**C# Interview Questions and Answers**

In this article, I am going to discuss the most frequently asked

**50 C# Interview Questions and Answers**. This is part 1 of the C# .NET Interview Questions and Answers article series and hence in this article, we are going to discuss frequently asked basic C# Interview Questions and Answers and in our upcoming articles, we will discuss the experienced Interview questions. As part of this article, we are going to discuss the following C# Interview Questions with Answers.

1. **What is COM and what are the disadvantages of COM?**
2. **What .NET Represents?**
3. **What is a Framework and what does the .NET Framework provide?**
4. **Explain CLR and its Execution Process.**
5. **What is exactly .NET?**
6. **What are the language and its need?**
7. **What are Technology and its need?**
8. **What is Visual Studio?**
9. **Explain about BCL.**
10. **What is the Just-In-Time (JIT) compilation?**
11. **What are Metadata and an assembly?**
12. **What are the differences between managed code and unmanaged code?**
13. **What is C#?**
14. **What is the difference between an EXE and a DLL?**
15. **What’s the difference between IEnumerable<T> and List<T>?**
16. **Why is class an abstract data type?**
17. **What are the new features introduced in C# 7?**
18. **Why should you override the ToString() method?**
19. **What is the difference between string keyword and System.String class?**
20. **Are string objects mutable or immutable in C#?**
21. **What do you mean by String objects are immutable?**
22. **What is a verbatim string literal and why do we use it?**
23. **How do you create empty strings in C#?**
24. **What is the difference between System.Text.StringBuilder and System.String?**
25. **How do you determine whether a String represents a numeric value?**
26. **What is the difference between int.Parse and int.TryParse methods?**
27. **What are Properties in C#? Explain with an example?**
28. **What are the different types of properties available in C#?**
29. **What are the advantages of using properties in C#?**
30. **What is a static property? Give an example?**
31. **What is Virtual Property in C#? Give an example?**
32. **What is an Abstract Property in C#? Give an example?**
33. **Can you use virtual, override, or abstract keywords on an accessor of a static property?**
34. **What are the 2 broad classifications of data types available in C#?**
35. **How do you create user-defined data types in C#?**
36. **Difference between int and Int32 in C#**
37. **What are the differences between value types and reference types?**
38. **What do you mean by casting a data type?**
39. **What are the 2 kinds of data type conversions available in C#?**
40. **What is the difference between an implicit conversion and an explicit conversion?**
41. **What is the difference between int.Parse and int.TryParse methods?**
42. **What is Boxing and Unboxing in C#?**
43. **What happens during the process of boxing?**
44. **What are Access Modifiers in C#?**
45. **Can we use all access modifiers for all types?**
46. **Can derived classes have greater accessibility than their base types?**
47. **Can the accessibility of a type member be greater than the accessibility of its containing type?**
48. **Can destructors have access modifiers?**
49. **What do protected internal access modifiers mean?**
50. **Can you specify an access modifier for an enumeration?**

**What is COM?**

1. COM stands for Component Object Model.
2. COM is one of Microsoft Technology. Using this technology we can develop windows applications as well as web applications.

In earlier COM, VB is the programming language that is used to implement windows applications and ASP is used to implement web applications.

**What are the disadvantages of COM?**

The major two disadvantages of COM is

1. Incomplete object-oriented programming means it will not support all the features of OOPs.
2. Platform dependent means COM applications can run on only Windows OS.

**What .NET Represents?**

1. NET stands for Network Enabled Technology.
2. In .NET dot (.) refers to object-oriented and NET refers to the internet.

So the complete .NET means through object-oriented we can implement internet applications.

**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.

**What does the .NET Framework provide?**

.NET Framework provides two things such as

1. [**BCL (Base Class Libraries)**](https://dotnettutorials.net/lesson/dotnet-framework/)
2. [**CLR (Common Language Runtime)**](https://dotnettutorials.net/lesson/common-language-runtime-dotnet/)

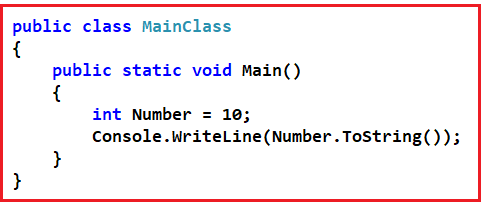
**Explain about BCL.**

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**

**Explain CLR and its Execution process.**

CLR is the core component under the [**.NET framework**](https://dotnettutorials.net/lesson/dotnet-program-execution-process/) which is responsible for converting MSIL code into native code and then execution. Let us understand the Execution flow of CLR with an example. Please have a look at the following code.



In .NET, the code is compiled twice.

1. In 1st compilation source code (High-Level Code) is compiled by the respective language compiler and the language compiler generates intermediate code which is also known as MSIL (Microsoft Intermediate Language) or IL (Intermediate language code) Or Managed code.
2. In the 2nd compilation, MSIL is converted into Native Code (Machine code) using CLR.

Always 1st compilation is slow and 2nd compilation is first.

**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.

**Explain different types of .NET Framework.**

The .Net framework is available in three different types

1. .NET Framework: This is the general version required to run .NET applications on Windows OS only.
2. .NET mono Framework: This is required if we want to run .NET applications on other OS like Unix, Linux, MAC OS, etc.
3. .NET Compact Framework: This is required to run .NET applications on other devices like mobile phones and smartphones.

**Note**: MSIL is only CPU dependent and will run only on Windows OS only using .NET Framework because .NET Framework is designed for Windows OS only.

There is another company known as “NOVEL” that designed a separate framework known as “MONO Framework”. Using this framework we can run MSIL on different OS Like Linux, UNIX, Mac, BSD, OSX, etc. .NET is platform-dependent using the .NET framework but independent using the MONO framework.

**What is not .NET?**

1. .NET is not an Operating system.
2. It is not an application or package.
3. .NET is not a database.
4. It is not an ERP application.
5. .NET is not a Testing Tool.
6. It is not a programming language.

**What is exactly .NET?**

.NET is a framework tool that supports many programming languages and many technologies. It supports 60+ programming languages. In 60+ programming languages, 9 are designed by Microsoft and the remaining are designed by Non-Microsoft.

**Microsoft designed programming languages are as follows**

1. VB.NET
2. C#.NET
3. VC++.NET
4. J#.NET
5. F#.NET
6. Jscript.NET
7. WindowsPowerShell
8. Iron Phyton
9. Iron Ruby

**Technologies supported by the .NET framework are as follows**

1. ASP.NET (Active Server Pages.NET)
2. ADO.NET (Active Data Object.NET)
3. WCF (Windows Communication Foundation)
4. WPF (Windows Presentation Foundation)
5. WWF (Windows Workflow Foundation)
6. AJAX (Asynchronous JavaScript and XML)
7. LINQ (Language Integrated Query)
8. ASP.NET MVC (Model View Controller)
9. ASP.NET WEB API

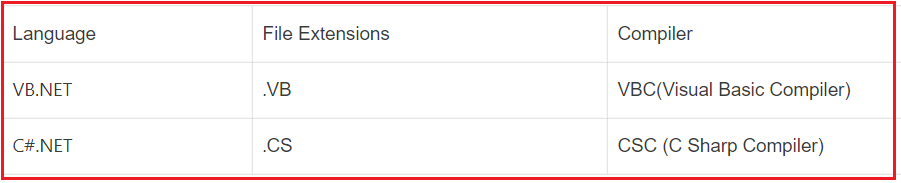
**What are the language and its need?**

1. A language acts as the mediator between the programmer and the system.
2. It offers some rules and regulations for writing the program.
3. The language also offers some libraries which are required for writing the program.
4. The collection of programs is called software.

**What are Technology and its needs?**

Technology is designed for a particular purpose. For example development of web-related applications in .NET using a technology ASP.NET. But the technology does not offer any specific rules for writing the programs. That’s why technology can’t be implemented individually.

VB.NET, C#.NET both are programming languages. Using these two languages we can implement windows/desktop applications individually. Every language is having its own compiler



**VB.NET, C#.NET:**

VB.NET and C#.NET both are programming languages. Using these two programming languages we can develop windows applications.

**ASP.NET:**

1. ASP.NET is a part of the .NET Framework.
2. ASP.NET is a technology that provides a set of predefined classes. Using these classes we can implement web applications.
3. ASP.NET is needed language support.

**ADO.NET:**

1. ADO stands for ActiveX Data Objects.
2. The ADO.NET is a .NET database technology.
3. ADO.NET provides a set of predefined classes. Using these predefined classes we can perform the operations with the database server.

**WCF:**

1. WCF stands for Windows Communication Foundation.
2. The WCF is a distributed technology. Using this technology we can implement SOA (Service Oriented Architecture) programming.
3. SOA programming provides communication between heterogeneous applications.
4. Applications that are developed using different technologies or different programming languages are known as heterogeneous applications.

**WPF**:

1. The WPF stands for windows presentation foundation.
2. WPF is a .NET technology using this technology we can create 2D, 3D, graphics, animations for windows application.
3. Using this technology we can also create our own audio/video players and also implement gaming applications.

**LINQ**:

1. LINQ stands for Language Integrated Query.
2. It is query-writing Technology.
3. LINQ offers to write queries in the programming code itself.
4. This concept is introduced in .NET framework 3.5
5. LINQ queries applying in database data and non-database data also.

**What is Visual Studio?**

Visual Studio is a Microsoft IDE tool that is needed to develop applications with the .NET framework. The IDE integrates 3 features

1. Editor
2. Compiler
3. Interpreter

**What is .Net?**

1. .NET is a programming framework created by Microsoft that developers can use to create applications more easily. The framework provides libraries commonly used by developers. The .NET Base Class Libraries (BCL) serves that purpose.
2. The .NET provides language interoperability across several programming languages. Programs are written for .NET Framework execute in a software environment called Common Language Runtime (CLR).
3. CLR is the foundation of the .NET framework which provides services like safety, memory management, garbage collection, and exception handling.
4. CLR manages the execution of code. The code that is managed by CLR is called managed code. Unmanaged code does not get managed by CLR. CLR’s interoperability helps to interact between managed and unmanaged code.
5. Common Language Specification (CLS) and Common Type System (CTS) as part of CLR. CTS is responsible for interpreting data types into a common format. CLS provides the ability of code to interact with code that is written with a different programming language.

**What is the Just-In-Time (JIT) compilation?**

The MSIL is the language that all of the .NET languages compile down to. After they are in this intermediate language, a process called Just-In-Time compilation occurs when resources are used from our application at runtime.

**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 needed to run.

**What is an assembly?**

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

**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 will be executed by the CLR 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.

**What is C#?**

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

**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**](https://dotnettutorials.net/lesson/assembly-dll-exe/). 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

**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**.

**Why is class an abstract data type?**

A Class is an Abstract Data Type because it specifies what data members and member functions (methods) contain in it (class), but it does not provide information on how those are implemented. That makes Class Abstract and Class is User Defined DataType. So, it’s an Abstract Data Type

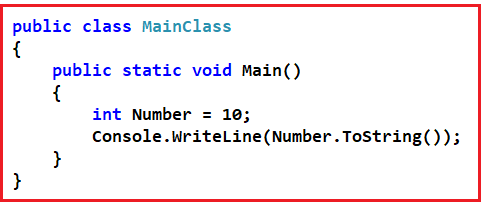
**What are the new features introduced in C# 7?**

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

**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 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 simple 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 what 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**())**;

**}**

**}**

**What is the difference between string keyword and System.String class?**

Actually there is no difference. The**string** keyword is an alias for **System.String** class. Therefore **System.String** and **string** keywords both are the same, and we can use whichever naming convention we prefer. The String class provides many methods for safely creating, manipulating, and comparing strings.

**Are string objects mutable or immutable?**

String objects are immutable.

**What do you mean by String objects are immutable?**

This C# Interview Question is frequently asked .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.

**What is a verbatim string literal and why do we use it?**

The “@” symbol is the verbatim string literal. Use verbatim strings for convenience and better readability when the string text contains backslash characters, for example in file paths. Because verbatim strings preserve newline characters as part of the string text, they can be used to initialize multiline strings. Use double quotation marks to embed a quotation mark inside a verbatim string. The following example shows some common uses for verbatim strings:

string ImagePath = @"C:\Images\Buttons\SaveButton.jpg";

//Output: C:\Images\Buttons\SaveButton.jpg

string MultiLineText = @"This is multiline

Text written to be in

three lines.";

/\* Output:

This is multiline

Text written to be in

three lines.

\*/

string DoubleQuotesString = @"My Name is ""Pranaya.""";

//Output: My Name is "Pranaya."

**Will the following code compile and run?**

**string str = null;**  
**Console.WriteLine(str.Length);**

The above code will compile but at runtime **System.NullReferenceException** will be thrown.

**How do you create empty strings in C#?**

Using string.empty as shown in the example below.

**string EmptyString = string.empty;**

**What is the difference between System.Text.StringBuilder and System.String?**

This is one of the frequently asked C#.NET Interview Questions. Objects of type StringBuilder are mutable whereas objects of type System.String is immutable. As StringBuilder objects are mutable, they offer better performance than string objects of type System.String. The StringBuilder class is present in System.Text namespace where String class is present in System namespace.

**How do you determine whether a String represents a numeric value?**

To determine whether a String represents a numeric value, we can use the **TryParse** method as shown in the example below. If the string contains non-numeric characters or the numeric value is too large or too small for the particular type you have specified, TryParse returns false and sets the out parameter to zero. Otherwise, it returns true and sets the out parameter to the numeric value of the string.

string str = "One";

**int** i = 0;

**if(int**.TryParse**(**str,out i**))**

**{**

Console.WriteLine**(**"Yes string contains Integer and it is " + i**)**;

**}**

**else**

**{**

Console.WriteLine**(**"string does not contain Integer"**)**;

**}**

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

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.

**What are Properties in C#? Explain with an example?**

It is one of the frequently asked C# Interview Questions. Properties in C# are class members that provide a flexible mechanism to read, write, or compute the values of private fields. Properties can be used as if they are public data members, but they are actually special methods called accessors. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

In the example below \_firstName and \_lastName are private string variables that are accessible only inside the Customer class. \_firstName and \_lastName are exposed using FirstName and LastName public properties respectively. The get property accessor is used to return the property value, and a set accessor is used to assign a new value. These accessors can have different access levels. The value keyword is used to define the value being assigned by the set accessor. The FullName property computes the full name of the customer. The FullName property is read-only because it has only the get accessor. Properties that do not implement a set accessor are read-only.

The code block for the get accessor is executed when the property is read and the code block for the set accessor is executed when the property is assigned a new value.

**class** Customer

**{**

// Private fileds not accessible outside the class.

**private** string \_firstName = string.Empty;

**private** string \_lastName = string.Empty;

**private** string \_coutry = string.Empty;

// public FirstName property exposes \_firstName variable

**public** string FirstName

**{**

**get**

**{**

**return** \_firstName;

**}**

**set**

**{**

\_firstName = **value**;

**}**

**}**

// public LastName property exposes \_lastName variable

**public** string LastName

**{**

**get**

**{**

**return** \_lastName;

**}**

**set**

**{**

\_lastName = **value**;

**}**

**}**

// FullName property is readonly and computes customer full name.

**public** string FullName

**{**

**get**

**{**

**return** \_lastName + ", " + \_firstName;

**}**

**}**

//Country Property is Write Only

**public** string Country

**{**

**set**

**{**

\_coutry = **value**;

**}**

**}**

**}**

**class** MainClass

**{**

**public** **static** **void** Main**()**

**{**

Customer CustomerObject = new Customer**()**;

//This line will call the set accessor of FirstName Property

CustomerObject.FirstName = "David";

//This line will call the set accessor of LastName Property

CustomerObject.LastName = "Boon";

//This line will call the get accessor of FullName Property

Console.WriteLine**(**"Customer Full Name is : " + CustomerObject.FullName**)**;

**}**

**}**

**What are the different types of properties available in C#?**

In C#, there are three types of Properties available. They are shown in the following image.

