**1) Anonymous methods**

Anonymous methods provide a technique to pass a code block as a delegate parameter. Anonymous methods are the methods without a name, just the body.

You need not specify the return type in an anonymous method; it is inferred from the return statement inside the method body.

**Writing an Anonymous Method**

Anonymous methods are declared with the creation of the delegate instance, with a delegate keyword.

The following example demonstrates two ways of instantiating a delegate:

1. Associating the delegate with an anonymous method.
2. Associating the delegate with a named method (DoWork).

// Declare a delegate.

delegate void Printer(string s);

class TestClass

{

static void Main()

{

// Instantiate the delegate type using an anonymous method.

Printer p = delegate(string j)

{

System.Console.WriteLine(j);

};

// Results from the anonymous delegate call.

p("The delegate using the anonymous method is called.");

// The delegate instantiation using a named method "DoWork".

p = DoWork;

// Results from the old style delegate call.

p("The delegate using the named method is called.");

}

// The method associated with the named delegate.

static void DoWork(string k)

{

System.Console.WriteLine(k);

}

}

**/\* Output:**

The delegate using the anonymous method is called.

The delegate using the named method is called.

**\*/**

**2) What is Managed or Unmanaged Code?**

**Managed Code**

The code, which is developed in .NET framework is known as managed code. This code is directly executed by CLR with the help of managed code execution. Any language that is written in .NET Framework is managed code.

**Unmanaged Code**

The code, which is developed outside .NET framework is known as unmanaged code. Applications that do not run under the control of the CLR are said to be unmanaged. 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.

**3) What is Boxing and Unboxing?**

**Boxing:** Value to Reference

public void method()

{

int i = 10;

object o = i;

Console.WriteLine(o);

}

**Unboxing:** Reference to Value

public void method()

{

object o = 100;

int i = (int)o;

Console.WriteLine(i);

}

**4) What is the difference between Interface and Abstract Class?**

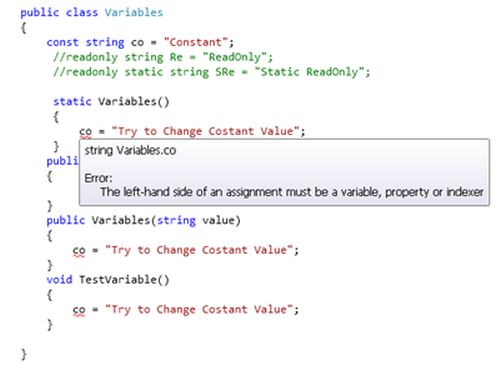
1. A class can implement any number of interfaces but a subclass can at most use only one abstract class.
2. An abstract class can have non-abstract methods (concrete methods) while in case of interface all the methods has to be abstract.
3. An abstract class can declare or use any variables while an interface is not allowed to do so.
4. In an abstract class all data member or functions are private by default while in interface all are public, we can’t change them manually.
5. In an abstract class we need to use abstract keyword to declare abstract methods while in an interface we don’t need to use that.
6. An abstract class can’t be used for multiple inheritance while interface can be used as multiple inheritance.
7. An abstract class use constructor while in an interface we don’t have any type of constructor.

**5) Difference between Const, ReadOnly and ReadOnlyStatic in C#.**

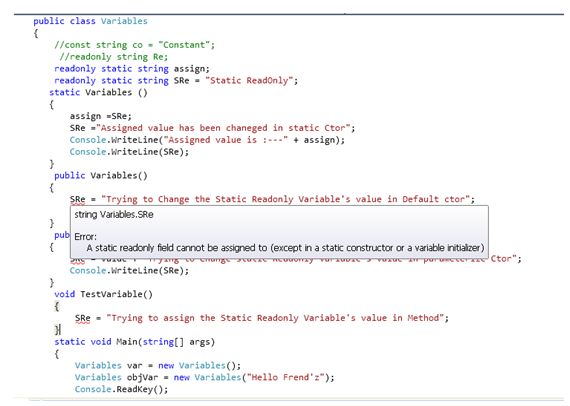
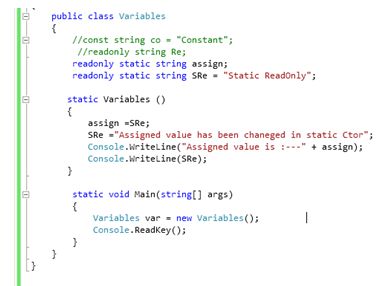
**const:** const is nothing but "constant", a variable of which the value is constant but at compile time. And it's mandatory to assign a value to it. By default, a **const is static and we cannot change the value of a const variable throughout the entire program**.







**ReadOnlyStatic:** A ReadonlyStatic type variable's value can be assigned at runtime or assigned at compile time and changed at runtime. But this variable's **value can only be changed in the static constructor**. And cannot be changed further. It can change only once at runtime. Let's understand it practically.



**readonly:** readonly is the keyword whose value we can **change during runtime or we can assign it at run time but only through the non-static constructor.** Not even a method. Let's see:

publicclasstest1

{

readonly int b = 100;

test1()

{

b = 1000;

}

static test1()

{

//b = 1000; gives error

}

Void change()

{

//b = 200; gives error

}

}

**6) Can “this” be used within a static method?**

**We can't use this in static method because keyword 'this' returns a reference to the current instance of the class containing it**. Static methods (or any static member) do not belong to a particular instance. They exist without creating an instance of the class and call with the name of a class not by instance so we can’t use this keyword in the body of static Methods,

Public class test

{

int a = 10;

static test()

{

a = 400;

//this.a = 400; //gives error

}

Static void show()

{

//this.a = 440; //gives error

a = 400;

}

}

**7) What is the difference between dispose and finalize methods in c#?**

**Finalize:**

* Finalize **used to free unmanaged resources those are not in use** like files, database connections in application domain and more, held by an object before that object is destroyed.
* In the Internal process **it is called by Garbage Collector and can’t called manual by user code or any service.**
* Finalize belongs to **System.Object** class.
* When you use destructor to release resources it implicitly calls Finalize method of the base class. A destructor and finalizer are basically interchangeable concepts.

