1.What is a Framework?

A framework is a software. Or we can say that a framework is a collection of many small technologies integrated together to develop applications that can be executed anywhere.

2.What does the .NET Framework provide?

.NET Framework provides two things such as

1. **BCL (Base Class Libraries)**
2. **CLR (Common Language Runtime)**

Explain BCL.

1. Base Class Libraries are designed by Microsoft.
2. Without BCL we can't write any code in .NET so BCL also was known as the Building block of Programs of .NET.
3. These are installed into the machine when we installed the .NET framework into the machine.

BCL contains predefined classes and these classes are used for the purpose of application development. The physical location of BCL is **C:\Windows\assembly**

3.What is JIT?

1. JIT stands for Just-in-time.
2. JIT is the component of CLR that is responsible for converting MSIL code into Native code or Machine code.
3. This Native code or Machine code is directly understandable by the operating system.

4.What is metadata?

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

1. Description of assembly.
2. Identity (name, version, culture, public key).
3. The types that are exported
4. Other assemblies that this assembly depends on.
5. Security permissions are needed to run.

5.What is an assembly?

Assemblies are the building block of .NET framework applications; they form the fundamental unit of deployment, version control, reuse, activation scoping and security permissions.

6.What are the differences between managed code and unmanaged code?

This is one of the frequently asked C# Interview Questions and Answers. Let us discuss the difference between them.

Managed code/methods:

Machine instructions are in MSIL format and located in assemblies that will be executed by the CLR and will have the following advantages

1. Memory management to prevent memory leaks in program code.
2. Thread execution

3. Code safety verification
4. Compilation.

Unmanaged code/ methods:
The Unmanaged codes are the instructions that are targeted for specific platforms.
Unmanaged code will exist in any of the formats.

1. COM/COM+ components
2. Win 32 Dlls/system DLLs
3. As these codes are in native formats of OS, these instructions will be executed faster compared with JIT compilation and execution of managed code.

7.What is C#?

C# is an object-oriented typesafe and managed language that is compiled by the .Net framework to generate Microsoft Intermediate Language.

8.What is the difference between an EXE and a DLL?

This is one of the frequently asked C# Interview Questions and Answers. Let us understand the difference between **Exe and DLL**. EXE is an executable file and can run by itself as an application whereas DLL is usually consumed by an EXE or by another DLL and we cannot run or execute DLL directly.
For example in .NET compiling a Console Application or a Windows Application generates EXE, whereas compiling a Class Library Project or an ASP.NET web application generates DLL. In the .NET framework, both EXE and DLL are called assemblies.
A DLL can be reused in the application whereas an exe file can never be reused in an application. EXE stands for executable, and DLL stands for Dynamic Link Library

9.What's the difference between IEnumerable<T> and List<T>?

1. **IEnumerable** is an interface, whereas **List** is one specific implementation of IEnumerable. A list is a class.
2. FOR-EACH loop is the only possible way to iterate through a collection of **IEnumerable** whereas **List** can be iterated using several ways. The list can also be indexed by an int index. The element of a list collection can be added to and removed from and have items inserted at a particular index but these are not possible with a collection of type IEnumerable.
3. **IEnumerable** doesn't allow random access, whereas **List** does allow random access using the integral index.
4. In general, from a performance standpoint, iterating through **IEnumerable** is much faster than iterating through a **List**.

10.What are the new features introduced in C# 7?

This is a very commonly asked C# interview question. This question is basically asked to check if you are passionate about catching up with the latest technological advancements. The list below shows the new features that are introduced in C# 7. Let's have a look at the new features that are introduced as part of C# 7

1. Out variables
2. Pattern matching
3. Digit Separators
4. Tuples

5. Deconstruction (Splitting Tuples)
6. Local functions
7. Literal improvements
8. Ref returns and locals
9. Generalized async return types
10. More expression-bodied members
11. Throw expressions
12. Discards
13. Async main
14. Default literal expressions
15. Inferred tuple element names

11.Why should you override the ToString() method?

This C# Interview Question is one of the most frequently asked .NET questions. All types in .Net inherit from the **System.Object** class directly or indirectly. Because of this inheritance, every type in .Net inherits the ToString() method from System.Object class. To understand this better, please have a look at the example.

In the above example **Number.ToString()** method will correctly give the string representation of int 10 when we call the ToString() method. If we have any user-defined class like the Customer class as shown in the below example and when we call the ToString() method the output does not make any sense i.e. in the output you simply get the class name.

```
public class Customer
{
public string FirstName;
public string LastName;
}
public class MainClass
{
public static void Main()
{
Customer C = new Customer();
```

```
C.FirstName = "David";
C.LastName = "Boon";
Console.WriteLine(C.ToString());
}
}
```

But what if we want to print the first name and last name of the customer when we call the toString method on the customer object. Then we need to override the ToString() method, which is inherited from the **System.Object** class. The code sample below shows how to override the ToString() method in a class, that would give the output that we want.

```
public class Customer
{
public string FirstName;
public string LastName;
public override string ToString()
{
return LastName + ", " + FirstName;
}
}
public class MainClass
{
public static void Main()
{
Customer C = new Customer();
C.FirstName = "David";
C.LastName = "Boon";
Console.WriteLine(C.ToString());
}
}
```

12.What do you mean by String objects are immutable?

This C# Interview question is frequently asked in .NET Interviews. String objects are immutable means they cannot be changed once they are created. All of the String methods and C# operators that appear to modify a string actually return the results in a new string object. In the following example, when the contents of s1 and s2 are concatenated to form a single string, the two original strings are unmodified. The += operator creates a new string that contains the combined contents. That new object is assigned to the variable s1, and the original object that was assigned to s1 is released for garbage collection because no other variable holds a reference to it.

```
string s1 = "First String ";
string s2 = "Second String";
// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;
System.Console.WriteLine(s1);
// Output: First String Second String
```

What will be the output of the following code?

```
string str1 = "Hello ";
string str2 = str1;
str1 = str1 + "C#";
System.Console.WriteLine(str2);
```

The output of the above code is “Hello” and not “Hello C#”. This is because, if you create a reference to a string, and then “modify” the original string, the reference will continue to point to the original object instead of the new object that was created when the string was modified.

13.What are the differences between value types and reference types?

This is one of the frequently asked C# Interview Questions and Answers. Value types are stored on the stack whereas reference types are stored on the managed heap. The Value type variables directly contain their values whereas reference variables hold only a reference to the location of the object that is created on the managed heap.

There is no heap allocation or garbage collection overhead for value-type variables. As reference types are stored on the managed heap, they have the overhead of object allocation and garbage collection.