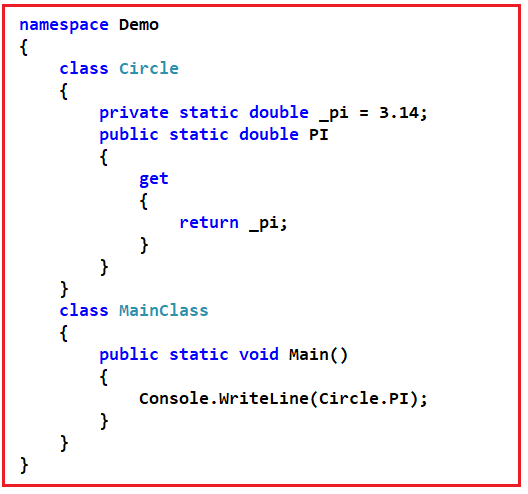
1. **Read-Only Properties:**Properties without a set accessor are considered read-only. In our example, FullName is a read-only property.
2. **Write Only Properties:**Properties without a get accessor are considered write-only. In our example, the Country is a write-only property.
3. **Read Write Properties:**Properties with both a get and set accessor are considered read-write properties. In our example, FirstName and LastName are read-write properties.

**What are the advantages of using properties in C#?**

1. Properties can validate data before allowing a change.
2. It can transparently expose data on a class where that data is actually retrieved from some other source such as a database.
3. Properties can take action when data is changed, such as raising an event or changing the value of other fields.

**What is a static property? Give an example?**

A property that is marked with a static keyword is considered as a static property. This makes the property available to callers at any time, even if no instance of the class exists. In the example below PI is a static property.



**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. BankCustomer class inherits from 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**)**;

**}**

**}**

**What is an Abstract Property in C#? Give an example?**

A property that is marked with the abstract keyword is considered abstract property. An abstract property should not have any implementation in the class. The derived classes must write their own implementation. In the example below FullName property is abstract in the Customer class. BankCustomer class overrides the inherited abstract FullName property with its own implementation.

**using** *System;*

**abstract** **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 abstract

**public** **abstract** string FullName

**{**

**get**;

**}**

**}**

**class** BankCustomer : Customer

**{**

// Overriding the FullName abstract 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**)**;

**}**

**}**

**Can you use virtual, override, or abstract keywords on an accessor of a static property?**

No, it is a compile-time error to use a virtual, abstract, or override keywords on an accessor of a static property.

**Is C# a strongly-typed language?**

Yes

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

1. Built-in data types.
2. User-defined data types.

**Give some examples of built-in data types in C#?**

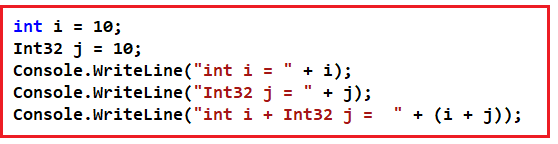
1. Int
2. Float
3. Bool

**How do you create user-defined data types in C#?**

You use the struct, class, interface, and enum constructs to create your own custom types. The .NET Framework class library itself is a collection of custom types provided by Microsoft that you can use in your own applications.

**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**

enum Test: **int**

**{**

abc = 1

**}**

**I can think of only the following minor differences between int and Int32**

1. One of the differences is in readability. When we use Int32, we are being explicit about the size of the variable.
2. To use Int32, either we need to use using System declaration or specify the fully qualified name (System.Int32) whereas with int it is not required.

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

1. Value Types
2. Reference Types

**If you define a user-defined data type by using the struct keyword, Is it a value type or reference type?**

Value Type

**If you define a user-defined data type by using the class keyword, Is it a value type or reference type?**

Reference type

**Are Value types sealed?**

Yes, Value types are sealed.

**What is the base class from which all value types are derived?**

System.ValueType

**Give examples of value types?**

1. Enum
2. Struct

**Give examples for reference types?**

1. Class
2. Delegate
3. Array
4. Interface

**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, they 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.

**What do you mean by casting a data type?**

Converting a variable of one data type to another data type is called casting. This is also called data type conversion.

**What are the 2 kinds of data type conversions available in C#?**

**Implicit conversions:**No special syntax is required because the conversion is typesafe and no data will be lost. Examples include conversions from smaller to larger integral types and conversions from derived classes to base classes.

**Explicit conversions:**Explicit conversions require a cast operator. The source and destination variables are compatible, but there is a risk of data loss because the type of the destination variable is a smaller size than (or is a base class of) the source variable.

**What is the difference between an implicit conversion and an explicit conversion?**

Explicit conversions require a cast operator whereas an implicit conversion is done automatically. The Explicit conversion can lead to data loss whereas with implicit conversions there is no data loss.

**What type of data type conversion happens when the compiler encounters the following code?**

**ChildClass CC = new ChildClass();**  
**ParentClass PC = new ParentClass();**

**Implicit Conversion:** For reference types, an implicit conversion always exists from a class to any one of its direct or indirect base classes or interfaces. No special syntax is necessary because a derived class always contains all the members of a base class.

**Will the following code compile?**

**double d = 9999.11;**  
**int i = d;**

No, the above code will not compile. Double is a larger data type than the integer. An implicit conversion is not done automatically because there is a data loss. Hence we have to use explicit conversion as shown below.

**double d = 9999.11;**  
**int i = (int)d; //Cast double to int.**

**If you want to convert a base type to a derived type, what type of conversion do you use?**

Explicit conversion as follows. Create a new derived type.  
**Car C1 = new Car();**

Implicit conversion to the base type is safe.  
**Vehicle V = C1;**

Explicit conversion is required to cast back to the derived type. The code below will compile but throw an exception at runtime if the right-side object is not a Car object.  
**Car C2 = (Car) V;**

**What operators can be used to cast from one reference type to another without the risk of throwing an exception?**

The is and as operators can be used to cast from one reference type to another without the risk of throwing an exception.

**If casting fails what type of exception is thrown?**

**InvalidCastException**

**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.

**What are Boxing and Unboxing?**

**Boxing**– Converting a value type to 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**

**Is boxing an implicit conversion?**

Yes, boxing happens implicitly.

**Is unboxing an implicit conversion?**

No, unboxing is an explicit conversion.

**What happens during the process of boxing?**

This is one of the frequently asked C# Interview Questions and Answers. Boxing is used to store value types in the garbage-collected heap. Boxing is an implicit conversion of a value type to the type object or to any interface type implemented by this value type. Boxing a value type allocates an object instance on the heap and copies the value into the new object. Due to this boxing and unboxing can have a performance impact.

**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.

**What are Access Modifiers used for?**

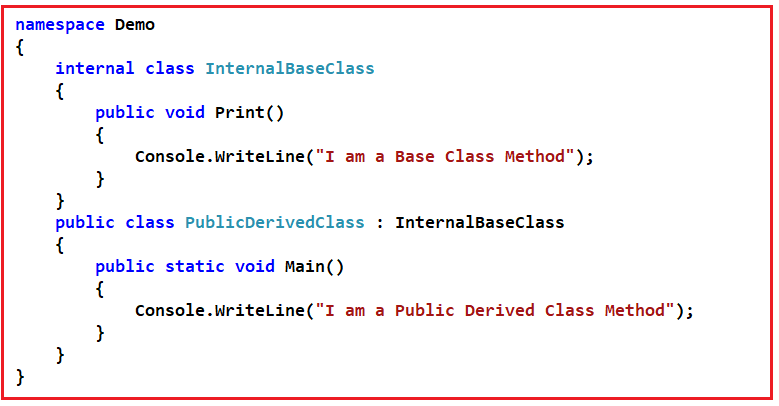
Access Modifiers are used to control the accessibility of types and members within the types.

**Can we use all access modifiers for all types?**

No, not all access modifiers can be used by all types of members in all contexts, and in some cases, the accessibility of a type member is constrained by the accessibility of its containing type.

**Can derived classes have greater accessibility than their base types?**

No, Derived classes cannot have greater accessibility than their base types. For example, the following code is illegal.



When we compile the above code an error will be generated stating “**Inconsistent accessibility: base class InternalBaseClass is less accessible than class PublicDerivedClass**“. To make this simple, you cannot have a public class **PublicDerivedClass** that derives from an internal class **InternalBaseClass**. If this were allowed, it would have the effect of making public, because all protected or internal members of A are accessible from the derived class.

**Is the following code legal in C#?**

**private** **class** Test

**{**

**public** **static** **void** Main**()**

**{**

**}**

**}**

No, a compile-time error will be generated stating **“Namespace elements cannot be explicitly declared as private, protected, or protected internal“**

**Can you declare struct members as protected?**

No, struct members cannot be declared protected. This is because structs do not support inheritance.

**Can the accessibility of a type member be greater than the accessibility of its containing type?**

No, the accessibility of a type member can never be greater than the accessibility of its containing type. For example, a public method declared in an internal class has only internal accessibility.

**Can destructors have access modifiers?**

No, destructors cannot have access modifiers.

**What does protected internal access modifiers mean?**

The protected internal access means protected OR internal, not protected, AND internal. In simple terms, a protected internal member is accessible from any class in the same assembly, including derived classes. To limit access to only derived classes in the same assembly, declare the class itself internal, and declare its members as protected.

**What is the default access modifier for a class, struct, and an interface declared directly with a namespace?**

internal

**Will the following code compile in C#?**

**interface** IInterface

**{**

**public** **void** Save**()**;

**}**

No, you cannot specify the access modifier for an interface member. Interface members are always public.

**Can you specify an access modifier for an enumeration?**

Enumeration members are always public, and no access modifiers can be specified.

**Inheritance and Interface Interview Questions in C# with Answers**

In this article, I am going to discuss the most frequently asked **Inheritance and Interface Interview Questions and Answers in C#**. Please read our previous article where we discussed the most frequently asked [**C# Interview Questions**](https://dotnettutorials.net/lesson/basic-csharp-interview-questions/) and Answers. As part of this article, we are going to discuss the following **Inheritance and Interface Interview Questions in C#** with answers.

1. **What are the 4 pillars of any object-oriented programming language?**
2. **Do structs support inheritance in C#?**
3. **What is the main advantage of using inheritance in C#?**
4. **Does C# support multiple class inheritance?**
5. **Why does C# not support multiple class inheritance?**
6. **What are the differences between interfaces and abstract classes?**
7. **When do you choose interface over an abstract class or vice versa?**
8. **What are the advantages of using interfaces?**
9. **Can an Interface contain fields?**
10. **What is the difference between class inheritance and interface inheritance?**
11. **Can an interface inherit from another interface?**
12. **Can you create an instance of an interface?**
13. **What do you mean by “Explicitly Implementing an Interface”? Give an example?**
14. **When to use Interface?**

**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**](https://dotnettutorials.net/lesson/inheritance-c-sharp/), but they can implement interfaces.

**What is the main advantage of using inheritance?**

Code reuse

**Is the following code legal?**

**class** ChildClass : ParentClassA, ParentClassB

**{**

**}**

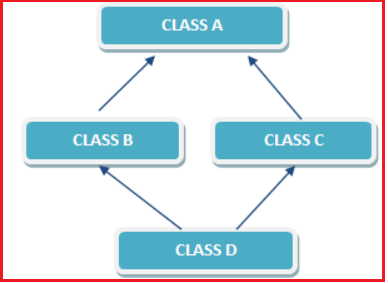
No, a child class can have only one base class. We cannot specify 2 base classes at the same time. C# supports single class inheritance only. Therefore, we can specify only one base class to inherit from. However, it does allow [**multiple interface inheritance**](https://dotnettutorials.net/lesson/multiple-inheritance-csharp/).

**Does C# support multiple class inheritance?**

No, C# supports single class inheritance only. However, classes can implement [**multiple interfaces**](https://dotnettutorials.net/lesson/multiple-inheritance-csharp/) at the same time.

**Why does C# not support multiple class inheritance?**

C# does not support multiple class inheritance because of the diamond problem that is associated, with multiple class inheritance. Let us understand the diamond problem of multiple class [**inheritance**](https://dotnettutorials.net/lesson/inheritance-c-sharp/)with an example.



As shown in the image above, I have 2 classes – Class B and Class C and Both of these classes are inherited from Class A. Now, we have another class i.e. Class D which is inherited from both Class B and Class C

So if a method in Class D calls a method defined in Class A and Class D has not overridden the invoked method. But both Class B and Class C have overridden the same method differently. Now, the ambiguity is, from which class does, Class D inherits the invoked method: Class B, or Class C?

In order not to have these problems, C# does not support multiple class inheritance.

**What is the difference between interfaces and abstract classes?**

There are several differences between an abstract class and an [**interface**](https://dotnettutorials.net/lesson/interface-c-sharp/)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**](https://dotnettutorials.net/lesson/interface-c-sharp/)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**](https://dotnettutorials.net/lesson/multiple-inheritance-csharp/) 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.

**When do you choose interface over an abstract class or vice versa?**

If we have an implementation (function with the body) that will be the same for all the derived classes, then it is better to go for an abstract class instead of an interface. When we have an interface, we can move our implementation to any class that implements the interface. Whereas, when we have an abstract class, we can share implementation for all derived classes in one central place, and avoid code duplication in derived classes.

**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#**](https://dotnettutorials.net/lesson/inheritance-c-sharp/).
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

**What is the difference between class inheritance and interface inheritance?**

Classes and structs can inherit from interfaces just like how classes can inherit a base class or struct. However, there are 2 differences.

A class or a struct can inherit from more than one [**interface**](https://dotnettutorials.net/lesson/interface-c-sharp/)at the same time whereas a class or a struct cannot inherit from more than one class at the same time

When a class or struct inherits an interface, it inherits only the method names and signatures, because the interface itself contains no implementations.

**Can an interface inherit from another interface?**

Yes, an interface can inherit from another interface. It is possible for a class to inherit an interface multiple times, through base classes or interfaces it inherits. In this case, the class can only implement the interface one time, if it is declared as part of the new class. If the inherited interface is not declared as part of the new class, its implementation is provided by the base class that declared it. It is possible for a base class to implement interface members using virtual members; in that case, the class inheriting the interface can change the interface behavior by overriding the virtual members.

**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.

**What do you mean by “Explicitly Implementing an Interface”? Give an example?**

If a class is implementing the inherited interface member by prefixing the name of the interface, then the class is “Explicitly Implementing an Interface member”. The disadvantage of Explicitly Implementing an [**Interface**](https://dotnettutorials.net/lesson/interface-c-sharp/)member is that the class object has to be typecasted to the interface type to invoke the interface member. An example is shown below.

**namespace** *Interfaces*

**{**

**interface** Car

**{**

**void** Drive**()**;

**}**

**class** Demo : Car

**{**

// Explicit implementation of an interface member

**void** Car.Drive**()**

**{**

Console.WriteLine**(**"Drive Car"**)**;

**}**

**static** **void** Main**()**

**{**

Demo DemoObject = new Demo**()**;

//DemoObject.Drive();

// Error: Cannot call explicitly implemented interface method

// using the class object.

// Type cast the demo object to interface type Car

**((**Car**)**DemoObject**)**.Drive**()**;

**}**

**}**

**}**

**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.

**Abstract and Sealed Class Interview Questions in C#**

**Abstract and Sealed Class Interview Questions in C#**

In this article, I am going to discuss the most frequently asked **Abstract and Sealed Class Interview Questions in C#** with Answers. Please read our previous article where we discussed the most frequently asked [**Interface and Inheritance Interview Questions and Answers**](https://dotnettutorials.net/lesson/inheritance-interface-interview-questions-answers-csharp/) in C#. As part of this article, we are going to discuss the following Abstract and Sealed Class Interview Questions in C# with answers.

