Microsoft .NET

Syllabus

C#   +   ASP.NET

Evolution of Compute languages

- Interpreted languages [BASIC]
- Structured programming languages [C, PASCAL]
- Object oriented [C++]
- Platform independent [Java]
- NET Framework [C#, VB .NET]

C - Language

- Memory management.
- Difficult to debug/manage/understand code.
- Does not depict the real word scenario. (No object oriented concept)

C++

- Bjarne Stroustrup [1979] C with classes
- 1983 renamed as C++
- Not **platform independent**.
- No inbuilt support for **internet** application development.
- No **Unicode** support.

Java

- Oak – 1991
- Not language independence.
- Java and Windows are not closely coupled.

Issues with Windows & Visual Studio

- Backward compatibility
- ‘DLL Hell’
- No Unicode support in VS 6.0
- Parallel to Java ?

.NET Framework

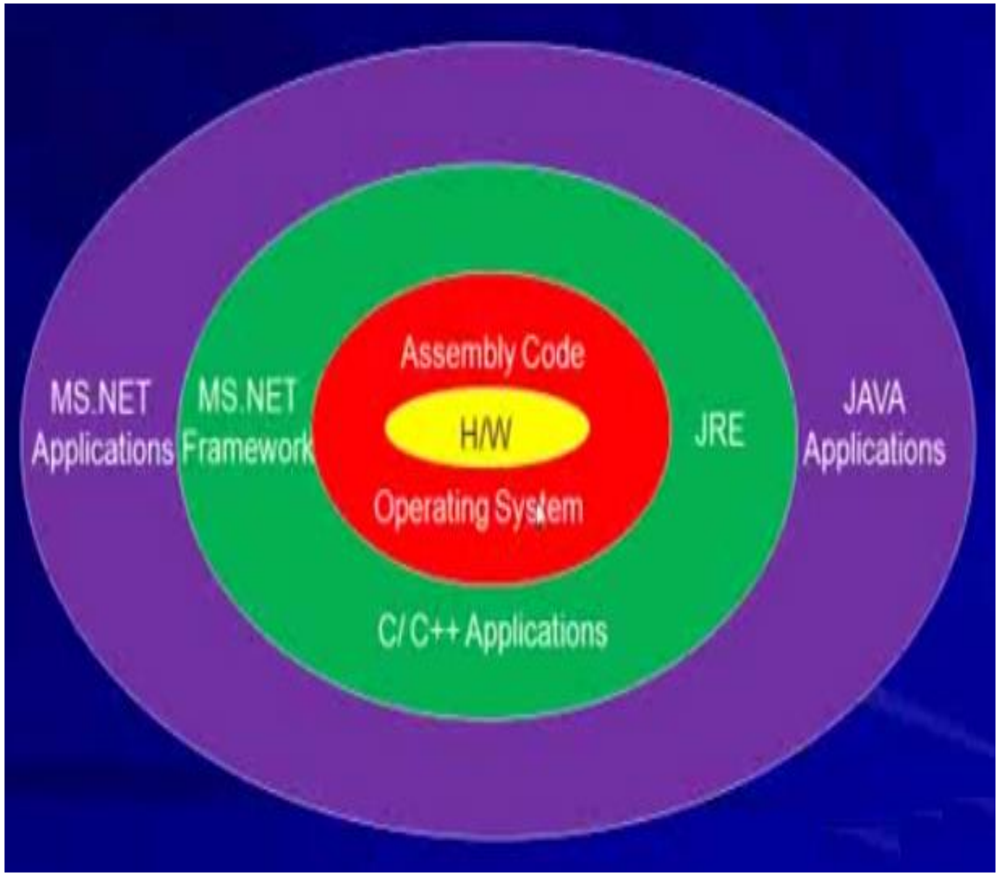
Objective

- To develop framework for and language interoperability.
- **Language interoperability** is the capability of two different programming languages interact and operate on the same kind of data structures.
- Is it platform independent?

Definitions

- **Platform** is an environment for developing and executing applications.
- **Framework** is standards and ready to use infrastructure (collection of classes, interfaces, etc.) used for developing particular type of applications.

Platforms


Assembly Code
MS.NET Applications
MS.NET Framework
H/W
JRE
JAVA Applications
Operating System
C/ C++ Applications

.NET Framework ?

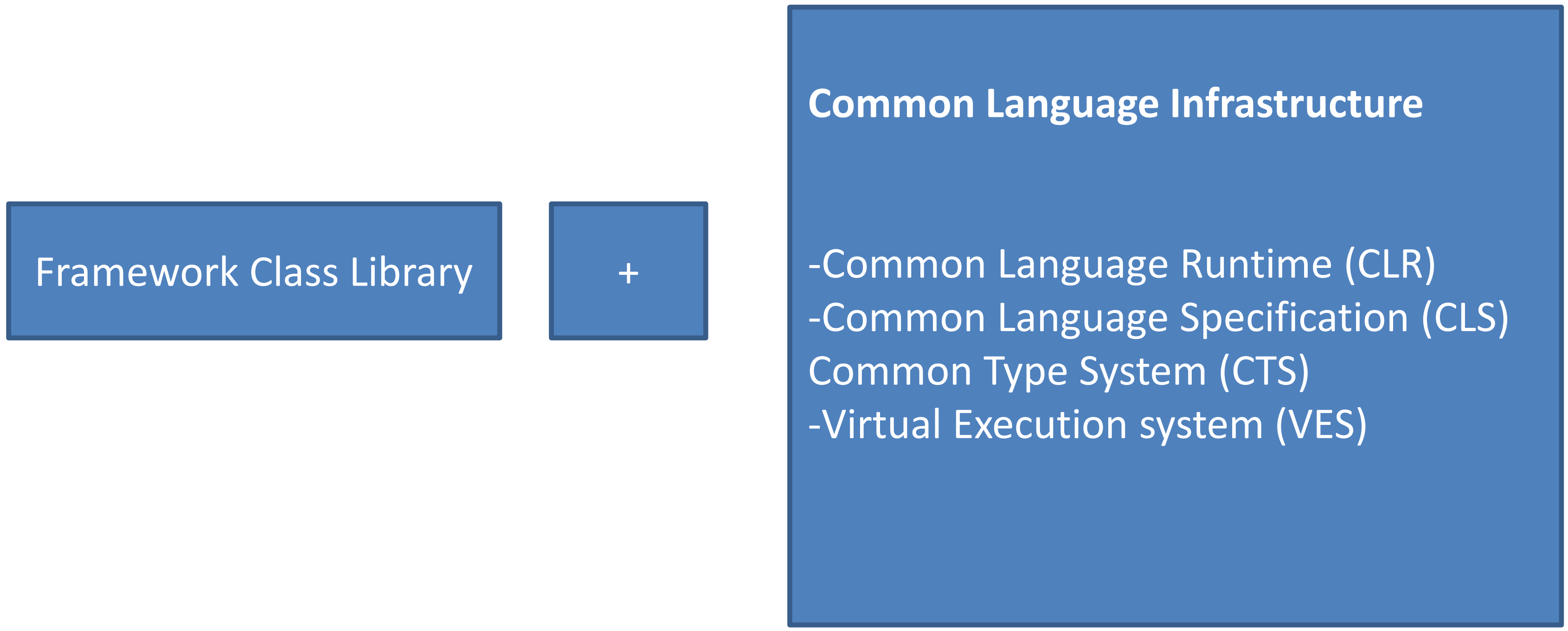
The .NET Framework is a development platform for building apps for Windows, Windows Phone, Windows Server, and Microsoft Azure.

It consists of the common language runtime (CLR) and the .NET Framework class library, which includes classes, interfaces, and value types that support an extensive range of technologies.

The .NET Framework provides a managed execution environment, simplified development and deployment, and integration with a variety of programming languages, including Visual Basic and Visual C#.

.NET Framework ?

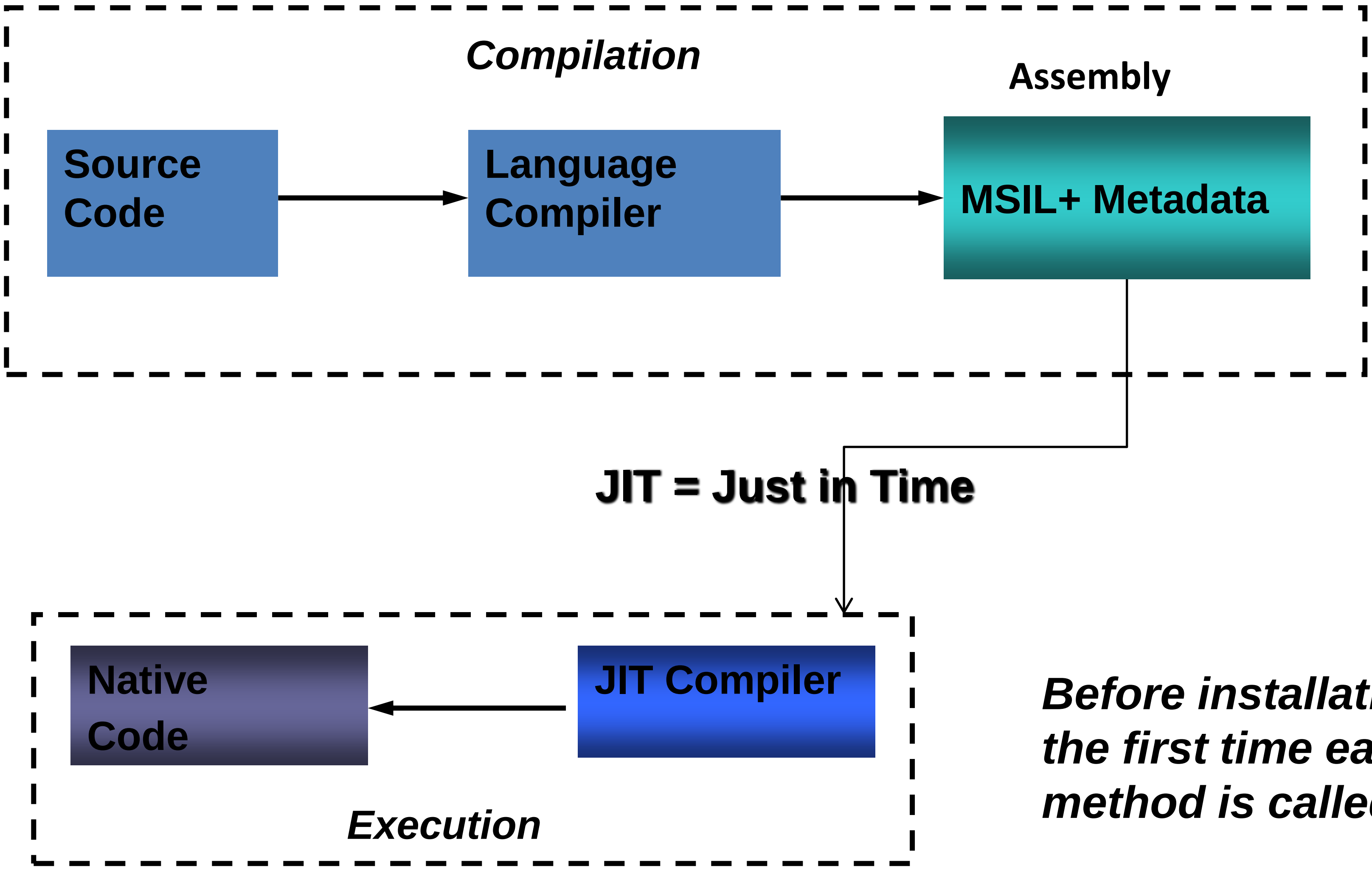
.NET Framework is collection of .NET Framework class libraries, CLR, CTS, CLS, JIT.


Framework Class Library
+
Common Language Infrastructure
-Common Language Runtime (CLR)
-Common Language Specification (CLS)
Common Type System (CTS)
-Virtual Execution system (VES)

Sample C# program

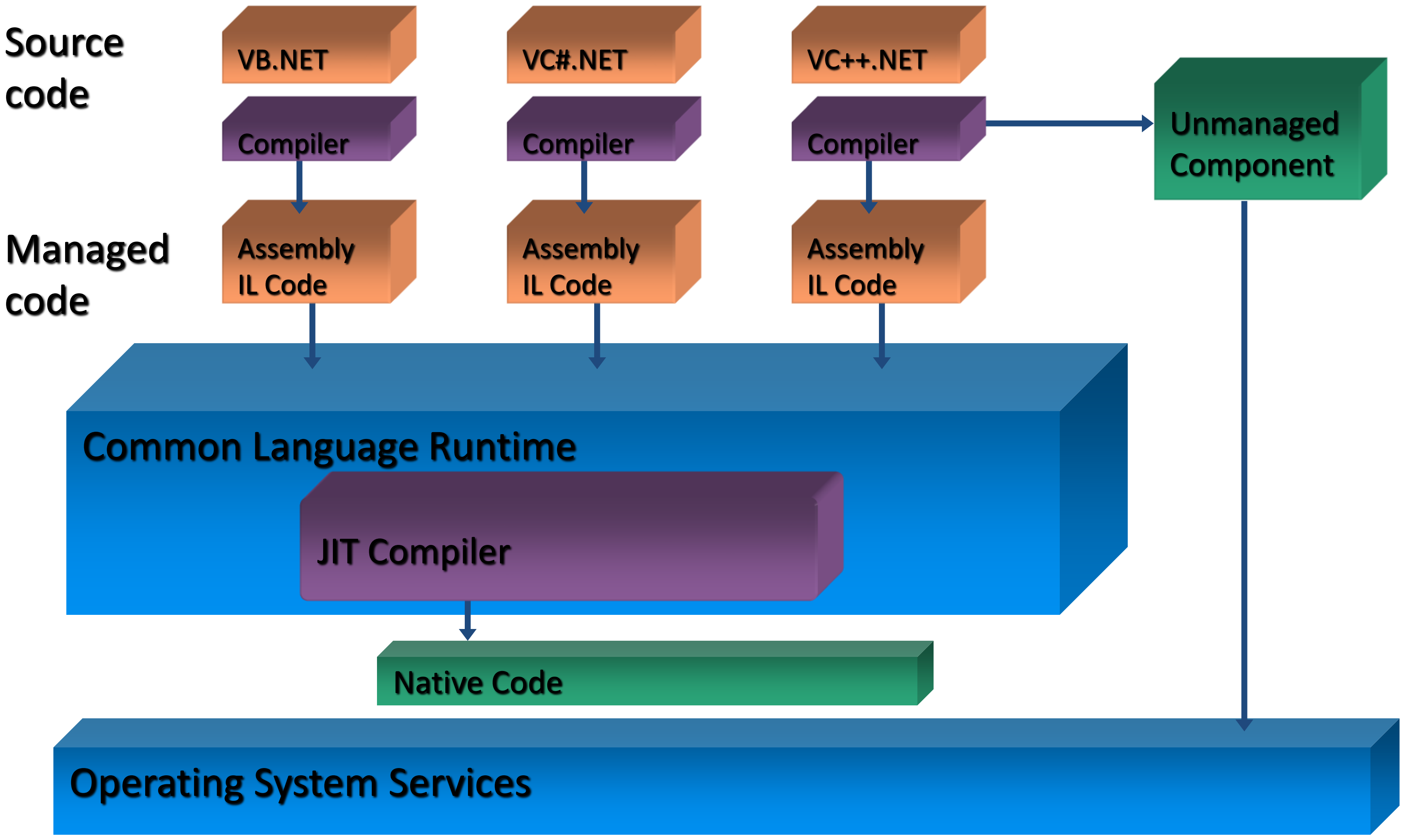
```
using system;
namespace MyNamespace
{
        class TestClass
        {
                Static voic Main()
                {
                        Console.WriteLine(“Welcome to C#”);
                }
        }
}
```

Compilation and Execution



Before installation or the first time each method is called

.NET Architecture



MSIL

- Microsoft Intermediate Language (MSIL) is a CPU-independent set of instructions that can be efficiently converted to the native code. During the runtime the Common Language Runtimes (CLR)'s Just In Time (JIT) compiler converts the Microsoft Intermediate Language (MSIL) code into native code to the Operating System.