Value Types cannot inherit from another class or struct. Value types can only inherit from interfaces. Reference types can inherit from another class or interface.

My understanding is that just because structs inherit from System.ValueType, cannot inherit from another class, because we cannot do multiple-class inheritance.

Structs can inherit from System.ValueType class but cannot be inherited by any other types like Structs or Class. In other words, Structs are like Sealed classes that cannot be inherited.

14.What is the difference between int.Parse and int.TryParse methods?

This is one of the frequently asked C# Interview Questions and Answers. The parse method throws an exception if the string you are trying to parse is not a valid number whereas TryParse returns false and does not throw an exception if parsing fails. Hence TryParse is more efficient than Parse.

15.What are Boxing and Unboxing?

Boxing – Converting a value type to a reference type is called boxing. An example is shown below.

```
int i = 101;
object obj = (object)i; // Boxing
```

Unboxing – Converting a reference type to a value type is called unboxing. An example is shown below.

```
obj = 101;
i = (int)obj; // Unboxing
```

16.What are Access Modifiers in C#?

This is one of the frequently asked C# Interview Questions and Answers. In C# there are 5 different types of Access Modifiers.

1. **Public:** The public type or member can be accessed by any other code in the same assembly or another assembly that references it.
2. **Private:** The type or member can only be accessed by code in the same class or struct.
3. **Protected:** The type or member can only be accessed by code in the same class or struct, or in a derived class.
4. **Internal:** The type or member can be accessed by any code in the same assembly, but not from another assembly.
5. **Protected Internal:** The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.

17.Difference between int and Int32 in C#

This is one of the frequently asked C# Interview Questions and Answers. **Int32** and **int** are **synonymous**, both of them allow us to create a 32-bit integer. int is shorthand notation (alias) for Int32. When declaring an integer in a c# program most of us prefer using int over Int32. Whether we use **int** or **Int32** to create an integer, the behavior is identical.

I think the only place where **Int32** is not allowed is when creating an enum. The following code will raise a compiler error stating – **Type byte, sbyte, short, ushort, int, uint, long, or ulong expected.**

```
enum Test : Int32
{
abc = 1
}
```

The following code will compile just fine

What are the 2 types of data types available in C#?

1. Value Types
2. Reference Types

18.What is Virtual Property in C#? Give an example.

This is one of the most frequently asked C#.NET Interview Questions. A property that is marked with a virtual keyword is considered virtual property. Virtual properties enable derived classes to override the property behavior by using the override keyword. In the example below FullName is a virtual property in the Customer class. The BankCustomer class inherits from the Customer class and overrides the FullName virtual property. In the output, you can see the overridden implementation. A property overriding a virtual property can also be sealed, specifying that for derived classes it is no longer virtual.

```
class Customer
{
private string _firstName = string.Empty;
private string _lastName = string.Empty;
public string FirstName
{
get
{
return _firstName;
}
set
{
_firstName = value;
}
}
public string LastName
{
get
```

```
{
return _lastName;
}
set
{
_lastName = value;
}
}
// FullName is virtual
public virtual string FullName
{
get
{
return _lastName + ", " + _firstName;
}
}
}
class BankCustomer : Customer
{
// Overriding the FullName virtual property derived from customer class
public override string FullName
{
get
{
return "Mr. " + FirstName + " " + LastName;
}
}
}
class MainClass
{
public static void Main()
{
BankCustomer BankCustomerObject = new BankCustomer();
BankCustomerObject.FirstName = "David";
BankCustomerObject.LastName = "Boon";
Console.WriteLine("Customer Full Name is : " + BankCustomerObject.FullName);
```

```
}
}
```

19.Types of Variables in a Class in C#:

Now, let us understand the different kinds of variables a class can have and their behavior. Basically, there are four types of variables that we can declare inside a class in C#. They are as follows:

1. **Non-Static/Instance Variable**
2. **Static Variable**
3. **Constant Variable**
4. **Readonly Variable**

The behavior of all these different variables is going to vary. Let us understand each of these variables in C#.

20.Constant Variables in C#:

In C#, if we declare a variable by using the const keyword, then it is a constant variable and the value of the constant variable can't be modified once after its declaration. So, it is mandatory to initialize the constant variable at the time of its declaration only. Suppose, you want to declare a constant PI in your program, then you can declare the constant variable as follows:

const float PI = 3.14f;

```
using System;
namespace ConstDemo
{
    class Program
    {
        //we need to assign a value to the const variable
        //at the time of const variable declaration else it will
        //give compile time error
        const float PI = 3.14f; //Constant Variable
        static void Main(string[] args)
        {
            //Const variables are static in nature
            //so we can access them by using class name
            Console.WriteLine(Program.PI);
            //We can also access them directly within the same class
            Console.WriteLine(PI);

            //We can also declare a constant variable within a function
            const int Number = 10;
            Console.WriteLine(Number);

            //Once after declaration we cannot change the value
            //of a constant variable. So, the below line gives an error
```

```
            //Number = 20;

            Console.ReadLine();
        }
    }
}
```

Points to Remember while Working with Const Variable in C#:

1. The keyword **const** is used to create a "**constant**" variable. It means it will create a variable whose value is never going to be changed. In simple words, we can say that the variable whose value cannot be changed or modified once after its declaration is known as a constant variable.
2. Constants are static by default.
3. It is mandatory to initialize a constant variable at the time of its declaration.
4. The behavior of a constant variable is the same as the behavior of a static variable i.e. maintains only one copy in the life cycle of the class and initializes immediately once the execution of the class start (object not required)
5. The only difference between a static and constant variable is that the static variables can be modified whereas the constant variables in C# can't be modified once it is declared.

Read-Only Variables in C#

When we declare a variable by using the readonly keyword, then it is known as a read-only variable and these variables can't be modified like constants but after initialization. That means it is not mandatory to initialize a read-only variable at the time of its declaration, they can also be initialized under the constructor. That means we can modify the read-only variable value only within a constructor.

The behavior of read-only variables will be similar to the behavior of non-static variables in C#, i.e. initialized only after creating the instance of the class and once for each instance of the class is created. So, we can consider it as a non-static variable, and to access readonly variables we need an instance.

```
using System;
namespace TypesOfVariables
{
    internal class Program
    {
        readonly int x; //Readonly Variable
        static void Main(string[] args)
        {
            Program obj1 = new Program();
            //Accessing Readonly variable using instance
            Console.WriteLine($"{obj1.x}");
            Console.Read();
        }
    }
}
```

Points to Remember while working with Read-Only Variable in C#:

1. The variable which is created by using the **readonly** keyword is known as a read-only variable in C#. The read-only variable's value cannot be modified once after its initialization.
2. It is not mandatory or required to initialize the read-only variable at the time of its declaration like a constant. You can initialize the read-only variables under a constructor but the most important point is that once after initialization, you cannot modify the value of the readonly variable outside the constructor.
3. The behavior of a read-only variable is similar to the behavior of a non-static variable. That is, it maintains a separate copy for each object. The only difference between these two is that the value of the non-static variable can be modified from outside the constructor while the value of the read-only variable cannot be modified from outside the constructor body.

