**C# 4.0 New Features**

**What are the new features introduced in c# 4.0?**  
  
This is very commonly asked **c# interview question**. This question is basically asked to check, if you are passionate about catching up with latest technological advancements. The list below shows a few of the new features introduced in c# 4.0. If you are aware of any other new features, please submit those using the from at the end of this post.  
  
**1.** Optional and Named Parameters  
**2.** COM Interoperability Enhancements  
**3.** Covariance and Contravariance  
**4.** Dynamic Type Introduction

**Explicit Interface Implementation**

This C# Interview Question was asked my a member of this blog. Please refer to the example code below.  
  
How to implement the void Method() of the interface in the following case ?  
  
class B   
{  
   public void Method()   
   {  
       // some code here  
       //...  
   }  
}  
  
**Youtube video on Explicit Interface Implementation**  
   
  
interface I   
{  
   void Method();  
}  
  
class D : B, I   
{  
   // how to implement the void Method() of the interface  
   // public void I.Method() { ... }  
}   
  
To implement void Method, we use explicit interface implementation technique as shown below.  
  
using System;  
namespace SampleConsole  
{  
class Program  
{  
   static void Main()  
   {  
      //To Call Class B Method  
      D d = new D();  
      d.Method();  
  
      //To Call the Interface Method  
      I i = new D();  
      i.Method();  
  
      //Another way to call Interface method  
      ((I)d).Method();  
   }  
}  
class B  
{  
   public void Method()  
   {  
      Console.WriteLine("Void Method - B");  
   }  
}  
interface I  
{  
   void Method();  
}  
class D : B, I  
{  
   void I.Method()  
   {  
      Console.WriteLine("Void Method - I");  
   }  
}   
}

**Difference between EXE and DLL**

1. .EXE is an executable file and can run by itself as an application, where as .DLL is usullay consumed by a .EXE or by another .DLL and we cannot run or execute .DLL directly.  
  
2. For example, In .NET, compiling a Console Application or a Windows Application generates .EXE, where as compiling a Class Library Project or an ASP.NET web application generates .DLL. In .NET framework, both .EXE and .DLL are called as assemblies.  
  
3. .EXE stands for executable, and .DLL stands for Dynamic Link Library  
  
If you are aware of any other differences please post them using the form below.

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

**Why C# does not support multiple class inheritance?**  
or  
**What are the problems of 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 with an example.

|  |
| --- |
| **Here is a youtube video on Problems of multiple class inheritance**   http://3.bp.blogspot.com/-pDOmliP7miA/Td6r5G94IsI/AAAAAAAAAGc/S69vr0wPUSc/s1600/Multiple+Class+Inheritance.png |

**As shown in the image above:**  
**1.** I have 2 classes - **ClassB** and **ClassC**  
**2.** Both of these classes inherit from **ClassA**  
**3.** Now, we have another class, **ClassD** which inherits from both **ClassB** and **ClassC**  
  
So, if a method in **ClassD** calls a method defined in **ClassA** and **ClassD**has not overriden the invoked method. But both **ClassB**and **ClassC**have overridden the same method differently. Now, the ambiguity is, from which class does, **ClassD**inherit the invoked method: **ClassB**, or **ClassC**?  
  
**In order not to have these problems, C# does not support multiple class inheritance.**

**What are the difference between interfaces and abstract classes**

**There are several differences between an abstract class and an interface as listed below.**  
   
**Here is a youtube video on difference between abstract classes and interfaces**  
  
**1.** Abstract classes can have implementations for some of its members, but the interface can't have implementation for any of its members.  
  
**2.** Interfaces cannot have fields where as an abstract class can have fields.  
  
**3.** An interface can inherit from another interface only and cannot inherit from an abstract class, where as an abstract class can inherit from another abstract class or another interface.  
  
**4.** A class can inherit from multiple interfaces at the same time, where as a class cannot inherit from multiple classes at the same time.  
  
**5.** Abstract class members can have access modifiers where as interface members cannot have access modifiers.  
  
**Another common C# Interview Question, that is commonly asked is, When do you choose interface over an abstract class or vice versa?**  
A general rule of thumb is, If you have an implementation that will be the same for all the derived classes, then it is better to go for an abstract class instead of an interface. So, when you have an interface, you can move your implementation to any class that implements the interface. Where as, when you have an abstract class, you can share implementation for all derived classes in one central place, and avoid code duplication in derived classes.

**Where did use delegates in your project - Part 1**

**Where did you use delegates in your project?**  
or  
**How did you use delegates in your project?**  
or  
**Usage of delegates in a Real Time Project?**  
  
**This is a very common c sharp interview question**. Delegates 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** http://4.bp.blogspot.com/-adgPsWbpsUg/Td1gZljJpvI/AAAAAAAAAGM/be0NdNwxofI/s1600/EmployeeClass.png |

**Delegates Example Video - Part 1**  
   
[**Click here for Delegates Example Video - Part 2**](http://csharp-video-tutorials.blogspot.com/2012/06/part-38-c-tutorial-delegates-example-ii.html)  
The **Employee** class has the following properties.  
**1. Id**  
**2. Name**  
**3. Experience**  
**4. Salary**  
  
Now, I want you 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 hard coded. At times, we may promote employees **based on their experience** and at times we may promote them**based on their salary** or may be **some other condition**. So, the logic to promote employees **should not be hard coded with in the method**.  
  
