework helps in managing code effectively.

Common Language Runtime (CLR)

The primary function of a runtime is to support and manage the execution of code targeted for a language or a platform. For example, the Microsoft VC++ requires the msvcrt60.dll that contains its core support functionality. Even languages like Java have a run time, in the form of Java Virtual Machine.

The .Net platform also comes with a runtime that is officially called as the Common Language Runtime or simply the CLR. The CLR is designed to support a variety of different types of applications, from Web server applications to applications with traditional rich Windows user interface. Though the role of the CLR is similar to its counterparts in other languages or platforms, there are some key differences that make it one of the major features of the .NET platform. Here are the key differences between the .NET CLR and runtime of other languages:

- It is a common runtime for all languages targeting the .NET platform.
- It acts as an agent that manages code at execution time and also provides core services such as memory management, thread management and remoting.

- It enforces strict type safety and other forms of code accuracy that ensure security and robustness.
- It is responsible for enabling and facilitating the Common Type System. The Common Type System allows classes that are written in any .NET language to interoperate with—even inherit from, with overrides—classes written in any language. So your COBOL.NET program can interoperate with your C#, VB.NET, Eiffel.NET and with any other .NET language programs.
- It offers a mechanism for cross-language exception handling.
- It provides a more elegant way for resolving the versioning issues (also referred to as the Dll Hell in our classic COM).
- It provides a simplified model for component interaction.

Code that targets the runtime is known as managed code, while code that does not target the runtime is known as unmanaged code. Managed code requires a runtime host to start it. The responsibility of the runtime host is to load the runtime into a process, create the application domains (we'll look at this in detail later) within the process, and loads the user code into the application domains. While we can write our own runtime hosts using the set of APIs provided by Microsoft, the .NET platform by default ships with runtime hosts that include the following.

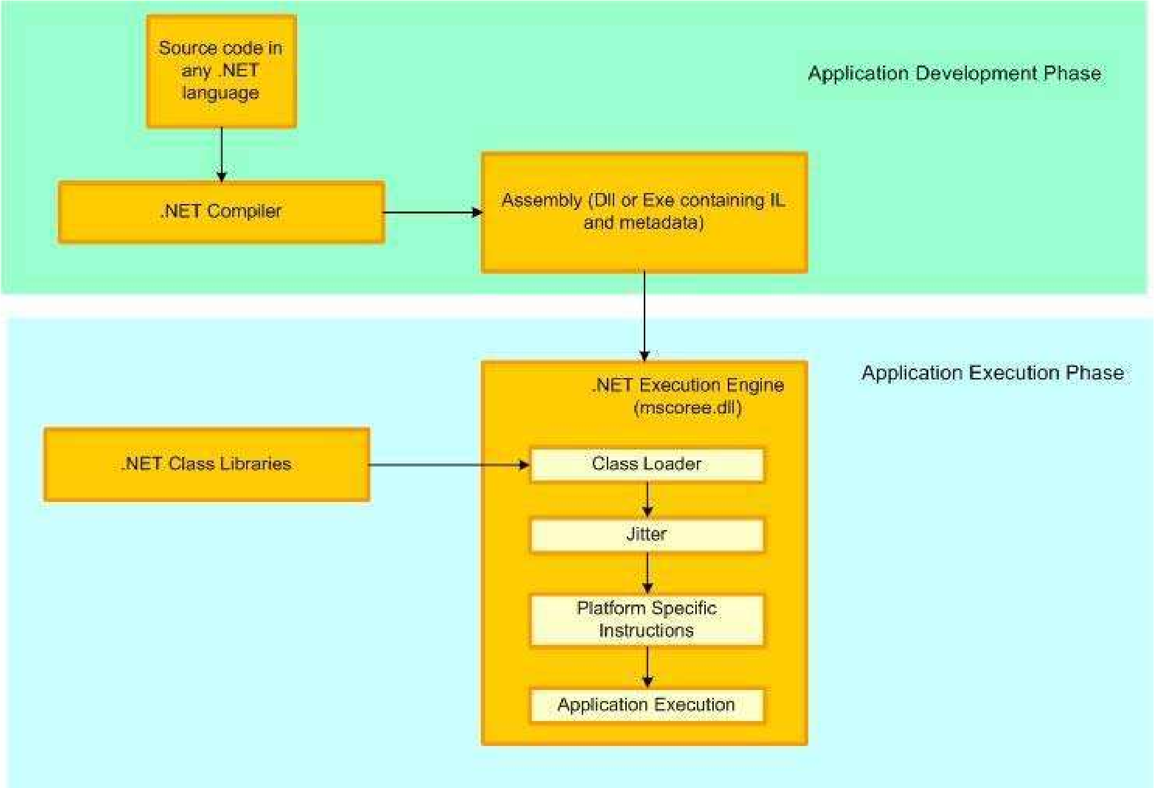
ASP.NET – Loads the runtime into the process that is to handle the Web request. ASP.NET also creates an application domain for each Web application that will run on a Web server.

Microsoft Internet Explorer – Creates application domains in which to run managed controls. The .NET Framework supports the download and execution of browser-based controls. The runtime interfaces with the extensibility mechanism of Microsoft Internet Explorer through a mime filter to create application domains in which to run the managed controls. By default, one application domain is created for each Web site.

Shell executables – Invokes runtime hosting code to transfer control to the runtime each time an executable is launched from the shell.

Now that you have understood conceptually the key features of the CLR in .NET framework, you can begin to look into the physical implementation and execution of code in the CLR.

The following figure illustrates the flow of activities from the source code to its execution.


Source code in any .NET language
Application Development Phase
.NET Compiler
Assembly (Dll or Exe containing IL and metadata)
.NET Execution Engine (mscoree.dll)
Application Execution Phase
.NET Class Libraries
Class Loader
Jitter
Platform Specific Instructions
Application Execution

3.2 First VB.NET / C# program

To start of with any language it’s always worth writing a simple program, which actually does nothing but displays a “HelloWorld” string in the screen. Taking this simple program we will try to figure out how .NET framework delivers a successful HelloWorld.exe. Well to write such a complex program we will go for our favorite editor, the choice is unanimous, it’s “Notepad”.

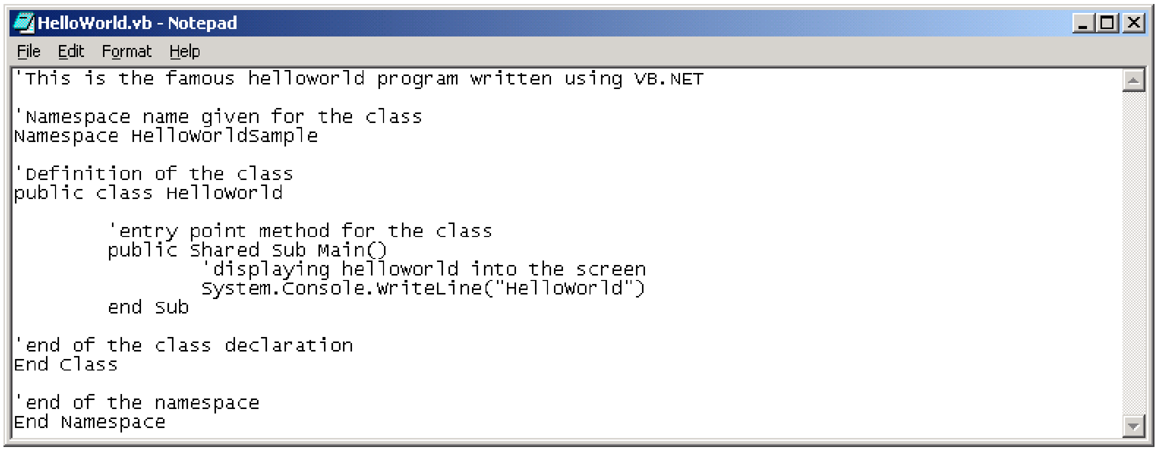
First VB.NET Program

Figure showing HelloWorld program written using VB.NET

```
'This is the famous HelloWorld Program written using VB.NET
Namespace HelloWorldSample

    'Definition of the Class
    Public Class HelloWorld
        'entry point method for the Class
        Public Shared Sub Main()
```

```
      System.Console.WriteLine("HelloWorld")
    End Sub
    'end of Class Declaration
  End Class
  'end of the Class Module
  'end of namespace
End Namespace
```

<u>This is how it goes in your favorite editor 'Notepad'</u>



```
'This is the famous helloworld program written using VB.NET

'Namespace name given for the class
Namespace HelloWorldSample

'Definition of the class
public class HelloWorld

        'entry point method for the class
        public Shared Sub Main()
                'displaying helloworld into the screen
                System.Console.WriteLine("HelloWorld")
        end sub

'end of the class declaration
End Class

'end of the namespace
End Namespace
```

Now let us spend sometime in examining the HelloWorld program to find out what's new in writing code through any .NET language.

The lines in the program that starts with a '(single quote) are comment entries like in other programming languages which are excluded in the compilation process. Like VB the way in which comment entries are represented remains the same in VB.NET.

Namespace HelloWorldSample - The keyword "Namespace" is new to some programmers who are not familiar with C++.

'Namespace' – a keyword in .NET is used to avoid name collisions i.e. For example, you develop a library which has a class named "File" and you use some other library which also has a class named "File", in those cases there are chances of name collision. To avoid this you can give a namespace name for your class, which should be meaningful. It is always better to follow the syntax (MS Recommended) given below while giving names for your namespaces

CompanyName.TechnologyName

However the hierarchy can be extended based on the implementation of the classes in the library.

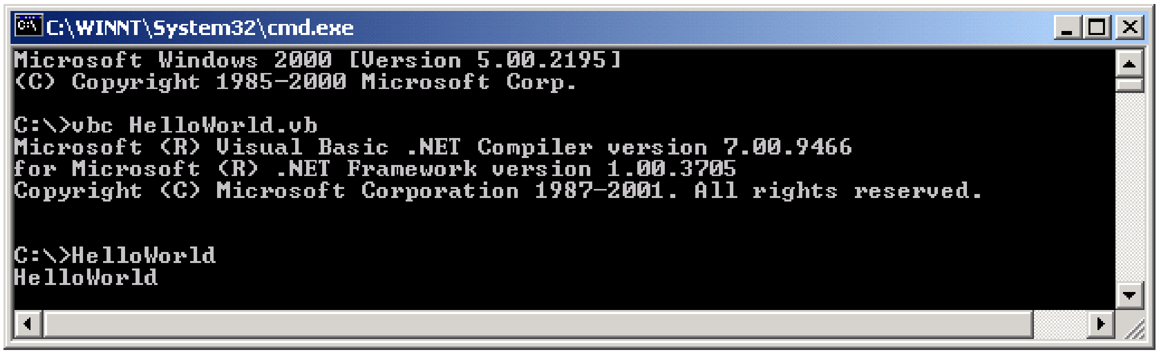
Public Class HelloWorld - This is the *class* declaration in VB.NET; the interesting thing for VB developers is that VB.NET is a fully object-oriented language (so everything is a Class here) . The class always ends with an “End Class”.

‘Public’ - is the modifier to determine the scope of the class (for other modifiers refer .NET framework SDK documentation or later parts of this tutorial). HelloWorld is the class name given for the class. Consumers of the class will be accessing through this name only

Public Shared Sub Main () - This is called as the entry point function because the runtime after loading your applications searches for an entry point from which the actual execution starts. C/C++ programmers will find this method very familiar (VB Programmers remember Sub Main). All Applications (exe) must have a definition for the Main Method. Try removing the Main method from your application and the compiler will complain that "No Start Point Defined". This means that the Main Method is the starting point of any application, in other words When you execute your Application "Main" method is called first automatically.
'Public' - This is the Access modifier for the Method. Since the Main method should be accessible to everyone in order for the .NET Runtime to be able to call it automatically it is always defined as public.
'Shared' - indicates that the method is a Class Method. Hence it can be called without making an instance of the class first.

Now its time to compile and execute this complex program. To compile the above piece of code you can use VB.NET compiler. To run the VB.NET compiler make sure you set your path variable to point to the place where your VB.NET compiler is available. *(To set a new value in the path variable, go to control panel and double click System icon, then choose advanced tab and click Environment Variables button to add or edit the environmental variables)*

Figure shows compilation of the HelloWorld program for VB.NET



```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>vbc HelloWorld.vb
Microsoft (R) Visual Basic .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.00.3705
Copyright (C) Microsoft Corporation 1987-2001. All rights reserved.


C:\>HelloWorld
HelloWorld
```

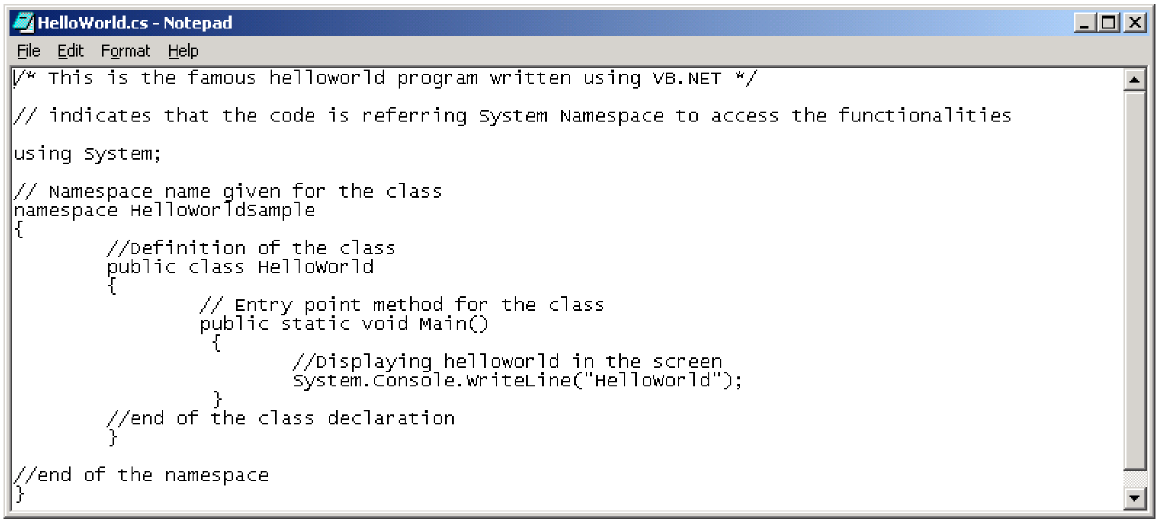
The compiler used here is “vbc”, which is a visual basic .net compiler accepts the source file “HelloWorld.vb” compiles the same to produce a program that’s not true executable, instead it generates something called assembly. Here the VB.NET compiler produces a

Managed Code/Intermediate Language (MSIL) format that uses instructions which are CPU-independent.

First C#.NET Program

```
/* This is the famous helloworld program written using C#.NET */

/* Indicates that the code is referring System Namespace to access the functionality's of System.dll */

using System;

// Namespace name given for the class
namespace HelloWorldSample
{
    //Definition of the class
    public class HelloWorld
    {
        // Entry point method for the class
        public static void Main()
        {
            //Displaying helloworld in the screen
            System.Console.WriteLine("HelloWorld");
        }
        //end of the class declaration
    }

    //end of the namespace
}
```

Figure showing HelloWorld program written using C#.NET in Notepad



```
/* This is the famous helloworld program written using VB.NET */

// indicates that the code is referring System Namespace to access the functionalities

using System;

// Namespace name given for the class
namespace HelloWorldSample
{
        //Definition of the class
        public class HelloWorld
        {
                // Entry point method for the class
                public static void Main()
                 {
                        //Displaying helloworld in the screen
                        System.Console.WriteLine("HelloWorld");
                 }
        //end of the class declaration
        }

//end of the namespace
}
```

The lines in the program that starts with a // and /*….*/ (comment blocks) are comment entries like in other programming languages which are excluded in the compilation process. For C or C++ programmers the C# style of coding sounds great because it almost follows the same style.

namespace HelloWorldSample - The keyword "namespace" is new to some programmers who are not familiar with C++.

'namespace' – a keyword in .NET is used to avoid name collisions i.e. For example, you develop a library which has a class named "File" and you use some other library which also has a class named "File", in those cases there are chances of name collision. To avoid this you can give a namespace name for your class, which should be meaningful. It is always better to follow the syntax (MS Recommended) given below while giving names for your namespaces

CompanyName.TechnologyName

However the hierarchy can be extended based on the implementation of the classes in the library.

public class HelloWorld - This is the *class* declaration in C#.NET; the interesting thing for C++ or Java developers is that they can apply the OOPS concepts that are supported by C#.NET . The class always ends with an "End Class".
'public' - is the modifier to determine the scope of the class (for other modifiers refer .NET framework SDK documentation or later parts of this tutorial). HelloWorld is the class name given for the class. Consumers of the class will be accessing through this name only.

public static void Main () - This is called as the entry point function because the runtime after loading your applications searches for an entry point from which the actual execution starts. C/C++ programmers will find this method very familiar (VB Programmers remember Sub Main). All Applications (exe) must have a definition for the Main Method. Try removing the Main method from your application and the compiler will complain that "No Start Point Defined". This means that the Main Method is the starting point of any application, in other words When you execute your Application "Main" method is called first automatically.

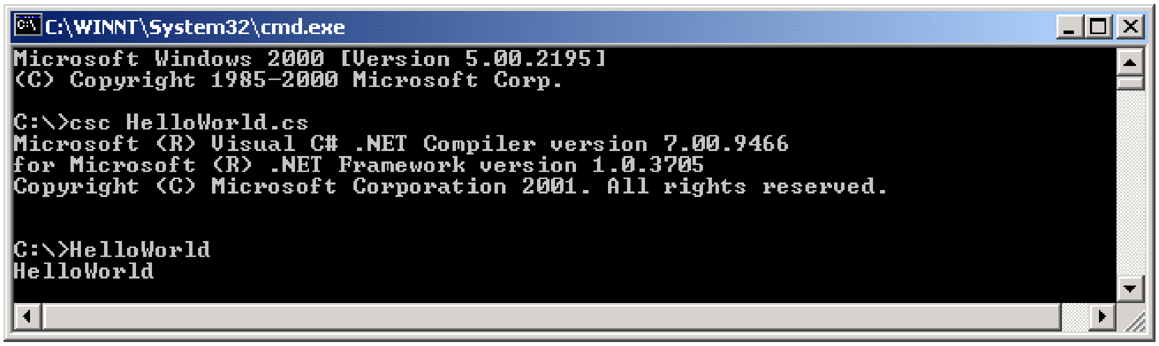
'public' - This is the Access modifier for the Method. Since the Main method should be accessible to everyone in order for the .NET Runtime to be able to call it automatically it is always defined as public.

'static' - indicates that the method is a Class Method. Hence it can be called without making an instance of the class first.

'void' – indicates the return type of the Main function, here in this case the Main function returns nothing so it is mentioned as void, for functions that returns value should have appropriate type such as long, string etc.,

Now its time to compile and execute this complex program. To compile the above piece of code you can use C# compiler. To run the C# compiler make sure you set your path variable to point to the place where your C# compiler is available. *(To set a new value in the path variable, go to control panel and double click System icon, then choose advanced tab and click Environment Variables button to add or edit the environmental variables)*

Figure shows compilation of the HelloWorld program using C# compiler



```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>csc HelloWorld.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.


C:\>HelloWorld
HelloWorld
```

The compiler used here is "csc", which is a visual basic .net compiler accepts the source file "HelloWorld.cs" compiles the same to produce a program that's not true executable, instead it generates something called assembly.

Assembly

An assembly is a grouping of files deployed as a single file. An assembly almost always consists of at least two files: the *executable* and the *manifest*. The manifest is a list of all the files that exist inside the assembly. The executable content inside the assembly is referred to individually as a *module*. Conceptually, modules correspond to DLLs or EXEs; each module contains metadata, in addition to the metadata of its parent assembly. The assembly format is an enhanced version of the current Portable Executable (PE) format (your normal Windows .EXE file format).

Manifest

Manifest is considered as the integral part of every assembly that renders the assembly self-describing. The assembly manifest contains the assembly's metadata and it also establishes the assembly identity, specifies the files that make up the assembly implementation, specifies the types and resources that make up the assembly, itemizes the compile-time dependencies on other assemblies, and specifies the set of permissions required for the assembly to run properly.

Metadata

The standard PE header comes at the beginning of the file. Inside the file is the CLR header, followed by the data required to load the code into its process space—referred to as *metadata*. It describes to the execution engine how the module should be loaded, what additional files it needs, how to load those additional files, and how to interact with COM and the .NET runtime.
Metadata also describes the methods, interfaces, and classes contained in the module or assembly. The information the metadata provides allows the JIT compiler to compile and run the module. The metadata section exposes much of your application's internals and eases the transition from disassembled IL to useful code.

3.3 JIT (Just–in-Time Compiler) & Debugging

The .NET Runtime ships with a Just-In-Time (JIT or JITter) compiler, which will convert the MSIL code in to the native code (CPU Specific executable code). So whatever code we write will be complied in to MSIL format and the JIT takes over when you run it.

The .NET runtime/Common Language Runtime (CLR) ships three different classes of JITters. The **Main JIT** compiler converts the MSIL code it to native code with out any optimizations. The **JIT** compiler takes the MSIL code and optimizes it. So this compiler requires lot of resources like, time to compile, larger memory footprint, etc. The **PreJIT** is based on the **Main JIT** and it works like the traditional compilers (compiles MSIL to native code during compilation time rather than runtime). This compiler is usually used at the time of installation.

No matter whatever language we used to develop the HelloWorld program, it’s a known fact that compiler’s are going to generate a MSIL format, once our code has been converted in to MSIL format, from MSIL format all the code that we write will be converted to native code in the same way whether if it is a VB.NET source or C# source.

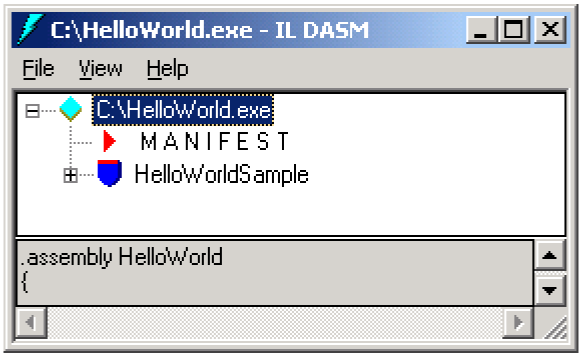
Intermediate Language (IL)

To support our discussion let us examine the IL code of HelloWorld program written using VB.NET and C#. To visualize the IL code Microsoft provides a disassembler tool through which you can easily see the IL code

To use the tool, choose command prompt and type ILDASM->ILDASM dialog is shown-> choose file open dialog and select the assembly

(make sure you set your path variable to point to the place where your ILDASM is available)

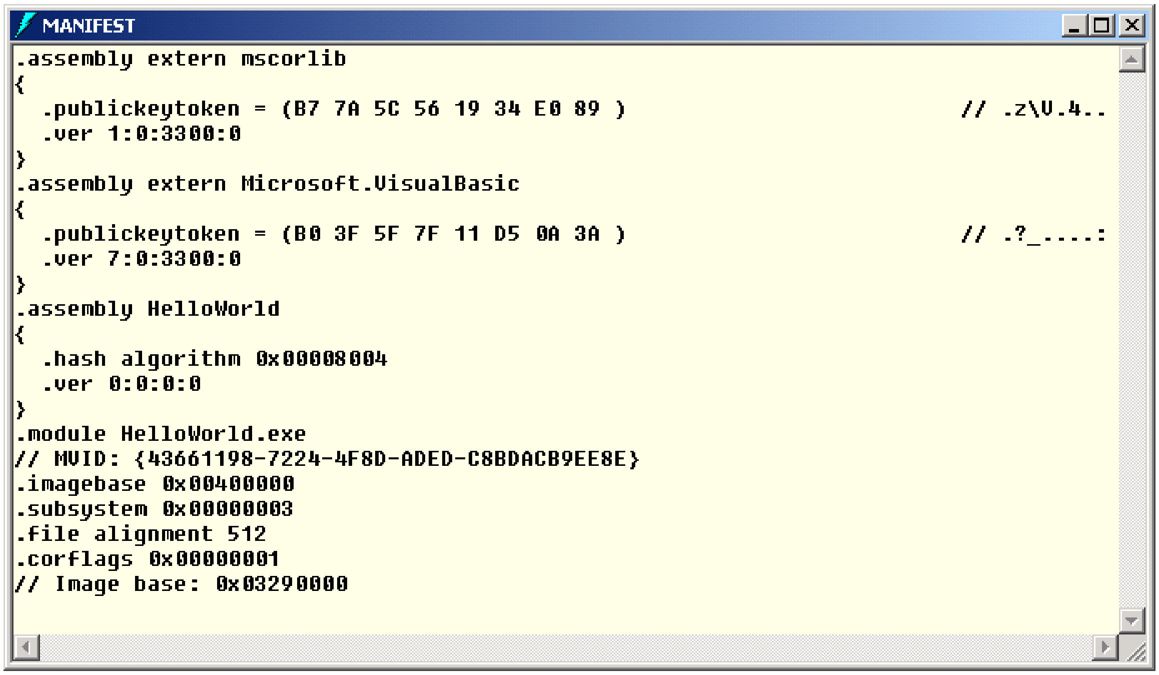
Figure showing disassembled HelloWorld program



```
.assembly HelloWorld
{
```

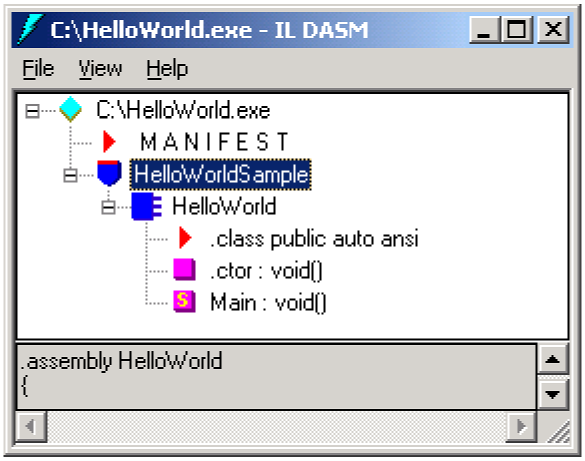
The above window showing a tree displays the path of the assembly as the root node, manifest information and namespace information as the child node (if you do not specify the namespace for the class then class name will be shown instead of namespace).

Figure showing manifest information of helloworld program



```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\V.4..
  .ver 1:0:3300:0
}
.assembly extern Microsoft.VisualBasic
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )                         // .?_....:
  .ver 7:0:3300:0
}
.assembly HelloWorld
{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module HelloWorld.exe
// MVID: {43661198-7224-4F8D-ADED-C8BDACB9EE8E}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x03290000
```

The manifest information shows the dependent assemblies like mscorlib, Microsoft.VisualBasic and their versions and it self describes the HelloWorld assembly. Since we have a simple program, which does not contain any embedded resource, the manifest does not include any information on those.

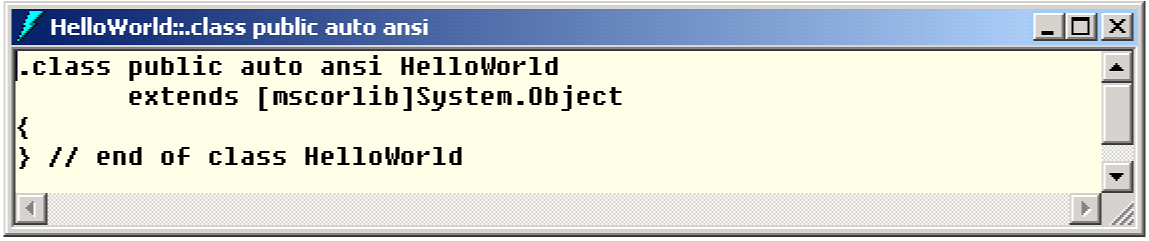


```
.assembly HelloWorld
{
```

Figure showing list of information present in the namespace

The above figure shows the list of information present within the namespace. In general the namespace contains the list of classes, structures, delegates, enums etc., In this case it shows the HelloWorld class which in turn contains the methods present in the class. It also shows the following information.

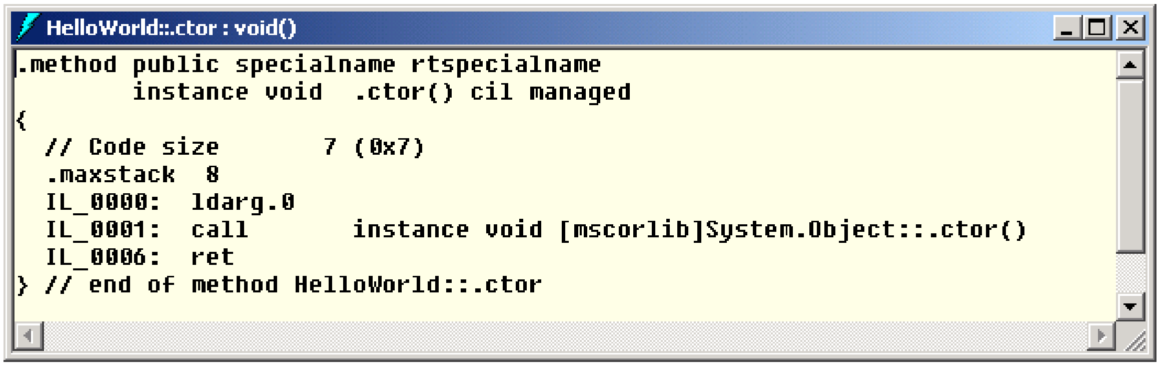
- *.class public auto ansi*



```
.class public auto ansi HelloWorld
       extends [mscorlib]System.Object
{
} // end of class HelloWorld
```

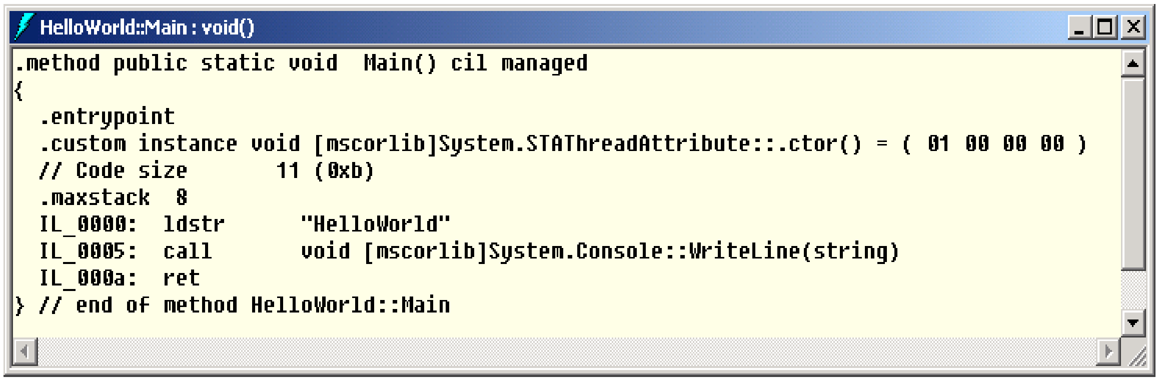
The above figure shows that HelloWorld is derived from System.Object, System.Object is the base class in the .NET framework

- *.ctor : void()*



```
.method public specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       7 (0x7)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
  IL_0006:  ret
} // end of method HelloWorld::.ctor
```

The above figure shows the IL code of the constructor of HelloWorld Class, you can see that it in turn calls System.Object::.ctor(), which is the base class's constructor



```
.method public static void  Main() cil managed
{
  .entrypoint
  .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() = ( 01 00 00 00 )
  // Code size       11 (0xb)
  .maxstack  8
  IL_0000:  ldstr      "HelloWorld"
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000a:  ret
} // end of method HelloWorld::Main
```

- *Main : void()*

The above figure shows the IL code of the Main function, which is the entry point for that assembly. It also shows the method "System.Console::WriteLine" is called with the string "HelloWorld " within the Main function

Compiler Options

If you can recollect the statement we have used to compile the HelloWorld program is *vbc HelloWorld.vb for Vb.NET and csc HelloWorld.cs for C#*, in this we have used the default settings of the compiler. Let us spend sometime in compiling the same code with some important options of the vbc /csc compiler.
In our program we have referred System.Dll assembly, in real life we would be application-referring lot of assemblies, in those cases the compiler should be intimated about the references. We can achieve this by the option mentioned below

/reference:<file-list> - needs to be used to indicate the list of references used by the application, in short it can also be represented as "/r". In our case it will be represented like this statement given below

```
vbc  /reference:"System.dll"  HelloWorld.vb  for Vb.NET

csc  /reference:"System.dll"  HelloWorld.cs  for C#.NET
```

The compiler by default will produce a HelloWorld.exe, in case you want to create a module or a library, then you have to specify the target in the compiler. It can be done like this

```
vbc /target:library /reference:”System.dll” HelloWorld.vb => to generate a library

csc /target:library /reference:”System.dll” HelloWorld.cs => to generate a library
```

Executing the above line of statement in the command prompt will generate a HelloWorld.dll, in the same manner a *module* can be generated by applying this switch /target:module

In case if we would like to give a different name to the assembly file then the statement given below can be applied

```
vbc /target:exe /out:”SampleHelloWorld.exe” /reference:”System.dll” HelloWorld.vb

csc /target:exe /out:”SampleHelloWorld.exe” /reference:”System.dll” HelloWorld.cs
```

In the above statement the switch */out:<filename>* is used to give a different name to the output assembly file.
The above compiler statements what we have seen is for simple applications, let us assume we have an application which is a Win32 executable and it has got resources, which could be embedded or linked (More on resource file later). An embedded resource could be an image for the splash screen, in those cases the following compiler options will be used

/target:winexe - used to create a Win32 executable file

/linkresource:<resource file(s)> - used to link a resource file to the assembly

/resource:<resource file(s)> - used to embed a resource file to the assembly

/imports:<import list> - used to include the list of namespaces used by the assembly

For other compiler options refer .Net framework SDK documentation.

