1. What is C#?

C# is a computer programming language. C# was developed by Microsoft in 2000 to provide a modern general-purpose programming language that can be used to develop all kinds of software targeting various platforms including Windows, Web, and Mobile using just one programming language. Today, C# is one of the most popular programming languages in the world. Millions of software developers use C# to build all kinds of software.

C# is the primary language for building Microsoft .NET software applications. Developers can build almost every kind of software using C# including Windows UI apps, console apps, backend services, cloud APIs, Web services, controls and libraries, serverless applications, Web applications, native iOS and Android apps, AI and machine learning software, and blockchain applications.

C# with the help of Visual Studio IDE provides a rapid application development. C# is a modern, object-oriented, simple, versatile, and performance-oriented programming language. C# is developed based on the best features and use cases of several programming languages including C++, Java, Pascal, and SmallTalk.

C# syntaxes are like C++. .NET and the C# library is similar to Java. C# supports modern object-oriented programming language features including Abstraction, Encapsulation, Polymorphism, and Inheritance. C# is a strongly typed language and most types are inherited by the Object class.

C# supports concepts of classes and objects. Classes have members such as fields, properties, events, and methods. Here is a detailed article on C# and OOP.

C# is versatile, modern, and supports modern programming needs. Since its inception, C# language has gone through various upgrades. The latest version of C# is v8.0. Here is a detailed series on C# 8 - Learn C# 8.0 Step by Step.

2. What is an object in C#?

C# language is an object-oriented programming language. Classes are the foundation of C#. A class is a template that defines what a data structure will look like, and how data will be stored, managed, and transferred. A class has fields, properties, methods, and other members.

While classes are concepts, objects are real. Objects are created using class instances. A class defines the type of an object. Objects store real values in computer memory.

Any real-world entity which has some certain characteristics or that can perform some work is called an Object. This object is also called an instance, i.e. a copy of an entity in a programming language. Objects are instances of classes.

For example, we need to create a program that deals with cars. We need to create an entity for the car. Let’s call it a class, Car. A car has four properties, i.e., model, type, color, and size.

To represent a car in programming, we can create a class, Car, with four properties, Model, Type, Color, and Size. These are called members of a class. A class has several types of members, constructors, fields, properties, methods, delegates, and events. A class member can be private, protected, and pubic. Since these properties may be accessed outside the class, these can be public.

An object is an instance of a class. A class can have as many instances as needed. For example, Honda Civic is an instance of Car. In real programming, Honda Civic is an object. Honda Civic is an instance of the class Car. The Model, Type, Color, and Size properties of Honda Civic are Civic, Honda, Red, 4 respectively. BMW 330, Toyota Carolla, Ford 350, Honda CR4, Honda Accord, and Honda Pilot are some more examples of objects of Car.

To learn more about real-world examples of objects and instance, please read, Object Oriented Programming with Real World Scenario.

3. What is Managed or Unmanaged Code?

Managed Code

“Managed code is the code that is developed using the .NET framework and its supported programming languages such as C# or VB.NET. Managed code is directly executed by the Common Language Runtime (CLR or Runtime) and its lifecycle including object creation, memory allocation, and object disposal is managed by the Runtime. Any language that is written in .NET Framework is managed code".

Unmanaged Code

The code that is developed outside of the .NET framework is known as unmanaged code.

“Applications that do not run under the control of the CLR are said to be unmanaged. Languages such as C or C++ or Visual Basic are unmanaged.
The object creation, execution, and disposal of unmanaged code is directly managed by the programmers. If programmers write bad code, it may lead to memory leaks and unwanted resource allocations.”

The .NET Framework provides a mechanism for unmanaged code to be used in managed code and vice versa. The process is done with the help of wrapper classes.

4. What is Boxing and Unboxing in C#?

Boxing and Unboxing both are used for type conversions.

The process of converting from a value type to a reference type is called boxing. Boxing is an implicit conversion. Here is an example of boxing in C#.

```
// Boxing
int anum = 123;
Object obj = anum;
Console.WriteLine(anum);
Console.WriteLine(obj);
```

The process of converting from a reference type to a value type is called unboxing. Here is an example of unboxing in C#.

```
// Unboxing
Object obj2 = 123;
int anum2 = (int)obj;
Console.WriteLine(anum2);
Console.WriteLine(obj);
```

5. What is the difference between a struct and a class in C#?

Class and struct are both user-defined data types, but have some major differences:

Struct
The struct is a value type in C# and it inherits from System.Value Type.
Struct is usually used for smaller amounts of data.
Struct can’t be inherited from other types.
A structure can't be abstract.
No need to create an object with a new keyword.
Do not have permission to create any default constructor.
Class
The class is a reference type in C# and it inherits from the System.Object Type.
Classes are usually used for large amounts of data.
Classes can be inherited from other classes.
A class can be an abstract type.
We can create a default constructor.

About Structure:
structure is a data type of a value type. A struct keyword is used when you are going to define a structure. A structure represents a record and this record can have many attributes that define the structure. You can define a constructor but not destructor for the structure. You can implement one or more interfaces within the structure. You can specify a structure but not as abstract, virtual, or protected. If you do not use the new operator the fields of the structure remain unassigned and you cannot use the object till you initialize the fields.

6. What is the difference between Interface and Abstract Class in C#?

Here are some of the common differences between an interface and an abstract class in C#.
A class can implement any number of interfaces but a subclass can at most use only one abstract class.
An abstract class can have non-abstract methods (concrete methods) while in case of interface, all the methods have to be abstract.
An abstract class can declare or use any variables while an interface is not allowed to do so.
In an abstract class, all data members or functions are private by default while in an interface all are public, we can’t change them manually.
In an abstract class, we need to use abstract keywords to declare abstract methods, while in an interface we don’t need to use that.
An abstract class can’t be used for multiple inheritance while the interface can be used as multiple inheritance.
An abstract class use constructor while in an interface we don’t have any type of constructor

About Interface:

An Interface is an abstract class which has only public abstract methods, and the methods only have the declaration and not the definition. These abstract methods must be implemented in the inherited classes.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace DemoApplication
{
 interface Guru99Interface
 {
  void SetTutorial(int pID, string pName);
  String GetTutorial();
 }

 class Guru99Tutorial : Guru99Interface
 {
  protected int TutorialID;
  protected string TutorialName;

  public void SetTutorial(int pID, string pName)
  {
   TutorialID = pID;
   TutorialName = pName;
  }

  public String GetTutorial()
  {
   return TutorialName;
  }

  static void Main(string[] args)
  {
   Guru99Tutorial pTutor = new Guru99Tutorial();

   pTutor.SetTutorial(1,".Net by Guru99");

   Console.WriteLine(pTutor.GetTutorial());

   Console.ReadKey();
  }
 }
}
```

7. What is enum in C#?

An enum is a value type with a set of related named constants often referred to as an enumerator list. The enum keyword is used to declare an enumeration. It is a primitive data type that is user-defined.

An enum type can be an integer (float, int, byte, double, etc.). But if you use it beside int it has to be cast.

An enum is used to create numeric constants in the .NET framework. All the members of enum are enum type. There must be a numeric value for each enum type.

The default underlying type of the enumeration element is int. By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1.
enum Dow {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
Some points about enum,
Enums are enumerated data types in c#.
Enums are not for the end-user, they are meant for developers.
Enums are strongly typed constant. They are strongly typed, i.e. an enum of one type may not be implicitly assigned to an enum of another type even though the underlying value of their members is the same.
Enumerations (enums) make your code much more readable and understandable.
Enum values are fixed. Enum can be displayed as a string and processed as an integer.
The default type is int, and the approved types are byte, sbyte, short, ushort, uint, long, and ulong.
Every enum type automatically derives from System.Enum and thus we can use System.Enum methods on enums.
Enums are value types and are created on the stack and not on the heap.

8. What is the difference between “continue” and “break” statements in C#?

Using break statement, you can 'jump out of a loop' whereas by using a continue statement, you can 'jump over one iteration' and then resume your loop execution.

Eg. Break Statement

```
using System;
using System.Collections;
using System.Linq;
using System.Text;
namespace break_example {
  Class brk_stmt {
    public static void main(String[] args) {
      for (int i = 0; i <= 5; i++) {
        if (i == 4) {
          break;
        }
        Console.WriteLine("The number is " + i);
        Console.ReadLine();
      }
    }
  }
}
```

Eg. Continue Statement

```
using System;
using System.Collections;
using System.Linq;
using System.Text;
namespace continue_example {
  Class cntnu_stmt {
    public static void main(String[] {
        for (int i = 0; i <= 5; i++) {
          if (i == 4) {
            continue;
          }
          Console.WriteLine("The number is "+ i);
            Console.ReadLine();
          }
        }
      }
    }
```

9. What is the difference between constant and readonly in C#?

Const is nothing but "constant", a variable of which the value is constant but at compile time. 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.

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.

Example

We have a Test Class in which we have two variables, one is readonly and the other is a constant.