To achieve this, we can make use of **delegates**. So, now I would design my class as shown below. We also, created a **delegate EligibleToPromote**. This delegate takes **Employee** object as a parameter and **returns a boolean**. In the **Employee** class, we have **PromoteEmpoloyee** method. This method takes in a list of Employees and a Delegate of type **EligibleToPromote** as parameters. The method, then loops thru each employee object, and passes it to the delegate. If the delegate returns true, then them Employee is **promoted**, else not promoted. So, with in the method **we have not hard coded any logic on how we want to promote employees**.

|  |
| --- |
| http://2.bp.blogspot.com/-3to5riGc2kY/Td1hJs_6yzI/AAAAAAAAAGQ/_tcxn_FPbYM/s1600/DelegatesUsage.png |

**Where did use delegates in your project - Part 2**

[In Part 1, we have seen how to create our Employee class. If you have not read Part 1, please do so before reading this part.](http://venkatcsharpinterview.blogspot.com/2011/05/where-did-use-delegates-in-your-project.html)  
  
So now, the client who uses the **Employee** class has the flexibility of determining the logic on **how they want to promote their employees** a 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.  
  
  
**Delegates Example Video - Part 2**  
   
[**Click here for Delegates Example Video - Part 1**](http://csharp-video-tutorials.blogspot.com/2012/06/part-37-c-tutorial-delegates-example-i.html)  
Notice, the **private Promote method** that we have created. This method has the logic on 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 **EligibleToPromote** delegate. This is very important, because **Promote method cannot be passed as a parameter to the delegate if the signature differs**. This is the reason why delegates are called as type safe function pointers.

|  |
| --- |
| http://3.bp.blogspot.com/--kxdxt9E_08/Td1tSmptWgI/AAAAAAAAAGU/yb3q1RFr5lo/s1600/Jobs.png |

So if we did not have the concept of delegates, it would not have been possible to pass a function as a parameter. As 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.  
  
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 Lambda expressions is shown below. Private Promote method is no longer required now.

|  |
| --- |
| http://2.bp.blogspot.com/-0zGvUux4LAc/Td1tnfgPaPI/AAAAAAAAAGY/j15XIOTmWXc/s1600/Lambda+Expressions.png |

**The output of the above program should be as below.**  
Employee Test 2 Promoted  
Employee Test 3 Promoted

**C# Interview Questions on Delegates**

**Youtube video on Delegates**  
   
[Click here for Video on Multicast Delegates](http://csharp-video-tutorials.blogspot.com/2012/06/part-39-c-tutorial-multicast-delegates.html)  
**What is a delegate?**  
A **delegate** is a type safe function pointer. Using delegates you can pass methods as parameters. To pass a method as a parameter, to a delegate, the signature of the method must match the signature of the delegate. This is why, delegates are called type safe function pointers.  
  
**What is the main use of delegates in C#?**  
Delegates are mainly used to define call back methods.  
  
**What do you mean by chaining delegates?**  
Or  
**What is a multicast delegate?**  
The capability of calling multiple methods on a single event is called as chaining delegates. Let me give you an example to understand this further.  
**1.** Create a new asp.net web application  
**2.** Drag and drop a button control and leave the ID as Button1.   
**3.** On the code behind file, add the code shown below.

|  |
| --- |
| http://1.bp.blogspot.com/-xMM0HEBoJuw/Tdwdd6GlOBI/AAAAAAAAAGE/BnIVkuIvSGI/s1600/Delegates.png |

When you click the Button now, both Method1 and Method2 will be executed. So, this capability of calling multiple methods on a single event is called as chaining delegates. In the example, we are using EventHandler delegate, to hook up Method1 and Method2 to the click event of the button control. Since, the EventHandler delegate is now pointing to multiple methods, it is also called as multicast delegate.  
  
**Will the following code compile?**

|  |
| --- |
| http://2.bp.blogspot.com/-G0LapeMmn_s/TdweDuU3wCI/AAAAAAAAAGI/oFqF0N5qBHQ/s1600/Delegates+Example.png |

No, the code does not compile. For the code to compile, the signature of Method1 should match the signature of SampleDelegate.

**What are the advantages of using interfaces**

|  |
| --- |
| Interfaces are very powerful. If properly used, **interfaces provide all the advantages** as listed below.   **1.** Interfaces allow us to implement polymorphic behaviour. Of course, abstract classes can also be used to implement polymorphic behaviour.  **2.** Interfaces allow us to develop very loosely coupled systems. |

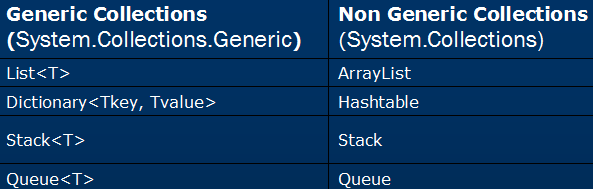
**3.** Interfaces enable mocking for better unit testing.  
  
**4.** Interfaces enables us to implement multiple class inheritance in C#.  
  
**5.** Interfaces are great for implementing Inverson of Control or Dependancy Injection.  
  
**6.** Interfaces enable parallel application development.   
  
**If you can think of any other advantages of using interfaces, or if you can add more explanation, please feel free to do so using the form below.**  
  
[For basic interview questions on interfaces, you can read this article.](http://venkatcsharpinterview.blogspot.com/2009/07/c-interview-questions-related-to.html)

**C# Interview Questions related to Interfaces.**

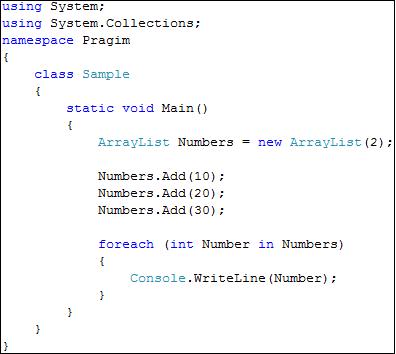
**Explain what is an Interface in C#?**  
An Interface in C# is created using the interface keyword. An example is shown below.  
  