The .NET Framework is designed to fulfill the following objectives:

- To provide a consistent **object-oriented programming environment.**
- To provide a code-execution environment that minimizes **software deployment** and **versioning conflicts.**
- To provide a code-execution environment that guarantees **safe execution of code**.
- To ensure that code based on the .NET Framework can **integrate with any other code**.

.NET Framework class Library

- The .NET Framework class library is a library of classes, interfaces, and value types that are included in the Microsoft .NET Framework.
- The .NET base classes are a massive collection of managed code classes that have been written by Microsoft, and which allow you to do almost any of the tasks that were previously available through the Windows API.
- Based on single inheritance.
- This means that you can either instantiate objects of whichever .NET base class is appropriate, or you can derive your own classes from them.
- The great thing about the .NET base classes is that they have been designed to be very intuitive and easy to use.

.NET Framework class Library

C:\Windows\assembly

- system – contains classes and base classes that define **commonly used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.** Other classes provide services supporting data type conversion, method parameter manipulation, mathematics, remote and local program invocation, application environment management, and supervision of managed and unmanaged applications.
- System.Collections – contains interfaces and classes that define various collections of objects, such as lists, queues, arrays, hashtables, and dictionaries.
- System.Collections.Generic – contains interfaces and classes that define generic collections, which allow users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections.
- System.IO – contains types that allow synchronous and asynchronous reading and writing on data streams and files.
- System.Text – contains classes representing ASCII, Unicode, UTF-7, and UTF-8 character encodings; abstract base classes for converting blocks of characters to and from blocks of bytes; and a helper class that manipulates and formats **String** objects without creating intermediate instances of **String**.
- System.Threading – provides classes and interfaces that enable multithreaded programming. This namespace includes a **ThreadPool** class that manages groups of threads, a **Timer**class that enables a delegate to be called after a specified amount of time, and a **Mutex** class for synchronizing mutually exclusive threads.

Common Language Infrastructure

The Common Language Infrastructure (CLI) is an open specification developed by Microsoft and standardized by ISO[1] and ECMA[2].

- **The Common Type System (CTS)**

A set of data types and operations that are shared by all CTS-compliant programming languages.

- **The Common Language Specification (CLS)**

A set of base rules to which any language targeting the CLI should conform in order to interoperate with other CLS-compliant languages. The CLS rules define a subset of the Common Type System.

- **JIT Just in time Compiler (Virtual executers)**

The VES loads and executes CLI-compatible programs, using the metadata to combine separately generated pieces of code at runtime.

Common Type System (CTS)

- All .NET programming languages uses the same representation for common data types (e.g. int, double etc.)
- In Microsoft's .NET Framework, the Common Type System (CTS) is a **standard that specifies how type definitions and specific values of types are represented in computer memory.**
- Why?

-It is intended to allow programs written in different programming languages to easily share information.

Common Language Specification (CLS)

A set of base rules to which any language targeting the CLI should conform, in order to interoperate with other CLS-compliant languages.

Global Assembly Cash (GAC)

- 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 designated to be shared by several applications on the computer.

Global Assembly Cash (GAC)

Starting with the .NET Framework 4, the default location for the global assembly cache is **%windir%\Microsoft.NET\assembly**.

In earlier versions of the .NET Framework, the default location is**%windir%\assembly**.

Common Language Specification (CLS)

- Public fields should not have unsigned types like uint or ulong,.
- public methods should not return unsigned types, parameters passed to public function should not have unsigned types. However unsigned types can be part of private members.
- Unsafe types like pointers should not be used with public members. However they can be used with private members.
- Class names and member names should not differ only based on their case. For example we cannot have two methods named MyMethod and MYMETHOD.
- Only properties and methods may be overloaded, Operators should not be overloaded.

What is Assembly ?

- Assemblies are the building blocks of .NET Framework applications.
- They form the fundamental unit of deployment, version control, reuse, 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.

Assembly Content

Assembly Manifest + MSIL

An **Assembly Manifest** is a file that contains Metadata about .NET Assemblies.

Information	Description
Assembly name	A text string specifying the assembly's name.
Version number	A major and minor version number, and a revision and build number. The common language runtime uses these numbers to enforce version policy.
Culture	Information on the culture or language the assembly supports. This information should be used only to designate an assembly as a satellite assembly containing culture- or language-specific information. (An assembly with culture information is automatically assumed to be a satellite assembly.)
Strong name information	The public key from the publisher if the assembly has been given a strong name.
List of all files in the assembly	A hash of each file contained in the assembly and a file name. Note that all files that make up the assembly must be in the same directory as the file containing the assembly manifest.
Type reference information	Information used by the runtime to map a type reference to the file that contains its declaration and implementation. This is used for types that are exported from the assembly.
Information on referenced assemblies	A list of other assemblies that are statically referenced by the assembly. Each reference includes the dependent assembly's name, assembly metadata (version, culture, operating system, and so on), and public key, if the assembly is strong named.

Managed and Unmanaged code

- Managed code is the code that is written to target the services of the managed runtime execution environment such as Common Language Runtime in .Net Technology.
- Unmanaged code compiles straight to machine code and directly executed by the Operating System. The generated code runs natively on the host processor and the processor directly executes the code generated by the compiler. It is always compiled to target a specific architecture and will only run on the intended platform.

To view contents of Assembly

- ILDASM utility.

C# Compiler

C#

C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe

VB .Net

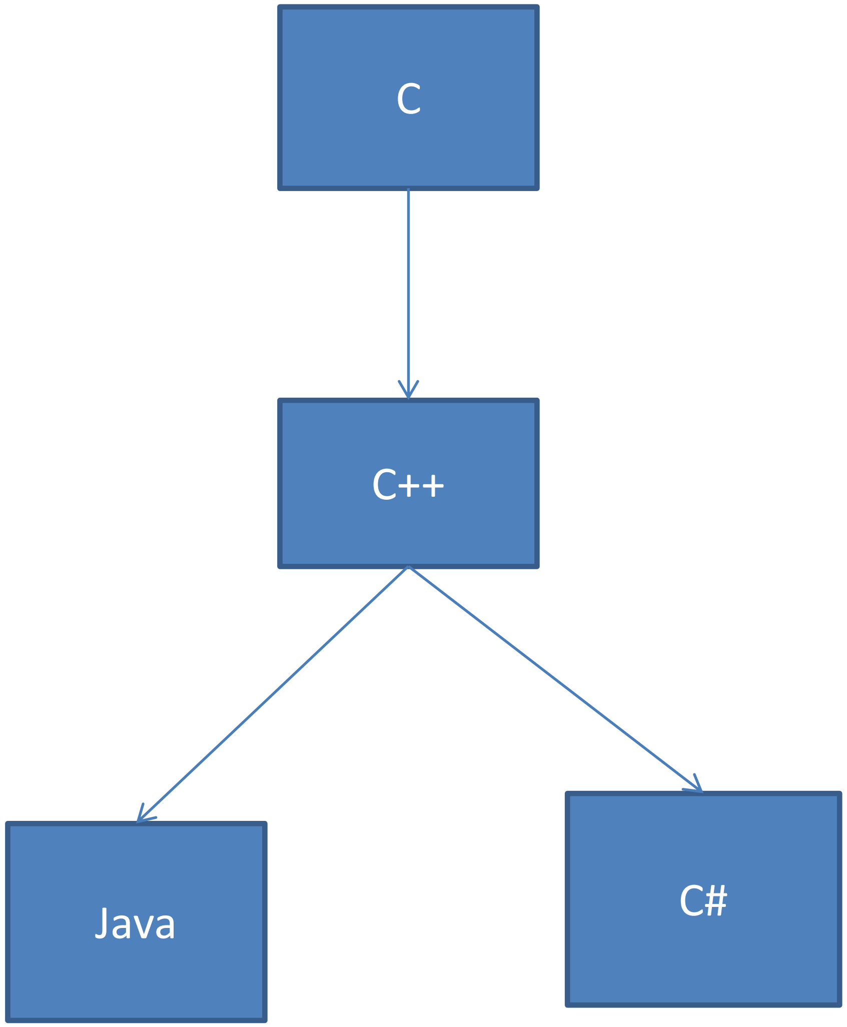
C:\Windows\Microsoft.NET\Framework\v4.0.30319\vbc.exe

C#

C# is .NET compliant programming language from Microsoft.

C# is object oriented programming language.

C# is strongly typed.


C
C++
Java
C#

Visual Studio

- How to open new project.
- What is solution.
- If Solution is not visible?
- Shortcut keys.

Microsoft Visual Studio 2010

- Solution can contain multiple projects.
- Start>All programs>Microsoft Visual Studio 2010> Microsoft Visual Studio 2010

Microsoft Visual Studio 2010

- Solution is collection of projects.
- Solution can contain multiple projects.
- If the solution contain multiple projects, select particular project as startup project to run that project.
- Select particular class as startup to run particular class within the project.

Reading and writing console.

- Console.ReadLine();
- Console.WriteLine();

Visual Studio

- Console Application
- Windows Forms
- WPF
- WCF
- WWF
- Class libraries/ Assemblies.

Types in C#

- C# has two distinct groups of data types, called **Value Types and Reference Types.**
- Value Types directly contain their own data.
- Reference Types are accessed indirectly using reference variables that point to the object
- All Types in C#, both value types and reference types inherit from the object super class

Types

- Value Type can be:
  - Simple Type
  - Structs
  - enums
- Reference Type can be:
  - Classes
  - Interfaces
  - delegates
  - arrays
- C# provides a set of predefined struct types, called simple types, identified using reserved words.

Value Types

- C# provides a set of predefined struct types, called simple types, identified using reserved words.
- These reserved words are aliases for predefined struct types in System namespace.
- Simple types can be divided into four categories:
  - Integral Types
  - Floating point Types
  - Decimal Type
  - Bool Type

Integral Types Table

Following table shows the sizes and ranges of the integral types, which constitute a subset of simple types.

Type	Range	Size
sbyte	-128 to 127	Signed 8-bit integer
byte	0 to 255	Unsigned 8-bit integer
char	U+0000 to U+ffff	Unicode 16-bit character
short	-32,768 to 32,767	Signed 16-bit integer
ushort	0 to 65,535	Unsigned 16-bit integer
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint	0 to 4,294,967,295	Unsigned 32-bit integer
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

The following table shows the keywords for built-in C# types, which are aliases of predefined types in the System namespace.

C# Type	.NET Framework Type
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

Default Values Table

Value type	Default value
bool	false
byte	0
char	'\0'
decimal	0.0M
double	0.0D
enum	The value produced by the expression (E)0, where E is the enum identifier.
float	0.0F
int	0
long	0L
sbyte	0
short	0
struct	The value produced by setting all value-type fields to their default values and all reference-type fields to null.
uint	0
ulong	0
ushort	0

- No overflow exception for integer arithmetic operations.
- Use **checked** statement to handle exceptions.

Floating Point Types

- Floating Point Types

  C# supports two Floating Point Types:

  - float 32 bits 7 bits precision
  - double 64 bits 15 bits precision

- Floating point types also provides a set of values:
  - Positive and Negative zero
  - Positive and Negative Infinity
  - NaN (Not a Number), these are produce as a result of invalid floating-point operations

Decimal

- Decimal Type

  Decimal type is suitable for financial and monetary calculation.

  Decimal 128 bits 1.0 X 10^{-28} to 7.9 X 10^{28}

  28 - 29 significant bits

- If one of the operator of a binary operator is of type decimal, then the other operator must be a integral part or of type decimal.

bool

bool Type represents Boolean values.

- Possible values of type bool are :
  - true
  - false
- No standard conversion is exist between bool and other types
- Bool value cannot be used in places of an integral value and vise versa

Literals in C#

- Literal is a textual representation of a particular value.
- Literals give hint to compiler and programmers about their type.
- The compilers will interpret them as specific type.

Literals

char c1 = ‘\u0066’

char c2 = ‘\t’   (tab)

int value = 0x21; //33 in decinmal

Literals

- F – float e.g. 1.33F
- D –Double e.g. 196.89D
- M –Decimal e.g. 9.99M
- L – Long e.g. 12L

String type

- String type is built in reference type in C#
- Escape sequence \ is used for special characters.

Verbatim Literal:

- Verbatim literal is a string with an @ symbol prefix. E.g. @”Hello to C#”
- Verbatim literals make escape sequence translate as normal printable characters to enhance readability.

Type conversion

- **Implicit conversions**: No special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.
- **Explicit conversions (casts)**: Explicit conversions require a cast operator or use **Convert** class.

Convertion

Implicit numeric conversion

From	To
sbyte	short, int, long, float, double or decimal
byte	short, ushort, int, unit, long, ulong, float, double or float
short	int, long, float, double, or decimal
ushort	int, uint, long, ulong, float, double, or decimal
int	long, float, double, or decimal
uint	long, ulong, float, double, or decimal
long	float, double, or decimal
ulong	float, double, or decimal
float	double

Arithmetic Operators

- + Addition
- - Subtratction
- * Multiplication
- / Division
- % Modulus
- += Increment
- -= Decrement

Relational Operators

- == Equal to
- != Not equal to
- > Greater than
- < Less than
- >= Greater than or equal to
- <= Less then or equal to

Logical Operators

&	And
|	OR
^	XOR
||	Short-circuit OR
&&	Short-circuit AND
!	NOT

Identifier

- In C# identifier is a name assigned to a method, a variable, or any other user-defined item.
- Identifier can be one or more characters long.
- Identifier may start with any letter of the alphabet or an underscore. Next may be a letter, a digit, or an underscore.
- Uppercase and lowercase are different.

Reference Type

- The following keywords are used to declare reference types:
- class
- interface
- delegate
- C# also provides the following built-in reference types:
- object
- string

Difference between Value and Reference Type

Value type are stored in **Stack.**

Reference type are stored in **Managed Heap.**

Value types are destroyed immediately after the scope is lost. For reference variable, only the reference variable is destroyed after scope is lost. The object is later destroyed by Garbage Collector.

Void type

- When used as the return type for a method, **void** specifies that the method doesn't return a value.
- **void** isn't allowed in the parameter list of a method. A method that takes no parameters and returns no value is declared as follows:

```
public void SampleMethod(){    // Body of the method.}
```

Microsoft Visual Studio 2010

- Solution is collection of projects.
- Solution can contain multiple projects.
- If the solution contain multiple projects, select particular project as startup project to run that project.
- Select particular class as startup to run particular class within the project.

Classes and objects

Class is template for creating objects.

Creating class is creating new type in C#

Class is reference type.

Purpose of create class is encapsulate attributes and behaviour of an entity in single unit.

How to create class

access-modifier class *Class-Name*

{

}

Class members

```
Class
{
        instance variables (fields)
        static members
        constants
        constructors
        destructors
        properties
        indexers
        delegates
        events
        operators
}
```

object

- The process of creating object from a class is called Instantiation.
- Objects can be instantiated in two ways:

```
Box mybox = new Box();
(or)
Box mybox;
mybox = new Box();
```

- When an object is declared its value is NULL

C# Strongly typed

Strongly Typed

C# is a strongly-typed language.

Every variable and constant has a type, as does every expression that evaluates to a value.

Every method signature specifies a type for each input parameter and for the return value.

Methods

Types and Type members

Access specifiers

var type

- var is called implicitly called local variable.

Beginning in Visual C# 3.0, variables that are declared at method scope can have an implicit type **var**. An implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type. The following two declarations of i are functionally equivalent:

- var i = 10; // implicitly typedint i = 10; //explicitly typed.

Exception Handling

- An Exception is an unforeseen error when program is running
- Exception is runtime error.

Examples

- Trying to read a file that does not exist throws [FileNotFountException]
- Trying to read from a database table which does not exist. [SqlException]
- Divide number by zero. [DivideByZeroException]

Exception Handling

An Exception is actually a class in System namespace.

Exception Handling

```
try
{
  - - - -
}
Catch( Exception1 ex1)
{

}
Catch ( Exception1 ex1)
{
}
-
-
Finally
{

}
```

We use try, catch and finally block for exception handling

try – The code that possibly cause exception is in try block.

catch – handles exceptions

finally – cleans and frees resources that the class was holding during the program execution. Finally block is optional.

Exception Handling

Specific exceptions will be caught before general exception, so specific exceptions should be always on top of base class exception. Otherwise you will encounter a compiler error.

To create custom Exceptions derive class from Exception and override constructor.