```
class Test {
  readonly int read = 10;
  const int cons = 10;
  public Test() {
    read = 100;
    cons = 100;
  }
  public void Check() {
    Console.WriteLine("Read only : {0}", read);
    Console.WriteLine("const : {0}", cons);
  }
}
```

Here, I was trying to change the value of both the variables in constructor, but when I try to change the constant, it gives an error to change their value in the block that I have to call at run time.

Finally, remove that line of code from class and call this Check() function like in the following code snippet:

```
class Program {
  static void Main(string[] args) {
      Test obj = new Test();
      obj.Check();
      Console.ReadLine();
  }
}
class Test {
  readonly int read = 10;
  const int cons = 10;
  public Test() {
      read = 100;
  }
  public void Check() {
      Console.WriteLine("Read only : {0}", read);
      Console.WriteLine("const : {0}", cons);
  }
}
```

10. What is the difference between ref and out keywords?

Ref

It is necessary the parameters should initialize before it pass to ref.
It is not necessary to initialize the value of a parameter before returning to the calling method.
The passing of value through ref parameter is useful when the called method also need to change the value of passed parameter.
When ref keyword is used the data may pass in bi-directional.

Out
It is not necessary to initialize parameters before it pass to out.
It is necessary to initialize the value of a parameter before returning to the calling method.
The declaring of parameter through out parameter is useful when a method return multiple values.
When out keyword is used the data only passed in unidirectional.

An argument passed as ref must be initialized before passing to the method whereas out parameter needs not to be initialized before passing to a method.

11. Can "this" be used within a static method?

We can't use 'this' in a static method because the 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 are called with the name of a class, not by instance, so we can't use this keyword in the body of static Methods. However, in the case of Extension Methods, we can use the parameters of the function.
Let's have a look at the "this" keyword.

The "this" keyword in C# is a special type of reference variable that is implicitly defined within each constructor and non-static method as a first parameter of the type class in which it is defined.

12. What are Properties in C#?

C# properties are members of a C# class that provide a flexible mechanism to read, write or compute the values of private fields, in other words, by using properties, we can access private fields and set their values. Properties in C# are always public data members. C# properties use get and set methods, also known as accessors, to access and assign values to private fields.

What are accessors?

The get and set portions or blocks of a property are called accessors. These are useful to restrict the accessibility of a property. The set accessor specifies that we can assign a value to a private field in a property. Without the set accessor property, it is like a readonly field. With the 'get' accessor we can access the value of the private field. In other words, it returns a single value. A Get accessor specifies that we can access the value of a field publically.

We have three types of properties: Read/Write, ReadOnly, and WriteOnly. Let's see each one by one.

13. What are extension methods in C#?

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.

How to use extension methods?

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.

14. What is the difference between the dispose and finalize methods in C#?

In finalize and dispose, both methods are used to free unmanaged resources.

Finalize
Finalize is used to free unmanaged resources that are not in use, like files, database connections in the application domain and more. These are resources held by an object before that object is destroyed.
In the Internal process, it is called by Garbage Collector and can’t be called manual by user code or any service.
Finalize belongs to System.Object class.
Implement it when you have unmanaged resources in your code, and make sure that these resources are freed when the Garbage collection happens.
Dispose
Dispose is also used to free unmanaged resources that are not in use like files, database connections in the Application domain at any time.
Dispose is explicitly called by manual user code.
If we need to use the dispose method, we must implement that class via IDisposable interface.
It belongs to IDisposable interface.
Implement this when you are writing a custom class that will be used by other users.

15. What is the difference between String and StringBuilder in C#?

StringBuilder and string are both used to string values, but both have many differences on the bases of instance creation and also in performance.

String

A string is an immutable object. Immutable is when we create string objects in code so we cannot modify or change that object in any operations like insert new value, replace or append any value with the existing value in a string object. When we have to do some operations to change string simply it will dispose of the old value of string object and it will create a new instance in memory for hold the new value in a string object, for example:

ASP.NET

Note
It’s an immutable object that holds a string value.
Performance-wise, string is slow because it creates a new instance to override or change the previous value.
String belongs to the System namespace.
StringBuilder

System.Text.Stringbuilder is a mutable object which also holds the string value, mutable means once we create a System.Text.Stringbuilder object. We can use this object for any operation like insert value in an existing string with insert functions also replace or append without creating a new instance of System.Text.Stringbuilder for every time so it’s using the previous object. That way, it works fast compared to the System.String. Let’s see an example to understand System.Text.Stringbuilder.

ASP.NET
Note
StringBuilder is a mutable object.
Performance-wise StringBuilder is very fast because it will use the same instance of StringBuilder object to perform any operation like inserting a value in the existing string.
StringBuilder belongs to System.Text.Stringbuilder namespace.

16. What are delegates in C# and the uses of delegates?

A Delegate is an abstraction of one or more function pointers (as existed in C++; the explanation about this is out of the scope of this article). The .NET has implemented the concept of function pointers in the form of delegates. With delegates, you can treat a function as data. Delegates allow functions to be passed as parameters, returned from a function as a value and stored in an array.
Delegates have the following characteristics:
Delegates are derived from the System.MulticastDelegate class.
They have a signature and a return type. A function that is added to delegates must be compatible with this signature.
Delegates can point to either static or instance methods.
Once a delegate object has been created, it may dynamically invoke the methods it points to at runtime.
Delegates can call methods synchronously and asynchronously.
The delegate contains a couple of useful fields. The first one holds a reference to an object, and the second holds a method pointer. When you invoke the delegate, the instance method is called on the contained reference. However, if the object reference is null then the runtime understands this to mean that the method is a static method. Moreover, invoking a delegate syntactically is the exact same as calling a regular function. Therefore, delegates are perfect for implementing callbacks.

Why Do We Need Delegates?

Historically, the Windows API made frequent use of C-style function pointers to create callback functions. Using a callback, programmers were able to configure one function to report back to another function in the application. So the objective of using a callback is to handle button-clicking, menu-selection, and mouse-moving activities. But the problem with this traditional approach is that the callback functions were not type-safe. In the .NET framework, callbacks are still possible using delegates with a more efficient approach. Delegates maintain three important pieces of information:
The parameters of the method.
The address of the method it calls.
The return type of the method.
A delegate is a solution for situations in which you want to pass methods around to other methods. You are so accustomed to passing data to methods as parameters that the idea of passing methods as an argument instead of data might sound a little strange. However, there are cases in which you have a method that does something, for instance, invoking some other method. You do not know at compile time what this second method is. That information is available only at runtime, hence Delegates are the device to overcome such complications.

Different types of Delegates are:

Single Delegate: A delegate that can call a single method.
Multicast Delegate: A delegate that can call multiple methods. + and – operators are used to subscribe and unsubscribe respectively.
Generic Delegate: It does not require an instance of the delegate to be defined. It is of three types, Action, Funcs and Predicate.
Action– In the above example of delegates and events, we can replace the definition of delegate and event using Action keyword. The Action delegate defines a method that can be called on arguments but does not return a result
Public delegate void deathInfo();
Public event deathInfo deathDate;
//Replacing with Action//
Public event Action deathDate;

Action implicitly refers to a delegate.

Func– A Func delegate defines a method that can be called on arguments and returns a result.
Func <int, string, bool> myDel is same as delegate bool myDel(int a, string b);

Predicate– Defines a method that can be called on arguments and always returns the bool.
Predicate<string> myDel is same as delegate bool myDel(string s);

17. What are sealed classes in C#?

Sealed classes are used to restrict the inheritance feature of object-oriented programming. Once a class is defined as a sealed class, the class cannot be inherited.

In C#, the sealed modifier is used to define a class as sealed. In Visual Basic .NET the Not Inheritable keyword serves the purpose of the sealed class. If a class is derived from a sealed class then the compiler throws an error.

If you have ever noticed, structs are sealed. You cannot derive a class from a struct.

The following class definition defines a sealed class in C#:

```
// Sealed class
sealed class SealedClass
{
}
```

18. What are partial classes?

A partial class is only used to split the definition of a class in two or more classes in the same source code file or more than one source file. You can create a class definition in multiple files, but it will be compiled as one class at run time. Also, when you create an instance of this class, you can access all the methods from all source files with the same object.

Partial Classes can be created in the same namespace. It isn't possible to create a partial class in a different namespace. So use the "partial" keyword with all the class names that you want to bind together with the same name of a class in the same namespace. Let's see an example:

19. What is the difference between boxing and unboxing in C#?

Boxing and Unboxing are both used for type converting, but have some differences:

Boxing

Boxing is the process of converting a value type data type to the object or to any interface data type which is implemented by this value type. When the CLR boxes a value means when CLR converting a value type to Object Type, it wraps the value inside a System.Object and stores it on the heap area in the application domain.

Unboxing

Unboxing is also a process that is used to extract the value type from the object or any implemented interface type. Boxing may be done implicitly, but unboxing has to be explicit by code.

20. What is IEnumerable<> in C#?

IEnumerable is the parent interface for all non-generic collections in System.Collections namespace like ArrayList, HastTable etc. that can be enumerated. For the generic version of this interface as IEnumerable<T> which a parent interface of all generic collections class in System.Collections.Generic namespace like List<> and more.

In System.Collections.Generic.IEnumerable<T> have only a single method which is GetEnumerator() that returns an IEnumerator. IEnumerator provides the power to iterate through the collection by exposing a Current property and Move Next and Reset methods if we don't have this interface as a parent so we can't use iteration by foreach loop or can't use that class object in our LINQ query.

21. What is the difference between late binding and early binding in C#?

Early Binding and Late Binding concepts belong to polymorphism in C#. Polymorphism is the feature of object-oriented programming that allows a language to use the same name in different forms. For example, a method named Add can add integers, doubles, and decimals.

Polymorphism we have 2 different types to achieve that:
Compile Time also known as Early Binding or Overloading.
Run Time is also known as Late Binding or Overriding.
Compile Time Polymorphism or Early Binding

In Compile time polymorphism or Early Binding, we will use multiple methods with the same name but different types of parameters, or maybe the number of parameters. Because of this, we can perform different-different tasks with the same method name in the same class which is also known as Method overloading.

Run Time Polymorphism or Late Binding

Run time polymorphism is also known as late binding. In Run Time Polymorphism or Late Binding, we can use the same method names with the same signatures, which means the same type or the same number of parameters, but not in the same class because the compiler doesn't allow for that at compile time. Therefore, we can use that bind at run time in the derived class when a child class or derived class object will be instantiated. That's why we call it Late Binding. We have to create my parent class functions as partial and in driver or child class as override functions with the override keyword.

22. What are the differences between IEnumerable and IQueryable?

Before we go into the differences, let's learn what the IEnumerable and IQueryable are.

IEnumerable

Is the parent interface for all non-generic collections in System.Collections namespace like ArrayList, HastTable, etc. that can be enumerated. The generic version of this interface is IEnumerable<T>, which a parent interface of all generic collections class in System.Collections.Generic namespace, like List<> and more.

IQueryable

As per MSDN, the IQueryable interface is intended for implementation by query providers. It is only supposed to be implemented by providers that also implement IQueryable<T>. If the provider does not also implement IQueryable<T>, the standard query operators cannot be used on the provider's data source.

The IQueryable interface inherits the IEnumerable interface so that if it represents a query, the results of that query can be enumerated. Enumeration causes the expression tree associated with an IQueryable object to be executed. The definition of "executing an expression tree" is specific to a query provider. For example, it may involve translating the expression tree to an appropriate query language for the underlying data source. Queries that do not return enumerable results are executed when the Execute method is called.

IEnumerable exists in System.Collections Namespace.
IEnumerable is the best way to write query on collections data types like list,Array etc

IEnumerable is the return type for LINQ to Object and LINQ to XML.
IEnumerable doesn’t support lazy loading, SO it is not recommended approach for paging scenario.
Querying data from a database, IEnumerable execute a select query on the server side, load data in-memory on a client-side and then filter data.
IEnumerable Extension methods take functional objects.

IQueryable exists in System. Linq Namespace.
It is the best way to write query data like remote database, service collections.
It is the return type for LINQ to SQL Queries.
Both IEnumerable and IQueryable are forward collection.
IQueryable support lazy loading so we can use paging scenario
Querying data from a database, IQueryable execute the select query on the server side with all filters.
IQueryable Extension methods take expression objects means expression tree.

23. What happens if the inherited interfaces have conflicting method names?

If we implement multiple interfaces in the same class with conflict method names, we don’t need to define all. In other words, we can say if we have conflict methods in the same class, we can’t implement their body independently in the same class because of the same name and same signature. Therefore, we have to use the interface name before the method name to remove this method confiscation. Let’s see an example:

```
interface testInterface1 {
   void Show();
}
interface testInterface2 {
   void Show();
}
class Abc: testInterface1,
   testInterface2 {
      void testInterface1.Show() {
         Console.WriteLine("For testInterface1 !!");
      }
      void testInterface2.Show() {
         Console.WriteLine("For testInterface2 !!");
      }
   }
```

Now see how to use these in a class:

```
class Program {
   static void Main(string[] args) {
      testInterface1 obj1 = new Abc();
      testInterface1 obj2 = new Abc();
      obj1.Show();
      obj2.Show();
      Console.ReadLine();
   }
}
```

24. What are the Arrays in C#?

In C#, an array index starts at zero. That means the first item of an array starts at the 0th position. The position of the last item on an array will total the number of items - 1. So if an array has 10 items, the last 10th item is in the 9th position.

In C#, arrays can be declared as fixed-length or dynamic.

A fixed-length array can store a predefined number of items.

A dynamic array does not have a predefined size. The size of a dynamic array increases as you add new items to the array. You can declare an array of fixed length or dynamic. You can even change a dynamic array to static after it is defined.

Let's take a look at simple declarations of arrays in C#. The following code snippet defines the simplest dynamic array of integer types that do not have a fixed size.

```
int[] intArray;
```

As you can see from the above code snippet, the declaration of an array starts with a type of array followed by a square bracket ([]) and the name of the array.

The following code snippet declares an array that can store 5 items only starting from index 0 to 4.

```
int[] intArray;
intArray = new int[5];
```

The following code snippet declares an array that can store 100 items starting from index 0 to 99.

```
int[] intArray;
intArray = new int[100];
```

25. What is the Constructor Chaining in C#?

Constructor chaining is a way to connect two or more classes in a relationship as Inheritance. In Constructor Chaining, every child class constructor is mapped to a parent class Constructor implicitly by base keyword, so when you create an instance of the child class, it will call the parent's class Constructor. Without it, inheritance is not possible.

26. What's the difference between the Array.CopyTo() and Array.Clone()?

The Array.Clone() method creates a shallow copy of an array. A shallow copy of an Array copies only the elements of the Array, whether they are reference types or value types, but it does not copy the objects that the references refer to. The references in the new Array point to the same objects that the references in the original Array point to.

The CopyTo() static method of the Array class copies a section of an array to another array. The CopyTo method copies all the elements of an array to another one-dimension array. The code listed in Listing 9 copies contents of an integer array to an array of object types.

27. Can Multiple Catch Blocks be executed in C#?

We can use multiple catch blocks with a try statement. Each catch block can catch a different exception. The following code example shows how to implement multiple catch statements with a single try statement.