using System;  
namespace Interfaces  
{  
interface IBankCustomer  
{  
void DepositMoney();  
void WithdrawMoney();  
}  
public class Demo : IBankCustomer  
{  
public void DepositMoney()  
{  
Console.WriteLine("Deposit Money");  
}  
  
public void WithdrawMoney()  
{  
Console.WriteLine("Withdraw Money");  
}  
  
public static void Main()  
{  
Demo DemoObject = new Demo();  
DemoObject.DepositMoney();  
DemoObject.WithdrawMoney();  
}  
}  
}  
   
**Interfaces Video**  
   
[**Click here for video on Explicit Interface Implementation**](http://csharp-video-tutorials.blogspot.com/2012/06/part-31-c-tutorial-explicit-interface.html)  
**In our example we created IBankCustomer interface. The interface declares 2 methods.**  
**1.** void DepositMoney();  
**2.** void WithdrawMoney();  
  
Notice that method declarations does not have access modifiers like public, private, etc. By default all interface members are public. It is a compile time error to use access modifiers on interface member declarations. Also notice that the interface methods have only declarations and not implementation. It is a compile time error to provide implementation for any interface member. In our example as the Demo class is inherited from the IBankCustomer interface, the Demo class has to provide the implementation for both the methods (WithdrawMoney() and DepositMoney()) that is inherited from the interface. If the class fails to provide implementation for any of the inherited interface member, a compile time error will be generated. Interfaces can consist of methods, properties, events, indexers, or any combination of those four member types. When a class or a struct inherits an interface, the class or struct must provide implementation for all of the members declared in the interface. The interface itself provides no functionality that a class or struct can inherit in the way that base class functionality can be inherited. However, if a base class implements an interface, the derived class inherits that implementation.  
  
**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.  
**1.** A class or a struct can inherit from more than one interface at the same time where as A class or a struct cannot inherit from more than one class at the same time. An example depicting the same is shown below.  
  
using System;  
namespace Interfaces  
{  
interface Interface1  
{  
void Interface1Method();  
}  
interface Interface2  
{  
void Interface2Method();  
}  
class BaseClass1  
{  
public void BaseClass1Method()  
{  
Console.WriteLine("BaseClass1 Method");  
}  
}  
class BaseClass2  
{  
public void BaseClass2Method()  
{  
Console.WriteLine("BaseClass2 Method");  
}  
}  
  
//Error : A class cannot inherit from more than one class at the same time  
//class DerivedClass : BaseClass1, BaseClass2  
//{  
//}  
  
//A class can inherit from more than one interface at the same time  
public class Demo : Interface1, Interface2  
{  
public void Interface1Method()  
{  
Console.WriteLine("Interface1 Method");  
}  
  
public void Interface2Method()  
{  
Console.WriteLine("Interface2 Method");  
}  
  
public static void Main()  
{  
Demo DemoObject = new Demo();  
DemoObject.Interface1Method();  
DemoObject.Interface2Method();  
}  
}  
}  
  
**2.** 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, you cannot create an instance of an interface.  
  
**If a class inherits an interface, what are the 2 options available for that class?**  
**Option 1:** Provide Implementation for all the members inheirted from the interface.  
  
namespace Interfaces  
{  
interface Interface1  
{  
void Interface1Method();  
}  
  
class BaseClass1 : Interface1  
{  
public void Interface1Method()  
{  
Console.WriteLine("Interface1 Method");  
}  
public void BaseClass1Method()  
{  
Console.WriteLine("BaseClass1 Method");  
}  
}  
}  
  
**Option 2:** If the class does not wish to provide Implementation for all the members inheirted from the interface, then the class has to be marked as abstract.  
  
namespace Interfaces  
{  
interface Interface1  
{  
void Interface1Method();  
}  
  
abstract class BaseClass1 : Interface1  
{  
abstract public void Interface1Method();  
public void BaseClass1Method()  
{  
Console.WriteLine("BaseClass1 Method");  
}  
}  
}  
  
**A class inherits from 2 interfaces and both the interfaces have the same method name as shown below. How should the class implement the drive method for both Car and Bus interface?**  
namespace Interfaces  
{  
interface Car  
{  
void Drive();  
}  
interface Bus  
{  
void Drive();  
}  
  
class Demo : Car,Bus  
{  
//How to implement the Drive() Method inherited from Bus and Car  
}  
}  
  
To implement the Drive() method use the fully qualified name as shown in the example below. To call the respective interface drive method type cast the demo object to the respective interface and then call the drive method.  
  
using System;  
namespace Interfaces  
{  
interface Car  
{  
void Drive();  
}  
interface Bus  
{  
void Drive();  
}  
  
class Demo : Car,Bus  
{  
void Car.Drive()  
{  
Console.WriteLine("Drive Car");  
}  
void Bus.Drive()  
{  
Console.WriteLine("Drive Bus");  
}  
  
static void Main()  
{  
Demo DemoObject = new Demo();  
((Car)DemoObject).Drive();  
((Bus)DemoObject).Drive();  
}  
}  
}  
  
**What do you mean by "Explicitly Implemeting 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 Implemeting an Interface member". The disadvantage of Explicitly Implemeting an Interface member is that, the class object has to be type casted to the interface type to invoke the interface member. An example is shown below.  
  
using System;  
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();  
}  
}  
}

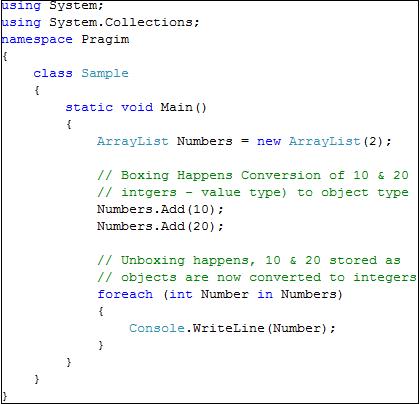
**Advantages and disadvantages of using generics in C#**