21.WHAT IS THE DIFFERENCE BETWEEN "STRING" AND "STRINGBUILDER"? WHEN TO USE WHAT?

String is IMMUTABLE in C# - It means if you defined one string then you couldn't modify it. Every time you will assign some value to it, it will create a new string.
StringBuilder is MUTABLE - This means that if any manipulation will be done on string, then it will not create a new instance every time.
If you want to change a string multiple times, then StringBuilder is a better option from **performance** point of view because it will not require new memory every time.

Example to Proves C# strings are Immutable:

Let us see an example to understand C# strings are Immutable. Please copy and paste the following code. As you can see here we have a heavy loop. As part of the Loop, we assign a value to the string str variable. Here, we are using GUID to generate a new value, and each time it will create a new value and assign it to the str variable. Again, we are using Stopwatch to check how much time it took to execute the loop.

```
using System;
using System.Diagnostics;

namespace StringDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            string str = "";
            Console.WriteLine("Loop Started");
            var stopwatch = new Stopwatch();
```

```
            stopwatch.Start();
            for (int i = 0; i < 30000000; i++)
            {
                 str = Guid.NewGuid().ToString();
            }
            stopwatch.Stop();

            Console.WriteLine("Loop Ended");
            Console.WriteLine("Loop Exceution Time in MS :" +
stopwatch.ElapsedMilliseconds);

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

Example using StringBuilder in C#:

Let us understand how to overcome the **String Concatenation Problem in C#** using the **StringBuilder** class. In the following example, we are using the StringBuilder class to concatenate strings. Here, first, we create an instance of the StringBuilder class and then use the **Append** method of the **StringBuilder** class to concatenate the string.

```
using System;
using System.Diagnostics;
using System.Text;

namespace StringDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            StringBuilder stringBuilder = new StringBuilder();
            Console.WriteLine("Loop Started");
            var stopwatch = new Stopwatch();

            stopwatch.Start();
            for (int i = 0; i < 30000; i++)
            {
                stringBuilder.Append("DotNet Tutorials");
            }
            stopwatch.Stop();

            Console.WriteLine("Loop Ended");
            Console.WriteLine("Loop Exceution Time in MS :" +
stopwatch.ElapsedMilliseconds);

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

22.What are the 4 pillars of any object-oriented programming language?

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

Do structs support inheritance?

No, structs do not support **inheritance**, but they can implement interfaces.

What is the main advantage of using inheritance?

Code reuse

Is the following code legal?

```
class ChildClass : ParentClassA, ParentClassB
{

}
```

No, a child class can have only one base class. We cannot specify 2 base classes at the same time. C# supports single class inheritance only. Therefore, we can specify only one base class to inherit from. However, it does allow **multiple interface inheritance**.

Does C# support multiple class inheritance?

No, C# supports single class inheritance only. However, classes can implement **multiple interfaces** at the same time.

23.What is the difference between interfaces and abstract classes?

There are several differences between an abstract class and an **interface** as listed below.

1. Abstract classes can have implementations for some of their members, but the interface can't have the implementation for any of its members.
2. Interfaces cannot have fields where an abstract class can have fields.
3. An interface can inherit from another **interface** only and cannot inherit from an abstract class whereas an abstract class can inherit from another abstract class or another interface.
4. A class can inherit from **multiple interfaces** at the same time, whereas a class cannot inherit from multiple abstract classes at the same time.
5. Abstract class members can have access modifiers where as interface members cannot have access modifiers as they are by default public.

24.What are the advantages of using interfaces?

This is the **most commonly asked interview question**. This interview question is being asked in almost all the dot net interviews. It is very important that we understand all the concepts of interfaces and abstract classes. Interfaces are very powerful. If properly used, **interfaces provide all the advantages** as listed below.

1. Interfaces allow us to implement polymorphic behavior. Of course, abstract classes can also be used to implement polymorphic behavior.
2. The Interfaces allow us to develop very loosely coupled systems.
3. Interfaces enable mocking for better unit testing.
4. The Interfaces enable us to implement multiple **inheritances in C#**.
5. Interfaces are great for implementing Inversion of Control or Dependency Injection.
6. The Interfaces enable parallel application development.

Can an Interface contain fields?

No, an Interface cannot contain fields

Can you create an instance of an interface?

No, we cannot create an instance of an interface.

If a class inherits an interface, what are the 2 options available for that class?

Option1: Provide Implementation for all the members, inherited from the interface.
Option2: If the class does not wish to provide Implementation for all the members inherited from the interface, then the class has to be marked as abstract.

When to use Interface?

If your child classes should implement a certain group of methods/functionalities but each of the child classes is free to provide its own implementation then use interfaces.

When to use Abstract Classes in C#?

When we have a requirement where our base class should provide the default implementation of certain methods whereas other methods should be open to being overridden by child classes use abstract classes.

25 .What is an Abstract Class in C#?

A class that is declared by using the keyword abstract is called an abstract class. An abstract class is a partially implemented class used for developing some of the operations which are common for all next level subclasses. So it contains both abstract methods, concrete methods including variables, properties, and indexers.
It is always created as a superclass next to the interface in the object inheritance hierarchy for implementing common operations from the interface.
An abstract class may or may not have abstract methods. But if a class contains an abstract method then it must be declared as abstract. The abstract class cannot be instantiated directly. It's compulsory to create/derive a new class from an abstract class in order to provide the functionality to its abstract functions.

Can you create an instance of an abstract class?

No, abstract classes are incomplete and we cannot create an instance of an abstract class.

What is a Sealed Class?

A sealed class is a class that cannot be inherited from. That means if we have a class called Customer that is marked as sealed. No other class can inherit from the Customer class.

What is the abstract method?

A method that does not have the body is called an abstract method. It is declared with the modifier abstract. It contains only the Declaration/signature and does not contain the implementation/ body of the method. An abstract function should be terminated with a semicolon. Overriding of an abstract function is compulsory.

When to use the abstract method?

Abstract methods are usually declared where two or more subclasses are expected to fulfill a similar role in different ways.

Can a sealed class be used as a base class?

No, the sealed class cannot be used as a base class. A compile-time error will be generated.

What type of members can we define in an abstract class?