```
using System;
class MyClient {
  public static void Main() {
    int x = 0;
    int div = 0;
    try {
      div = 100 / x;
      Console.WriteLine("Not executed line");
    } catch (DivideByZeroException de) {
      Console.WriteLine("DivideByZeroException");
    } catch (Exception ee) {
      Console.WriteLine("Exception");
    } finally {
      Console.WriteLine("Finally Block");
    }
    Console.WriteLine("Result is {0}", div);
  }
}
```

28. What are Singleton Design Patterns and how to implement them in C#?

What is a Singleton Design Pattern?
Ensures a class has only one instance and provides a global point of access to it.
A Singleton is a class that only allows a single instance of itself to be created and usually gives simple access to that instance.
Most commonly, singletons don't allow any parameters to be specified when creating the instance since the second request of an instance with a different parameter could be problematic! (If the same instance should be accessed for all requests with the same parameter then the factory pattern is more appropriate.)
There are various ways to implement the Singleton Pattern in C#. The following are the common characteristics of a Singleton Pattern.
A single constructor, that is private and parameterless.
The class is sealed.
A static variable that holds a reference to the single created instance, if any.
A public static means of getting the reference to the single created instance, creating one if necessary.
Example of how to write code with Singleton:

```
namespace Singleton {
  class Program {
    static void Main(string[] args) {
      Calculate.Instance.ValueOne = 10.5;
      Calculate.Instance.ValueTwo = 5.5;
      Console.WriteLine("Addition : " + Calculate.Instance.Addition());
      Console.WriteLine("Subtraction : " + Calculate.Instance.Subtraction());
      Console.WriteLine("Multiplication : " + Calculate.Instance.Multiplication());
      Console.WriteLine("Division : " + Calculate.Instance.Division());
      Console.WriteLine("\n----------------------\n");
      Calculate.Instance.ValueTwo = 10.5;
      Console.WriteLine("Addition : " + Calculate.Instance.Addition());
      Console.WriteLine("Subtraction : " + Calculate.Instance.Subtraction());
      Console.WriteLine("Multiplication : " + Calculate.Instance.Multiplication());
      Console.WriteLine("Division : " + Calculate.Instance.Division());
      Console.ReadLine();
    }
  }
  public sealed class Calculate {
    private Calculate() {}
```

```
        private static Calculate instance = null;
        public static Calculate Instance {
            get {
                if (instance == null) {
                    instance = new Calculate();
                }
                return instance;
            }
        }
        public double ValueOne {
            get;
            set;
        }
        public double ValueTwo {
            get;
            set;
        }
        public double Addition() {
            return ValueOne + ValueTwo;
        }
        public double Subtraction() {
            return ValueOne - ValueTwo;
        }
        public double Multiplication() {
            return ValueOne * ValueTwo;
        }
        public double Division() {
            return ValueOne / ValueTwo;
        }
    }
}
```

29. Difference between Throw Exception and Throw Clause

The basic difference is that the Throw exception overwrites the stack trace. This makes it hard to find the original code line number that has thrown the exception.

Throw basically retains the stack information and adds to the stack information in the exception that it is thrown.

Let's see what it means to better understand the differences. I am using a console application to easily test and see how the usage of the two differ in their functionality.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace TestingThrowExceptions {
    class Program {
        public void ExceptionMethod() {
            throw new Exception("Original Exception occurred in ExceptionMethod");
        }
        static void Main(string[] args) {
            Program p = new Program();
            try {
               p.ExceptionMethod();
            } catch (Exception ex) {
               throw ex;
            }
         }
      }
   }
```

Now run the code by pressing the F5 key and see what happens. It returns an exception and look at the stack trace.

30. What are Indexers in C#?

C# introduces a new concept known as Indexers which are used for treating an object as an array. The indexers are usually known as smart arrays in C#. They are not an essential part of object-oriented programming.

Defining an indexer allows you to create classes that act as virtual arrays. Instances of that class can be accessed using the [] array access operator.

Creating an Indexer

```
< modifier > <
return type > this[argument list] {
   get {
      // your get block code
   }
   set {
      // your set block code
   }
}
```

In the above code,

<modifier>

can be private, public, protected or internal.

<return type>

can be any valid C# types.

31. What is a multicast delegate in C#?

Delegate is one of the base types in .NET. Delegate is a class that is used to create and invoke delegates at runtime.

A delegate in C# allows developers to treat methods as objects and invoke them from their code.

Implement Multicast Delegates Example:

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

```
delegate void MDelegate();
class DM {
  static public void Display() {
    Console.WriteLine("Meerut");
  }
  static public void print() {
    Console.WriteLine("Roorkee");
  }
}
class MTest {
  public static void Main() {
    MDelegate m1 = new MDelegate(DM.Display);
    MDelegate m2 = new MDelegate(DM.print);
    MDelegate m3 = m1 + m2;
    MDelegate m4 = m2 + m1;
    MDelegate m5 = m3 - m2;
    m3();
    m4();
    m5();
  }
}
```

32. Difference between the Equality Operator (==) and Equals() Method in C#

Both the == Operator and the Equals() method are used to compare two value type data items or reference type data items. The Equality Operator (==) is the comparison operator and the Equals() method compares the contents of a string. The == Operator compares the reference identity while the Equals() method compares only contents. Let's see with some examples.

In this example, we assigned a string variable to another variable. A string is a reference type and in the following example, a string variable is assigned to another string variable so they are referring to the same identity in the heap and both have the same content so you get True output for both the == Operator and the Equals() method.

```
using System;

namespace ComparisionExample {

  class Program {

    static void Main(string[] args) {

      string name = "sandeep";

      string myName = name;

      Console.WriteLine("== operator result is {0}", name == myName);

      Console.WriteLine("Equals method result is {0}", name.Equals(myName));

      Console.ReadKey();
```

```
        }

    }

}
```

33. What's the Difference between the Is and As operator in C#

"is" operator

In C# language, we use the "is" operator to check the object type. If two objects are of the same type, it returns true, else it returns false.

Let's understand this in our C# code. We declare two classes, Speaker and Author.

```
class Speaker {
    public string Name {
        get;
        set;
    }
}
class Author {
    public string Name {
        get;
        set;
    }
}
```

Now, let's create an object of type Speaker:

```
var speaker = new Speaker { Name="Gaurav Kumar Arora"};
```

Now, let’s check if the object is Speaker type:

```
var isTrue = speaker is Speaker;
```

In the preceding, we are checking the matching type. Yes, our speaker is an object of Speaker type.

```
Console.WriteLine("speaker is of Speaker type:{0}", isTrue);
```

So, the results are true.

But, here we get false:

```
var author = new Author { Name = "Gaurav Kumar Arora" };
var isTrue = speaker is Author;
Console.WriteLine("speaker is of Author type:{0}", isTrue);
```

Because our speaker is not an object of Author type.

"as" operator

The "as" operator behaves in a similar way as the "is" operator. The only difference is it returns the object if both are compatible with that type. Else it returns a null.

Let's understand this in our C# code.

```
public static string GetAuthorName(dynamic obj)
  {
     Author authorObj = obj as Author;
     return (authorObj != null) ? authorObj.Name : string.Empty;
  }
```

We have a method that accepts a dynamic object and returns the object name property if the object is of the Author type.

Here, we’ve declared two objects:
var speaker = new Speaker { Name="Gaurav Kumar Arora"};
var author = new Author { Name = "Gaurav Kumar Arora" };
The following returns the "Name" property:
var authorName = GetAuthorName(author);
Console.WriteLine("Author name is:{0}", authorName);
It returns an empty string:
authorName = GetAuthorName(speaker);
Console.WriteLine("Author name is:{0}", authorName);
Learn more about is vs as operators here, "is" and "as" Operators of C#

34. How to use Nullable<> Types in C#?

A nullable type is a data type is that contains the defined data type or the null value.

This nullable type concept is not compatible with "var".

Any data type can be declared nullable type with the help of operator "?".

For example, the following code declares the int 'i' as a null.
int? i = null;
As discussed in the previous section "var" is not compatible with nullable types. So, if you declare the following, you will get an error.
var? i = null;

"is" operator

In C# language, we use the "is" operator to check the object type. If two objects are of the same type, it returns true, else it returns false.

Let's understand this in our C# code. We declare two classes, Speaker and Author.

```
class Speaker {
  public string Name {
    get;
    set;
  }
}
class Author {
  public string Name {
    get;
    set;
  }
}
```

Now, let's create an object of type Speaker:
var speaker = new Speaker { Name="Gaurav Kumar Arora"};
Now, let’s check if the object is Speaker type:
var isTrue = speaker is Speaker;
In the preceding, we are checking the matching type. Yes, our speaker is an object of Speaker type.

```
Console.WriteLine("speaker is of Speaker type:{0}", isTrue);
```
So, the results are true.

But, here we get false:
```
var author = new Author { Name = "Gaurav Kumar Arora" };
var isTrue = speaker is Author;
Console.WriteLine("speaker is of Author type:{0}", isTrue);
```
Because our speaker is not an object of Author type.

"as" operator

The "as" operator behaves in a similar way as the "is" operator. The only difference is it returns the object if both are compatible with that type. Else it returns a null.