.NET Debugging

Debugging is the most important feature of any programming language and Visual Studio .NET IDE provides this feature in an effective manner (but you can still do pretty good job with the .NET SDK alone). Application source code goes through two distinct steps before a user can run it. First, the source code is compiled to Microsoft Intermediate Language (MSIL) code using a .NET compiler. Then, at runtime, the MSIL code is compiled to native code. When we debug a .NET application, this process works in reverse. The debugger first maps the native code to the MSIL code. The MSIL code is then mapped back to the source code using the programmer's database (PDB) file. In order to debug an application, these two mappings must be available to the .NET runtime environment.

To accomplish the mapping between the source code and the MSIL, use the/debug:pdbonly compiler switch to create the PDB file (Note: When building ASP.NET applications, specify the compilation setting debug="true" in the application's Web.config file). The second mapping between the MSIL code and native code is accomplished by setting the JITTracking attribute in our assembly. By specifying the /debug compiler switch, the PDB file is created and the JITTracking attribute is enabled. When using this compiler switch, a debugger can be attached to an application loaded outside of the debugger.

Once the required mappings exist, there are several means by which to debug our applications. We can use the integrated debugger within Visual Studio .NET, or, if we prefer, we can use DbgClr, a GUI-based debugger. There is also a command line debugger, CorDBG that is included in the .NET Framework SDK.

3.4 Managed Vs. Unmanaged Methods/Transitions

In .Net Framework CLR provides execution of instruction targeted for CLR and the instructions that are not targeted for CLR. The instructions targeted for CLR are called as managed code and other type of code is called unmanaged code. After going through this topic you will know the following things mentioned below

Difference between Managed Method and Unmanaged Method.
Difference between Managed Type and Unmanaged Type.
How to call unmanaged methods in managed methods.
How to use unmanaged types.

When an application is launched for execution, first request is given to the Operating System, the OS will load the executable file in memory and starts executing the instruction from the entry point function in the executable file. Where in .NET executable file contains four main components the CLR header, the Metadata, the MSIL Code, and Native code.

CLR header will be used by the managed code in the module which will have the version number of the CLR on which the module is built and entry point method of the module in the executable.

Metadata describes the types used in the managed code, combination of CLR header and MSIL Code is the compiled format of a .Net language on a .Net Compiler, which will not have the instruction in targeted machine instruction format, which will again get compiled by the JIT Compiler

Native Code contains the machine instruction, which will be directly executed by the OS. Not all the .NET PE will have the Native code. PE of type EXE's will be having a native method like main() called as "unmanaged stub" which will be an entry point for the OS to execute code, that function will jump to _CorExeMain function located in MSCoree.dll which will be executed by the OS to initialize CLR and attach the running .NET module to CLR. Once CLR is initialized and loaded CLR will start executing the assembly by executing the managed entry point function specified in the CLR header of the file.

Managed Code

Machine instructions in MSIL format and located in Assemblies will be executed by the CLR, will have the following intrinsic advantages,
Memory management to prevent memory leaks in the program code,

- Thread execution,
- Code safety verification,
- Compilation,

Executed on many platforms like Windows 95, Windows 98, Windows 2000,and other system services.

Managed methods will be marked as "cil" in MSIL code. Methods marked with "cil" will be compiled by mscorjit.dll before execution. C# and VB.NET will generate only managed code. (Managed C++ can generate managed code by specifying "#pragma managed")

Unmanaged Code

Unmanaged codes are the instructions, which are targeted for specific platforms.
Unmanaged code will exist in any of the format,
A code instructions in managed C++ with "#pragma unmanaged"
COM/COM+ Components

Win32 Dlls/System Dlls

As these codes are in native format of OS, these instructions will be executed faster compared with JIT compilation and execution of Managed code.

Managed and Unmanaged transitions

As we get more benefits from managed code, still we may need to use unmanaged code. .Net provides many ways to access unmanaged code in managed code. Managed-Unmanaged transitions are achieved in .Net by set of services called Platform Invocation Services (P/Invoke) and IJW(It Just Works).

P/Invoke services are targeted for unmanaged code, which exists as COM/COM+ components and Win32 DLLs. COM/COM+ components will accessed by the concept called COM Interop - is a mechanism in which existing COM components will be accessed through a wrapper class called COM Callable Wrapper (CCW) in managed code without modifying the existing COM components. Using P/Invoke mechanism we can call Windows DLL functions in managed code.
IJW(It Just Works) targets code instructions built on C++ managed extensions, This mechanism is only for the code in Managed C++. In this way we can call the unmanaged methods directly by the managed code.

For example following code calls the MessageBox function in User32.dll(VB.NET)

```
Imports System.Runtime.InteropServices
Public Class Win32
Declare Auto Function MessageBox Lib "user32.dll" (ByVal
hWnd As Integer, _
ByVal txt As String, ByVal caption As String, ByVal Typ As
Integer) As Integer
End Class
Module Module1
     Sub Main()
         Win32.MessageBox(0, "Hello world" , "Temp path
is", 0)
     End Sub
End Module
```

Declare is the statement to state the Win32 API functions located in the Win32 DLLs. And the respective arguments declared with CLR data type.

In C#, we need to use the extern keyword with the attribute DLL Import to specify the Win32 DLL and the function should be declared as static function.

```
using System;
class PInvokeDemo
{
[dllimport("user32.dll")]
public static extern int MessageBox(int hwnd, string msg,
string caption, int type);

     public static int Main()
     {
          MessageBox(0, "Hello World!", "Tutorial", 1);
          return 0;
     }
}
```

In the above example MessageBox function is accessed from user32.dll by using the attribute DLL Import, and declared as static function

Managed Types and Unmanaged Types

We have seen how to call an unmanaged code in a managed code, now the question is how unmanaged code understands the managed data type and vice the versa. We will see how *string* will be sent from managed code and returned back to managed code. When passing a *string* value as an input argument to an unmanaged code CLR will take care of converting that to a native string type.

When we try to call a function, which returns string then the managed code, has to allocate memory and send to the function in unmanaged code. For example if we want to retrieve the OS Temp path then we can call GetTempPath API function of “kernel32.dll”,

Following code snippet shows how to call the function with string as an out argument with VB.NET.

```
Imports System.Runtime.InteropServices

Public Class Win32

           Declare    Auto    Function    MessageBox    Lib

"user32.dll" (ByVal hWnd As Integer, _

               ByVal   txt   As   String,   ByVal   caption   As

String, ByVal Typ As Integer) As Integer
```

```
      Declare Auto Function GetTempPath Lib "kernel32.dll"
(ByVal   lenOfChar   As   Integer,   ByVal   strData   As
System.Text.StringBuilder) As Integer
End Class
Module Module1
    Sub Main()
        Dim l As Integer
        Dim data As String
       Dim tempPath As System.Text.StringBuilder = New
System.Text.StringBuilder(255)
        l = 255
        Win32.GetTempPath(l, tempPath)
        Win32.MessageBox(0, tempPath.ToString(), "Temp path
is", 0)
    End Sub
End Module
```

Following code snippet shows how to call the function with string as an out argument with C#.

```
using System;
using System.Text;
class PInvokeDemo
{
[ dllimport("kernel32") ]
public static extern int GetTempPath ( int size,
StringBuilder buf );

     public static int Main()
     {
          const int size = 255;
          StringBuilder tempPath = new StringBuilder ( size
);
          GetTempPath ( size, tempPath );
          System.Console.WriteLine ( tempPath );
          return 0;
     }
}
```

The above code uses StringBuilder class of System.Text namespace to allocate string with 255 characters (Just think for this as a fixed string we used to have in VB).
Some of the functions in managed code will be accepting arguments as structures, structures in unmanaged code will be having set of member fields located in memory as the order in which it has been declared, that is the layout of member variable location is fixed. But in .Net, structures will have fields of managed data type and these member fields will automatically change the memory location of the structure. CLR will automatically move the data members to improve memory usage and performance. When an unmanaged method, which expects an argument as a structure, then managed code has to declare the structure so that it can be accessed by the unmanaged code. But .NET structures will have auto memory layout of data members, so to pass structures from managed code to unmanaged code has to be declared with an attribute *StructLayout* in System.Runtime.InteropServices namespace.

StructLayout is used with an enumerated value LayoutKind with following options given below:

- Auto – default option which changes the member field memory layout.
- Sequential - specifies member variable should be placed in a sequential order as specified while declaring the type.
- Explicit- specifies the exact location of the member variable in the structure.

3.5 Summary

Over the course of topics covered in this session you have seen how to create a simple HelloWorld program, to know the internals of the .NET Framework Runtime. For further understanding or clarification you can always use the .NET framework SDK help documentation and MSDN online.

4. Language Features of C#

Section Owner: Gurneet Singh (MVP)
Content Contributors: Amit Kukreja, Arvind Rangan, Reshmi Nair

4.1 History of C#

.NET framework offers a myriad of languages which puts us programmers into a deep thought process about which programming language best suits our needs.

Which language is the "best" language choice? If you are a VB wizard, should you take the time to learn C# or continue to use VB.NET? Are C# ASP.NET pages "faster" than VB .NET ASP.NET pages? These are questions that you may find yourself asking, especially when you're just starting to delve into .NET. Fortunately the answer is simple: there is no "best" language. All .NET languages use, at their root, functionality from the set of classes provided by the .NET Framework. Therefore, everything you can do in VB.NET you can do in C#, and vice-a-versa.

The differences occur in three main areas: syntax, object-oriented principles, and the Visual Studio .NET IDE. Syntax concerns the statements and language elements. Object Oriented differences are less obvious, and concern differences in implementation and feature sets between the two languages. IDE differences include things like compiler settings or attributes. There is also a fourth area of difference: language features that are present in one language but have no equivalent in the other.

If you are more familiar with Java, JScript, or C/C++, you may find C#'s syntax more familiar than VB.NET's.

A good question that has to be answered in order to keep you interested in learning C# is Why should you learn another programming language when you already doing enterprise development in C++ or Java. The very answer at the first go will be C# is intended to be the premier language for writing NGWS (Next Generation of Windows Services) applications in the enterprise computing world.

The programming language C# derives from C and C++; however apart from being entirely object oriented it is type safe and simple too. If you are a C/C++ programmer your learning curve should be flat. Many C# statements including expressions and operators have been taken directly taken from your favourite language

An important point about C# is that it simplifies and modernizes C++ in the areas of classes, namespaces and exception handling. Much of complex features have not been included or in fact hidden in C# to make it easer to use and more importantly less error prone for e.g. no more macros, templates and no multiple inheritances (Few of you might not like it.)

C# provides you with convenient features like garbage collection, versioning and lot more.

The only expense that I can think of is that your code operates in safe mode, where no pointers are allowed. However if you want to use pointers you are not restricted from using it via unsafe code- and no marshalling is involved when calling the unsafe code.

So you will learn a great deal of this new language in the coming Sections and see for yourself that how C# resembles or not resembles your favorite language

4.2 Language Fundamentals in C#

Constants & Variables

A variable is a named memory location. They are programming elements that can change during program execution. Data that needs to be stored in memory & accessed at a later time are stored in variables. Instead of referring to the memory location by the actual memory address you refer to it with a variable name.

Variables are declared as follows

```
        int a;
```

They can also be initialized at the time of declaration as follows:

```
        int a = 10;
```

Constants are very similar to variables. The main difference is that the value contained in memory cannot be changed once the constant is declared. When you declare a constant its value is also specified and this value cannot be changed during program execution.

Constants are used in situations where we need to keep the value in some memory location constant. If you use hard-coded values, and the value is changed then it has to be changed in all the locations in the code where it has been used. Instead if we are using constants, all we will need to do is to change the value of the constant. This would propagate the changes to our entire application.

Constants are declared as follows

```
        const int a;
```

Simple Types (Primitive Data types)
Simple or value type variables are those, which are assigned space in the stack instead of the heap. All the primitive types such as int, double etc are value type variables. The simple types basically consist of Boolean and Numeric types, where Numeric is further divided into Integral and Floating Point.

The first rule of value types is that they cannot be null. Anytime you declare a variable of value type, you have allocated the number of bytes associated with that type on the stack and are working directly with that allocated array of bits. In addition, when you pass a variable of value type, you are passing that variable’s value and not a reference to the underlying object.

Object Type
Object type or reference type variables are those, which are allocated storage space in the heap. Reference type objects can be null. When a reference type is allocated under the covers a value is allocated on the heap and a reference to that value is returned. There are basically four reference types: classes, interfaces, delegates and arrays.

Class Type
Custom data types are available in .NET framework in the form of classes or class type. It is nothing but a set of data and related behavior that is defined by the developer.

Object type and class type are both reference type variables. The only difference comes from the fact that object type consists of objects predefined and available with the .NET framework such as string whereas class type consists of custom user defined data types such as the class employee given below.

```
class employee
{
int empid;
string empname
public employee()
{
   empid = 10;
   empname = “Reshmi”;
}
}
```

Overloading and Overriding of the Class

Overloading provides the ability to create multiple methods or properties with the same name, but with different parameters lists. This is a feature of polymorphism. A simple example would be an addition function, which will add the numbers if two integer parameters are passed to it and concatenate the strings if two strings are passed to it.

```
using System;
public class test
{
public int Add(int x , int y)
{
 return(x + y);
}
public string Add(String x, String y  )
{
return (x + y);
}
public static void Main()
 {
   test a = new test ();
   int b;
   String c;
   b = a.Add(1, 2);
   c = a.Add("Reshmi", " Nair");
   Console.WriteLine(b);
   Console.WriteLine(c);
 }
}

O/P:
3
Reshmi Nair
```

Overriding

Class inheritance causes the methods and properties present in the base class also to be derived into the derived class. A situation may arise wherein you would like to change the functionality of an inherited method or property. In such cases we can override the method or property of the base class. This is another feature of polymorphism.

```
public abstract class shapes
{
        public abstract void display()
        {
                Console.WriteLine("Shapes");
        }
}



public class square: shapes
{
        public override void display()
        {
                Console.WriteLine("This is a square");
                }
}

public class rectangle:shapes
{
public override void display()
{
        Console.WriteLine("This is a rectangle");
        }
}
```

The above example is just an indication to how overriding can be implemented in C#.

Properties

Properties are named members of classes, structs, and interfaces. They provide a flexible mechanism to read, write, or compute the values of private fields through accessors.

Properties are an extension of fields and are accessed using the same syntax. Unlike fields, properties do not designate storage locations. Instead, properties have accessors that read, write, or compute their values.

Get accessor

The execution of the **get** accessor is equivalent to reading the value of the field.

The following is a **get** accessor that returns the value of a private field name:

```
private string name;   // the name field
public string Name   // the Name property
{
   get
   {
      return name;
   }
}
```

Set accessor

The **set** accessor is similar to a method that returns **void**. It uses an implicit parameter called **value**, whose type is the type of the property. In the following example, a **set** accessor is added to the Name property:

```
public string Name
{
  get
  {
    return name;
  }
  set
  {
    name = value;
  }
}
```

When you assign a value to the property, the **set** accessor is invoked with an argument that provides the new value. For example:

e1.Name = "Reshmi";   // The set accessor is invoked here

It is an error to use the implicit parameter name (**value**) for a local variable declaration in a **set** accessor.

How to make a Property Read Only/Write Only

There are times when we may want a property to be read-only – such that it can't be changed.

This is where read-only properties come into the picture. A Read Only property is one which includes only the get accessor, no set accessor.

```
public  read Only int empid
{
  get
  {
    return empid;
  }
}
```

Similar to read-only properties there are also situations where we would need something known as write-only properties. In this case the value can be changed but not retrieved. To create a write-only property, use the WriteOnly keyword and only implement the set block in the code as shown in the example below.

```
public  writeOnly int e
{
  set
  {
    e = value
  }
}
```

Structures

A structure allows you to create your own custom data types and it contains one or more members that can be of different data types. It can contain fields, methods, etc.

Structures are very similar to classes but there are some restrictions present in the case of structures that are absent in the case of classes. For example you cannot initialize structure members. Also you cannot inherit a structure whereas classes can be inherited. Another important feature of structures differentiating it from classes is that a structure can't have a default parameter-less constructor or a destructor. A structure is created on the stack and dies when you reach the closing brace in C# or the End structure in VB.NET.

But one of the most important differences between structures and classes is that structures are referenced by value and classes by reference. As a value type, allocated on the stack, structs provide a significant opportunity to increase program efficiency. Objects on the stack are faster to allocate and de-allocate. A struct is a good choice for data-bound objects, which don't require too much memory. The memory requirements should be considered based on the fact that the size of memory available on the stack is limited than the memory available on the heap.
Thus we must use classes in situations where large objects with lots of logic are required.

Struct – Code: Sample code showing the Class vs. Structures

```
using System;
class Test {
int classvar ;
int anothervar =20;
public Test ( )
{
classvar = 28;
}
 public static void Main()
     {
     Test  t  = new Test();
     ExampleStruct strct = new ExampleStruct(20);
     System.Console.WriteLine(strct.i);
     strct.i = 10;
     System.Console.WriteLine(t.classvar);
     System.Console.WriteLine(strct.i);
     strct.trialMethod();
     }
}
```

```
struct ExampleStruct {
 public int i;
 public ExampleStruct(int j)
{
  i = j;
 }

 public void trialMethod()
{
  System.Console.WriteLine("Inside Trial Method");
 }
}

O/P:-
     28
     20
     10
           Inside Trial Method
```

In the above example, I have declared and used a constructor with a single parameter for a structure. Instead if I had tried to use a default parameter-less parameter I would have got an error. But the same is possible in the case of classes as shown by the default parameter-less constructor, which initializes the classvar variable to 28.
Another point to note is that a variable called anothervar has been declared and initialized within the class whereas the same cannot be done for members of a structure.

Why Namespaces

Namespaces are used in .Net to organize class libraries into a hierarchical structure and reduce conflicts between various identifiers in a program. By helping organize classes, namespaces help programmers manage their projects efficiently and in a meaningful way that is understood by consumers of the class library. Namespaces enables reusable components from different companies to be used in the same program without the worry of ambiguity caused by multiple instances of the same identifier.

Namespaces provide a logical organization for programs to exist. Starting with a top-level namespace, sub-namespaces are created to further categorize code, based upon its purpose.

In .Net, the base class library begins at the **System** namespace. There are several classes at the **System** level such as **Console**, **Exception** etc. The namespace name gives a good idea of the types of classes that are contained within the namespace. The fully qualified name of a class is the class name prefixed with the namespace name. There are also several nested namespaces within the **System** namespace such as **System.Security**, **System.IO**, **System.Data, System.Collections** etc.

Reducing conflict is the greatest strength of namespaces. Class and method names often collide when using multiple libraries. This risk increases as programs get larger and include more third-party tools.

Boxing Conversions

Boxing is the implicit conversion of a value type to a reference type or to any interface type implemented by this value type. This is possible due to the principle of type system unification where everything is an object.

When boxing occurs, the contents of value type are copied from the stack into the memory allocated on the managed heap. The new reference type created contains a copy of the value type and can be used by other types that expect an object reference. The value contained in the value type and the created reference types are not associated in any way. If you change the original value type, the reference type is not affected. Boxing, thus, enables everything to appear to be an object, thereby avoiding the overhead required if everything actually were an object.

Example:

```
int n = 10;
Object obj;
obj = n;
```

Explanation:
In the above code segment, a **value-type** variable **n** is declared and is assigned the value **10**. The next statement declares an **object-type** variable **obj**. The last statement implicitly performs **boxing** operation on the variable **n**.

UnBoxing Conversions

UnBoxing is the explicit conversion from a reference type to a value type or from an interface type to a value type that implements the interface.

When unboxing occurs, memory is copied from the managed heap to the stack. For an unboxing conversion to a given value type to succeed at run time, the value of the source argument must be a reference to an object that was previously created by boxing a value of that value type otherwise an exception is thrown.

Example:

```
int n = 10;
int j;
Object obj;
obj = n;
j = (int)obj;
```

Explanation:
In the above code segment, another integer variable **j** is declared. The last statement performs explicit conversion of object-type to value-type i.e. integer.

Boxing and UnBoxing have performance implications. Every time a value type is boxed, a new reference type is created and the value type is copied onto the managed heap. Depending on the size of the value type and the number of times value types are boxed and unboxed, the CLR can spend a lot of CPU cycles just doing these conversions.

It is recommended to perform boxing and unboxing in a scenario where you have to pass a value parameter multiple times to a method that accepts a reference parameter. In such a case, it is advantageous to box the value parameter once before passing it multiple times to methods that accept reference methods.

Enumerations

Enumerations are types that inherit from **System.Enum.** The elements of an enumeration are expressed in words rather than numbers, which makes it convenient for understanding the meaning of the value being used. Enumerations symbolically represent a set of values of one of the primitive integral types.

The type of the elements of an enumeration can be **byte, short, int or long**. If no type is specified explicitly, the default type is **int**.

Example:

```
enum month : byte
{Jan = 2, Feb = 5, Mar = 10};
```

Explanation:
In the above code segment, an enumeration type **month** is declared. The underlying type of the elements has been specified as **byte**. It has three elements viz: **Jan, Feb and Mar.** These three elements have been assigned specific values. In case of an enumeration, if no values are specified, the value of the first element corresponds to 0 and so on.

Delegates

The runtime supports constructs called delegates, which enable late-bound operations such as method invocation and callback procedures. With delegates, a program can dynamically call different methods at runtime. They are type safe, secure, managed objects that always point to a valid object and cannot corrupt the memory of another object. The closest equivalent of a delegate in other languages is a function pointer, but whereas a function pointer can only reference **static** functions, a delegate can reference both **static** and instance methods. Delegates are **Marshal by Value** Objects.

The members of a delegate are the members inherited from class **System.Delegate**.
A delegate defines the signature and return type of a method. The resulting delegate can reference any method with a matching signature. Each instance of a delegate can forward a call to one or more methods that take those parameters and return the same type. Once a method has been assigned to a delegate, it is called when the delegate is invoked.

Example:

```
public delegate int calculation(int a,int b);

class mainclass
{
     calculation calc_delegate;
            public int add(int num1,int num2)
     {
          return num1 + num2;
     }

            static void Main()
     {
        int result;
        mainclass obj = new mainclass();
        obj.calc_delegate = new calculation(obj.add);

        result = obj.calc_delegate(50,70);
           }
}
```

Explanation:
Four steps are required to implement delegates viz.

- **Defining Delegates**
  The foremost step is to define the delegate. The definition of the delegate specifies the method signature, return type of the method, access modifier and the delegate name. The method signature specifies the order and type of each argument.

  The definition of a delegate is indicated by the usage of the **delegate** keyword. As shown in the above code segment, the delegate name is **calculation,** it's access modifier is **public,** it receives two integer arguments and returns an integer value.

- **Creating Delegate Method Handler(s)**
  The next step is to define the method(s) that will be associated with the delegate.
  In the above code segment, a method named **add** is defined. This method must have same method signature as that of the delegate, as shown in the above code segment.

- **Hooking up Delegates and Method Handlers**
  For a delegate method handler to be invoked, it must be assigned to a delegate object.
  In the above code, the delegate object is **calc_delegate** and is hooked up to the method handler **add**.

- **Invoking the method through the Delegate**

The last step is to invoke the methods that are associated with the delegate. A delegate method handler is invoked by making a method call on the delegate itself. This causes the method handler to invoke with the assigned input parameters as if they were invoked directly by the program, as shown in the above code.

4.3 Control Statements

C# has statements such as if ….. else ….. and switch…case which help you to conditionally execute program.
C# provides you with various looping statements, such as do… while, while, for and foreach….in.

1. **The if ….else….. Statement**

Consider a student marks and grade evaluation. For Marks above 75 the grade is 'A' and for below 75 is 'B'. In this situation when you need to execute some code based on some condition, you can make use of, **if …. else …...**

The Cultural *syntax* normally used is as follows:

```
if (condition)
{
        Executable statements when the condition is True
}
else
{
        Executable statements when the Condition is False
}
```

OR using else if for Advanced Decision making

```
if (condition)
{
        Executable statements
}
else if (condition)
{
        Executable statements
}
```

Single if can have multiple else if with conditions, as mentioned above in *else if* format.

Nesting if …else Constructs

if (*condition***)**

```
{
            if (condition)
            {
                        Executable statements when the condition2 is TRUE
            }
            else
            {
                        Executable Statements
            }
else
{
            Executable statements
}
```

One important thing to keep in mind when nesting if…else constructs is that you must have remember to close the brace ({ }) for every brace that you open.

2. The switch…case Statement.

The **switch** statement is a control statement that handles multiple selections by passing control to one of the **case** statements within its body.

The **switch…case** Statement is similar to **if…else**. The only difference between two is that **if** and **else if** can evaluate different expressions in each statement, but the **switch** statement can evaluate only one expression.

The drawback of if...else construct is that it isn't capable of handling a decision situation without a lot of extra work. One such situation is when you have to perform different actions based on numerous possible values of an expression, not just True or False. For instance performing actions based on Students Grade.

```
if ( Grade.Equals (“A”))
{
      …….
}
else if (Grade.Equals (“B”))
{
      ……
}
else if (Grade.Equals (“C”))
{
      ……
}
else if (Grade.Equals (“D”))
{
      ……
}
else
{
.….
}
```

As you see the structure can be a bit hard to read and if the conditions increase you may end up writing a confusing and an unreadable piece of Code

The **switch** uses the result of an expression to execute different set of statements.
The *syntax* for the **select…case** Statement is as follows:

```
switch (expression)
{
  case constant-expression:
    statement
    jump-statement
  default:
    statement
    jump-statement]
}
```

Notice that the *jump-statement* is required after each block, including the last block whether it is a **case** statement or a **default** statement.

Note: default is used to define the code that executes only when the expression doesn’t evaluate to any of the values in case Statements .Use of default case is optional

Let’s see the same example as above but this time with switch case.

```
switch(grade)
{
      case  “A”:
             Executable statements
             jump-statement
     case  “B”:
            Executable statements
             jump-statement
     case  “C”:
             Executable statements
             jump-statement
     case  “D”:
             Executable statements
             jump-statement
     default :
             Executable statements
```

```
               jump-statement
}
```

Branching is performed using jump statements, which cause an immediate transfer of the program control.(break, continue, default ,goto ,return)

Evaluating More than one possible Value in a case Statement is not possible in C#, but VB.Net does allow evaluating more than one Value.

3. for Statements.

The **for** loop executes a statement or a block of statements repeatedly until a specified expression evaluates to **false.**

for ([initializers]; [expression]; [iterators]) statement
where:

Initializers: A comma separated list of expressions or assignment statements to initialize the loop counters.

Expression : expression is used to test the loop-termination criteria.

Iterators : Expression statement(s) to increment or decrement the loop counters.

Example print numbers from 1 To 100

```
for (int intctr =1; intctr <= 100; intctr++)
     Debug.WriteLine(intctr);
```

This routine starts a loop with a for statement after a variable intctr is declared. This loop initializes intctr to 1 and then prints 1 through 100 to the output window.

Note: you can declare variable in the Initialization part of the for loop separated by comma.

Example : print even number from 1 to 100

```
for(int i=2; i <= 100; i = i + 2)
     Debug.WriteLine(i.ToString());

Example : To Sum the total of all number from 1 to 10
for( i =1 ; i < 11 ; i++)
     sum = sum + i ;

Example : The statement below can be used as an infinite loop.
for ( ; ; );
```

Example of use of for loop.

Let us see how to a write table of 2 using for loop.

```
for(int j = 1,i = 2; j <= 10; j++)
     Debug.WriteLine("2 X " +  j.ToString() + " = " + i*j );
```

Output*:*

```
2 X 1 = 2
..
..
..
..
..
…
2 X 10 = 20
```

An Example of Nested For loop.

Let us write a small code to display a structure of stars '*' in triangle format.

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

Let us have a label with name **stars**. Increase the height of the label to get a clear view of the image.

```
string star="";
for(int i = 0; i < 5 ;i++) // First loop to count the rows
{
     for (int j = 0; j <= i; j++)  // Second loop to count
the columns
     {
star = star + " * ";
     }
     Debug.WriteLine(star);
     star = "";
}
```

Note: For better readability you must always indent your Codes.

4. foreach…in Statement

The **foreach…in** Statement is used to repeat a set of statements for each element in an array or collection.

The **foreach…in** statement is executed if there is at least one item in an array of collection. The Loop repeats of each element in an array or collection.

The *syntax* for the **foreach…in** statement as follows:

foreach (type *Component* **in** Set **)**
{
Executable statements
}

Component is the variable used to refer to the elements of an array or a collection.
Set refers to an array or any collection object.
e.g.

```
string[]  weeks  = {"Monday", "Tuesday", "Wednesday",
"Thursday",
               "Friday", "Saturday", "Sunday"};
      foreach(string eachday in weeks)
     {
            MessageBox.Show(eachday);
     }
```

An example for using foreach element in a collection of string into a single string element.
Each element of array which is of type string is read from the collection and stored into the single string type object.

5. while…Statement

The **while…** Statement is used to repeat set of executable statements as long as the condition is true.

The *syntax* for the **while…** statement is as follows:

while (*Condition)*
{
Executable Statements

```
                }
```

In this if the condition is satisfied then the statements are executed. Else it will not enter the **while** condition at all.

example of infinite loop is
while (true);

```
Example print numbers from 1 to 100
int i = 1;
while ( i <= 100)
{
     Debug.WriteLine(i.ToString());
     i++;
}

Example print even numbers from 1 to 100
Int i = 2;
While( i <=100)
{
     Debug.WriteLine(i.ToString());
     i = i + 2;
}
```

6. do...while Statement

The **do...while** Statement is similar to **while…** Statement**.**

```
do
{
        Statements to Execute
}
while (condition)
```

```
example print numbers from 1 to 100
int i = 1;
do
{
     Debug.WriteLine(i.ToString());
     i++;
}while( i  <= 100);

example print even numbers from 1 to 100
int i = 2;
do
{
     Debug.WriteLine(i.ToString());
     i = i + 2;
}while( i  <= 100);
```

A Complete Example with set of control statements.

We will create a C# application, which will accept students name and its grade. Depending upon the type of grade it will add remarks.

```
int value=0, ctr=0;

//Accept a number from the user
Console.Write("Enter the number of students : ");
value = Int32.Parse(Console.ReadLine());


string [] arrName= new string[value];
string sGrade ="";
string [] arrRemarks= new string[value];

while (ctr < value)
{
     //Accept the name of the students
     Console.Write("Enter the name of the Student" + (ctr +
1) + " : ");
     arrName[ctr] = Console.ReadLine();

     //Accept the grade of the Student
     Console.Write("Enter the grade of the student
A/B/C/D/F : " );
     sGrade = Console.ReadLine();

     // Assign remarks to  students
     switch (sGrade.ToUpper())
     {
          case "A":
               arrRemarks[ctr] = "Excellent";
               break;
          case "B":
               arrRemarks[ctr] = "Good";
               break;
          case "C":
               arrRemarks[ctr] = "Fair";
          case "D":
               arrRemarks[ctr] = "Poor";
               break;
          case "F":
               arrRemarks[ctr] = "Fail";
               break;
          default:
               Console.WriteLine("Incorrect value entered
");
```

```
               return; // To come out of the program
     }

ctr = ctr + 1;
}

// Display the summary on the Console
for (ctr = 0 ;ctr< value; ctr=ctr+1)
{
      if (arrRemarks[ctr].Equals("fail"))
     {
          Console.WriteLine(arrName[ctr] + " has failed in_
exams ");
     }
      else
     {
           Console.WriteLine(arrName[ctr] +  "'s
performance is " +  arrRemarks[ctr]);
     }

}
```

Note use of ToUpper() and ToLower() used to Convert all alphabetic characters have been converted to Upper Case / Lower Case .