For example, below is a declaration of a destructor for class Customer:

class Customer

{

~Customer() // destructor

{

// cleanup statements...

}

}

The destructor of Customer class implicitly calls Finalize method of the base class of the object. Therefore, the previous destructor code is implicitly translated to below code

protected override void Finalize()

{

try

{

// Cleanup statements...

}

finally

{

base.Finalize();

}

}

**Dispose:**

* Dispose is also **used to free unmanaged resources those are not in use** like files, database connections in Application domain at any time.
* Dispose explicitly **it is called by manual user code**. If we **need to dispose method** so must **implement that class by IDisposable interface**.
* **It belongs to IDisposable interface**.

public class Customer : IDisposable

{

private StringReader \_reader;

// to detect redundant calls

private bool disposed = false;

public Customer()

{

this.\_reader = new StringReader();

}

protected virtual void Dispose(bool disposing)

{

if (!disposed)

{

if (disposing)

{

if (reader != null) {

this.\_reader.Dispose();

}

}

disposed = true;

}

}

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

}

**8) What is the difference between string and StringBuilder in c#?**

**String:**

* String is a Reference type, because it’s does not have default allocation size.
* String is an **immutable** object. Immutable like when we create string object in code so **we cannot modify or change that object**
* **Performance wise string is slow because its’ create a new instance to override or change the previous value.**
* String belongs to System namespace.

string str = "Raj";

str.Insert(1, "itendra"); // str : “Raj”

var dd = str.Insert(1, "itendra"); // dd : Ritendraaj

**StringBuilder:**

* StringBuilder is also a Reference type.
* StringBuilder is a **mutable** object.
* **Performance wise StringBuilder is very fast because it will use same instance of StringBuilder object to perform any operation like insert value in existing string.**
* StringBuilder belongs to **System.Text.Stringbuilder** namespace.

StringBuilder sb = new StringBuilder("Jeet"); // sb : “Jeet”

sb.Insert(1, "itendra"); // sb : Jitendraeet

**9) Difference b/w ref and out keyword**

The out and ref keywords are useful **when we want to return a value in the same variables which are passed as an argument.**

**ref:**

1. The parameter or argument must be initialized first before it is passed to ref.(sending time)
2. It is not compulsory to initialize a parameter value before using it in a calling method.
3. When we use REF, data can be passed bi-directionally.
4. **public** **static** string GetNextName(ref **int** id)
5. {
6. string returnText = "Next-" + id.ToString();
7. id += 1;
8. **return** returnText;
9. }
10. **static** **void** Main(string[] args)
11. {
12. **int** i = 1;  //must be initialize before pass to ref
13. Console.WriteLine("Previous value of integer i:" + i.ToString());
14. string test = GetNextName(ref i);
15. Console.WriteLine("Current value of integer i:" + i.ToString());
16. }

**out:**

1. It is not compulsory to initialize a parameter or argument before it is passed to an out.
2. A parameter value must be initialized within the calling method before its use.
3. When we use OUT data is passed only in a unidirectional way (from the called method to the caller method).
4. **public** **static** string GetNextNameByOut(out **int** id)
5. {
6. id = 1;
7. string returnText = "Next-" + id.ToString();
8. **return** returnText;
9. }
10. **static** **void** Main(string[] args)
11. {
12. **int** i = 0;  //No need to initialize before pass to out
13. Console.WriteLine("Previous value of integer i:" + i.ToString());
14. string test = GetNextNameByOut(out i);
15. Console.WriteLine("Current value of integer i:" + i.ToString());
16. }

**9) What is delegates in C# and uses of delegates?**

C# delegates are same as pointers to functions, in C or C++. **A delegate Object is a reference type variable that use to holds the reference to a method.** The **reference can be changed at runtime** which is hold by an object of delegate, a delegate object can hold many functions reference.



**10) What is the Difference between Array and ArrayList in C#.Net?**

**Array:**

1. Array is a specific data type storage.
2. No need to do the type casting.
3. Element cannot be inserted or deleted in between.
4. There are no built-in members to do ascending and descending.

**ArrayList:**

1. ArrayList can be stored everything as object.
2. Every time type casting has to do.
3. Elements can be inserted or deleted.
4. ArrayList has many methods to do operation like Sort, Insert, Remove, Binary Search etc.

**11) Exception Filters(.NET Framework 4.6 and C# 6.0 and VS 2013/2015)**

Basically, Exception Filter is one of the new features of C# v6.0 that allows us to specify a conditional clause for each catch block. In other words now we can write a catch block that will handle the exception of a specific type only when a certain condition is true that is written in an exception filter clause.

try

{

throw new Exception("ErrorType2");

}

catch(Exception ex) when (ex.Message == "ErrorType1")

{

Console.WriteLine("Error Message : " + ex);

}

catch(Exception ex) when (ex.Message == "ErrorType2")

{

Console.WriteLine("Error Message : " + ex);

}

**Note:***In the preceding code snippet we saw that instead of handling all the exceptions of a specific type, we handled the specific exception. In this code the try block is throwing the exception ErrorType2. We suppose that it's base type is IndexOutOfRangeException. So it will be caught by both the catch block that is getting the IndexOutOfRangeException but it will execute only by the first catch block that met the specified condition into the Exception Filter. Here ErrorType1 matches the first exception, in other words if(ex.Message=="ErrorType1").*

**12) Null Propagation Operator (?.) (.NET Framework 4.6 and C# 6.0 and VS 2013/2015)**

**?? : null-coalescing operator**

**??= : null-coalescing assignment operator**

C# 6.0 introduced the null propagation operator for checking for null values in code. The null propagation operator is denoted by "?". It has the property that **if we make a reference variable in our code and want to check its value for null or not null before invoking the object so using the null propagation operator we can check it and remove the null exception error**. It also increases the reliability of code and reduces the lines of code. That makes the code cleaner and easier.