**Generics Video Tutorial**  
   
To better understand the advantages and disadvantages of generics, it is better you read the below 2 articles first.  
[1. Click here to read about Advantages and disadvantages of Arrays](http://venkatcsharpinterview.blogspot.com/2011/03/what-are-advantages-and-disadvantages.html)  
[2. Click here to read about Advantages and disadvantages of System.Collections](http://venkatcsharpinterview.blogspot.com/2011/03/what-are-advantages-and-disadvantages_18.html)  
  
  
In Microsoft.NET version 1.0 there were collections, such as the ArrayList for working with groups of objects. An ArrayList is much like an array, except it could automatically grow and offered many convenience methods that arrays don't have. The problem with ArrayList and all the other .NET v1.0 collections is that they operate on type object. Since all objects derive from the object type, you can assign anything to an ArrayList. The problem with this is that you incur performance overhead converting value type objects to and from the object type and a single ArrayList could accidentally hold different types, which would cause a hard to find errors at runtime because you wrote code to work with one type. Generic collections fix these problems.  
  
  
A generic collection is strongly typed (type safe), meaning that you can only put one type of object into it. This eliminates type mismatches at runtime. Another benefit of type safety is that performance is better with value type objects because they don't incur overhead of being converted to and from type object. With generic collections, you have the best of all worlds because they are strongly typed, like arrays, and you have the additional functionality, like ArrayList and other non-generic collections, without the problems.  
  
  
It is always good to use generics rather than using ArrayList,Hashtable etc, found in System.Collections namespace. The only reason why you may want to use System.Collections is for backward compatibility.  
  
**I cannot think of any disadvantages of using generics at the moment. Please feel free to comment if you are aware of any disadvantages.**  
  
The screen shot below shows, the generics collection classes and their respective non generic counterparts.

[](http://4.bp.blogspot.com/-aUqn8qPD0lI/TcMOlgHgsmI/AAAAAAAAAEk/-jcmT9lGvNo/s1600/Generics.png)

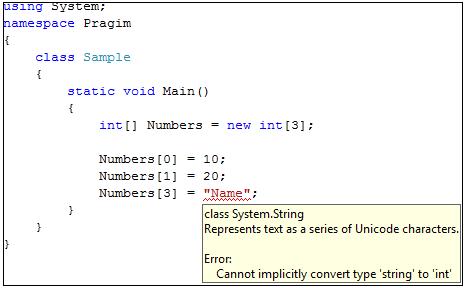
**What are the advantages and disadvantages of using collection classes present in System.Collections namespace**

We will understand the **advantages and disadvantages**of using collection classes present in System.Collections namespace, using ArrayList collection class. The **same advantages and disadvantages** apply to all other collection classes like **Stack, Queue and Hashtable classes**.  
  
  
**Advantages of using ArrayList:**  
**1.** ArrayList can grow in size dynamcally. In the **example below**, the Numbers ArrayList initial size is set 2. But we have added 3 elements. This proves that ArrayList, and the rest of the collection classes like Stack, Queue and Hashtable can grow in size dynamically. If Numbers, was an integer array, then we would have run into Index Out of Range compiler error.  
  
[](https://lh3.googleusercontent.com/-h-3Kn7oLbpM/TYPtPrvdkKI/AAAAAAAAAEY/dpMyimin5eg/s1600/ArrayList+Grows+in+size+dynamically.jpg)  
  
  
**2.** ArrayList provide several convinient methods to add and remove elements to the collection. You can use the Add(), Remove() etc which are very handy to add and remove elements respectively. Similarly the Stack, Queue and Hashtable classes have thier respective methods, to add or remove the elements.  
  
**Disadvantages of using ArrayList:**  
ArrayList and all other collection classes like stack, queue and hashtable which are present in System.Collection namespace operate on object and hence are loosely typed. The loosely typed nature of these collections make them vulnerable to runtime errors. [Click here for an example on how the loosely typed nature of an ArrayList can cause runtime erros.](http://venkatcsharpinterview.blogspot.com/2011/03/what-are-advantages-and-disadvantages.html)   
  
Loosley typed collections can also cause performance overhead, because boxing and unboxing happens. In the example below, Numbers is an arraylist. We are stroing 10 and 20 which are integers and value types. Since, arraylist operate on object type, and object type is a   
  
reference type, the value 10 is boxed and converted into a reference type. The same is the case with integer 20. If we store 100 integers in the arraylist. All the 100 intgers are boxed, meaning converted into reference types and then stored in the collection.  
  
When we try to retrieve the elements out of the collection, we covert the object type back to integer type, unboxing happens. So this unnecessary boxing and unboxing happens behind the scenes everytime we add and remove value types to the collection classes present in System.Collections namespace. This can severly affect the performance, especially if your collections are large. To solve this problem, we have generics introduced in dotnet. [Click here for the advantages and disadvantages of using generics.](http://venkatcsharpinterview.blogspot.com/2011/05/advantages-and-disadvantages-of-using.html)

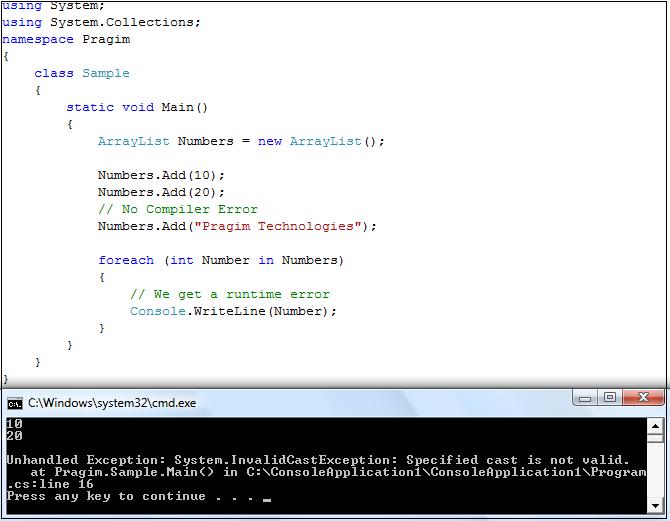
[](https://lh6.googleusercontent.com/-ebn8dGu3Lt4/TYPwQDe6-WI/AAAAAAAAAEc/NO4XnOpJcPw/s1600/Boxing+and+Unboxing.jpg)

**What are the advantages and disadvantages of using arrays**

**Advantages of using arrays:**  
**1.** Arrays are **strongly typed**, meaning you can only have one type of elements in the array. The strongly typed nature of arrays gives us 2 advantages.**One**, the performance will be much better because **boxing and unboxing**will not happen. **Second**, run time errors can be prevented because of type mis matches. Type mis matches and runtime errors are most commonly seen with collection classes like ArrayList, Queue, Stack etc, that are present in System.Collections namespace.   
  