1. **When to use Abstract Classes in C#?**
2. **What is an Abstract Class?**
3. **Can you create an instance of an abstract class?**
4. **What is a Sealed Class?**
5. **What is the abstract method in C#?**
6. **When to use the abstract method?**
7. **Can a sealed class be used as a base class?**
8. **Can an abstract class have a constructor? If so what is the use?**
9. **We cannot create an instance of an abstract class. So, what is the use of a constructor in an abstract class?**
10. **An abstract method in an abstract class does not have any implementation, so what is the use of calling it from the abstract class constructor?**
11. **When should a class be declared as abstract?**
12. **When should a method be declared as sealed?**
13. **What is the difference between the private and sealed method?**
14. **When should a class be declared as sealed?**
15. **What are the differences between an abstract class and a sealed class?**
16. **Why should the method have an abstract keyword if it does not have the body?**
17. **What are the characteristics of an abstract class?**
18. **Why can the abstract class not be instantiated?**
19. **Who will provide the implementation (body) for abstract methods?**
20. **What type of members can we define in an abstract class?**
21. **Will abstract class members are created when a subclass object is created?**
22. **How can we execute static and non-static concrete members of the abstract class?**
23. **Can we declare the abstract method as static?**
24. **Can we declare the concrete class as abstract?**
25. **Explain the differences between overriding methods and abstract methods?**
26. **What is the need for abstract classes in application development?**

**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.

**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.

**Can an abstract class have a constructor? If so what is the use?**

Yes, an abstract class can have a constructor. In general, a class constructor is used to initialize fields. Along the same lines, an abstract class constructor is used to initialize fields of the abstract class. We would provide a constructor for an abstract class if we want to initialize certain fields of the abstract class before the instantiation of a child-class takes place. An abstract class constructor can also be used to execute code that is relevant for every child’s class. This prevents duplicate code.

**We cannot create an instance of an abstract class. So, what is the use of a constructor in an abstract class?**

Though we cannot create an instance of an abstract class, we can create instances of the classes that are derived from the abstract class. So, when an instance of a derived class is created, the parent abstract class constructor is automatically called.

**Note:**Abstract classes can’t be directly instantiated. The abstract class constructor gets executed through a derived class. So, it is a good practice to use a protected access modifier with the abstract class constructor. Using public doesn’t make sense.

**An abstract method in an abstract class does not have any implementation, so what is the use of calling it from the abstract class constructor?**

If we want the abstract method to be invoked automatically whenever an instance of the class that is derived from the abstract class is created, then we would call it in the constructor of the abstract class.

**When should a class be declared as abstract?**

A class should be declared as abstract

1. When If it has any abstract methods
2. If it does not provide implementation to any of the abstract methods it inherited
3. When it does not provide implementation to any of the methods of an interface

**When should a method be declared as sealed in C#?**

If we don’t want to allow subclasses to override the superclass method and to ensure that all sub-classes use the same superclass method logic then that method should be declared as sealed.

The sealed method cannot be overridden in sub-classes violation leads to a Compile-time error:

**What is the difference between the private and sealed method in C#?**

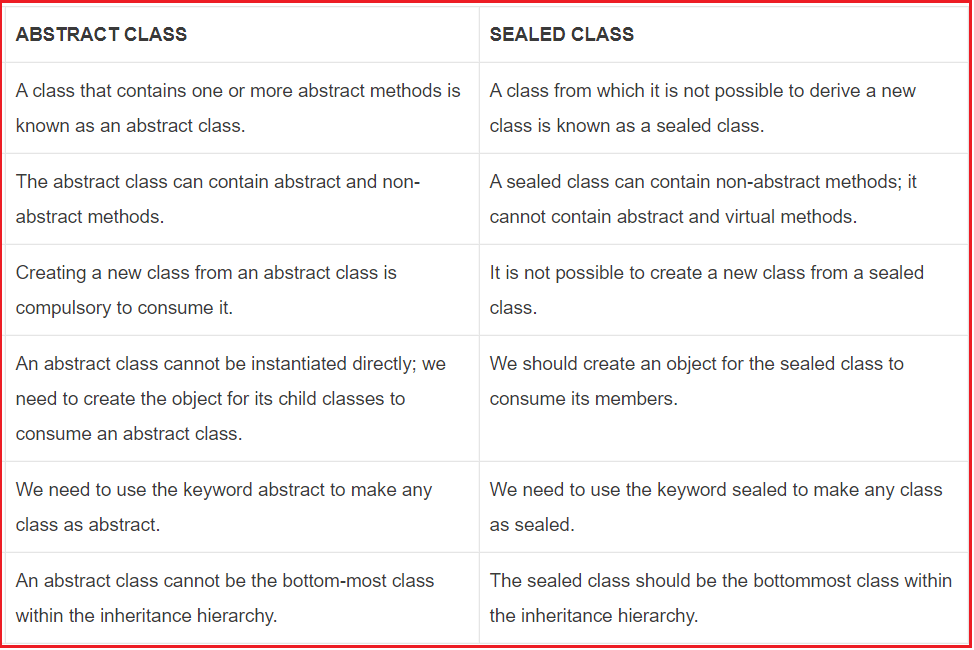
The private method is not inherited whereas the sealed method is inherited but cannot be overridden. So private method cannot be called from sub-classes whereas the sealed method can be called from sub-classes. The same private method can be defined in sub-class and it does not lead to Compile-time error.

**When should a class be declared as sealed?**

In the below situations we must define the class as sealed

1. If we don’t want to override all the methods of our class in sub-classes.
2. If we don’t want to extend our class functionality.

**What are the differences between an abstract class and a sealed class in C#?**



**Why should the method have an abstract keyword if it does not have the body?**

In a class, we are allowed only to define a class with the body. Since we are changing its default property (which means removing its body) it must have the abstract keyword in its prototype.

**What are the characteristics of an abstract class in C#?**

1. The abstract class can contain both abstract methods and non-abstract (concrete) methods.
2. It can contain both static and instance variables.
3. The abstract class cannot be instantiated but its reference can be created.
4. If any class contains abstract methods then it must be declared by using the keyword abstract.
5. An abstract class can contain sealed methods.
6. The abstract method or class cannot be declared as sealed.
7. A subclass of an abstract class can only be instantiated if it implements all of the abstract methods of its superclass. Such classes are called concrete classes to differentiate them from abstract classes.

**Why can the abstract class not be instantiated?**

Because it is not fully implemented in the class as its abstract methods cannot be executed. If the compiler allows us to create the object for the abstract class, then we can invoke the abstract method using that object which cannot be executed by CLR at runtime. Hence to restrict calling abstract methods, the compiler does not allow us to instantiate an abstract class.

**Who will provide the implementation (body) for abstract methods?**

Sub-class developers provide the body for abstract methods according to their business requirements. Basically, in projects, abstract methods (method prototype) are defined by the superclass developer and they are implemented by sub-class developers.

**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.

**Explain the differences between overriding methods and abstract methods in C#?**

The concept of the abstract method is similar to the concept of method overriding because in method overriding if a Parent class contains any **virtual** methods in it, then those methods can be re-implemented under the child class by using the **override** modifier.

In a similar way, if a parent class contains any **abstract** methods in it, those abstract methods must be implemented under the child class by using the same **override** modifier.

The main difference between method overriding and abstract method is in the case of method overriding the child class re-implementing the method is optional but in the case of the abstract method, the child class implementing the method is mandatory.

**What is the need for abstract classes in application development?**

The concepts of abstract methods and abstract classes are an extension to the inheritance wherein inheritance we have been discussing that with the help of a parent class we can provide property to the child class that can be consumed by the child classes which gives us re-usability.

Along with the parent providing property to the children, the parent can also impose the restriction on the children with the help of abstract methods so that all the child classes have to full fill the restriction without failing.

**Polymorphism Interview Questions and Answers in C#**

**Polymorphism Interview Questions and Answers in C#**

In this article, I am going to discuss the most frequently asked **Polymorphism Interview Questions and Answers in C#**. Please read our previous article where we discussed the most frequently asked [**Abstract and Sealed Class Interview Questions in C#**](https://dotnettutorials.net/lesson/abstract-class-sealed-class-interview-questions-csharp/) with Answers. As part of this article, we are going to discuss the following Polymorphism Interview Questions in C# with Answers.

1. **What is Polymorphism in C#?**
2. **Explain the different types of Polymorphism in C#?**
3. **What is compile-time Polymorphism in C#?**
4. **What is Runtime Polymorphism in C#?**
5. **Explain different types of Overloading in C#?**
6. **What is function overloading?**
7. **When should we overload methods in C#?**
8. **What are the advantages of using overloading OR what are the disadvantages if we define methods with a different name?**
9. **When is a method considered as an overloaded method?**
10. **Can we overload methods in the same class?**
11. **What is the execution control flow of overloaded methods?**
12. **What is inheritance based overloading?**
13. **What is the function/method overriding?**
14. **When must a method be overridden?**
15. **When is a sub-class method treated as an overriding method?**
16. **How can we override a parent class method under child class?**
17. **How can we execute the superclass method if it is overridden in the sub-class?**
18. **What is the difference between function overloading and function overriding?**
19. **What is method hiding?**
20. **What is the difference between Method Overriding and Method Hiding?**
21. **When can a derived class override a base class member?**
22. **What is the difference between a virtual method and an abstract method?**
23. **What is the difference between a virtual method and an abstract method?**

**What is Polymorphism in C#?**

Polymorphism is one of the primary pillars of object-oriented programming. It allows us to invoke derived class methods through a base class reference variable during runtime.

In the base class, the method is declared as virtual, and in the derived class, we override the same method. The virtual keyword indicates that the method can be overridden in any derived class.

The word Polymorphism is derived from the Greek word, where Poly means many, and morph means faces/ behaviors. So polymorphism means the ability to take more than one form.

The same function/ operator will show different behaviors when passed different types of values or the different number of values. So in simple words, we can say that behaving in different ways depending upon the input received is known as polymorphism i.e. whenever the input changes automatically the output or the behavior also changes.

**We can implement polymorphism in our application using three different approaches like**

1. Overloading
2. Overriding
3. Hiding

**Overloading again is of three types**

1. [**Method overloading**](https://dotnettutorials.net/lesson/function-overloading-csharp/)
2. Operator overloading
3. Constructor overloading

**Explain the different types of Polymorphism in C#?**

There are two types of polymorphism

1. Static polymorphism/compile-time polymorphism /early binding
2. Dynamic polymorphism / Run-time polymorphism /late binding

**What is compile-time Polymorphism in C#?**

This is one of the frequently asked [**C# Polymorphism interview**](https://dotnettutorials.net/lesson/polymorphism-csharp/) 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**](https://dotnettutorials.net/lesson/polymorphism-csharp/)or early binding. Static polymorphism is achieved by using function overloading and operator overloading

**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**](https://dotnettutorials.net/lesson/polymorphism-csharp/)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.

**Explain different types of Overloading in C#?**

Again overloading is classified into three types, such as

1. [**Method overloading / Function overloading**](https://dotnettutorials.net/lesson/function-overloading-csharp/)
2. Constructor overloading
3. Operator overloading.

**What is function overloading in C#?**

Function overloading and [**method overloading**](https://dotnettutorials.net/lesson/function-overloading-csharp/) terms are used interchangeably. [**Method overloading**](https://dotnettutorials.net/lesson/function-overloading-csharp/) allows a class to have multiple methods with the same name but with a different signature. So in C# functions can be overloaded based on the number, type (int, float, etc), and kind (Value, Ref or Out) of parameters.

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

A function overloading can be compared with person overloading. If a person has already some work to do and if we are assigning additional work to the person then the person will be overloaded.

In the same way, a function will have already some work to do and if we assign different work to the same function, then we say the function is overloaded. It is an approach of defining multiple methods with the same method name by changing the signature. Changing the signature means we can either change the no of parameters being passed to the method or type of parameters being passed to the method or order of parameters being passed to the function.

**When should we overload methods?**

To execute the same logic with different types of arguments we should overload methods. For example to add two integers, two floats and two strings we should define three methods with the same name as shown in the below application

**What are the advantages of using overloading OR what are the disadvantages if we define methods with a different name?**

If we overload the method, the user of our application gets comfort feeling in using the method with the impression that he/she calling one method bypassing different types of values.

The best example for us is the “WriteLine()” method. It is an overloaded method, not a single method of taking different types of values.

**When is a method considered as an overloaded method?**

If two methods have the same method name those methods are considered overloaded methods.

Then the rule we should check is both methods must have different parameter types/list/order. But there is no rule on return type, non-accessibility modifier and accessibility modifier means overloading methods can have their own return type, non-accessibility modifier, and accessibility modifier because overloading methods are different methods

**Can we overload methods in the same class?**

Yes, it is possible no CE, no RE. Methods can be overloaded in the same or in super and subclasses because overloaded methods are different methods.

But we can’t override the method in the same class it leads to CE: “method is already defined” because overriding methods are the same methods with a different implementation.

**What is the execution control flow of overloaded methods?**

The compiler always checks for the called method definition in reference variable type class with the given argument type parameter. So in searching and executing a method definition, we must consider both reference variable type and argument type. Referenced variable type for deciding from which class method should be to bind. Argument type for deciding which overloaded method should be a bind. For example:

**B b = new B();**  
**A a = new B();**  
**b.m1(50) => b.m1(int);**In this method call we should search m1() method definition in B class with integer parameter at the time of program compilation and bind that method definition.

**a.m1(50); => a.m1(int);**In this method call we should search m1() method defined in class A with int parameter not in class B even though the object is B.

**What is inheritance based overloading?**

A method that is defined in a class can be overloaded under its child class if we overload a method in this process we call it inheritance-based overloading.

**What is the function/method overriding?**

Redefining the superclass non-static method in the subclass with the same prototype is called [**method overriding**](https://dotnettutorials.net/lesson/function-overriding-csharp/). The overriding method is always executed from the class of the current object.

In object-oriented programming [**method overriding**](https://dotnettutorials.net/lesson/function-overriding-csharp/) is a language feature that allows a subclass to provide a specific implementation of a method that is already provided by one of its superclasses.

The implementation of the subclass overrides (replaces) the implementation of superclass methods. So the overridden method is always executed from the object whose object is stored in the reference variable. The superclass method is called the overridden method and the sub-class method is called the overriding method.

**When must a method be overridden?**

If superclass method logic is not fulfilling sub-class business requirements, the subclass should override that method with the required business logic. Usually, superclass methods are defined with generic logic which is common for all sub-classes.

**When is a sub-class method treated as an overriding method?**

If a method in sub-class contains the same signature as the superclass non-private method then the subclass method is treated as the overriding method and the superclass method is treated as the overridden method.

**How can we override a parent class method under child class?**

If we want to override a parent class method in its child class, first the method in the parent class must be declared as virtual by using the keyword virtual then only the child classes get the permission for overriding that method. Declaring the method as virtual is marking the method is overridable.