We can define all static and non-static members including properties, fields, indexes, and also abstract methods.

Will abstract class members are created when a subclass object is created?

Yes, its non-static members get memory when its concrete sub-class object is created.

How can we execute static and non-static concrete members of the abstract class?

Static members can be executed directly from its main method and its non-static members are executed by using its concrete sub-class object.

Can we declare the abstract method as static?

No, we are not allowed to declare the abstract method as static. It leads to CE: the illegal combination of modifier abstract and static. If the compiler allows us to declare it as static, it can be invoked directly which cannot be executed by CLR at runtime. Hence to restrict in calling abstract methods compiler does not allow us to declare the abstract method as static.

Can we declare the concrete class as abstract?

Yes, it is allowed. Defining a class as abstract is a way of preventing someone from instantiating a class that is supposed to be extended first. To ensure our class non-static members are only accessible via sub-class object we should declare the concrete class as abstract.

26. What is compile-time Polymorphism in C#?

This is one of the frequently asked **C# Polymorphism interview** questions. In the case of compile-time polymorphism, the object of the class recognizes which method to be executed for a particular method call at the time of program compilation and binds the method call with method definition.

This happens in the case of overloading because in the case of overloading each method will have a different signature and basing on the method call we can easily recognize the method which matches the method signature. It is also called static **polymorphism** or early binding. Static polymorphism is achieved by using function overloading and operator overloading

27 .What is Runtime Polymorphism in C#?

This is also one of the frequently asked Polymorphism interview questions in C#. In the case of runtime **polymorphism** for a given method call, we can recognize which method has to be executed exactly at runtime but not in compilation time because in the case of

overriding and hiding we have multiple methods with the same signature. So which method to be given preference and executed that is identified at runtime and binds the method call with its suitable method. It is also called dynamic polymorphism or late binding. Dynamic polymorphism is achieved by using function overriding.

28. What is the difference between Method Overriding and Method Hiding?

This is one of the frequently asked **Polymorphism** interview questions in C#. A parent class method can be redefined under its child class using two different approaches.

1. **Method Overriding.**
2. **Method Hiding.**

In **Method overriding**, the parent class gives permission for its child class to override the method by declaring it as **virtual**. Now the child class can override the method using the **Override** keyword as it got permission from the parent. The parent class methods can be redefined under child classes even if they were not declared as **Virtual** by using the **'new'** keyword.

In **method overriding** a base class reference variable pointing to a child class object will invoke the overridden method in the child class. In method hiding a base class reference variable pointing to a child class object will invoke the hidden method in the base class.

For hiding the base class method from the derived class simply declare the derived class method with the new keyword. Whereas in C#, for overriding the base class method in a derived class, we need to declare the base class method as virtual and the derived class method as the override.

If a method is simply hidden then the implementation to call is based on the compile-time type of the argument "this". Whereas if a method is overridden then the implementation to be called is based on the run-time type of the argument "this". New is reference-type specific, overriding is object-type specific.

29 .What is the difference between a virtual method and an abstract method?

This is one of the frequently asked C# Polymorphism interview questions. A virtual method must have a body whereas an abstract method should not have a body. A Base class virtual method may or may not be overridden in the Derived class whereas a Base class Abstract method has to be implemented by the derived class.

30 .What is a Constructor in C#?

Constructors are the special types of methods of a class that get executed when its object is created. The Constructors in C# are responsible for object initialization and memory allocation of its class and the new keyword role is creating the object.

Points to Remember.

1. The constructor name should be the same as the class name.
2. It should not contain return type even void also.
3. It should not contain modifiers.
4. In its logic return statement with value is not allowed.

Note:

1. It can have all five accessibility modifiers.
2. The Constructor can have parameters.
3. It can have throws clause it means we can throw an exception from a constructor.

4. The Constructor can have logic, as part of logic it can have all C#.NET legal statements except return statement with value.
5. We can place the return; in a constructor.

Can we define a method with the same class name in C#?

No, it is not allowed to define a method with the same class name. It will give a compile-time error.

How many types of constructors are there in C#.net?

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Static Constructor
5. Private Constructor

31. Explain the static constructor in C#?

This is one of the frequently asked Constructor Interview Questions in C#.
We can create a constructor as static and when a constructor is created as static, it will be invoked only once. There is no matter how many numbers of instances (objects) of the class are created but it is going too invoked only once and that is during the creation of the first instance (object) of the class.
The static constructor is used to initialize static fields of the class and we can also write some code inside the static constructor that needs to be executed only once.
Static data fields are created only once in a class even though we are created any number of objects.

1. There can be only one static constructor in a class.
2. The static constructor should be without any parameter.
3. It can only access the static members of the class.
4. There should not be any access modifier in the static constructor definition.
5. If a class is static then we cannot create the object for the static class.
6. Static constructor will be invoked only once i.e. at the time of first object creation of the class, from 2nd object creation onwards static constructor will not be called.

Can we initialize non-static data fields within the static constructor?

It is not possible to initialize non-static data fields within the static constructor, it raises a compilation error.

Can we initialize static data fields within the non-static constructor?

Yes, we can initialize static data fields within a non-static constructor but after then they lose their static nature.

Can we initialize static data fields in both static and non-static constructor?

Yes, we can initialize static data fields in both static and non-static constructors but static data fields lose their static nature.

32. Explain Private constructor in C#?

This is one of the frequently asked Constructor Interview Questions in C#. We can also create a constructor as private. The constructor whose accessibility is private is known as the private constructor. When a class contains a private constructor then we cannot create an object for the class outside of the class. Private constructors are used to creating an

object for the class within the same class. Generally, private constructors are used in the Remoting concept.

When is a destructor method called?

A destructor method gets called when the object of the class is destroyed.

33. When to use a Private constructor in c#?

This is one of the frequently asked Constructor Interview Questions in C#.
There are several reasons for using private constructors

1. When we want the caller of the class only to use the class but not instantiate.
2. If you want to ensure a class can have only one instance at a given time, i.e. private constructors are used in implementing Singleton() design pattern.
3. When a class has several overloads of the constructor, and some of them should only be used by the other constructors and not external code.

Can you mark a static constructor with access modifiers?

No, we cannot use access modifiers on the static constructor.

Can you have parameters for static constructors?

No, static constructors cannot have parameters.

What happens if a static constructor throws an exception?

If a static constructor throws an exception, the runtime will not invoke it a second time, and the type will remain uninitialized for the lifetime of the application domain in which your program is running.

Give 2 scenarios where static constructors can be used?

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. Static constructors are also useful when creating wrapper classes for unmanaged code when the constructor can call the LoadLibrary method.

What is Destructor?