**In the example below**, Numbers is an integer array. When we try to store a string in the integer array, a **compiler error**is reported stating cannot implicitly convert string to integer. This is why we call arrays are strongly typed.

[](https://lh6.googleusercontent.com/-x_RnG_rBFUo/TYPqdWR5jWI/AAAAAAAAAEQ/hIQkJmvBMpc/s1600/Strongly+Typed+Arrays.jpg)

**In the example below**, Numbers is an ArrayList. Collections of type arraylist are loosely typed. This means any type of elements can be added to the collection. ArrayList operate on object type, which makes them loosely typed. No compiler error is reported, but when we run the application, a runtime error is reported as shown. In software development, it is always better to catch errors at compile time rather than at runtime.

[](https://lh5.googleusercontent.com/-Wp4U0V8qQp8/TYPr4TfsxXI/AAAAAAAAAEU/Iljwvj-zLuM/s1600/Loosely+Typed+ArrayList.jpg)

**Disadvantages of using arrays:**  
**1.** Arrays are fixed in size and cannot grow over time, where ArrayList in System.Collections namespace can grow dynamically.  
**2.** Arrays are zero index based, and hence a little difficult to work with. The only way to store or retrieve elements from arrays, is to use integral index. Arrays donot provide convinient methods like Add(), Remove() etc provided by collection classes found in System.Collections or System.Collections.Generics namespaces, which are very easy to work with.  
  
  
[Another common C# interview question, that is asked is, what are the advantages and disadvantages of using collection classes present in System.Collections namespace. Click here to learn about the advantages and disadvantages of using Systen.Collections namespace classes.](http://venkatcsharpinterview.blogspot.com/2011/03/what-are-advantages-and-disadvantages_18.html)

**Remoting - Interview Questions**

**What is .NET Remoting?**  
.NET Remoting allows objects to interact with one another across application domains.  
  
  
**What are the 2 message encoding formats supported by .NET Remoting and when do you choose one over the other?**  
**Message Encoding Formats:**  
1. Binary encoding.  
2. XML encoding.  
  
  
Applications can use binary encoding where performance is critical, or XML encoding where interoperability with other remoting frameworks is essential.  
  
  
**What are the two types of .NET remote objects?**  
**1. Client-activated objects**- Client-activated objects are under the control of a lease-based lifetime manager that ensures that the object is garbage collected when its lease expires.  
**2. Server-activated objects** - In the case of server-activated objects, developers have a choice of selecting either a "single call" or "singleton" model. The lifetime of singletons are also controlled by lease-based lifetime.  
  
  
**What is considered as Remote Object?**  
Any object outside the application domain of the calling appication is considered remote object, even if the objects are executing on the same machine.  
  
**Can you treat every object as a remote object?**  
Objects that cannot be serialized cannot be passed to a different application domain and are therefore nonremotable.  
  
  
**What are the ways in which an object can be serialized?**  
1. Mark your class with serializable attribute.  
2. Make your class implement ISerializable interface.  
  
  
**How can you change an object into a remote object?**  
Any object can be changed into a remote object by deriving it from MarshalByRefObject.   
  
  
**What happens when a client activates a remote object?**  
When a client activates a remote object, it receives a proxy to the remote object. All operations on this proxy are appropriately indirected to enable the remoting infrastructure to intercept and forward the calls appropriately.  
  
  
**What are proxy objects and what is the use of these proxy objects?**  
Proxy objects are created when a client activates a remote object. The proxy object acts as a representative of the remote object and ensures that all calls made on the proxy are forwarded to the correct remote object instance.

**C# Interview Questions on partial classes, structs and methods.**

**What is a partial class. Give an example?**  
A **partial class** 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. 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 the parts uses partial keyword and public access modifier.  
  
using System;  
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.** All the parts must be available at compile time to form the final class.  
**3.** All the parts must have the same access modifiers - public, private, protected etc.  
**4.** Any class members declared in a partial definition are available to all the other parts.   
**5.** The final class is the combination of all the parts at compile time.  
  
**What are the advantages of using partial classes?**  
**1.** When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.  
  
**2.** When working with automatically generated source, 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?**  
Yes, it is possible to create partial structs, interfaces and methods. We can create partial structs, interfaces and methods the same way as we create partial classes.  
  
**Will the following code compile?**  
using System;  
namespace PartialClass  
{  
  public partial class Student  
  {  
    public void Study()  
    {  
      Console.WriteLine("I am studying");  
    }  
  }  
  public abstract partial class Student  
  {  
    public void Play()  
    {  
      Console.WriteLine("I am Playing");  
    }  
  }  
  public class Demo  
  {  
    public static void Main()  
    {  
      Student StudentObject = new Student();  
    }  
  }  
}  
  
No, a compile time error will be generated stating "Cannot create an instance of the abstract class or interface "PartialClass.Student". This is because, if any part is declared abstract, then the whole class becomes abstract. Similarly if any part is declared sealed, then the whole class becomes sealed and if any part declares a base class, then the whole class inherits that base class.  
  