If the child class wants to override the parent class virtual method then the child class can do it with the help of the override modifier. But overriding the method under child class is not mandatory for the child classes. The Syntax is given below:

**Class1:**  
**Public virtual void show() //virtual function (overridable)**

**Class2: Class1**  
**Public override void show() //overriding**

Even if the method declared as virtual the child class may or may not override the method

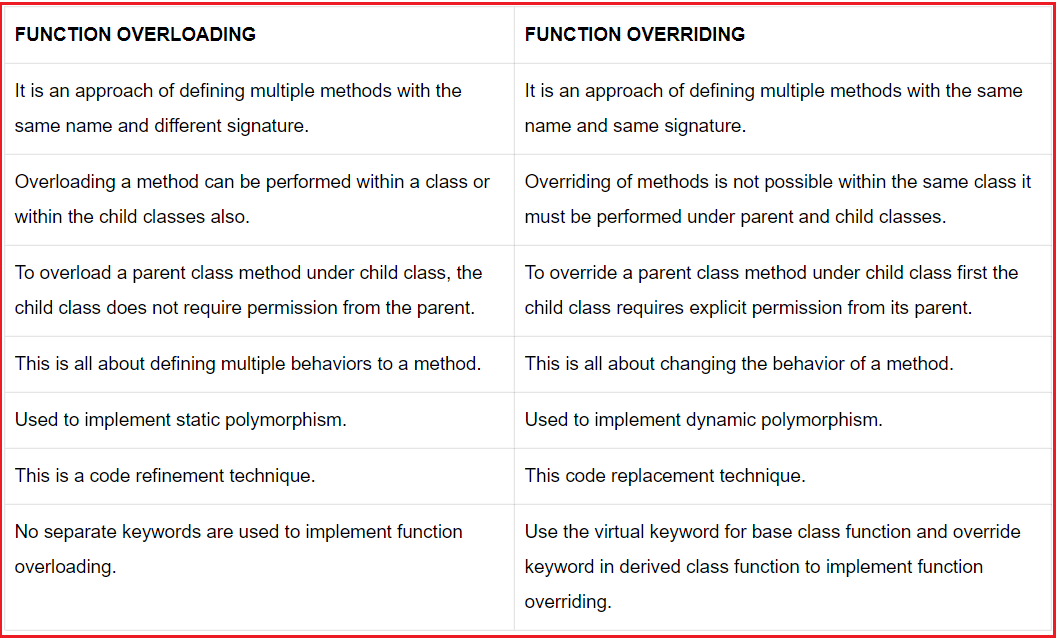
In overriding, the parent class defines a method as virtual and gives it to the child class to consume that method. So the child class now consumes the method as it is or overrides that method as per the requirement of the child class. So overriding the parent class virtual method under a child class is only optional.

**How can we execute the superclass method if it is overridden in the sub-class?**

After re-implementing parent class methods under child class, the object of the child class calls its own methods but not its parent class method, whereas if we want to still consume or call the parent class’s methods from child class, it can be done in two different ways.

By creating a parent class object under the child class, we can call the parent class methods from the child class. Or by using the base keyword, we can call parent class methods from child class, but this and base keyword cannot be used under the static block.

**What is the difference between function overloading and function overriding?**



**What is method hiding?**

Use the new keyword to hide a base class member. We will get a compile warning if we miss the new keyword. This is also used for re-implementing a parent class method under child class. Reimplementing parent class methods under child classes can be done using two different approaches, such as

1. [**Method overriding**](https://dotnettutorials.net/lesson/function-overriding-csharp/)
2. [**Method hiding**](https://dotnettutorials.net/lesson/function-hiding-csharp/)

In the first case, we re-implement the parent class methods under child classes with the permission of the parent class because here in parent class the method is declared as virtual giving the permission to child classes for overriding the methods.

In the 2nd approach, we re-implement the method of parent class even if those methods are not declared as virtual that is without parent permission we are re-implementing the methods. The Syntax is given below.

**Class1:**  
**Public void display()**

**Class2 : Class1**  
**Public new void display()**

Using the new keyword for re-implementing the methods in the child class is optional and if used will give information to hiding.

**What is the difference between Method Overriding and Method Hiding?**

This is one of the frequently asked [**Polymorphism**](https://dotnettutorials.net/lesson/polymorphism-csharp/)interview questions in C#. A parent class method can be redefined under its child class using two different approaches.

1. [**Method Overriding.**](https://dotnettutorials.net/lesson/function-overriding-csharp/)
2. [**Method Hiding.**](https://dotnettutorials.net/lesson/function-hiding-csharp/)

In [**Method overriding**](https://dotnettutorials.net/lesson/function-overriding-csharp/), 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**](https://dotnettutorials.net/lesson/function-overriding-csharp/) 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.

**When can a derived class override a base class member?**

A derived class can override a base class member only if the base class member is declared as virtual or abstract.

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

A virtual method must have a body whereas an abstract method should not have a body.

**Can fields inside a class be virtual?**

No, Fields inside a class cannot be virtual. Only methods, properties, events, and indexers can be virtual.

**Can you access a hidden base class method in the derived class?**

Yes, Hidden base class methods can be accessed from the derived class by casting the instance of the derived class to an instance of the base class as shown in the example below.

**public** **class** BaseClass

**{**

**public** **virtual** **void** Method**()**

**{**

Console.WriteLine**(**"I am a base class method."**)**;

**}**

**}**

**public** **class** DerivedClass : BaseClass

**{**

**public** new **void** Method**()**

**{**

Console.WriteLine**(**"I am a child class method."**)**;

**}**

**public** **static** **void** Main**()**

**{**

DerivedClass DC = new DerivedClass**()**;

**((**BaseClass**)**DC**)**.Method**()**;

**}**

**}**

**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.

**Partial Class and Nested Types Interview Questions and Answers in C#**

**Partial Class and Nested Types Interview Questions and Answers in C#**

In this article, I am going to discuss the most frequently asked **Partial Class and Nested Types Interview Questions and Answers in C#**. Please read our previous article where we discussed the most frequently asked [**Polymorphism Interview Questions in C#**](https://dotnettutorials.net/lesson/polymorphism-interview-questions-answers-csharp/) with Answers. As part of this article, we are going to discuss the following [**Partial Class**](https://dotnettutorials.net/lesson/partial-classes-partial-methods-csharp/)**and Nested Types Interview Questions and Answers** in C#.

1. **What is a partial class? Explain with an example.**
2. **What are the advantages of using Partial Classes in C#?**
3. **Is it possible to create partial structs, interfaces, and methods in C#?**
4. **Can you create partial delegates and enumerations?**
5. **Can different parts of a partial class inherit from different interfaces?**
6. **Can you specify nested classes as Partial Classes?**
7. **How do you create partial methods in C#?**
8. **What is the use of partial methods?**
9. **What is a Nested Type in C#? Give an example?**
10. **Can the nested class access the Containing class. Give an example?**

**What is a partial class? Explain with an example.**

A [**partial class**](https://dotnettutorials.net/lesson/partial-classes-partial-methods-csharp/) is a class whose definition is present in 2 or more files. Each source file contains a section of the class, and all parts are combined when the application is compiled.

To split a class definition, use the partial keyword as shown in the example below. The student class is split into 2 parts. The first part defines the study() method and the second part defines the Play() method. When we compile this program both the parts will be combined and compiled.

**Note that both parts use partial keyword and public access modifier.**

**namespace** *PartialClass*

**{**

**public** **partial** **class** Student

**{**

**public** **void** Study**()**

**{**

Console.WriteLine**(**"I am studying"**)**;

**}**

**}**

**public** **partial** **class** Student

**{**

**public** **void** Play**()**

**{**

Console.WriteLine**(**"I am Playing"**)**;

**}**

**}**

**public** **class** Demo

**{**

**public** **static** **void** Main**()**

**{**

Student StudentObject = new Student**()**;

StudentObject.Study**()**;

StudentObject.Play**()**;

**}**

**}**

**}**

**It is very important to keep the following points in mind when creating partial classes.**

1. All the parts must use the partial keyword.
2. The final class is the combination of all the parts at compile time.
3. All the parts must be available at compile time to form the final class.
4. Any class members declared in a partial definition are available to all the other parts.
5. All the parts must have the same access modifiers – public, private, protected, etc.

**What are the advantages of using Partial Classes in C#?**

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

When working with an automatically generated source, the code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.

**Is it possible to create partial structs, interfaces, and methods in C#?**

Yes, it is possible to create partial structs, interfaces, and methods. We can create partial structs, interfaces, and methods in the same way as we create [**partial classes**](https://dotnettutorials.net/lesson/partial-classes-partial-methods-csharp/).

**Can you create partial delegates and enumerations?**

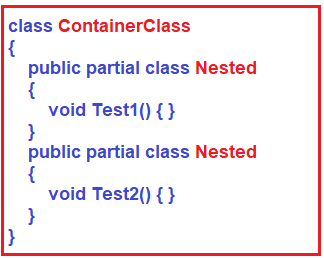
No, you cannot create partial delegates and enumerations.

**Can different parts of a partial class inherit from different interfaces?**

Yes, different parts of a [**partial class**](https://dotnettutorials.net/lesson/partial-classes-partial-methods-csharp/) can inherit from different interfaces.

**Can you specify nested classes as Partial Classes?**

Yes, nested classes can be specified as partial classes even if the containing class is not partial. An example is shown below.



**How do you create partial methods in C#?**

To create a **partial method** we create the declaration of the method in one part of the partial class and implementation in the other part of the partial class. The implementation is optional.

If the implementation is not provided, then the method and all the calls to the method are removed at compile time. Therefore, any code in the partial class can freely use a partial method, even if the implementation is not supplied. No compile-time or run-time errors will result if the method is called but not implemented.

In summary, a partial method declaration consists of two parts i.e. the definition and the implementation. These may be in separate parts of a [**partial class**](https://dotnettutorials.net/lesson/partial-classes-partial-methods-csharp/), or in the same part. If there is no implementation declaration, then the compiler optimizes away both the defining declaration and all calls to the method.

**The following are the points to keep in mind when creating partial methods.**

1. Partial method declarations must begin with the partial keyword.
2. The return type of a partial method must be void.
3. Partial methods can have ref but not out parameters.
4. The Partial methods are implicitly private, and therefore they cannot be virtual.
5. Partial methods cannot be extern, because the presence of the body determines whether they are defining or implementing.

**What is the use of partial methods?**

**Partial methods** can be used to customize generated code. They allow for a method name and signature to be reserved, so that generated code can call the method but the developer can decide whether to implement the method. Much like partial classes, partial methods enable code created by a code generator and code created by a human developer to work together without run-time costs.

**I have a class that is split into two partial classes. These two partial classes have the same methods. What happens when we compile**

At compile time all partial classes will be combined together to form a single final class. In the same class, we cannot have multiple methods with the same name. But of course method overloading possible.

**Will the following code compile?**

**public** **class** Example

**{**

**static** **void** Main**()**

**{**

TestStruct T = new TestStruct**()**;

Console.WriteLine**(**T.i**)**;

**}**

**}**

**public** struct TestStruct

**{**

**public** **int** i = 10;

//Error: cannot have instance field initializers in structs

**}**

No, a compile-time error will be generated stating “within a struct declaration, fields cannot be initialized unless they are declared as const or static”

**What do you mean by saying a “class is a reference type”?**

A class is a reference type means when an object of the class is created, the variable to which the object is assigned holds only a reference to that memory. When the object reference is assigned to a new variable, the new variable refers to the original object. Changes made through one variable are reflected in the other variable because they both refer to the same data.

**What do you mean by saying a “struct is a value type”?**

A struct is a value type mean when a struct has created the variable to which the struct is assigned to hold the struct’s actual data. When the struct is assigned to a new variable, it is copied. The new variable and the original variable, therefore, contain two separate copies of the same data. Changes made to one copy do not affect the other copy.

**When do you generally use a class over a struct?**

A class is used to model more complex behavior or data that is intended to be modified after a class object is created. A struct is best suited for small data structures that contain primarily data that is not intended to be modified after the struct is created.

**List the 5 different access modifiers in C#?**

1. Public
2. Protected
3. Internal
4. protected internal
5. private

**If you do not specify an access modifier for a method, what is the default access modifier?**

private

**Classes and structs support inheritance. Is this statement true or false?**

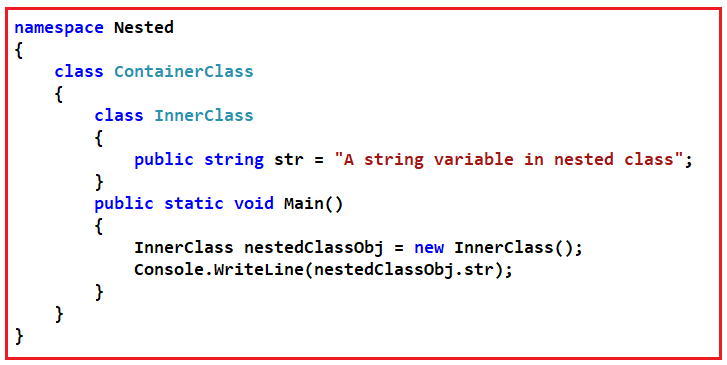
False, only classes support inheritance. structs do not support inheritance.

**If a class derives from another class, will the derived class automatically contain all the public, protected, and internal members of the base class?**

Yes, the derived class will automatically contain all the public, protected, and internal members of the base class except its constructors and destructors.

**What is a Nested Type in C#? Give an example?**

A type (class or a struct) defined inside another class or struct is called a nested type. An example is shown below. InnerClass is inside ContainerClass, Hence InnerClass is called a nested class.



**Will the following code compile in C#?**

**namespace** *Nested*

**{**

**class** ContainerClass

**{**

**class** InnerClass

**{**

**public** string str = "A string variable in nested class";

**}**

**}**

**class** Demo

**{**

**public** **static** **void** Main**()**

**{**

InnerClass nestedClassObj = new InnerClass**()**;

Console.WriteLine**(**nestedClassObj.str**)**;

**}**

**}**

**}**

No, the above code will generate a compile-time error stating – **The type or namespace name ‘InnerClass’ could not be found (are you missing a using directive or an assembly reference?)**. This is because InnerClass is inside ContainerClass and does not have any access modifier. Hence inner class is like a private member inside ContainerClass. For the above code to compile and run, we should make InnerClass public and use the fully qualified name when creating the instance of the nested class as shown below.

**namespace** *Nested*

**{**

**class** ContainerClass

**{**

**public** **class** InnerClass

**{**

**public** string str = "A string variable in nested class";

**}**

**}**

**class** Demo

**{**

**public** **static** **void** Main**()**

**{**

ContainerClass.InnerClass nestedClassObj = new ContainerClass.InnerClass**()**;

Console.WriteLine**(**nestedClassObj.str**)**;

**}**

**}**

**}**

**Can the nested class access, the Containing class. Give an example?**

Yes, the nested class or inner class can access the containing or outer class as shown in the example below. Nested types can access private and protected members of the containing type, including any inherited private or protected members.

**namespace** *Nested*

**{**

**class** ContainerClass

**{**

string OuterClassVariable = "I am an outer class variable";

**public** **class** InnerClass

**{**

ContainerClass ContainerClassObject = new ContainerClass**()**;

string InnerClassVariable = "I am an Inner class variable";

**public** InnerClass**()**

**{**

Console.WriteLine**(**ContainerClassObject.OuterClassVariable**)**;

Console.WriteLine**(**this.InnerClassVariable**)**;

**}**

**}**

**}**

**class** Demo

**{**

**public** **static** **void** Main**()**

**{**

ContainerClass.InnerClass nestedClassObj = new ContainerClass.InnerClass**()**;

**}**

**}**

**}**

**What is the output of the following program?**

**namespace** *Nested*

**{**

**class** ContainerClass

**{**

**public** ContainerClass**()**

**{**

Console.WriteLine**(**"I am a container class"**)**;

**}**

**public** **class** InnerClass : ContainerClass

**{**

**public** InnerClass**()**

**{**

Console.WriteLine**(**"I am an inner class"**)**;

**}**

**}**

**}**

**class** DemoClass : ContainerClass.InnerClass

**{**

**public** DemoClass**()**

**{**

Console.WriteLine**(**"I am a Demo class"**)**;

**}**

**public** **static** **void** Main**()**

**{**

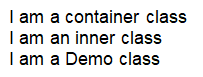
DemoClass DC = new DemoClass**()**;

**}**

**}**

**}**

**Output:**



The above program has used the concepts of inheritance and nested classes. The Container Class is at the top of the inheritance chain. The nested InnerClass derives from outer ContainerClass. Finally, the Demo Class derives from the nested Inner Class. As all the 3 classes are related by inheritance we have the above output.

**Constructor Interview Questions and Answers in C#**

**Constructor Interview Questions and Answers in C#**

In this article, I am going to discuss the most frequently asked **Constructor Interview Questions and Answers in C#**. Please read our previous article where we discussed the most frequently asked [**Partial Class Interview Questions and Answers**](https://dotnettutorials.net/lesson/partial-class-interview-questions-answers-csharp/) in C#. As part of this article, we are going to discuss the following Constructor Interview Questions in C# with Answers.