Access Specifies

- **Access Modifiers (Access Specifier)** describes as the scope of accessibility of an Object and its members.
- All C# types and type members have an accessibility level.
- We can control the scope using access specifies. We are using access modifiers for providing security of our applications.
- C# provide five access specifies- ***public, private , protected , internal and protected internal*** .

public

public is the most common access specifier in C# . It can be access from anywhere, that means there is no restriction on accessibility. The scope of the accessibility is inside class as well as outside. The type or member can be accessed by any other code in the same assembly or another assembly that references it.

private

The scope of the accessibility is limited only inside the classes or struct in which they are declared. The private members cannot be accessed outside the class and it is the least permissive access level.

<u>protected :</u>

The scope of accessibility is limited within the class or struct and the class derived (Inherited) from this class.

internal

The internal access modifiers can access within the program that contain its declarations and also access within the same assembly level but not from another assembly.

protected internal

Protected internal is the same access levels of both protected and internal. It can access anywhere in the same assembly and in the same class also the classes inherited from the same class .

Properties in C#

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field.

Properties can be used as if they are public data members, but they are actually special methods called *accessors*. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

Properties

Is mechanism in C# to assign values and get values for instance variables in a class.

- We use get and set assessors to implement properties.
- Property with both get and set accessor is called Read/Write property.
- Property with only get accessor is called Read only property.
- Property with only set accessor is called Write only property.
- In a property declaration, both the get and set accessors must be declared inside the body of the property.

Advantage: The advantage of properties over traditional Get() and set()   methods is that you can access them as if they were public fields.

Auto Implemented properties

- Introduced in .NET 3.0
- If there is no additional logic in the property accessors, then we can make use of auto implemented properties.
- Auto implemented properties reduce amount of code that we have to write.
- When we use auto implemented properties the compiler created a private anonymous field that can only be accessed through the properties get and set accessors.

Static and instance Class members

- When class members include a static modifier, the member is called as static member.
- When no static member is present the member is called as non static member or instance member.
- Static members are invoked using class name where as instance members are invoked using instance of the object.

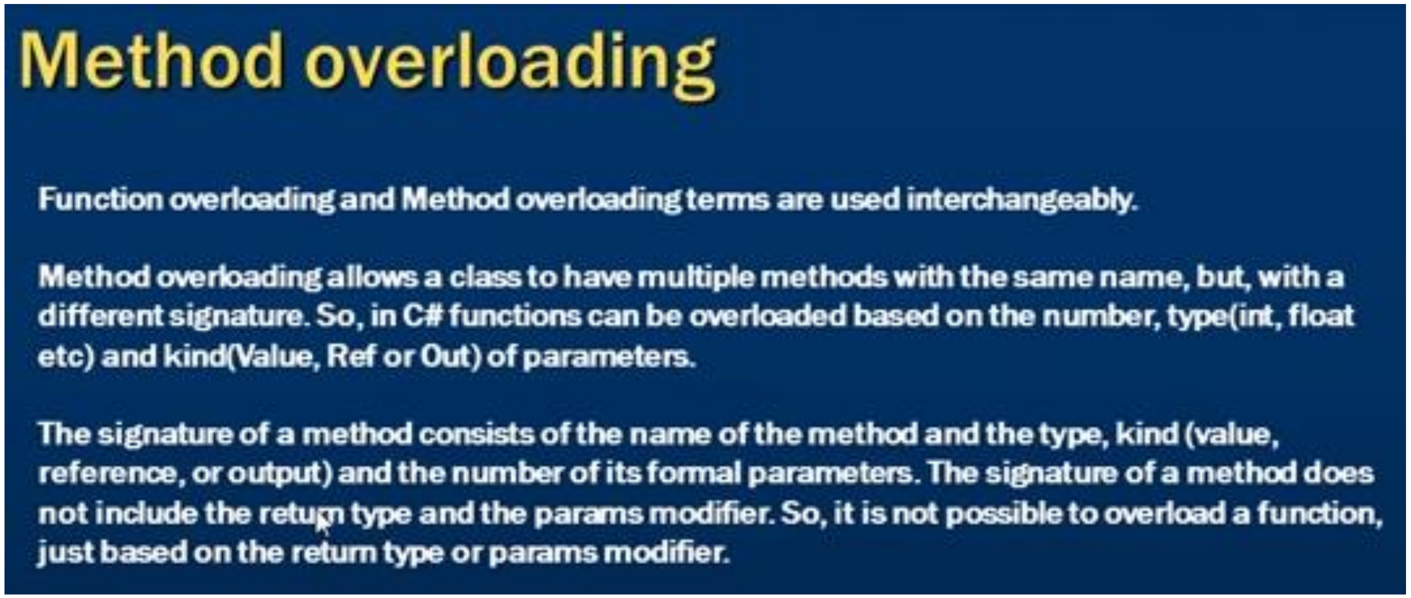
Static Constructors

Static constructors are used to initialize static fields in the class.

You declare a static constructor using the key word static in front of the constructor name.

Static constructor is called only once, no matter how many instances you create.

Static constructors are called before instance constructors.



Method overloading

Function overloading and Method overloading terms are used interchangeably.

Method overloading allows a class to have multiple methods with the same name, but, with a different signature. So, in C# functions can be overloaded based on the number, type(int, float etc) and kind(Value, Ref or Out) of parameters.

The signature of a method consists of the name of the method and the type, kind (value, reference, or output) and the number of its formal parameters. The signature of a method does not include the return type and the params modifier. So, it is not possible to overload a function, just based on the return type or params modifier.

Delegates

A delegate is a type that represents references to methods with a particular parameter list and return type.

Delegate is type safe function pointer.

When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type.

You can invoke (or call) the method through the delegate instance.

Delegates are used to pass methods as arguments to other methods.

Delegates

Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate.

The method can be either static or an instance method. This makes it possible to programmatically change method calls, and also plug new code into existing classes.

Delegates

- Delegates are like C++ function pointers but are type safe.
- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.
- Delegates can be chained together; for example, multiple methods can be called on a single event.

Multicast Delegeate

Multicast delegate is a delegate that has a references to more then one function. i.e. a delegate pointing to more then one function.

When u invoke the delegate all the functions that the delegate is pointing to invoke are invoked.

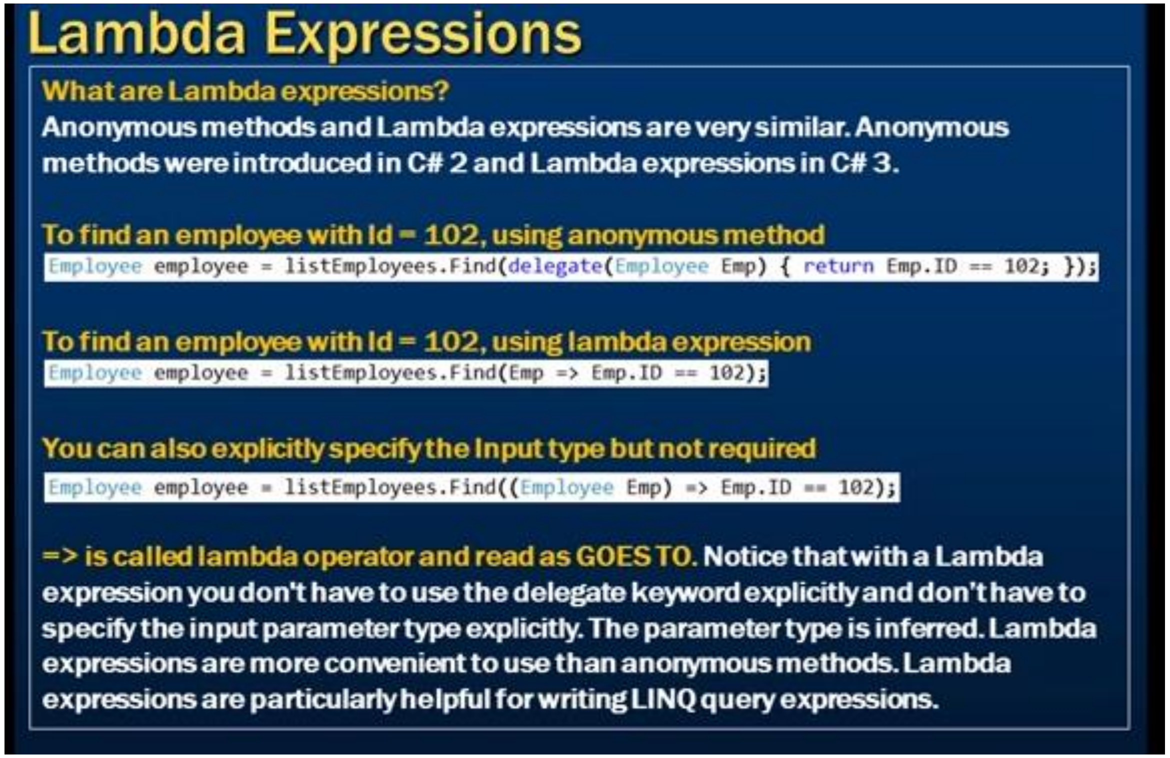
Multicast by chaining using plus sign; and we use – sign to remove them. [Registering the method with the delegate]They are called delegates invocation list. They are exceuted in the same order. -= to remove the delegate.

Anonymous methods

- Anonymous method is a method without name. This is introduced in C# 2.0.
- They provide us a way of creating delegate instances without having to write a separate method.

Lamda Expressions

- Anonymous methods were introduced in C# 2.0 and lamda expressions in c# 3.0



Lambda Expressions

What are Lambda expressions?

Anonymous methods and Lambda expressions are very similar. Anonymous methods were introduced in C# 2 and Lambda expressions in C# 3.

To find an employee with Id = 102, using anonymous method

```
Employee employee = listEmployees.Find(delegate(Employee Emp) { return Emp.ID == 102; });
```

To find an employee with Id = 102, using lambda expression

```
Employee employee = listEmployees.Find(Emp => Emp.ID == 102);
```

You can also explicitly specify the Input type but not required

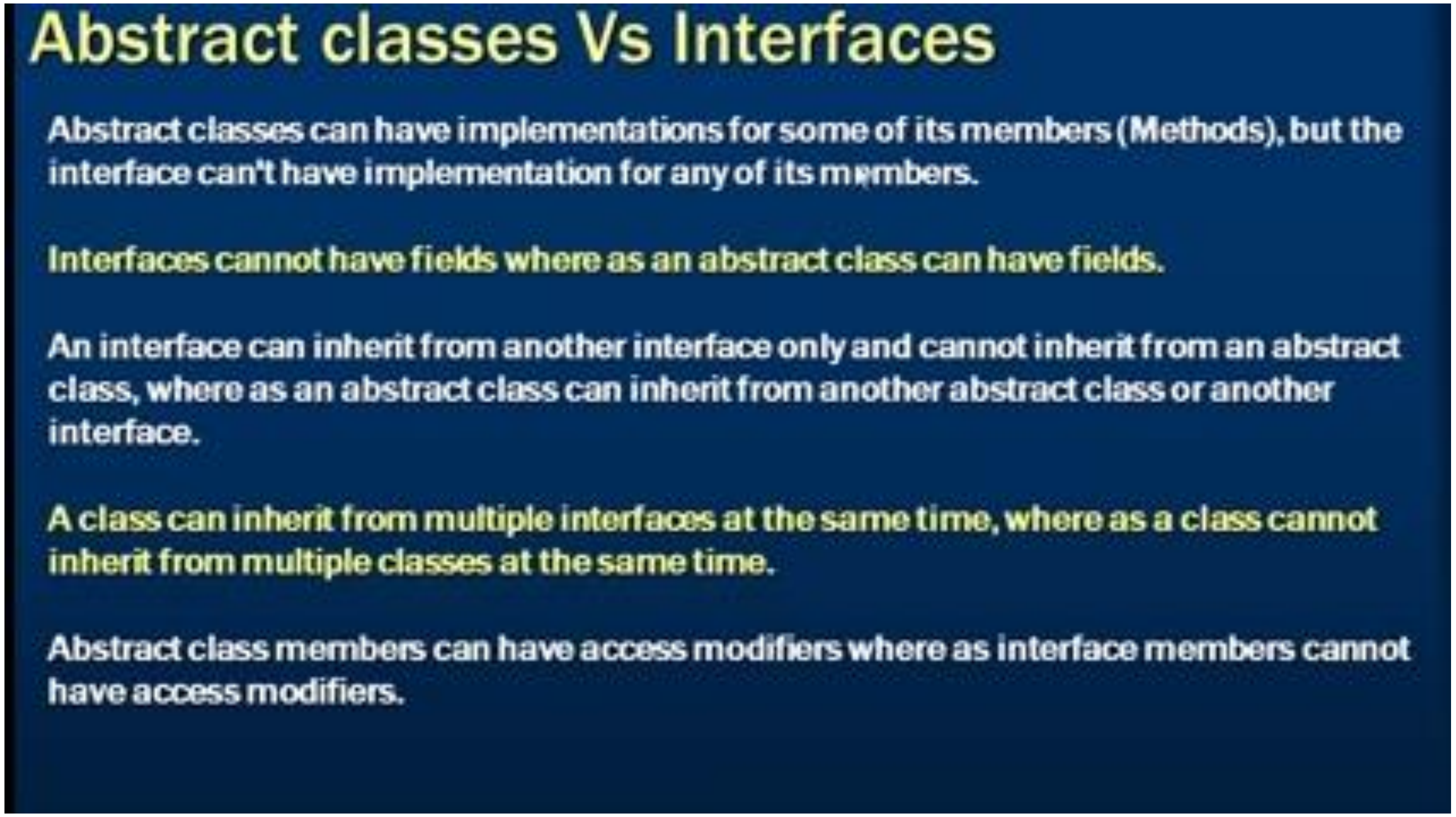
```
Employee employee = listEmployees.Find((Employee Emp) => Emp.ID == 102);
```

=> is called lambda operator and read as GOES TO. Notice that with a Lambda expression you don't have to use the delegate keyword explicitly and don't have to specify the input parameter type explicitly. The parameter type is inferred. Lambda expressions are more convenient to use than anonymous methods. Lambda expressions are particularly helpful for writing LINQ query expressions.

Extension methods

- Extension methods provide a means by which functionality can be added to a class without using the normal inheritance mechanism.
- Extension methods are useful in LINQ
- An extension method is a static method that must be contained within a static, non-generic class..
- The type of its first parameter determines the type of object on which the extension method can be called.
- The first parameter must be modified by this.
- Ref page[602]

Difference between abstract class and interfaces



Abstract classes Vs Interfaces

Abstract classes can have implementations for some of its members (Methods), but the interface can't have implementation for any of its members.

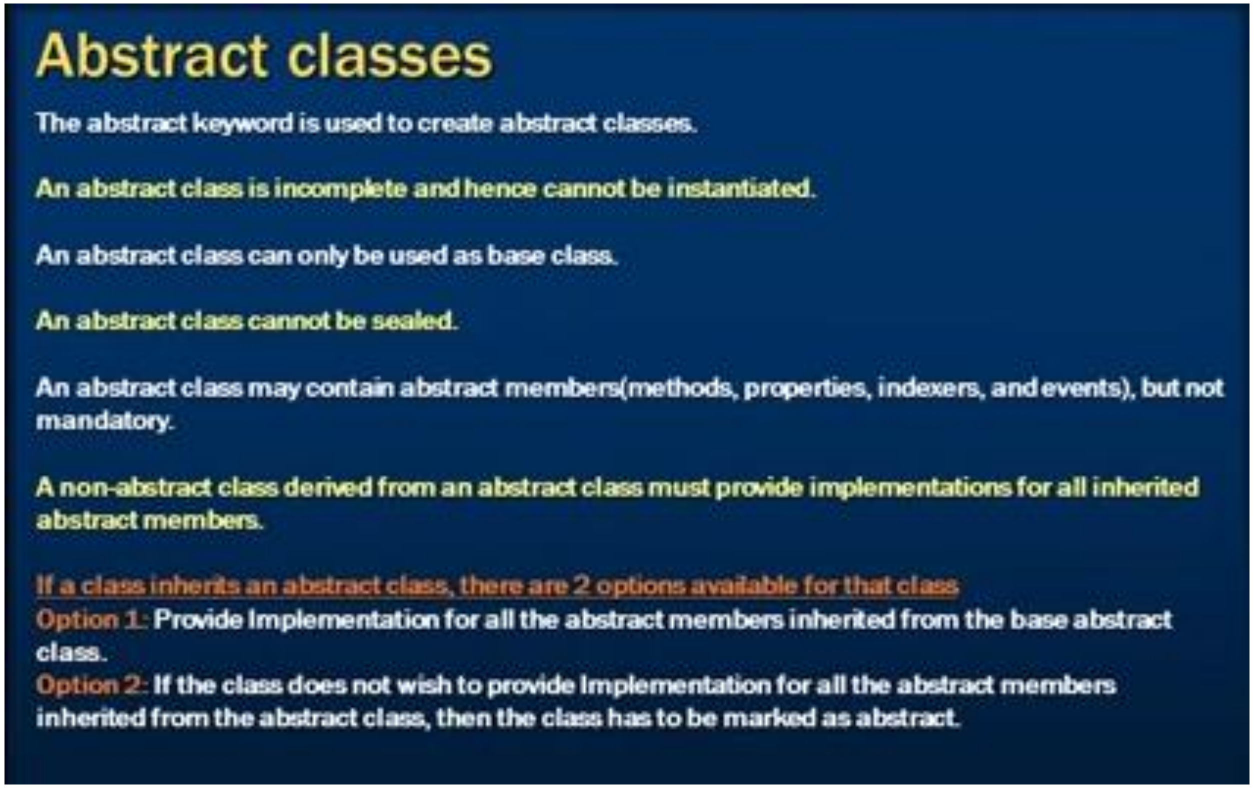
Interfaces cannot have fields where as an abstract class can have fields.