A Destructor has the same name as the class with a tilde character and is used to destroy an instance of a class.

Can a class have more than 1 destructor?

No, a class can have only 1 destructor.

Can structs in C# have destructors?

No, structs can have constructors but not destructors, only classes can have destructors.

Can you pass parameters to destructors?

No, you cannot pass parameters to destructors. Hence, you cannot overload destructors.

Can you explicitly call a destructor?

No, you cannot explicitly call a destructor. Destructors are invoked automatically by the garbage collector.

Why is it not a good idea to use Empty destructors?

When a class contains a destructor, an entry is created in the Finalize queue. When the destructor is called, the garbage collector is invoked to process the queue. If the destructor is empty, this just causes a needless loss of performance.

34 .What is a Delegate in C#? Explain with one example.

We can call a method that is defined in a class in two ways

Using Object: We can call the method using the object of the class if it is a non-static method or we can call the method through class name if it is a static method.

Using Class Name: We can call a method by using a delegate also. Calling a method using delegate will be faster in execution compared to the first process.

A delegate is also a user-defined type and before invoking a method using delegate we must have to define that delegate first. A delegate is a type-safe function pointer that means a delegate holds the reference of a method and then calls the method for execution.

The signature of the delegate must match with the signature of the function, the delegate points to otherwise we will get a compiler error. This is the reason delegates are called type-safe function pointers.

A Delegate is similar to a class. We can create an instance of it and when we do so, we pass the function name as a parameter to the delegate constructor, and it is the function name that the delegate points to.

Types of Delegates in C#:

Delegates are classified into two types such as

1. Single cast delegate
2. Multicast delegate

If a delegate is used for invoking a single method then it is called a single cast delegate or unicast delegate. OR the delegates that represent only a single function is known as a single cast delegate.

If a delegate is used for invoking multiple methods then it is known as the multicast delegate. OR the delegates that represent more than one function are called Multicast delegate.

Note: If we want to call multiple methods using a single delegate the I/O parameters of all those methods must be the same.

Tip to remember delegate syntax:

Delegates syntax look very much similar to a method with a delegate keyword.

Sample Delegate Program:

```
namespace Demo
{
    // Delegate Declaration.
    public delegate void HelloFunctionDelegate(string Message);
    class Pragim
    {
        public static void Main()
        {
            // Create an instance of the delegate and pass the function name as
            // the parameter to the constructor. The passed function signature
            // must match the signature of the delegate
            HelloFunctionDelegate del = new HelloFunctionDelegate(Pragim.Hello);

            // Invoke the delegate, which will invoke the method
            del("Hello from Delegte");
            Console.ReadKey();
        }
        public static void Hello(string strMessge)
        {
            Console.WriteLine(strMessge);
        }
    }
}
```

35.What is Multicast Delegate in C#? Explain with one example.

A Multicast delegate is a delegate that has references to more than one function. When we invoke a multicast delegate, all the functions are invoked that the delegate is pointing to. There are 2 approaches to create a multicast delegate.

Approach1:

```
namespace Sample

{

public delegate void SampleDelegate();

public class Sample

{

static void Main()

{

SampleDelegate del1 = new SampleDelegate(SampleMethodOne);

SampleDelegate del2 = new SampleDelegate(SampleMethodTwo);

SampleDelegate del3 = new SampleDelegate(SampleMethodThree);

// In this example del4 is a multicast delegate. We use +(plus)

// operator to chain delegates together and -(minus) operator to remove.
```

```
SampleDelegate del4 = del1 + del2 + del3 - del2;
del4();
}
public static void SampleMethodOne()
{
Console.WriteLine("SampleMethodOne Invoked");
}
public static void SampleMethodTwo()
{
Console.WriteLine("SampleMethodTwo Invoked");
}
public static void SampleMethodThree()
{
Console.WriteLine("SampleMethodThree Invoked");
}
}
}
```

Approach2:

```
namespace Sample
{
public delegate void SampleDelegate();
public class Sample
{
static void Main()
{
// In this example del is a multicast delegate. You use += operator
```

```
// to chain delegates together and -= operator to remove.
SampleDelegate del = new SampleDelegate(SampleMethodOne);
del += SampleMethodTwo;
del += SampleMethodThree;
del -= SampleMethodTwo;
del();
}
public static void SampleMethodOne()
{
Console.WriteLine("SampleMethodOne Invoked");
}
public static void SampleMethodTwo()
{
Console.WriteLine("SampleMethodTwo Invoked");
}
public static void SampleMethodThree()
{
Console.WriteLine("SampleMethodThree Invoked");
}
}
}
```

Note*:* A multicast delegate invokes the methods in the invocation list, in the same order in which they are added.
If the delegate has a return type other than void and if the delegate is a multicast delegate, only the value of the last invoked method will be returned. Along the same lines, if the delegate has an out parameter, the value of the output parameter will be the value assigned by the last method.

36 .Where do you use multicast delegates?

Multicast delegate makes the implementation of the observer design pattern very simple. The observer pattern is also called a publish/subscribe pattern.

37 .Where did you use delegates in your project? Or how did you use delegates in your project?

The Delegate is one of the very important aspects to understand. Most of the interviewers ask you to explain the usage of delegates in a real-time project that you have worked on. Delegates are extensively used by framework developers. Let us say we have a class called Employee as shown below.

Employee Class

```
public class Employee

{

public int ID { get; set; }

public string Name { get; set; }

public int Experience { get; set; }

public int Salary { get; set; }

}
```

The **Employee** class has the following properties.

1. **Id**
2. **Name**
3. **Experience**
4. **Salary**

Now I want to write a method in the Employee class which can be used to promote employees. The method should take a list of Employee objects as a parameter and should print the names of all the employees who are eligible for a promotion. But the logic based on which the employee gets promoted should not be hardcoded. At times we may promote employees based on their experience and at times we may promote them based on their salary or maybe some other condition. So, the logic to promote employees should not be hard-coded within the method.

How to achieve?

To achieve this we can make use of delegates. So now I would design my class as shown below. We also created a delegate EligibleToPromotion. This delegate takes the Employee object as a parameter and returns a boolean. In the Employee class, we have the PromoteEmpoloyee method. This method takes a list of Employees and a Delegate of the type EligibleToPromotion as parameters. The method then loops through each employee object and passes it to the delegate. If the delegate returns true, then the Employee is promoted, else not promoted. So within the method, we have not hardcoded any logic on how we want to promote employees.

```
namespace DelegateDemo

{

public delegate bool EligibleToPromotion(Employee EmployeeToPromotion);
```

```
public class Employee
{
public int ID { get; set; }
public string Name { get; set; }
public int Experience { get; set; }
public int Salary { get; set; }
public static void PromoteEmployee(List<Employee> lstEmployees,
EligibleToPromotion IsEmployeeEligible)
{
foreach (Employee employee in lstEmployees)
{
if (IsEmployeeEligible(employee))
{
Console.WriteLine("Employee {0} Promoted", employee.Name);
}
}
}
}
```

So now the client who uses the Employee class has the flexibility of determining the logic on how they want to promote their employees as shown below. First create the employee objects – E1, E2, and E3. Populate the properties for the respective objects. We then create an employeeList to hold all the 3 employees.