*class Program*

{

static void Main(string[] args)

{

var a = new first

{

number = 1234

};

Console.WriteLine(a.age == null ?10001 : a.age.no1);//old method

Console.WriteLine(a.age?.no1 ?? 10001); //new code

Console.Read();

}

}

*class first*

{

public int number { get; set; }

public second age { get; set; }

}

*class second*

{

public int no1 { get; set; }

public int no2 { get; set; }

}

**13) String interpolation (.NET Framework 4.6 and C# 6.0 and VS 2013/2015)**

String interpolation provides a more readable and convenient syntax to create formatted strings than a string composite formatting feature. The following example uses both features to produce the same output.

string name = "Mark";

var date = DateTime.Now;

***// Composite formatting:***

Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date);

Console.WriteLine(String.Format("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date));

***// String interpolation:***

Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");

// Both calls produce the same output that is similar to:

// Hello, Mark! Today is Wednesday, it's 19:40 now.

**14) Dictionary initializer (.NET Framework 4.6 and C# 6.0 and VS 2013/2015)**

We can now directly create a key and assign a value to this key. Hence, as per the new technique, we can also have the code given below.

Dictionary<int, string>dic = new Dictionary<int, string>

{

//New // Old

[1] = "User A", //{ 1, "User A" },

[2] = "User B", //{ 1, "User B" },

[3] = "User C", //{ 1, "User C" },

};

foreach (var item in dic)

{

Console.WriteLine($"Key is {item.Key} and Value is: {item.Value}");

}

**15) Caller info attributes (.NET Framework 4.5 and C# 5.0 and VS 2012/2013)**

Caller information gives information about the caller of any function. This concept was introduced in C# 5.0. In C# 5.0 the caller info feature has introduced and using this feature we can easily get information about caller function. It uses *System.Runtime.CompilerServices;*

These three types of attributes that are used in tracking information.

1. **CallerFilePath**: Sets the information about caller's source code file.
2. **CallerLineNumber**: Sets the information about caller's line number.
3. **CallerMemberName**: Sets the information about caller member name.

class Program

{

public void Call(string text, [CallerFilePath] string File\_name = "",

[CallerLineNumber] int Line = 0,

[CallerMemberName] string member\_name = "")

{

Console.WriteLine(text); //**This is Caller Method**

Console.WriteLine(File\_name); //**Complete file path**

Console.WriteLine(Line); //Line number of **Obj.Call("This is Caller Method");**

Console.WriteLine(member\_name); //**Main**

}

static void Main(string[] args)

{

Program Obj = new Program();

Obj.Call("This is Caller Method");

Console.ReadLine();

}

}

**16) Asynchronous programming (.NET Framework 4.5 and C# 5.0 and VS 2012/2013)**

Asynchrony is essential for activities that are potentially blocking, such as web access. Access to a web resource sometimes is slow or delayed. If such an activity is blocked in a synchronous process, the entire application must wait. In an asynchronous process, the application can continue with other work that doesn't depend on the web resource until the potentially blocking task finishes.

***Async improves responsiveness***

*The async and await keywords in C# are the heart of async programming. Asynchronous methods that you define by using the async keyword are referred to as async methods.*

The following characteristics summarize what makes the previous example an async method.

* The method signature includes an async modifier.
* The name of an async method, by convention, ends with an "Async" suffix.
* The return type is one of the following types:

1. **Task<TResult>** if your method has a return statement in which the operand has type TResult.
2. **Task** if your method has no return statement or has a return statement with no operand.
3. **void** if you're writing an async event handler.

**Threads**

Async methods are intended to be non-blocking operations. An await expression in an async method doesn’t block the current thread while the awaited task is running. Instead, the expression signs up the rest of the method as a continuation and returns control to the caller of the async method.

The async and await keywords don't cause additional threads to be created. **Async methods don't require multithreading** because **an async method doesn't run on its own thread.** The method runs on the current synchronization context and uses time on the thread only when the method is active. You can use **Task.Run** to move CPU-bound work to a background thread, but a background thread doesn't help with a process that's just waiting for results to become available.

*An async method typically contains one or more occurrences of an await operator*, *but the absence of await expressions doesn’t cause a compiler error. If an async method doesn’t use an await operator to mark a suspension point, the method executes as a synchronous method does, despite the async modifier. The compiler issues a warning for such methods.*

*The 'await' operator can only be used within an async method.*

**a**) ­//definition

Task<int>Test()

{

return Task.Run(() =>

{

return 5;

});

}

//calling :obj.Test();

**b)** //definition

async Task<int>sum(int a, int b)

{

return a + b;

}

//calling :Task<int> c = obj.sum(3, 3);

**c)** //definition

async void show(string name)

{

Console.WriteLine(name);

}

//calling :obj.show("rahul");

**d)**//definition

async Task disp()

{

Task<int>tt = Test();

await tt;

await Task.Run(() =>

{

Console.WriteLine("Bye");

});

}

**e) //definition**

Task Num()

{

return Task.Run(() =>

{

Console.WriteLine("Bye");

});

}

//calling :obj.Num();

**17) Dynamic binding (.NET Framework 4.0 and C# 4.0 and VS 2010)**

1. The purpose of using dynamic binding is to avoid compile time checking of the code.
2. dynamic is a keyword that is used to make a property or a method dynamic.
3. When we make dynamic type, compiler does not check it at compile-time. Compiler checks it only at run time.
4. property created using dynamic keyword works like object. Dynamic variables are compiled into type object variables and exist only at compile time, not at run time.

The type of dynamic and object both are similar. We can check it by using the following code.

public static void Main(string[] args)

{

dynamic v = 1;

object v1 = 1;

Console.WriteLine(v.GetType());

Console.WriteLine(v1.GetType());

}

**Output:**

System.Int32

System.Int32

Now, let's manipulate the both objects and see the working difference.

public static void Main(string[] args)

{

dynamic v = 1;

object v1 = 1;

// Modifying Objects

v = v + 3;

v1 = v1 + 5; //Operator '+' cannot be applied to operands of type 'object' and 'int'

Console.WriteLine(v);

Console.WriteLine(v1);

}

Above, it produces a compile-time error due to object v1. The dynamic object does not check at compile time so, it does not produce any error at compiler-time.