1. **What is a Constructor in C#?**
2. **Can we define a method with the same class name in C#?**
3. **How many types of constructors are there in C#.net?**
4. **Explain the Default constructor?**
5. **Explain the system-defined default constructor?**
6. **When must the developer provide the constructor explicitly?**
7. **Explain user-defined default constructor?**
8. **When should we define the parameterized constructor in a class?**
9. **Explain about parameterized constructor?**
10. **How many constructors can be defined in a class?**
11. **What is a copy constructor in C#?**
12. **Explain the static constructor in C#?**
13. **Can we initialize non-static data fields within the static constructor?**
14. **Can we initialize static data fields within the non-static constructor?**
15. **Can we initialize static data fields in both static and non-static constructor?**
16. **Explain Private constructor in C#?**
17. **When is a destructor method called?**
18. **When will be the object of a class gets destroyed?**
19. **In C# what will happen if you do not explicitly provide a constructor for a class?**
20. **Structs are not referenced types. Can structs have constructors?**
21. **We cannot create instances of static classes. Can we have constructors for static classes?**
22. **Can you prevent a class from being instantiated?**
23. **Can a class or a struct have multiple constructors?**
24. **Can a child class call the constructor of a base class?**
25. **If a child class instance is created, which class constructor is called first – base class or child class?**
26. **Can a class have a static constructor in C#?**
27. **Can you mark a static constructor with access modifiers?**
28. **Can you have parameters for static constructors?**
29. **What happens if a static constructor throws an exception?**
30. **Give 2 scenarios where static constructors can be used?**
31. **What is Destructor?**
32. **Can a class have more than 1 destructor?**
33. **Can structs in C# have destructors?**
34. **Can you pass parameters to destructors?**
35. **Can you explicitly call a destructor?**
36. **Why is it not a good idea to use Empty destructors?**
37. **What is a Constructor in C#?**
38. **Is it possible to force the garbage collector to run?**
39. **Usually, in .NET, the CLR takes care of memory management. Is there any need for a programmer to explicitly release memory and resources? If yes, why and how?**
40. **When do we generally use destructors to release resources?**
41. **When to use a Private constructor in c#?**

**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

**Explain about Default constructor?**

Constructor without parameter is called as default constructor. Again default constructor is classified into two types.

1. System-defined default constructor
2. User-defined default constructor

**Explain the system-defined default constructor?**

If as a programmer we are not defined any constructor explicitly then the system will provide one constructor at the time of compilation and that constructor is called a system-defined default constructor and that constructor will assign default values to the data fields (non-static variables).

As this constructor is created by the system this is also called a system-defined default constructor. The system will provide the default constructor only if as a programmer we are not defined any constructor explicitly.

**When must the developer provide the constructor explicitly?**

If we want to execute some logic at the time of object creation, that logic may be object initialization logic or some other useful logic, the developer must provide the constructor explicitly.

**Explain user-defined default constructor?**

The constructor which is defined by the developer without any parameter is called a user-defined default constructor. This constructor does not accept any argument but we can write the logic in its body.

The drawback of the default constructor is every instance (object) of the class will be initialized with the same values and it is not possible to initialize each instance of the class to different values.

**When should we define the parameterized constructor in a class?**

If we want to initialize objects dynamically with the user given values then we should define the parameterized constructor.

**Explain about parameterized constructor?**

The developer given constructor with parameters is called the parameterized constructor. The advantage of a parameterized constructor is we can initialize each instance of the class with different values. That means using a parameterized constructor we can store a different set of values into different objects created to the class.

**How many constructors can be defined in a class?**

In a class, we can define multiple constructors but every constructor must have a different parameters type and parameter order. So in a class, we can define one no-argument constructor plus ‘n’ number of parameterized constructors.

**What is a copy constructor in C#?**

The constructor takes a parameter of class type is called the copy constructor and this constructor is used to copy one object data into another object. The main purpose of the copy constructor is to initialize a new object (instance) with the values of an existing object (instance).

That means this constructor is used to copy the data of an existing object into a newly created object that’s why this constructor is called the copy constructor.

**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.

**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.

**When will be the object of a class gets destroyed?**

This is one of the frequently asked Constructor Interview Questions and Answers in C#. The object of a class will be destroyed by the garbage collector in any of the following cases

**Case1:** At the end of a program execution each and every object that is associated with the program will be destroyed by the garbage collector.

**Case2:** The Implicit calling of the garbage collector occurs sometime in the middle of the program execution provided the memory is full so that the garbage collector will identify unused objects of the program and destroys them.

**Case3:** The Explicit calling of the garbage collector can be done in the middle of program execution with the help of the “GC.Collect()” statement so that if there are any unused objects associated with the program will be destroyed in the middle of the program execution by the garbage collector.

**In C# what will happen if you do not explicitly provide a constructor for a class?**

If you do not provide a constructor explicitly for your class, C# will create one by default that instantiates the object and sets all the member variables to their default values.

**Structs are not referenced types. Can structs have constructors?**

Yes, even though Structs are not referenced types, structs can have constructors.

**We cannot create instances of static classes. Can we have constructors for static classes?**

Yes, static classes can also have constructors.

**Can you prevent a class from being instantiated?**

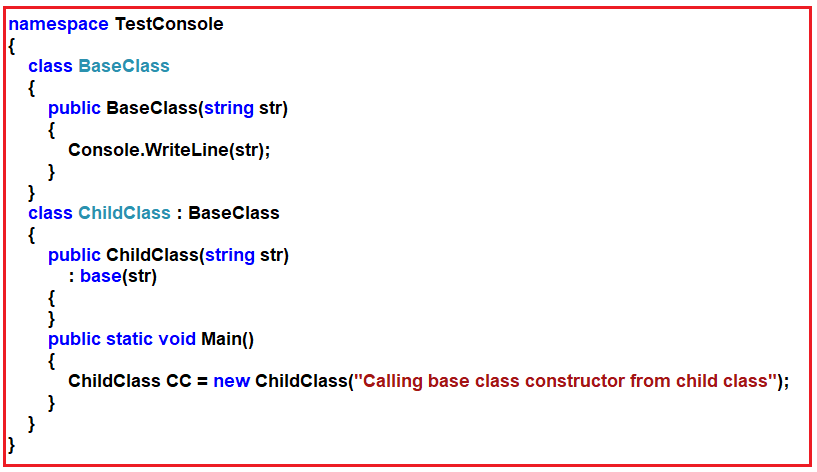
Yes, a class can be prevented from being instantiated by using a private constructor.

**Can a class or a struct have multiple constructors?**

Yes, a class or a struct can have multiple constructors. Constructors in C# can be overloaded.

**Can a child class call the constructor of a base class?**

Yes, a child class can call the constructor of a base class by using the base keyword as shown in the example below.

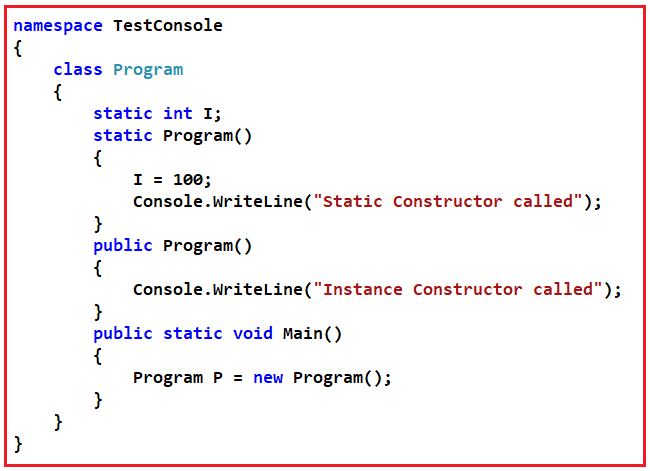


**If a child class instance is created, which class constructor is called first – base class or child class?**

When an instance of a child class is created, the base class constructor is called before the child class constructor. An example is shown below.

**Can a class have a static constructor in C#?**

Yes, a class can have a static constructor. Static constructors are called automatically, immediately before any static fields are accessed, and are generally used to initialize static class members. It is called automatically before the first instance is created or any static members are referenced. Static constructors are called before instance constructors. An example is shown below.



**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.

**Is it possible to force the garbage collector to run?**

Yes, it possible to force the garbage collector to run by calling the Collect() method, but this is not considered a good practice because this might create a performance overhead. Usually, the programmer has no control over when the garbage collector runs. The garbage collector checks for objects that are no longer being used by the application. If it considers an object eligible for destruction, it calls the destructor(if there is one) and reclaims the memory used to store the object.

**Usually, in .NET, the CLR takes care of memory management. Is there any need for a programmer to explicitly release memory and resources? If yes, why and how?**

If the application is using an expensive external resource, it is recommended to explicitly releasing the resource before the garbage collector runs and frees the object. We can do this by implementing the Dispose method from the IDisposable interface that performs the necessary cleanup for the object. This can considerably improve the performance of the application.

**When do we generally use destructors to release resources?**

If the application uses unmanaged resources such as windows, files, and network connections, we use destructors to release resources.

**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.

**Functions, Fields, and Constants Interview Questions and Answers in C#**

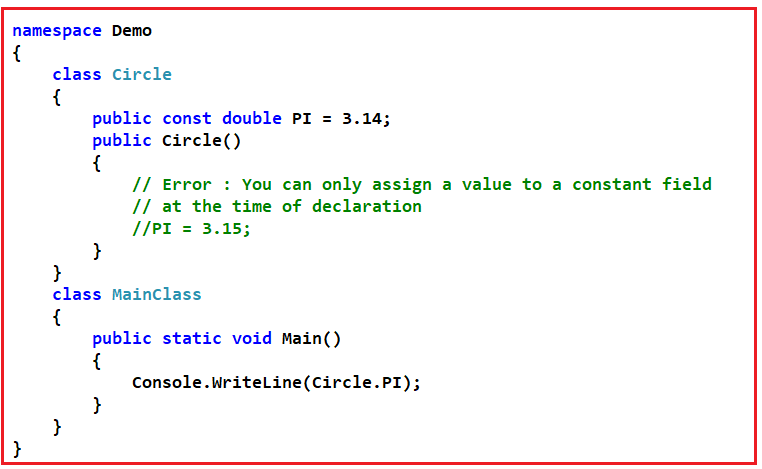
**Functions, Fields, and Constants Interview Questions and Answers in C#**

In this article, I am going to discuss the most frequently asked **Functions, Fields, Constants Interview Questions, and Answers** in C#. Please read our previous article where we discussed the most frequently asked [**Constructor Interview Questions in C#**](https://dotnettutorials.net/lesson/constructor-interview-questions-answers-csharp/) with Answers. As part of this article, we are going to discuss the following Functions Interview Questions in C# with Answers.

1. **What are the constants in C#?**
2. **Can you declare a class or a struct as constant?**
3. **Does C# support const methods, properties, or events?**
4. **Can you change the value of a constant filed after its declaration?**
5. **How do you access a constant field declared in a class in C#?**
6. **What is the difference between const and read-only?**
7. **What are the 2 broad classifications of fields in C#?**
8. **What are the instance fields in C#?**
9. **What is a static field in C#?**
10. **Can you declare a field read-only in C#?**
11. **What is the difference between a constant and a static read-only field?**
12. **Static members cannot access an object can you give the reason why?**
13. **Can we Overload the main() method in C#?**
14. **Why is C# main method static?**
15. **What is the difference between method parameters and method arguments?**
16. **Explain the difference between passing parameters by value and passing parameters by reference in C#?**
17. **Can we pass value types by reference to a method?**
18. **If a method’s return type is void, can you use a return keyword in the method?**
19. **What is the difference between static class and class with static methods? In which case I should use either of them?**
20. **What is a recursive function in c#? Give an example.**

**What are the constants in C#?**

Constants in C# are immutable values that are known at compile-time and do not change for the life of the program. The Constants are declared using the const keyword. Constants must be initialized as they are declared. You cannot assign a value to a constant after it is declared. An example is shown below.



**Can you declare a class or a struct as constant?**

No, User-defined types including classes, structs, and arrays, cannot be const. Only the C# built-in types excluding **System.Object** may be declared as const. Use the read-only modifier to create a class, struct, or array that is initialized one time at runtime (for example in a constructor) and thereafter cannot be changed.

**Does C# support const methods, properties, or events?**

No, C# does not support const methods, properties, or events.

**Can you change the value of a constant filed after its declaration?**

No, you cannot change the value of a constant filed after its declaration. In the example below, the constant field PI is always 3.14, and it cannot be changed even by the class itself. In fact, when the compiler encounters a constant identifier in the C# source code (for example, PI), it substitutes the literal value directly into the intermediate language (IL) code that it produces. Because there is no variable address associated with a constant at runtime, const fields cannot be passed by reference.

**class** Circle

**{**

**public** const **double** PI = 3.14;

**}**

**How do you access a constant field declared in a class in C#?**

Constants are accessed as if they were static fields because the value of the constant is the same for all instances of the type. You do not use the static keyword to declare them. Expressions that are not in the class that defines the constant must use the class name, a period, and the name of the constant to access the constant. In the example below the constant field, PI can be accessed in the Main method using the class name and not the instance of the class. Trying to access a constant field using a class instance will generate a compile-time error.

**class** Circle

**{**

**public** const **double** PI = 3.14;

**}**

**class** MainClass

**{**

**public** **static** **void** Main**()**

**{**

Console.WriteLine**(**Circle.PI**)**;

Circle C = new Circle**()**;

// Error : PI cannot be accessed using an instance

// Console.WriteLine(C.PI);

**}**

**}**

**What is the difference between const and read-only?**

The Read-only value can be changed at runtime however const value can never change.

**What are the 2 broad classifications of fields in C#?**

1. Instance fields
2. Static fields

**What are the instance fields in C#?**

Instance fields are specific to an instance of a type. If you have a class T, with an instance field F, you can create two objects of type T, and modify the value of F in each object without affecting the value in the other object.

**What is a static field in C#?**

A static field belongs to the class itself and is shared among all instances of that class. Changes made from instance A will be visible immediately to instances B and C if they access the field.

**Will the following code compile in C#?**

**class** Area

**{**

**public** **static** **double** PI = 3.14;

**}**

**class** MainClass

**{**

**public** **static** **void** Main**()**

**{**

Area A = new Area**()**;

Console.WriteLine**(**A.PI**)**;

**}**

**}**

No, a compile-time error will be generated stating the “**Static member ‘Area.PI’ cannot be accessed with an instance reference; qualify it with a type name instead**“. This is because the PI is a static field. Static fields can only be accessed using the name of the class and not the instance of the class. The above sample program is rewritten as shown below.

**class** Area

**{**

**public** **static** **double** PI = 3.14;

**}**

**class** MainClass

**{**

**public** **static** **void** Main**()**

**{**

Console.WriteLine**(**Area.PI**)**;

**}**

**}**

**Can you declare a field readonly in C#?**

Yes, a field can be declared readonly. A read-only field can only be assigned a value during initialization or in a constructor. An example is shown below.

**class** Area

**{**

**public** **readonly** **double** PI = 3.14;

**}**

**class** MainClass

**{**

**public** **static** **void** Main**()**

**{**

Area A = new Area**()**;

Console.WriteLine**(**A.PI**)**;

**}**

**}**

**Will the following code compile?**

**class** Area

**{**

**public** **readonly** **double** PI = 3.14;

**}**

**class** MainClass

**{**

**public** **static** **void** Main**()**

**{**

Area A = new Area**()**;

A.PI = 3.15;

Console.WriteLine**(**A.PI**)**;

**}**

**}**

No, PI is readonly. You can only read the value of PI in the Main() method. You cannot assign any value to the PI.