An interface can inherit from another interface only and cannot inherit from an abstract class, where as an abstract class can inherit from another abstract class or another interface.

A class can inherit from multiple interfaces at the same time, where as a class cannot inherit from multiple classes at the same time.

Abstract class members can have access modifiers where as interface members cannot have access modifiers.

Abstract class



Abstract classes

The abstract keyword is used to create abstract classes.

An abstract class is incomplete and hence cannot be instantiated.

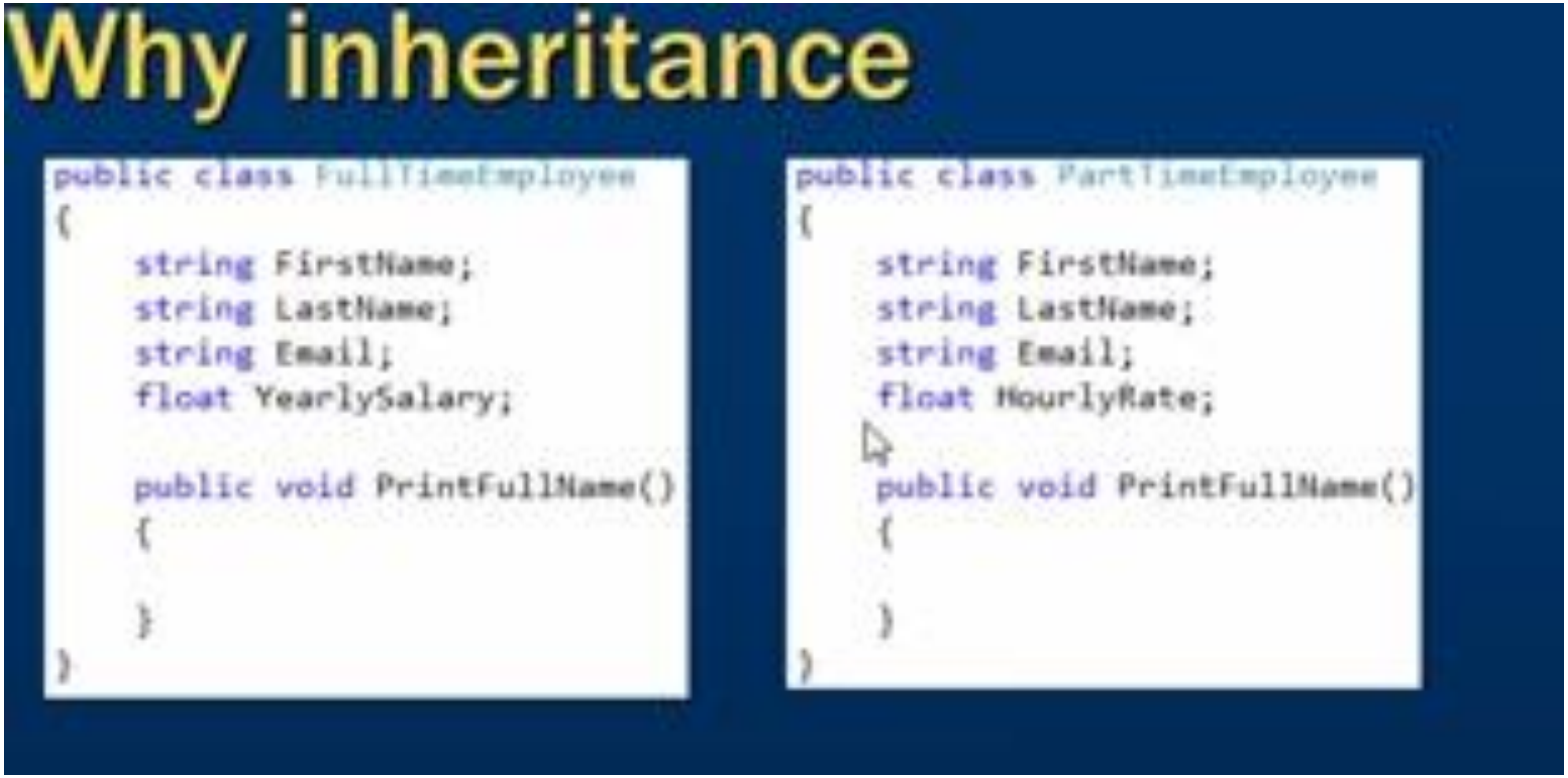
An abstract class can only be used as base class.

An abstract class cannot be sealed.

An abstract class may contain abstract members(methods, properties, indexers, and events), but not mandatory.

A non-abstract class derived from an abstract class must provide implementations for all inherited abstract members.

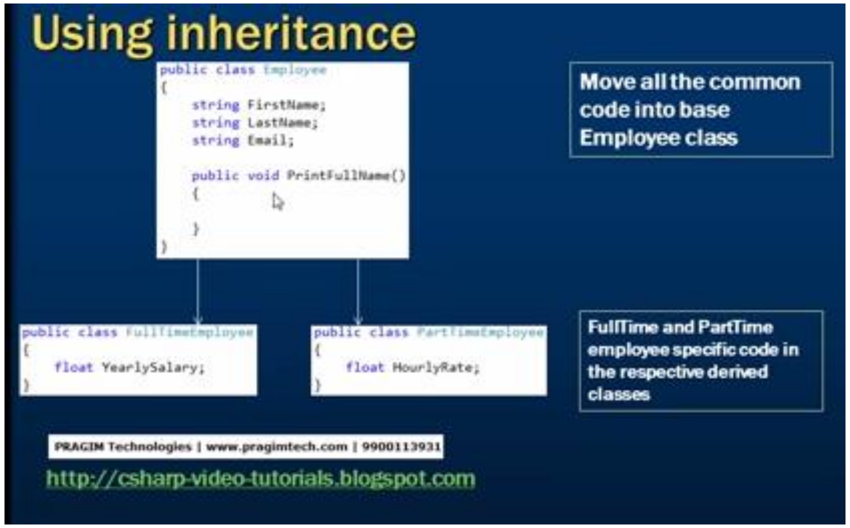
If a class inherits an abstract class, there are 2 options available for that class
Option 1: Provide Implementation for all the abstract members inherited from the base abstract class.
Option 2: If the class does not wish to provide Implementation for all the abstract members inherited from the abstract class, then the class has to be marked as abstract.



Why inheritance

```
public class FullTimeEmployee
{
    string FirstName;
    string LastName;
    string Email;
    float YearlySalary;

    public void PrintFullName()
    {

    }
}
```

```
public class PartTimeEmployee
{
    string FirstName;
    string LastName;
    string Email;
    float HourlyRate;

    public void PrintFullName()
    {

    }
}
```



Using inheritance

```
public class Employee
{
    string FirstName;
    string LastName;
    string Email;

    public void PrintFullName()
    {

    }
}
```

Move all the common code into base Employee class

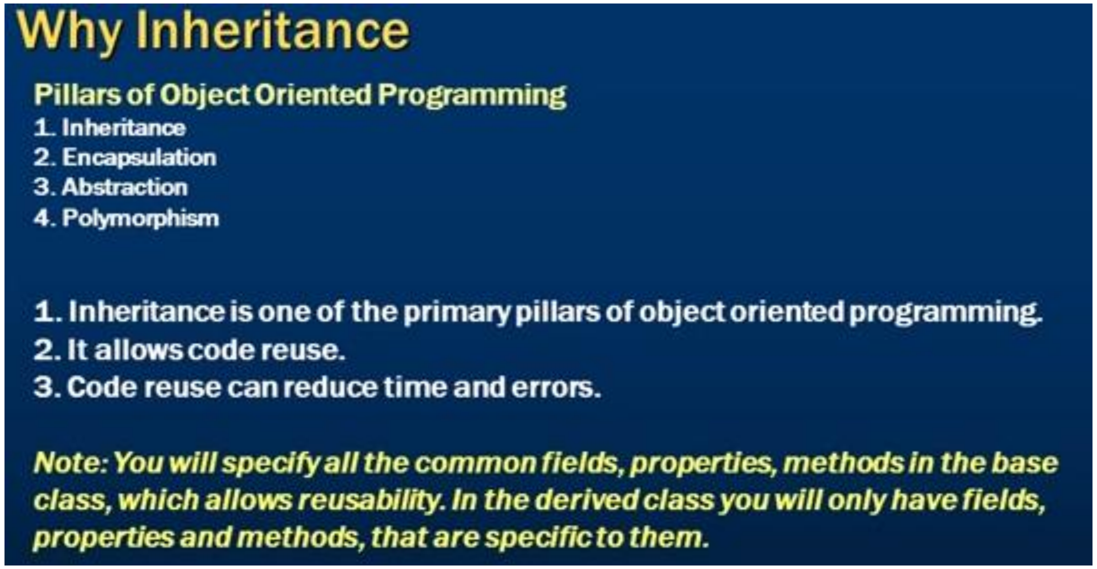
```
public class FullTimeEmployee
{
    float YearlySalary;
}
```

```
public class PartTimeEmployee
{
    float HourlyRate;
}
```

FullTime and PartTime employee specific code in the respective derived classes

PRAGIM Technologies | www.pragimtech.com | 9900113931

http://csharp-video-tutorials.blogspot.com

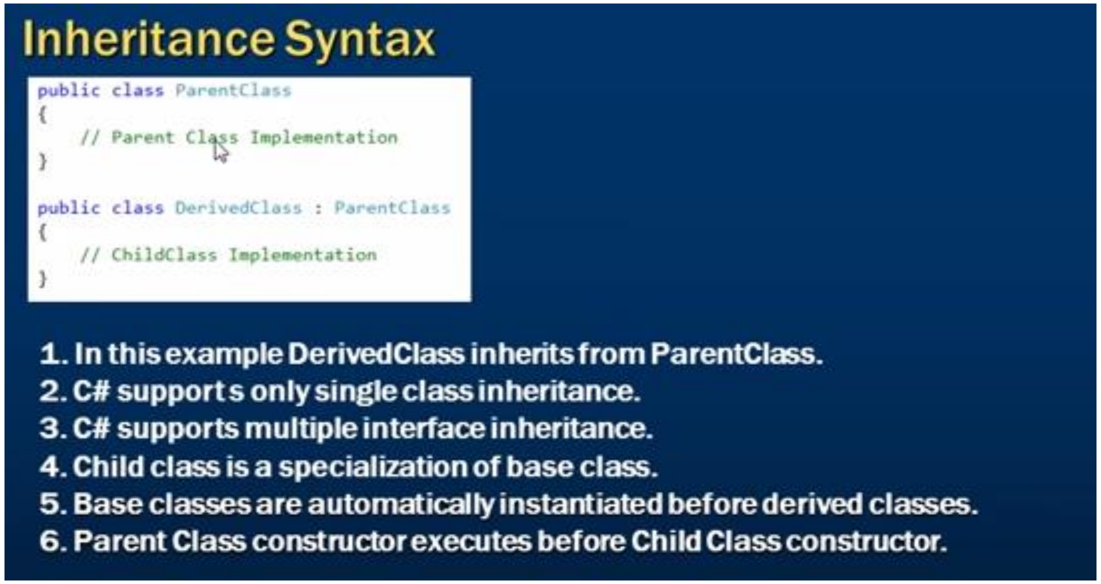

Why Inheritance

Pillars of Object Oriented Programming

1. Inheritance
2. Encapsulation
3. Abstraction
4. Polymorphism

1. Inheritance is one of the primary pillars of object oriented programming.
2. It allows code reuse.
3. Code reuse can reduce time and errors.

Note: You will specify all the common fields, properties, methods in the base class, which allows reusability. In the derived class you will only have fields, properties and methods, that are specific to them.



Inheritance Syntax

```
public class ParentClass
{
    // Parent Class Implementation
}

public class DerivedClass : ParentClass
{
    // ChildClass Implementation
}
```

1. In this example DerivedClass inherits from ParentClass.
2. C# support s only single class inheritance.
3. C# supports multiple interface inheritance.
4. Child class is a specialization of base class.
5. Base classes are automatically instantiated before derived classes.
6. Parent Class constructor executes before Child Class constructor.

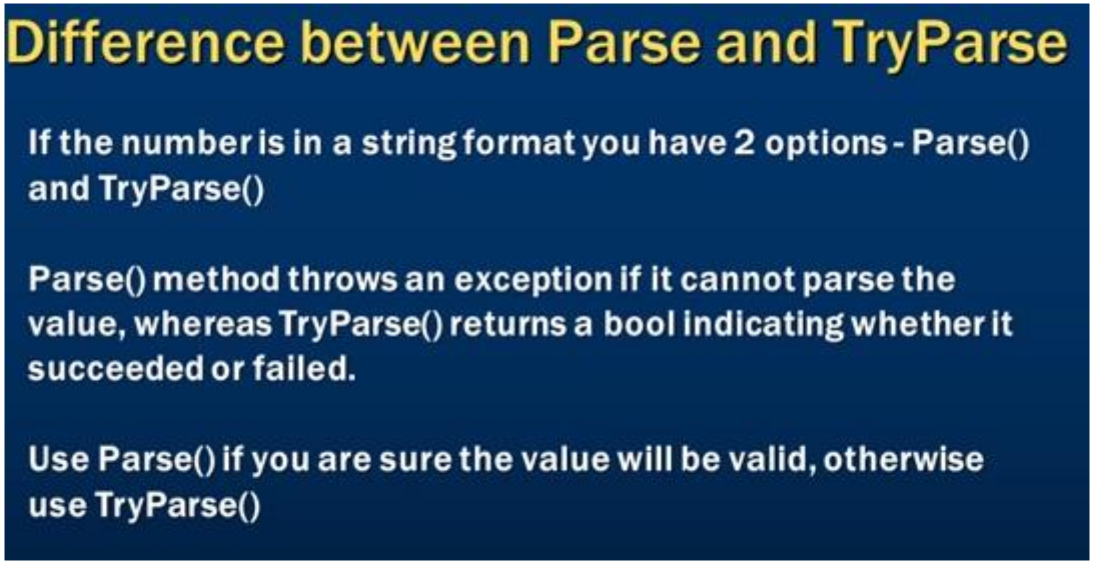
Difference between Convert.ToString and ToString()

- Convert.ToString handles null.
- ToString does not handle null. Throw null reference exception.

Data type convertions in C3

- Implicit
- Explicit
  - Use castint (int) (float). This will not throw exception
  - Convert.ToInt32(). Will throw exception.