**Dynamic Properties and Methods**

public class Student

{

// Creating dynamic property

public dynamic Name { get; set; }

// Creating a dynamic method

public dynamic ShowMSG(string msg)

{

return msg;

}

}

public class DynamicExample

{

public static void Main(string[] args)

{

Student student = new Student();

student.Name = "Peter";

Console.WriteLine(student.Name);

// Storing result in dynamic object

dynamic msg = student.ShowMSG("Welcome to the javapoint");

Console.WriteLine(msg);

}

}

**Output:**

Peter

Welcome to the javapoint

**18) Named and Optional Arguments (.NET Framework 4.0 and C# 4.0 and VS 2010)**

**C# Named Arguments:**

Generally, when we make named arguments, the arguments are evaluated in the order in which they are passed.

This *Named* feature allows us to associate argument name with its value at the time of function calling.

It helps us, not to remember the order of parameters. If we know the parameters names, we can pass that in any order. Let's see an example.

static string GetFullName(string firstName, string lastName)

{

return firstName + " " + lastName;

}

public static void Main(string[] args)

{

string fullName1 = GetFullName("Rahul", "Kumar"); // Without named arguments

string fullName2 = GetFullName(firstName:"Rahul", lastName:"Kumar"); // Named arguments

string fullName3 = GetFullName(lastName:"Rahul", firstName:"Kumar"); // Changing order

Console.WriteLine(fullName1);

Console.WriteLine(fullName2);

Console.WriteLine(fullName3);

}

**Output:**

Rahul Kumar

Rahul Kumar

Kumar Rahul

**C# Optional Arguments:**

1. In C#, a method may contain required or optional parameters. A method that contains optional parameters does not force to pass arguments at calling time.
2. It means we call method without passing the arguments.
3. The optional parameter contains a default value in function definition. If we do not pass optional argument value at calling time, the default value is used.
4. Make sure, ***all the optional parameters are placed at the end of the parameter list***. Otherwise, compiler throws a compile-time error.

public static void Main(string[] args)

{

add(12,12); // Passing both arguments

add(10); // Passing only required argument

sum();

}

static void add(int a, int b = 10) // second parameter is optional

{

Console.WriteLine(a+b);

}

static void sum(int a=10,int b=20)

{

Console.WriteLine(a+b);

}

static void sub(int a=10, int b) //Error:Optional parameters must appear after all required parameters

{

Console.WriteLine(a+b);

}

**19) (.NET Framework 3.0/3.5 and C# 3.0 and VS 2008)**

1. ***Auto-implemented properties*** //public int age { get; set; }
2. ***Anonymous types*** // var name = “Rahul”
3. ***Lambda expression :*** A lambda expression with an expression on the right side of the => operator is called an expression lambda. **Syntax:**

**(input-parameters) => expression**

eg. (x, y) => x == y

(int x, string s) =>s.Length> x

delegate bool D2(int i, int j);

void largest()

{

D2 del2 = (x, y) =>{ return x > y; };

Console.WriteLine(del2(5, 10));

}

1. ***Query expressions*:** Query Expression is an expression that is **written by using LINQ query syntax**. The LINQ (Language Integrated Query) is a language that is used to construct a query. C# Query Expression contains set of clauses and use query expression similar to SQL.

static void Main()

{

// Data source.

int[] scores = { 90, 71, 82, 93, 75, 82 };

// Query Expression.

IEnumerable<int> scoreQuery = //query variable

from score in scores //required

where score > 80 // optional

order by score descending // optional

select score; //must end with select or group

// Execute the query to produce the results

foreach (int testScore in scoreQuery)

{

Console.WriteLine(testScore);

}

}

// **Outputs:** 93 90 82 82

Var empData =

from emp in entity.Employees//query variable

let sal = emp.Salary //range variable

group emp by sal into empGroup

where empGroup.Key> 35000

order by empGroup.Key

select empGroup;

Order (FLGWOS) : from, let, group, where, order, select.

1. **Extension methods:** 
   * Extension methods enable you to add methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.
   * An extension method is a special kind of **static method**, but they are called as if they were instance methods on the extended type.
   * An extension method is a static method of a static class, where the "this" modifier is applied to the first parameter. The type of the first parameter will be the type that is extended.
   * Extension methods are only in scope when you explicitly import the namespace into your source code with a using directive.

**Example1 :**

using System;

using System.Text;

namespace ClassLibExtMethod

{

public class Class1

{

public string Display()

{

return ("I m in Display");

}

public string Print()

{

return ("I m in Print");

}

}

}

**\*\*Include the above Class Library .dll in new Console Application**

using System;

using System.Text;

using ClassLibExtMethod;

namespace ExtensionMethod1

{

public static class XX

{

public static void NewMethod(this Class1 ob)

{

Console.WriteLine("Hello I m extended method");

}

}

class Program

{

static void Main(string[] args)

{

Class1 ob = new Class1();

ob.Display();

ob.Print();

ob.NewMethod();

Console.ReadKey();

}

}

}

**Example2 :**

public static class IntExtensions

{

//Extnsion Method

public static bool IsGreaterThan(this int i, int value)

{

return i> value;

}

public static int IntegerExtension(this string str)

{

return Int32.Parse(str);

}

}

static void Main(string[] args)

{

int I =10;

bool result = i.IsGreaterThan(100);

Console.WriteLine("Result: {0}", result);

string str = "123456";

int num = str.IntegerExtension();

Console.WriteLine("The output using extension method: {0}", num);

}

**20) (.NET Framework 2.0 and C# 2.0 and VS 2005)**

1. **Anonymous methods:** As the name suggests, an anonymous method is a method without a name. Anonymous methods in C# can be defined using the delegate keyword and can be assigned to a variable of delegate type. **For eg.**

public delegate void Print(int value);

static void Main(string[] args)

{

Print print = delegate(int val) {

Console.WriteLine("Inside Anonymous method. Value: {0}", val);

};

print(100);

}

1. **Partial Types:**

**Partial Class**

When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.