**What is wrong with the sample program below?**

**class** Area

**{**

**public** const **double** PI = 3.14;

**static** Area**()**

**{**

Area.PI = 3.15;

**}**

**}**

**class** MainClass

**{**

**public** **static** **void** Main**()**

**{**

Console.WriteLine**(**Area.PI**)**;

**}**

**}**

You cannot assign a value to the constant PI field.

**What is the difference between a constant and a static read-only field?**

A static readonly field is very similar to a constant, except that the C# compiler does not have access to the value of a static read-only field at compile time, only at runtime.

**Static members cannot access an object can you give the reason why?**

Because static member belongs to the class not to a specific instance of that class, so we access them using the class name itself. Static members are instantiated before the main method of class calls.

**Can we Overload the main() method in C#?**

Yes, We can overload the main() method. A C# class can have any number of main() methods.

But to run the C# class, the class should have the main() method with signature as “**public static void main(String[] args)**”. If we do any modification to this signature, the compilation will be successful. But, we will get the runtime error as the main method not found.

**Why is the C# main method static?**

The main method is static because it is available to run when our program starts and as it is the entry point of the program it runs without creating an instance of the class. In other words, static functions exist before a class is instantiated so static is applied to the main entry point (main method).

Static methods are methods that do not require any object whenever they are called. These methods are loaded even before the class is loaded in the memory. It means that even before the object is being created, the method is already loaded into the memory.

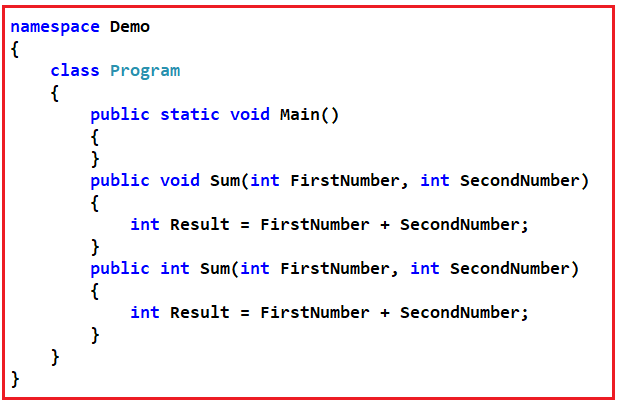
Other than this, Main() is the door for any program. It means whenever we run a program, the compiler looks out for the Main method. If there is the main method then the content inside it is executed. The main method is the first access point for any program and has to be called automatically. Since it is static it gets loaded automatically even before the object of that class is being created and Main() does not require an object to be called

**You have a component with 5 parameters and deployed to client-side now you changed your method which takes 6 parameters. How can you deploy this without affecting the client’s code?**

Instead of adding the 6th parameter to the existing method, write a new overloaded method with 6 parameters.

So when the old application calls this method, the method with 5 parameters will execute and the method with 6 parameters will be used by the new application. In this way, we can provide backward compatibility to old applications.

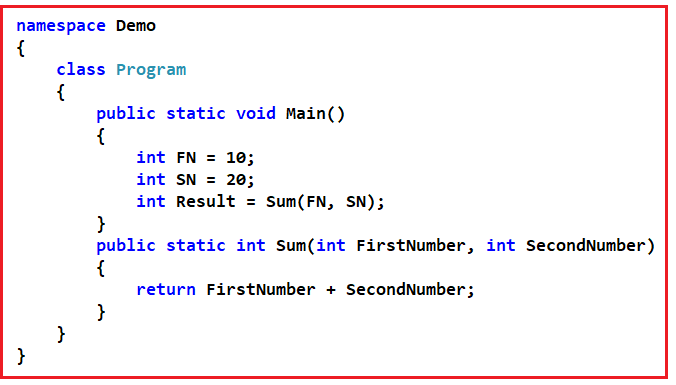
**Is the following code legal in C#?**



No, the above code does not compile. You cannot overload a method based on the return type. To overload, a method in C# either the number or type of parameters should be different. In general, the return type of a method is not part of the signature of the method for the purposes of method overloading. However, it is part of the signature of the method when determining the compatibility between a delegate and the method that it points to.

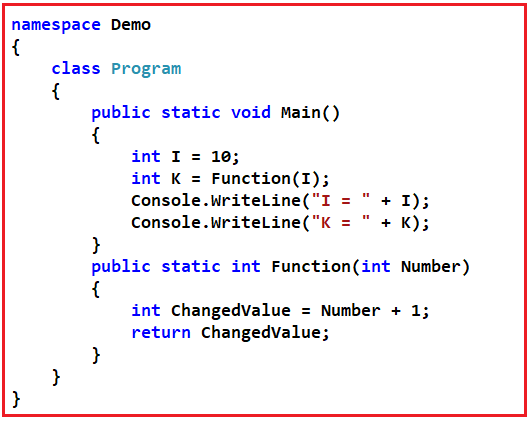
**What is the difference between method parameters and method arguments? Give an example.**

In the example below FirstNumber and SecondNumber are method parameters whereas FN and SN are method arguments. The method definition specifies the names and types of any parameters that are required. When calling code calls the method, it provides concrete values called arguments for each parameter. The arguments must be compatible with the parameter type but the argument name (if any) used in the calling code does not have to be the same as the parameter named defined in the method.

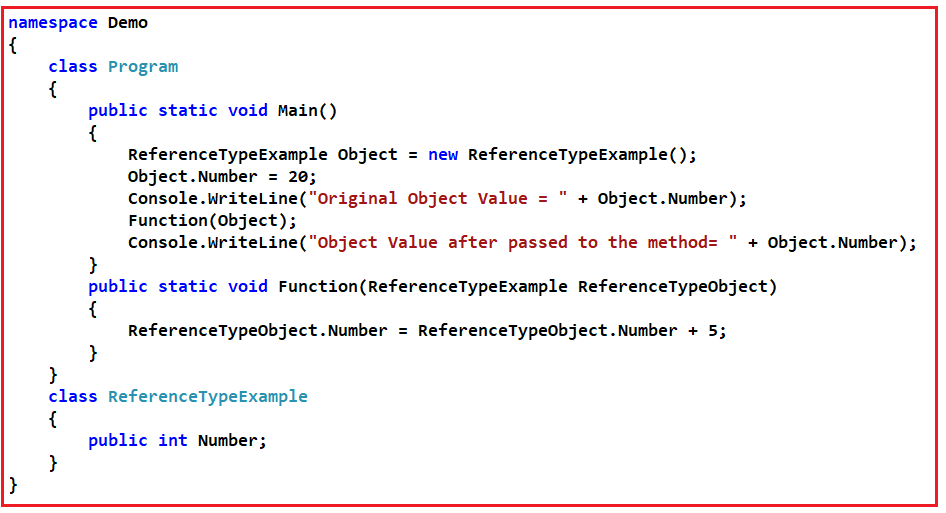


**Explain the difference between passing parameters by value and passing parameters by reference in C#?**

We can pass parameters to a method by value or by reference. By default, all value types are passed by value whereas all reference types are passed by reference. By default, when a value type is passed to a method, a copy is passed instead of the object itself. Therefore, changes to the argument have no effect on the original copy in the calling method. An example is shown below.

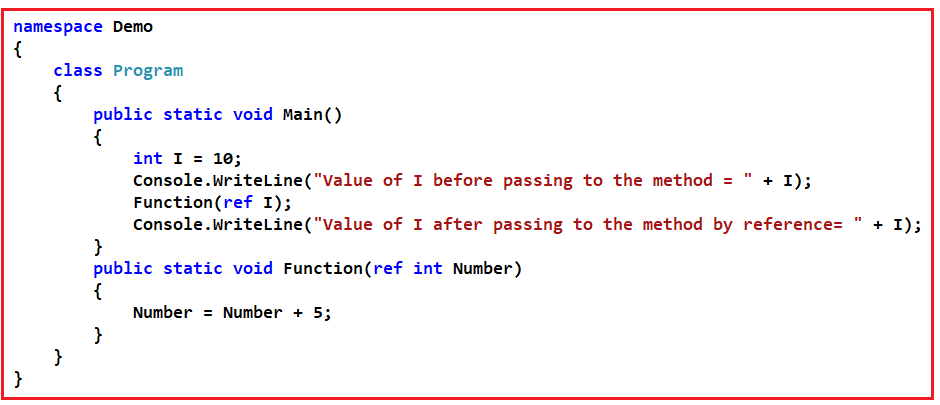


By default, reference types are passed by reference. Whenever an object of a reference type is passed to a method the reference type points to the original object, not a copy of the object. Changes made through this reference will, therefore, be reflected in the calling method. Reference types are created by using the class keyword as shown in the example below.



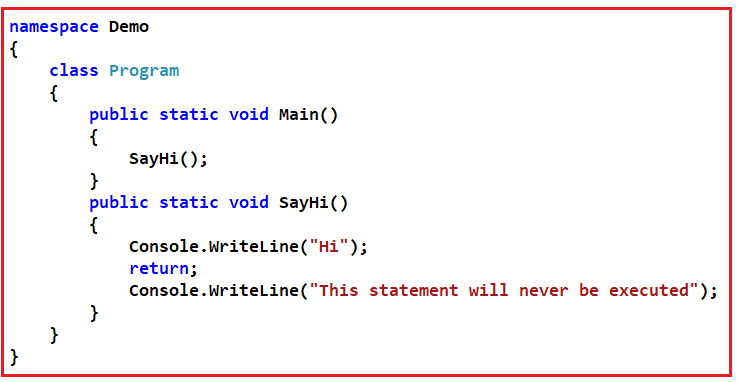
**Can you pass value types by reference to a method?**

Yes, we can pass value types by reference to a method. An example is shown below.



**If a method’s return type is void, can you use a return keyword in the method?**

Yes, Even though a method’s return type is void, you can use the return keyword to stop the execution of the method as shown in the example below.



**What is the difference between static class and class with static methods? In which case I should use either of them?**

If your class has only static members you never need an instance of that class so you should make the class itself static but if your class has instance members (non-static) then you have to make your class an instance class to access its instance members via instances of your class.

**Static class:** We cannot instantiate, inherit the static class.

**Class with full of static methods:** We can instantiate this class. But logically we are not going to do anything with these instances. So practically this scenario doesn’t exist. However, it might be useful in inheriting. But practically I haven’t come across such a situation anywhere.

When it some to the static class, we use it to extend a particular object. For example, we have the accounts object. We need to add a method HasBalance(). We can do it by extension method.

However, it is very common to have classes with both static and non-static methods.

**What is a recursive function in c#? Give an example.**

A recursive function is a function that calls itself.

**Delegate Interview Questions and Answers in C#**

**Delegate Interview Questions and Answers in C#**

In this article, I am going to discuss the most frequently asked **Delegate Interview Questions and Answers** in C#. Please read our previous article where we discussed the most frequently asked [**Functions. Fields, and Constants Interview Questions in C#**](https://dotnettutorials.net/lesson/functions-interview-questions-answers-csharp/)with Answers. As part of this article, we are going to discuss the following Delegates Interview Questions and Answers in C#.

1. **What is a Delegate in C#? Explain with one example.**
2. **Types of Delegates in C#.**
3. **What is Single Cast Delegate in C#**
4. **What is Multicast Delegate in C#? Explain with one example.**
5. **Where do you use multicast delegates?**
6. **Where did you use delegates in your project?**

**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.

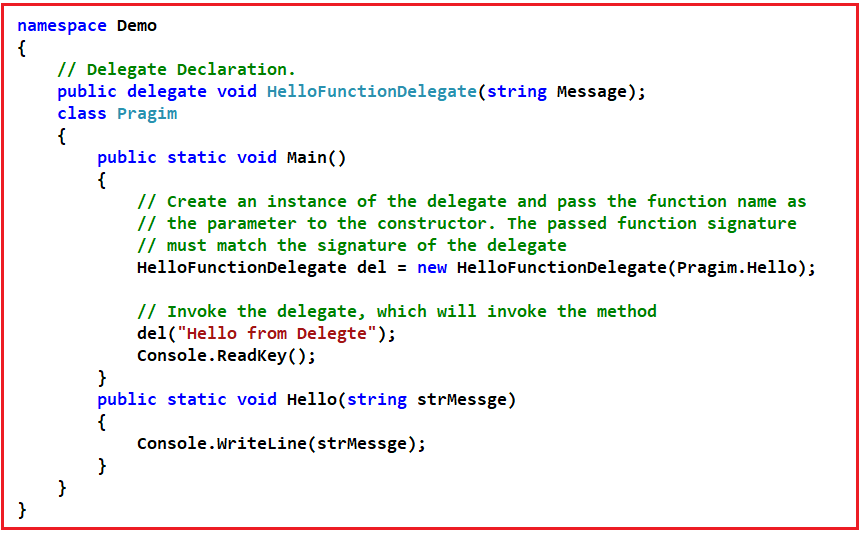
If a delegate is used for invoking multiple methods then it is known as the multicast delegate. OR the delegates that represent more than one function are called Multicast delegate.

**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:**



**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.

**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.

**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**)**;

**}**

**}**

# Multithreading and Deadlock Interview Questions and Answers in C#

**Multithreading and Deadlock Interview Questions and Answers in C#**

In this article, I am going to discuss the most frequently asked **Multithreading and Deadlock Interview Questions and Answers** in C#. Please read our previous article where we discussed the most frequently asked [**Delegates Interview Questions in C#**](https://dotnettutorials.net/lesson/delegate-interview-questions-answers-csharp/) with Answers. As part of this article, we are going to discuss the following Multithreading and Deadlock Interview Questions and Answers in C#.

1. **What are the Thread and Process?**
2. **What is the difference between Process and Thread?**
3. **Why we need Multi-threading in our project?**
4. **What are the advantages and disadvantages of multithreading in C#?**
5. **How can we create a Thread in C#?**
6. **Why does a delegate need to be passed as a parameter to the Thread class constructor?**
7. **How to pass the parameter in Thread?**
8. **Why we need a ParameterizedThreadStart delegate?**
9. **When to use ParameterizedThreadStart over ThreadStart delegate?**
10. **How to pass data to the thread function in a type-safe manner?**
11. **How to retrieve the data from a thread function?**
12. **What is the difference between Threads and Tasks?**
13. **What is the Significance of Thread.Join and Thread.IsAlive functions in multithreading?**
14. **What happens if shared resources are not protected from concurrent access in a multithreaded program?**
15. **How to protect shared resources from concurrent access?**
16. **What are the Interlocked functions?**
17. **What is Lock?**
18. **What is the Difference between Monitor and lock in C#?**
19. **Explain why and how a deadlock can occur in multithreading with an example?**
20. **How to resolve a deadlock in a multithreaded program?**
21. **What is AutoResetEvent and how it is different from ManualResetEvent?**
22. **What is the Semaphore?**
23. **Explain Mutex and how it is different from other Synchronization mechanisms?**
24. **What is the Race condition?**
25. **How can you share data between multiple threads?**
26. **What are Concurrent Collection Classes?**
27. **What is synchronization and why it is important?**
28. **Explain the four necessary conditions for Deadlock?**
29. **What is LiveLock?**

**What are the Thread and Process?**

**Process –** Process is something that the operating system uses to execute a program by providing the resources required. Each process has a unique process id associated with it. We can view the process within which a program is running using the windows task manager.

**Thread –**A Thread is a lightweight process that is responsible for executing application code. A process has at least one thread which is commonly called the main thread which actually executes the application code. A single process can have multiple threads.

Every application by default contains one thread to execute the program and that thread is known as the main thread. So every program by default is a single-threaded model.

**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 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 lightweight than process. 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.