Conversion from string to number



Difference between Parse and TryParse

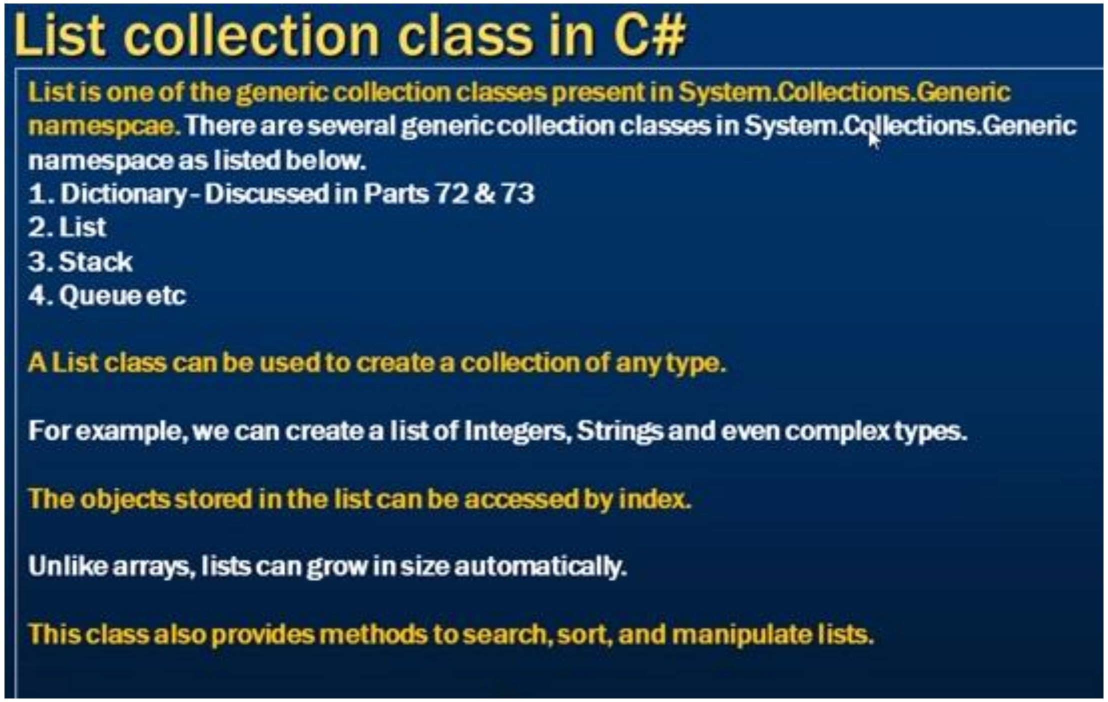
If the number is in a string format you have 2 options - Parse() and TryParse()

Parse() method throws an exception if it cannot parse the value, whereas TryParse() returns a bool indicating whether it succeeded or failed.

Use Parse() if you are sure the value will be valid, otherwise use TryParse()

Generics

List Class



List collection class in C#

List is one of the generic collection classes present in System.Collections.Generic namespcae. There are several generic collection classes in System.Collections.Generic namespace as listed below.

1. Dictionary - Discussed in Parts 72 & 73
2. List
3. Stack
4. Queue etc

A List class can be used to create a collection of any type.

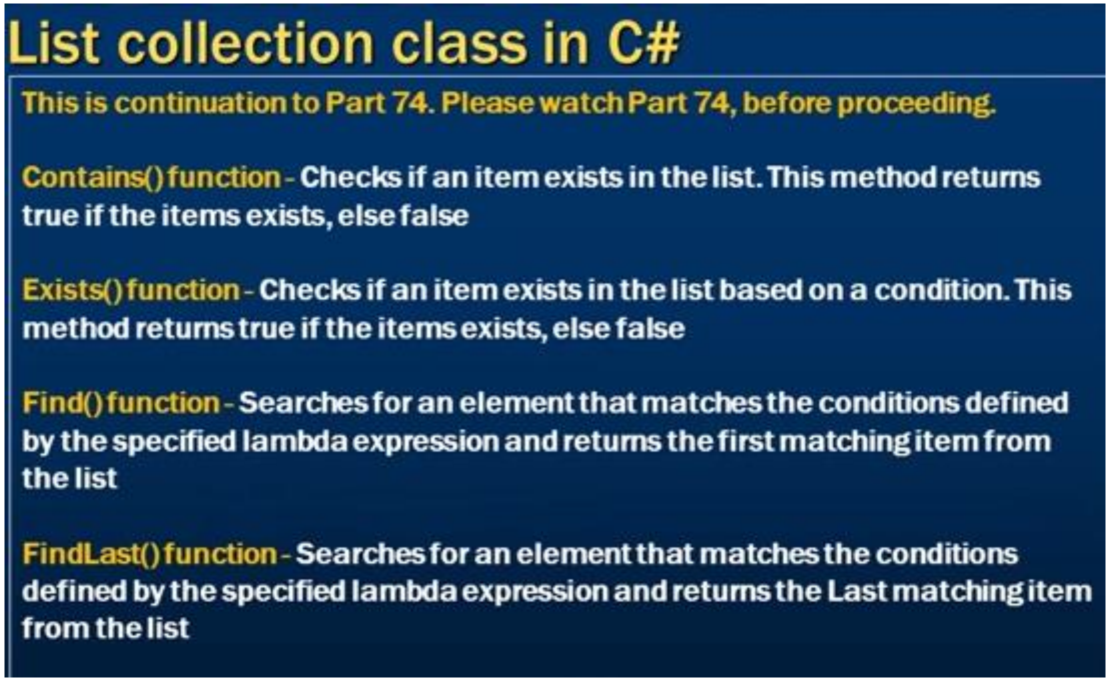
For example, we can create a list of Integers, Strings and even complex types.

The objects stored in the list can be accessed by index.

Unlike arrays, lists can grow in size automatically.

This class also provides methods to search, sort, and manipulate lists.

List Class



List collection class in C#

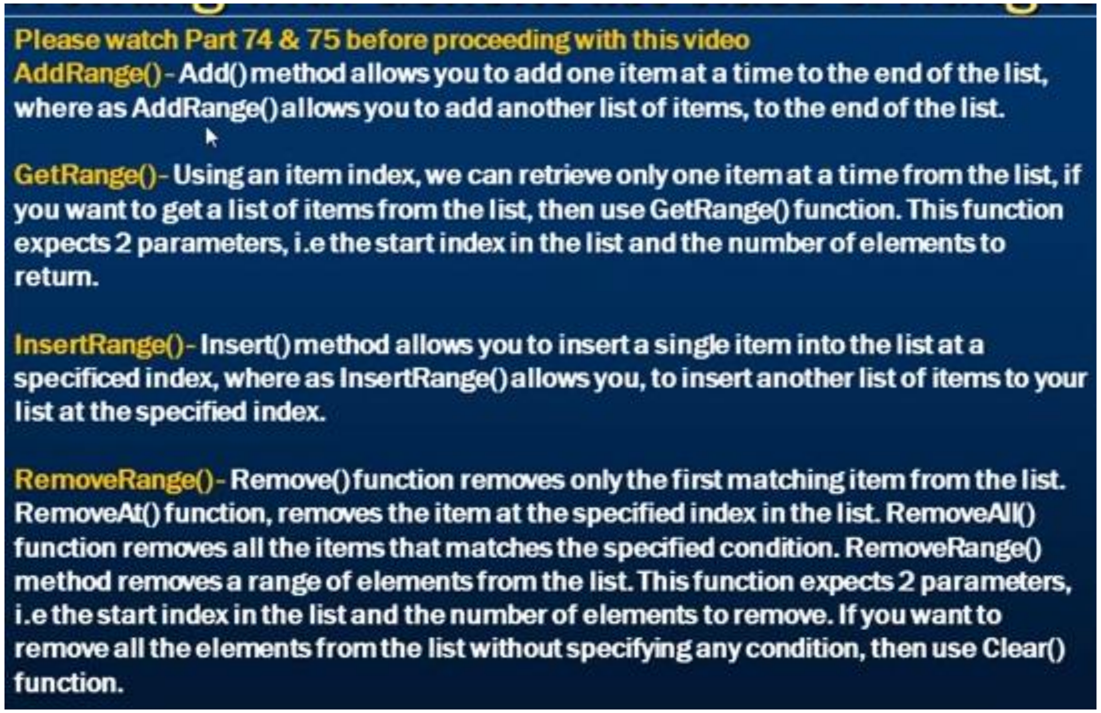
This is continuation to Part 74. Please watch Part 74, before proceeding.

Contains() function - Checks if an item exists in the list. This method returns true if the items exists, else false

Exists() function - Checks if an item exists in the list based on a condition. This method returns true if the items exists, else false

Find() function - Searches for an element that matches the conditions defined by the specified lambda expression and returns the first matching item from the list

FindLast() function - Searches for an element that matches the conditions defined by the specified lambda expression and returns the Last matching item from the list


Please watch Part 74 & 75 before proceeding with this video

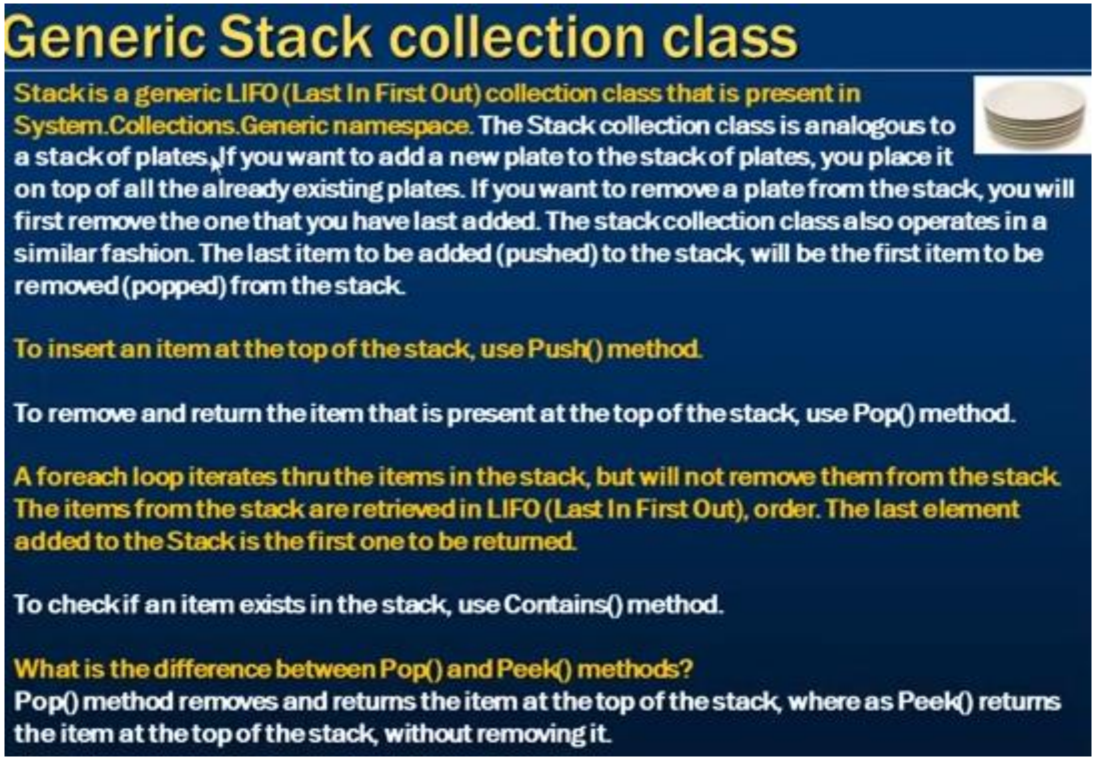
AddRange() - Add() method allows you to add one item at a time to the end of the list, where as AddRange() allows you to add another list of items, to the end of the list.

GetRange() - Using an item index, we can retrieve only one item at a time from the list, if you want to get a list of items from the list, then use GetRange() function. This function expects 2 parameters, i.e the start index in the list and the number of elements to return.

InsertRange() - Insert() method allows you to insert a single item into the list at a specificed index, where as InsertRange() allows you, to insert another list of items to your list at the specified index.

RemoveRange() - Remove() function removes only the first matching item from the list. RemoveAt() function, removes the item at the specified index in the list. RemoveAll() function removes all the items that matches the specified condition. RemoveRange() method removes a range of elements from the list. This function expects 2 parameters, i.e the start index in the list and the number of elements to remove. If you want to remove all the elements from the list without specifying any condition, then use Clear() function.

Stack



Generic Stack collection class

Stack is a generic LIFO (Last In First Out) collection class that is present in System.Collections.Generic namespace. The Stack collection class is analogous to a stack of plates. If you want to add a new plate to the stack of plates, you place it on top of all the already existing plates. If you want to remove a plate from the stack, you will first remove the one that you have last added. The stack collection class also operates in a similar fashion. The last item to be added (pushed) to the stack, will be the first item to be removed (popped) from the stack.

To insert an item at the top of the stack, use Push() method.

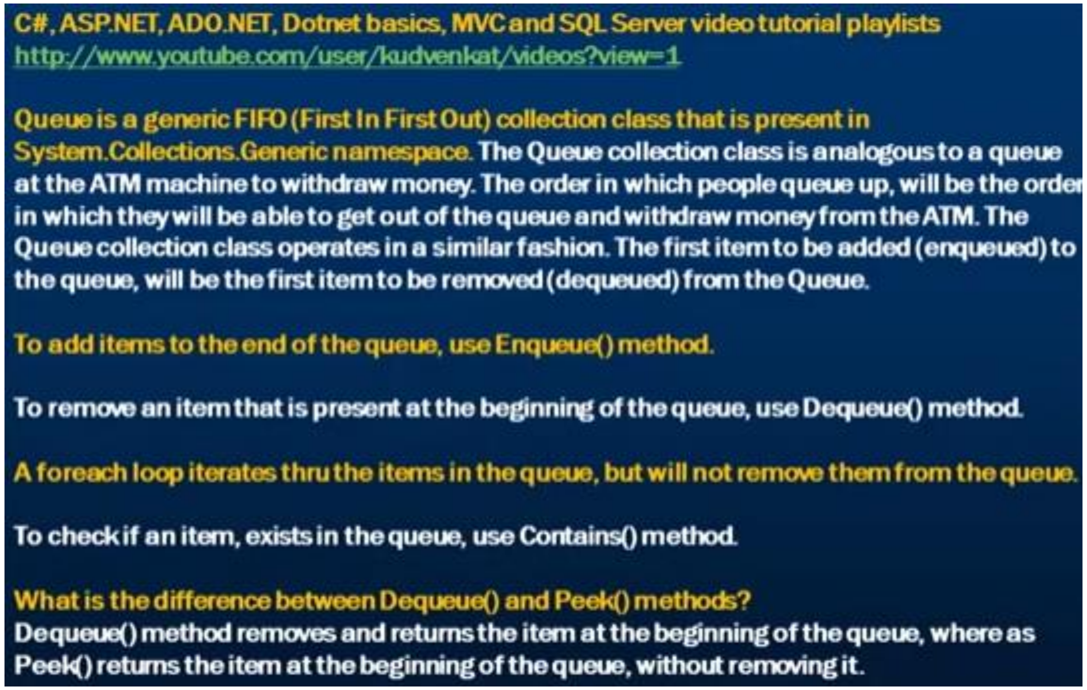
To remove and return the item that is present at the top of the stack, use Pop() method.

A foreach loop iterates thru the items in the stack, but will not remove them from the stack. The items from the stack are retrieved in LIFO (Last In First Out), order. The last element added to the Stack is the first one to be returned.

To check if an item exists in the stack, use Contains() method.

What is the difference between Pop() and Peek() methods?
Pop() method removes and returns the item at the top of the stack, where as Peek() returns the item at the top of the stack, without removing it.

Queue



C#, ASP.NET, ADO.NET, Dotnet basics, MVC and SQL Server video tutorial playlists
http://www.youtube.com/user/kudvenkat/videos?view=1

Queue is a generic FIFO (First In First Out) collection class that is present in System.Collections.Generic namespace. The Queue collection class is analogous to a queue at the ATM machine to withdraw money. The order in which people queue up, will be the orde in which they will be able to get out of the queue and withdraw money from the ATM. The Queue collection class operates in a similar fashion. The first item to be added (enqueued) to the queue, will be the first item to be removed (dequeued) from the Queue.

To add items to the end of the queue, use Enqueue() method.