*public partial class CoOrds*

{

private int x;

private int y;

public CoOrds(int x, int y)

{

this.x = x;

this.y = y;

}

}

*public partial class CoOrds*

{

public void PrintCoOrds()

{

Console.WriteLine("CoOrds: {0},{1}", x, y);

}

}

*class TestCoOrds*

{

static void Main()

{

CoOrds myCoOrds = new CoOrds(10, 15);

myCoOrds.PrintCoOrds();

// Keep the console window open in debug mode.

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

}

**Partial Method**

A partial method has its signature defined in one part of a partial type, and its implementation defined in another part of the type. Partial methods enable class designers to provide method hooks, similar to event handlers, that developers may decide to implement or not. If the developer does not supply an implementation, the compiler removes the signature at compile time. The following conditions apply to partial methods:

* **Signatures in both parts of the partial type must match.**
* **If method return type is void no access modifier is required.**
* **Return type is required in non-void method.**

The following example shows a partial method defined in two parts of a partial class:

namespace PM

{

partial class A

{

partial void OnSomethingHappened(string s);

public partial int Add();

}

// This part can be in a separate file.

partial class A

{

// Comment out this method and the program

// will still compile.

public partial void OnSomethingHappened(String s)

{

Console.WriteLine("Something happened: {0}", s);

}

public partial int Add()

{

return 0;

}

}

}

**The following example shows that you can also develop partial structs and interfaces.**

partial interface ITest

{

void Interface\_Test();

}

partial interface ITest

{

void Interface\_Test2();

}

partial struct S1

{

void Struct\_Test() { }

}

partial struct S1

{

void Struct\_Test2() { }

}

The partial keyword indicates that other parts of the class, struct, or interface can be defined in the namespace. All the parts must use the partial keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as public, private, and so on.

*If any part is declared abstract, then the whole type is considered abstract. If any part is declared sealed, then the whole type is considered sealed. If any part declares a base type, then the whole type inherits that class.*

*The partial modifier is not available on delegate or enumeration declarations.*

1. **Generics**

A generic class can be defined using angle brackets <>. Generics allow you to define a class with placeholders for the type of its fields, methods, parameters, etc. Generics replace these placeholders with some specific type at compile time.

class Program

{

public delegate T add<T>(T param1, T param2);

static void Main(string[] args)

{

add<int> sum = AddNumber;

Console.WriteLine(sum(10, 20));

add<string>conct = Concate;

Console.WriteLine(conct("Hello","World!!"));

}

public static int AddNumber(int val1, int val2)

{

return val1 + val2;

}

public static string Concate(string str1, string str2)

{

return str1 + str2;

}

}

1. **Nullable types**

Nullable<int>i = null; OR int? i = null;

int j = i ?? 0;

Console.WriteLine(j); //0 is the output

1. **Iterators**

foreach worked with collection and arrays

**21) Difference between for and foreach loop**

1. The **for loop** executes a statement or a block of statements repeatedly until a specified expression evaluates to false **while** the **foreach** statement repeats a group of embedded statements for each element in an array or an object collection. You do not need to specify the loop bounds minimum or maximum.
2. **foreach** treats everything as a collection and **reduces the performance**. *foreach creates an instance of an enumerator (returned from GetEnumerator()) and that enumerator also keeps state throughout the course of the foreach loop. It then repeatedly calls for the Next() object on the enumerator and runs your code for each object it returns.*
3. Using **for loop** we **iterate** the array in **both direction**, that is **from index 0 to 9 and from 9 to 0 while** using **foreach loop**, the iteration is possible **in forward direction only**.
4. For loop is faster than foreach loop.
5. **for** (**int** i = 1; i <= 5; i++)
6. {
7. i = i + i;
8. }
9. **int**[] tempArr = **new** **int**[] { 0, 1, 2, 3, 5, 8, 13 };
10. **foreach** (**int** i **in** tempArr)
11. {
12. i = i + 1;  //Cannot assign to 'i' because it is a 'foreach iteration variable'
13. }

**21) HashSet<T>**

C# HashSet class can be used to store, remove or view elements. It does not store duplicate elements. **It is suggested to use HashSet class if you have to store only unique elements**. It is found in **System.Collections.Generic** namespace.

public static void Main(string[] args)

{

// Create a set of strings

var names = new HashSet<string>();

names.Add("Sonoo");

names.Add("Ankit");

names.Add("Peter");

names.Add("Irfan");

names.Add("Ankit");//will not be added

// Iterate HashSet elements using foreach loop

foreach (var name in names)

{

Console.WriteLine(name);

}

}

**22) Differences between Task and Thread**

1. The Thread class is used for creating and manipulating a thread in Windows. A Task represents some asynchronous operation and is part of the Task Parallel Library, a set of APIs for running tasks asynchronously and in parallel.
2. The task can return a result. There is no direct mechanism to return the result from a thread.
3. Task supports cancellation through the use of cancellation tokens. But Thread doesn't.
4. A Task can have multiple processes happening at the same time. Threads can only have one task running at a time.
5. We can easily implement Asynchronous using ’async’ and ‘await’ keywords.
6. A new Thread**() is not dealing with Thread pool** thread, **whereas Task does use thread pool** thread.
7. A Task is a higher level concept than Thread.

**Thread**

When we execute things on multiple threads, it’s not guaranteed that the threads are separated across multiple processors.

Thread represents an actual OS-level thread, with its own stack and kernel resources.

The problem with Thread is that OS threads are costly. Each thread you have consumes a non-trivial amount of memory for its stack, and adds additional CPU overhead as the processor context-switch between threads. Instead, it is better to have a small pool of threads execute your code as work becomes available.