**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:**

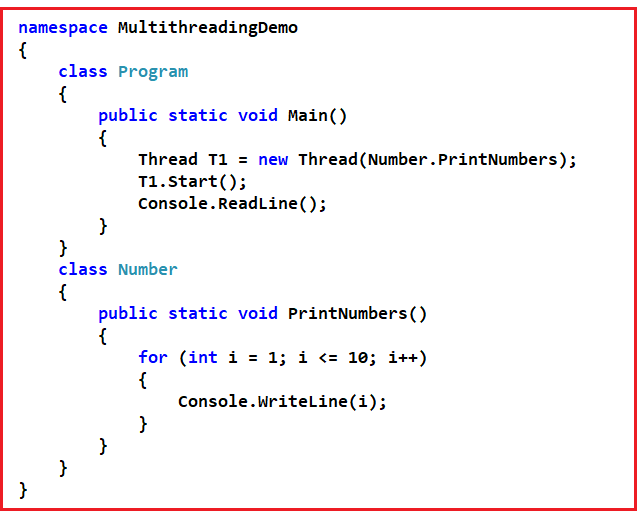
1. On a single-core/processor machine threading can affect performance negatively as there is overhead involved with context-switching.
2. Have to write more lines of code to accomplish the same task.
3. Multithreaded applications are difficult to write, understand, debug, and maintain.

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

**How can we create a Thread in C#?**

To create a THREAD, we need to create an instance of Thread class (Thread class provided by System.Threading namespace) and to its constructor, we have to pass the function name as a parameter that we want the thread to execute. Then we need to call the start method of the Thread class.

**Let us understand this with an example as shown below.**



**In the above example, the following statement does the same**  
**Thread T1 = new Thread(Number.PrintNumbers); T1.Start();**

**We can rewrite the above line using the ThreadStart delegate as shown below.**  
**Thread T1 = new Thread(new ThreadStart(Number.PrintNumbers));**  
**T1.Start();**

**Why does a delegate need to be passed as a parameter to the Thread class constructor?**

As we know the purpose of creating a Thread is to execute a function. We also know that a delegate is a type-safe function pointer meaning it points to a function that the thread has to execute. In short, all threads require an entry point to start execution. Any thread that we create will need an explicitly defined entry point i.e. a pointer to the function from where they should begin execution. So threads always require a delegate.

**In the code below, we are not explicitly creating the ThreadStart delegate, then how is it working here?**

**Thread T1 = new Thread(Number.PrintNumbers);**

It’s working in spite of not creating the ThreadStart delegate explicitly because the framework is doing it automatically for us. We can also rewrite the same line using delegate() keyword as shown below.  
**Thread T1 = new Thread(delegate() { Number.PrintNumbers(); });**

We can also rewrite the same line using the lambda expression as shown below.  
**Thread T1 = new Thread(() => Number.PrintNumbers());**

**How to pass the parameter in Thread?**

In the constructor of Thread, we can pass the method name which accepts only a single parameter. Then we have to pass the parameter into the Start method.

**Why we need a ParameterizedThreadStart delegate?**

When we need to pass data to the thread function then in such situations we need to use **ParameterizedThreadStart delegate.**Here is an example that shows the usage of a ParameterizedThreadStart delegate.

**namespace** *MultithreadingDemo*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

Console.WriteLine**(**"Please enter the target number"**)**;

**object** target = Console.ReadLine**()**;

// Create an instance of ParameterizedThreadStart delegate

ParameterizedThreadStart parameterizedThreadStart =

new ParameterizedThreadStart**(**Number.PrintNumbers**)**;

Thread T1 = new Thread**(**parameterizedThreadStart**)**;

// Pass the traget number to the start function, which

// will then be passed automatically to PrintNumbers() function

T1.Start**(**target**)**;

**}**

**}**

**class** Number

**{**

**public** **static** **void** PrintNumbers**(object** target**)**

**{**

**int** number = 0;

**if** **(int**.TryParse**(**target.ToString**()**, out number**))**

**{**

**for** **(int** i = 1; i **<**= number; i++**)**

**{**

Console.WriteLine**(**i**)**;

**}**

**}**

**}**

**}**

**}**

**The code in the Main() function can also be written as shown below.**

**public** **static** **void** Main**()**

**{**

Console.WriteLine**(**"Please enter the target number"**)**;

**object** target = Console.ReadLine**()**;

Thread T1 = new Thread**(**Number.PrintNumbers**)**;

T1.Start**(**target**)**;

**}**

**In the above example, we are not explicitly creating an instance of the ParameterizedThreadStart delegate. Then how is it working?**

It’s working because the compiler implicitly converts **new Thread(Number.PrintNumbers)**  to

**new Thread(new ParameterizedThreadStart(Number.PrintNumbers));**

**When to use ParameterizedThreadStart over ThreadStart delegate?**

We need to use ParameterizedThreadStart delegate if we have some data to pass to the Thread function, otherwise, just use ThreadStart delegate.

**Please note:**Using ParameterizedThreadStart delegate and Thread.Start(Object) method to pass data to the Thread function is not type-safe as they operate on object datatype and any type of data can be passed. In the above, If we try to change the data type of the target parameter of PrintNumbers() function from object to int, a compiler error will be raised as the signature of PrintNumbers() function does not match with the signature of ParameterizedThreadStart delegate.

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

This MultiThreading Interview Questions 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**)**;

**}**

**}**

**}**

**}**

**How to retrieve the data from a thread function?**

Retrieving data from the Thread function using the callback method.

**namespace** *ThreadStartDelegateExample*

**{**

// Step 1: Create a callback delegate. The actual callback method

// signature should match with the signature of this delegate.

**public** **delegate** **void** SumOfNumbersCallback**(int** sumOfNumbers**)**;

// Step 2: Create Number class to compute the sum of numbers and

// to call the callback method

**class** Number

**{**

// The traget number this class needs to compute the sum of numbers

**int** \_target;

// Delegate to call when the the Thread function completes

// computing the sum of numbers

SumOfNumbersCallback \_callbackMethod;

// Constructor to initialize the target number and the callback delegateinitialize

**public** Number**(int** target, SumOfNumbersCallback callbackMethod**)**

**{**

this.\_target = target;

this.\_callbackMethod = callbackMethod;

**}**

// This thread function computes the sum of numbers and then invokes

// the callback method passing it the sum of numbers

**public** **void** ComputeSumOfNumbers**()**

**{**

**int** sum = 0;

**for** **(int** i = 1; i **<**= \_target; i++**)**

**{**

sum = sum + i;

**}**

**if** **(**\_callbackMethod != **null)**

**{**

\_callbackMethod**(**sum**)**;

**}**

**}**

**}**

// Step 3: This class consumes the Number class created in Step 2

**class** Program

**{**

// Callback method that will receive the sum of numbers

**public** **static** **void** PrintSumOfNumbers**(int** sum**)**

**{**

Console.WriteLine**(**"Sum of numbers is " + sum**)**;

**}**

**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 callback delegate and to it's constructor

// pass the name of the callback function (PrintSumOfNumbers)

SumOfNumbersCallback callbackMethod = newSumOfNumbersCallback**(**PrintSumOfNumbers**)**;

// Create an instance of Number class and to it's constrcutor pass the target

// number and the instance of callback delegate

Number number = new Number**(**target, callbackMethod**)**;

// Create an instance of Thread class and specify the Thread function to invoke

Thread T1 = new Thread**(**new ThreadStart**(**number.ComputeSumOfNumbers**))**;

T1.Start**()**;

**}**

**}**

**}**

**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 Task does not complete until its child 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.

**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 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 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"**)**;

**}**

**}**

**}**

**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.

**How to protect shared resources from concurrent access?**

This Multithreading Interview Questions asked in almost all interviews. So let 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++;

**}**

**}**

**}**

**Which option is better?**

**From a performance perspective using the Interlocked class is better over using locking.**Locking locks out all the other threads except a single thread to read and increment the Total variable. This will ensure that the Total variable is updated safely. The downside is that since all the other threads are locked out, there is a performance hit.

The Interlocked class can be used with addition/subtraction (increment, decrement, add, etc.) on an int or long field. The Interlocked class has methods for incrementing, decrementing, adding, and reading variables atomically.

**The following code prints the time taken in ticks.**1 millisecond consists of 10000 ticks.

**public** **static** **void** Main**()**

**{**

Stopwatch stopwatch = Stopwatch.StartNew**()**;

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**)**;

stopwatch.Stop**()**;

Console.WriteLine**(**"Time Taken in Ticks = " + stopwatch.ElapsedTicks**)**;

**}**

**Please Note:** You can use the TimeSpan object to find ticks per second, ticks per millisecond, etc. Stopwatch class is in **System.Diagnostics** namespace.

**What are the Interlocked functions?**

Interlocked functions in .NET are useful in multithreading programs to safely change the value of shared variables. By default, C# variables are not thread-safe. When we apply addition, subtraction, or checking the value of variables multiple threads can corrupt the variables. For preventing these dirty reads, we can use Interlocked functions.

Interlocked functions can only work on int, long, double, and float data types

**What is Lock?**

The Lock is another synchronization mechanism in C# and one of the famous multi-threading interview questions in .NET. It restricts the critical region so that only one thread can enter a critical region at a time.

The lock needs an object to continue its operation. It applies a lock on a target object and only one thread can lock that target object at a time

**What is the Difference between Monitor and lock in C#?**

I think you need to be prepared for this **Multithreading Interview Questions and Answer**if you are preparing to attend the interview questions on Multithreading. So let us discuss this question in details

1. The Lock is just a shortcut for the Monitor statement. Compiler internally converts lock statement to **Monitor.Enter** and **Exit** statements.
2. Monitor class provides some useful method which is not in lock statement. These methods are very useful in advanced scenarios.
3. The monitor provides TryEnter method. This method is useful when we need to provide a timeout value.
4. TryEnter is also useful when we have to check whether the lock is taken or not. We can pass a boolean parameter which returns true if the lock is taken else returns false.
5. The Pulse method notifies a waiting thread of a change in the locked object’s state.
6. Wait method releases the current acquired lock and blocks the current thread until it reacquires the lock.

Both **Monitor class**and **lock**provides a mechanism to protect the shared resources in a multithreaded application. The lock is the shortcut for Monitor.Enter with the try and finally.

**This means that the following code**

**static** **object** \_lock = new **object()**;

**public** **static** **void** AddOneMillion**()**

**{**

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

**{**

**lock** **(**\_lock**)**

**{**

Total++;

**}**

**}**

**}**

**can be rewritten as shown below:**

**static** **object** \_lock = new **object()**;

**public** **static** **void** AddOneMillion**()**

**{**

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

**{**

// Acquires the exclusive lock

Monitor.Enter**(**\_lock**)**;

**try**

**{**

Total++;

**}**

**finally**

**{**

// Releases the exclusive lock

Monitor.Exit**(**\_lock**)**;

**}**

**}**

**}**

**From C# 4, it is implemented slightly differently as shown below**

**static** **object** \_lock = new **object()**;

**public** **static** **void** AddOneMillion**()**

**{**

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

**{**

**bool** lockTaken = **false**;

// Acquires the exclusive lock

Monitor.Enter**(**\_lock, **ref** lockTaken**)**;

**try**

**{**

Total++;

**}**

**finally**

**{**

// Releases the exclusive lock

**if** **(**lockTaken**)**

Monitor.Exit**(**\_lock**)**;

**}**

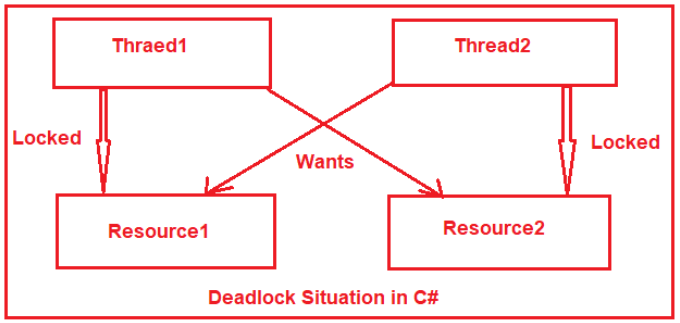
**}**

**}**

**So, in short, the lock is a shortcut and it’s the option for basic usage.** If we need more control to implement advanced multithreading solutions using TryEnter(), Wait(), Pulse(), & PulseAll() methods, then the Monitor class is our option.

**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.



**How to resolve a deadlock in a multithreaded program?**

There are several techniques to avoid and resolve deadlocks. For example

1. Acquiring locks in a specific defined order
2. Mutex class
3. Monitor.TryEnter() method

**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 unblocks many threads simultaneously. But AutoResetEvent is used for unblocks 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.

**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 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.

**What is the Race condition?**

A race condition happens when two or more threads want to update shared data at the same time.

**What is the volatile keyword?**

Volatile is used for serializing the access of variables without using the synchronization primitives. You can use volatile with below types:

1. References type
2. Pointer types
3. Values types
4. IntPtr

**How can you share data between multiple threads?**

There are two ways to share data between multiple threads:

1. Concurrent collection classes
2. Using Synchronization Primitives

**What are Concurrent Collection Classes?**

.NET Framework class library comes with Concurrent collection classes so that multiple threads can share collection data between them without using synchronization primitives.

There are four types of Concurrent classes.

1. ConcurrentQueue
2. ConcurrentStack
3. ConcurrentDictionary
4. ConcurrentBag

**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 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.

**Can you count some names of Synchronization primitives?**

1. Monitor
2. Mutex
3. Spinlock
4. ReaderWriterLock
5. Semaphore
6. AutoResetEvent
7. ManualResetEvent
8. Interlocked
9. CountDownEvent
10. Barrier

**What are the four necessary conditions for Deadlock?**

1. **Mutual Exclusion**: Resources involved must be unshareable between multiple threads.
2. **Hold and Wait**: Threads hold the resources they have allocated and waiting for the resources they want.
3. **No pre-emption**: If the thread locks the resource, other threads cannot take the resource until the thread releases it.
4. **Circular Wait**: A circular chain must exist in which each thread waiting for other threads to release resources.

**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.

**Exception Handling Interview Questions in C#**

**Exception Handling Interview Questions in C# with Answers**

In this article, I am going to discuss the most frequently asked **Exception Handling Interview Questions in C#** with Answers. Please read our previous article where we discussed the most frequently asked [**Multithreading and Deadlock Interview Questions in C#**](https://dotnettutorials.net/lesson/multi-threading-interview-questions/)with Answers. As part of this article, we are going to discuss the following Exception Handling Interview Questions in C# with Answers.

1. **Explain different types of errors in C#.**
2. **What is a Compilation Error and Runtime Error in C#?**
3. **What is an Exception?**
4. **Who is responsible for abnormal termination of the program whenever runtime errors occur?**
5. **What happens if an exception is raised in the program?**
6. **What CLR does when a logical mistake occurred in the program?**
7. **Explain Exception Handling?**
8. **Why we need exception handling?**
9. **What is the Exception Handling Procedure?**
10. **How can we handle the exception in .NET?**
11. **Explain the difference between Error and Exception in C#?**
12. **What is the difference between System exceptions and Application exceptions?**
13. **Explain about try-catch implementation.**
14. **Explain the Different Properties of the Exception class in C#.**
15. **Can we catch all exceptions using a single catch block?**
16. **When should we write multiple catch blocks for a single try block?**
17. **Explain about the finally block**
18. **Why we need finally block in the real-time project?**
19. **How many ways we can use try-catch and finally?**
20. **What happens if the finally block throws an exception?**
21. **What is the difference between the “throw” and “throw ex” in .NET?**
22. **What is Inner Exception?**

**Explain different types of errors in C#.**

When we write and execute our code in the .NET framework then there is possible of three types of error occurrences they are

1. Compilation errors
2. Runtime errors

**What is a Compilation Error in C#?**

An error that occurs in a program at the time of compilation is known as the compilation error. These errors occur due to syntax mistakes under the program.

These errors occur by typing the wrong syntax like missing double quotes and terminators, typing wrong spelling for keywords, assigning wrong data to a variable, trying to create an object for abstract class and interface, etc.

That means this error occurs due to the poor understanding of the programming language. These errors can be identified by the programmer and can be rectified before the execution of the program only. So these errors do not cause any harm to the program execution.