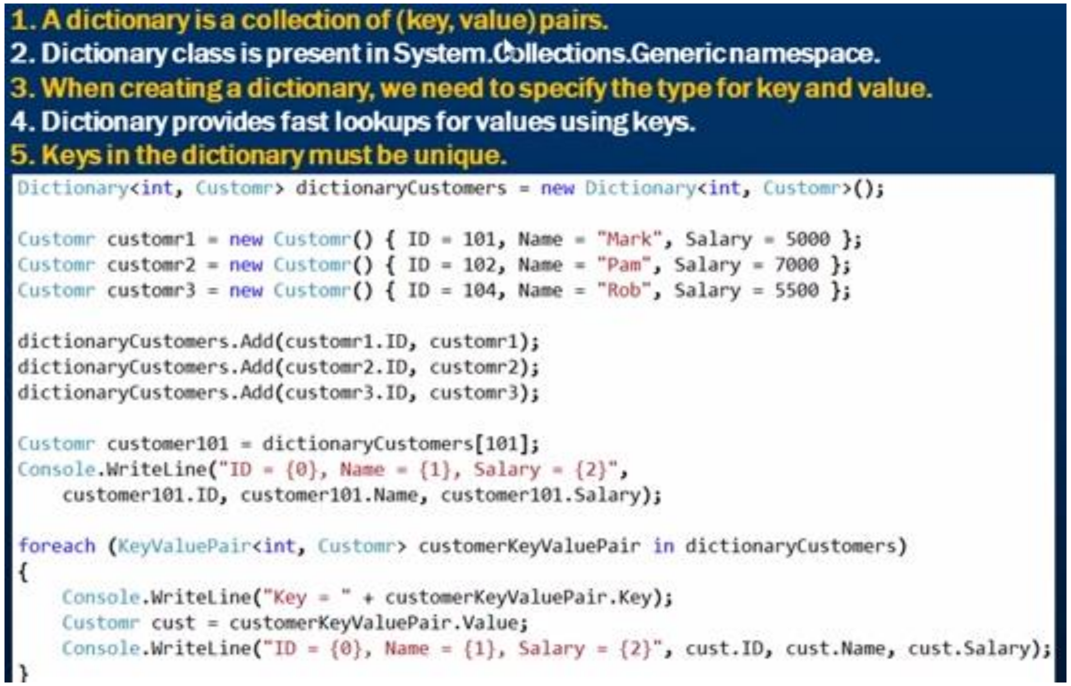
To remove an item that is present at the beginning of the queue, use Dequeue() method.

A foreach loop iterates thru the items in the queue, but will not remove them from the queue.

To check if an item, exists in the queue, use Contains() method.

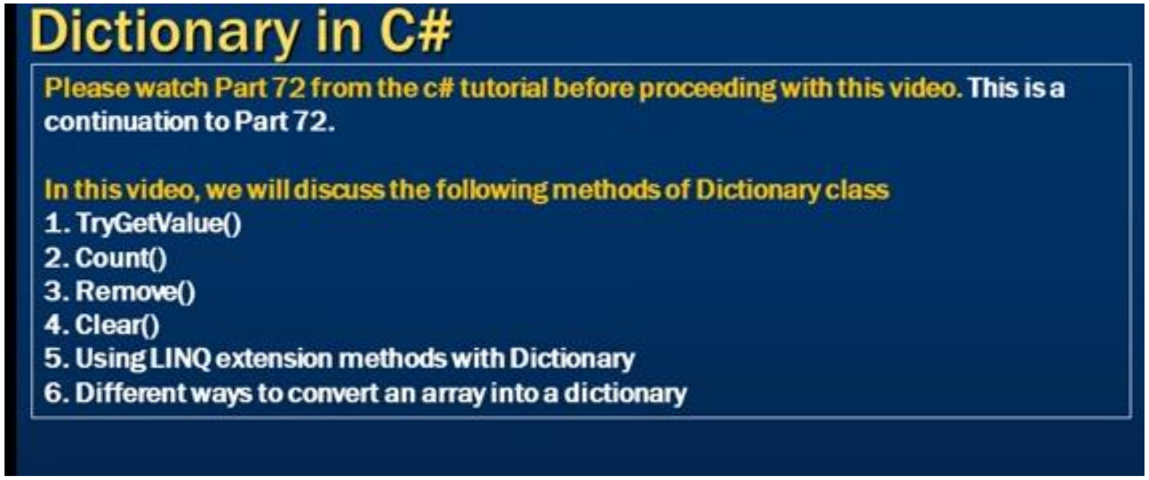
What is the difference between Dequeue() and Peek() methods?
Dequeue() method removes and returns the item at the beginning of the queue, where as Peek() returns the item at the beginning of the queue, without removing it.

Dictionary



1. A dictionary is a collection of (key, value) pairs.
2. Dictionary class is present in System.Collections.Generic namespace.
3. When creating a dictionary, we need to specify the type for key and value.
4. Dictionary provides fast lookups for values using keys.
5. Keys in the dictionary must be unique.

```
Dictionary<int, Customr> dictionaryCustomers = new Dictionary<int, Customr>();

Customr customr1 = new Customr() { ID = 101, Name = "Mark", Salary = 5000 };
Customr customr2 = new Customr() { ID = 102, Name = "Pam", Salary = 7000 };
Customr customr3 = new Customr() { ID = 104, Name = "Rob", Salary = 5500 };

dictionaryCustomers.Add(customr1.ID, customr1);
dictionaryCustomers.Add(customr2.ID, customr2);
dictionaryCustomers.Add(customr3.ID, customr3);

Customr customer101 = dictionaryCustomers[101];
Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}",
    customer101.ID, customer101.Name, customer101.Salary);

foreach (KeyValuePair<int, Customr> customerKeyValuePair in dictionaryCustomers)
{
    Console.WriteLine("Key = " + customerKeyValuePair.Key);
    Customr cust = customerKeyValuePair.Value;
    Console.WriteLine("ID = {0}, Name = {1}, Salary = {2}", cust.ID, cust.Name, cust.Salary);
}
```



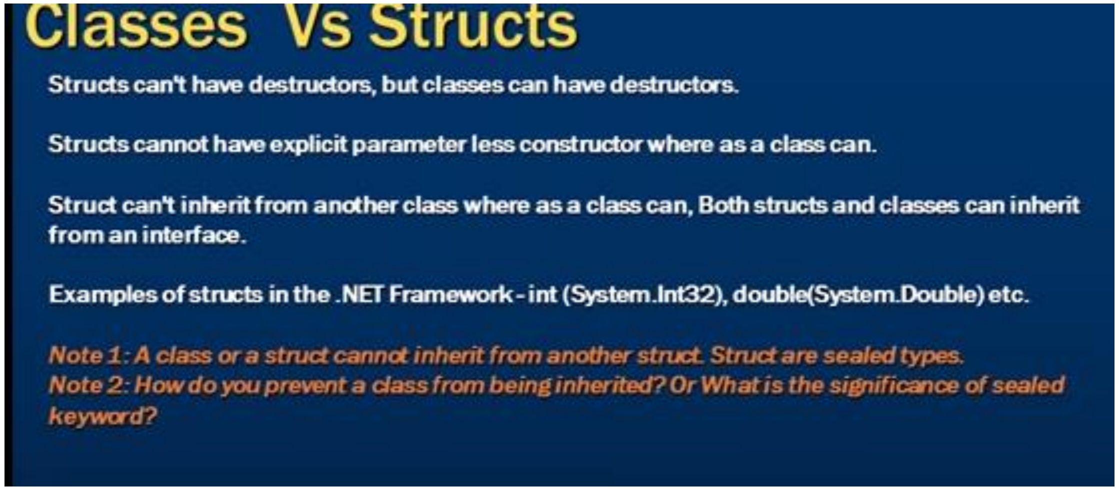
Dictionary in C#

Please watch Part 72 from the c# tutorial before proceeding with this video. This is a continuation to Part 72.

In this video, we will discuss the following methods of Dictionary class

1. TryGetValue()
2. Count()
3. Remove()
4. Clear()
5. Using LINQ extension methods with Dictionary
6. Different ways to convert an array into a dictionary

Difference between class and structs



Classes Vs Structs

Structs can't have destructors, but classes can have destructors.

Structs cannot have explicit parameter less constructor where as a class can.

Struct can't inherit from another class where as a class can, Both structs and classes can inherit from an interface.

Examples of structs in the .NET Framework - int (System.Int32), double(System.Double) etc.

Note 1: A class or a struct cannot inherit from another struct. Struct are sealed types.
Note 2: How do you prevent a class from being inherited? Or What is the significance of sealed keyword?

- rem test.bat
- @echo off
- MainReturnValTest
- @if "%ERRORLEVEL%" == "0" goto good

- :fail
-     echo Execution Failed
-     echo return value = %ERRORLEVEL%
-     goto end

- :good
-     echo Execution succeeded
-     echo Return value = %ERRORLEVEL%
-     goto end

- :end

Types in C#

- C# has two distinct groups of data types, called **Value Types and Reference Types.**
- Value Types directly contain their own data.
- Reference Types are accessed indirectly using reference variables that point to the object
- All Types in C#, both value types and reference types inherit from the object super class

Types

- Value Type can be:
  - Simple Type
  - Structs
  - enums
- Reference Type can be:
  - Classes
  - Interfaces
  - delegates
  - arrays
- C# provides a set of predefined struct types, called simple types, identified using reserved words.

Value Types

- C# provides a set of predefined struct types, called simple types, identified using reserved words.
- These reserved words are aliases for predefined struct types in System namespace.
- Simple types can be divided into four categories:
  - Integral Types
  - Floating point Types
  - Decimal Type
  - Bool Type

Integral Types Table
Following table shows the sizes and ranges of the integral types, which constitute a subset of simple types.

Type	Range	Size
sbyte	-128 to 127	Signed 8-bit integer
byte	0 to 255	Unsigned 8-bit integer
char	U+0000 to U+ffff	Unicode 16-bit character
short	-32,768 to 32,767	Signed 16-bit integer
ushort	0 to 65,535	Unsigned 16-bit integer
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint	0 to 4,294,967,295	Unsigned 32-bit integer
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

The following table shows the keywords for built-in C# types, which are aliases of predefined types in the System namespace.

C# Type	.NET Framework Type
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

Default Values Table

Value type	Default value
bool	false
byte	0
char	'\0'
decimal	0.0M
double	0.0D
enum	The value produced by the expression (E)0, where E is the enum identifier.
float	0.0F
int	0
long	0L
sbyte	0
short	0
struct	The value produced by setting all value-type fields to their default values and all reference-type fields to null.
uint	0
ulong	0
ushort	0

Checked and unckecked

- No overflow exception for integer arithmetic operations.
- Use **checked** statement to handle exceptions.

Floating Point Types

- Floating Point Types

  C# supports two Floating Point Types:

float	32 bits	7 bits precision
double	64 bits	15 bits precision

- Floating point types also provides a set of values:
  - Positive and Negative Infinity
  - NaN (Not a Number), these are produce as a result of invalid floating-point operations

Decimal

- Decimal Type

  Decimal type is suitable for financial and monetary calculation.

Decimal 128 bits 1.0 X 10^{-28} to 7.9 X 10^{28}

28 - 29 significant bits

- If one of the operator of a binary operator is of type decimal, then the other operator must be a integral part or of type decimal.

bool

bool Type represents Boolean values.

- Possible values of type bool are :
  - true
  - false
- No standard conversion is exist between bool and other types
- Bool value cannot be used in places of an integral value and vise versa

Literals in C#

- Literal is a textual representation of a particular value.
- Literals give hint to compiler and programmers about their type.
- The compilers will interpret them as specific type.

Literals

char c1 = ‘\u0066’

char c2 = ‘\t’   (tab)

int value = 0x21; //33 in decinmal

Literals

- F – float e.g. 1.33F
- D –Double e.g. 196.89D
- M –Decimal e.g. 9.99M
- L – Long   e.g. 12L

String type

- String type is built in reference type in C#
- Escape sequence \ is used for special characters.

Verbatim Literal:

- Verbatim literal is a string with an @ symbol prefix. E.g. @”Hello to C#”
- Verbatim literals make escape sequence translate as normal printable characters to enhance readability.

Type conversion

- **Implicit conversions**: No special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.
- **Explicit conversions (casts)**: Explicit conversions require a cast operator or use **Convert** class.

Convertion

Implicit numeric conversion

From	To
sbyte	short, int, long, float, double or decimal
byte	short, ushort, int, unit, long, ulong, float, double or float
short	int, long, float, double, or decimal
ushort	int, uint, long, ulong, float, double, or decimal
int	long, float, double, or decimal
uint	long, ulong, float, double, or decimal
long	float, double, or decimal
ulong	float, double, or decimal
float	double

Arithmetic Operators

- + Addition
- - Subtratction
- * Multiplication
- / Division
- % Modulus
- += Increment
- -= Decrement

Relational Operators

- == Equal to
- != Not equal to
- > Greater than
- < Less than
- >= Greater than or equal to
- <= Less then or equal to

Logical Operators

• &	And
• |	OR
• ^	XOR
• ||	Short-circuit OR
• &&	Short-circuit AND
• !	NOT

Identifier

- In C# identifier is a name assigned to a method, a variable, or any other user-defined item.
- Identifier can be one or more characters long.
- Identifier may start with any letter of the alphabet or an underscore. Next may be a letter, a digit, or an underscore.
- Uppercase and lowercase are different.

Reference Type

- The following keywords are used to declare reference types:
- class
- interface
- delegate
- C# also provides the following built-in reference types:
- object
- string

Difference between Value and Reference Type

Value type are stored in **Stack.**

Reference type are stored in **Managed Heap.**

Value types are destroyed immediately after the scope is lost. For reference variable, only the reference variable is destroyed after scope is lost. The object is later destroyed by Garbage Collector.

Void type

- When used as the return type for a method, **void** specifies that the method doesn't return a value.
- **void** isn't allowed in the parameter list of a method. A method that takes no parameters and returns no value is declared as follows:

```
public void SampleMethod(){    // Body of the method.}
```

Microsoft Visual Studio 2010

- Solution is collection of projects.
- Solution can contain multiple projects.
- If the solution contain multiple projects, select particular project as startup project to run that project.
- Select particular class as startup to run particular class within the project.

Classes and objects

Class is template for creating objects.

Creating class is creating new type in C#

Class is reference type.

Purpose of create class is encapsulate attributes and behaviour of an entity in single unit.

How to create class

access-modifier class *Class-Name*

{

}

Class members

```
Class
{
        instance variables (fields)
        static members
        constants
        constructors
        destructors
        properties
        indexers
        delegates
        events
        operators
}
```

object

- The process of creating object from a class is called Instantiation.
- Objects can be instantiated in two ways:

```
Box mybox = new Box();
(or)
Box mybox;
mybox = new Box();
```

- When an object is declared its value is NULL

C# Strongly typed

Strongly Typed

C# is a strongly-typed language.

Every variable and constant has a type, as does every expression that evaluates to a value.

Every method signature specifies a type for each input parameter and for the return value.

Arrays

- In C#, arrays is a data structure.
- Array is a collection of similar data types.
- Array resides in consecutive memory locations.
- The individual array elements are distinguished by an index value, which starts with zero.
- This means, the highest index number of an array is always the no of elements minus one.

ARRAYS

- Advantages :

  Arrays are strongly typed.

- Disadvantages:

  Arrays can not grow in size once initialized.

  Have to rely on integral indices to store and retrieve items from the array

ARRAY’S In C#

- In C#, array element type is any one of the primitive data type or any reference type.

- In C# arrays are treated as objects having System.Array as their base class.

- C# supports two types of arrays.
  - Single Dimensional Arrays
  - Multi dimensional Arrays

Single Dimension Array

- The general form for creating an array in C# is:

  Syntax: element-type[ ] name;

  name = new element-type[n];

  - Here, the first statement declares an array.
  - The second statement allocates memory space for n elements.

- ☞ C# allows you to have a declaration and allocation in a single statement like below:

  Syntax:

  element-type[ ] name = new element-type[n];

Single Dimension Array

- Examples:

```
float[ ] amt = new float[6];
char[ ] cx = new char[9];
string[ ] s;
s = new string[8];
box [] bx = new box[7];
```

- Once memory is allocated, all the elements of arrays are initialized to their default values.
- Single dimension array can be initialized while declaring or separately after declaring.
- We can find out the number of elements in an array using *Array.Length* property

Single Dimension Array

- Initialization

  Syntax:

```
type [] x = new type[n];
x [0]= e1;
x [1]= e2;
.
.
x [n-1]= en;
```

  EX:

```
int [] x = new int[2];
x[0]= 10;
x[1]=11;
```

Single Dimension Array

- Initialization

  Syntax:

  type[] x = {e1, e2, e3, ….eN};

  Here the size of the array is determined by the number of elements listed in { }.