**ThreadPool**

ThreadPool is a wrapper around a pool of threads maintained by the CLR. ThreadPool gives you no control at all; you can submit work to execute at some point, and you can control the size of the pool, but you can't set anything else. You can't even tell when the pool will start running the work you submit to it.

**Task**

Task is a lightweight object for managing a parallelizable unit of work. It can be used whenever you want to execute something in parallel. Parallel means the work is spread across multiple processors to maximize computational speed.

**23) What is Microsoft .Net?**

.Net is a programming framework developed by Microsoft, which can be used to build different types of applications like – Console, Windows, Web application and Mobile based applications. It provides controlled environment with built-in tools for developing, installing and executing different types of applications.

The main two components of .Net Framework are Common Language Runtime (CLR) and .Net Framework Class Library (FCL).

**24) Why we choose .Net?**

Developing applications using .Net framework is very robust and highly secure with great quality. .Net platform reduces development time, creates quality, reliable, and scalable applications that ensure smooth functioning of complex business applications. Hence it helps customer to improve their business easily.

**The .NET Framework is designed (purpose) to fulfil the following objectives:**

1. To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
2. To provide a code-execution environment that **minimizes software deployment and versioning conflicts.**
3. To provide a code-execution environment that **promotes safe execution of code,** including code created by an unknown or semi-trusted third party.
4. To provide a code-execution environment that **eliminates the performance problems** of scripted or interpreted environments.
5. To make the developer experience consistent across **widely varying types of apps, such as Windows-based apps and Web-based apps.**
6. To build all communication on industry standards to ensure that code based on the .NET Framework integrates with any other code.

C# Latest Version: 7.1

**C# is an object-oriented programming language**, meaning it can increase productivity in the development process. **C# boasts type-safety, garbage collection, simplified type declarations, versioning and scalability support, and other features that make developing solutions faster and easier.**

**ASP.NET Core (.NET Framework: .NET Core and C# : 7.0)**

**Latest Version: 2.1.5**

.NET Core is a cross-platform version of .NET for building websites, services, and console apps.

1. This framework is a complete rewrite that combines the previously separate ASP.NET MVC and ASP.NET Web API into a single programming model.
2. ASP.NET Core applications supports side by side versioning in which different applications, running on the same machine, can target different versions of ASP.NET Core. This is not possible with previous versions of ASP.NET.

**Components of .Net Core**

1. Entity Framework (EF) Core
2. Identity Core
3. MVC Core
4. Razor Core

**Features of .Net Core**

1. **No-compile developer experience** (i.e. compilation is continuous, so that the developer does not have to invoke the compilation command)
2. Modular framework distributed as NuGet packages
3. Cloud-optimized runtime (optimised for the internet)
4. **Side-by-side app versioning when targeting .NET Core.**
5. **A** **light-weight and modular HTTP request pipeline.**
6. **Build and run cross-platform ASP.NET Core apps on Windows, Mac, and Linux**
7. Open-source and community-focused

**Why ASP.NET Core?**

**Supports Multiple Platforms:** ASP.NET Core applications can run on Windows, Linux, and Mac. So you don't need to build different apps for different platforms using different frameworks.

**Fast:** ASP.NET Core no longer depends on System.Web.dll for browser-server communication. ASP.NET Core allows us to include packages which we need for our application. This reduces the request pipeline and improves the performance and scalability.

**IoC Container:** It includes built-in IoC container for automatic dependency injection which makes it maintainable and testable.

**Integration with Modern UI Frameworks:** It allows you to use and manage modern UI frameworks such as AngularJS, ReactJS, Umber, Bootstrap etc. using Bower (a package manager for the web).

**Hosting:** ASP.NET Core web application can be hosted on multiple platforms with any web server such as IIS, Apache etc. It is not dependent only on IIS as a standard .NET Framework.

**Code Sharing:** It allows you to build a class library which can be used with other .NET frameworks such as .NET Framework 4.x or Mono. Thus a single code based can be shared across frameworks.

**Side-by-Side App Versioning:** ASP.NET Core runs on .NET Core which supports simultaneous running of multiple versions of applications.

**Smaller Deployment Footprint:** ASP.NET Core application runs on .NET Core which is smaller than full .NET Framework. So, the application which uses only a part of .NET CoreFX will have smaller deployment size. This reduces the deployment foot print.

**Difference between C# and VB**

1. **VB is NOT Case Sensitive where C# is**. Allows you to create modular maintainable applications and reusable codes.
2. **VB is NOT pure object oriented where C# is**.
3. **Automatic Garbage Collection** erase and remove all the garbage present on the system

**Benefits of C#**

1. Object- Oriented Language
2. Automatic Garbage Collection
3. Cross Platform
4. Better Integration with other language(CLR compatible)
5. Easy-to-Development

**What is .Net?**

It is Framework

**What is .Net Framework? (Latest Versin: 4.7.1) .Net Core**

It is a software development framework developed by Microsoft. Using it we can create:

1. Console Application
2. Windows Application
3. Web Application (ASP.NET, MVC, Sliverlight)
4. Mobile Application
5. Services like Web Services, WCF Services, Web API
6. Customized Libraries

**.Net Framework Design Principle**

The following design principles of the .Net framework is what makes it very relevant to create .Net based applications.

1. **Interoperability -** The .Net framework provides a lot of backward support. Suppose if you had an application built on an older version of the .Net framework, say 2.0. And if you tried to run the same application on a machine which had the higher version of the .Net framework, say 3.5. The application would still work. This is because with every release, Microsoft ensures that older framework versions gel well with the latest version.
2. **Portability-** Applications built on the .Net framework can be made to work on any Windows platform. And now in recent times, Microsoft is also envisioning to make Microsoft products work on other platforms, such as iOS and Linux.
3. **Security -** The .NET Framework has a good security mechanism. The inbuilt security mechanism helps in both validation and verification of applications. Every application can explicitly define their security mechanism. Each security mechanism is used to grant the user access to the code or to the running program.
4. **Memory management -** The Common Language runtime does all the work or memory management. The .Net framework has all the capability to see those resources, which are not used by a running program. It would then release those resources accordingly. This is done via a program called the **"Garbage Collector"** which runs as part of the .Net framework. The garbage collector runs at regular intervals and keeps on checking which system resources are not utilized, and frees them accordingly.
5. **Simplified deployment -** The .Net framework also have tools, which can be used to package applications built on the .Net framework. These packages can then be distributed to client machines. The packages would then automatically install the application.