4.4 Arrays

Till now we have been using variable to store values. We might come across a situation when we might need to store multiple values of similar type. Such as names of 100 students in a school. One way to do it is to declare 100 variables and store all the names.

A much more simple and efficient way of storing these variable is using *Arrays*. An *Array* is a memory location that is used to store multiple values.

All the values in an array are of the same type, such as int or string and are referenced by their index or subscript number, which is the order in which these values are stored in the array. These values are called the elements of the array.
The number of elements that an array contains is called the length of the array.

In C# all arrays are inherited from the *System.Array* Class.

Arrays can be single or multidimensional. You can determine the dimensions of an array by the number of subscripts that are used to identify the position of any array element.
A single dimensional array is identified by only a single subscript and an element in a two-dimensional array is identified by two subscripts.
Arrays in C# also support the concept of Jagged Arrays.

The dimension has to be declared before using them in a program. The array declaration comprises the name of the array and the number of elements the array can contain.

```
The Syntax of single dimension array is as follows.

Datatype [] ArrayName  = new DataType[number of elements];

e.g.
string [] studentname = new string [5];
```

You can assign the values at runtime or even at the design time.

Design time declaration:
Studentname [0]=”Rohan”
Studentname [1]=”Mohan”
…..
Studentname[10]=”Nitin”

All arrays starts with the index of ***0*** i.e. All arrays are Zero Based. This implies that above array can store 10 elements. Here ***0***, is the starting index or the lower bound of the array and 9 is the upper bound while the length of the array is 10.

Example 1.

We will create a C# Console Application that will accept the names of students in an single dimension array and display it back.

```
int value = 0,  cnt = 0;

//Accept how many students names to enter
Console.Write("Enter the number of students name to enter:
");
value = System.Int32.Parse(Console.ReadLine()) ;


string[] arrnames = new string [value];

for(cnt = 0; cnt<value;cnt++)
{
     Console.Write("Enter the name of student " + (cnt + 1)
+  ":", "Student Name");
     arrnames[cnt] = Console.ReadLine();
}

Console.WriteLine("Pulling Values from the Array");

//Display the entered value to the text box
for(cnt = 0; cnt < value; cnt++)
{
     Console.WriteLine(arrnames[cnt]);
}
```

Above example will accept number of names to be entered and will add the names in a loop and then redisplay it on the Console. Note that we have not written any error handling code that is left to the reader as an exercise.

The *Syntax* for multi-dimension arrays is as follows:
Previously we saw how we can store multiple names of students. But, if we want to store related data of students like first name, middle name, last name. In such situations you can use multi dimension arrays, such as two-or-three dimension arrays.

```
Datatype [] ArrayName  = new Datatype[number of 1st element,
number of 2nd element,….];
```

e.g.
string[,] studentdetails = new string [10,2];

Index positions of array elements.

0,0	0,1
1,0	1,1
2,0	2,1
3,0	3,1
…	
10,0	10,1

studentdetails(0,0) = “Manoj”
studentdetails(0,1) = “Malik”

To display “Malik” we need to use the index position of the array and say , Studentdetails [0,1].

Example 2.

We will create a C# Console Application, which will accept Student Name, Address and city name and display it on the Console.

```
string [,] arrsummary = new string[3, 3];
int i=0, j=0;

//As we wanted just 3 columns we have set it to 2, else if
u want to be two only then while declaring the array make
it (2,2) as the lower index is 0.

for(i = 0;i<=2;i++)
{
     for(j = 0;j<=2;j++)
     {
          Console.WriteLine("Enter the value for " + i + "
row and " + j + " column, Summary");
          arrsummary[i, j] = Console.ReadLine();
     }
}


Console.WriteLine();

//Display the values in the summary array.
for(i = 0;i<=2;i++)
{
     string s = "";
     for(j = 0;j<=2;j++)
     {
          if (s.Equals(""))
          {
               s =  arrsummary[i, j];
          }
          else
          {
               s =  s + " - " + arrsummary[i, j];
          }
     }
     Console.WriteLine(s);
}
```

Jagged Arrays

A jagged array is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes. A jagged array is sometimes called an "array-of-arrays."

//Decalaring a Jagged Array
int[][] myJaggedArray = new int[3][];

```
//Initialize the elements of the Jagged Array
myJaggedArray[0] = new int[5];
myJaggedArray[1] = new int[4];
myJaggedArray[2] = new int[2];

//Fill array elements with values.
myJaggedArray[0] = new int[] {1,3,5,7,9};
myJaggedArray[1] = new int[] {0,2,4,6};
myJaggedArray[2] = new int[] {11,22};

You can access individual array elements like these examples:
// Assign 33 to the second element of the first array:
myJaggedArray[0][1] = 33;
// get the value of second element of the third array:
int i = myJaggedArray[2][1];
```

Few Important methods in arrays.

GetUpperBound(), GetLowerBound() are the functions used to get the bound of a array. These methods of the array class**.** You can use it with single dimension as well as for multi-dimensional arrays.

GetLowerBound() is to get the upper limit of an array.
GetUpperBound is to get the lower limit of an array.

e.g.

```
string[]  weeks  = {"Monday", "Tuesday", "Wednesday", "Thursday",
                              "Friday", "Saturday", "Sunday"};
MessageBox.Show(weeks.GetUpperBound(0).ToString());
//the above statement returns 6 as the upper bound for the array weeks.
```

Syntax : arrayname.GetUpperBound/GetLowerBound(dimension)

Dimension refers to the which upper/lower bound should be found, 0 for first, 1 for second and so on.

e.g.

```
string [,,] student_details = new string[10,20,15];
int upperlimit = 0;

// This will return 10 for the 1st row element
upperlimit = student_details.GetUpperBound(0);
MessageBox.Show(upperlimit.ToString());

// This will return 20 for the 2nd row of element
upperlimit = student_details.GetUpperBound(1);
MessageBox.Show(upperlimit.ToString());
```

For all GetLowerBound(dimension) it will return 0 as the base lower bound is zero in .NET

5. Language Features of VB.NET

Section Owner: Gurneet Singh (MVP)
Content Contributors: Amit Kukreja, Arvind Rangan, Reshmi Nair

5.1 History of VB.NET

.NET framework offers a myriad of languages which puts us programmers into a deep thought process about which programming language best suits our needs.

Which language is the "best" language choice? If you are a VB wizard, should you take the time to learn C# or continue to use VB.NET? Are C# ASP.NET pages "faster" than VB.NET ASP.NET pages? These are questions that you may find yourself asking, especially when you're just starting to delve into .NET. Fortunately the answer is simple: there is no "best" language. All .NET languages use, at their root, functionality from the set of classes provided by the .NET Framework. Therefore, everything you can do in VB.NET you can do in C#, and vice-a-versa.

The differences occur in three main areas: syntax, object-oriented principles, and the Visual Studio .NET IDE. Syntax concerns the statements and language elements. Object Oriented differences are less obvious, and concern differences in implementation and feature sets between the two languages. IDE differences include things like compiler settings or attributes. There is also a fourth area of difference: language features that are present in one language but have no equivalent in the other.

If you are more familiar with Java, JScript, or C/C++, you may find C#'s syntax more familiar than VB.NET's.

If you've been doing VB for the past five years, there's no reason to think you have to now switch to a new language (although you should always look to be learning new things).

The fact is that VB.NET has all the facilities of VB such as not being case-sensitive, having the option of using IntelliSense etc in addition to which you have an ocean of new ideas & concepts thrown in for the benefit of programmers.

VB has matured as a language and if you do not know it already, its almost 11 years since VB was born. It now provides all facilities for distributed computing and Internet programming for which it was not useful earlier. With VB.NET, due to the .NET framework all the classes and all the namespaces available with the other languages are made available to VB also. This is in addition to the drag and drop facility of building forms and web pages which was always the attraction in the use of VB. Thus it has the dual advantage of ease of use and the availability of advanced features.

As already stated earlier both VB.NET and C# are equally powerful. So the primary reason for which VB could not reach the zenith of popularity has been eradicated and programmers who have been waiting for OOPs concepts to be incorporated into VB are rewarded with this offering of Microsoft.

5.2 Language Fundamentals in VB.NET

Constants &Variables

A variable is a named memory location. They are programming elements that can change during program execution. Data that needs to be stored in memory & accessed at a later time are stored in variables. Instead of referring to the memory location by the actual memory address you refer to it with a variable name.

Variables are declared as follows

```
        Dim a as Integer
```

They can also be initialized at the time of declaration as follows:

```
        Dim a as Integer = 10
```

Constants are very similar to variables. The main difference is that the value contained in memory cannot be changed once the constant is declared. When you declare a constant its value is also specified and this value cannot be changed during program execution.

Constants are used in situations where we need to keep the value in some memory location constant. If you use hard-coded values, and the value is changed then it has to be changed in all the locations in the code where it has been used. Instead if we are using constants, all we will need to do is to change the value of the constant. This would propagate the changes to our entire application.

Constants are declared as follows

```
        Const x as Integer
```

VB.NET supports block-level scoping of variables. That is you can declare and use variables as and when you need them. Thus, if a variable is required within a ‘for’ block it can be declared within the block and its scope will be the end of the block.

Simple Types (Primitive Data types)

Simple or value type variables are those, which are assigned space in the stack instead of the heap. All the primitive types such as int, double etc are value type variables. The simple types basically consist of Boolean and Numeric types, where Numeric is further divided into Integral and Floating Point.

The first rule of value types is that they cannot be null. Anytime you declare a variable of value type, you have allocated the number of bytes associated with that type on the stack and are working directly with that allocated array of bits. In addition, when you pass a variable of value type, you are passing that variable’s value and not a reference to the underlying object.

Object Type

Object type or reference type variables are those, which are allocated storage space in the heap. Reference type objects can be null. When a reference type is allocated under the covers a value is allocated on the heap and a reference to that value is returned. There are basically four reference types: classes, interfaces, delegates and arrays.

Class Type

Custom data types are available in .NET framework in the form of classes or class type. It is nothing but a set of data and related behavior that is defined by the developer.

Object type and class type are both reference type variables. The only difference comes from the fact that object type consists of objects predefined and available with the .NET framework such as string whereas class type consists of custom user defined data types such as the Class Employee given below.

```
Class Employee
    Dim empid As Integer
    Dim empname As String
    Public Sub New()
        empid = 10
        empname = "Reshmi"
    End Sub
End Class
```

Overloading and Overriding of the Class

Overloading provides the ability to create multiple methods or properties with the same name, but with different parameters lists. This is a feature of polymorphism. It is accomplished by using the **Overloads** keyword in VB.NET. A simple example would be an addition function, which will add the numbers if two integer parameters are passed to it and concatenate the strings if two strings are passed to it.

```
Class test
      Public Overloads Function Add(ByVal x As Integer, ByVal y As
Integer)
            Return x + y
      End Function

      Public Overloads Function Add(ByVal x As String, ByVal y As
String)
            Return x & y
      End Function

        Shared Sub main()
            Dim a As new test
            Dim b As Integer
            Dim c As String
            b = a.Add(1, 2)
            c = a.Add("Reshmi", " Nair")
            System.Console.Writeline(b)
            System.Console.Writeline(c)

      End Sub
End Class

O/P:
3
Reshmi Nair
```

Overriding

Class inheritance causes the methods and properties present in the base class also to be derived into the derived class. There might arise a situation wherein you would like to change the functionality of an inherited method or property. In such cases we can override the method or property of the base class. This is another feature of polymorphism. You can accomplish this in VB.NET by using the **Overridable** keyword with the base class method and the **Overrides** keyword with the derived class method.

```
Public Class shapes
        Public Overridable Sub display()
            Console.WriteLine("Shapes")
        End Sub

    End Class

    Public Class square
        Inherits shapes

        Public Overrides Sub display()
```

```
            Console.WriteLine("This is a square")

        End Sub
    End Class
    Public Class rectangle
        Inherits shapes

        Public Overrides Sub display()
            Console.WriteLine("This is a rectangle")
        End Sub
    End Class
```

The above example is just an indication to how overriding can be implemented in either VB.NET.

Properties

Properties are named members of classes, structs, and interfaces. They provide a flexible mechanism to read, write, or compute the values of private fields through accessors.

Properties are an extension of fields and are accessed using the same syntax. Unlike fields, properties do not designate storage locations. Instead, properties have accessors that read, write, or compute their values.

Get accessor
The execution of the **get** accessor is equivalent to reading the value of the field.

The following is a **get** accessor that returns the value of a private field name:

```
Dim name as String ’ the name field

Property Name() As String ’ the name property

      Get

            Return name

      End Get

End Property


Set accessor
```

The **set** accessor is similar to a method that returns **void**. It uses an implicit parameter called **value**, whose type is the type of the property. In the following example, a **set** accessor is added to the Name property:

```
Dim name as String ’ the name field

Property Name() As String ’ the name property

      Get

            Return name
      End Get
      Set(ByVal Value As String)
            Name = value
      End Set
End Property
```

When you assign a value to the property, the **set** accessor is invoked with an argument that provides the new value. **For example:**

e1.Name = "Reshmi"   // The set accessor is invoked here

It is an error to use the implicit parameter name (**value**) for a local variable declaration in a **set** accessor.

How to make a Property Read Only/Write Only

There are times when we may want a property to be read-only – such that it can't be changed. This is where read-only properties come into the picture. A Read Only property is one which includes only the get accessor, no set accessor.

For instance,

```
Public ReadOnly Property EmpID() as Integer
      Get
            Return empid
      End Get
End Property
```

Similar to read-only properties there are also situations where we would need something known as ***write-only*** properties. In this case the value can be changed but not retrieved. To create a write-only property, use the WriteOnly keyword and only implement the set block in the code as shown in the example below.

```
Public WriteOnly Property e as string
      Set
            e = Value
      End Set
End Property
```

Structures :

A structure allows you to create your own custom data types and it contains one or more members that can be of different data types. It can contain fields, methods, Etc., Structures are very similar to classes but there are some restrictions present in the case of structures that are absent in the case of classes. For example you cannot initialize structure members. Also you cannot inherit a structure whereas classes can be inherited. Another important feature of structures differentiating it from classes is that a structure can't have a default parameter-less constructor or a destructor. A structure is created on the stack and dies when you reach the closing brace in C# or the End structure in VB.NET.

But one of the most important differences between structures and classes is that structures are referenced by value and classes by reference. As a value type, allocated on the stack, structs provide a significant opportunity to increase program efficiency. Objects on the stack are faster to allocate and de-allocate. A struct is a good choice for data-bound objects, which don't require too much memory. The memory requirements should be considered based on the fact that the size of memory available on the stack is limited than the memory available on the heap.
Thus we must use classes in situations where large objects with lots of logic are required.

Struct – Code: Sample code showing the Class vs. Structures

```
Imports System

Class Test
      Dim classvar As Integer
      Dim anothervar As Integer = 20
      Sub New()
            classvar = 28
      End Sub

      Structure ExampleStruct
            Dim i As Integer
            Sub New(ByVal j As Integer)
                  i = j
            End Sub

            Sub trialMethod()
                  Console.WriteLine("Inside Trial Method")
            End Sub

      End Structure

      Shared Sub main()
            Dim t As New Test()
            Dim strct As New ExampleStruct(20)
            Console.WriteLine(strct.i)
            strct.i = 10
            Console.WriteLine(t.classvar)
            Console.WriteLine(strct.i)
            strct.trialMethod()
      End Sub

End Class
```

```
O/P: -
      28
      20
      10
             Inside Trial Method
```

In the above example, I have declared and used a constructor with a single parameter for a structure. Instead if I had tried to use a default parameter-less parameter I would have got an error. But the same is possible in the case of classes as shown by the default parameter-less constructor, which initializes the classvar variable to 28.
Another point to note is that a variable called anothervar has been declared and initialized within the class whereas the same cannot be done for members of a structure.

Why Namespaces

Namespaces are used in .Net to organize class libraries into a hierarchical structure and reduce conflicts between various identifiers in a program. By helping organize classes, namespaces help programmers manage their projects efficiently and in a meaningful way that is understood by consumers of the class library. Namespaces enables reusable components from different companies to be used in the same program without the worry of ambiguity caused by multiple instances of the same identifier.

Namespaces provide a logical organization for programs to exist. Starting with a top-level namespace, sub-namespaces are created to further categorize code, based upon its purpose.

In .Net, the base class library begins at the **System** namespace. There are several classes at the **System** level such as **Console**, **Exception** etc. The namespace name gives a good idea of the types of classes that are contained within the namespace. The fully qualified name of a class is the class name prefixed with the namespace name. There are also several nested namespaces within the **System** namespace such as **System.Security**, **System.IO**, **System.Data, System.Collections** etc.

Reducing conflict is the greatest strength of namespaces. Class and method names often collide when using multiple libraries. This risk increases as programs get larger and include more third-party tools.

Boxing Conversions

Boxing is the implicit conversion of a value type to a reference type or to any interface type implemented by this value type. This is possible due to the principle of type system unification where everything is an object.

When boxing occurs, the contents of value type are copied from the stack into the memory allocated on the managed heap. The new reference type created contains a copy of the value type and can be used by other types that expect an object reference. The value contained in the value type and the created reference types are not associated in any way. If you change the original value type, the reference type is not affected. Boxing, thus, enables everything to appear to be an object, thereby avoiding the overhead required if everything actually were an object.

Example:

VB.NET

```
Dim n as Integer = 10
Dim obj as Object
obj = n
```

Explanation:
In the above code segment, a **value-type** variable **n** is declared and is assigned the value **10**. The next statement declares an **object-type** variable **obj**. The last statement implicitly performs **boxing** operation on the variable **n**.

UnBoxing Conversions

UnBoxing is the explicit conversion from a reference type to a value type or from an interface type to a value type that implements the interface.

When unboxing occurs, memory is copied from the managed heap to the stack. For an unboxing conversion to a given value type to succeed at run time, the value of the source argument must be a reference to an object that was previously created by boxing a value of that value type otherwise an exception is thrown.

VB.Net does not support the ability to explicitly unbox values. It relies on the helper functions in the **Microsoft.VisualBasic.Helpers** namespace to carry out unboxing. Since these helper functions are considerably less efficient than C# support for explicit unboxing. Thus it is recommended to avoid excessive use of variables of type **Object.**

Boxing and UnBoxing have performance implications. Every time a value type is boxed, a new reference type is created and the value type is copied onto the managed heap. Depending on the size of the value type and the number of times value types are boxed and unboxed, the CLR can spend a lot of CPU cycles just doing these conversions.

It is recommended to perform boxing and unboxing in a scenario where you have to pass a value parameter multiple times to a method that accepts a reference parameter. In such a case, it is advantageous to box the value parameter once before passing it multiple times to methods that accept reference methods.

Enumerations

Enumerations are types that inherit from **System.Enum.** The elements of an enumeration are expressed in words rather than numbers, which makes it convenient for understanding the meaning of the value being used. Enumerations symbolically represent a set of values of one of the primitive integral types.

The type of the elements of an enumeration can be **Byte, Short, Integer or Long**. If no type is specified explicitly, the default type is **Integer**.

Example:

```
Enum month As Byte
        Jan = 2
        Feb = 5
        Mar = 10
End Enum
```

Explanation:
In the above code segment, an enumeration type **month** is declared. The underlying type of the elements has been specified as **Byte**. It has three elements viz: **Jan, Feb and Mar.** These three elements have been assigned specific values. In case of an enumeration, if no values are specified, the value of the first element corresponds to 0 and so on.

<u>Delegates</u>

The runtime supports constructs called delegates, which enable late-bound operations such as method invocation and callback procedures. With delegates, a program can dynamically call different methods at runtime. They are type safe, secure, managed objects that always point to a valid object and cannot corrupt the memory of another object. The closest equivalent of a delegate in other languages is a function pointer, but whereas a function pointer can only reference **Shared** functions, a delegate can reference both **Shared** and instance methods. Delegates are **Marshal by Value** Objects.

The members of a delegate are the members inherited from class **System.Delegate**.
A delegate defines the signature and return type of a method. The resulting delegate can reference any method with a matching signature. Each instance of a delegate can forward a call to one or more methods that take those parameters and return the same type. Once a method has been assigned to a delegate, it is called when the delegate is invoked.

Example:

```
Module delegate_example

        Delegate Function calculation(ByVal a As Integer, ByVal b As Integer) As
Integer

        Public Function add(ByVal num1 As Integer, ByVal num2 As Integer) As Integer
                add = num1 + num2
```

```
    End Function

    Sub Main()
        Dim calc_delegate As New calculation(AddressOf add)
        Dim result As Integer
        result = calc_delegate(50, 70)
    End Sub

End Module
```

Explanation:
Four steps are required to implement delegates viz.

- **Defining Delegates**
  The foremost step is to define the delegate. The definition of the delegate specifies the method signature, return type of the method, access modifier and the delegate name. The method signature specifies the order and type of each argument.

  The definition of a delegate is indicated by the usage of the **Delegate** keyword. As shown in the above code segment, the delegate name is **calculation,** it's access modifier is **public,** it receives two integer arguments and returns an integer value.

- **Creating Delegate Method Handler(s)**
  The next step is to define the method(s) that will be associated with the delegate.
  In the above code segment, a method named **add** is defined. This method must have same method signature as that of the delegate, as shown in the above code segment.

- **Hooking up Delegates and Method Handlers**
  For a delegate method handler to be invoked, it must be assigned to a delegate object. In the above code, the delegate object is **calc_delegate** and is hooked up to the method handler **add**.

- **Invoking the method through the Delegate**
  The last step is to invoke the methods that are associated with the delegate. A delegate method handler is invoked by making a method call on the delegate itself. This causes the method handler to invoke with the assigned input parameters as if they were invoked directly by the program, as shown in the above code.

5.3 Features of VB.NET

Option Explicit and Option Strict

Option Explicit and Option Strict are compiler options that can be globally assigned to a project and are interpreted at compile time. Setting these options enables programmers to resolve some of the errors (e.g. typological errors) at compile time and thus prevent runtime errors.

Option Explicit

Option Explicit was a feature of VB 6.0 and it has been made a part of .NET environment too. This option can be used only at the module level. When this option is turned on, it forces explicit declaration of variables in that module. This option can be turned "On" or "Off". When it is not specified, by default, it is set to "Off".

Syntax: Option Explicit [On / Off]

When it is set to "On", it checks for any undeclared variables in the module at compile time. If any undeclared variable is found, it generates a compile time error since the compiler would not recognize the type of the undeclared variable. When it is set to "On", variables can be declared using Dim, Public, Private or ReDim statements. Setting this option to "On" helps programmers do away with any typological errors in the code.

When it is set to "Off", all undeclared variables are considered to be of type Object.
It is preferable to set this option to "On".

Option Strict

Visual Basic language in general does not require explicit syntax to be used when performing operations that might not be optimally efficient (e.g. late binding) or that might fail at run time (e.g. narrowing conversions). This permissive semantics often prevents detection of coding errors and also affects the performance of the application.

VB.NET enables a programmer to enforce strict semantics by setting this option to "On". When used, this option should appear before any other code. This option can be set to "On" or "Off". If this statement is not specified, by default, it is set to "Off".

Syntax: Option Strict [On / Off]

When it is set to "On", it disallows any narrowing conversions to occur without an explicit cast operator, late binding and does not let the programmer omit "As" clause in the declaration statement. Since setting it to "On" requires explicit conversion, it also requires that the compiler be able to determine the type of each variable. Thus it is implied that Option Strict also means Option Explicit.

Visual Basic .NET allows implicit conversions of any data type to any other data type. However, data loss can occur if the value of one data type is converted to a data type with less precision or a smaller capacity. Setting this option to "On" ensures compile-time notification of these types of conversions so they may be avoided.

Explicit conversions happen faster than the implicit conversions performed by the system. In case of implicit conversions, the system has to identify the types involved in the conversion and then obtain the correct handler to perform the conversion. In case of explicit conversions, processing time gets reduced since the type of conversion is explicitly mentioned.

From programmer point of view, explicit conversion may seem to be a burden since it slows the development of a program by forcing the programmer to explicitly define each conversion that needs to occur.

It is always recommended that you set Option Strict to "On" and use explicit conversions.

ByVal is Default

When implementing functions/procedures we often need to pass information. This information can be passed to the called function/procedure through parameters. Parameters can be passed by value or by reference.

When ByVal keyword is used, it causes the parameter to be passed by value. In this case, a copy of the original parameter is passed to the called module. Thus any changes made to the copy of the parameter do not affect the original value of the parameter.

When ByRef keyword is used, it sends a reference (pointer) to the original value to the called module rather than its copy. Thus any changes made to the parameter in the function/procedure will cause the original value to be modified.

With ByRef you are exposing a variable to modification which can lead to an unexpected behavior. If that procedure calls another procedure and passes the same parameter **ByRef**, the chances of unintentionally changing the original variable are increased.

In VB.NET, every parameter, by default, is passed by value if nothing is explicitly specified. Thus it protects arguments against modification.

Example:

```
Public Class SampleClass
      Sub Proc(ByVal a As Integer, b As Integer, ByRef c As Integer)
            a = a + 2
            b = b + 3
            c = c + 4
      End Sub
End Class
Sub Main()
      Dim num1, num2, num3 As Integer
      Dim Obj As New SampleClass()
      num1 = 2
      num2 = 3
      num3 = 4
      Obj.Proc(num1, num2, num3)
      System.Console.WriteLine(num1)
      System.Console.WriteLine(num2)
      System.Console.WriteLine(num3)
End Sub
```

Explanation:
In the class **SampleClass,** there is one procedure **Proc** that takes three integers as arguments. In **Main(),** a variable of type S**ampleClass** is defined and three other integer variables **num1,num2,num3** are declared and given some initial values. The procedure is then called passing these three variables as arguments. First variable **num1** is passed **by value,** second variable **num2** is also passed **by value** since nothing is specified explicitly and the last variable **num3** is passed by reference since it is explicitly mentioned.

After calling the procedure, when the values of the three variables are checked, you will notice that **num1** and **num2** retain their original values since they are passed by value whereas the value of **num3** changes to 8 since it was passed by reference.

Sub and Functions with Parenthesis

In VB.NET, both functions and procedures require parentheses around the parameter list, even if it is empty.

Example:
Consider the following code that contains two functions and one procedure. The first function named "HelloWorld" accepts no argument and returns "Hello World". The second function named "HelloName" accepts an argument and appends that argument to the string "Hello World" and finally returns the appended string. The procedure named "Hello" displays the string "Hello World" on the console.

```
Public Class SampleClass

        Function HelloWorld( ) As String
            Helloworld = "Hello  World"
        End Function

        Function HelloName(ByVal name As String) As String
            HelloName = "Hello " & name
        End Function
        Sub Hello()
            System.Console.WriteLine("Hello  World")
        End Sub

End Class

Sub Main()

     Dim Obj As New SampleClass()
     Dim str1 As String
     Dim str2 As String

     str1 = Obj.HelloWorld()
     str2 = Obj.HelloName(" Everybody")
     Obj.Hello()

End Sub
```

Explanation:
As mentioned, while calling the procedure **Hello,** parentheses have to be used even though it does not require any argument. Similar treatment has to done for functions.

Structures Replace User Defined Types

In VB.NET, structures can be defined using **Structure** keyword. Unlike user defined types, structures share many features with classes. Explicit mention of access modifier for each member is necessary in VB.NET. Access modifiers can be **Public**, **Protected**, **Friend**, **Protected Friend**, **or Private**. You can also use the **Dim** statement, which defaults to public access.

Let us take an example of a Person that will have some characteristics as Name, Address, Tel, Age etc. Now if we were to use this number of times, it will be better to have a type called Person. The example below shows how to handle this.

Structure Person

```
Structure Person
        Dim Name as String
        Private Address as String
        Dim Tel as String
        Public Age as Integer
End Structure
```

As shown in the above code, explicit mention of the access specifier for each member is a must.
Structures, like classes, can contain data members as well as data methods. Structures are value types and are hence allocated on the stack where as classes are reference types and require heap allocation.

Block Level Scope

The scope of block level variables is restricted to the block in which they are defined. This block can be a function, procedure, a loop structure etc. A block level variable is not accessible anywhere, outside its block.

Example:

Consider the sample code segment.

```
Sub BlockScope()
        Dim a As Integer = 1
        Dim i As Integer = 1
        While (i <= 2)
                Dim j As Integer = 1
                j = j + 1
                i = i + 1
        End While
a = a + j
End Sub
```

Explanation:

The above code segment will give a compile-time error, since the scope of the variable 'j' is restricted to the **while block** and hence it is not accessible outside the **while block**. Though block level variables cannot be accessed outside their block, their lifetime is still that of the procedure containing them.

Early Vs Late Binding

In case of early binding, the compiler knows the object's data type at compile time and hence can directly compile code to invoke the methods on the object. This enables the compiler to discover the appropriate method and ensure that the referenced method does exist and the parameters provided are in sync with that of the referenced method.

Since object types are known ahead of time, the IDE aids the programmer by providing support for IntelliSense. This helps the programmer do away with any typological and syntactical errors. If the method being referenced is not found, the programmer is notified at compile time thus letting him rectify it rather than making him wait till the application is run.

In case of late binding, the compiler cannot determine the object's data type and thus the code interacts with the object dynamically at runtime. To make an object late-bound, it is defined as a variable of type **Object.** This variable can reference any type of object and allows programmers to attempt arbitrary method calls against the object even though the **Object** datatype does not implement those methods. Since type of the object is not known until runtime, neither compile-time syntax checking nor IntelliSense is possible. The typological errors also go undetected until the application is run. However, there is an unprecedented flexibility, since code that makes use of late binding can talk to any object from any class as long as those objects implement the methods we require.

The discovery of the referenced method is done dynamically done at runtime and is then invoked. This discovery takes time and the mechanism used to invoke a method through late binding is not as efficient as that used to call a method that is known at compile time. Thus, though late binding is flexible, it is error-prone and slower as compared to early binding.

When Option Strict is "On", it does not support late binding since the datatype of the object should be known at compile time.

Example:

```
Consider the below given code segment

Public Class SampleClass

    Public Sub Proc()
        MessageBox("This method is discovered dynamically at runtime")
    End Sub

End Class
Public Class MainClass
   Shared Sub Main()
      Dim Obj As Object
      Obj = New SampleClass()
      Obj.Proc()
   End Sub

End Class
```

Explanation:

The above code segment implements late-binding. There are two classes viz. **SampleClass** and **MainClass**. The class **SampleClass** contains a procedure named **Proc** that takes no arguments. This procedure displays an appropriate message in a message box. The class **MainClass** declares a variable **Obj** of type **Object.** Thus at compile-time, the actual data type of the variable is not known. At runtime, the system finds that it is referencing a variable of type **Sampleclass.** This reference to the class is achieved using **new** operator. The referenced method **Proc** is discovered at runtime and is finally invoked.

Ctype Function

Ctype is a general cast keyword that coerces an expression into any type. It returns the result of explicitly converting an expression to a specified data type, object, structure, class, or interface. If no conversion exists from the type of the expression to the specified type, a compile-time error occurs. When **Ctype** is used to perform explicit conversion, execution is faster since it is compiled inline and hence no call to a procedure is involved to perform the conversion.

Syntax: Ctype (expression, type name)

Example:

```
Consider the below given code segment

Public Class TrialClass
   Sub Proc (obj as Object)
      Dim Obj1 As OtherClass( )
      Obj1 =  Ctype (obj, OtherClass)
      Obj1.OtherProc( )
   End Sub
End Class
```

Explanation:

The variable **Obj1** is of type **OtherClass.** This class has a procedure named **OtherProc.** In the above code segment, the procedure **Proc** takes one parameter of type **Object.** The Ctype statement gains an early bound reference to the object of type **OtherClass.** Thus performance benefits of early binding can be achieved.

Changes to Boolean Operators

In Visual Basic.NET, **And, Or, Xor, and Not** are the boolean operators and **BitAnd, BitOr, BitXor, and BitNot** are used for bitwise operations.

VB.NET has introduced the concept of **short-circuiting.** According to this concept, if the first operand of an **And** operator evaluates to **False**, the remainder of the logical expression is not evaluated. Similarly, if the first operand of an **Or** operator evaluates to **True**, the remainder of the logical expression is not evaluated.

To perform the above mentioned functionality, VB.NET has introduced two new operators viz. **AndAlso and OrElse. AndAlso** performs short-circuiting logical conjunction on two expressions whereas **OrElse** performs short-circuiting logical disjunction on two expressions

```
Syntax : expression1 AndAlso expression2
         expression1 OrElse expression2
```

Structured Error Handling

The whole idea behind error handling is to accurately trap the error. VB.NET has introduced a structured approach for handling errors, in order to keep in sync with the features offered by all Object Oriented languages viz. C#, Java etc. This structured approach is implemented using a **Try…Catch…Finally** block structure and is known as **Exception Handling**.