**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 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.  
  
class ContainerClass  
{  
  public partial class Nested  
  {  
    void Test1() { }  
  }  
  public partial class Nested  
  {  
    void Test2() { }  
  }  
}  
  
**How do you create partial methods?**  
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. The definition, and the implementation. These may be in separate parts of a partial class, 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 partial keyword.  
**2.** The return type of a partial method must be void.  
**3.** Partial methods can have ref but not out parameters.  
**4.** 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.

**Nested Types in C#**

**What is a nested type. 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 as nested class.  
  
using System;  
namespace Nested  
{  
  class ContainerClass  
  {  
    class InnerClass  
    {  
      public string str = "A string variable in nested class";  
    }  
  
    public static void Main()  
    {  
      InnerClass nestedClassObj = new InnerClass();  
      Console.WriteLine(nestedClassObj.str);  
    }  
  }  
}  
  
**Will the following code compile?**  
using System;  
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 bcos 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.  
  
using System;  
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.  
  
using System;  
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 ouput of the following program?**  
using System;  
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:**  
I am a container class  
I am an inner class  
I am a Demo class  
  
The above program has used the concepts of inheritance and nested classes. The ContainerClass is at the top in the inheritance chain. The nested InnerClass derives from outer ContainerClass. Finally the DemoClass derives from nested InnerClass. As all the 3 classes are related by inheritance we have the above output.

**C# Interview Questions on Destructors**

**What is a 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 garbage collector to run?**  
Yes, it possible to force garbage collector to run by calling the Collect() method, but this is not considered a good practice because this might create a performance over head. 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 expensive external resource, it is recommend to explicitly release 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.

### **What is a constructor in C#?** Constructor is a class method that is executed when an object of a class is created. Constructor has the same name as the class, and usually used to initialize the data members of the new object.  **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 reference types. Can structs have constructors?** Yes, even though Structs are not reference 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 as shown in the example below. using System; namespace TestConsole {   class Program   {     public static void Main()     {       //Error cannot create instance of a class with private constructor       SampleClass SC = new SampleClass();     }   }   class SampleClass   {     double PI = 3.141;     private SampleClass()     {     }   } } **Can a class or a struct have multiple constructors?** Yes, a class or a struct can have multiple constructors. Constructors in csharp 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. using System; namespace TestConsole {   class BaseClass   {     public BaseClass(string str)     {       Console.WriteLine(str);     }   }   class ChildClass : BaseClass   {     public ChildClass(string str): base(str)     {     }     public static void Main()     {       ChildClass CC = new ChildClass("Calling base class constructor from child class");     }   } } **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. using System; namespace TestConsole {   class BaseClass   {     public BaseClass()     {       Console.WriteLine("I am a base class constructor");     }   }   class ChildClass : BaseClass   {     public ChildClass()     {       Console.WriteLine("I am a child class constructor");     }     public static void Main()     {       ChildClass CC = new ChildClass();     }   } } **Will the following code compile?** using System; namespace TestConsole {   class BaseClass   {     public BaseClass(string str)     {       Console.WriteLine(str);     }   }   class ChildClass : BaseClass   {     public ChildClass()     {       Console.WriteLine("I am a child class constructor");     }     public static void Main()     {       ChildClass CC = new ChildClass();     }   } } No, the above code will not compile. This is because, if a base class does not offer a default constructor, the derived class must make an explicit call to a base class constructor by using the base keyword as shown in the example below. using System; namespace TestConsole {   class BaseClass   {     public BaseClass(string str)     {       Console.WriteLine(str);     }   }   class ChildClass : BaseClass   {     //Call the base class contructor from child class     public ChildClass() : base("A call to base class constructor")     {       Console.WriteLine("I am a child class constructor");     }     public static void Main()     {       ChildClass CC = new ChildClass();     }   } } **Can a class have static constructor?** Yes, a class can have 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. using System; namespace TestConsole {   class Program    {     static int I;     static Program()     {       I = 100;       Console.WriteLine("Static Constructor called");     }     public Program()     {       Console.WriteLine("Instance Constructor called");     }     public static void Main()     {       Program P = new Program();     }   } } **Can you mark static constructor with access modifiers?** No, we cannot use access modifiers on 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?** **1.** 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. **2.** Static constructors are also useful when creating wrapper classes for unmanaged code, when the constructor can call the LoadLibrary method.

### C# Interview Questions on Methods / Functions

**Methods Video Tutorial**  
   
[**Click here for video tutorial on Different Types of Method Parameters**](http://csharp-video-tutorials.blogspot.com/2012/06/part-17-c-tutorial-method-parameters.html)  
**Is the following code legal?**  
using System;  
namespace Demo  
{  
 class Program  
 {  
  public static void Main()  
  {  
  
  }  
  public void Sum(int FirstNumber, int SecondNumber)  
  {  
   int Result = FirstNumber + SecondNumber;  
  }  
  
  public int Sum(int FirstNumber, int SecondNumber)  
  {  
   int Result = FirstNumber + SecondNumber;  
  }  
 }  
}  
  
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 where as FN and LN 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.  
  
using System;  
namespace Demo  
{  
 class Program  
 {  
  public static void Main()  
  {  
   int FN = 10;  
   int SN = 20;  
   //FN and LN are method arguments  
   int Total = Sum(FN, SN);  
   Console.WriteLine(Total);  
  }  
  //FirstNumber and SecondNumber are method parameters  
  public static int Sum(int FirstNumber, int SecondNumber)  
  {  
   int Result = FirstNumber + SecondNumber;  
   return Result;  
  }  
 }  
}  
  
  
**Explain the difference between passing parameters by value and passing parameters by reference with an example?**  
We can pass parameters to a method by value or by reference. By default all value types are passed by value where as 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.  
  
using System;  
namespace Demo  
{  
 class Program  
 {  
  public static void Main()  
  {  
   int I = 10;  
   int K = Function(I);  
  