Let's understand this in our C# code.
```
public static string GetAuthorName(dynamic obj)
 {
   Author authorObj = obj as Author;
   return (authorObj != null) ? authorObj.Name : string.Empty;
 }
```
We have a method that accepts a dynamic object and returns the object name property if the object is of the Author type.

Here, we've declared two objects:
```
var speaker = new Speaker { Name="Gaurav Kumar Arora"};
var author = new Author { Name = "Gaurav Kumar Arora" };
```
The following returns the "Name" property:
```
var authorName = GetAuthorName(author);
Console.WriteLine("Author name is:{0}", authorName);
```
It returns an empty string:
```
authorName = GetAuthorName(speaker);
Console.WriteLine("Author name is:{0}", authorName);
```

35. What are Different Ways a Method can be Overloaded?

Method overloading is a way to achieve compile-time polymorphism where we can use a method with the same name but different signatures. For example, the following code example has a method volume with three different signatures based on the number and type of parameters and return values.

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

namespace Hello_Word {
  class overloding {
    public static void Main() {
        Console.WriteLine(volume(10));
        Console.WriteLine(volume(2.5F, 8));
        Console.WriteLine(volume(100L, 75, 15));
        Console.ReadLine();
    }

    static int volume(int x) {
        return (x * x * x);
    }

    static double volume(float r, int h) {
        return (3.14 * r * r * h);
    }

    static long volume(long l, int b, int h) {
        return (l * b * h);
    }
  }
}
```

Note
If we have a method that has two parameter object type and has the same name method with two integer parameters, when we call that method with int value, it will call that method with integer parameters instead of the object type parameters method.

36. What is an Object Pool in .Net?

Object Pooling in .NET allows objects to keep in the memory pool so the objects can be reused without recreating them. This article explains what object pooling is in .NET and how to implement object pooling in C#.

What does it mean?

Object Pool is a container of objects that are ready for use. Whenever there is a request for a new object, the pool manager will take the request and it will be served by allocating an object from the pool.

How does it work?

We are going to use the Factory pattern for this purpose. We will have a factory method, which will take care of the creation of objects. Whenever there is a request for a new object, the factory method will look into the object pool (we use Queue object). If there is any object available within the allowed limit, it will return the object (value object), otherwise, a new object will be created and give you back.

37. What are Generics in C#?

Generics 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.

You write the specifications for the class or the method, with substitute parameters for data types. When the compiler encounters a constructor for the class or a function call for the method, it generates code to handle the specific data type.

Generic classes and methods combine reusability, type safety and efficiency in a way that their non-generic counterparts cannot. Generics are most frequently used with collections and the methods that operate on them. Version 2.0 of the .NET Framework class library provides a new namespace, System.Collections.Generic, that contains several new generic-based collection classes. It is recommended that all applications that target the .NET Framework 2.0 and later use the new generic collection classes instead of the older non-generic counterparts such as ArrayList.

Features of Generics

Generics are a technique that enriches your programs in the following ways:
It helps you to maximize code reuse, type safety, and performance.
You can create generic collection classes. The .NET Framework class library contains several new generic collection classes in the System.Collections.Generic namespace. You may use these generic collection classes instead of the collection classes in the System.Collections namespace.
You can create your own generic interfaces, classes, methods, events, and delegates.
You may create generic classes constrained to enable access to methods on specific data types.
You may get information on the types used in a generic data type at run-time using reflection.

38. Describe Accessibility Modifiers in C#

Access modifiers are keywords used to specify the declared accessibility of a member or a type.

Access modifiers are keywords used to specify the scope of accessibility of a member of a type or the type itself. For example, a public class is accessible to the entire world, while an internal class may be accessible to the assembly only.

Why use access modifiers?

Access modifiers are an integral part of object-oriented programming. Access modifiers are used to implement the encapsulation of OOP. Access modifiers allow you to define who does or who doesn't have access to certain features.

In C# there are 6 different types of Access Modifiers:

Modifier Description
public There are no restrictions on accessing public members.
private Access is limited to within the class definition. This is the default access modifier type if none is formally specified
protected Access is limited to within the class definition and any class that inherits from the class
internal Access is limited exclusively to classes defined within the current project assembly
protected internal Access is limited to the current assembly and types derived from the containing class. All members in the current project and all members in derived class can access the variables.
private protected Access is limited to the containing class or types derived from the containing class within the current assembly.

39. What is a Virtual Method in C#?

A virtual method is a method that can be redefined in derived classes. A virtual method has an implementation in a base class as well as derived the class. It is used when a method's basic functionality is the same but sometimes more functionality is needed in the derived class. A virtual method is created in the base class that can be overridden in the derived class. We create a virtual method in the base class using the virtual keyword and that method is overridden in the derived class using the override keyword.

When a method is declared as a virtual method in a base class then that method can be defined in a base class and it is optional for the derived class to override that method. The overriding method also provides more than one form for a method. Hence, it is also an example of polymorphism.

When a method is declared as a virtual method in a base class and that method has the same definition in a derived class then there is no need to override it in the derived class. But when a virtual method has a different definition in the base class and the derived class then there is a need to override it in the derived class.

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member if no derived class has overridden the member.

Virtual Method
By default, methods are non-virtual. We can't override a non-virtual method.
We can't use the virtual modifier with static, abstract, private or override modifiers.

40. What is the Difference between an Array and ArrayList in C#?

Must include System namespace to use array.
Array Declaration & Initialization:
int[] arr = new int[5]
int[] arr = new int[5]{1, 2, 3, 4, 5};
int[] arr = {1, 2, 3, 4, 5};
Array stores a fixed number of elements. The size of an Array must be specified at the time of initialization.
Array is strongly typed. This means that an array can store only specific type of items\elements.
No need to cast elements of an array while retrieving because it is strongly typed and stores a specific type of items only.
Performs faster than ArrayList because it is strongly typed.
Use static helper class Array to perform different tasks on the array.

ArrayList
Must include System.Collections namespace to use ArraList.
ArrayList Declaration & Initialization:
ArrayList arList = new ArrayList();
arList.Add(1);
arList.Add("Two");
arList.Add(false);
ArrayList grows automatically and you don't need to specify the size.
ArrayList can store any type of items\elements.
The items of ArrayList need to be cast to an appropriate data type while retrieving. So, boxing and unboxing happens.
Performs slows because of boxging and unboxing.
ArrayList itself includes various utility methods for various tasks.

41. What are Value types and Reference types in C#?

In C#, data types can be of two types, value types, and reference types. Value type variables contain their object (or data) directly. If we copy one value type variable to another then we are actually making a copy of the object for the second variable. Both of them will independently operate on their values, Value type data types are stored on a stack and reference data types are stored on a heap.

In C#, basic data types include int, char, bool, and long, which are value types. Classes and collections are reference types.

42. What is Serialization in C#?

Serialization in C# is the process of converting an object into a stream of bytes to store the object to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

There are three types of serialization,
Binary serialization (Save your object data into binary format).
Soap Serialization (Save your object data into binary format; mainly used in network-related communication).
XmlSerialization (Save your object data into an XML file).

43. How do you use the "using" statement in C#?

There are two ways to use the using keyword in C#. One is as a directive and the other is as a statement. Let's explain!
using Directive
Generally, we use the using keyword to add namespaces in code-behind and class files. Then it makes available all the classes, interfaces and abstract classes and their methods and properties on the current page. Adding a namespace can be done in the following two ways:
Using Statement
This is another way to use the using keyword in C#. It plays a vital role in improving performance in Garbage Collection.

44. What is a Jagged Array in C#?

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

A special type of array is introduced in C#. A Jagged Array is an array of an array in which the length of each array index can differ.

Example
```
int[][] jagArray = new int[5][];
```
In the above declaration, the rows are fixed in size. But columns are not specified as they can vary.

Declaring and initializing a jagged array.
```
int[][] jaggedArray = new int[5][];
jaggedArray[0] = new int[3];
jaggedArray[1] = new int[5];
jaggedArray[2] = new int[2];
jaggedArray[3] = new int[8];
jaggedArray[4] = new int[10];
jaggedArray[0] = new int[] { 3, 5, 7, };
jaggedArray[1] = new int[] { 1, 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 1, 6 };
jaggedArray[3] = new int[] { 1, 0, 2, 4, 6, 45, 67, 78 };
jaggedArray[4] = new int[] { 1, 0, 2, 4, 6, 34, 54, 67, 87, 78 };
```

45. What is Multithreading with .NET?

Multithreading allows a program to run multiple threads concurrently. This article explains how multithreading works in .NET. This article covers the entire range of threading areas from thread creation, race conditions, deadlocks, monitors, mutexes, synchronization and semaphores and so on.

The real usage of a thread is not about a single sequential thread, but rather using multiple threads in a single program. Multiple threads running at the same time and performing various tasks are referred to as Multithreading. A thread is considered to be a lightweight process because it runs within the context of a program and takes advantage of the resources allocated for that program.

A single-threaded process contains only one thread while a multithreaded process contains more than one thread for execution.

46. What are Anonymous Types in C#?

Anonymous types allow us to create new types without defining them. This is a way of defining read-only properties in a single object without having to define each type explicitly. Here, Type is generated by the compiler and is accessible only for the current block of code. The type of properties is also inferred by the compiler.

We can create anonymous types by using "new" keyword together with the object initializer.

Example