Try statement comes before the block of code that needs to tested for errors, **Catch** statement handles specific errors and hence surrounds the block of code that handles those errors and **Finally** block of code is always executed and contains cleanup routines for exception situations. Since **Catch** block is specific to the type of error that needs to be caught, a single **Try** statement can have multiple **Catch** blocks associated with it.

Example:

```
Consider the below given code segment

Dim num1 As Integer
Dim num2 As Integer
Dim num3 As Integer
Dim str As String

num1 = CType(System.Console.ReadLine( ), Integer)
num2 = CType(System.Console.ReadLine( ), Integer)
str = System.Console.ReadLine( )

Try
    num3 = num1 / num2
    num1 = CType(str, Integer)

Catch e1 As System.InvalidCastException
 System.Console.WriteLine("There is a casting error")

Catch e2 As System.OverflowException
 System.Console.WriteLine("There is an overflow error")

Finally
     System.Console.Writeline("Please enter valid input")
End Try
```

Explanation:

The above code segment has been written with an intention of creating an error to explain the structured way of trapping errors. As shown, it accepts two integers and one string from the user and tries to divide first number by the second number and also tries to convert the entered string to an integer. Since the statements that divide the numbers and perform casting are critical, they are put in the **Try…Catch…Finally** block structure.

The code segment has two **Catch** statements viz. first statement handles the casting (wrong format) error while the second one handles the overflow (division) error.
If the user will enter zero as the second number, then the code will throw an overflow exception which will be handled by displaying the error message on the Console. Similarly, if the user will enter a string say "Hello" as the third argument, the code will throw an invalidcast exception which will be handled by displaying the error message on the Console. In both the cases, the **Finally** block of code is always executed. **Finally** block is even executed, when there is no error in the program. Thus it is the right place to perform cleanup routines.

This structured approach provided by VB.NET lets the programmer track the precise location of the error in the code.

<u>Data Type Changes</u>

Integer

Integer Type	*VB.NET*	*CLR Type*
8-bit Integer	*Byte*	*System.Byte*
16-bit Integer	*Short*	*System.Int16*
32-bit Integer	*Integer*	*System.Int32*
64-bit Integer	*Long*	*System.Int64*

Boolean

A Boolean variable can be assigned one of the two states viz. **True** or **False**.
In VB.NET, when numeric types are converted to **Boolean** values, 0 becomes **False** and all other values become **True**.
When Boolean values are converted to **Integer** values, **True** maps to -1 and **False** maps to 0.

String
To be inline with other .NET languages, VB.NET has updated string length declaration. In VB.NET, you cannot declare a string to have a fixed length. You must declare the string without a length. When a value gets assigned to the string, the length of the value determines the length of the string.

5.4 Control Statements

VB.Net has statements such as If .. Then ..else and Select …. Case, which help you to conditionally execute program.

VB.Net provides you with various looping statements, such as Do… Loop, While…. End While, and For… Next.

1. **The If….Then….Else…End if Statement**

Consider a student marks and grade evaluation. For Marks above 75 the grade is 'A' and for below 75 is 'B'. In this situation when you need to execute some code based on some condition, you can make use of, **If…then…else…end if.**

The Cultural *syntax* normally used is as follows:

If *condition* **Then**
Executable statements when the condition is True
Else

```
        Executable statements when the Condition is False
    End If
```

OR using Elseif for Advanced Decision making

```
    If condition Then
        Executable statements
    ElseIf condition Then
        Executable statements
    End If
```

Single if can have multiple else with conditions, as mentioned above in *elseif* format and finally a single **End If** for the main **If** condition.

Nesting IF…Then Constructs

```
If condition Then
     If condition2 Then
         Executable statements when the condition2 is TRUE
     Else
         Executable Statements
    End if
Else
    Executable statements
End If
```

One important thing to keep in mind when nesting IF…Then constructs is that you must have corresponding End If statement for every IF ..Then statement, unless the If then statement executes only one statement and that statement appears on the same line as If…Then

2. The Select…Case Statement(Evaluating an Expression for Multiple Values)

The **Select…Case** Statement is similar to **If…Else…End if**. The only difference between two is that **If** and **elseif** can evaluate different expressions in each statement, but the **Select** statement can evaluate only one expression.

The drawback of IF...Then construct is that it isn't capable of handling a decision situation without a lot of extra work. One such situation is when you have to perform different actions based on numerous possible values of an expression, not just True or False. For instance performing actions based on Students Marks.

```
If  intmarks  >35 Then
    …….
Elseif intmarks >50 then
    ……
```

```
Elseif intmarks>65 then
        .....
Elseif intmarks>75 then
        ...
Else
        ....
End If
```

As you see the structure can be a bit hard to read and if the conditions increase you may end up writing a confusing and an unreadable piece of Code

The **Select** uses the result of an expression to execute different set of statements.
The *syntax* for the **Select…Case** Statement is as follows:

```
        Select Case [expression]

                Case [expression list]
                        Executable statements.
                Case Else
                        Executable statements.
                        Exit Select  - to exit from the select
        End Select
```

Note: Case Else is used to define the code that executes only when the expression doesn't evaluate to any of the values in Case Statements .Use of Case Else is optional

Lets see the same example as above but this time with Select Case

```
Select Case intmarks
      Case Is >35
            Executable statements
      Case Is >50
            Executable statements
      Case Is>65
            Executable statements
      Case Is >75
            Executable statements
      Case Else
            Executable statements
End Select
```

Evaluating More than one possible Value in a Case Statement

Select Case helps you to use some more advanced expression comparisons. Like,you can specify multiple comparisons in a Single Case statement by just using comma. Lets see how it does

```
Select Case strColor
```

```
Case Is=”Red”,”Blue”,”Magenta”
      ‘Color is a Dark Shade
Case Is =”Cream”,”white”
      ‘Color is a Cool Shade
End Select
```

Another comparison expression used is keyword ***To***, Visual Basic.NET evaluates the expression and finds out whether it is in the range mentioned and if yes the Statement is executed. Please note that when using *To*, ***you can’t include Is = as you can with the simple expression***

```
Select Case intmarks
Case 1 to 35
      ‘Executable statements
Case 36 to 50
      ‘Executable Statements
End Select
```

3. For…Next Statement

The **For…Next** Statements are used repeat a set of statements for specific number of times.

The *syntax* for the **For…Next** Statements is as follows:

```
For counter = <start value> to <end value> [Step Value]
      Executable Statements
      Exit For
Next [counter]
```

Counter is any numeric value.

Start value is the initial value of the counter.

End value is the final value of the counter.

Step Value is the value by which the counter is incremented. It can be positive or negative. The default value is 1.

Exit For is used to exit the **For…Next** loop at any time. When *Exit for* is encountered ,the execution jumps to the statement following Next

Next is the statement the marks the end of the **For** statement. As soon as the program encounters the **Next** statement, the step value is added to the counter and the next iteration of the loop takes place.

```
Dim intctr as Integer
```

```
For intctr=1 to 100
      Debug.WriteLine(intctr)
Next intctr
```

This routine starts a loop with a For statement after a variable intctr is declared. This loop initializes intctr to 1 and then prints 1 through 100 to the output window. It prints in steps of 1 as Step has been omitted here, so the default is 1

Example of use of STEP in For....Loop.

Let us write a table of 2 using *step* in for loop.
Add a label with name it as **lbtables** and make it bit bigger on the screen.

```
Dim j = 1
For i = 2 To 20 Step 2
Me.lbtables.Text = Me.lbtables.Text & "2 X " & j.ToString & " = " &
                    i.ToString & vbCrLf
j = j + 1
Next
```

Output:

```
2 X 1 = 2
..
..
..
..
..
...
2 X 10 = 20
```

An Example of Nested For loop.

Let us write a small code to display a structure of stars '' in triangle format.*

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

Let us have a label with name **stars**. Increase the height of the label to get a clear view of the image.

```
Dim star As String
Dim i, j As Integer

For i = 0 To 5 ' First loop to count the rows
```

```
   For j = 0 To i ' Second loop to count the columns
      star = star & " * "
   Next
Me.stars.Text = Me.stars.Text & star & vbCrLf ' To print *
star = ""
Next
```

4. For Each…Next Statement

The **For Each…Next** Statement is used to repeat a set of statements for each element in an array or collection.
The **For Each…Next** statement is executed if there is at least one item in an array of collection.
The Loop repeats of each element in an array or collection.

The *syntax* for the **For Each…Next** statement as follows:

For Each *Component* In *Set*
Executable statements
Next

Component is the variable used to refer to the elements of an array or a collection.
Set refers to an array or any collection object. e.g.

```
Dim weeks() As String = {"Monday", "Tuesday", "Wednesday", "Thursday",_
                         "Friday", "Saturday", "Sunday"}
        Dim eachday As String
        For Each eachday In weeks
            MsgBox(eachday)
        Next
```

An example for using for each element in a collection of string into a single string element.
Each element of array which is of type string is read from the collection and stored into the single string type object.

5. While…End Statement

The **While…End** Statement is used to repeat set of executable statements as long as the condition is true.
The *syntax* for the **While…End** statement is as follows:

While *Condition*
Executable Statements

End While

In this if the condition is satisfied then the statements are executed. Else it will not enter the **While** condition at all.

6. Do...Loop Statement

The **Do…Loop** Statement is similar to **While…End.** Here we have two types of formatting the loop.

a) **Do While / Until** *Condition* Executable Statements **Loop**
b) **Do** Executable Statements **Loop While/Until** *Condition*

The Difference is in a) The loop will be executed if the condition is satisfied, but in b) The Loop will be executed at least once even if the condition does not satisfy.

```
Do While Expression
        [Statements]
Loop
```

```
Do Until Expression
        [Statements]
Loop
```

Note: For VB programmers While Wend is not supported it is ***While… End*** *now*

A Complete Example with set of control statements.

We will create a VB.Net application, which will accept students name and its grade. Depending up the type of grade it will add remarks.

```
   txtsummary.Text = ""

        Dim value, ctr As Integer

        'Accept a number from the user
        value = CInt(InputBox("Enter the number of students"))

        'Check if the validity of the number
        If value <= 0 Then
            MsgBox("Enter details of at least one student", "Error")
        End If

        Dim arrName(value) As String
        Dim sGrade As String
        Dim arrRemarks(value) As String

        While ctr < value

            'Accept the name of the students
               arrName(ctr) = InputBox("Enter the name of the Student"_
& ctr + 1, "Enter Details")
            'Accept the grade of the Student
               sGrade = InputBox("Enter the grade of the student" &
"(_A/B/C/D/F)", "Grade Details")

            ' Assign remarks to  students
            Select Case UCase(sGrade)
                Case "A"
                    arrRemarks(ctr) = "Excellent"
                Case "B"
                    arrRemarks(ctr) = "Good"
                Case "C"
                    arrRemarks(ctr) = "Fair"
                Case "D"
                    arrRemarks(ctr) = "Poor"
                Case "F"
                    arrRemarks(ctr) = "Fail"
                Case Else
                    MsgBox("Incorrect value entered ", _
MsgBoxStyle.Critical)
                    Exit Sub ' To come out of the program
            End Select
            ctr = ctr + 1
        End While
        ' Display the summary in the text box
        For ctr = 0 To value - 1
           If txtsummary.Text = "" Then
                If LCase(arrRemarks(ctr)) = "fail" Then
                  txtsummary.Text = arrName(ctr) & " has failed _
                        in exams" & vbCrLf
                Else
                   txtsummary.Text = arrName(ctr) & "'s performance is_
" & arrRemarks(ctr) & vbCrLf
                End If
           Else
                If LCase(arrRemarks(ctr)) = "fail" Then
                    txtsummary.Text = txtsummary.Text & arrName(ctr) &_
" has failed in exams" & vbCrLf
                Else
                    txtsummary.Text = txtsummary.Text & arrName(ctr) &_
"'s performance is " & arrRemarks(ctr) &_ vbCrLf
                End If
            End If
        Next
```

5.5 Arrays

Till now we have been using variable to store values. We might come across a situation when we might need to store multiple values of similar type. Such as names of 100 students in a school. One way to do it is to declare 100 variables and store all the names.

A much more simple and efficient way of storing these variable is using *Arrays*. An *Array* is a memory location that is used to store multiple values.

All the values in an array are of the same type, such as Integer or String and are referenced by their index or subscript number, which is the order in which these values are stored in the array. These values are called the elements of the array.
The number of elements that an array contains is called the length of the array.

In VB.Net all arrays are inherited from the *System.Array* Class.

Arrays can be single or multidimensional. You can determine the dimensions of an array by the number of subscripts that are used to identify the position of any array element.

A single dimensional array is identified by only a single subscript and an element in a two-dimensional array is identified by two subscripts.

The dimension has to be declared before using them in a program. The array declaration comprises the name of the array and the number of elements the array can contain.

The *Syntax* of single dimension array is as follows.

Dim ArrayName (number of elements) as Element Data type.

e.g.

```
Dim studentname(10) as string
Or
Dim studentname() as string = new string(10)

You can assign the values at runtime or even at the design time.

Design time declaration:
Studentname(0)=”Rohan”
Studentname(1)=”Mohan”
…..
Studentname(10)=”Nitin”
```

All arrays starts with the index of ***0*** i.e. All arrays are Zero Based and there is no provision of an option Base Statement where in you can specify the Lower Bound . This implies that above array can store 11 elements. Here ***0***, is the starting index or the lower bound of the array. The lower bound is fixed for all the arrays.

Example 1.

We will create a VB.Net application that will accept the names of students in an single dimension array and display it back.

Add a textbox and set the name property to txtnames. Set the multilane property of the text box to true.

Put a button and write the following in the onclick event.

```
txtnames.Text = ""

Dim value, count As Integer
'Accept how many students names to enter
value = CInt(InputBox("Enter the number of students name to enter:"))

Dim arrnames(value) As String
Dim cnt As Integer

For cnt = 0 To value
```

```
arrnames(cnt) = InputBox("Enter the name of student " & cnt + 1 & ":",
"Student Name")
Next

'Display the entered value to the text box

For cnt = 0 To value
    If txtnames.Text = "" Then
         txtnames.Text = arrnames(cnt) & vbCrLf ' for carriage returns
    Else
         txtnames.Text = txtnames.Text & arrnames(cnt) & vbCrLf
    End If
Next
```

Above example will accept number of names to be entered and will add the names in a loop and then redisplay it in a text box.

The *Syntax* for multi-dimension arrays is as follows:
Previously we saw how we can store multiple names of students. But, if we want to store related data of students like first name, middle name, last name. In such situations you can use multi dimension arrays, such as two-or-three dimension arrays.

Dim ArrayName (number of 1st element, number of 2nd element,….) as element data type.

Or

Simpler form would be

Dim ArrayName(number of rows, number of columns) as element data type of two dimension.

e.g.
Dim studentdetails(10,2) as string

Index positions of array elements.

0,0	0,1
1,0	1,1
2,0	2,1
3,0	3,1
…	
10,0	10,1

studentdetails(0,0) = “Manoj”
studentdetails(0,1) = “Malik”

To display “Malik” we need to use the index position of the array and say ,

Studentdetails(0,1).

Example 2.

We will create a VB.Net application, which will accept Student Name, Address and city name and display it in the text box in a formatted way.

Mohan	#2/b,4th lane	Kanpur
Mike	8th block csd	NY
Lim	Chou Lane	Hong Kong

As in the earlier example we will create a text box , change its name and its multi line property to true.

Change the text box to txtsummary.
Add a button and write the below code in that.

```
Dim arrsummary(3, 3) As String
Dim i, j As Integer

'As we wanted just 3 columns we have set it to 2, else if u want to be
two only then while declaring the array make it (2,2) as the lower
index is 0.

For i = 0 To 2
  For j = 0 To 2
       arrsummary(i, j) = InputBox("Enter the value for " & i & " row _
                           and " & j & " column ", "Summary")
   Next
Next




'Display the values in the summary array.

  For i = 0 To 2
      For j = 0 To 2
          If txtsummary.Text = "" Then
             txtsummary.Text = arrsummary(i, j)
          Else
            txtsummary.Text = txtsummary.Text & "-" & arrsummary(i, j)
          End If
       Next
       txtsummary.Text = txtsummary.Text & vbCrLf
  Next
```

Dynamic Arrays.

Till now what we read were about fixed arrays. Let us see how we can manipulate the size of an array at run time.
Many times we feel that the size of array in not enough or too much then required. As we know that array will allocate memory location when its declared, so to release or add more we need to change the dimension of the array which has been pre-declared.

We can create a dynamic array by not specifying the size of the array at the time of array declaration.

Syntax:
Dim student_names() as string

In the above *syntax* you will see that number of elements are not mentioned. Now to re-declare the elements we make use of **ReDim** for an array to change its dimension.

e.g.

```
Dim student_names() as string
ReDim student_names(10)
```

ReDim can also change the size of multi-dimensional arrays.
You can change the number of dimensions in an array, but you cannot make a multi-dimensional array to a single dimension or lesser than the original dimension.

e.g.

```
Dim student_details(10,15) as string 'Declaring the array

ReDim student_details(10,25) 'Resizing the array
```

This statement does not change the data type of the array element or initialize the new values for the array elements.
This statement can be used at the procedure level only not at the class level or module level.
ReDim statements reinitializes the value of arrays with the respective data type declared by the array.
If u have initialized the array to a some values it will be lost during the time of resizing and the default values will be restored in those elements.

E.g.

```
'Declaring the array and initializing the value for the array
dim students_names() as string = {"Rahul"}

'This will display the value Rahul
```

```
msgbox(students  names(0))

Now resizing the array.
ReDim students_names(10)
'This will give a fixed size to 10 elements

msgbox(students_names(0))
```

This will display a blank value, as during the resizing the values of the array are reinitialized to default value of string which is blank.

Now to avoid such problems we will make use of a keyword called **Preserve** while resizing the array to new value.

Using the above example , we will make changes in the declaration.

```
ReDim students_names(10) 'The old declaration
```

```
ReDim Preserve students_names(10)
```

```
Students_names(1) = “Alex”
Students_names(2) = “Michael”
```

```
Msgbox(students_names(0))
```

This will display the value as ‘Rahul’ which we had initialized before resizing it. So **Preserve** will restore all the initialized value of elements declared in an array.
If the array is an two or more dimensional array , you can only change the size of the last dimension by using **preserve** keyword.

```
Dim student_details(10,20) as string
```

```
ReDim preserve student_details(15,25)
```

This will raise error because we are trying to change the first dimension also. So u can only change the last dimension in case of multi-dimensional array.

Few Important methods in arrays.

Ubound() and Lbound().

Ubound is to get the upper limit of an array.
Lbound is to get the lower limit of an array by default lower limit is 0.

e.g.

```
Dim weeks() As String = {"Monday", "Tuesday", "Wednesday", "Thursday",
                         "Friday", "Saturday", "Sunday"}
Dim upper As Integer
Dim lower As Integer

upper = UBound(weeks)
lower = LBound(weeks)

MsgBox(upper.ToString & " - " & lower.ToString)

You will get 6 - 0 as the answer.
```

If you count the number of elements initialized elements its seven, but all arrays starts with lower bound as 0. So from 0 – 6 is equal to 7 elements.

This works for single dimension , but for multi-dimensions,
We make use of the following:

Getupperbound(), getlowerbound() these functions are methods of the array class. You can use it with single dimension also but best for multi-dimensional arrays.

Syntax : arrayname.getupperbound/getlowerbound(dimension)

Dimension refers to the which upper/lower bound should be found, 0 for first, 1 for second and so on.

example.

```
Dim student_details(10,20,15)
Dim upperlimit as integer

upperlimit = student_details.getupperbound(0)
' This will return 10 for the 1st row element
upperlimit = student_details.getupperbound(1)
' This will return 20 for the 2nd row of element
```

For all getlowerbound(dimension) it will return 0 as the base lower bound is zero in .NET

6. Object Oriented Programming Concepts

Section Owner: Akila Manian (MVP)
Content Contributors: Shankar N.S., Swati Panhale

There are various approaches to solve a problem. Moreover, these approaches are to a great extent dependant upon how each one of us tries to analyze and solve the problem.

We have a very simple assignment:

Lets say we want to calculate a distance formula for traveling from one place to another. The distance formula has to include all the attributes of the journey. So let us see what data do we have-

Variables:

Location: Have to go from one city to another (both the cities can be anywhere in the world).
Modes of travel: Car/Bus/Train
Date constraints: Departure/Arrival Date.
Time Preference: Morning/Afternoon /Evening
Distance: Break Journey
Travel Cost. Etc

Now, we have termed these data items as "***variables***" because their value would be changing based upon the various choices made.

Even for such a trivial problem, there are being so many options/constraints, there are so many approaches to arrive at a decision. On similar lines, given a problem and basic resources (which also act as constraints), various algorithms can do the task programmatically. An algorithm is nothing but the thought process/approach involved.

A good approach should

- Be generic so that it works well with all possible combinations of inputs.
- Flexible / adaptable to absorb new inputs. (New destination, routes, rates, timings or even new mode of travel - say space travel)
- Give solutions in the desired timeframe.
- Make best use of resources available. (Optimize the solution)
- Cost effective.
- Simple enough.

Primitively, there was a very straightforward manner to write applications. Straightforward in the sense that the various tasks in the application would be identified and would be automated. This approach did work for many scenarios but when it came to the robustness or maintenance of the application, this approach proved to be insufficient.

In this section we will first discuss what was the procedural approach and how the design of an application be made using this approach. By trying to figure out the negative points in the approach we will then appreciate the advantages of object oriented programming by discussing it as a solution to overcome the shortcomings of procedural programming.

6.1 Concept of Procedural Programming

The whole core of procedural programming lies in deriving a straightforward sequential step-by-step guide to do a particular task. Let us understand this by analyzing a case study.

Lets take up a classic Payroll application that deals with various types of specifications for different employees. Lets say that this application is developed in one of the best procedural languages – C.

So, the “Employee” in the application will be represented by a structure, which will contain all the Employee attributes as data members of the structure. Assume that the application deals with three types of employees – Clerk, Manager and Marketing executive. All the employees do have some common attributes and certain specific allowances (lets not talk of deductions!). The clerk gets medical allowance, the executive gets the traveling allowance and the manager gets house rent allowance and dearness allowance. Our Employee structure might look something like this:

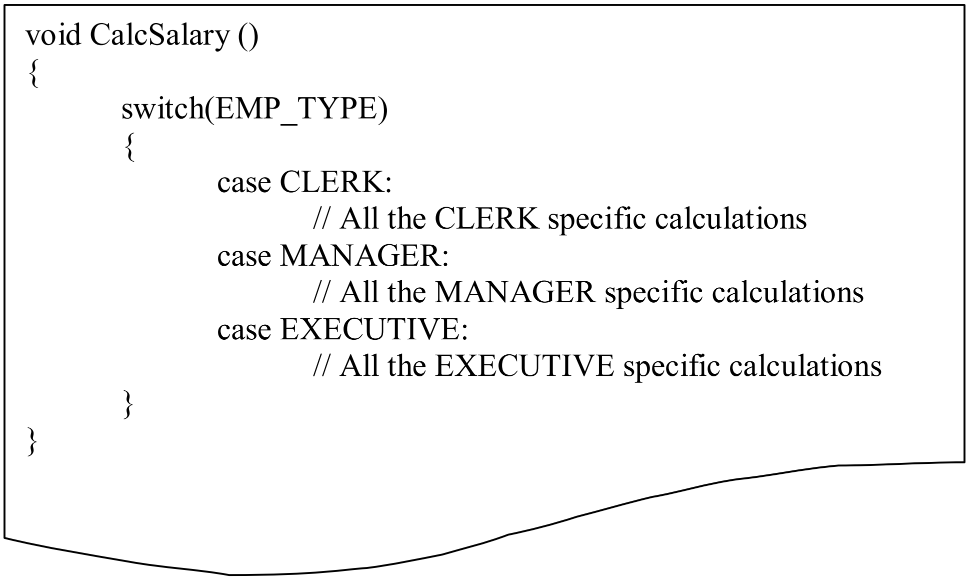
```
struct Employee
{
        int  Id;
        char Name[25];
        char *Address;
        char Designation[20];
        double Basic;

        float Med_Allow;
        float Tra_Allow;
        float HRA;
        float DA;

}
```

The application would have functions, which would act upon the data, and do the necessary functionalities. So there would be at least following functions apart from others:

```
void AddDetails() ;     // To add details of the
                        //  employee to the structure
void PrintDetails() ;   // To print the details of a
                         //  particular Employee

double CalcSalary();    // To calculate the salary of a
                          // employee
```

This also could have been done having three different structures one for each Employee type but this would increase the overhead in programming in the functions and there would be a requirement of declaring three different arrays; one to store all clerk variables, one for manager variables and one for executive. So let us have a common structure, which would suite all the employee types.

If we try to figure out the central algorithm of every function it would be quite monotonous wherein every function would have a strict type inspection routine to check the type of Employee every time. Because the functionalities differ for every type of Employee.



```
void CalcSalary ()
{
        switch(EMP_TYPE)
        {
                case CLERK:
                        // All the CLERK specific calculations
                case MANAGER:
                        // All the MANAGER specific calculations
                case EXECUTIVE:
                        // All the EXECUTIVE specific calculations
        }
}
```

This kind of type inspection will be featuring in every function, which would be dependant upon the type of the Employee.

Now, if we have to add a new Employee type to this application, which has its own specification about, the allowances received try to figure out the changes that we will have to do in the current case study. Not only will the Employee structure have to be modified but also even the functions have to be changed in order to accommodate the new Employee type. So just as we are currently having a case statement to correspond to one employee type, we will have to introduce one more case statement corresponding to the new employee type. In addition, while calculating the salary we might introduce some local variables in the function to do the necessary calculations. There is a probability that the new additions may lead to certain bugs being introduced in the current “working” code. So let us list down the various problems we would face in this application:

Maintenance: -- If the application has to support some change in the existing business logic for a particular employee type there might be more problems introduced since it is the same function that would be called for the different employee types.

Enhancement: -- When the application has to be enhanced further to add a new type of employee there would be changes made in all the functions, which depend upon the type of employee. Hence enhancing an application further would be quite hectic. Why only new type of employee? Even if have to add a little more functionality to an existing function it would prove quite cryptic.

Extensibility: -- The current design of the application does not allow us to have extensibility easily. Therefore, if we have to add more functions, which would do certain tasks for all the employee types or maybe for some of the type of employees; the new function also will have a strict type inspection routine to check for the type of employee.

Storage: -- A small but significant problem. Whenever we have the data saved in a persistent storage i.e. having the data into files on the hard disk we will have to take care of saving the data along with the appropriate type of the employee. Also, while reading the data from the file its necessary to read the type first and then accordingly initialize the members in the structure.

If we try to look for the core problem in the application design which can be qualified as a cause for all the problems discussed above it definitely would be an attempt to design a common algorithm to suit all the types of employees. It would definitely prove helpful if rather than concentrating on the procedures we concentrate more upon the entities in the application.

6.2 Object Oriented Programming

Now with the major shortcomings of procedural programming let us look at how a different approach would help us. As mentioned earlier it is necessary to concentrate more upon the entities in the application and not only upon the tasks done by the application.

Based upon this bottom line we have a certain set of rules defined as object-oriented paradigm. If a programming language satisfies these rules i.e. provides certain features or keywords to implement these rules it would be qualified as an object oriented programming language.

Lets discuss the major conventions for object-oriented programming.

6.3 Classes

One of the major problems in the earlier approach was also the data and the functions working upon the data being separate. This leads to the necessity of checking the type of data before operating upon the data. The first rule of the object-oriented paradigm says that if the data and the functions acting upon the data can go together let them be together. We can define this unit, which contains the data and its functions together as a ***class***. A class can also be defined as a programmatic representation of an entity and the behavior of that entity can be represented by the functions in the class. In our earlier case study the employee, can be represented as a class. A class will contain its data into various variables, which would be termed as data members and the behavior of the class, which will be encapsulated, as functions will be termed as member functions.

6.4 Encapsulation

Many a times when we use certain tools, we hardly pay attention to the details about the functionality of the tool. We hardly pay attention to the various other units, which make up the tool. This behavior to ignore unwanted details of an entity is termed as abstraction.

Now if the details are unwanted why show them to the user? Therefore, the creator might attempt to hide these unwanted details. This behavior is termed as encapsulation. So we can say that encapsulation is an implementation of abstraction. Encapsulation directly leads to two main advantages:

Data Hiding: -- The user of the class does not come to know about the internals of the class. Hence, the user never comes to know about the exact data members in the class. The user interacts with the data members only through the various member functions provided by the class.

Data Security: - Since the data, members are not directly available to the user directly but are available only through the member functions a validity check can always be imposed to ensure that only valid data is been inserted into the class. So a Date class, which contains individual data members for storing date, month and year, will have it ensured the month is never 13 or the date is never exceeding 31.

6.5 Inheritance

What is common between a father and a son? At least one thing would be common – their assets!

In real life, inheritance allows us to reuse things that belong to a particular entity. Also, in object oriented world a class can inherit the properties and functionalities defined in some another class so that they can be reused. Then we have to be a bit careful in designing these classes because reusability cannot be done unless the classes are of the same type. So the class which would be reusing the functionalities of the other class in object oriented terms we would say that the class is "***deriving***" from the former class and is termed as the derived class. The class that is being "***derived from"*** is termed as the base class. Inheritance directly results in the following benefits: --

Reusability: -- Inheritance results in functionalities defined in one class being reused in the derived classes. So the efforts of rewriting the same functionality for every derived class is being saved. This definitely saves a lot of development time.

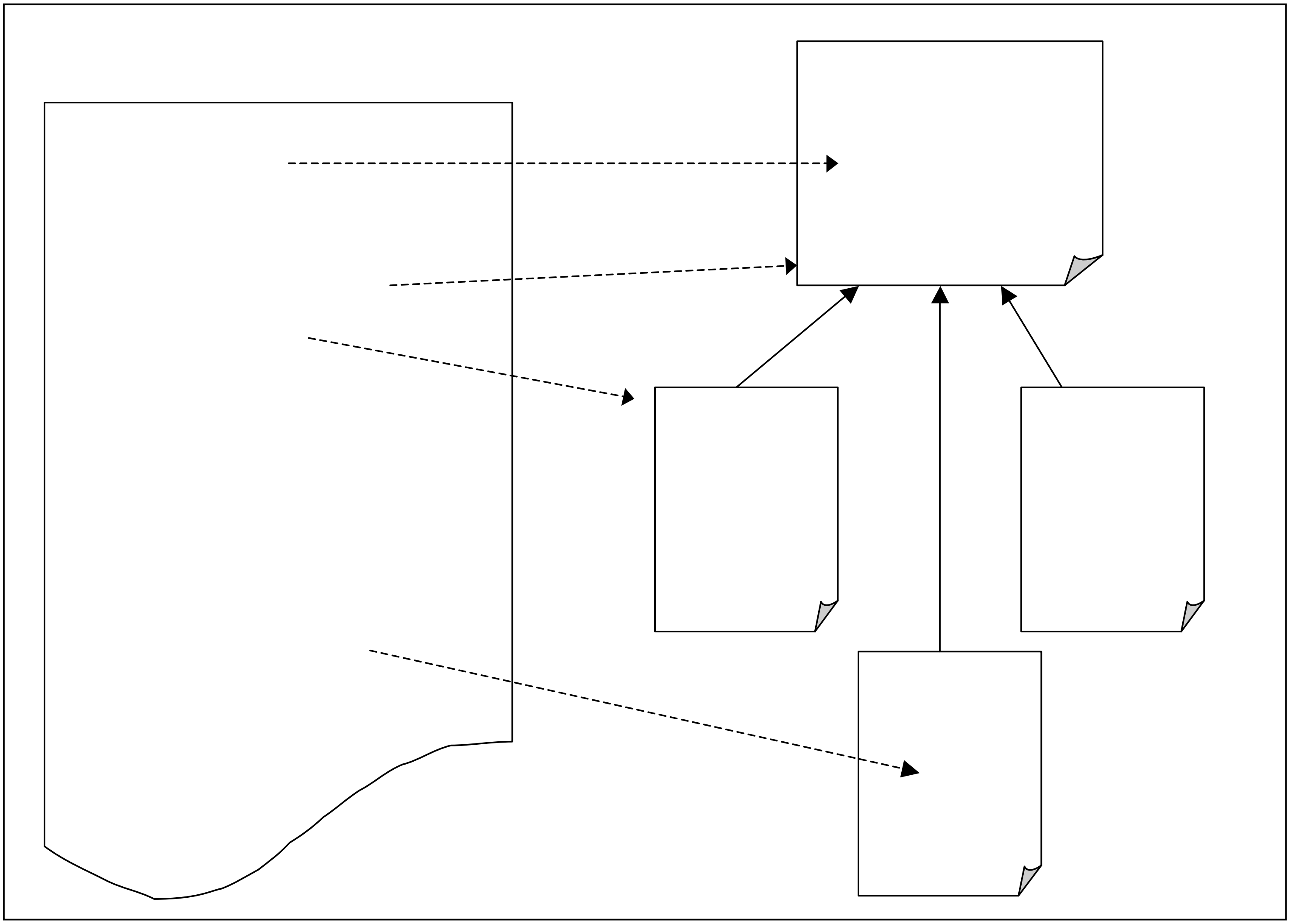
Enhancement and Specification: -- Due to the characteristic of inheritance, we can club the common functionalities in the base class and have the specific functionalities in the derived class. This feature can be used to have a functionality defined in the base class to be further modified for betterment or specification by the derived class. This mechanism of redefining the functionality of the base class in the derived class is termed as "*overriding"*

Avoiding type inspection:-- In the case study that we discussed to understand procedural approach we had a strict type inspection routine at the library end wherein in every function in the library we had to check for the type of the employee for whom the work has to be done. With inheritance, we would have a common base class called as "Employee" which would have all the common functionalities defined where as the specific routines do be done for various types of employees would go in the respective derived classes. So there would be class defined for every type of employee and the class would all the specifications for that type of employee and would be derived from the base class Employee to inherit the common functionalities. Therefore, in the functions now we wont have to check for the type of employee every time because every employee type has its own specific routines defined within it.