  The above initialization can be done using new operator

  type[] x = new type[ ]{e1, e2, e3, ….eN};

  Alternative syntax can also be used:

  type[] x = new type[N]{e1, e2, e3, ….eN-1};

Single Dimension Array

- Iterating through array

```
int [ ] x = {1,2,3,4,5,6};

for (int i = 0; i<x.Length-1; i++)
        Console.WriteLine(x[i]);

foreach (int i in x)
        Console.WriteLine(i);
```

MULTI DIMENSIONAL ARRAY

- Multi Dimensional arrays can be either:
  - Rectangle Array
  - Jagged Array
- Multidimensional arrays are nothing but, array of arrays.
- In rectangular array, all the arrays at one level must have the same dimensions
- In Jagged array, all the arrays at one level need not have same dimensions

Rectangle Array

- Two dimensional array:

  Declaration:
  - type [,] xy = new type [m,n];
  - type [,] xy;
  - xy = new type [m,n];

  Initialization
  - Individual elements
  - type [,]xy = {{e1,e2}, {e3,e4}, …. {eN-1,eN}};
  - type [,]xy = new int[n, m]{{e1,e2}, {e3,e4}, …. {eN-1,eN}};

Rectangle Array

- Rectangle Array

  Initializing example:
  - int [,] xy = new int[2,2];   // length 4, rank 2

    xy[0,0] = 34;   xy[0,1] = 24;

    xy[1,0] = 44;   xy[1,1] = 45;

  - int [,] xy = {{24,34},{44,45}};

  - int [,] xy = new int [2,2] {{24,34},{44,45}};
- Length property of returns the total number of elements in the array.

Rectangle Array

- Three dimensional array:

  int [, ,] xyz = new int [n1, n2, n3];

- Four dimensional array:

  int [, , ,] xyz = new int [n1, n2, n3, n4];

- The number of commas in the declaration will be one less than the dimensions (rank) of the array.
- We can find out the rank of an array using *Array.Rank* property

Jagged Array

- Jagged arrays are used when we need multidimensional array, but where the size of internal arrays is not same.
- Declaration:

type [] [] x;	Declares a single dimensional array of single dimensional arrays.
type [] [,] x;	Declares a single dimensional array of two dimensional arrays.
type [] [,] [] x;	Declares a single dimensional array of two dimensional arrays of single dimensional arrays.

Jagged Array

- Example:

```
int [] [] x = new int [4] [];          // length is 4, rank is 1
x [0] = new int [3]{1,2,3};
x [1] = new int [2]{1,2};
x [2] = new int [5]{1,2,3,4,5};
x [3] = new int [7]{1,2,3,4,5,6,7};
```

- Jagged arrays can not be initialized while declaring them like other arrays.
- Rank property of a jagged array return the rank of the first array.

```
int [,] [] [,,] x ;          //Rank of this array is 2
```

Jagged Array

- Iterating through array

```
int [] [] x = new int [3] [];
x [0] = new int [3]{1,2,3};
x [1] = new int [2]{1,2};
x [2] = new int [5]{1,2,3,4,5};
```

Statements

Statements

- C# programs are made up of classes which consists of properties and methods, which are in-turn a set of statements.
- C# provides a variety of statements:

Labeled statement	Declaration statement
expression statement	selection statement
iteration statement	jump statement
try statement	checked statement
unchecked statement	
using statement	

Statements

- End Point

  End point of a statement is the location that immediately follows the statement.
- Reachability

  If a statement can possibly be reached by execution, the statement is said to be reachable.

  If there is no possibility that a statement will be executed, the statement is said to be unreachable.

  Compiler generates warning when it come across a unreachable statement.

Statements

- Block

  A block is used to execute a set of statements as a single unit of execution.

  Block statements are kept between “{” and “}”.

  Blocks can be empty { }.

Statements

- Labeled Statements

  Statements that are prefixed with a label are called Labeled Statements.

  Syntax: *label*: *statement*

  Ex: Begin: x = 0;

  The scope of the label is the block in which it is declared, including any nested blocks.

  Within the scope of the label, a label can be referred using goto statement.

  Ex: goto Begin;

Statements

- Declaration Statements

  A declaration statement declares a local variable or Local constant.

  Variables and constants declared in a block become local to that block.

  Local variable declaration

  Syntax: *type identifier = expression*$_{opt}$

type	- value or reference type
identifier	- name of the variable

  Scope of a local variable starts immediately after its declaration, extends to the end of block.

Statements

- Declaration Statements

  Local constant declaration

  Syntax:

  const *type identifier* = *constant-expression*

const	- keyword
type	- value or reference type
identifier	- name of the constant

  Scope of a local constant starts immediately after its declaration, extends to the end of block.

  Within a scope its an error to declare more than one variable or constant with the same name.

Selection Statements

- C# supports two types of selection statements.
  - **if**
  - **switch**
- This statements allow you to control the flow of your program's execution based upon conditions known only during run time.
- A C# *if* statement is a test of any Boolean expression
- It can be used to route program execution through two different paths.

Selection Statements

- *if* statement

  General form:

  ```
  if (boolean expression)
          statement or block

  if (boolean expression)
          statement or block
  else if_opt
          statement or block
  ```
- We can also have nested if statements.

Selection Statements

- *if* statement

  Ex:

```
if (x >0)
        Console.WriteLine(“x is positive”);
else if (x < 0){
        Console.WriteLine(“x is negative”);
        if (x < 100)
                Console.WriteLine(“x is too small”);
}
else
        Console.WriteLine(“x is zero”);
```

Selection Statements

- *Switch* statement

  It is ideal for testing a single expression against a series of possible values and executing the code associated with the matching case statement.
- The general form is:

```
switch (expression) {
        case value1:  statements opt
                      break;
        case value2:  statements opt
                      break;
        ..
        case valueN:  statements opt
                      break;
        default:      statements opt
}
```

Selection Statements

- *Switch* statement

Ex:

```
int i = 2;
switch(i) {
        case 0: Console.WriteLine(“i is Zero”);
                break;
        case 1: Console.WriteLine(“i is one”);
                break;
        case 2: Console.WriteLine(“i is Two”);
                break;
        default: Console.WriteLine(“i is > 2);
}
```

- The statement list of each case typically ends with break, goto case, or goto default statement.

Selection Statements

- *Nested switch* statement

  You can use a switch as part of the statement sequence of an outer switch.

```
switch(expression) {
        case 1:switch(expression) {
                    case 1: /* statements */ break;
                    case 2: /* statements */ break;
                    default /* statements */
               }
               break;
        case 2:statements
               break;
        default:
               statements
}
```

Iteration Statements

- C# supports four types of iteration statements. They are :
  - while statement
  - do statement
  - for statement
  - foreach statement
- Normally, these statements are called as loops.
- A loop repeatedly executes the same set of instructions until a termination condition is met.

Iteration Statements

- While statement

  It repeats a statement or block while its controlling expression is true.

  General form is:

```
while(condition)
        statement opt
        block opt
```

Ex:

```
int i = 0;
while ( i < 10){
        Console.WriteLine(i);
        i ++;
}
```

Iteration Statements

- Do - While statement

  do statement guarantees the execution of the statements atleast once unlike while statement.

  General form is:

```
do {
        statements opt
} while(condition) ;
```

Ex:

```
int i = 0;
do {
        Console.WriteLine(i);
        i ++;
} while ( i < 10);
```

Iteration Statements

- For statement
  - evaluates a sequence of initialization expressions and
  - while a condition is true, repeatedly executes an embedded statement and
  - evaluates a sequence of iteration expressions

General form is:

for (*initializer*$_{opt}$; *condition* $_{\text{opt}}$; *iterator* $_{\text{opt}}$)

Statement

block

Ex:

```
for (int i =0 ; i < 10; i++)
        Console.WriteLine(i);
```

Iteration Statements

- For statement
- initializer if present, consists of local variable declaration.
- condition, if present, must be a boolean expression
- condition, if not present , will be treated as equivalent to *true*
- iterator, if present, consists of a list statement-expressions separated by commas

Iteration Statements

- Foreach statement

  foreach statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

  General Form:

  foreach (*type identifier* in *collectionType*)

  statement

  block

type identifier	-	iteration variable, acts as a local variable for the embedded statement

Iteration Statements

- Foreach statement

type identifier	-	Any attempt to modify this iteration variable will generate a compile type error.
collectionType	-	an object with a public instance method GetEnumerator(). Arrays in C# are also of type collection.

Ex:

```
int[] ia ={1,2,3,4};
foreach (int i in ia)
        Console.Write(i);
```

Output:

```
1234
```

Jump Statements

- Jump statements are used to transfer control to another part of the program.
- C# language supports five types of jump statements:
  - break
  - continue
  - goto
  - return
  - throw

Jump Statements

- Break

  The break statement exits the nearest enclosing switch, while, do, for or foreach statement.

  General form: break;

  Ex:

```
int i = 0;
while (i < 10){
        Console.Write(i);
        if (i == 5)
                break;
        i++;
}
```

  Output:

```
012345
```

Jump Statements

- Continue

  The continue statement starts a new iteration of the nearest enclosing switch, while, do, for or foreach statement.

  General form: continue;

  Ex:

```
int i = 0;
while (i < 10){
        i++;
        if ((i % 2) == 0)
                continue;
        Console.Write(i);
}
```

  Output:

```
13579
```

Jump Statements

- goto

  goto statement transfers control to a statement market by a label.

  General form: goto *label*;

  goto case *const-expression*;

  goto default;

  - *goto case* and *goto default* are used inside a switch statement.
  - *goto case* transfers the control to the case whose value is mentioned after case.
  - *goto default* transfers the control to the default case of the switch statement

Jump Statements

- Goto

Ex:
```
int i = 0;
while (i < 10){
        i++;
        if (i == 5)
                goto finish;
        Console.Write(i);
}
finish: Console.Write(“ - Values are displayed”);
```

Output:
```
1234 - Values are displayed
```

Properties

INTRODUCTION

- Properties can be one of the member of classes, structs, and interfaces.
- From an application developer’s view, properties behave very much like function variables.
- Developers can read, write or compute property values as if they are variables, but properties are more powerful than variables for two reasons:

1. **Application developers can set properties at design time.**
2. **Properties can hide implementation details.**

PROPERTIES

- Properties are an extension of instance variables.
- Properties can be accessed like instance variable. i-e using the dot operator.

DECLARATION

☞ Syntax:

modifier data-type identifier

{accessor-declaration}

modifiers	Optional. Valid modifiers are: new, static, virtual, abstract, override, public, private, internal and protected.
type	indicates the data type of the property.
identifier	indicates the name of the property.
accessor-declaration	is the declaration part of the property. It is used to read and write the property.

ACCESSOR DECLARATION

- The accessor of a property contains the executable statements associated with getting (reading or computing) or setting (writing) the property.
- The accessor declarations can contain a **get** accessor, a **set** accessor, or both. The declarations take the following forms:

  set{   accessor body }

  get{   accessor body }

- *accessor body* is the block, which contains the statements to be executed when the accessor is invoked

GET ACCESSOR

- The body of the **get** accessor is similar to that of a method.
- It must return a value of the property type.
- The execution of the **get** accessor is equivalent to reading the value of the field.
- When you reference the property, except as the target of an assignment, the **get** accessor is invoked to read the value of the property.
- The following is a **get** accessor that returns the value of a private field *address*:

GET ACESSOR EXAMPLE

```
class Customer{
      private string address;
      public string Address{
         get{
                 return address;
         }
      }
}
Customer c1 = new Customer();
Console.WriteLine(c1.Address); //  the get Accessor
                                   method is invoked
```

SET ACESSOR

- 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.
- When you assign a value to the property, the set accessor is invoked with an argument that provides the new value.
- In the following example, a set accessor is added to the *Address* property:

SET ACCESSOR EXAMPLE

```
class Customer{
    public string Address {
        get    {
                return address;          }
        set    {
                address = value;         }
    }
}

c1.Address = "7, II Main Road,UI Colony";
                              // set Accessor is called
```

IMPORTANT POINTS

- A property with a get accessor only is called a read-only property. You cannot assign a value to a read-only property.
- A property with a set accessor only is called a write-only property.
- A property with both get and set accessors is a read-write property.
- In a property declaration, both the get and set accessors must be declared inside the body of the property.

IMPORTANT POINTS

- It is a bad programming style to change the state of the object by using the get accessor.
- In C#, if the base class property and the derived class property has the same name, then derived class property hides the base class property.
- Whenever, a property is declared as abstract, then in the sub class property should override it using *override* keyword.

EXAMPLE

```
using System;
public class A {
   private string region;
   public string RegionX   {      get      { return region;   }
                                  set      {  region = value; }
   }
}

public class B : A {
   private string region;
   public new string RegionX { get     {   return region; }
                               set     {   region = value; }
   }
}
```

EXAMPLE

```
public class PropertyHideEx {
  public static void Main() {
      B d1 = new B();
      d1.RegionX = "North";
      Console.WriteLine("Region in Sub class: ”+d1.RegionX);
      ((A)d1).RegionX = "South";
      Console.WriteLine("Region in super class ”+
                                                            ((A)d1).RegionX );
  }
}
```

EXAMPLE

```
using System;
public abstract class A {
   public abstract int Depth   {      get;        set;    }
}
public class B : A {
       private int depth;
       public override int Depth {        get { return depth; }
                                          set { depth = value; }
       }
}
public class PropertyAbstractEx {
   public static void Main()    {
      B b1 = new B();      b1.Depth = 12;
      Console.WriteLine("Depth is : {0}",b1.Depth);   }
}
```

PROPERTIES IN INTERFACES

- Properties can be declared on interfaces.
- The body of the accessor of an interface property consists of a semicolon.
- The purpose of the accessors is to indicate whether the property is read-write, read-only, or write-only.
- The general form properties in interfaces is:

  new data-type identifier {interface accessors}
- If the two interfaces has the same property, which is implemented in the same class, then you have to qualify the property using the interface name.

EXAMPLES

```
using System;
public interface IDemo
{   string Name {get;   set;} }

public interface ITest
{   string Name {get;   set;} }

public class A : IDemo, ITest {
 private string cname="John", sname="Dave";
   string IDemo.Name  {        get {   return cname;    }
                               set {   sname = value; }   }
   string ITest.Name  {        get {   return sname;    }
                                set {  sname = value; }   }
}
```

EXAMPLES

```
public class PropertyInterfaceEx {
    public static void Main() {
        IDemo d1 = new A();
        d1.Name = "Customer";
        Console.WriteLine("Name is: {0}",d1.Name);

        ITest d2 = new A();
        d2.Name = "Customer";
        Console.WriteLine("Name is: {0}",d2.Name);
    }
}
```

Classes And Objects

Classes

- Creating a class is equivalent to creating a new Type in C#
- Structure of a class

  class *classname{*

  class-members

  }
- Class members constitute of fields, constants, methods, properties, events, indexers, operators, constructors, destructors and Types

Fields

- Fields are variables associated with a class.
- Fields are also referred as *instance variables.*

```
using System;
class Box {
        public double width;
        public double height;
        public double depth;}
```

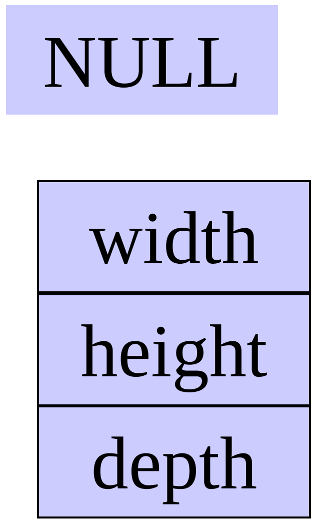
is equivalent to:

```
        using System;
        class Box {
                public double width, height, depth; }
```

Object

- The process of creating object from a class is called Instantiation.
- Objects can be instantiated in two ways:

```
Box mybox = new Box();
(or)
Box mybox;
mybox = new Box();
```



- When an object is declared its value is NULL

Example

```
using System;
class Box {
        public double width;  public double height;  public double depth;
}
class BoxDemo {
static void Main() {
        Box mybox = new Box();
        double vol;
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;
        vol = mybox.width * mybox.height * mybox.depth;
        Console.WriteLine(“Volume is “ + vol);}
}
```

Constants

- Constant represent a constant value in a class.
- Constants are computed at compile-time.