```
var anonymousData = new
{
   ForeName = "Jignesh",
   SurName = "Trivedi"
};
Console.WriteLine("First Name : " + anonymousData.ForeName);
```

Anonymous Types with LINQ Example

Anonymous types are also used with the "Select" clause of LINQ query expression to return a subset of properties.

Example

If any object collection has properties calling FirstName, LastName, DOB, etc... and you want only FirstName and LastName after the Querying the data, then:

```
   class MyData {
      public string FirstName {
         get;
         set;
      }
      public string LastName {
         get;
         set;
      }
      public DateTime DOB {
         get;
         set;
      }
      public string MiddleName {
```

```
      get;
      set;
    }
  }
  static void Main(string[] args) {
    // Create Dummy Data to fill Collection.
    List < MyData > data = new List < MyData > ();
    data.Add(new MyData {
      FirstName = "Jignesh", LastName = "Trivedi", MiddleName = "G", DOB = new
DateTime(1990, 12, 30)
    });
    data.Add(new MyData {
      FirstName = "Tejas", LastName = "Trivedi", MiddleName = "G", DOB = new DateTime(1995,
11, 6)
    });
    data.Add(new MyData {
      FirstName = "Rakesh", LastName = "Trivedi", MiddleName = "G", DOB = new DateTime(1993,
10, 8)
    });
    data.Add(new MyData {
      FirstName = "Amit", LastName = "Vyas", MiddleName = "P", DOB = newDateTime(1983, 6,
15)
    });
    data.Add(new MyData {
      FirstName = "Yash", LastName = "Pandiya", MiddleName = "K", DOB = newDateTime(1988,
7, 20)
    });
  }
  var anonymousData = from pl in data
  select new {
    pl.FirstName, pl.LastName
  };
  foreach(var m in anonymousData) {
    Console.WriteLine("Name : " + m.FirstName + " " + m.LastName);
  }
}
```

47. What is a Hashtable in C#?

A Hashtable is a collection that stores (Keys, Values) pairs. Here, the Keys are used to find the storage location and is immutable and cannot have duplicate entries in a Hashtable. The .Net Framework has provided a Hash Table class that contains all the functionality required to implement a hash table without any additional development. The hash table is a general-purpose dictionary collection. Each item within the collection is a DictionaryEntry object with two properties: a key object and a value object. These are known as Key/Value. When items are added to a hash table, a hash code is generated automatically. This code is hidden from the developer. Access to the table's values is achieved using the key object for identification. As the items in the collection are sorted according to the hidden hash code, the items should be considered to be randomly ordered.

The Hashtable Collection

The Base Class libraries offer a Hashtable Class that is defined in the System.Collections namespace, so you don't have to code your own hash tables. It processes each key of the hash that you add every time and then uses the hash code to look up the element very quickly. The capacity of a hash table is the number of elements the hash table can hold. As elements are added to a hash table, the capacity is automatically increased as required through reallocation. It is an older .Net Framework type.

Declaring a Hashtable

The Hashtable class is generally found in the namespace called System.Collections. So to execute any of the examples, we have to add using System.Collections; to the source code. The declaration for the Hashtable is:

48. What is LINQ in C#?

LINQ stands for Language Integrated Query. LINQ is a data querying methodology that provides querying capabilities to .NET languages with a syntax similar to a SQL query.

LINQ has a great power of querying on any source of data. The data source could be collections of objects, database or XML files. We can easily retrieve data from any object that implements the IEnumerable<T> interface.

Advantages of LINQ
LINQ offers an object-based, language-integrated way to query over data no matter where that data came from. So through LINQ, we can query a database and XML as well as collections.
Compile-time syntax checking.
It allows you to query collections like arrays, enumerable classes, etc... in the native language of your application, like in VB or C# in much the same way you would query a database using SQL.

49. What is File Handling in C#.Net?

The System.IO namespace provides four classes that allow you to manipulate individual files, as well as interact with a machine directory structure. The Directory and File directly extend System.Object and supports the creation, copying, moving and deletion of files using various static methods. They only contain static methods and are never instantiated. The FileInfo and DirecotryInfo types are derived from the abstract class FileSystemInfo type and they are typically employed for obtaining the full details of a file or directory because their members tend to return strongly typed objects. They implement roughly the same public methods as a Directory and a File but they are stateful and members of these classes are not static.

50. What is Reflection in C#?

Reflection is the process of runtime type discovery to inspect metadata, CIL code, late binding, and self-generating code. At the run time by using reflection, we can access the same "type" information as displayed by the ildasm utility at design time. The reflection is analogous to reverse engineering in which we can break an existing *.exe or *.dll assembly to explore defined significant contents information, including methods, fields, events, and properties.

You can dynamically discover the set of interfaces supported by a given type using the System.Reflection namespace.

Reflection typically is used to dump out the loaded assemblies list, their reference to inspect methods, properties etcetera. Reflection is also used in the external disassembling tools such as Reflector, Fxcop, and NUnit because .NET tools don't need to parse the source code similar to C++.

Metadata Investigation

The following program depicts the process of reflection by creating a console-based application. This program will display the details of the fields, methods, properties, and interfaces for any type within the mscorlib.dll assembly. Before proceeding, it is mandatory to import "System.Reflection".

Here, we are defining a number of static methods in the program class to enumerate fields, methods, and interfaces in the specified type. The static method takes a single "System.Type" parameter and returns void.

```
static void FieldInvestigation(Type t) {
  Console.WriteLine("*********Fields*********");
  FieldInfo[] fld = t.GetFields();
  foreach(FieldInfo f in fld) {
    Console.WriteLine("-->{0}", f.Name);
  }
}