6.6 Polymorphism

The word “polymorphism” means “different forms”. Applied in object-oriented paradigm it means the ability of an entity to exhibit different forms at runtime. However, why would such a kind of feature be required? One major reason to have this is to eliminate the type inspection. As we can see in the earlier case study that we discussed there would also be a type inspection checking at the client application level where in the employee entities would be used. So just as in every functionality, we had checked for the type of employee we will also have to check in the main function about the type of employee we are handling. With polymorphism, we can have this level of type inspection also being eradicated totally.

Mapping the procedural approach to an object oriented scenario



How does .NET support object oriented programming?

As we have discussed in the earlier sections .NET happens a to be a "complete framework". The basic approach adopted by the framework is object-oriented and the framework would support only object-oriented code. So no more C and COBOL applications! Although C++ and OO-COBOL would work perfectly fine.

Since the framework is inherently object-oriented everything i.e. every data type in the framework will be a class. Unlike C++, even the primitive data types will be given by the framework as a set of classes.

The framework also provides us with a rich set of classes which can be used by instantiating them or writing new classes by deriving from the framework classes. The framework does have a systematic organization of classes wherein the Object class from the System namespace (we will discuss namespaces in details later) tops the chart. All the other classes are derived from the Object class.

Component Oriented Programming

However, is it really enough just to have an object oriented approach? Well, now with the increasing influence of the web and code reusability getting extended across the language barriers its very much essential to even extend “Object Oriented approach “ itself.

We need to extend the definition of the “class” which happens to be the basic element of object oriented programming. We know that the class serves as an abstraction or simulation of a real life entity. However, in order to have a total encapsulation the internals of the class has to be totally hidden. Secondly, the class should be instantiable across various programming languages to have a more range of reusability. The class should ideally support properties (will be discussed a little later) to offer more user friendliness and overcoming the incompatibilities.

Therefore, a class must have the following extra abilities in addition to what it serves in object-oriented scenario:

- Encapsulated data and implementations
- Properties
- Language Interoperable

We can term instance of such a class as a “component”. VB.NET and CSharp happen to be component- oriented languages in a way that every class created in any of these languages when instantiated results in a component.

6.7 Understanding CSharp and VB.NET as Object Oriented Programming languages

After knowing the object-oriented concepts let us examine how these concepts can be implemented in CSharp (C#) or VB.NET. These two languages are been introduced along with the .NET framework and are totally .NET compliant. So VB.NET will be a natural upgrade for VB programmers and CSharp for C++ programmers. We will be looking at the various syntaxes of both these languages to implement object oriented programming.

Classes

As we have discussed earlier classes are going to be an integrated unit consisting of the data and the functions that would act upon the data.

The Employee class would be something like this:

VB.Net

```
Public Class Employee
        Dim Empid As Integer
        Dim EmpName As String
        Dim EmpAddress As String
        Dim Basic As Double

         Public Sub AddDetails ()
           ---------
           ---------
         End Sub

          Public Sub PrintDetails ()
           ---------
           ---------
         End Sub

          Public Sub CalcSalary ()
           ---------
           ---------
         End Sub

End Class
```

C#

```
public class Employee
{
        int       Empid;
        string  EmpName;
        string  EmpAddress;
        double Basic;

         public void AddDetails ()
          {
           ---------
           ---------
           }
          public void PrintDetails ()
          {
           ---------
           ---------
          }
          public void CalcSalary ()
          {
           ---------
           ---------
           }
}
```

The attributes in our employee entity are defined as data members and the functionalities would be the member functions. So our classes happen to be independent entities in our application. Dependant upon the application the number of classes in the application might vary. We need to be a bit careful about the design of our classes specifically when we decide the data members and the member functions. We will have to ensure complete atomicity and that no class would be allowed to access the data of any other class unless it’s a strict constraint in the design of the classes.

Then just by declaring he class we have laid down a specification. The class would be actually in form or will be allocated memory only when we create an instance of that class i.e. creating an “object” of the class. Therefore, a class is just a blue print of an entity, which just tells what does the entity have, and how does the entity behave.

Based upon the program requirement we can have this object created either at compilation time (stack segment of the process memory) or at runtime (on the heap). But if the object is created at runtime it will also be necessary to de-allocate the object to avoid a memory leak. .NET framework provides us with an automatic memory management system. In addition, for this system to monitor our memory its mandatory to have all our objects created at runtime. Hence, in .NET scenario we are going to have only heap-based objects, which would get created at runtime.

At times we might in our application require multiple instances of the same class. The multiple objects of the class are going to have only their set of data members and they would share the copy of member functions. Then how would the data of the appropriate object get modified. As like any other programming language, the .NET languages also support the “this” reference, which would be an implicit argument of every function (non static function). This reference contains the address of the object through which the function was called.

Just as an application would require multiple instances of a same class an application also might require objects of different classes. These classes might be from different libraries. There are high chances that two different libraries might have a class of the same name. These naming clashes are solved in .NET by grouping classes logically in namespaces. So we can group different classes of a library into one or more namespaces. Whenever we use a library, we can specify to the compiler about the namespace we are referring.

Encapsulation

Encapsulation is all about hiding the data and ensuring its security. The data should not be accessible to any external entity unless the data is not that crucial.

C# and VB.NET support various access specifiers to encapsulate the data and the functions. The specifiers allow us to define the levels of access for the data members. Every member that we declare in a class has to have to its access specifiers as a part of the declaration statement itself. Unlike C++ , both C# and VB.NET do not support access grouping.

C# Access Specifiers

Access	Non static members
Do not require an object instance	Require an object instance
Are termed as class level members	Are termed as object level members
Static functions do not receive the “this” pointer as an implicit argument	Non static functions do receive the “this” pointer as an implicit member
Can access only other static and not the non static members	Can access both static as well as non static members

Constructors

Constructors are special member functions, which are used for initializing the class data members. In the earlier object oriented programming languages constructors were quite important since initialization of the data members was not allowed at the time of declaration. C# however allows us to have the member variables to be initialized along with declarations. Then why are constructors required? Well, the variables may not always be initialized to some constants but can also be initialized with the values specified by the user. There can also be certain kind of processing to be done while a class is instantiated. So a constructor happens to be quite important member function in the class as far as initialization is concerned.

C# supports following types of constructors

- Default Constructors
- Parameterized constructors
- Private constructors
- Static constructors

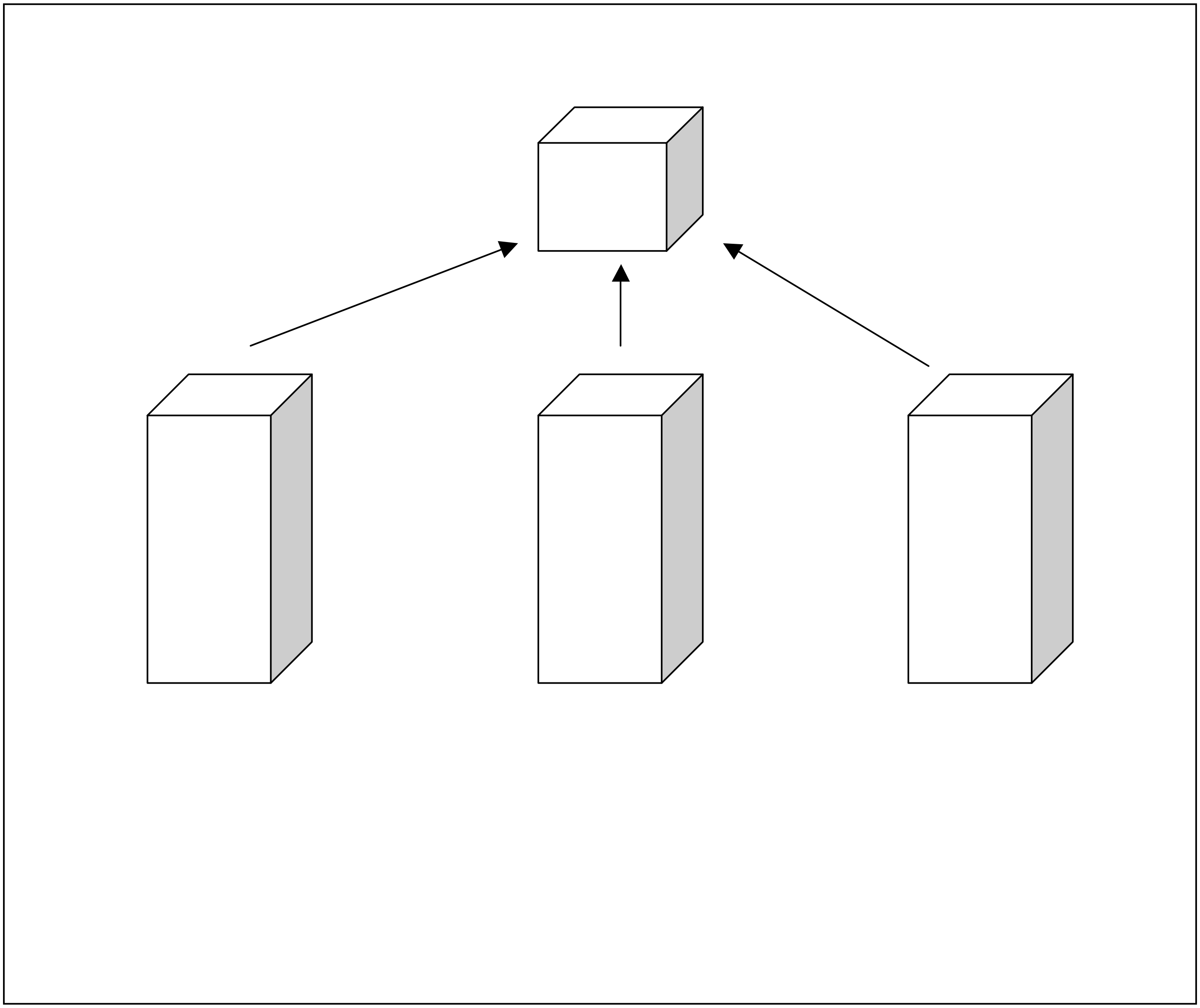
Destructors

C# also supports automatic memory cleanup by having a garbage collector element. So no more botheration for de-allocating memory from the heap and no more nightmares about dangling pointers. Then if there is a memory management mechanism why would we require destructors?

Static Members

Let us consider a scenario wherein we have two different MS WORD windows opened with some documents opened. We select some text from one of the windows, copy it and try to paste it another window. Now we know that both these windows happen to be two separate entities. So lets imagine two objects of the same kind to have similar kind of interaction. Definitely we would a common memory are which both the objects would be able to access. This memory area also has to restricted to the objects of that class only. The key point over here is the common area available to the objects.

In a second scenario, let us consider a requirement of generating ids automatically. Again, there has to be some variable, which would be common to all objects of that class and would keep on incrementing.



Static members	Non static members
Do not require an object instance	Require an object instance
Are termed as class level members	Are termed as object level members
Static functions do not receive the “this” pointer as an implicit argument	Non static functions do receive the “this” pointer as an implicit member
Can access only other static and not the non static members	Can access both static as well as non static members

Properties

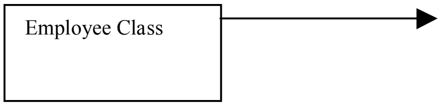
Classes were proposed in object oriented programming paradigm for one of the reasons of having data security. As we know, the members of a structure in C happen to be public and can be accessed freely outside the structure. As a result, there is not any check as to what data is been inserted into the structure members. Hence, in classes we have the “private” specifier by which we can avoid the direct access of the data members of the class. Then we lose the user friendliness of accessing a variable rather than calling a function.

Properties happen to be a fantastic blend of both the things. A property constitutes of a private level member to store the data and accessor and mutator methods to interact with the variable. Now what is new in that? The beauty is that the property would be accessed by the user as a local variable but internally the compiler will convert the access statements into appropriate function calls. So the user application always is under the impression that a variable is been accessed where as the validity of the data is been checked with the methods associated. The data stored in the variable is termed as value of the property.

Inheritence

It is one of the commonly used features in OOPS to avoid the code duplication. It implements code reutilization in class declaration. Let us take an example and discuss how the code reutilization is achieved with inheritence. Normally we create a singe class to represent an entity and its operations. Look at the following example

□□

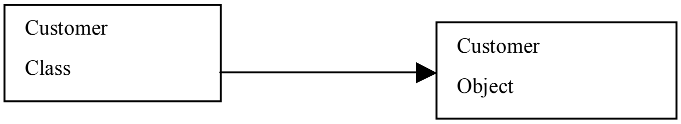


```
Class Employee
    Public EmpId As Integer
    Private Sal As Double = 0
    Public Basic As Double
    Public Allowance As Double
    Public Deducions As Double
    Public FirstName As String
    Public LastName As String
    Public Address As String
    Public Pincode As String
    Public Sub DisplayInfo()
        Dim msg As String
        msg = FirstName & " " & LastName & vbCrLf
        msg = msg & Address & vbCrLf
        msg = msg & "PIN – " & Pincode
        Msgbox(msg)
    End Sub
    Public ReadOnly Property Salary() As Double
        Get
            Return Sal
        End Get
    End Property
    Public Sub ProcessSalary()
        Sal = Basic + Allowance - Deductions
    End Sub
End Class
```

```
class Employee
{
    public int EmpId;
    private double Sal = 0;
    public double Basic;
    public double Allowance;
    public double Deducions;
    public string FirstName;
    public string LastName;
    public string Address;
    public string Pincode;
    public void  DisplayInfo()
   {
        string msg;
        msg = FirstName + " " + LastName + vbCrLf;
        msg = msg + Address + vbCrLf;
        msg = msg + "PIN – " + Pincode;
        MesssgeBox.Show(msg);
    }
    public double Salary
    {
        get
        {
            return Sal;
         }
    }
    public void ProcessSalary()
    {
        Sal = Basic + Allowance – Deductions;
    }
}
```

In the above example, *employee class* contains methods and properties defined in its structure. Employee object is an instance.

□
□
□



```
Class Customer
   Public CustId As Integer
   Public DebitBalance As Double
   Public FirstName As String
   Public LastName As String
   Public Address As String
   Public Pincode As String
   Public Sub DisplayInfo()
      Dim msg As String
      msg = FirstName & " " & LastName & vbCrLf
      msg = msg & Address & vbCrLf
      msg = msg & "PIN – " & Pincode
   End Sub
   Public ReadOnly Property Debit() As Double
      Get
         Return DebitBalance
      End Get
   End Property
End Class
```

```
class Customer
{
   public int CustId;
   public double DebitBalance;
   public string FirstName;
   public string LastName;
   public string Address;
   public string Pincode;
   public void DisplayInfo()
   {
      string msg;
      msg = FirstName + " " + LastName ;
      msg = msg + Address ;
      msg = msg + "PIN – " ;
   }
   public double Debit()
   {
      get
      {
         return DebitBalance;
      }
   }
}
```

Customer class contains methods and properties defined in its structure. Customer object is an instance.

In these two classes, you might have observed the person identification is same in both Employee and Customer class. it means firstname, lastname, address and pincode variable members and *displayInfo* method is same in both the classes.

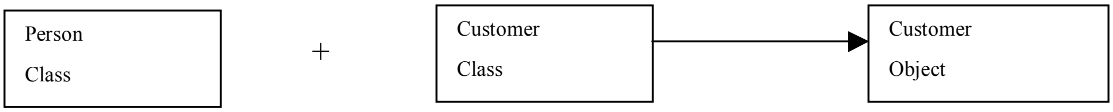
So this common information can be isolated and written in separate class and inherited into the respective employee and customer class. it is shown in the following example.

Base Class:-

```
Class Person
    Public FirstName As String
    Public LastName As String
    Public Address As String
    Public Pincode As String
    Public Sub DisplayInfo()
        Dim msg As String
        msg = FirstName & " " & LastName & vbCrLf
        msg = msg & Address & vbCrLf
        msg = msg & "PIN – " & Pincode
        Msgbox(msg)
    End Sub
End Class
```

```
class Person
{
    public string FirstName;
    public string LastName;
    public string Address;
    public string Pincode;
    public void DisplayInfo()
    {
        string msg;
        msg = FirstName + " " + LastName;
        msg = msg + Address ;
        msg = msg + "PIN – " + Pincode;
        MessageBox.Show(msg);
    }
}
```

□□
□□
□

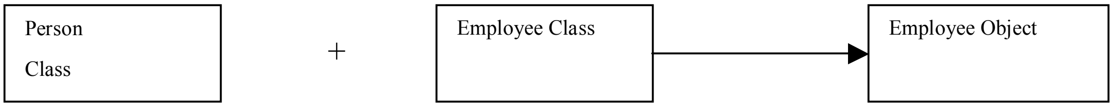


Derived Class:-

```
Class Customer
    Inherits Person
    Public CustId As Integer
    Public DebitBalance As Double
    Public ReadOnly Property Debit() As Double
        Get
            Return DebitBalance
        End Get
    End Property
End Class
```

```
class Customer:Person
{
    public int CustId ;
    public double DebitBalance;
    public double Debit()
    {
        get
        {
            return DebitBalance;
        }
    }
}
```

□□□□
□



Derived Class:-

```
Class Employee
    Inherits Person
    Public EmpId As Integer
    Private Sal As Double = 0
    Public Basic As Double
    Public Allowance As Double
    Public Deductions As Double
    Public ReadOnly Property Salary() As Double
        Get
            Return Sal
        End Get
    End Property
    Public Sub ProcessSalary()
        Sal = Basic + Allowance - Deductions
    End Sub
End Class
```

```
class Employee: Person
{
   public int EmpId ;
   private double Sal = 0;
   public double Basic ;
   public double Allowance;
   public double Deductions;
   public double Salary
   {
      get
       {
         return Sal;
      }
   }
   public void  ProcessSalary()
   {
      Sal = Basic + Allowance – Deductions;
   }
}
```

In the above mentioned example *Person class* holds the common data (Firstname, Lastname… etc) and method displayInfo(). It has been inherited in both *employee* and *customer class*. So these two achieves the same functionality of what we have seen before inheritance. By this point we conclude *inheritance* implements reuse of the same code with multiple classes.

One more advantage with inheritance is extensibility of the of the derived class code. It means the *employee* and *customer class* be extended by including its own methods and properties with the *person (Inherited) class*. Here the extended members in Employee class are *EmpId, Allowance, ProcessSalary method and Salary property*. The same thing follows in *customer class* with *CustId, DebitBalance and Debit property*.

You might have observed the keywords Base class and Derived class in the above session. Let us see what it means.

Base class:- A class which contains common properties and methods that can shared with other classes by inheritance is called *Base class*. Ex:- Person class

Derived class:- A class which inherits the base class is knows as *Derived class*. ex:- Employee class and Customer class.

Implementation:- A derived class can inherit only one base class. its shown in the above examples, ie., *employee class* inherits *person class* and *customer class* inherits *person class*.

You can inherit the base class into derived class using **Inherits** keyword**.**

ex:-

```
Class Employee
                Inherits Person
                :
                :
End Class
```

```
class Employee:Person
{
                :
                :
}
```

Protected Keyword:- We have already seen the usage of Public and Private keyword.

As we know, all the **Public** keyword declarations in the class will be accessed by the object users and the derived class (the class which inherits the base class). **Private** keyword declarations can be accessed only within the class (it means the class in which the declaration is done).

You may think why this *Protected* keyword declaration is required.

Its functionality is a hybrid of public and protected keyword. So, its very important in class inheritance, because in the situation where the data is to be communicated only between the base and derived classes irrespective of the external object user (means the end user) the *protected* keyword is used

Let us take an example and see how it will be used

Base Class:-

```
Class Person
   Public FirstName As String
   Public LastName As String
   Public Address As String
   Public Pincode As String
   Protected DateOFBirth As DateTime
   Public Sub DisplayInfo()
      Dim msg As String
      msg = FirstName & " " & LastName & vbCrLf
      msg = msg & Address & vbCrLf
      msg = msg & "PIN – " & Pincode
      msg = msg & "Date of Birth : " & DateOFBirth.ToString
   End Sub
End Class
```

```
class Person
{
   public string FirstName ;
   public string LastName ;
   public string Address;
   public string Pincode;
   protected DateTime DateOFBirth ;
   public void  DisplayInfo()
   {
      string msg;
      msg = FirstName + " " + LastName;
      msg = msg + Address ;
      msg = msg + "PIN – " + Pincode;
      msg = msg + "Date of Birth : " + DateOFBirth.toString;
   }
}
```

The Protected variable **dateofbirth** is accessed in the *displayinfo method* of the *base class* itself.

Derived Class:-

```
Class Employee
    Inherits Person
    Public EmpId As Integer
    Private Sal As Double = 0
    Public Basic As Double
    Public Allowance As Double
    Public Deducions As Double
    Public ReadOnly Property Salary() As Double
        Get
            Return Sal
        End Get
    End Property
    Public Sub ProcessSalary()
        Sal = Basic + Allowance – Deductions
    End Sub
    Public ReadOnly Property Age() As Integer
        Get
            Dim personAge As Integer
            personAge = Date.Now.Subtract(DateofBirth).Days
            Return personAge
        End Get
    End Property
End Class
```

```
class Employee: Person
{
    public int EmpId ;
    private double Sal = 0;
    public double Basic;
    public double Allowance;
    public double Deducions;
    public double Salary
    {
        get
        {
            return Sal;

        }
    }
    public void ProcessSalary()
    {
        Sal = Basic + Allowance – Deductions;
    }
    public int Age
    {
        get
        {
            int personage;
            personAge = Date.Now.Subtract(DateofBirth).Days;
            return personage;

        }
    }
}
```

As in the same way of base class the protected variable *dateofbirth* of the base class is accessed in the derived class. So the protected variable in the base class looks like a private variable for the derived class and cannot be accessed by its object users (means outside the class environment).

Instantiation of the Derived Class :- After declaration of the derived class we can create the object instance of the derived class and use it for the specific task. This is called *Object Instantiation*. With the instance of the derived class you can access all the *public properties* and *methods* of both the base and derived classes.

Let us take an *employee class* example.

```
 Dim objEmployee1 As New Employee() 'Create an Instance of the
     'Employee class
     objEmployee1.EmpId = 100                    'Derived Class member
     objEmployee1.firstname = "Rama"             'Base Class member
     objEmployee1.lastname = "S"        'Base Class member
     objEmployee1.Address = "#8, Kalidasa road, Mysore"
                                                 'Base Class member
     objEmployee1.pin = "570002"        'Base Class member
     objEmployee1.Basic = 5000          'Derived Class member
     objEmployee1.allowances = 4000     'Derived Class member
     objEmployee1.Deductions = 1000     'Derived Class member
     objEmployee1.ProcessSalary()       'Derived Class member
     objEmployee1.DisplayInfo()         'Base Class member
```

```
 Employee objEmployee1 =  new Employee(); 'Create an Instance of the 'Employee
class
     objEmployee1.EmpId = 100;                   'Derived Class member
     objEmployee1.firstname = "Rama";            'Base Class member
     objEmployee1.lastname = "S";       'Base Class member
     objEmployee1.Address = "#8, Kalidasa road, Mysore";
                                                 'Base Class member
     objEmployee1.pin = "570002";       'Base Class member
     objEmployee1.Basic = 5000;         'Derived Class member
     objEmployee1.allowances = 4000;    'Derived Class member
     objEmployee1.Deductions = 1000 ;            'Derived Class member
     objEmployee1.ProcessSalary();      'Derived Class member
     objEmployee1.DisplayInfo();        'Base Class member
```

In the above code, object instance *objEmployee* of *Employee class* is created. And then all the public members of both base and derived class are accessed and manipulated.

<u>System.Object:- </u> This is the root class for all the objects in the .NET framework, from which all the other classes are derived. It contains some basic methods and properties, which can be accessed from all the object instances of the .NET framework.

Look into the code which calls system.object methods.

```
Dim Obj As New System.Object()
          Obj.ToString()
          Obj.GetHashCode()
          Obj.GetType()
Dim objEmployee As New Employee()
          objEmployee.ToString()
          objEmployee.GetHashCode()
          objEmployee.GetType()
```

```
System.Object Obj = new System.Object();
          Obj.ToString();
          Obj.GetHashCode();
          Obj.GetType();
Employee objEmployee = New Employee();
          objEmployee.ToString();
          objEmployee.GetHashCode();
          objEmployee.GetType();
```

The above code shows some of the methods that can be accessed directly with the instance of the *system.object* ie., *Obj* and also the same methods can be accessed from *objEmployee* too. So, *objEmployee* is inherited from *System.Object* class.

6.8 Polymorphism

It is the capability to have methods and properties in multiple classes that have the same name can be used interchangeably, even though each class implements the same properties or methods in different ways.

Let us understand what is polymorphism with the following example.

Now we will consider the maintenance of company information which includes *employee* and *customer* details. A *person* (base class) is defined to hold the common information of the individual. The base class maintains contact information of the person and manipulates the data. It contains *save method* to update the contact information of the person.

Two derived classes, Employee and Customer are used to process the employee and customer details. These two derived classes inherits *person class* to manipulate identity of the person instead of rewriting the same code again in the derived classes. Since each derived class needs to use the *displayinfo method* to display the additional information with the contact details, the *displayinfo method* of the base class will be overwritten in the respective derived class using *overrides* keyword. The overriding member signature in the derived class must be as the base class signature.

Signature includes the member type, member name, parameters datatype and return datatype.

Look into the following example and see the implementation of base class (Person) and the derived class (Employee), observe the changes in the *displayInfo* method in both the classes.

Ex:-

```
Class Person
    Private Name As String
    Private Address As String
    Public ReadOnly Property PName() As String
        Get
            Return Name
        End Get
    End Property
    Public ReadOnly Property PAddress() As String
        Get
            Return Address
        End Get
    End Property
    Public Overridable Function DisplayInfo() As String
        Dim msg As String
        msg = "Name : " & Name & vbCrLf
        msg = msg & "Address : " & Address & vbCrLf
        Return msg
    End Function
    Public Sub Save(ByVal parName As String, ByVal parAddress As String)
        Name = parName
        Address = parAddress
    End Sub
End Class
```

```
class Person
{
    private string Name;
    pPrivate string Address;
    public string PName()
    {
        get
        {
            return Name;
        }
    }
    public string Paddress
    {
        get
        {
            return Address;
        }
    }
    public virtual string  DisplayInfo()
    {
        string msg;
        msg = "Name : " + Name;
        msg = msg + "Address : " + Address;
        return msg;
    }
    public void Save(string parName, string parAddress)
    {
        Name = parName;
        Address = parAddress;
    }
}
```

Derived Class:-

```
Class Employee
    Inherits Person
    Public EmpId As Integer
    Private Sal As Double = 0
    Public Basic As Double
    Public Allowance As Double
    Public Deductions As Double
    Public Overrides Function DisplayInfo() As String
        Dim msg As String
        msg = MyBase.DispalyInfo()
        msg = msg & "ID : " & EmpId.ToString & vbCrLf
        msg = msg & "Basic : " & Basic.ToString & vbCrLf
        msg = msg & "Allowances : " & Allowance.ToString & vbCrLf
        msg = msg & "Deductions : " & Deductions.ToString & vbCrLf
        msg = msg & "Net Salary : " & Sal.ToString & vbCrLf
               return(msg)
    End Function
    Public ReadOnly Property Salary() As Double
        Get
            Return Sal
        End Get
    End Property
    Public Sub ProcessSalary()
        Sal = Basic + Allowance - Deductions
    End Sub
End Class
```

```
class Employee: Person
{
    public int EmpId;
    private double Sal = 0;
    public double Basic;
    public double Allowance;
    public double Deductions;
    public override string DisplayInfo()
    {
        string msg ;
        msg = base.DispalyInfo();
        msg = msg + "ID : " + EmpId.ToString;
        msg = msg + "Basic : " + Basic.ToString;
        msg = msg + "Allowances : " + Allowance.ToString;
        msg = msg + "Deductions : " + Deductions.ToString;
        msg = msg + "Net Salary : " + Sal.ToString ;
               return(msg)
    }
    public double Salary
    {
        get
        {
            return Sal;
        }
    }
    public void ProcessSalary()
    {
        Sal = Basic + Allowance – Deductions;
    }
}
```

The following keywords are used in achieving polymorphism.

- Overridable:- A method or property defined in the base class can be overwritten in the derived class.
- Overrides:- Indicates the method or property is being overwritten in the derived class.
- Mustoverrides:- A method or property defined in the base class must be overwritten in the derived class
- Notoverridable:- A method or property defined in the base class must not be overwritten in the derived class.

Virtual Members :-Virtual members are those that can be overridden and replaced by the derived classes. They are declared with *Overridable* keyword. The methods or properties which doesn’t contain overridable keyword are called *Non-Virtual* members.

For ex:- *DisplayInfo* method is an virtual method of *Person* base class, because it has been overwritten in the Employee derived class with new *DisplayInfo* method with same signature.

Restricting Polymorphism :-

We have already seen polymorphism is implemented with *Overridable* and *Overrides* keyword. Some situation arises where we need to break the polymorphism effect of the base class method. It means changing the signature of the base class member while overriding it in the derived class violates the polymorhism rule *"signatures must be same"*. Restriction is required to completely change the implementation of the member in the derived class. So it restricts the polymorphism of the base class member with new implementation in the derived class. The *Shadow* keyword used to implement this concept.

Take an example shown below which contains two classes 1) Person Class 2) Employee class.

Person class is a base class contains method to be overwritten from the derived class. The signature of the overridable method is

```
Public Overridable Function DisplayInfo() as String
```

Employee class is a derived class which inherits *person class* and restricts the implementation of the polymorphism method *displayinfo* of the base class with *shadows* keyword.

```
Public Shadows Sub DisplayInfo()
```

From the above explanation, we can observe that the signature of the *DisplayInfo* member is changed from *function* in the base class to the *subroutine* in the derived class with complete change in the implementation.

Ex:-

Base Class:-

```
Class Person
    Public Name As String
    Public Address As String
    Public Overridable Function DisplayInfo() As String
        Dim msg As String
        msg = Name & vbCrLf
        msg = msg & Address & vbCrLf
        Return msg
    End Function
    Public Sub Save(ByVal parName As String, ByVal parAddress As String)
        name = parName
        address = parAddress
    End Sub
End Class
```

```
class Person
{
    public string Name;
    public string Address ;
    public virtual string DisplayInfo()
    {
        string msg;
        msg = Name ;
        msg = msg + Address;
        return msg;
    }
    public void Save(string parName, string parAddress)
    {
        name = parName;
        address = parAddress;
    }
}
```

Derived Class:-

```
Class Employee
    Inherits Person
    Public EmpId As Integer
    Private Sal As Double = 0
    Public Basic As Double
    Public Allowance As Double
    Public Deductions As Double
    Public Shadows Sub DisplayInfo()
        Dim msg As String
        msg = MyBase.DisplayInfo("")
        msg = msg & "ID : " & EmpId.ToString & vbCrLf
        msg = msg & "Basic : " & Basic.ToString & vbCrLf
        msg = msg & "Allowances : " & Allowance.ToString & vbCrLf
        msg = msg & "Deductions : " & Deductions.ToString & vbCrLf
        msg = msg & "Net Salary : " & Basic.ToString & vbCrLf
        MsgBox(msg)
    End Sub
    Public ReadOnly Property Salary() As Double
        Get
            Return Sal
        End Get
    End Property
    Public Sub ProcessSalary()
        Sal = Basic + Allowance - Deductions
    End Sub
End Class
```

```
class Employee: Person
{
   public int EmpId ;
   private double Sal = 0;
   public double Basic;
   public double Allowance;
   public double Deductions;
   public new void DisplayInfo
   {
      string msg;
      msg = Base.DisplayInfo("");
      msg = msg + "ID : " + EmpId.ToString;
      msg = msg + "Basic : " + Basic.ToString;
      msg = msg + "Allowances : " + Allowance.ToString;
      msg = msg + "Deductions : " + Deductions.ToString;
      msg = msg + "Net Salary : " + Basic.ToString;
      MessageBox.Show(msg);
   }
   public double Salary
   {
      get
      {
         return Sal;
      }
   }
   public void ProcessSalary()
   {
      Sal = Basic + Allowance – Deductions;
   }
}
```

6.9 Abstract Classes (Virtual Class)

So far, we have seen how to inherit the class, how to overload and override methods. In the examples shown in inheritance topic, parent class has been useful in both inheritance and create an instance also.