   Console.WriteLine("I = " + I);  
   Console.WriteLine("K = " + K);  
  }  
  public static int Function(int Number)  
  {  
   int ChangedValue = Number + 1;  
   return ChangedValue;  
  }  
 }  
}  
  
By default, reference types are passed by reference. When an object of a reference type is passed to a method, the reference 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.  
  
using System;  
namespace Demo  
{  
 class Program  
 {  
  public static void Main()  
  {  
   ReferenceTypeExample Object = new ReferenceTypeExample();  
   Object.Number = 20;  
   Console.WriteLine("Original Object Value = " + Object.Number);  
   Function(Object);  
   Console.WriteLine("Object Value after passed to the method= " + Object.Number);  
  }  
  public static void Function(ReferenceTypeExample ReferenceTypeObject)  
  {  
   ReferenceTypeObject.Number = ReferenceTypeObject.Number + 5;  
  }  
 }  
  
 class ReferenceTypeExample  
 {  
  public int Number;  
 }  
}  
  
**Can you pass value types by reference to a method?**  
Yes, we can pass value types by by reference to a method. An example is shown below.  
  
using System;  
namespace Demo  
{  
 class Program  
 {  
  public static void Main()  
  {  
   int I = 10;  
   Console.WriteLine("Value of I before passing to the method = " + I);  
   Function(ref I);  
   Console.WriteLine("Value of I after passing to the method by reference= " + I);  
  }  
  public static void Function(ref int Number)  
  {  
   Number = Number + 5;  
  }  
 }  
}  
  
**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.  
using System;  
namespace Demo  
{  
 class Program  
 {  
  public static void Main()  
  {  
   SayHi();  
  }  
  public static void SayHi()  
  {  
   Console.WriteLine("Hi");  
   return;  
   Console.WriteLine("This statement will never be executed");  
  }  
 }  
}

**C# Interview Questions on Properties**

**Why do we need Properties - Video**  
   
[**Click here for Video on Properties**](http://csharp-video-tutorials.blogspot.com/2012/06/part-27-c-tutorial-properties-in-c.html)  
**What are Properties in C#. Explain with an example?**  
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 which 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. Full Name property is readonly, 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.  
  
  
using System;  
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);  
   }  
}  
  
  
**Explain the 3 types of properties in C# with an example?**  
**1.** Read Only Properties: Properties without a set accessor are considered read-only. In the above example FullName is read only property.  
**2.** Write Only Properties: Properties without a get accessor are considered write-only. In the above example Country is write only property.  
**3.** Read Write Properties: Properties with both a get and set accessor are considered read-write properties. In the above example FirstName and LastName are read write properties.  
  
**What are the advantages of properties in C#?**  
**1.** Properties can validate data before allowing a change.  
**2.** Properties 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 an 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 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.  
  
using System;  
class Circle  
{  
   private static double \_pi = 3.14;  
   public static double PI  
   {  
      get  
      {  
         return \_pi;  
      }  
   }  
}  
class MainClass  
{  
   public static void Main()  
   {  
      Console.WriteLine(Circle.PI);  
   }  
}  
  
**What is a virtual property. Give an example?**  
A property that is marked with 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 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 over riden implementation. A property overriding a virtual property can also be sealed, specifying that for derived classes it is no longer virtual.  
  
  
using System;  
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  
{  
   // Overiding 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. Give an example?**  
A property that is marked with 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  
{  
   // Overiding 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.

**C# Interview Questions on Constants**

**What are constants in C#?**  
Constants in C# are immutable values which are known at compile time and do not change for the life of the program. 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 isdeclared. An example is shown below.  
  
using System;  
class Circle  
{  
   public const double PI = 3.14;  
   public Circle()  
   {  
      //Error : You can only assign a value to a constant field at the time of declaration  
      //PI = 3.15;  
   }  
}  
class MainClass  
{  
   public static void Main()  
   {  
      Console.WriteLine(Circle.PI);  
   }  
}  
  
**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 readonly 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 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 run time, const fields cannot be passed by reference.  
  
using System;  
class Circle  
{  
   public const double PI = 3.14;  
}  
  
**How do you access a constant field declared in a class?**  
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 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.  
  
using System;  
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);  
   }  
}

**C# Interview Questions on Fields**

**What are the 2 broad classifications of fields in C#?**  
**1.** Instance fields  
**2.** Static fields  
  
**What are 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?**  
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?**  
using System;  
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 "Static member 'Area.PI' cannot be accessed with an instance reference; qualify it with a type name instead". This is because 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.  
  
using System;  
class Area  
{  
   public static double PI = 3.14;  
}  
class MainClass  
{  
   public static void Main()  
   {  
      Console.WriteLine(Area.PI);  
   }  
}  
  
**Can you declare a field readonly?**  
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.  
  
using System;  
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?**  
  
using System;  
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 PI.  
  
**What is wrong with the sample program below?**  
  
using System;  
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 readonly 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 run time.

**C# Interview Questions on Access Modifiers**

**What are Access Modifiers in C#?**  
In C# there are 5 different types of Access Modifiers.  
**Public**  
The public type or member can be accessed by any other code in the same assembly or another assembly that references it.  
  
**Private**The type or member can only be accessed by code in the same class or struct.  
  
**Protected**  
The type or member can only be accessed by code in the same class or struct, or in a derived class.  
  
**Internal**The type or member can be accessed by any code in the same assembly, but not from another assembly.  
  
**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 accessibilty of types and members with in the types.  
  
**Can you use all access modifiers for all types?**  
No, Not all access modifiers can be used by all types or 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.  
using System;  
internal class InternalBaseClass  
{  
   public void Print()  
   {  
      Console.WriteLine("I am a Base Class Method");  
   }  
}  
public class PublicDerivedClass : InternalBaseClass  
{  
   public static void Main()  
   {  
      Console.WriteLine("I am a Public Derived Class Method");  
   }  
}  
  
When you 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 B that derives from an internal class A. If this were allowed, it would have the effect of making A public, because all protected or internal members of A are accessible from the derived class.  
  