Protected Internal : Accessable only to child class with in the same application

Internal : Accessable to any class or method in current assembly(same application)

Protected : Accessable only to child class

Private : Accessable only in the same class.

Public : Accessable to entire application.

**Collections:** Consists of set of classes that provides array functionality in a superior manner.

There are the following 2 types of collections:

1. Non-Generic
2. Generic

**Non-generic** **Generic**

ArrayList -------------> List

HashTable -------------> Dictionary

SortedList -------------> SortedList (Arranges all the items in sorted order.)

Stack -------------> Stack

Queue -------------> Queue

**1. Non-Generic**

1. Each element can represent a value of a different type.
2. Array Size is not fixed.
3. Elements can be added / removed at runtime.

**2. Generic Collections**

Generic Collections work on the specific type that is specified in the program whereas non-generic collections work on the object type.

1. Specific type
2. Array Size is not fixed
3. Elements can be added / removed at runtime.

**Generics:** It *allow you to delay the specification of the data type* of programming elements in a class or a method, until it is actually used in the program. In other words, **generics allow you to write a class or method that can work with any data type.**

**Features**

It helps you to maximize

1. Code reuse
2. Type safety
3. Performance.
4. You can create your own generic interfaces, classes, methods, events, and delegates.

**Example**

using System;

using System.Collections.Generic;

namespace GenericApplication {

public class MyGenericArray<T> { //Generic Type Class

private T[] array; //Generic Type Method

public MyGenericArray(int size) {

array = new T[size + 1];

}

public T getItem(int index) {

return array[index];

}

public void setItem(int index, T value) {

array[index] = value;

}

}

class Tester {

static void Main(string[] args) {

//declaring an int array

MyGenericArray<int> intArray = new MyGenericArray<int>(5);

//setting values

for (int c = 0; c < 5; c++) {

intArray.setItem(c, c\*5);

}

//retrieving the values

for (int c = 0; c < 5; c++) {

Console.Write(intArray.getItem(c) + " ");

}

Console.WriteLine();

//declaring a character array

MyGenericArray<char> charArray = new MyGenericArray<char>(5);

//setting values

for (int c = 0; c < 5; c++) {

charArray.setItem(c, (char)(c+97));

}

//retrieving the values

for (int c = 0; c< 5; c++) {

Console.Write(charArray.getItem(c) + " ");

}

Console.WriteLine();

Console.ReadKey();

}

}

}

**What is the use of yield keyword:**

yield keyword help us :

* + When you want to do Custom iteration without temp collection.
  + Stateful iteration.

Consider a list as given below :

static void FillValues()

{

myList.Add(1);

myList.Add(2);

myList.Add(3);

myList.Add(4);

myList.Add(5);

}

1. **Custom Iteration Example**

static IEnumerable<int> Filter()

{

//List<int> temp = new List<int>();

foreach (var i in myList)

{

if (i > 3)

//temp.Add(i);

yield return i;

}

//return temp;

}

static void Main(string[] args)

{

FillValues();

foreach (var i in Filter())

{

Console.WriteLine(i);

}

Console.ReadLine();

}

1. **Stateful Iteration Example**

static IEnumerable<int> RunningTotal()

{

int runningTotal = 0;

foreach (var i in myList)

{

runningTotal += i;

yield return runningTotal;

}

}

static void Main(string[] args)

{

FillValues();

foreach (var i in RunningTotal())

{

Console.WriteLine(i);

}

Console.ReadLine();

}

**Reverse each word in a string**

string inputString = "one two three four five";

string resultString = string.Join(" ", inputString

.Split(' ')

.Select(x => new String(x.Reverse().ToArray())));

**Func, Action and Predicate**

* **Func is a delegate** that points to a method that accepts one or more arguments and **returns a value.**
* **Action is a delegate** that points to a method which in turn accepts one or more arguments but **returns no value**. In other words, you should use Action when your delegate points to a method that returns void.
* A **Predicate is a delegate** that accepts one or more generic parameters and **returns a Boolean value** — you can assume it is something like Func<T,bool>. **Predicate delegates are typically used to perform search operations on some data based on a set of criteria**.

**e.g. for Ation Delegate**

static void Main(string[] args)

{

Action<int, int> sum = new Action<int, int>(Add);

sum(10, 20);

}

public static void Add(int a, int b)

{

Console.WriteLine(a + b);

}

**e.g. for Func Delegate**

static void Main(string[] args)

{

Func<int, int, int> sum = new Func<int, int, int>(Add);

Console.WriteLine(sum(10, 20));

}

public static int Add(int a, int b)

{

return a + b;

}

**e.g. for Predicate Delegate**

static void Main(string[] args)

{

List<Customer> custList = new List<Customer>();

custList.Add(new Customer { Id = 1, FirstName = "Joydip", LastName = "Kanjilal);

custList.Add(new Customer { Id = 2, FirstName = "Steve", LastName = "Jones);

Predicate<Customer> hydCustomers = x => x.Id == 1;

Customer customer = custList.Find(hydCustomers);

Console.WriteLine(customer.FirstName);

}

**What is Reflection in c#?**

Reflection in C# is used to retrieve metadata on types at runtime. In other words, you can use reflection to inspect metadata of the types in your program dynamically -- you can retrieve information on the loaded assemblies and the types defined in them.