Actually in some situation the classes are required only for inheritance, such type of classes are called **Abstract classes.** This concept is very useful when creating a framework for the applications where there will not be any implementation for the methods and properties in the abstract class. It just defines the structure.

Abstract classes are declared using MustInherit and MustOverride keyword.

Syntax:-

```
Public MustInherit Class AbstractBaseClass
   Public MustOverride Sub DoSomething()
   Public MustOverride Sub DoOtherStuff()
End Class
```

```
public abstract class AbstractBaseClass
{
   public abstract void DoSomething();
   public abstract void DoOtherStuff();
}
```

MustInherit keyword is used with the class name, where as ***MustOverride*** keyword is used with the members of the class.

Implementaion:-

```
Public Class DerivedClass
    Inherits AbstractBaseClass
    Public Overrides Sub DoSomething()
        MsgBox("This method is overrides the Base class DoSomething to implement the
method functionality”)
    End Sub
    Public Overrides Sub DoOtherStuff()
        MsgBox(“This method is overrides the Base class DoOtherStuff  to implement the
method functionality”)
    End Sub
End Class
```

```
public class DerivedClass : AbstractBaseClass
{

    public Override void DoSomething()
    {
        MessagegBox.Show("This method is overrides the Base class DoSomething to
implement the method functionality”);
    }
    public Override void DoOtherStuff()
    {
        MessaggeBox.Show(“This method is overrides the Base class DoOtherStuff  to
implement the method functionality”);
    }
}
```

MustInherit forces the classes to be inherited from the derived class and write an implementation code for the methods and properties in the derived class before using it.

As in the above example, any class that inherits AbstractBaseClass must implement the *Dosomething* and *DoOtherStuff* methods.

We cannot create an instance of the *AbstractBaseClass* as shown below.

```
Dim obj as New AbstractBaseClass()
‘Error in Decleration
```

```
AbstractBaseClass obj = new AbstractBaseClass()
‘Error in Decleration
```

Restricting Inheritence

If we want to prevent a class being used as a *Base class* we can use ***NotInheritable*** keyword with the class declaration.

```
Public NotInheritable Class NormalClass
    'Decleration of Class members

End Class
```

```
public sealed class NormalClass
{
    'Decleration of Class members

}
```

For example when we want an employee class need not to be used as a Base class, we can declare the employee class structure with *NotInheritable* keyword. So that it can be used only for instantiation.

VB.NET

```
Class NotInheritable Employee
    Inherits Person
    Public EmpId As Integer
    Private Sal As Double = 0
    Public Basic As Double
    Public Allowance As Double
    Public Deductions As Double
    Public Shadows Sub DisplayInfo()
        Dim msg As String
        msg = MyBase.DisplayInfo("")
        msg = msg & "ID : " & EmpId.ToString & vbCrLf
        msg = msg & "Basic : " & Basic.ToString & vbCrLf
        msg = msg & "Allowances : " & Allowance.ToString & vbCrLf
        msg = msg & "Deductions : " & Deductions.ToString & vbCrLf
        msg = msg & "Net Salary : " & Basic.ToString & vbCrLf
        MsgBox(msg)
    End Sub
    Public ReadOnly Property Salary() As Double
        Get
            Return Sal
        End Get
    End Property
    Public Sub ProcessSalary()
        Sal = Basic + Allowance - Deductions
    End Sub
End Class
```

Following decleration is not posssible, .NET generates error!!!!

```
Public Class Staff
Inherits Employee
End Class
```

C#

```
class sealed Employee: Person
{
    public int EmpId ;
    private double Sal = 0;
    public double Basic;
    public double Allowance;
    public double Deductions;
    public new void DisplayInfo
    {
        string msg;
        msg = Base.DisplayInfo("");
        msg = msg + "ID : " + EmpId.ToString;
        msg = msg + "Basic : " + Basic.ToString;
        msg = msg + "Allowances : " + Allowance.ToString;
        msg = msg + "Deductions : " + Deductions.ToString;
        msg = msg + "Net Salary : " + Basic.ToString;
        MessageBox.Show(msg);
    }
    public double Salary
    {
        get
        {
            return Sal;
        }
    }
    public void ProcessSalary()
    {
        Sal = Basic + Allowance – Deductions;
    }
}
```

Following decleration is not posssible, .NET generates error!!!!

```
Public Class Staff: Employee
{
 ----
}
```

6.10 Interfaces

An interface is like an abstract class which allows the derived class to *inherit more than one interfaces* into it.

We have already seen an *abstract class* can have methods with or without implementation. But an *interface* can only have members without implementation. So it provides only structure of the objects like abstract class.

To implement an interface in VB.NET, we use the keyword **Implements** and we must provide implementations for all the methods of the interface we implements.

Defining an Interface:- Interfaces are declared with the following structure

```
Public Interface <Name of the interface>
          :
          <decleration of memebrs of the interface, means methods and properties
structure>
:
End Interface
```

Let us take up a example and see how the declaration been done

```
Interface IShape
   Sub Draw(ByVal coord() As ArrayList)
End Interface
```

```
interface IShape
{
   void Draw(ArrayList coord);
}
```

Implementation of Interface:- Interfaces are implemented using **Implements** keyword. All the members of the interface are implemented with Implements keyword.

"To implement a interface, we must implement all the methods and properties defined by the interface"

If more than one interface is to be implemented, then interfaces names should separated by commas with the implements keyword while defining a class.

Lets us take an example of implementation:-

```
Public Class Drawing
    Implements Ishape
    Public Sub Draw(ByVal parCoord() As ArrayList)        Console.Write("Draw a Circle")
    End Sub
End Class
```

```
Public Class Drawing: Ishape
{

   Public void Draw (ArrayList parCoord)
   {
     Console.Write("Draw a Circle");
   }
}
```

In the example, we are implementing Isahpe interface, which contains *Draw* method into the *Drawing* Class.

Difference between Abstract class and Interfaces

Abstract Class	Interface
Only one Abstract class can be inherited into the derived class.	Interfaces enable multiple inheritance to the object
Members of the abstract class can or cannot have implemention	Interfaces contains only the definitions for the menbers without implementation

When to use the interfaces in programming?

Solution:- The situation at which we need to implement the functionalities of two or more objects into one derived object. So it makes the derived object to refer to the interface methods and properties for syntax verification.

6.11 Delegates and Events

Delegates:-

The Delegate class is one of the most important classes to know how to use when programming .NET. Delegates are often described as the 'backbone of the event model' in VB.NET. In this we are going to introduce you to delegates (Sytem.Delegate class) and show you how powerful of an event mechanism through delegates.

Delegates are implemented using the delegate class found in the System namespace. A delegate is nothing more than a class that derives from System.MulticastDelegate. A delegate is defined as a data structure that refers to a static method or to a class instance and an instance method of that class. In other words, a delegate can be used like a type safe pointer. You can use a delegate to point to a method, much like a callback or the 'AddressOf' operator in VB.NET. Delegates are often used for asynchronous programming and are the ideal method for generically defining events.

Before you get into the actual delegate class let us see some code which simulates the delegate functionality through simple class code for the sake of understanding. We have a simple class called *'VBDelegate'* with two static methods named *'CallDelegate'* and *'DisplayMessage'* as shown below. When the *CallDelegate* method is called,(when the program is run) to display the message. Now normally, if we had a class called *'VBDelegate'* with a method named *'DisplayMessage'*

VB.NET

```
Public Class VBDelegate
            'Static method to call displaymessage function
  Public Shared Sub CallDelegate()
    DisplayMessage("Some Text")
  End Sub

     'Static Function to display the message
  Private Shared Function DisplayMessage(ByVal strTextOutput As String)
    MsgBox(strTextOutput)
     End Function

End Class
```

C#

```
Public Class CSDelegate
{
  'Static method to call displaymessage function
  Public static void CallDelegate()
  {
    DisplayMessage("Some Text")
  }

  'Static Function to display the message
  Private static void DisplayMessage(String strTextOutput)
```

```
    {
        MessageBox.Show(strTextOutput)
    }

}
```

There is nothing wrong with this approach at all. In fact, it is probably more commonly used than delegates. However, it does not provide much in terms of flexibility or a dynamic event model. With delegates, you can pass a method along to something else, and let *it* execute the method instead. Perhaps you do not know which method you want to call until runtime, in which case, delegates also come in handy.

Now we will implement the same functionality as before, using a actual delegate class of .NET.

Delegates in VB.NET are declared like:
Delegate [Function/Sub] *methodname*(arg1,arg2..argN)

The declared delegate *methodname* will have the same method signature as the methods they want to be a delegate for. This example is calling a shared method..

VB.NET

```
Class VBDelegate
    'Declaration of delegate variable with arguments
    Delegate Function MyDelegate(ByVal strOutput As String)
    'Function to call the delegates
    Public Shared Sub CallDelegates()
        'Declare variables of type Mydelegate points to the
        'function MessageDisplay with AddressOf operator
        Dim d1 As New MyDelegate(AddressOf MesssageDisplay)
        Dim d2 As New MyDelegate(AddressOf MesssageDisplay)
        'Pass the arguments to the function through delegates
        d1("First Delegation ")
        d2("Second Delegation ")
    End Sub
    'Function to display the message
    Private Shared Function MesssageDisplay(ByVal strTextOutput As String)
        MsgBox(strTextOutput)
    End Function
End Class
```

C#

```
Class CSDelegate
{
   'Declaration of delegate variable with arguments
    delegate void MyDelegate(String strOutput);
    'Function to call the delegates
    Public static void CallDelegates()
    {
        'Declare variables of type Mydelegate points to the
        'function MessageDisplay
        MyDelegate d1 = New MyDelegate(MesssageDisplay);
        MyDelegate d2 = New MyDelegate(MesssageDisplay);
        'Pass the arguments to the function through delegates
        d1("First Delegation ");
        d2("Second Delegation ");
    }
    'Function to display the message
    Private static void MesssageDisplay(String strTextOutput)
    {
       MessgeBox.Show(strTextOutput);
    }
}
```

The Output to the display window is:

First Delegation in one message window

And

Second Delegation in the second message window

What has happened in the above code? Let's take a look

First, we defined a delegate. Remember, when defining a delegate it is very similar to stating the signature of a method. We have said we want a delegate that can accept a string as an argument so basically, this delegate can work with any method that takes the same argument(s).

In our *CallDelagates* method, we create two instances of our *'MyDelegate'*. Then, we pass into MyDelegate's constructor the address of our *'DisplayMessage'* method. This means that the method we pass into the delegate's constructor (in this case it's *'DisplayMessage'* method) must have a method signature that accepts a string object as an input argument, just like our delegate does. Now, you might be thinking, "why are we passing in the address of a method when we defined our delegate to accept a string object as it's input argument?" In the code above, we are telling the delegate which method to call, not which string we're passing in. Carefully understand this concept.

Then finally we have our *'DisplayMessage'* method, which takes the string passed in by the delegate and tells it what string is to be displayed.

The same approach is used while handling the events in the .NET. This topic is just to understand how delegates works in .NET.

<u>Events:-</u>

Events are nothing but situations at which the message is sent to an object to signal the occurrence of the action. These actions can be caused by the user interaction or within the object itself.

For example the class designer can create an event and raise the same through its methods and properties. These events will be captured by the object user and perform his/her required operation.

Let us take a simple example of VB.NET Button class. Button is a class defined in *System.Controls.FormsControls* nameplace. The users create an instance of the *Button class* and select the **onclick event** related to the button object and write an action to be performed on click of the button.

Actually this event is already defined in *button class* and will be raised when the user clicks the button. The following example shows how the event can be defined, raised in the class and used by the object user.

Declaration:- The event member will be declared with *Event* keyword. Basically the event member should always be *public.* Because these are the members will be used by the external users.

```
'Declare a class which operates on the Employee collection database
'This class is used to find some summarised operation on the Employee
'collction database, which means finding the relavent employee ‘information, 'getting the
total no. of employees in the collection and ‘others   -  Its just 'an example to explain how
event works
Public Class EmployeeCollection
    'Declare an event which will be raised after finding the data
    'Keyword ‘Event’ is used the declare the events
    Public Event FindResult(ByVal blnFound As Boolean)
'This method is to find the data from employee colletion database and 'raise the findresult
event to return the result
    Public Sub FindData(ByVal Name As String)
        'find the Employee with name and return the result as boolean, if
  'the data is found then raise FindResult with True else with
  'False
        Dim found As Boolean
        found = FineEmployee(Name)
        If found Then
            'Raise the event with parameter
            RaiseEvent FindResult(True)
        Else
            'Raise the event with parameter
            RaiseEvent FindResult(False)
        End If
    End Sub
End Class
```

Usage:- In order to access the events of the objects, the object should be declared with *withevents* clause. This is shown in the following example with form load event.

```
'Declare the object with WithEvents clause to create an instance
Dim WithEvents objEmpColl As EmployeeCollection = New EmployeeCollection()
Public Sub load()
    'Find the Employee with name Rama in the Employee collection
    objEmpColl.FindData("Rama")
End Sub
'The following event will be raised after the search operation
Private Sub objObject_FindResult(ByValue blnFound as Boolean) Handles
objObject.FindResult
    If blnFound Then
        MsgBox("The given Employee is Found in the collection")
    Else
        MsgBox("The given Employee is not Found")
    End If
End Sub
```

6.12 Structures

Structures are used to create a variable set of different datatypes in VB.NET (In earlier versions of VB we use TYPE and END TYPE to define it). Here it is defined with STRUCTURE and END STRUCTURE keyword.

It supports allmost all the features of OOPS, like

- Implementing interfaces
- Constructors, methods, properties, fields, constants and events
- Shared constructors

Ex: -

Defining the structure:-

VB.NET

```
Structure Person
    Public FirstName As String
    Public LastName As String
    Public Address As String
    Public Pincode As String
    Public DateOFBirth As DateTime
    Public Sub DisplayInfo()
        Dim msg As String
        msg = FirstName & " " & LastName & vbCrLf
        msg = msg & Address & vbCrLf
        msg = msg & "PIN – " & Pincode
        msg = msg & "Date of Birth : " & DateOFBirth.ToString
    End Sub
End Structure
```

C#

```
struct Person
{
    Public String FirstName;
    Public String LastName ;
    Public String Address ;
    Public String Pincode ;
    Public DateTime DateOFBirth;
    Public void DisplayInfo()
    {
        String msg=new String();
        msg = FirstName + " " + LastName ;
        msg = msg + Address ;
        msg = msg + "PIN – " + Pincode;
        msg = msg + "Date of Birth : " + DateOFBirth.ToString;
    }
}
```

In the example a *Person* structure is declared to hold the a person's details like name, address, pincode etc., we have already seen this person details in terms of a class object. Basically the structures are used to hold the set of values like array. Theoritically arrays holds a set of single datatype values, but structures holds a set of different datatype values. Due to the OOPS feature of .NET we can implement the methods and properties in the structures too. This is shown in the *person structure.* Here the *Firstname,Lastname,Address,Pincode,Dateofbirth* are all the variables of *person structure* with different datatype declarations, where *DisplayInfo* is a method to disputably the information.

Variables holds the data and methods operates on the data stored in it as in the normal class object.

Usage of the structure:-

VB.Net

```
'Creating an instance of the structure like a normal class variable
Dim varPerson As Person = New Person()
'Setting the structure variables with the values
varPerson.firstname = “Rama”
varPerson.lastname = “Lakan”
varPerson.address = “Mysore”
varPerson.pincode = “570002”
varPerson.dateofbirth = “25/06/1977”
'Calling the strcuture method to manipulate and display the person address
varPerson.displayinfo()
```

C#

```
Person varPerson =New Person();
'Setting the structure variables with the values
varPerson.firstname = “Rama”;
varPerson.lastname = “Lakan”;
varPerson.address = “Mysore”;
varPerson.pincode = “570002”;
varPerson.dateofbirth = “25/06/1977”;
'Calling the strcuture method to manipulate and display the person address
varPerson.displayinfo();
```

6.13 Sample Application: OOPS

Heres a small application created to demonstrate Object Oriented / Component Oriented features in .NET using C#.

Accounts are created & managed by the application for a Bank. There are 2 types of account that can be created

1. Savings
2. Current

- The opening balance for Saving account is 1000, & for Current is 5000
- The minimum balance for Saving Account is 1000 & that for Current should not be less than the ODA.
- Account ID should be auto generated, Name can be modified & Balance can be updated only through transactions.

Based on the above requirement, Account component library has been built & a sample test application

Note: This is a simple application to demonstrate OOPS/Component oriented features.

1. Bank project is a Class Library. **Click here**.
2. BankClient is a Windows Application. **Click here**.

7. Error and Exception Handling

Section Owner: Saurabh Nandu (MVP)
Content Contributors: Aravind Corera (MVP)

7.1 Need for Error Handling

Picture this: You're doing a demo of your application for a client and then all of a sudden as if proving Murphy's Law, the inevitable happens. Boom! … Your client gets to see a dialog with cryptic messages indicating that your application has crashed. You certainly wish that such a thing should never ever happen, but that wish would hold true only in an ideal world. The truth however is that unforeseen errors are bound to happen, one way or another in spite of careful coding accompanied with fairly rigorous testing. Keeping this in mind, we as developers need to adopt defensive coding strategies with effective error handling techniques to trap and handle errors and exceptions, and to provide the user with adequate information on the nature and cause of the error before exiting from the application when an unexpected error occurs.

An Exception is an abnormal/exceptional/unexpected condition that disrupts the normal execution of an application. A distinction needs to be made between expected conditions and unexpected conditions. Say for example you are writing a very large file to the hard-disk, it's imperative that you should first check the disk space first, since there could be an expected condition that the hard-disk might be low on space, such conditions are not ideal candidates that warrant exceptions to be thrown. But on the contrary, what if an unexpected hard-disk hardware failure occurred while writing to the file - This is an ideal candidate that warrants exceptions to be thrown. Always remember that though exception handling is very important, judicious use of this feature should be made since there is a performance cost when you use it.

Error and exception handling are a part and parcel of every good language, framework class library, or application and should form a part of your applications too right from the planning stages. The .NET Framework Class Library (FCL) and Common Language Runtime (CLR) provide a rich infrastructure to catch and handle exceptions. The CLR infrastructure is also designed for cross-language exception handling - What this means is that if a VB.NET component throws back an exception to a C# application that's consuming the component, the C# application will be able to catch the error and obtain rich error information on the error that has occurred. Through the rest of the tutorial, we'll see how to catch and handle exceptions in the .NET framework using C# and VB.NET.

7.2 Old-school unstructured exception handling in VB 6.0 and its disadvantages

You are well aware that the exception-handling model in VB 6.0 required you to jump through hoops in order to handle errors and gracefully exit from a procedure or function. The exception-handling model that VB 6.0 supported was unstructured and required you to make jumps to a label or a line number that contained the error-handling code to handle the error. Further more, you had to take precautions to exit the procedure in a normal execution flow (when no error occurs) by not allowing it to fall through to the error-handling block. If you had clean up code that was supposed to execute both when an error occurred as well as during a normal execution flow, then you had to make yet another jump to a label or line number that contains the block of cleanup code. The whole approach was unstructured and often got very messy. To see what we mean, and why this method of exception handling can be best termed as unstructured, recollect fond nostalgic memories at how you typically handled exceptions in VB 6.0 using the code snippet shown below, and be prepared to bid goodbye to this pattern of exception handling:

```
Private Sub BookTicketsForMovie()

    On Error GoTo BookTickets_ErrHandler

    ' Perform business logic to book tickets.
    ' Possibility that exceptions could be raised here, say when all
    ' tickets are sold out.

BookTickets_Exit:

    ' Ignore any errors here since this is just a resource clean up block
    On Error Resume Next

    ' Resource clean up occurs here
    ' Maybe close up the DB Connection to the movie ticket database and so on

    'Exit the procedure
    Exit Sub

BookTickets_ErrHandler:

    ' Handle the error based on the error type.
    ' Need to pick up the value of Err.Number and then decide accordingly
    ' how to handle the error.

    MsgBox "Error encountered is: " & Err.Number
    MsgBox "Error message is: " & Err.Description

    ' Go through the usual cleanup procedure before exiting the procedure
    Resume BookTickets_Exit

End Sub
```

As seen above, you typically use an *On Error GoTo ...* statement to redirect any errors to error-handling code marked by a label or line number, such as the *BookTickets_ErrHandler*

handler in our example. The VB 6.0 runtime populates the *Err* object that contains all the details about the error that occurred, whose properties can be examined for the error type that occurred and then a decision can be taken as to how to handle that error. This is a really cumbersome task if your program is capable of generating a lot of error types, since we need through the rigmarole of determining the exact type of error that occurred using decision statements such as *Select…Case* or *If…Then…Else* before actually handling it. If you had cleanup code that needs to be executed before exit from the procedure, such as the one under the *BookTickets_Exit* label in our example, then you had to make an additional jump to the cleanup code or call a common cleanup procedure. As a result, the code becomes very unstructured and difficult to maintain, thus allowing occasional bugs to creep in silently. VB.NET supports this form of unstructured exception handling too. But it's generally not recommended that you use unstructured exception handling in VB.NET because of performance constraints and code maintenance nightmares.
It is recommended that use VB.NET's structured approach to handling errors, which is the topic of our next section. So be prepared to say a tearful goodbye to unstructured exception handling and say a big hello to the über powerful world of structured exception handling.

7.3 Structured Exception Handling in C#/VB.NET

Structured Exception Handling (SEH) allows you enclose code that can possibly encounter errors or raise exceptions within a protected block. You can define exception handler filters that can catch specific exception types thrown by the code within the protected block. Lastly, you can create a block of cleanup code that guarantees to execute always – both when an exception is thrown as well as on a normal execution path.

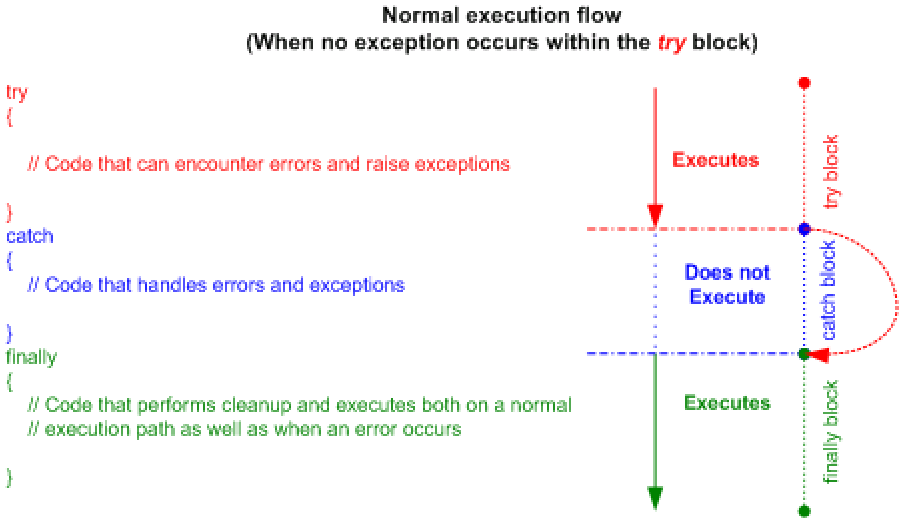
C# and VB.NET supports the following SEH keywords to raise and handle exceptions:

- ***try***
- ***throw***
- ***catch***
- ***finally***

To facilitate structured exception handling in C#/VB.NET, you typically enclose a block of code that has the potential of throwing exceptions within a *try* block. A *catch* handler associated with the exception that is thrown catches the exception. There can be one or more *catch* handlers and each catch handler can be associated with a specific exception type that it can handle. The *catch* blocks are generally used to gracefully inform the user of the error and to possibly log the error to a log file or to the system event log. You can optionally have a *finally* block that contains code that will execute both when an execution is thrown as well as on a normal execution flow. In other words, code within the *finally* block is guaranteed to always execute and usually contains cleanup code. So how can you raise an exception? . To do that your code needs to throw an exception using the *throw* keyword. The exception that is thrown is an object that is derived from the ***System.Exception*** class. We'll examine this class in detail in the next section.

Now let's consider a normal execution flow in a program where no error occurs within the *try* block.

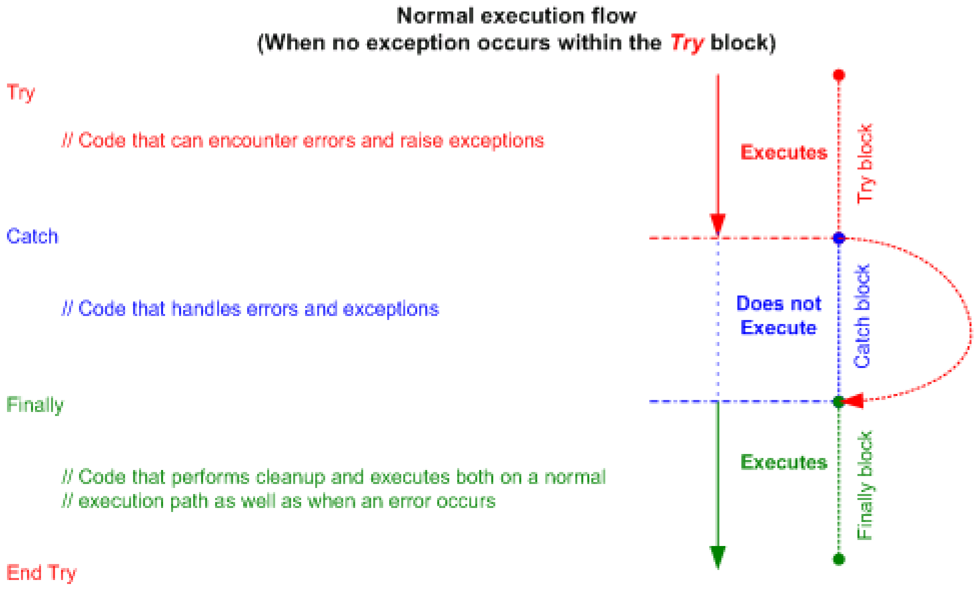
Explanation in C#



Normal execution flow
(When no exception occurs within the *try* block)

```
try
{

   // Code that can encounter errors and raise exceptions

}
catch
{
   // Code that handles errors and exceptions

}
finally
{
   // Code that performs cleanup and executes both on a normal
   // execution path as well as when an error occurs

}
```

Explanation in VB.NET



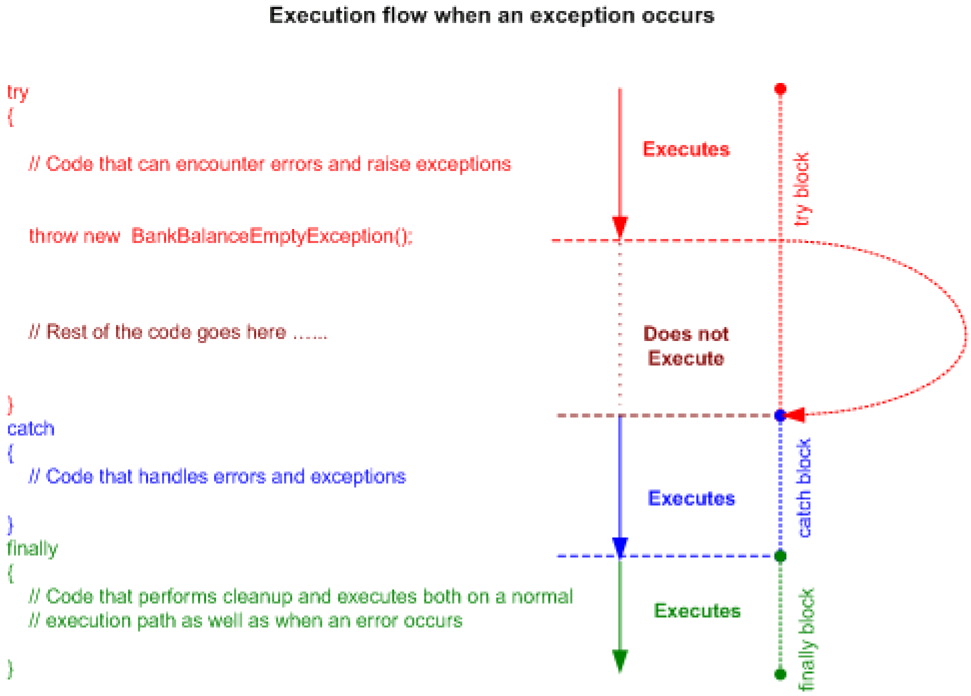
Normal execution flow
(When no exception occurs within the *Try* block)

```
Try

        // Code that can encounter errors and raise exceptions



Catch


        // Code that handles errors and exceptions



Finally


        // Code that performs cleanup and executes both on a normal
        // execution path as well as when an error occurs


End Try
```

In this case, the code within the *try* block does not throw an exception and therefore, the code within the *catch* block never executes. Once the code within the *try* block completes, the execution path resumes by executing the code in the *finally* block. As mentioned earlier, the code within the *finally* block executes whether or not an error had occurred.

Let’s turn our attention to a scenario where the code within the *try* block raises an exception.

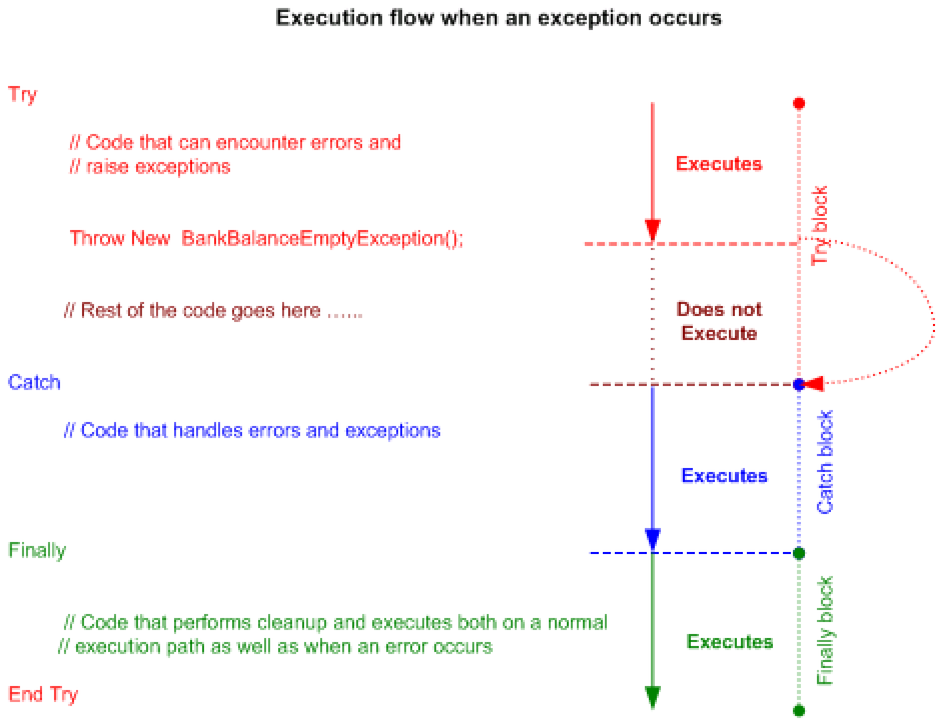
Explanation in C#



Execution flow when an exception occurs

```
try
{
    // Code that can encounter errors and raise exceptions

    throw new  BankBalanceEmptyException();


    // Rest of the code goes here ......


}
catch
{
    // Code that handles errors and exceptions

}
finally
{
    // Code that performs cleanup and executes both on a normal
    // execution path as well as when an error occurs

}
```

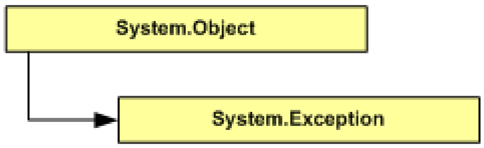
Explanation in VB.NET


Execution flow when an exception occurs
Try
// Code that can encounter errors and
// raise exceptions
Throw New BankBalanceEmptyException();
// Rest of the code goes here ......
Catch
// Code that handles errors and exceptions
Finally
// Code that performs cleanup and executes both on a normal
// execution path as well as when an error occurs
End Try
Executes
Does not
Execute
Executes
Executes
Try block
Catch block
Finally block

When the exception is thrown, execution flow is transferred to the *catch* block that is capable of handling the exception thus skipping the rest of the code below the line that threw the exception in the *try* block. The code in the *catch* block handles the exception appropriately and then execution flow moves to the code in the *finally* block.