Notice the Promote method that we have created. This method has the logic of how we want to promote our employees. The method is then passed as a parameter to the delegate. Also, note this method has the same signature as that of the EligibleToPromotion delegate. This is very important because the Promote method cannot be passed as a parameter to the delegate if the signature differs. This is the reason why delegates are called type-safe function pointers.

```
namespace DelegateDemo
{
```

```
public class Employee
{
public int ID { get; set; }
public string Name { get; set; }
public int Experience { get; set; }
public int Salary { get; set; }
public static void PromoteEmployee(List<Employee> lstEmployees,
EligibleToPromotion IsEmployeeEligible)
{
foreach (Employee employee in lstEmployees)
{
if (IsEmployeeEligible(employee))
{
Console.WriteLine("Employee {0} Promoted", employee.Name);
}
}
}
}
class Program
{
static void Main()
{
Employee emp1 = new Employee()
{
ID = 101,
Name = "Pranaya",
```

```
Experience = 5,
Salary = 10000
};
Employee emp2 = new Employee()
{
ID = 102,
Name = "Kumar",
Experience = 10,
Salary = 20000
};
Employee emp3 = new Employee()
{
ID = 103,
Name = "Rout",
Experience = 20,
Salary = 30000
};
List<Employee> lstEmployess = new List<Employee>();
lstEmployess.Add(emp1);
lstEmployess.Add(emp2);
lstEmployess.Add(emp3);
EligibleToPromotion eligibleTopromote = new
EligibleToPromotion(Program.Promote);
Employee.PromoteEmployee(lstEmployess, eligibleTopromote);
Console.ReadKey();
}
```

```
public static bool Promote(Employee employee)
{
if (employee.Salary > 10000)
{
return true;
}
else
{
return false;
}
}
}
```

So if we did not have the concept of delegates it would not have been possible to pass a function as a parameter. As the Promote method in the Employee class makes use of delegate, it is possible to dynamically decide the logic on how we want to promote employees.

Using Lambda expressions

In C Sharp 3.0 Lambda expressions are introduced. So you can make use of lambda expressions instead of creating a function and then an instance of a delegate and then passing the function as a parameter to the delegate. The sample example rewritten using the Lambda expression is shown below. The private Promote method is no longer required now.

```
class Program
{
static void Main()
{
Employee emp1 = new Employee()
{
ID = 101,
```

```
Name = "Pranaya",
Experience = 5,
Salary = 10000
};
Employee emp2 = new Employee()
{
ID = 102,
Name = "Kumar",
Experience = 10,
Salary = 20000
};
Employee emp3 = new Employee()
{
ID = 103,
Name = "Rout",
Experience = 20,
Salary = 30000
};
List<Employee> lstEmployess = new List<Employee>();
lstEmployess.Add(emp1);
lstEmployess.Add(emp2);
lstEmployess.Add(emp3);
Employee.PromoteEmployee(lstEmployess, x => x.Experience > 5);
}
}
```

38. What is the difference between Process and Thread?

This is one of the most frequently asked Multithreading Interview Questions in C#. A process is started when you start an Application. The process is a collection of resources like virtual address space, code, security contexts, etc. A process can start multiple threads. Every process starts with a single thread called the primary thread. You can create n number of threads in a process. Threads share the resources allocated to the process. A process is the parent and threads are his children.

Why do we need Multi-threading in our project?

This is one of the most frequently asked Multithreading Interview Questions in C#.NET. Let us discuss this question. Multi-threading is used to run multiple threads simultaneously. Some main advantages are:

1. You can do multiple tasks simultaneously. For e.g. saving the details of the user to a file while at the same time retrieving something from a web service.
2. Threads are much more lightweight than processes. They don't get their own resources. They used the resources allocated to a process.
3. Context-switch between threads takes less time than process.

39 .What are the advantages and disadvantages of multithreading?

I think this MultiThreading Interview Question is the most asked interview question in the dot net. So let us discuss the advantages and disadvantages

Advantages of multithreading:

1. To maintain a responsive user interface
2. It makes efficient use of processor time while waiting for I/O operations to complete.
3. To split large, CPU-bound tasks to be processed simultaneously on a machine that has multiple CPUs/cores.

Disadvantages of multithreading:

1. On a single-core/processor machine threading can affect performance negatively as there is overhead involved with context-switching.
2. Have to write more lines of code to accomplish the same task.
3. Multithreaded applications are difficult to write, understand, debug, and maintain.

Please Note: Only use multithreading when the advantages of doing so outweigh the disadvantages.

40.How to pass data to the thread function in a type-safe manner?

This MultiThreading Interview Question in C# can be asked in almost all interviews. To pass data to the Thread function in a type-safe manner, encapsulate the thread function and the data it needs in a helper class and use the ThreadStart delegate to execute the thread function. An example is shown below.

```
namespace ThreadingExample
{
```

```
class Program
{
public static void Main()
{
// Prompt the user for the target number
Console.WriteLine("Please enter the target number");
// Read from the console and store it in target variable
int target = Convert.ToInt32(Console.ReadLine());
// Create an instance of the Number class, passing it
// the target number that was read from the console
Number number = new Number(target);
// Specify the Thread function
Thread T1 = new Thread(new ThreadStart(number.PrintNumbers));
// Alternatively we can just use Thread class constructor as shown below
// Thread T1 = new Thread(number.PrintNumbers);
T1.Start();
}
}
// Number class also contains the data it needs to print the numbers
class Number
{
int _target;
// When an instance is created, the target number needs to be specified
public Number(int target)
{
// The targer number is then stored in the class private variable _target
this._target = target;
}
// Function prints the numbers from 1 to the traget number that the user provided
public void PrintNumbers()
{
for (int i = 1; i <= _target; i++)
{
Console.WriteLine(i);
}
}
```

```
}
}
```

41 .What is the difference between Threads and Tasks?

This is one of the most frequently asked Multithreading Interview Questions in C#. Let us understand the differences between them.