To work with reflection in .Net, you should include the System.Reflection namespace in your program. In using reflection, you get objects of the type "Type" that can be used to represent assemblies, types, or modules. You can use reflection to create an instance of a type dynamically and even invoke methods of the type.

static void Main(string[] args)

{

Type type = typeof(Customer);

PropertyInfo[] propertyInfo = type.GetProperties();

Console.WriteLine("The list of properties of the Customer class are:--");

foreach (PropertyInfo pInfo in propertyInfo)

{

Console.WriteLine(pInfo.Name);

}

}

**Covariance and Contravariance (C#)**

In C#, covariance and contravariance enable implicit reference conversion for array types, delegate types, and generic type arguments. Covariance preserves assignment compatibility and contravariance reverses it.

// Assignment compatibility.

string str = "test";

// An object(str) of a more derived type(string) is assigned to an object(obj) of a less derived type(obj).

object obj = str;

// Covariance.

IEnumerable<string> strings = new List<string>();

// An object that is instantiated with a more derived type argument

// is assigned to an object instantiated with a less derived type argument.

// Assignment compatibility is preserved.

IEnumerable<object> objects = strings;

// Contravariance.

// Assume that the following method is in the class:

// static void SetObject(object o) { }

Action<object> actObject = SetObject;

// An object that is instantiated with a less derived type argument

// is assigned to an object instantiated with a more derived type argument.

// Assignment compatibility is reversed.

Action<string> actString = actObject;

Covariance for arrays enables implicit conversion of an array of a more derived type to an array of a less derived type. But this operation is not type safe, as shown in the following code example.

object[] array = new String[10];

// The following statement produces a run-time exception.

// array[0] = 10;

**Covariance in delegates allows flexibility in the return type of delegate methods.**

Consider the following architecture:

public class Small

{

}

public class Big : Small

{

}

public class Bigger : Big

{

}

public delegate Small covarDel(Big mc);

public class Program

{

public static Big Method1(Big bg)

{

Console.WriteLine("Method1");

return new Big();

}

public static Small Method2(Big bg)

{

Console.WriteLine("Method2");

return new Small();

}

public static void Main(string[] args)

{

covarDel del = Method1;

Small sm1 = del(new Big());

del = Method2;

Small sm2 = del(new Big());

}

}

**C# Contravariance**

**Contravariane is applied to parameters**. Cotravariance allows a method with the parameter of a base class to be assigned to a delegate that expects the parameter of a derived class.

You can also use covariance and contravariance in the same method as shown below.

delegate Small covarDel(Big mc);

class Program

{

static Big Method4(Small sml)

{

Console.WriteLine("Method3");

return new Big();

}

static void Main(string[] args)

{

covarDel del = Method4;

//passing new Big() in place of new Small() is Contravariance and return type of del(new Big()) which is of type Big is assigned to it's base type is Covariance

Small sm = del(new Big());

}

}

**is in c#**

* The "is" operator is used to check whether the run-time type of an object is compatible with a given type or not.
* It used to **check the object type and it returns a bool value**: true if the object is the same type and false if not.

class Class1

{

}

class Class2

{

}

public class IsTest

{

public static void Test(object o)

{

Class1 a;

Class2 b;

if (o is Class1)

{

Console.WriteLine("o is Class1");

a = (Class1)o;

}

else if (o is Class2)

{

Console.WriteLine("o is Class2");

b = (Class2)o;

}

else

{

Console.WriteLine("o is neither Class1 nor Class2.");

}

}

public static void Main()

{

Class1 c1 = new Class1();

Class2 c2 = new Class2();

Test(c1);

Test(c2);

Test("Passing String Value instead of class");

Console.ReadKey();

}

}

**Output:**

o is Class1

o is Class2

o is neither Class1 nor Class2.

**as in c#**

* The ‘as’ operator does the same job of ‘is’ operator but the difference is instead of bool, it returns the object if they are compatible to that type, else it returns null.
* In other words, the ‘as’ operator is used to perform conversions between compatible types.

Using the same example of is operator

public static void Main()

{

object[] myObjects = new object[6];

myObjects[0] = new Class1();

myObjects[1] = new Class2();

myObjects[2] = "string";

myObjects[3] = 32;

myObjects[4] = null;

for (int i = 0; i < myObjects.Length; ++i)

{

string s = myObjects[i] as string;

Console.Write("{0}:", i);

if (s != null)

Console.WriteLine("'" + s + "'");

else

Console.WriteLine("not a string");

}

Console.ReadKey();

}

**Output:**

0:not a string

1:not a string

2:'string'

3:not a string

4:not a string

5:not a string

**Advantages of as over is:**

In case of ‘**is’** operator, to type cast, we need to do two steps:

1. Check the Type using is
2. If it’s true then Type cast

Actually this affects the performance since each and every time the CLR will go through the inheritance hierarchy, checking each base type against the specified type.

To avoid this, use ‘**as’**, it will do it in one step. Only for checking the type should we use the ‘**is’** operator.

**Task.WaitAll(task1,task2)**

The Task.WaitAll blocks the current thread until all other tasks have completed execution.

**Task.WhenAll(task1,task2)**

The Task.WhenAll method is used to create a task that will complete if and only if all the other tasks have completed.

WhenAll returns a Task. When you call this method it returns a task that is waiting for all the tasks to complete. To actually wait for the other tasks to complete you would wait on the Task that was returned. WaitAll could be implemented using WhenAll like this:

**Task.WhenAll(task1, task2).Wait();**

**Task.WhenAny(task1,task2)**

We use Task.WhenAny when we have a collection of tasks, but we only interested in the first finished task. It can happen for example when we have couple of async API that all of them do the same thing. But we want to receive the result from the one that return the result first.

**Task.WaitAny(task1,task2)**

The main difference between these two is that Task.WaitAny is a blocking operation.

Basically **Wait()** and **Sleep()** are both thread blocking operations, in other words they force a thread to sit idle instead of doing work elsewhere. **Delay()** on other hand uses a timer internally that releases the thread in use until the delay is complete