7.4 System.Exception: The mother of all exceptions

Let's take some time to look at the parent exception class in the .NET framework – the *System.Exception* class. All exception classes are directly or indirectly derived from this class.



To see how to put this class to use, let's quickly dive into a simple example where we can see all the exception handling constructs (*try*, *catch*, *throw*, and *finally*) in action. To start with fire up your favorite text editor (my personal favorite is Notepad) and type in the following code:

Code listing in C#

```
using System;

class HelloWorld
{
   static void Main(string[] args)
   {
      try
      {
         Console.WriteLine("Here we go...");

         // Throw an exception
         throw new Exception("Oops !. Your computer is on fire !!");

         // This line should never execute
         Console.WriteLine("How on earth did I get called ?");
      }
      catch(System.Exception ex)
      {
         // Display the error message
         Console.WriteLine("Caught exception : {0}", ex.Message);
      }
      finally
      {
         // This should always get called
         Console.WriteLine("In finally");
      }
   }
}
```

Code listing in VB.NET

```
Imports System

Module HelloWorld

   ' The Main entry point of the application
   Sub Main()

      Try

         Console.WriteLine("Here we go...")

         ' Throw an exception
         Throw New Exception("Oops !. Your computer is on fire !!")

         ' This line should never execute
         Console.WriteLine("How on earth did I get called ?")

      Catch ex As Exception

         Console.WriteLine("Caught exception : {0}", ex.Message)

      Finally

         ' This should always get called
         Console.WriteLine("In finally")

      End Try
   End Sub

End Module
```

So what we have here is a *try* block that throws an exception by creating an instance of the *System.Exception* class with a descriptive error message. There's a *catch* block to handle exceptions of type *System.Exception*. The code in the *catch* block just displays the error message. Finally, the *finally* block (no pun intended) logs a message that confirms that it did execute even though an error was thrown.

So let's save this program to a file named *HelloWorld* (with the appropriate extension depending on the language you are using - .cs or .vb), the Hello World of Exception handling if you will.

To compile the C# program, type in the following command from the DOS command line:

```
csc  /target:exe HelloWorld.cs
```

To compile the VB.NET program, type in the following command from the DOS command line:

```
vbc  /target:exe HelloWorld.vb
```

This will generate an executable named *HelloWorld.exe*. Run the program and here's the output that you get:

```
Here we go...
Caught exception : Oops !. Your computer is on fire !!
In finally
```

You'll notice that the exception that was thrown is caught by the *catch* block and that any statements that occur in the *try* block below the line that threw the exception are not executed. Notice that when you compiled the program in C#, the compiler generated the following warning:

Warning in C#

```
HelloWorld.cs(16,4): warning CS0162: Unreachable code detected
```

This goes to show that the C# compiler detected that the statement that follows the *throw* statement in the *try* block would never get executed because of the exception that was thrown, and thus warned us of unreachable code.

We saw how the *Message* property of the *System.Exception* class can be used to get a descriptive error message for the exception. Similarly, let's examine the some of the other important properties of the *System.Exception* class. Let's start by modifying the *catch* block in example that we just saw with the following code:

Code listing in C#

```
// Replace the catch block in our previous example with the following code

catch(System.Exception ex)
{
   Console.WriteLine("Caught exception : {0}", ex.Message);
   Console.WriteLine("Source of the exception is : {0}", ex.Source);
   Console.WriteLine("Method that threw the exception is : {0}",
                     ex.TargetSite.Name);
   Console.WriteLine("Info on this exception is available at : {0}",
                     ex.HelpLink);
   Console.WriteLine("Stack trace of this exception: {0}", ex.StackTrace);
}
```

Code listing in VB.NET

```
' Replace the catch block in our previous example with the following code

Catch ex As Exception

   Console.WriteLine("Caught exception : {0}", ex.Message)
   Console.WriteLine("Source of the exception is : {0}", ex.Source)
   Console.WriteLine("Method that threw the exception is : {0}", _
                      ex.TargetSite.Name)
   Console.WriteLine("Info on this exception is available at: {0}",ex.HelpLink)
   Console.WriteLine("Stack trace of this exception: {0}",ex.StackTrace)

Finally

   ' Rest of the code goes here . . .
```

Compile and run the application and observe the output that you get:

Output in C#

```
Here we go...
Caught exception : Oops !. Your computer is on fire !!
Source of the exception is : HelloWorld
```

```
Method that threw the exception is : Main
Info on this exception is available at:
Stack trace of this exception:    at HelloWorld.Main(String[] args)
In finally
```

Output in VB.NET

```
Here we go...
Caught exception : Oops !. Your computer is on fire !!
Source of the exception is : HelloWorld
Method that threw the exception is : Main
Info on this exception is available at:
Stack trace of this exception:    at HelloWorld.Main()
In finally
```

You'll notice that you can get rich information on the exception that occurred including details on the application and the method that threw the exception (through the *Source* and *TargetSite* properties respectively) and a complete stack trace of the exception in a string representation (using the *StackTrace* property). Note that if you compile your code in debug mode i.e. using the */debug+* compiler option, the *Source* and *StackTrace* properties will show you the actual line numbers in the source code which raised the exception. You'll notice that the *HelpLink* property, which is supposed to provide a link to help file or a URL that contains information on the exception that occurred, does not seem to return anything. This is because we did not set this property when throwing the exception. To do that you simply need to set the *HelpLink* property before raising the exception. Here's a snippet of code that shows how you can do that:

Code listing in C#

```
// Create an Exception
Exception exception = new Exception("Oops !. Your computer is on fire !!");

// Set the help file details
exception.HelpLink = "http://www.someurl.com/help/ComputerOnFireHelp.html";

// Throw the exception
throw exception;
```

Code listing in VB.NET

```
' Create an Exception
Dim excep as Exception = New Exception("Oops !. Your computer is on fire !!")

' Set the help file details
excep.HelpLink = "http://www.someurl.com/help/ComputerOnFireHelp.html"

' Throw the exception
```

```
Throw excep
```

Replacing the statement that throws the exception with the above 3 statements in our example application, compiling it, and running it will now yield the following results:

Output in C#

```
Here we go...
Caught exception : Oops !. Your computer is on fire !!
Source of the exception is : HelloWorld
Method that threw the exception is : Main
Info on this exception is available at: http://www.someurl.com/help/ComputerOnFireHelp.html
Stack trace of this exception:    at HelloWorld.Main(String[] args)
In finally
```

Output in VB.NET

```
Here we go...
Caught exception : Oops !. Your computer is on fire !!
Source of the exception is : HelloWorld
Method that threw the exception is : Main
Info on this exception is available at: http://www.someurl.com/help/ComputerOnFireHelp.html
Stack trace of this exception:    at HelloWorld.Main()
In finally
```

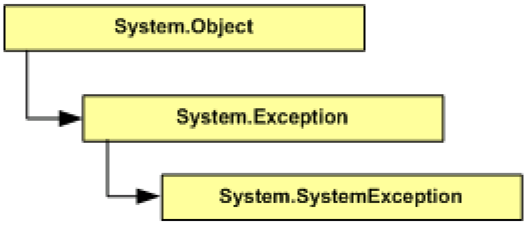
There's one other thing that you need to be aware of - The notion of an inner exception, that you can access using the *InnerException* property of the main exception. So what exactly is an inner exception? . Assume that you have a nice cool stock portal that allows customers to manage their stocks and investments. The stock portal uses a database to store data on customers and their portfolio. Now, let's say that you encounter a database specific error in your application. The last thing that you want to do is to display some cryptic ADO or OLEDB messages in your web pages that your customers don't care a hang about. In such cases, you might have a catch handler to catch database specific exceptions. What this catch handler would essentially do is to create a more generic exception that is application specific (maybe an exception that tells the user that the site encountered an internal error) and would assign the database specific exception to the application-specific exception's *InnerException* property. The *catch* handler then re-throws this application-specific exception expecting that one of the outer *catch* blocks will handle the generic exception. We'll see how to re-throw exceptions in the section, *Nesting try/catch/finally blocks and re-throwing exceptions.* Inner exceptions are very useful when you are dealing with exceptions that occur in multiple tiers of typical enterprise applications. This allows you to envelope specific exceptions that actually caused the error into more application-specific exception types, and at the same time allows clients to determine the specific exception type (*InnerException*) that caused the application-specific exception to be thrown.

Now since we know more about the *System.Exception* class, let's take a look at the types of exceptions and how they can be classified. Broadly, there are two types of exceptions:

- ❑ System exceptions (Exception classes derived from ***System.SystemException***)
- ❑ Application exceptions (Exception classes derived from ***System.ApplicationException***)

Understanding system exceptions:

System exceptions are pre-defined exceptions that ship with the .NET framework class library. For example, the *System.IO.IOException* class, which is predefined exception in the framework class library for handling input/output related errors on files, streams etc., is derived from the *System.SystemException* class.



There are tons of other similar predefined system exception classes that are defined and used in the FCL and which can be used in our applications as well. Let's take a look at a quick example on how to handle system exceptions in your application. We'll use the *System.DivideByZeroException* as our guinea pig here and simulate a situation where the FCL throws this exception. We'll handle this error and report the error to the user. Fire up Notepad, and type in the following code:

Code listing in C#

```
using System;

class MyDecimalDivider
{
   static void Main(string[] args)
   {
      try
      {
         // Trigger a divide by zero exception
         Decimal dResult = Decimal.Divide(5,0);

         // We should never get here
         Console.WriteLine("Result is : {0}", dResult);

      }
      catch(DivideByZeroException exDivByZero)
      {
          Console.WriteLine("Caught Divide By Zero exception: {0}",
                            exDivByZero.Message);
      }
      catch(Exception ex)
      {
         Console.WriteLine("Caught exception: {0}",ex.Message);
      }
      finally
      {
         // Should always execute
         Console.WriteLine("In finally");
      }
   }
}
```

Code listing in VB.NET

```
Imports System

Module MyDecimalDivider

   Sub Main()

      Try

         ' Trigger a divide by zero exception
         Dim dResult as Decimal = Decimal.Divide(5,0)

         ' We should never get here
         Console.WriteLine("Result is : {0}", dResult)

      Catch exDivByZero As DivideByZeroException

         Console.WriteLine("Caught Divide By Zero exception: {0}", _
                           exDivByZero.Message)

      Catch ex As Exception

         Console.WriteLine("Caught exception: {0}", ex.Message)

      Finally

         ' Should always execute
         Console.WriteLine("In finally")

      End Try

   End Sub

End Module
```

So essentially, what we're doing here is simulating a *DivideByZeroException* by calling the *Divide()* static method of the *System.Decimal* class and passing in a value of 0 for the divisor. A quick look at the documentation for the *Divide()* method will tell you that the method throws a *DivideByZeroException* when attempting to divide by 0. So we're setting up a *catch* block to handle exceptions of type *System.DivideByZeroException*.

Save the file to *MyDecimalDivider (with the appropriate extension .cs or .vb depending on the language that you are using)*. Let's get down to compiling the application. Type the following command from the DOS command prompt:

Compiling in C#

```
csc  /target:exe MyDecimalDivider.cs
```

Compiling in VB.NET

```
vbc  /target:exe MyDecimalDivider.vb
```

That takes care of generating an executable file named *MyDecimalDivider.exe*. Run the program and observe the output:

```
Caught Divide By Zero exception: Attempted to divide by zero.
In finally
```

There we go. As seen above, the *catch* handler for the *DivideByZeroException* took care of catching the exception that was raised when we attempted to divide 5 by 0. The *System.DivideByZeroException* is just one of the many predefined system exception classes in the FCL. For a complete list of the other system exception classes, swing by to: http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemsystemexceptionclasshierarchy.asp

Ordering catch handlers to filter exceptions:

Notice that we also have another *catch* block that handles the generic *System.Exception*. If none of the other *catch* handlers can handle an exception raised, the *System.Exception* catch handler will always lend a helping hand in catching and handling the exception, since the rest of the exception types are derived from this class. So that brings us to another thing that you need to remember - If you do not have a *catch* handler to handle a specific exception type, say *SomeException*, but do have a *catch* hander that can handle a type that is a super class of *SomeException*, then the *catch* handler associated with that super class will be asked to handle the exception. In our example, even if we did not have the *catch* handler for the *System.DivideByZeroException*, the *catch* handler for the *System.Exception* would have been able to handle the exception, since *System.DivideByZeroException* inherits from *System.ArithmeticException*, which in turn derives from *System.SystemException* and hence *System.Exception*.

Keeping this in mind, it is important to understand that the order in which you place your *catch* handlers plays a key role in determining the *catch* handler that will eventually handle your exceptions. As a general rule, always place exception types of more derived classes in an exception class hierarchy higher up in the chain and place base class (super class) exception types lower down in the chain. To illustrate this, let's slightly modify the earlier divide by zero example and note down a few observations:

Modify the *MyDecimalDivider.cs* code sample as shown below to introduce a *catch* handler for the *System.ArithmeticException*, which is the immediate base class of the *System.DivideByZeroException* and place that *catch* handler above the *catch* handler that handles the *DivideByZeroException*:
Code listing in C#

```
using System;

class MyDecimalDivider
{
   static void Main(string[] args)
   {
      try
      {
         // Trigger a divide by zero exception
         Decimal dResult = Decimal.Divide(5,0);

         // We should never get here
         Console.WriteLine("Result is : {0}", dResult);
      }
      catch(ArithmeticException exArithmetic)
      {
         Console.WriteLine("Caught Arithmetic exception: {0}",
                           exArithmetic.Message);
      }
      catch(DivideByZeroException exDivByZero)
      {
         Console.WriteLine("Caught Divide By Zero exception: {0}",
                           exDivByZero.Message);
      }
      catch(Exception ex)
      {
         Console.WriteLine("Caught exception: {0}", ex.Message);
      }
      finally
      {
         // Should always execute
         Console.WriteLine("In finally");
      }
   }
}
```

Code listing in VB.NET

```
Imports System

Module MyDecimalDivider

   Sub Main()

      Try

         ' Trigger a divide by zero exception
         Dim dResult as Decimal = Decimal.Divide(5,0)

         ' We should never get here
         Console.WriteLine("Result is : {0}", dResult)

      Catch exArithmetic As ArithmeticException

         Console.WriteLine("Caught Arithmetic exception: {0}", _
                           exArithmetic.Message)

      Catch exDivByZero As DivideByZeroException

         Console.WriteLine("Caught Divide By Zero exception: {0}", _
                           exDivByZero.Message)

      Catch ex As Exception

         Console.WriteLine("Caught exception: {0}", ex.Message)

      Finally

         ' Should always execute
         Console.WriteLine("In finally")

      End Try

   End Sub

End Module
```

Now save the file and compile the modified *MyDecimalDivider.cs/ MyDecimalDivider.vb* in the DOS command line using:

Compiling in C#

```
csc  /target:exe MyDecimalDivider.cs
```

Compiling in VB.NET

```
vbc  /target:exe  MyDecimalDivider.vb
```

Run the application *MyDecimalDivider.exe* and observe the output:

Output in C#

```
MyDecimalDivider.cs(21,9): error CS0160: A previous catch clause already catches all exceptions of this or a super type
('System.ArithmeticException')
```

The error says it all. The *ArithmeticException* catch handler has been placed above the *catch* handler that handles exception types of its subclass *DivideByZeroException*, which effectively hides the *catch* handler for the *DivideByZeroException*.

Output in VB.NET

```
Caught Arithmetic exception: Attempted to divide by zero.

In finally
```

Since *ArithmeticException*'s *Catch* handler has been placed above the *Catch* handler for its subclass exception type *DivideByZeroException,* the *Catch* handler for the *ArithmeticException* is asked to handle the error even though the actual exception type that was raised was *DivideByZeroException.*

You'll observe the same behavior if you place the *System.Exception Catch* handler above any of the other catch handlers. In order to give *DivideByZeroException*'s *Catch* handler the opportunity to handle the error, place it above the *Catch* handler that handles *ArithmeticException* exceptions (*DivideByZeroException'*s super class). To summarize, place *Catch* handler filters for specific exception types (sub classes) higher than the handlers for the more generic exception types (base classes).

Code listing in C#

```
// Rest of the code omitted for brevity . . .

catch(DivideByZeroException exDivByZero)
{
   Console.WriteLine("Caught Divide By Zero exception: {0}",
                     exDivByZero.Message);
}
catch(ArithmeticException exArithmetic)
{
   Console.WriteLine("Caught Arithmetic exception: {0}", exArithmetic.Message);
}
catch(Exception ex)
{
   Console.WriteLine("Caught exception: {0}", ex.Message);
}
```

Code listing in VB.NET

```
Try

// Rest of the code omitted for brevity . . .

Catch exDivByZero As DivideByZeroException

   Console.WriteLine("Caught Divide By Zero exception: {0}",  _
                     exDivByZero.Message)

Catch exArithmetic As ArithmeticException

   Console.WriteLine("Caught Arithmetic exception: {0}", exArithmetic.Message)

Catch ex As Exception

   Console.WriteLine("Caught exception: {0}", ex.Message)

Finally

   ' Should always execute
   Console.WriteLine("In finally")

End Try
```

Now let's try one more thing. We'll remove the *catch* handler for the *DivideByZeroException*, just leaving behind the catch handlers for the *System.ArithmeticException* and the *System.Exception* classes.

Go ahead and modify the *MyDecimalDivider* code by commenting out the *catch* handler for the *DivideByZeroException*. Compile and run the application. What output do you see this time?

```
Caught Arithmetic exception: Attempted to divide by zero.
In finally
```

As shown above, though there was no *catch* handler for the *DivideByZeroException* that was raised, the *catch* handler for *ArithemeticException* was able to *catch* the exception since the *ArithemeticException* class happens to be a base class of the *DivideByZeroException* class. Similarly, even if we didn't have the *catch* handler for the *ArithmeticException* class, the *System.Exception catch* handler would have still caught the exception (since it's the parent class for all exception types). So what happens if a *DivideByZeroException* is raised and you don't have any of the *catch* handlers (not even the *System.Exception catch* handler)? . You guessed right – the exception would turn into an unhandled exception crashing your application. Try this by removing the *try* block and all the *catch-finally* handlers and call *Decimal.Divide()* by passing a value of 0 for the divisor and notice what happens:

Output in C#

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
   at System.Decimal.Divide(Decimal d1, Decimal d2)
   at MyDecimalDivider.Main(String[] args)
```

Output in VB.NET

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
   at System.Decimal.Divide(Decimal d1, Decimal d2)
   at MyDecimalDivider.Main()
```

7.5 Handling exceptions that are not System.Exception compliant

What happens when your managed .NET code interacts with legacy libraries that are .NET agnostic. In general cases, when you interact with unmanaged code using the Platform Invoke (P/Invoke) or COM Interoperability mechanisms provided by the .NET framework, an exception raised from unmanaged code would be mapped back by the CLR into an appropriate .NET exception type. However, there are cases where legacy unmanaged libraries could possibly raise exceptions that are not *System.Exception* compliant, which cannot be mapped to a corresponding .NET exception type. In such cases, you can use a generic exception handler that can catch errors that are not .NET aware and not compliant with *System.Exception*. The generic catch handler contains only the *catch* keyword and does not specify an exception type filter. Here's an example code snippet with a generic catch handler:

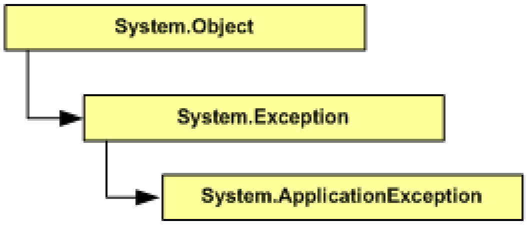
Code listing in C#

```
try
{
   // This is the try block
}
catch(Exception ex)
{
   // This is the catch block to handle System.Exception errors
}
catch
{
   // This is a generic catch block to handle calls to libraries that raise
   // exceptions which are not compliant with System.Exception. Can catch
   // any error that the other catch handlers cannot handle.
}
finally
{
   // This is the finally block
}
```

Code listing in VB.NET

```
Try

    ' This is the Try block

Catch ex As Exception

    ' This is the catch block to handle System.Exception errors

Catch

   ' This is a generic catch block to handle calls to libraries that raise
   ' exceptions which are not compliant with System.Exception. Can catch
   ' any error that the other catch handlers cannot handle.

Finally

  ' This is the finally block

End Try
```

7.6 Understanding Application exceptions (user-defined or custom exceptions)



Though the FCL supports a great deal of predefined system exception classes (as seen in the earlier section) that can be used to represent a large gamut of errors, there is always a need to model custom errors that represent failed run-of-the-mill business logic as well as other application specific error scenarios. In these situations, you need to turn to defining your own custom exceptions that are application specific. When defining application specific custom exceptions, you need to typically create a class that is derived from the *System.ApplicationException* class.

It is good practice and generally recommended that the application exception class be suffixed with *Exception*. For example, if you need to define an exception that indicates that a specific ticket to a movie is not available, you'd probably want to name it something like *TicketNotAvailableException*.

So let’s put this in practice and see how to define and use a custom application exception. The example we’ll take up is a class that simulates a television channel changer, which allows the user to surf television channels. We’ll assume that one of our business logic constraints is that we support only 80 channels and that the class is expected to flip channels only if the user enters a channel number between 1 and 80. If the user enters an invalid channel number, the class is expected throw an application exception, which indicates that the channel number entered is invalid. So let’s put together this application-specific exception class.

We’ll call the exception class *ChannelNotAvailableException* and derive this class from the *System.ApplicationException* class to indicate that this is an application specific exception. Next, we’ll have to create some constructors for this class. The best practice guidelines for exception handling recommend that we have two constructors in addition to the default no-argument constructor - One constructor that accepts the error message as a parameter and the other one that accepts both an error message and an inner exception as a parameter. Let’s take a look at the *ChannelNotAvailableException* class.

Code listing in C#

```
class ChannelNotAvailableException : System.ApplicationException
{
   public ChannelNotAvailableException()
   {
   }

   public ChannelNotAvailableException(String errorMessage) :
                                                       base(errorMessage)
   {
   }

   public ChannelNotAvailableException(String errorMessage,
          Exception innerException)  : base(errorMessage, innerException)
   {
   }
}
```

Code listing in VB.NET

```
Public Class ChannelNotAvailableException
     Inherits ApplicationException

   ' Default Constructor
   Public Sub New()

      ' Call the base class constructor
      MyBase.New()

   End Sub


   ' Constructor that takes message string
   Public Sub New(ByVal errorMessage As String)

      ' Call the base class constructor
      MyBase.New(errorMessage)

   End Sub


   ' Constructor that takes message string and inner exception
   Public Sub New(ByVal errorMessage As String,  _
                  ByVal innerException As Exception)

      ' Call the base class constructor
      MyBase.New(errorMessage, innerException)

   End Sub

End Class
```

The *ChannelNotAvailableException* class shown above is fairly trivial and you'll notice that the non-default constructors do nothing more than initializing their corresponding base class counter parts through the base class argument-list initializer. That's it – we're done setting up our custom exception class. We'll see how to put this to use in our TV channel surfer application. Let's put together some code for the channel surfer class.

Code listing in C#

```
class ChannelSurfer
{

   private const int MAX_CHANNELS = 80;
   private int m_nCurrentChannel;

   ChannelSurfer()
   {
      // Set channel 1 as the default
      m_nCurrentChannel = 1;
   }

   public int CurrentChannel
   {
      get
      {
         // Return the current channel
         return m_nCurrentChannel;
      }
   }

   // Rest of the class implementation goes here . . .
}
```

Code listing in VB.NET

```
Public Class ChannelSurfer

   Private Const MAX_CHANNELS As Integer = 80
   Private m_nCurrentChannel As Integer

   Public Sub New()

      MyBase.New()

      ' Set channel 1 as the default
      Me.m_nCurrentChannel = 1

   End Sub


   ReadOnly Property CurrentChannel() As Integer

      Get
         ' Return the current channel
         Return Me.m_nCurrentChannel
      End Get

   End Property


   ' Rest of the class implementation goes here . . .


End Class
```

The channel surfer class supports a read-only property named *CurrentChannel* that keeps track of the current channel being viewed. The viewer can move between channels by calling the *FlipToChannel()* method shown below:

Code listing in C#

```
class ChannelSurfer
{

   // Rest of the class implementation goes here . . .

   void FlipToChannel(int nChannelNumber)
   {
      if( (nChannelNumber < 1) ||  (nChannelNumber > MAX_CHANNELS))
      {
         throw new ChannelNotAvailableException("We support only 80 channels."
                     +  "Please enter a number between 1 and 80");
      }
      else
      {
         // Set the value of the current channel
         m_nCurrentChannel = nChannelNumber;
      }
   }

}
```

Code listing in VB.NET

```
Public Class ChannelSurfer

   ' Rest of the class implementation goes here . . .

   Sub FlipToChannel(ByVal nChannel As Integer)

      If ((nChannel < 1) Or (nChannel > MAX_CHANNELS)) Then

        ' Raise an exception
        Throw New ChannelNotAvailableException("We support only 80 channels." _
                 +  "Please enter a number between 1 and 80")
      Else

         ' Set the current channel
         Me.m_nCurrentChannel = nChannel

      End If

   End Sub

End Class
```

As seen above the *FlipToChannel()* method checks to see if the channel number that the user is requesting is between 1 and 80 and if so, sets the value of the *CurrentChannel* property to the requested channel. If the values are not within the specified range, the class throws the user-defined *ChannelNotAvailableException* exception. Let's use the application's *Main()* entry point as a test harness for the *ChannelSurfer* class. Take a look at the code below:

Code listing in C#

```
class ChannelSurfer
{

   // Rest of the class implementation goes here . . .

   static void Main(string[] args)
   {
      ChannelSurfer channelZapper = new ChannelSurfer();

      // Display a message
      Console.WriteLine("Press 'Q' or 'q' to quit zapping channels");

      // Set up an infinite loop
      for(;;)
      {
         try
         {
            // It's channel surfing time folks!
            Console.Write("Please enter a channel number and press 'Enter'");

            // Get the channel number from the user
            String strChannel = Console.ReadLine();

            // Check if the user wants to quit
            if(strChannel.Equals("Q") || strChannel.Equals("q")) break;

            // Convert the channel number to an integer
            int nChannel = Int32.Parse(strChannel);

            // Flip away to the requested channel
            channelZapper.FlipToChannel(nChannel);
         }
         catch(ChannelNotAvailableException exChannel)
         {
            Console.WriteLine("Channel not supported: {0}", exChannel.Message);
         }
         catch(FormatException exFormat)
         {
            Console.WriteLine("Caught a format exception: {0}",
                               exFormat.Message);
         }
         catch(Exception ex)
         {
            Console.WriteLine("Caught a exception: {0}", ex.Message);
         }
         finally
         {
            // What channel are we watching?
            Console.WriteLine("You are watching Channel : {0}",
                               channelZapper.CurrentChannel);
         }
      }
   }
}
```

Code listing in VB.NET

```
Module SurfChannelTestHarness

   Sub Main()

      Dim channelZapper As ChannelSurfer = New ChannelSurfer()

      ' Display a message
      Console.WriteLine("Press 'Q' or 'q' to Quit zapping Channels")


      ' Setup an infinite loop to ask the user for channel input
      Do

         Try

            ' It's channel surfing time folks !
            Console.Write("Please enter a channel number and press 'Enter' ")

            ' Get the channel number from the user
            Dim strChannel As String = Console.ReadLine()

            ' Check if the user wants to quit
            If (strChannel.Equals("Q") Or strChannel.Equals("q")) Then
               Exit Do
            End If

            ' Convert the channel number to an integer
            Dim nChannel As Integer = Int32.Parse(strChannel)
            ' Flip away to the requested channel
            channelZapper.FlipToChannel(nChannel)

         Catch exChannel As ChannelNotAvailableException

            Console.WriteLine("Channel not supported: {0}", _
                               exChannel.Message)

         Catch exFormat As FormatException

            Console.WriteLine("Caught a format exception: {0}", _
                               exFormat.Message)

         Catch ex As Exception

            Console.WriteLine("Caught a exception: {0}", ex.Message)

         Finally

            ' What channel are we watching ?
            Console.WriteLine("You are watching Channel: {0}", _
                               channelZapper.CurrentChannel)

         End Try

      Loop While True

   End Sub

End Module
```

The *Main()* entry point creates an instance of the *ChannelSurfer* class and sets up a loop that requests channel numbers from the user until the user presses the '*Q*' or '*q*' key to quit the application. When the channel number input is received, it calls the *FlipToChannel()* method of the *ChannelSurfer* object. The *FlipToChannel()* method call is enclosed within a *try* block and an appropriate *catch* handler for the *ChannelNotAvailableException* will catch the exception if an invalid channel number is passed to the *FlipToChannel()* method. Also, if the user enters non-numeric input, the *Int32.Parse()* method will throw a *System.FormatException* that will be caught by the *catch* handler that we've setup to handle *FormatException* exceptions.

Compile the file using the following command from the DOS command line:

Compiling in C#

```
csc  /target:exe ChannelSurfer.cs
```

Compiling in VB.NET

```
vbc  /target:exe ChannelSurfer.vb
```

That generates the executable file *ChannelSurfer.exe*. Run the application and feed it with input containing both valid and invalid input values. Here's a sample interaction with the *ChannelSurfer* application.

```
Press 'Q' or 'q' to quit zapping channels
Please enter a channel number and press 'Enter' 62
You are watching Channel : 62
Please enter a channel number and press 'Enter' 56
You are watching Channel : 56
Please enter a channel number and press 'Enter' 104
Channel not supported: We support only 80 channels. Please enter a number between 1
and 80
You are watching Channel : 56
Please enter a channel number and press 'Enter' abcd
Caught a format exception : Input string was not in a correct format.
You are watching Channel : 56
Please enter a channel number and press 'Enter' 15
You are watching Channel : 15
Please enter a channel number and press 'Enter' q
You are watching Channel : 15
```

You will notice from the output above that when the user enters *104* for the channel number, the *ChannelNotAvailableException* is thrown from the *FlipToChannel()* method, which is then handled by the *catch* handler. Similarly, when the user keys in a non-numeric value such as *abcd*, a *System.FormatException* is raised by the *Int32.Parse()* method, which then gets caught by the *catch* handler that filters the *FormatException* exceptions. Using application specific exceptions like *ChannelNotAvailableException* allows you to build exception-handling classes around your business-logic and application specific scenarios. Be sure to check if the framework provides a predefined exception class that suits the exception type that you want to handle. If so, reuse FCL provided system exceptions. Otherwise, feel free to model custom exception classes that are modeled around application specific exception scenarios.

7.7 Nesting try/catch/finally blocks and re-throwing exceptions

It should be noted that structured exception handling allows you to nest *try*/*catch*/*finally* blocks within one another. This allows multiple levels of nesting and if the inner *catch* blocks cannot handle a specific exception type, the runtime will look for a matching *catch* handler in one of the outer *try* blocks. This repeats until a matching *catch* handler is found in one of the enclosing blocks. If the outermost *try* block is reached and no such matching *catch* handler is found, the runtime forces an unhandled exception to be raised. Let's take a look at a quick example of how to nest *try*/*catch*/*finally* blocks within one another.

Code listing in C#

```
using System;

class HelloNested
{
   static void Main(string[] args)
   {
      try
      {
         try
         {
            throw new Exception("It's just too warm in here !");
         }
         catch(Exception exInner)
         {
            // Display the exception message
            Console.WriteLine("Inner catch caught an exception: {0}",
                               exInner.Message);
         }
         finally
         {
            // The inner finally block that executes always
            Console.WriteLine("Inner finally");
         }

         // Continue execution in the Outer try block
         Console.WriteLine("Continue executing in Outer ...");

      }
      catch(Exception exOuter)
      {
         // Display the exception message
         Console.WriteLine("Outer catch caught an exception: {0}",
                            exOuter.Message);
      }
      finally
      {
         // The outer finally block that executes always
         Console.WriteLine("Outer finally");
      }
   }
}
```

Code listing in VB.NET

```
Imports System


Module HelloNested

   Sub Main()


      ' This is the beginning of the Outer Try block
      Try

         ' This is the beginning of the Inner Try block
         Try

            Throw New Exception("It's just too warm in here !")
         Catch exInner As Exception

            ' Display the exception message
            Console.WriteLine("Inner catch caught an exception: {0}", _
                           exInner.Message)


         Finally

            ' The inner finally clause that executes always
            Console.WriteLine("Inner finally")


        ' The Inner Try/Catch/Finally blocks ends here
         End Try


        ' Continue execution in the Outer try block
        Console.WriteLine("Continue executing in Outer ...")


      Catch exOuter As Exception

         ' Display the exception message
         Console.WriteLine("Outer catch caught an exception: {0}", _
                            exOuter.Message)


      Finally

         ' The outer finally clause that executes always
         Console.WriteLine("Outer finally")


      ' The Outer Try/Catch/Finally blocks ends here
      End Try

    End Sub

End Module
```

As shown in the example above, we have an inner *try*/*catch*/*finally* triad nested within an outer *try* block. The code within the inner *try* block raises an exception, so the runtime will check to see if one of the inner *catch* handlers will be able to handle the exception. Only when none of the inner *catch* handlers can handle that exception type, will the *catch* handlers of the outer *try* block be examined if they'll be able to handle the error.