1. Tasks are the wrapper around Thread and ThreadPool classes. Below are some major differences between Threads and Tasks:
2. A Task can return a result but there is no proper way to return a result from Thread.
3. We can apply chaining to multiple tasks but we cannot in threads.
4. We can wait for Tasks without using Signalling. But in Threads, we have to use event signals like AutoResetEvent and ManualResetEvent.
5. We can apply the Parent/Child relationship in Tasks. A Task at one time becomes a parent of multiple tasks. Parent Tasks does not completed until their child's tasks are completed. We do not have any such mechanism in the Thread class.
6. Child Tasks can propagate their exceptions to the parent Task and All child exceptions are available in the AggregateException class.
7. Task has an in-build cancellation mechanism using the CancellationToken class.

42.What is the Significance of Thread.Join and Thread.IsAlive functions in multithreading?

This is one of the most frequently asked Multithreading Interview Questions in C#. Join blocks the current thread and makes it wait until the thread on which the Join method is invoked completes. The join method also has an overload where we can specify the timeout. If we don't specify the timeout the calling thread waits indefinitely, until the thread on which Join() is invoked completes. This overloaded Join (int millisecondsTimeout) method returns a boolean true if the thread has terminated otherwise false. Join is particularly useful when we need to wait and collect results from a thread execution or if we need to do some cleanup after the thread has been completed.

The IsAlive returns boolean True if the thread is still executing otherwise false.

Program code used in the demo:

```
namespace MultithreadingDemo

{

class Program

{

public static void Main()

{
```

```
Console.WriteLine("Main Thread Started" + Thread.CurrentThread.Name);
Thread T1 = new Thread(Program.Thread1Function);
T1.Start();
Thread T2 = new Thread(Program.Thread2Function);
T2.Start();
//if (T1.Join(1000))
//{
// Console.WriteLine("Thread1Function completed");
//}
//else
//{
// Console.WriteLine("Thread1Function hot not completed in 1 second");
//}
T1.Join();
T2.Join();
Console.WriteLine("Thread2Function completed");
for (int i = 1; i <= 10; i++)
{
if (T1.IsAlive)
{
Console.WriteLine("Thread1Function is still doing it's work");
Thread.Sleep(500);
}
else
{
```

```
Console.WriteLine("Thread1Function Completed");
break;
}
}
Console.WriteLine("Main Thread Completed");
Console.ReadLine();
}
public static void Thread1Function()
{
Console.WriteLine("Thread1Function started");
Thread.Sleep(5000);
Console.WriteLine("Thread1Function is about to return");
}
public static void Thread2Function()
{
Console.WriteLine("Thread2Function started");
}
}
}
```

43 .What happens if shared resources are not protected from concurrent access in a multithreaded program?

This is one of the most frequently asked **Multithreading Interview Questions** in C#.NET. The output or behavior of the program can become inconsistent. Let us understand this with an example.

```
namespace MultithreadingDemo
{
class Program
```

```
{
static int Total = 0;
public static void Main()
{
AddOneMillion();
AddOneMillion();
AddOneMillion();
Console.WriteLine("Total = " + Total);
Console.ReadLine();
}
public static void AddOneMillion()
{
for (int i = 1; i <= 1000000; i++)
{
Total++;
}
}
}
}
```

Output: Total = 3000000

The above program is a single-threaded program. In the **Main**() method, **AddOneMillion()** method is called 3 times, and it updates the Total field correctly as expected, and finally prints the correct total i.e. 3000000.

Now, let's rewrite the program using multiple threads.

```
namespace MultithreadingDemo
{
class Program
{
```

```
static int Total = 0;

public static void Main()

{

Thread thread1 = new Thread(Program.AddOneMillion);

Thread thread2 = new Thread(Program.AddOneMillion);

Thread thread3 = new Thread(Program.AddOneMillion);

thread1.Start();

thread2.Start();

thread3.Start();

thread1.Join();

thread2.Join();

thread3.Join();

Console.WriteLine("Total = " + Total);

Console.ReadLine();

}

public static void AddOneMillion()

{

for (int i = 1; i <= 1000000; i++)

{

Total++;

}

}

}

}
```

Every time we run the above program, we get a different output. The inconsistent output is because the Total field which is a shared resource is not protected from concurrent access by multiple threads. The operator ++ is not thread-safe.

44. How do protect shared resources from concurrent access?

This Multithreading Interview Question is asked in almost all interviews. So let's discuss this in detail. There are several ways to protect shared resources from concurrent access. Let's explore 2 of the options.

Using **Interlocked.Increment**() method: Modify **AddOneMillion()** method as shown below. The Interlocked.Increment() Method, increments a specified variable and stores the result, as an atomic operation

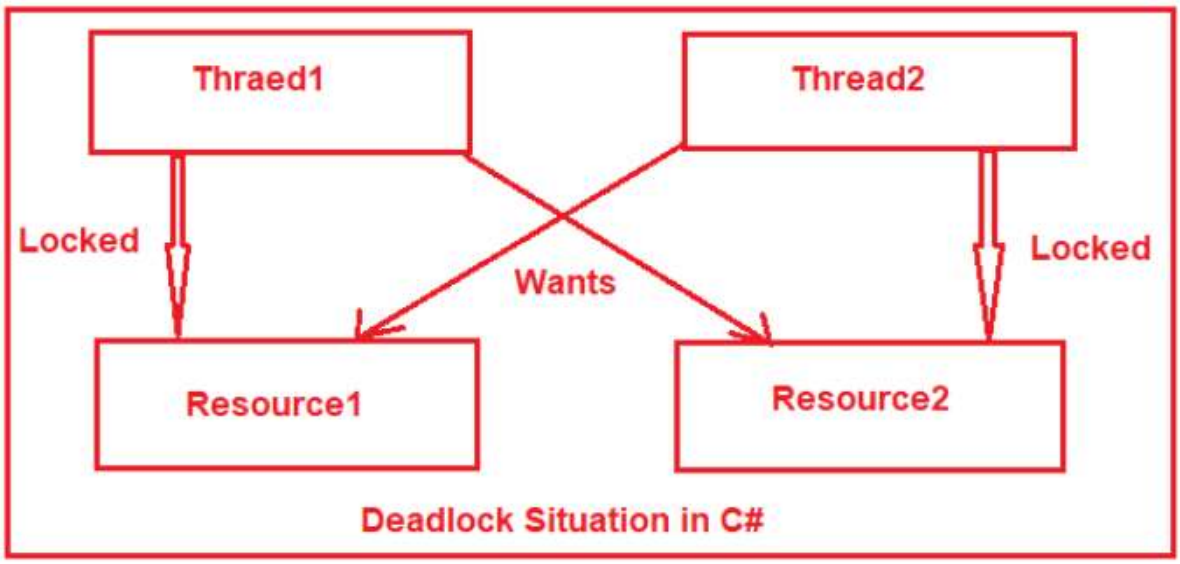
```
public static void AddOneMillion()
{
for (int i = 1; i <= 1000000; i++)
{
Interlocked.Increment(ref Total);
}
}
```

The other option is to use a lock.

```
static object _lock = new object();
public static void AddOneMillion()
{
for (int i = 1; i <= 1000000; i++)
{
lock (_lock)
{
Total++;
}
}
}
```

45.Explain why and how a deadlock can occur in multithreading with an example?