  Syntax: const *type identifier*= *const-expression*;

  Ex: const int amount = 100;
- Constant can itself participate in a constant expression

  Ex: const int total = amount+100;
- Constants are permitted to depend on other constants in the same program as long as there is no circular pointing.

Methods

- Methods represent the class behavior.

  Syntax:

  ```
  return-type method-name (parameter-list opt)
  {
     method-body;
  }
  ```

return-type	Method type specifies the type of value computed and returned by the method. Return type is void if the method does not return any value.

- Method name and parameter list put together is called the signature of the method.

Method with no return value

```
using System;
class Box {
        public double width;  public double height;  public double depth;
        public void volume ( ){
            console.WriteLine("Volume is" + (width * height *depth))
        }
}
class BoxDemo {
static void Main() {
        Box mybox = new Box();
        mybox.width = 10; mybox.height = 20; mybox.depth = 15;
        mybox.volume(); }
}
```

```
using System;
class Box {
      public double width;  public double height;  public double depth;
      public double volume ( ){
         return width * height *depth ;
      }
}
class BoxDemo {
static void Main() {
      Box mybox = new Box();
      mybox.width = 10; mybox.height = 20; mybox.depth = 15;
      double vol = mybox.volume();
      console.WriteLine("Volume is" +vol);}
}
```

Passing Arguments To Methods

- Parameter list to a method can be
  - fixed parameters
  - fixed parameters and parameter array
  - parameter array

  Ex:

  void calculate (int x, int y);

  void calculate (int x, int y, **params** int [ ] values);

  void calculate (**params** int [ ] values);

- If parameter array is used , we can pass single argument of the given array type, or zero or more arguments of the array element type

Passing Arguments To Methods

- Fixed parameters can be
  - Value parameters
  - Reference parameters
  - Out parameters
- Reference and Out parameters are declared using *ref* and *out* modifiers respectively.

Passing Arguments To Methods

- ☞ Value Parameters

  Parameters declared with no modifier prefixed to them are called value parameters.
- ☞ Value parameters are passed using Call-by-value.
- ☞ In call-by-value, the value of an argument is copied into the formal parameter of the method.
- ☞ The changes made to the parameter of the method have no effect on the argument used to call it.

Passing Arguments To Methods

```
using System;
class valparam {
        void change(int x){
             x = x * 10;
             Console.WriteLine(“Inside method change  x is”+x);
        }
        static void Main() {
             int  y = 10;
             valparam vp1 = new valparam();
             Console.WriteLine(“Before calling change  y is ” +y);
             vp1.change(y);
             Console.WriteLine(“After calling change  y is ” +y);}
}
```

Passing Arguments To Methods

- ☞ Reference Parameters

  Parameters can also be passed to the method using Call-By-Reference.
- ☞ In call-by-reference, the changes made to the Reference parameter will affect the actual argument passed.
- ☞ To pass parameters by reference, the modifier *ref* is prefixed to the parameters and arguments.

Passing Arguments To Methods

```
using System;
class refparam {
        void change(ref int x){
              x = x * 10;
              Console.WriteLine(“Inside method change  x is”+x);
        }
        static void Main() {
              int  y = 10;
              refparam vp1 = new refparam();
              Console.WriteLine(“Before calling change  y is ” +y);
              vp1.change(ref y);
              Console.WriteLine(“After calling change  y is ” +y);}
}
```

Passing Arguments To Methods

- ☞ Output Parameters

  Parameters declared using *out* modifier are called Output Parameters.
- ☞ Unlike value and reference parameters, arguments passed to output parameter need not be initialized before passing.
- ☞ Like reference parameters, arguments passed to output parameters are passed by reference.
- ☞ Output parameters are used when the method needs to return more than one value!.

Passing Arguments To Methods

```
using System;
class outparam {
      void change( out int x, out int y){
         x =10;
         y = 20;
      }
      static void Main() {
         int  a ;  // note that a is not initialized
         int  b;   // note that b is not initialized
         outparam op1 = new outparam();
         op1.change(out a, out b);
         Console.WriteLine(" a is "+ a +" and  b is "+b );}
      }
```

Passing Arguments To Methods

- ☞ Parameter Arrays

  Parameter arrays are declared using *params* modifier.
- ☞ If parameter list includes fixed parameters and parameter array, parameter array must be the last in the list.

Passing Arguments To Methods

```
using System;
class paramsparam {
        void disp( params int [] data){
           foreach (int i in data)
                       Console.WriteLine(i);
        }
        static void Main() {
           paramsparam pp1 = new outparam();
           pp1.disp(10,11,12);
           int [] val = new int[5]{11,12,13,14,15};
           pp1.disp(val);
        }
}
```

Passing Arguments To Methods

```
class allparams {

     void disp(int x, ref int y, out int z, params int[] data){
       x *= 10;
       y *= 10;
       z = x+y;
       Console.WriteLine("Data sent through parameter array");

       foreach(int i in data)
               Console.WriteLine(i);
       }
```

Passing Objects As Arguments To Methods

- ☞ Objects as parameters

  Objects can also be passed as parameters to methods.
- ☞ Object passed to a method are always passed by reference.
- ☞ Objects can also used to pass output parameters and parameter arrays.

Passing Objects As Arguments To Methods

```
using System;
class box{
        public double width, height, depth;

        public void increase(box b){
           b.width *=2;
           b.height *=2;
           b.depth *=2;
        }
}
```

Passing Objects As Arguments To Methods

```
class allparams {
static void Main() {
   box b1 = new box();
   b1.width =10;
   b1.height =20;
   b1.depth =30;
   Console.WriteLine("Box Details before calling increase");
   Console.WriteLine("width is "+b1.width);
   Console.WriteLine("height is "+b1.height);
   Console.WriteLine("depth is "+b1.depth);
```

Passing Objects As Arguments To Methods

```
        b1.increase(b1);
        Console.WriteLine("Box Details after calling increase");
        Console.WriteLine("width is "+b1.width);
        Console.WriteLine("height is "+b1.height);
        Console.WriteLine("depth is "+b1.depth);
    }
}
```

Output:

```
Box Details before calling increase
width is 10
height is 20
depth is 30
Box Details after calling increase
width is 20
height is 40
depth is 60
```

Constructors

- Constructor is a method of a class which is automatically called when a class is instantiated.
  - Constructor Will have the same name as class name
  - Constructors will not have any return type
  - Constructors will not have any return value
- Constructors are used to initialize the instance variables of a class
- A class can have more than one constructor, to take different arguments.
- Constructors are mostly public, we can also have private constructors, which means the class can’t be instantiated.

Constructors

- ☞ Ex:

```
using System;
class box{
   public double width, height, depth;
   public box(double w, double h, double d){
            width=w, height=h, depth=d;
   }
   static void Main(){
            box b1 = new box(10,20,30);
            Console.write(“width is:” + b1.width+“height is:”
            +b1.height+”depth is:”+ b1.depth);
   }
}
```

Constructors

- ☞ Ex:

```
using System;
class box{
   public double width, height, depth;
   public box(){
            width=10, height=20, depth=30;
   }
   public box(double w, double h, double d){
            width=w, height=h, depth=d;
   }
}
```

Instance Variable Hiding

☞ Ex:

```
using System;
class box{
   public double width, height, depth;
   public box(){
            width=10, height=20, depth=30;
   }
   public box(double width, double height, double depth){
            this.width=width;
            this.height=height;
            this.depth=depth;
   }
}
```

Private Constructors

☞ Ex:

```
using System;
class box{
   public double width, height, depth;
   private box(){
             width=10, height=20, depth=30;
   }
   private box(double width, double height, double depth){
             this.width=width;
             this.height=height;
             this.depth=depth;
   }
}
```

☞ Ex:

```
using System;
class box{
   public double width, height, depth;
   private box(){width=10, height=20, depth=30;}
   public static box getbox() { return new box(); }
}
class boxdemo{
   static void Main(){
            box b = new box();
            box b1 = box.getbox();
            Console.write("width is:" + b1.width+"height is:"
            +b1.height+"depth is:"+ b1.depth); }
}
```

ERROR

Optional Constructor Parameters

Ex:

```
using System;
class box{
        double width, height, depth;
        public box(): this (0,0,0) { }
        public box(double x):this(x,0,0) { }
        public box(double x, double y):this(x,y,0) { }
        public box(double x, double y, double z) {
            width=x; height=y;depth=z; }
        static void Main(String[] args){
            box b1 = new box();
            box b2 = new box(10);
            box b3 = new box(10,20);
            box b4 = new box(10,20,30); }
}
```

Default Constructor

- ☞ If a class doesn’t have a constructor, a default constructor is provided by complier.
- ☞ The default constructor simply invokes the parameter less constructor of the base class.

Static Constructor

- ☞ A constructor can be declared as static.
- ☞ A constructor declared as static will be executed as soon as the class is loaded.
- ☞ Like ordinary constructors, static constructors will have the same name as the class name, with no return type and no return value.

  Syntax: static *ClassName* ( )

  block
- ☞ Static constructors can not have any parameters.
- ☞ Static constructors cannot be called explicitly.

Class Loading

- ☞ A class is loaded before any instance of the class is created.
- ☞ A class is loaded before any of its static members are referenced.
- ☞ A class is loaded before any types that derive from it are loaded.
- ☞ A class cannot be loaded more than once during a single execution of a program.
- ☞ If a class has a static constructor, it is automatically called when the class is loaded.

Static Constructor

☞ **Ex**:

```
class one {     static one(){Console.WriteLine(“one constructor”);}
          static void meth1() {Console.WriteLine(“one meth1”); } }
class two {     static two(){Console.WriteLine(“two constructor”);}
        public void meth2() {Console.WriteLine(“two meth2”); } }
class demo{     static void Main(){      one.meth1();
                                         two t1 = new two();
                                         t1.meth2();          }
}
```

Output:

```
one constructor
one meth1
two constructor
two meth2
```

Destructors

- ☞ A destructor is a method in a class that is called when the object is destroyed.
- ☞ The process of calling destructor when the object is reclaimed by the garbage collection is called finalization.
- ☞ The name of a destructor method is same as the name of the class preceded by a '~' (tilde).
- ☞ As we are not sure when the object will be reclaimed by GC, it is advisable not to do anything significant in the destructor

Destructors

- Ex:

```
 using System;
 class box{
    public double width, height, depth;
    public box(double w, double h, double d){
               width=w, height=h, depth=d; }
    public ~box(){// this is executed when object is GC }
 }
```

- Like constructors destructors will not have any return type.
- Destructors don’t take any arguments.
- Destructors are always public.

Destructors

1. Destructors are invoked automatically, and cannot be invoked explicitly.
2. Destructors cannot be overloaded. Thus, a class can have, at most, one destructor.
3. Destructors are not inherited. Thus, a class has no destructors other than the one, which may be declared in it.
4. Destructors cannot be used with structs. They are only used with classes.

5. An instance becomes eligible for destruction when it is no longer possible for any code to use the instance.
6. Execution of the destructor for the instance may occur at any time after the instance becomes eligible for destruction.
7. When an instance is destroyed, the destructors in its inheritance chain are called, in order, from most derived to least derived.

Operator overloading

C# allows to define the meaning of an operator relative to a class that we create. This process is called operator overloading.

By overloading an operator, we can expand its usage to the class.

Example:

- Use + operator to add an element to linked list.
- Use + to push to stack and – to pop from stack.

Operator overloading

- Operator parameters must not use the ref or out modifiers.
- Operator method must be public and static
- For unary operator, the operand must be of the same type as the class for which the operator is being defined.
- For binary operator, at least one of the operands, must be of the same type as its class.
- The meaning of the operator will not change.
- You can not overload any C# operator for objects that you have not created.

Operator overloading

- Unary operator

```
 public static TwoDim operator – (TwoDim Tdobj)
{
        return (-Tdobj.x.-Tdobj)
}
```

Operator overloading

- Relational operators such as ==, <, >, can also be overloaded and the process is straightforward.
- These operators return bool.

- Bool type true and false can be overloaded.

public static bool operator true(Param-type operand)

Operator overloading

- Logical operators &, |,! can be overloaded.
- && and || can not be overloaded.

Structs And Enums

Structs

- Struct is a set of diverse types of data that may have different lengths grouped together under a unique declaration.
- Structs are defined using struct keyword.
- Structs are similar to classes, that can contain data members and function members.
- Structs are value types, where as classes are reference types.
- Recall that simple types in c# such as int, float, double… are in fact struct types declared in System namespace.

Structs

- Syntax:

modifiers struct *identifier* {
    struct-members
}

identifier	-	name of the struct
struct-members-		struct members can be constants, fields, methods, properties, events, indexers, operators, constructor, types

Ex:
```
struct student{
        public int id;
        public int marks;    }
```

Structs

- Struct members can be accessed using the same '.' operator
  - Ex:

```
Student s1;
s1. Id = 1;
s1.marks = 100;
```

- ☞ A struct can have another struct as its member.
  - Ex:

```
struct Father{ string name; int age}
struct student{ public Father f;
                int id; int marks;   }
student s1;
s1.f.age=45;
```

Structs

- We can also declared structs inside another struct (Nested structs)
  - Ex:

```
struct student{
          struct Father{ string name; int age}
          int id; int marks;
}

student s1;
s1. Father . Age = 45;

student.Father f1;
f1.age= 45;
```

Structs

- We can have two class variable refer to the same object.
- With structs, two struct variables can not point to the same value.

  -
    ```
    class A{
              public int p; }
    A a1 = new A();
    a1.p = 10;
    A a2 = a1;
    a2.p = 20;
    Console.WriteLine(a1.p);
    ```

    ```
    struct S{
              public int p; }
    S s1 = new S();
    s1.p = 10;
    S s2 = s1;
    s2.p = 20;
    Console.WriteLine(s1.p);
    ```

Class Vs Struct

- Inheritance

  All structs implicitly inherit from the class Object.

  Struct can’t be inherited.

  A struct can implement an interface.

  Abstract and Sealed modifiers are never permitted in a struct declaration.

  Struct function members can not be abstract or virtual.

  Override can be used only with the methods inherited from the Object class.

Class Vs Struct

- Arguments

  Structs can be passed as arguments to methods.

  Structs are always passed by value.

  Structs can be passed by reference using ref or out keywords.

- Constructors

  We can’t declare a parameter less constructor for a struct.

  Each struct will have one parameter less constructor implicitly.

  Structs can’t have destructors.

Class Vs Struct

- In structs, instance variable declaration is not permitted to include initialization.
- But static variables can be initialized.
- Conversions (Boxing and Unboxing)

  Objects of a class can be converted (casted) to another type.

  Struct variables can be converted to only Object type or interface type that is implemented by the struct.

  When a struct is either boxed or unboxed, a new copy of the struct is created.

Enums

- Enum type is a distinct for a group of name numeric constants.

  Syntax: *modifier*$_{opt}$ enum *enum-name :type* $_{opt}${
                enum-members
        }

  Ex: enum month{
                jan, feb, mar, apr, may, jun,
                jul, aug, sep, oct, nov, dec
        }

Enums

- Each of the named constants are given a default value, which starts from 0 and increased by 1 for the succeeding member.
- We can also provide the required members with our own values.
- The members for which value is not supplied will be assigned a value 1 greater than the previous member.
- The underlying type of the member constants in enum is int.

Enums

- A enum can also explicitly declare the underlying type of its member constants.

  Ex:
  ```
  enum month:long{
  jan, feb, mar
  }
  ```
- If the member constants are explicitly assigned, these values must be within the valid range of the underlying type.
- Types that can be used as underlying type are:

  byte, sbyte, short, ushort, int, unit, long and ulong

Enums

- Multiple enum members may share the same value.

Ex:

```
enum {    abc,
          def,
          ghi = def
}
```

Enums

- There are many static useful method available in System.Enum, that can use to query a enum.
- Enum.GetUnderlyingType (typeof(*enum*))

  Returns an instance of Type
- Enum.IsDefined(typeof(*enum*), "*member*")

  Returns a boolean value
- Enum.FromString(typeof(*enum*), "*member*")

  Returns the member value
- Enum.GetValues(typeof(enum))

  Returns an array of underlying type