Save the example shown above in a file named *HelloNested.cs/HelloNested.vb*. Compile the application by running the following command from the DOS command line:

Compiling in C#

```
csc  /target:exe HelloNested.cs
```

Compiling in VB.NET

```
vbc  /target:exe HelloNested.vb
```

Run the program *HelloNested.exe* and observe the output:

```
Inner catch caught an exception : It's just too warm in here !
Inner finally
Continue executing in Outer ...
Outer finally
```

As seen from the output above, the inner *catch* handler catches an exception raised by the code in the inner *try* block since it can handle *System.Exception* exceptions. Now replace the inner *catch* block in the above example to handle only *System.OverflowException* exceptions.

Code listing in C#

```
catch(OverflowException exInner)
{
   // Display the exception message
   Console.WriteLine("Inner catch caught an exception: {0}", exInner.Message);
}
```

Code listing in VB.NET

```
Catch exOverflow As OverflowException

   ' Display the exception message
   Console.WriteLine("Inner catch caught an overflow exception: {0}", _
                     exOverflow.Message)
```

Compile and run the modified program *HelloNested.exe* and observe the output:

```
Inner finally
Outer catch caught an exception : It's just too warm in here !
Outer finally
```

Notice that since the inner *catch* block can handle only *System.OverflowException* exceptions the runtime looks for a matching *catch* handler in the outer blocks and locates the outer *catch* handler, which subsequently handles the *System.Exception* exception.

Until now, we've seen how exceptions are raised by code enclosed within the *try* block. But also take note that you can throw exceptions from within *catch* and *finally* blocks too. There are times when a *catch* handler catches an exception and examines it only to find that it cannot handle the exception. In such cases, the *catch* handler can re-throw the exception hoping that one of the outer *catch* handlers will be able to catch the exception and handle it appropriately. In this case, the runtime checks for a matching *catch* handler in one of the enclosing outer *catch* blocks to handle the re-thrown exception. Let's modify the earlier example to re-throw the exception that we caught in the inner *catch* block. Modify the inner *catch* block in the *HelloNested* code as shown below:

Code listing in C#

```
// Rest of the code omitted for brevity . . .

try
{
   try
   {
      throw new Exception("It's just too warm in here !");
   }
   catch(Exception exInner)
   {
      // Display the exception message
      Console.WriteLine("Inner catch caught an exception: {0}",
                        exInner.Message);

      // Rethrow the exception
      throw exInner;
   }
   finally
   {
      // The inner finally block that executes always
      Console.WriteLine("Inner finally");
   }

   // Continue execution in the Outer try block
   Console.WriteLine("Continue executing in Outer ...");
}

// Rest of the code omitted for brevity . . .
```

Code listing in VB.NET

```
' Rest of the code omitted for brevity . . .

' This is the beginning of the Inner Try block
Try

   Throw New Exception("It's just too warm in here !")

Catch exInner As Exception

   ' Display the exception message
   Console.WriteLine("Inner catch caught an exception: {0}", exInner.Message)
   ' Rethrow the exception
   Throw exOverflow

Finally

   ' The inner finally clause that executes always
   Console.WriteLine("Inner finally")

   ' The Inner Try/Catch/Finally blocks ends here
End Try

' Rest of the code omitted for brevity . . .
```

You will notice that the inner *catch* block re-throws the exception that it catches and hopes that one of the outer *catch* handlers will be able to handle it. Compile and run the application. Observe the output:

```
Inner catch caught an exception : It's just too warm in here !
Inner finally
Outer catch caught an exception : It's just too warm in here !
Outer finally
```

You'll notice that both the inner and the outer *catch* handlers have a go at handling the exception. The inner *catch* block catches the exception and re-throws it. The re-thrown exception is then subsequently caught and handled by the outer *catch* handler. Take note that you can throw exceptions from *finally* blocks too.

How the CLR uses the call-stack to locate a matching *catch* handler:

When an exception occurs, the CLR tries to locate an appropriate *catch* handler (associated with the current *try* block), which is capable of handling the exception. If it cannot find an appropriate *catch* handler, then the next outer *try-catch* block is examined for appropriate *catch* handlers. This search continues until it finds a matching *catch* handler within the scope of the currently executing method (in C#) / procedure (in VB.NET). If it still cannot find a matching *catch* handler within the scope of the currently executing method/procedure, it pops the current method/procedure out of the call-stack thus causing the current method to lose scope, and then searches for matching *catch* handlers in the next method (the method that had originally called the current method/procedure) in the call-stack. If it cannot find a matching *catch* handler there too, it pops this method/procedure out and examines the next one in the call-stack. This stack unwinding continues until a matching *catch* handler is found for the exception that was thrown. If no such matching *catch* handler is found when the stack is completely unwound, then the exception becomes an unhandled exception.

Code listing in C#

```
using System;

// Rest of the code omitted for brevity...

class Diver
{
   static void Main(string[] args)
   {
      try
      {
         Console.WriteLine("Get Set Go...");

         // Call the DiveIn() static method
         Diver.DiveIn();

      }
      catch(SharkAttackException ex)
      {
         Console.WriteLine(ex.Message);
      }
      finally
      {
         // This should always get called
         Console.WriteLine("In Main finally");
      }
   }

   static void DiveIn()
   {
      try
      {
         // Call the DiveDeeper static method
         Diver.DiveDeeper();
      }
      catch(WaterTooColdException ex)
      {
         Console.WriteLine(ex.Message);
      }
      finally
      {
         // This should always get called
         Console.WriteLine("In DiveIn finally");
      }
   }

   static void DiveDeeper()
   {

      try
      {
         throw new SharkAttackException("Two hungry Great-White sharks " +
                                         "on the prowl");
      }
      catch(OutOfOxygenException ex)
      {
         Console.WriteLine(ex.Message);
      }
      finally
      {
         // This should always get called
         Console.WriteLine("In DiveDeeper finally");
      }
   }

}
```

Code listing in VB.NET

```
Imports System

' Rest of the code omitted for brevity...


Module Diver

   Sub Main()

      Try

         Console.WriteLine("Get Set Go...")

         ' Call the DiveIn() subroutine
         Call DiveIn

      Catch ex As SharkAttackException

         Console.WriteLine(ex.Message)

      Finally

         ' This should always get called
         Console.WriteLine("In Main finally")

      End Try

   End Sub


   Sub DiveIn()

      Try

         ' Call the DiveDeeper() subroutine
         Call DiveDeeper

      Catch ex As WaterTooColdException

         Console.WriteLine(ex.Message)

      Finally

         ' This should always get called
         Console.WriteLine("In DiveIn finally")

      End Try

   End Sub


   Sub DiveDeeper()

      Try
```

```
         Throw New SharkAttackException("Two hungry Great-White sharks " + _
                                        "on the prowl")

      Catch ex As OutOfOxygenException

         Console.WriteLine(ex.Message)

      Finally

         ' This should always get called
         Console.WriteLine("In DiveDeeper finally")

      End Try

   End Sub

End Module
```

Compile the application by running the following command from the DOS command line:

Compiling in C#

```
csc  /target:exe Diver.cs
```

Compiling in VB.NET

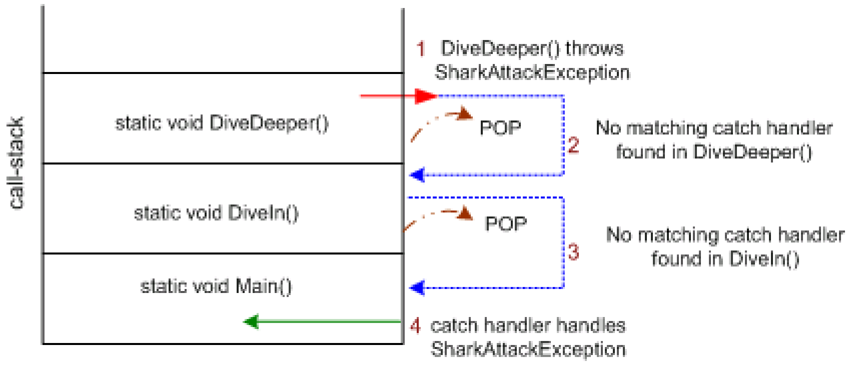
```
vbc  /target:exe Diver.vb
```

Run the program *Diver.exe* and observe the output:

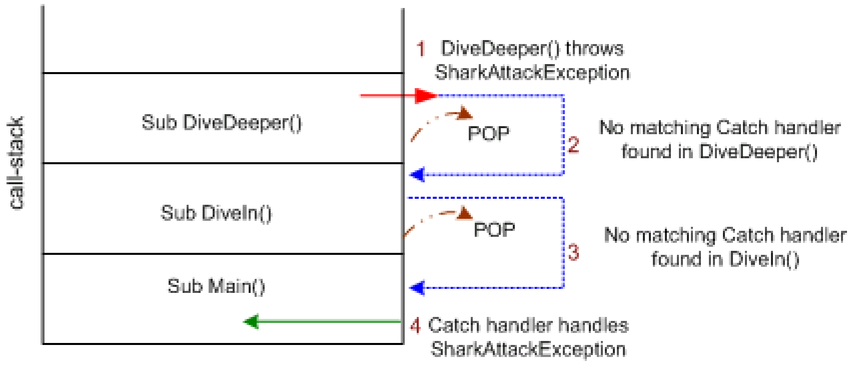
```
Get Set Go...
In DiveDeeper finally
In DiveIn finally
Two hungry Great-White sharks on the prowl
In Main finally
```

In the above code fragment, the *Main()* entry point in the program calls the *DiveIn()* method (in C#) / Subroutine (in VB.NET), which in turn calls the *DiveDeeper()* method/subroutine. Notice that the *DiveDeeper()* method throws a custom application-defined exception called *SharkAttackException*.

Explanation using C#


call-stack
static void DiveDeeper()
static void DiveIn()
static void Main()
1 DiveDeeper() throws SharkAttackException
POP
2
No matching catch handler found in DiveDeeper()
POP
3
No matching catch handler found in DiveIn()
4 catch handler handles SharkAttackException

Explanation using VB.NET


call-stack
Sub DiveDeeper()
Sub DiveIn()
Sub Main()
1 DiveDeeper() throws SharkAttackException
POP
2
No matching Catch handler found in DiveDeeper()
POP
3
No matching Catch handler found in DiveIn()
4 Catch handler handles SharkAttackException

When the *SharkAttackException* exception is thrown the CLR checks to see if the *catch* handlers associated with the *try* block in *DiveDeeper()* can handle the exception Since the only exception type handled by the *catch* handler in *DiveDeeper()* happens to be *OutOfOxygenException*, the CLR will pop the *DiveDeeper()* method out of the stack after executing the *finally* block in *DiveDeeper()*. It will then go on to search for a suitable *catch* handler in the next method/procedure in the call-stack, which happens to be the *DiveIn()* method/subroutine. Since the *try-catch* block in the *DiveIn()* method also happens to have a *catch* handler that handles only *WaterTooColdException* exceptions, this method/procedure is also popped out of the call-stack after executing its *finally* block. The *Main()* method/subroutine, which is the next in the call-stack is then examined for matching *catch* handler within the *try-catch* block. As you can see, the *try-catch* block within the *Main()* method/subroutine does have a *catch* handler that can handle the *SharkAttackException* and so control is eventually passed over to this *catch* handler after which the corresponding *finally* block is executed. Assuming that the *Main()* entry-point method/subroutine did not have an appropriate *catch* handler too, then the next pop operation would have completely unwound the call-stack thereby making the *SharkAttackException* an unhandled exception

Catching arithmetic overflow exceptions with C#'s checked keyword:

Never ever discount the destructive effects that arithmetic overflow exceptions bring to the stability of your software. For starters, recollect the tragic crash of the $7 billion Ariane 5 rocket – A crash that resulted because its software system attempted to convert a 64-bit floating-point number to a signed 16-bit integer, which subsequently caused an overflow exception, and worse yet, there was no exception handler to handle this exception. Sadly enough, the backup systems were also running on the same copy of the software, without the exception handler. Read an account of the Ariane 5 crash at:

http://www.cs.ucsd.edu/users/goguen/courses/230/ariane5nyt.pdf

Let's face it – How many times have we been in situations where we've stared in disbelief at our program spewing some insanely odd numerical output on arithmetic operations when it's fairly obvious that the output is in not even slightly connected to what was expected of the program. Let's quickly see what we mean here with an example. Consider the following C# program:

```
using System;

class ByteBites
{
   static void Main(string[] args)
   {
      byte b1 = 57;
      byte b2 = 200;
      byte bResult = (byte)(b1 + b2);

      Console.WriteLine("{0} + {1} = {2}", b1, b2, bResult);
   }
}
```

Save this program to a file called *ByteBites.cs*. Compile the program using the following command from the DOS command line:

```
csc  /target:exe ByteBites.cs
```

Run *ByteBites.exe* and observe the output:

```
57 + 200 = 1
```

Many of us know what went wrong here. It’s fairly obvious here that a *byte* data type can hold only values from 0 to 255. Yet we are trying to add two bytes whose result over shoots the range of values that the resultant byte can hold, resulting in an overflow, and hence the absurd result 1. This is what the less wary among us (at least I do) run into when choosing data types to work with, paying little attention to the range of data that the program expects these data types to store and handle. A subtle arithmetic addition operation that has the potential to generate an overflow operation is enough to send your application to the tomb. Most often, good testing practices catch these bugs during the testing phase. But it certainly might get past the QA team if the test data being fed to the program is not very exhaustive and if every possible test case is not being taken into account – The Ariane 5 crash of 1996 is a testimony to that. So as developers, we need to code defensively to *catch* such arithmetic overflow errors and handle them appropriately in the execution flow of the program as our application logic dictates, thus leaving no room for inconsistent or incorrect results to bring down the application to a grinding halt.

So let’s see where C# can help us here. C# provides the *checked* keyword to trap and handle such arithmetic overflows. When an arithmetic operation that is enclosed within a *checked* block (or *checked* expression) results in an overflow, the runtime generates a *System.OverflowException*. Compare this to our previous example where the overflow result was silently assigned to the resulting byte. So let’s modify the previous example to enclose the arithmetic addition of the two bytes within a *checked* block. The modified code is shown below:

```
using System;

class ByteBites
{
   static void Main(string[] args)
   {
      try
      {
         checked
         {
            byte b1 = 57;
            byte b2 = 200;
            byte bResult = (byte)(b1 + b2);

            Console.WriteLine("{0} + {1} = {2}", b1, b2, bResult);
         }
      }
      catch(OverflowException exOverflow)
      {
         Console.WriteLine("Caught overflow exception: {0}",
                           exOverflow.Message);
      }
   }
}
```

You'll notice that we now have a *checked* block that encloses the arithmetic operation, which in turn is enclosed within a *try* block. Compile and run the application. Notice the output:

```
Caught overflow exception: Arithmetic operation resulted in an overflow.
```

You'll notice that the addition operation generated a *System.OverflowException* because it was enclosed within a *checked* block. Remove the *checked* block and you'll notice that you'll again get back the cryptic 1 as the result. But then, wouldn't it be asking too much if we had to put each and every arithmetic operation that had the potential to generate an overflow within a *checked* block. Thankfully, there's an easier way to turn on arithmetic overflow checking for the entire application by using the */checked* compiler option when compiling your application. To test this, go ahead and remove the *checked* block that is enclosing the addition operation. This time compile the program with the */checked* switch turned on, by typing the following command in the DOS command line:

```
csc  /target:exe /checked ByteBites.cs
```

Run the program and observe the output that the program spews out:

```
Caught overflow exception: Arithmetic operation resulted in an overflow.
```

Notice that the */checked* option has the same effect as using the *checked* block around your arithmetic operations. This option thereby allows you to enforce arithmetic overflow checks and to catch such exceptions throughout your application. So what if you've turned on the */checked* option and want to selectively prevent certain parts in your application from generating an overflow exception when an overflow occurs. For example, assume you have a scenario where you need the check the value of the overflowed result to determine what action to take and so on. In such cases, you can use the *unchecked* keyword and enclose those arithmetic operations within an *unchecked* block so that an *OverflowException* is not generated for those operations. This is shown in the snippet of code below:

```
unchecked
{
   byte b1 = 57;
   byte b2 = 200;
   byte bResult = (byte)(b1 + b2);

   Console.WriteLine("{0} + {1} = {2}", b1, b2, bResult);
}
```

Of course, the above code fragment gives you yet another opportunity to see the infamous 1 as the result.

Using the *checked* keyword and */checked* compiler option judiciously in your C# applications can help you catch arithmetic overflow exceptions and to ensure that your application stays sane.

7.8 Parting thoughts…

It has always been our tendency to put together our application’s functional capabilities in leaps and bounds paying little attention to analyzing where those functional capabilities could possibly go wrong during execution and to handle those error conditions appropriately. Today’s software systems are increasingly expected to meet high levels of fault tolerance and reliability. To meet that objective, we need to adopt effective exception and error handling strategies in our applications to trap errors and to recover from them gracefully. The C#/VB.NET language supports powerful constructs to handle exceptions in our applications. Hopefully, this tutorial gave you an introduction on how to use those constructs in your applications to handle errors and exceptions. Put them to good use and you’ll be well on your way to writing robust, fault tolerant, and reliable applications.

8. Assemblies and Application Domains

Section Owner: Akila Manian (MVP)
Content Contributors: Narayana Rao Surapaneni (MVP)

8.1 Introduction

In Microsoft .NET, when an application is compiled, the output of the compilation produces what is known as an ***Assembly***. Two types of assemblies can be produced by the compilation procedure. One is the executable file (*.exe) and the other is a dynamic link library file (*.dll). Basically the assembly is the unit of deployment in Microsoft .NET and it can be thought of as a collection of types and resources that form a logical unit of functionality.

An assembly is a ***self-describing*** entity. It contains all information about the types (classes) contained in the assembly, all external references needed for executing the assembly and so on. This is possible with an assembly manifest. The ***manifest*** contains assembly's identity and version information, a file table containing all files that make up the assembly and the assembly reference list for all external dependencies. Thus assemblies do not need to depend on the registry values for compilation or execution.

An assembly contains manifest data and one or more modules. ***Manifest data*** contains information about the assembly and other list of assemblies that it depends on. It also contains all the publicly exposed types and resources. An assembly contains various modules. Each module contains metadata and IL.

Assemblies can be viewed by application developers with the help of a tool called ildasm (IL Disassembler) provided by the .NET Framework.

8.2 Assembly Types

Assemblies can be single file assemblies or multi file assemblies. In multi file assemblies one of the files must contain the assembly's manifest data. Multi file assemblies can have only one entry point even though the assembly can contain multiple code modules. A multi file assembly is created primarily for combining modules written in different programming languages. Once the assembly is created, the file that contains the assembly manifest (and hence the assembly) can be signed, or one can give the file (and the assembly) a strong name and put it in the global assembly cache.

Main uses of multi file assembly are for combining modules written in different programming languages. They enable optimization of downloading an application by putting seldom-used types in a module that is downloaded only when needed. The .NET Framework downloads a file only when it is referenced; keeping infrequently referenced code in a separate file from the application optimizes code download.

Let us look at an example of how to create multi file assembly.

AddModule.cs

Copy and Paste the following code into Notepad and save it as or AddModule.vb, depending on the language that you are using

Code Listing in C#

```
using System;

public class AddClass
{
    public int Add(int Operand1, int Operand2)
     {
       return Operand1 + Operand2;
    }
}
```

Compiling in C#

```
/Compilation csc /r:System.dll /t:Module AddModule.cs
```

Code Listing in VB.NET

```
Imports System
Public Module AddModule
  Public Class AddClass
    Function Add(ByVal Operand1 As Integer, ByVal Operand2 As Integer) As Integer
      Add = Operand1 + Operand2
    End Function
  End Class
End Module
```

Compiling in VB.NET

```
vbc /r:System.dll /t:Module AddModule.vb
```

This file is compiled with the target option as module. Hence an assembly is not created. The output is a file with an extension of ***.netmodule***.

Similarly create SubtractModule.cs/ as shown below

SubtractModule.vb

Code Listing in C#

```
using System;

  public class SubtractClass
  {
    public int Subtract(int Operand1 , int Operand2 )
    {
      return  Operand1 - Operand2;
    }
  }
```

Compiling in C#

```
csc /r:System.dll /t:Module SubtractModule.cs
```

Code Listing in VB.NET

```
Imports System
Public Module SubtractModule
  Public Class SubtractClass
    Function Subtract(ByVal Operand1 As Integer, ByVal Operand2 As Integer) As
Integer
      Subtract = Operand1 - Operand2
    End Function
  End Class
End Module
```

Compiling in VB.NET

```
vbc /r:System.dll /t:Module SubtractModule.vb
```

Now create the main module, which references the above modules. The code is as shown below for MainModule.cs/MainModule.vb

MainModule.cs

Code Listing in C#

```
using System;
public class MainModule
{
   public static void  Main()
   {
         int iOperand1, iOperand2, iResult ;

         iOperand1 = 22;
         iOperand2 = 11;
         iResult = 0;

         AddClass objAddClass = New AddClass();
         SubtractClass objSubtractClass = New SubtractClass();

         iResult = objAddClass.Add(iOperand1, iOperand2);
         Console.WriteLine(iResult.ToString());

         iResult = objSubtractClass.Subtract(iOperand1, iOperand2);
         Console.WriteLine(iResult.ToString());
         Console.ReadLine();
   }
}
```

Compiling in C#

```
Compilation csc /r:System.dll MainModule.cs
```

Code Listing in VB.NET

```
Imports System
Public Module MainModule
   Sub Main()
         Dim iOperand1, iOperand2, iResult As Integer

         iOperand1 = 22
         iOperand2 = 11
         iResult = 0

         Dim objAddClass As New AddClass
         Dim objSubtractClass As New SubtractClass

         iResult = objAddClass.Add(iOperand1, iOperand2)
         Console.WriteLine(iResult.ToString)

         iResult = objSubtractClass.Subtract(iOperand1, iOperand2)
         Console.WriteLine(iResult.ToString)
         Console.ReadLine()
   End Sub
End Module
```

Compiling in VB.NET

```
vbc /r:System.dll MainModule.vb
```

The code is compiled as follows

To create a multi file assemble, which contains the two modules created previously namely AddModule and SubtractModule, compile using the following command at the command prompt

Compiling in C#

```
csc /System.dll /addModule:AddModule.netmodule
/addModule:SubtractModule.netmodule MainModule.cs
```

Compiling in VB.NET

```
Vbc /System.dll /addModule:AddModule.netmodule
/addModule:SubtractModule.netmodule MainModule.vb
```

This process creates a multi file assembly. This assembly contains the two modules created previously namely AddModule and SubtractModule. Thus this assembly contains multiple modules. If ildasm utility is executed on the MainModule assembly, it shows that the manifest information in the MainModule contains references to the AddModule and the SubtractModule modules. That is the modules are linked to the main assembly by the information contained in the main assembly's manifest information.

8.3 Private Assemblies

A private assembly is an assembly that is deployed with an application and is available ***only for that application***. That is, other applications do not share the private assembly. Private assemblies are installed in a folder of the application's directory structure. Typically, this is the folder containing the application's executable file.

For most .NET Framework applications, you keep the assemblies that make up an application in the application's directory, in a subdirectory of the application's directory. You can override where the CLR looks for an assembly by using the <codeBase> element in a configuration file.

8.4 Shared Assemblies

A shared assembly is an assembly available for use by ***multiple applications*** on the computer. To make the assembly global, it has to be put into the Global Assembly Cache. Each computer where the common language runtime is installed has a machine-wide code cache called the global assembly cache. The global assembly cache stores assemblies specifically for sharing by several applications on the computer.

You should share assemblies by installing them into the global assembly cache only when you need to. As a general guideline, keep assembly dependencies private and locate assemblies in the application directory unless sharing an assembly is explicitly required.

This is achieved with the help of a global assembly cache tool (gacutil.exe) provided by the .NET Framework. One can also drag & drop the assemblies into the ***Global Assembly Cache*** directory.

However, when an assembly has to be put into the Global Assembly Cache it needs to be signed with a strong name. A ***strong name*** contains the assembly's identity i.e. it's text name, version number, and culture information strengthened by a public key and a digital signature generated over the assembly. This is because the CLR verifies the strong name signature when the assembly is placed in the Global Assembly Cache.

8.5 Application Domains

Introduction

Traditionally when many applications are executed on the same machine, they are isolated by something known as process. Ideally each application is loaded in its own process also known as address space. This isolation is need so that these applications do not tamper with other applications either intentionally or accidentally. Isolating applications is also important for application security. For example, one can run controls from several Web applications in a single browser process in such a way that the controls cannot access each other's data and resources.

There are however many instances in which one would like an application to have the ability to communicate with other applications. Since these applications are loaded into different address spaces, there must be some form of context switching needed to allow one application to communicate with another. Inter process communication has to rely on operating systems support to manage this context switching and it is generally an expensive operation. Context switching means saving a process's context, it could also mean swapping the process out to virtual memory (to the page file stored on disk). If a single machine has a large number of active processes, the CPU is often reduced to swapping processes in and out of memory continuously, a phenomenon known as thrashing.

Application Domains

There should be a method by which the different applications can be executed in isolation from each other and which would also allow these applications to communicate with each other in a better way than offered by context switching. Microsoft .NET has introduced the Application Domain concept for precisely this reason. Application domains provide a secure and versatile unit of processing that the CLR can use to provide isolation between applications. Further, one can specify custom security policies on an Application Domain to ensure that codes run in an extremely strict and controlled app domain. One can run several application domains in a single process with the same level of isolation that would exist in separate processes, but without incurring the additional overhead of context switching between the processes. In Microsoft .NET, code normally passes a verification process before it can be executed. This code is considered as type-safe code and this allows CLR to provide a great level of isolation at the process level. Type-safe codes have less chances of causing memory faults.

Code running in one application should not directly access code or resources from another application. The CLR enforces this isolation by preventing direct calls between objects in different application domains. Objects that pass between domains are either copied or accessed by proxy. If the object is copied, the call to the object is local. That is, both the caller and the object being referenced are in the same application domain. If the object is accessed through a proxy, the call to the object is remote. In this case, the caller and the object being referenced are in different application domains. As such, the metadata for the object being referenced must be available to both application domains to allow the method call to be JIT-compiled properly.

Application Domains And Assemblies

Before an assembly can be executed, it must be loaded into an application domain. By default, the CLR loads an assembly into an application domain containing the code that references it. In this way the assembly's data and code are isolated to the application using it. In case, multiple application domains reference an assembly, the assembly's code is shared amongst the different application domains. Such an assembly is said to be domain-neutral. An assembly is not shared between domains when it is granted a different set of permissions in each domain. This can occur if the runtime host sets an application domain-level security policy. Assemblies should not be loaded as domain-neutral if the set of permissions granted to the assembly is to be different in each domain.

Programming with Application Domains

Application domains are normally automatically created and managed by runtime hosts. However, Microsoft .NET also provides control to application developers to create and manage their own application domains. This would allow the developers to have control over loading and unloading the assemblies in different domains for performance reasons and maintain a high degree of isolation. Note that individual assemblies cannot be unloaded, the entire app domain has to be unloaded.

If development is being carried out with some code or components downloaded from the Internet, running it in its own application domain provides an excellent way to isolate the rest of the applications from this code.

The System namespace contains the class AppDomain. This class contains methods to create an application domain, to load and unload assemblies in the application domain.

An example will be useful to illustrate how application domains can be created and assemblies loaded and unloaded into the application domains. Note that only those assemblies that have been declared as Public can be loaded at runtime.

Copy and paste the following code into Notepad and save it as Display.cs/Display.vb, depending on the programming language used.

Display.vb

Code Listing in C#

```
using System;
public class Display
{
   public static void  Main()
   {
      Console.WriteLine("This is written by assembly 1");
      Console.ReadLine();
   }
}
```

Compiling in C#

```
Compilation csc /r:System.dll Display.cs
```

Code Listing in VB.NET

```
Imports System
Public Module Display
   Sub Main()
      Console.WriteLine("This is written by assembly 1")
      Console.ReadLine()
   End Sub
End Module
```

Compiling in VB.NET

```
vbc /r:System.dll Display.vb
```

Similarly, copy and paste the following code into Notepad and save it as Display2.cs/Display2.vb, depending on the programming language used.

Display2.cs

Code Listing in C#

```
Imports System
Public Module Display
   Sub Main()
      Console.WriteLine("This is written by assembly 2")
      Console.ReadLine()
   End Sub
End Module
```

Compiling in C#

```
Compilation csc /r:System.dll Display2.cs
```

Code Listing in VB.NET

```
Imports System
Public Module Display
   Sub Main()
      Console.WriteLine("This is written by assembly 2")
      Console.ReadLine()
   End Sub
End Module
```

Compiling in VB.NET

```
'Compilation vbc /r:System.dll Display2.vb
```

The following code - CreateAppDomain.cs/CreateAppDomain.vb contains the following code that shows how to create an application domain programmatically and how to load assemblies during runtime.

Code Listing in C#

```
using System;
using System.Reflection;
public class CreateAppDomain
{
  AppDomain m_objAppDomain;
  public static void Main()
  {
    String strAppDomainName1 ;
    String strAppDomainName2 ;
    String strAssemblyToBeExecuted ;
    strAppDomainName1 = "TestAppDomain1";
    strAppDomainName2 = "TestAppDomain2";

    strAssemblyToBeExecuted = "Display.exe";
    CreateApplicationDomain(strAppDomainName1, strAssemblyToBeExecuted);
    AppDomain.Unload(m_objAppDomain);

    strAssemblyToBeExecuted = "Display2.exe" ;
    CreateApplicationDomain(strAppDomainName2, strAssemblyToBeExecuted);
    AppDomain.Unload(m_objAppDomain);

  }

  private void  CreateApplicationDomain(String p_strAppDomainName , String
p_strAssemblyToBeExecuted)
{
    try
    {
      m_objAppDomain = AppDomain.CreateDomain(p_strAppDomainName);
      m_objAppDomain.ExecuteAssembly(p_strAssemblyToBeExecuted);
    }
    catch(AppDomainUnloadedException  objException )
    {
        Console.WriteLine("Unable to create application domain");
        Console.WriteLine("Exception is " + objException.toString());
    }
  }
}
```

Compiling in C#

```
Compilation csc /r:System.dll CreateAppDomain.cs
```

Code Listing in VB.NET

```
Imports System
Imports System.Reflection
Module CreateAppDomain
  Dim m_objAppDomain As AppDomain
  Sub Main()
    Dim strAppDomainName1 As String
    Dim strAppDomainName2 As String
    Dim strAssemblyToBeExecuted As String
    strAppDomainName1 = "TestAppDomain1"
    strAppDomainName2 = "TestAppDomain2"

    strAssemblyToBeExecuted = "Display.exe"
    CreateApplicationDomain(strAppDomainName1, strAssemblyToBeExecuted)
    AppDomain.Unload(m_objAppDomain)

    strAssemblyToBeExecuted = "Display2.exe"
    CreateApplicationDomain(strAppDomainName2, strAssemblyToBeExecuted)
    AppDomain.Unload(m_objAppDomain)

  End Sub

  Private Sub CreateApplicationDomain(p_strAppDomainName As String,
p_strAssemblyToBeExecuted As String)
    Try
      m_objAppDomain = AppDomain.CreateDomain(p_strAppDomainName)
      m_objAppDomain.ExecuteAssembly(p_strAssemblyToBeExecuted)
    Catch objException As AppDomainUnloadedException
        Console.WriteLine("Unable to create application domain")
        Console.WriteLine("Exception is " & objException.toString())
    End Try
  End Sub

End Module
```

Compiling in VB.NET

```
Compilation vbc /r:System.dll CreateAppDomain.vb
```

8.6 Conclusion

Thus in this article we have seen what assemblies are in Microsoft .NET. We have seen single file and multi file assemblies. We have also seen private and public assemblies. We have also taken a look at application domains and how they are supported in .NET.

Application domains offer all the benefits of process isolation, but are much more efficient than processes. The Microsoft .NET runtime host automatically manages the loading / unloading of the assemblies into the appropriate application domains. However, Microsoft .NET Framework class library also offers application developers with various classes that can be used to programmatically create application domains and ensure that the various applications can be isolated from each other. Also the inter-application communication is not that expensive because context switching is not involved in application communication using application domains in Microsoft .NET.