This is one of the most frequently asked Deadlock Interview Questions in C#. Let's say we have 2 threads Thread 1 and Thread 2 and 2 resources Resource 1 and Resource 2. Thread 1 has already acquired a lock on Resource 1 and wants to acquire a lock on Resource 2. At the same time, Thread 2 has already acquired a lock on Resource 2 and wants to acquire a lock on Resource 1. Two threads never give up their locks, hence a deadlock.



46 .What is AutoResetEvent and how it is different from ManualResetEvent?

This is one of the most frequently asked Deadlock Interview Questions in C#. The **AutoResetEvent** is used when we have to unlock only one single thread from several waiting blocked threads. Below are the differences from ManualResetEvent.

1. ManualResetEvent is used for unblocking many threads simultaneously. But AutoResetEvent is used for unblocking only one single thread.
2. You have to call Reset() method manually after calling Set() method to reset the ManualResetEvent. But AutoResetEvent Set() method automatically calls the Reset() method.

What is the Semaphore?

This is one of the most frequently asked Deadlock Interview Questions in C#. Semaphores are used when we have to restrict how many threads can enter a critical region. Semaphore is simply an int32 variable maintained by the kernel. We have initialized the Semaphore variable we specify the count of how many threads can enter into the critical region at a time. A thread waiting on a semaphore block when the semaphore is 0 and unblocks when the semaphore is greater than 0.

```
class Program
{
static Semaphore semaphore = new Semaphore(5, 5);
static void Main(string[] args)
```

```
{
Task.Factory.StartNew(() =>
{
for (int i = 1; i <= 15; ++i)
{
PrintSomething(i);
if (i % 5 == 0)
{
Thread.Sleep(2000);
}
}
});
Console.ReadLine();
}
public static void PrintSomething(int number)
{
semaphore.WaitOne();
Console.WriteLine(number);
semaphore.Release();
}
}
```

When we create instantiate a semaphore object, we have to provide two parameters in the constructor. The first one is the InitialCount and the second one is MaximumCount. MaximumCount denotes the maximum number of threads that can enter concurrently. InitialCount denotes the initial number of threads which can enter the Semaphore directly.

Threads enter the semaphore by calling the WaitOne method and release the semaphore by calling the Release method. You can release multiple threads bypassing the count in the Release method. By default Release method takes one and only releases one thread.

47 .What is Mutex and how it is different from other Synchronization mechanisms?

This is one of the most frequently asked Deadlock Interview Questions in C#. Mutex works similarly to AutoResetEvent and releases only one waiting thread at a time. In the AutoResetEvent any thread can call the Set() method and unblock a waiting thread. But the Mutex object remembers the thread which got the Mutex object and only that thread can release the Mutex.

Mutex object auto record the thread id which got the Mutex object and when a user calls the ReleaseMutex() method for releasing a Mutex object, it internally checks whether the releasing thread is the same as the thread which got the Mutex object if yes, then only it releases the Mutex object else it throws an exception.

Mutex famous example: The mutex is like a key to a toilet. One person can have the key – occupy the toilet – at the time. When finished, the person gives (frees) the key to the next person in the queue.

48 .What is synchronization and why it is important?

This is one of the most frequently asked Deadlock Interview Questions in C#. We use multiple threads to improve the performance of our application. When multiple threads shares data between there is a chance of data corruption. When one thread is writing to the variable and another thread is reading the same variable at the same time there is a chance of reading corrupt data.
To stop the dirty reads we use synchronization primitives.

49 .What is LiveLock?

This is one of the most frequently asked Deadlock Interview Questions in C#. A livelock is very similar to a deadlock except for involved threads states are continually changing their state but still, they cannot complete their work.
A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

50 .What is Exception Handling in C#?
Explain the difference between Error and Exception in C#?

This is one of the frequently asked Exception Handling Interview Questions in C#.
Exceptions are those which can be handled at the runtime whereas errors cannot be handled.
An exception is an object of a type deriving from the System.Exception class. The exception is thrown by the CLR (Common Language Runtime) when errors occur that are nonfatal and recoverable by user programs. It is meant to give you an opportunity to do something with a throw statement to transfer control to a catch clause in a try block.
The error is something that most of the time we cannot handle it. Errors are the unchecked exception and the developer is not required to do anything with these. Errors normally tend to signal the end of our program, it typically cannot be recovered from and should cause us to exit from the current program. It should not be caught or handled.
All the Errors are Exceptions but the reverse is not true. In general, Errors are which nobody can control or guess when it happened on the other hand Exception can be guessed and can be handled.

51 .Explain about try-catch implementation.

To implement the try-catch implementation .NET provides three keywords

1. Try
2. Catch
3. finally

try: try keyword establishes a block in which we need to write the exception causing and its related statements. That means exception causing statements must be placed in the try block so that we can handle and catch that exception for stopping abnormal termination and to display end-user understandable messages.

Catch: The catch block is used to catch the exception that is thrown from its corresponding try block. It has logic to take necessary actions on that caught exception.

Catch block syntax looks like a constructor. It does not take accessibility modifier, normal modifier, return type. It takes a single parameter of type Exception.

Inside catch block, we can write any statement which is legal in .NET including raising an exception. If the catch block is used without an exception class then it is known as a generic catch block. If the catch block is used with exception class then it is known as a specific catch block.

Finally: Finally establishes a block that definitely executes statements placed in it. Statements that are placed in finally block are always executed irrespective of the way the control is coming out from the try block either by completing normally or throwing an exception by catching or not catching.

SYNTAX TO USE TRY CATCH:

```
void SomeMethod()
{
     //Normal Statements
     try
     {
          //Exception causing and its related statement which
          //we don't want to execute when exception occurred
     }
    catch (SomeException1 one)
    {
          //Statements which required to execute when SomeException1
          //Occcurred in the program
    }
    catch (SomeException2 two)
    {
          //Statements which required to execute when SomeException2
          //Occcurred in the program
    }
    catch (Exception generic)
    {
          //This is generic exception handling block. If non of the above
          //Exception catch block handles the exception, then this catch
          //block is going to be execute
    }
   //Normal Statements
}
```