static void MethodInvestigation(Type t) {
  Console.WriteLine("*********Methods*********");
  MethodInfo[] mth = t.GetMethods();
  foreach(MethodInfo m in mth) {
    Console.WriteLine("-->{0}", m.Name);
  }
}
```

51. Define Constructors

A constructor is a special method of the class which gets automatically invoked whenever an instance of the class is created. Like methods, a constructor also contains the collection of instructions that are executed at the time of Object creation. It is used to assign initial values to the data members of the same class.

What is a constructor in C#?

A special method of the class that is automatically invoked when an instance of the class is created is called a constructor. The main use of constructors is to initialize the private fields of the class while creating an instance for the class. When you have not created a constructor in the class, the compiler will automatically create a default constructor of the class. The default constructor initializes all numeric fields in the class to zero and all string and object fields to null.

Some of the key points regarding constructor are
A class can have any number of constructors.
A constructor doesn't have any return type, not even void.
A static constructor can not be a parametrized constructor.
Within a class, you can create one static constructor only.

In C#, constructors can be divided into 5 types
Default Constructor
Parameterized Constructor
Copy Constructor
Static Constructor

Private Constructor
Now, let's see each constructor type with the example below.

Default Constructor in C#

A constructor without any parameters is called a default constructor; in other words, this type of constructor does not take parameters. The drawback of a default constructor is that every instance of the class will be initialized to the same values and it is not possible to initialize each instance of the class with different values. The default constructor initializes:
All numeric fields in the class to zero.
All string and object fields to null.

Parameterized Constructor in C#

A constructor with at least one parameter is called a parameterized constructor. The advantage of a parameterized constructor is that you can initialize each instance of the class with a different value.

Copy Constructor in C#

The constructor which creates an object by copying variables from another object is called a copy constructor. The purpose of a copy constructor is to initialize a new instance to the values of an existing instance.

Static Constructor in C#

When a constructor is created using a static keyword, it will be invoked only once for all of the instances of the class and it is invoked during the creation of the first instance of the class or the first reference to a static member in the class. A static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.

Some key points of a static constructor are:
A static constructor does not take access modifiers or have parameters.
A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.
A static constructor cannot be called directly.
The user has no control over when the static constructor is executed in the program.
A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.

Private Constructor in C#

When a constructor is created with a private specifier, it is not possible for other classes to derive from this class, neither is it possible to create an instance of this class. They are usually used in classes that contain static members only. Some key points of a private constructor are:

One use of a private constructor is when we have only static members.
It provides an implementation of a singleton class pattern
Once we provide a constructor that is either private or public or any, the compiler will not add the parameter-less public constructor to the class.

52. Can we use "this" command within a static method?

We can't use 'This' in a static method because we can only use static variables/methods in a static method.

53. What are circular references?

Circular reference is situation in which two or more resources are interdependent on each other causes the lock condition and make the resources unusable.

54. What are Custom Exceptions?

Sometimes there are some errors that need to be handled as per user requirements. Custom exceptions are used for them and are used defined exceptions.

55. What is the difference between method overriding and method overloading?

In method overriding, we change the method definition in the derived class that changes the method behavior. Method overloading is creating a method with the same name within the same class having different signatures.

56. What are the different ways a method can be overloaded?

Methods can be overloaded using different data types for a parameter, different order of parameters, and different number of parameters.

57. Why can't you specify the accessibility modifier for methods inside the interface?

In an interface, we have virtual methods that do not have method definition. All the methods are there to be overridden in the derived class. That's why they all are public.

58. How can we set the class to be inherited, but prevent the method from being over-ridden?

Declare the class as public and make the method sealed to prevent it from being overridden.

59. What are C# attributes and its significance?

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called reflection.

C# provides developers a way to define declarative tags on certain entities, eg. Class, method, etc. are called attributes. The attribute's information can be retrieved at runtime using Reflection.

60. Describe the different C# classes in detail.

There are 4 types of classes that we can use in C#:

Static Class: It is the type of class that cannot be instantiated, in other words, we cannot create an object of that class using the new keyword and the class members can be called directly using their class name.

Abstract Class: Abstract classes are declared using the abstract keyword. Objects cannot be created for abstract classes. If you want to use it then it must be inherited in a subclass. You can easily define abstract or non-abstract methods within an Abstract class. The methods inside the abstract class can either have an implementation or no implementation.

Partial Class: It is a type of class that allows dividing their properties, methods, and events into multiple source files, and at compile time these files are combined into a single class.

Sealed Class: One cannot inherit a sealed class from another class and restricts the class properties. Any access modifiers cannot be applied to the sealed class.

61. Explain how code gets compiled in C#?

It takes 4 steps to get a code to get compiled in C#. Below are the steps:

First, compile the source code in the managed code compatible with the C# compiler.
Second, combine the above newly created code into assemblies.
Third, load the CLR.
Last, execute the assembly by CLR to generate the output.

62. How you can define the exception handling in C#?

An exception is a raised problem that may occur during the execution of the program. Handling exceptions offers a simple way to pass the control within the program whenever an exception is raised. C# exceptions are handled by using 4 keywords and those are try, catch, finally, throw.

try: a raised exception finds a particular block of code to get handled. There is no limit on the number of catch blocks that you will use in your program to handle different types of exception raised.
catch: you can handle the raised exception within this catch block. You can mention the steps that you want to do to solve the error or you can ignore the error by suppressing it by the code.
Finally: irrespective of the error, if you still want some set of instructions to get displayed then you can use those statements within the finally block and it will display it on the screen.
throw: you can throw an exception using the throw statement. It will display the type of error you are getting.

63. Explain the concept of Destructor in detail. Explain it with an example.

A destructor is a member that works just the opposite of the constructor. Unlike constructors, destructors mainly delete the object. The destructor name must match exactly with the class name just like a constructor. A destructor block always starts with the tilde (~) symbol.

Syntax:

```
~class_name()
{
//code
}
```

A destructor is called automatically:

when the program finishes its execution.
Whenever a scope of the program ends that defines a local variable.
Whenever you call the delete operator from your program.

64. Define method overloading with example.

Method overloading allows programmers to use multiple methods but with the same name. Every defined method within a program can be differentiated on the basis of the number and the type of method arguments. It is a concept based on polymorphism.

Method overloading can be achieved by the following:

By changing the number of parameters in the given method
By changing the order of parameters passed to a method

By using different data types as the passed parameters

65. What are the control statements that are used in C#?

You can control the flow of your set of instructions by using control statements and we majorly focus on if statements. There are a few types of if statements that we consider for making situations to control the flow of execution within a program.

These are the 4 types of if statements:

If
If-else
Nested if
If-else-if
These statements are commonly used within programs.

If statements checks for the user given condition to satisfy their programming condition. If it returns true then the set of instructions will be executed.

Syntax:

```
If(any condition)
{
//code to be executed if the condition returns true
}
```

If-else statement checks for the given condition, if the condition turns out to be false then the flow will transfer to the else statement and it will execute the else instructions. In case, the if condition turns out to be true then the if instructions will get executed.

Syntax:

```
If(condition)
{
//code to be run if the condition is true
}
Else
{
//code to be run if the if-condition is false
}
```

Nested if statement checks for the condition, if the condition is true then it will check for the inner if statement and keeps going on for the last if statement. If any of the conditions are true then it will execute the particular if instructions and stops the if loop there.

Syntax:

```
If (condition to be checked)
{
//code
If(condition 2)
{
//code for if-statement 2
}
}
```

If else-if checks for the given condition, if the condition is not true then the control will go to the next else condition, if that condition is not true it will keep on checking for next else conditions. If any of the conditions did not pass then the last else instructions will get executed.

Syntax:

```
If(condition 1 to be checked)
{
//code for condition 1
}
Else (condition 2 to be checked)
{
//code for condition 2
}
Else
{
//code will run if no other condition is true
}
```

66. What is a different approach to the passing parameter in C#?

Parameters can be passed in three different ways to any defined methods and they are defined below:

Value Parameters: it will pass the actual value of the parameter to the formal parameter. In this case, any changes that are made into the formal parameter of the function will be having no effect on the actual value of the argument.

Reference Parameters: with this method, you can copy the argument that refers to the memory location into the formal parameter that means any changes made to the parameter affect the argument.

Output Parameters: This method returns more than one value to the method.

67. What are the different collection classes in C#?

Collection classes are classes that are mainly used for data storage and retrieval. These collection classes will serve many purposes like allocating dynamic memory during run time and you can even access the items of the collection using the index value that makes the search easier and faster. These collection classes belong to the object class.

There are many collection classes which are as follows:

Array list: it refers to the ordered collection of the objects that are indexed individually. You can use it as an alternative to the array. Using index you can easily add or remove the items off the list and it will resize itself automatically. It works well for dynamic memory allocation, adding or searching items in the list.

Hash table: if you want to access the item of the hash table then you can use the key-value to refer to the original assigned value to the variable. Each item in the hash table is stored as a key/value pair and the item is referenced with its key value.

Stack: it works on the concept of last-in and first-out collection of the objects. Whenever you add an item to the list it is called pushing and when you remove the item off the list it is called popping.

Sorted list: this collection class uses the combination of key and the index to access the item in a list.

Queue: this collection works on the concept of first-in and first-out collection of the object. Adding an item to the list is call enqueue and removing the item off the list is call deque.

BitArray: this collection class is used to represent the array in binary form (0 and 1). You can use this collection class when you do not know the number and the items can be accessed by using integer indexes that start from zero.

68. Explain the concept of thread in C#.

A thread can be defined as the execution flow of any program and defines a unique flow of control. You can manage these threads' execution time so that their execution does not overlap the execution of other threads and prevent deadlock or to maintain efficient usage of resources. Threads are lightweight programs that save the CPU consumption and increase the efficiency of the application. The thread cycle starts with the creation of the object of system.threading.thread class and ends when the thread terminates.

System.threading.thread class allows you to handle multiple threads and the first thread always runs in a process called the main thread. Whenever you run a program in C#, the main thread runs automatically.

69. What are Namespaces in C#?

Use of namespaces is for organizing large code projects. The most widely used namespace in C# is System. Namespaces are created using the namespace keyword. It is possible to use one namespace in another, known as Nested Namespaces.

70. Is it possible to use this keyword within a static method in C#?

A special type of reference variable, this keyword is implicitly defined with each non-static method and constructor as the first parameter of the type class, which defines it. Static methods don’t belong to a particular instance. Instead, they exist without creating an instance of the class and calls with the name of the class. Because this keyword returns a reference to the current instance of the class containing it, it can’t be used in a static method. Although we can’t use this keyword within a static method, we can use it in the function parameters of Extension Methods.

71. Why do we use Async and Await in C#?

Processes belonging to asynchronous programming run independently of the main or other processes. In C#, using Async and Await keywords for creating asynchronous methods.

72. What is the Race condition in C#?

When two threads access the same resource and try to change it at the same time, we have a race condition. It is almost impossible to predict which thread succeeds in accessing the resource first. When two threads try to write a value to the same resource, the last value written is saved.

73. What do you understand by Get and Set Accessor properties?

Made using properties, Get and Set are called accessors in C#. A property enables reading and writing to the value of a private field. Accessors are used for accessing such private fields. While we use the Get property for returning the value of a property, use the Set property for setting the value.

74. What is the lock statement in C#?

Lock statement is used to ensure that one thread doesn?t enter a critical section of code while another thread is in the critical section. If another thread attempts to enter a locked code it will wait, block, until the object is released.

75. What is Garbage Collection?

Garbage Collection is a process of releasing memory automatically occupied by objects which are no longer accessible.

In the common language runtime (CLR), the garbage collector (GC) serves as an automatic memory manager. The garbage collector manages the allocation and release of memory for an application. For developers working with managed code, this means that you don't have to write code to perform memory management tasks.

76. What are Events?

Events in C# follow a concept where it consists of a Publisher, Subscriber, Notification and a handler. You can think of an event as nothing but an encapsulated delegate.

Example:

```
public Delegate void TestEvent();
public TestEvent TestEvent1;
```

77. What are dynamic type variables in C#

You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.

Syntax for declaring a dynamic type is −

```
dynamic <variable_name> = value;
```

For example,

```
dynamic d = 20;
```

78. What is the use of Null Coalescing Operator (??) in C#?

The null coalescing operator is used with the nullable value types and reference types. It is used for converting an operand to the type of another nullable (or not) value type operand, where an implicit conversion is possible.

If the value of the first operand is null, then the operator returns the value of the second operand, otherwise it returns the value of the first operand.

79. Distinguish between finally and finalize blocks?

finally block is called after the execution of try and catch blocks, It is used for exception handling whether or not the exception has been caught this block of code gets executed. Generally, this block of code has a cleaner code.

The finalize method is called just before the garbage collection. Main priorities are to perform clean up operation for unmanaged code, it is automatically invoked when an instance is not subsequently called.

80. List down the fundamental OOP concepts? With real time example

There are four fundamental OOP (Object Oriented Programming) concept they are listed as follows:

Inheritance- Ever heard of this dialogue from relatives "you look exactly like your father/mother" the reason behind this is called 'inheritance'. From the Programming aspect, It generally means "inheriting or transfer of characteristics from parent to child class without any modification". The new class is called the derived/child class and the one from which it is derived is called a parent/base class.

Polymorphism- You all must have used GPS for navigating the route, Isn't it amazing how many different routes you come across for the same destination depending on the traffic, from a programming point of view this is called 'polymorphism'. It is one such OOP methodology where one task can be performed in several different ways. To put it in simple words, it is a property of an object which allows it to take multiple forms.

Encapsulation- In a raw form, encapsulation basically means binding up of data in a single class.A class shouldn't be directly accessed but be prefixed in an underscore.

Abstraction- Suppose you booked a movie ticket from bookmyshow using net banking or any other process. You don't know the procedure of how the pin is generated or how the verification is done. This is called 'abstraction' from the programming aspect, it basically means you only show the implementation details of a particular process and hide the details from the user. It is used to simplify complex problems by modeling classes appropriate to the problem.An abstract class cannot be instantiated which simply means you cannot create objects for this type of class. It can only be used for inheriting the functionalities.

Other Definition
===============
C# is an object-oriented programming language that supports 4 OOP concepts.

Encapsulation: defines the binding together code and the data and keeps it safe from any manipulation done by other programs and classes. It is a container that prevents code and data from being accessed by another program that is defined outside the container.

Abstraction: this concept of object-oriented protects everything other than the relevant data about any created object in order to increase efficiency and security within the program.

Inheritance: Inheritance is applied in such a way where one object uses the properties of another object.

Polymorphism: is a feature that allows one interface to act as a base class for other classes. This concept is often expressed as a "single interface but multiple actions".

<u>81. Explain Synchronous and Asynchronous Operations?</u>

Synchronization is a way of creating a thread-safe code where only a single thread will access the code in a given time. A synchronous call waits for completion of method and then continous the program flow. Synchronous programming adversely affects the UI operations that normally happens when user tries to perform time-consuming operations since only one thread is used.

In Asynchronous operation, the method call immediately returns allowing the program to perform other operations while the method called completes its share of work in certain circumstances.

<u>82. Explain Deadlock?</u>

A deadlock is a situation that arises when a process isn't able to complete it's execution because two or more than two processes are waiting for each other to finish. This usually occurs in multi-threading. In this, a shared resource is being held up by a process and another process is waiting for the first process to get over or release it, and the thread holding the locked item is waiting for another process to complete.

For recognizing deadlocks, one should look for threads that get stuck on one of the following:

.Result, .GetAwaiter().GetResult(), WaitAll(), and WaitAny() (When working with Tasks)
Dispatcher.Invoke() (When working in WPF)
Join() (When working with Threads)
lock statements (In all cases)
WaitOne() methods (When working with AutoResetEvent/EventWaitHandle/Mutex/Semaphore)

83. What is Thread Pooling?

A Thread pool is a collection of threads that perform tasks without disturbing the primary thread. Once the task is completed by a thread it returns to the primary thread.

This brings us to the end of this article on C# Interview Questions. I hope it helped in adding up to your knowledge. Wishing you all the best for your interview. Happy learning.

84. What are Regular expressions? Search a string using regular expressions?

Regular expression is a template to match a set of input. The pattern can consist of operators, constructs or character literals. Regex is used for string parsing and replacing the character string.

For Example:

* matches the preceding character zero or more times. So, a*b regex is equivalent to b, ab, aab, aaab and so on.

Searching a string using Regex:

```
static void Main(string[] args)
{
string[] languages = { "C#", "Python", "Java" };
foreach(string s in languages)
{
if(System.Text.RegularExpressions.Regex.IsMatch(s,"Python"))
{
Console.WriteLine("Match found");
}
}
}
```

The above example searches for “Python” against the set of inputs from the languages array. It uses Regex.IsMatch which returns true in case if the pattern is found in the input. The pattern can be any regular expression representing the input that we want to match.

85. Explain Publishers and Subscribers in Events.

Publisher is a class responsible for publishing a message of different types of other classes. The message is nothing but Event as discussed in the above questions.

From the Example in Q #32, Class Patient is the Publisher class. It is generating an Event deathEvent, which is received by the other classes.

Subscribers capture the message of the type that it is interested in. Again, from the Example of Q#32, Class Insurance and Bank are Subscribers. They are interested in event deathEvent of type void.

86. Explain Lock, Monitors, and Mutex Object in Threading.

Lock keyword ensures that only one thread can enter a particular section of the code at any given time. In the above Example, lock(ObjA) means the lock is placed on ObjA until this process releases it, no other thread can access ObjA.

Mutex is also like a lock but it can work across multiple processes at a time. WaitOne() is used to lock and ReleaseMutex() is used to release the lock. But Mutex is slower than lock as it takes time to acquire and release it.

Monitor.Enter and Monitor.Exit implements lock internally. a lock is a shortcut for Monitors. lock(objA) internally calls.

```
Monitor.Enter(ObjA);
try
{
}
Finally {Monitor.Exit(ObjA));}
```

87. What is dictinary class?

88. What is assembly in c#

89. What is difference between dll and exe

90. What is static class and why use it?

91. what is dependency injection and why use dependence injection

92. What is lambda expression in c#

93. What is anonymous method in c#?

94. What is stack and queue collection classes?

95. What is optional parameters in c#?

96. What is partial method in c#?

97. What is the difference between convert.tostring() and .tostring() method?

98. What is inner exception in c#?

99. Can you store different types in an array in c# - kud

100. Why and when should we use an abstract class -

101. What are the advantages of using interfaces