**What is Runtime Error in C#?**

The errors which are occurred at the time of program execution are called as a runtime error. These errors are like entering wrong data into a variable, trying to open a file for which there is no permission, trying to connect to the database with wrong user id and password, the wrong implementation of logic, missing of required resources, etc.

The Runtime errors are dangerous because whenever they occur in the program, the program terminates abnormally on the same line where the error gets occurred without executing the next line of code.

**What is an Exception?**

A runtime error is known as an exception. An exception cannot be identified and rectified by the programmer. The exception will cause abnormal termination of the program execution.

So these errors (exceptions) are very dangerous because whenever an exception occurs in the programs the program gets terminated abnormally on the same line where the error gets occurred without executing the next line of code.

**Who is responsible for abnormal termination of the program whenever runtime errors occur?**

The Objects of exception classes are responsible for abnormal termination of the program whenever runtime errors occur. These exception classes are predefined under BCL where a separate class is provided for each and every different type of exception like

1. IndexOutOfRangeException
2. FormatException
3. NullReferenceException
4. DivideByZeroException
5. FileNotFoundException
6. SQLException,
7. OverFlowException, etc.

Each exception class provides specific exception error message.

All exception classes are responsible for abnormal termination of the program as well as after abnormal termination of the program they will be displaying an error message which specifies the reason for abnormal termination i.e. they provide an error message specific to that error.

Whenever a runtime error occurs in a program, first the exception manager under CLR identifies the type of error that occur in the program, creates an object of the exception class related with that error and throws that object which will immediately terminate the program abnormally on the line where error got occur and display the error message related with that class.

**What happens if an exception is raised in the program?**

Program execution is terminated abnormally. It means statements placed after exception causing statement are not executed but the statements placed before that exception causing statement are executed by CLR.

**What CLR does when a logical mistake occurred in the program?**

This is one of the frequently asked Exception Handling Interview Questions in C#.

It creates an exception class object that is associated with that logical mistake and terminates the current method execution by throwing that exception object by using the “throw” keyword.

So we can say an exception is an event that occurs during the execution of a program that disrupts the normal flow of instruction execution.

**Below program shows program execution without exception:**

**namespace** *ExceptionHandlingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** a = 20;

**int** b = 10;

**int** c;

Console.WriteLine**(**"A VALUE = " + a**)**;

Console.WriteLine**(**"B VALUE = " + b**)**;

c = a / b;

Console.WriteLine**(**"C VALUE = " + c**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Below program shows program execution with exception:**

**namespace** *ExceptionHandlingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** a = 20;

**int** b = 0;

**int** c;

Console.WriteLine**(**"A VALUE = " + a**)**;

Console.WriteLine**(**"B VALUE = " + b**)**;

c = a / b;

Console.WriteLine**(**"C VALUE = " + c**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**  
**a value = 20**  
**b value = 0**  
**DivideByZeroException was unhandled Attempt to divide by zero.**

**Explanation:**CLR terminates this program execution by throwing DivideByZeroException because the logical mistake we committed is dividing integer number by integer zero. As we know it is not possible to divide an integer number by zero.

From the above program, we can define the exception technically as “An exception is an event because when an exception is raised CLR internally executes some logic to prepare that exception related messages”.

The exception is a signal because by looking into exception message developer will take necessary actions against that exception. An exception is an object because for throwing exception CLR or we should create an appropriate class object.

**Is the above exception message is user understandable?**

Definitely, no, the user cannot understand the above exception message because they are .NET based exception messages. So the user cannot take further decision alone to resolve the above problem. The developer should guide to solve the above problem.

**What is the solution for the above problem?**

It is developer responsibility to convert .NET exception messages into user understandable message format. To solve this problem developer should write exception handling code in a .NET program. Using exception handling code, the developer can catch the exception and can print and pass user understandable messages.

**What is Exception Handling in C#?**

This is one of the frequently asked Exception Handling Interview Questions in C#.

The process of catching the exception for converting CLR given exception message to end-user understandable message or for stopping the abnormal termination of the program whenever runtime errors are occurring is called exception handling. Once we handle an exception under a program we will be getting following advantages

1. We can stop the abnormal termination
2. We can perform any corrective action that may resolve the problem occurring due to abnormal termination.
3. Displaying a user-friendly error message, so that the client can resolve the problem provided if it is under his control.

**Why we need exception handling in C#?**

In projects, an exception is handled

1. To stop the abnormal termination of the program
2. To provide user understandable messages when an exception is raised. So that users can make the decision without the help of developers.

Basically, by implementing Exception handling we are providing life to a program to talk to the user on behalf of a developer.

**What is the Exception Handling Procedure?**

Exception handling is a 4 steps procedure

1. Preparing the exception objects appropriate to the current logical mistake.
2. Throwing that exception to the appropriate exception handler.
3. Catching that exception
4. Taking necessary actions against that exception

**How can we handle the exception in .NET?**

There are three methods to handle the exception in .NET

1. Logical implementation
2. Try catch implementation
3. On error go to implementation

**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.

**What is the difference between System exceptions and Application exceptions?**

System exceptions are derived directly from a base class System.SystemException. A System-level Exception is normally thrown when a nonrecoverable error has occurred.

Application exceptions can be user-defined exceptions thrown by the applications. If you are designing an application that needs to create its own exceptions class, you are advised to derive custom exceptions from the System.ApplicationException class. It is typically thrown when a recoverable error has occurred.

**What is the logical implementation?**

In this method, we handle the exception by using logical statements. In real-time programming, the first and foremost important always given to logical implementation. If it is not possible to handle an exception using logical implementation then we use try-catch implementation.

**Program to handle an exception using logical implementation.**

**namespace** *ExceptionHandlingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

b = **int**.Parse**(**Console.ReadLine**())**;

**int** a, b, c;

Console.WriteLine**(**"ENTER ANY TWO NUBERS"**)**;

a = **int**.Parse**(**Console.ReadLine**())**;

**if** **(**b == 0**)**

**{**

Console.WriteLine**(**"second number should not be zero"**)**;

**}**

**else**

**{**

c = a / b;

Console.WriteLine**(**"C VALUE = " + c**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

In the above example when the user entered the second number as zero exception will be raised and that is handled using logical implementation.

But while we are entering two numbers instead of the number if we entered any character then it will give you one exception that FormatException which is not handled in this program. So to handle such type of exception we need to go for Try catch implementation.

**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.

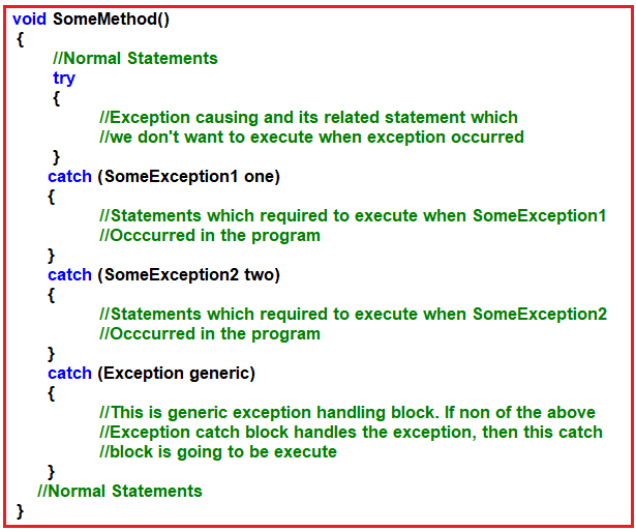
**Catch:**The catch block is used to catch the exception that is thrown from its corresponding try block. It has logic to take necessary actions on that caught exception.

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:**



**Once we enclose the code under try and catch blocks the execution takes place as following**

If all the statements under try are executed successfully from the last statement of trying the control directly jumps to the first statement that is present after the catch block (after all catch blocks) without executing catch block (it means there is no runtime error in the code at all ).

If any of the statements in the try causes an error from that statement without executing any other statements in try control directly jumps to the catch blocks searching for a catch block to handle that exception.

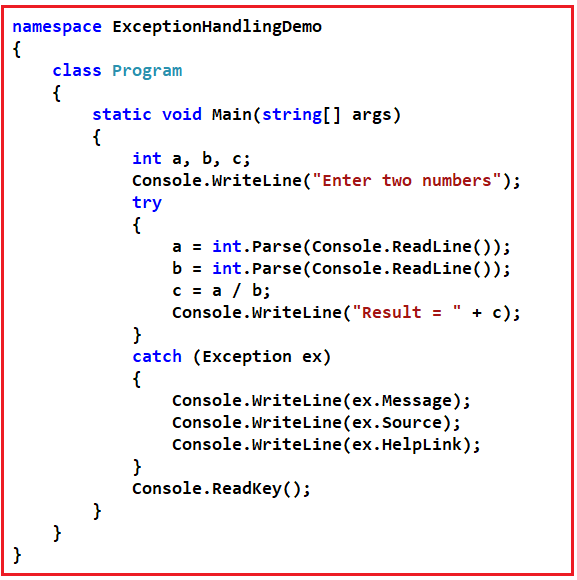
If a proper catch block is available that can handle the exception and abnormal termination stops there, executes the code under the catch block and from there again it jumps to the first statement after all the catch blocks. If a matching catch is not available abnormal termination occurs again.

**Explain the Properties of the Exception class.**

The exception class has 3 properties are as follows

1. **Message**: This property will store about the reason why the exception has occurred.
2. **Source**: This store name of the application from where the exception has been raised.
3. **Helplink**: This is used to provide a link to any file /URL to give helpful information to the user when an exception is raised.

**Example to show the use of Exception (superclass of all exception classes) class as an argument in catch blocks.**



In the above example, the superclass exception is used to handle the exception. But if we use super of any exception class when there is any relevant class is available, it will kill the execution performance of the program. So any time doesn’t use the superclass of an exception class to handle an exception when there is a relevant exception class is available.

**Can we catch all exceptions using a single catch block?**

Yes, we can catch all exceptions with a single catch block with parameter “Exception”. We should use this catch block only for stopping abnormal termination irrespective of the exceptions thrown from its corresponding try block.

It is always recommended to write catch blocks with exception parameters even though we are writing multiple catch blocks. It acts as a backup catch block.

**When should we write multiple catch blocks for a single try block?**

We should write multiple catch blocks for a single try block because of the following reasons

1. To print message specific to an exception or
2. To execute some logic specific to an exception

**Explain about the finally block.**

Finally establishes a block that definitely executes statements placed in it. The Statements which 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 the exception by catching or not catching.

**Why we need finally block in the real-time project?**

As per coding standard in finally block we should write resource releasing logic or clean up the code. Resource releasing logic means un-referencing objects those are created in the try block. For example, in real time projects, we create ADO.NET objects in the try block and at the end of the try block, we must close these objects.

Since the statements written in try and catch block are not guaranteed to be executed we must place them in finally block. For example, if we want to close ADO objects such as Connection object, Command object, etc. we must call the close() method in both try as well as in catch block to guarantee its execution.

Instead of placing the same close() method call statements in multiple places if we write it in finally block it is always executed irrespective of the exception raised or not raised.

**How many ways we can use try catch and finally?**

In can use in three different combinations

1. **Try and catch:** In this case, execution will be handled and stopping the abnormal termination.
2. **The Try, catch and finally:** In this case also exception will be handled stopping the abnormal termination along with the statements that are placed within the finally block gets executed at any cost.
3. **Try and finally:** In this case abnormal will not stop when a runtime error occurs because exceptions are not handled but even if the abnormal termination occurs also finally blocks get executed.

**What happens if the finally block throws an exception?**

The exception propagates up and should be handled at a higher level. If the exception is not handled at a higher level, the application crashes. The “finally” block execution stops at the point where the exception is thrown.

In the example below, notice that the **“finally”**block in the **“Hello()”**method throws an exception. **Hello()**method is being called in the **Main()** method and we don’t have any exception handling mechanism in place in the Main() method. So, the application crashes with the exception.

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Hello**()**;

**}**

**public** **static** **void** Hello**()**

**{**

**try**

**{**

// Some code

**}**

**catch**

**{**

// Some code

**}**

**finally**

**{**

Console.WriteLine**(**"This line will be executed"**)**;

**int** result = Convert.ToInt32**(**"TEN"**)**;

Console.WriteLine**(**"This line will NOT be executed"**)**;

**}**

**}**

**}**

On the other hand, if you include exception handling mechanism(try/catch) in the **Main()**method, then you will have the opportunity to handle the exception.

**public** **static** **void** Main**()**

**{**

**try**

**{**

Hello**()**;

**}**

**catch** **(**Exception ex**)**

**{**

// Process and log the exception

Console.WriteLine**(**ex.Message**)**;

**}**

**}**

**Irrespective of whether there is an exception or not “finally” block is guaranteed to execute.**

1. If the **“finally”**block is being executed after an exception has occurred in the try block,
2. and if that exception is not handled
3. and if the finally block throws an exception

**Then the original exception that occurred in the try block is lost.**

Here is an example:

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

**try**

**{**

Hello**()**;

**}**

**catch** **(**Exception ex**)**

**{**

Console.WriteLine**(**ex.Message**)**;

**}**

**}**

**public** **static** **void** Hello**()**

**{**

**try**

**{**

// This exception will be lost

**throw** new Exception**(**"Exception in TRY block"**)**;

**}**

**finally**

**{**

**throw** new Exception**(**"Exception in FINALLY block"**)**;

**}**

**}**

**}**

**What is the difference between the “throw” and “throw ex” in .NET?**

throw re-throws the exception that was caught, and preserves the stack trace. throw ex-throws the same exception, but resets the stack trace to that method. Unless you want to reset the stack trace (i.e. to shield public callers from the internal workings of your library), the throw is generally the better choice, since you can see where the exception originated.

Throw Syntax:

**try**

**{**

// do some operation that can fail

**}**

**catch** **(**Exception ex**)**

**{**

// do some local cleanup

**throw**;

**}**

//Throw ex Syntax:

**try**

**{**

// do some operation that can fail

**}**

**catch** **(**Exception ex**)**

**{**

// do some local cleanup

**throw** ex;

**}**

**What is Inner Exception?**

The InnerException property returns the Exception instance (original exception) that caused the current exception.

To look at the **inner exception**, we have to make this program cause an exception to fail. To do that you have 3 options

1. Enter a Character instead of a number (Causes Format Exception)
2. Or Enter a very big number that an integer cannot hold (Causes Over Flow Exception)
3. Or Enter Zero for Second Number (Causes Divide By Zero Exception)

**class** ExceptionHandling

**{**

**public** **static** **void** Main**()**

**{**

**try**

**{**

**try**

**{**

Console.WriteLine**(**"Enter First Number"**)**;

**int** FN = Convert.ToInt32**(**Console.ReadLine**())**;

Console.WriteLine**(**"Enter Second Number"**)**;

**int** SN = Convert.ToInt32**(**Console.ReadLine**())**;

**int** Result = FN / SN;

Console.WriteLine**(**"Result = {0}", Result**)**;

**}**

**catch** **(**Exception ex**)**

**{**

string filePath = @"C:\Sample Files\Log.txt";

**if** **(**File.Exists**(**filePath**))**

**{**

StreamWriter sw = new StreamWriter**(**filePath**)**;

sw.Write**(**ex.GetType**()**.Name + ex.Message + ex.StackTrace**)**;

sw.Close**()**;

Console.WriteLine**(**"There is a problem! Plese try later"**)**;

**}**

**else**

**{**

//To retain the original exception pass it as a parameter

//to the constructor, of the current exception

**throw** new FileNotFoundException**(**filePath + " Does not Exist", ex**)**;

**}**

**}**

**}**

**catch** **(**Exception ex**)**

**{**

//ex.Message will give the current exception message

Console.WriteLine**(**"Current or Outer Exception = " + ex.Message**)**;

//Check if inner exception is not null before accessing Message property

//else, you may get Null Reference Excception

**if** **(**ex.InnerException != **null)**

**{**

Console.WriteLine**(**"Inner Exception = ", ex.InnerException.Message**)**;

**}**

**}**

**}**

**}**