  
**Is the following code legal?**  
  
using System;  
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 modifier 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 accessibility 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?**  
using System;  
interface IExampleInterface  
{  
   public void Save();  
}  
  
No, you cannot specify access modifer 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.

**Why should you override the ToString() method**

**Why should you override the ToString() method?**  
All types in .Net inherit from system.object directly or indirectly. Because of this inheritance, every type in .Net inherit the ToString() method from System.Object class. Consider the example below.  
  
using System;  
public class MainClass  
{  
  public static void Main()  
  {  
   int Number = 10;  
   Console.WriteLine(Number.ToString());  
  }  
}  
  
In the above example Number.ToString() method will correctly give the string representaion of int 10, when you call the ToString() method.  
  
If you have a Customer class as shown in the below example and when you call the ToString() method the output doesnot make any sense. Hence you have to override the ToString() method, that is inherited from the System.Object class.  
  
using System;  
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());  
 }  
}  
  
  
The code sample below shows how to override the ToString() method in a class, that would give the output you want.  
  
  
using System;  
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());  
  }  
}  
  
**Conclusion :** If you have a class or a struct, make sure you override the inherited ToString() method.

**C# Interview Questions on polymorphism**

**Ploymorphism Video**  
   
[**Difference between method overriding and method hiding - Video**](http://csharp-video-tutorials.blogspot.com/2012/06/part-24-c-tutorial-difference-between.html)  
**Explain polymorphism in C# with a simple example?**  
Polymorphism allows you to invoke derived class methods through a base class reference during run-time. An example is shown below.  
*using System;  
public class DrawingObject  
{  
public virtual void Draw()  
{  
Console.WriteLine("I am a drawing object.");  
}  
}  
public class Triangle : DrawingObject  
{  
public override void Draw()  
{  
Console.WriteLine("I am a Triangle.");  
}  
}  
public class Circle : DrawingObject  
{  
public override void Draw()  
{  
Console.WriteLine("I am a Circle.");  
}  
}  
public class Rectangle : DrawingObject  
{  
public override void Draw()  
{  
Console.WriteLine("I am a Rectangle.");  
}  
}  
public class DrawDemo  
{  
public static void Main()  
{  
DrawingObject[] DrawObj = new DrawingObject[4];  
  
DrawObj[0] = new Triangle();  
DrawObj[1] = new Circle();  
DrawObj[2] = new Rectangle();  
DrawObj[3] = new DrawingObject();  
  
foreach (DrawingObject drawObj in DrawObj)  
{  
drawObj.Draw();  
}  
}  
}*  
**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 where as an abstract method should not have a body.  
  
**Can fields inside a class be virtual?**No, Fields inside a class cannot be virtua. Only methods, properties, events and indexers can be virtual.  
  
**Give an example to show for hiding base class methods?**  
Use the new keyword to hide a base class method in the derived class as shown in the example below.  
*using System;  
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();  
DC.Method();  
}  
}*  
**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.  
*using System;  
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();  
}  
}*

**# Interview Questions on Abstract and Sealed Class Members**

**What is an abstract class?**  
An abstract class is an incomplete class and must be implemented in a derived class.  
  
**Can you create an instance of an abstract class?**No, abstract classes are incomplete and you cannot create an instance of an abstract class.  
  
**What is a sealed class?**  
A sealed class is a class that cannot be inherited from. This means, If you have a class called Customer that is marked as sealed. No other class can inherit from Customer class. For example, the below code generates a compile time error "MainClass cannot derive from sealed type Customer.  
*using System;  
public sealed class Customer  
{  
}  
public class MainClass : Customer  
{  
public static void Main()  
{  
}  
}*  
**What are abstract methods?**  
Abstract methods are methods that only the declaration of the method and no implementation.  
  
**Will the following code compile?***using System;  
public abstract class Customer  
{  
public abstract void Test()  
{  
Console.WriteLine("I am customer");  
}  
}  
public class MainClass  
{  
public static void Main()  
{  
}  
}*No, abstract methods cannot have body. Hence, the above code will generate a compile time error stating "Customer.Test() cannot declare a body because it is marked abstract"  
  
**Is the following code legal?**  
*using System;  
public class Customer  
{  
public abstract void Test();  
}  
public class MainClass  
{  
public static void Main()  
{  
}  
}*  
No, if a class has even a single abstract member, the class has to be marked abstract. Hence the above code will generate a compile time error stating "Customer.Test() is abstract but it is contained in nonabstract class Customer"  
  
**How can you force derived classes to provide new method implementations for virtual methods?**  
Abstract classes can be used to force derived classes to provide new method implementations for virtual methods. An example is shown below.  
*public class BaseClass  
{  
public virtual void Method()  
{  
// Original Implementation.  
}  
}  
  
public abstract class AbstractClass : BaseClass  
{  
public abstract override void Method();  
}  
  
public class NonAbstractChildClass : AbstractClass  
{  
public override void Method()  
{  
// New implementation.  
}  
}*  
When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method. If a virtual method is declared abstract, it is still virtual to any class inheriting from the abstract class. A class inheriting an abstract method cannot access the original implementation of the method. In the above example, Method() on class NonAbstractChildClass cannot call Method() on class BaseClass. In this way, an abstract class can force derived classes to provide new method implementations for virtual methods.  
  
**Can a sealed class be used as a base class?**  
No, sealed class cannot be used as a base class. A compile time error will be generated.  
  
**Will the following code compile?***public abstract sealed class Test  
{  
public virtual void Method()  
{  
}  
}*No, a class cannot be marked as sealed and abstract at the same time. This is because by definition, a sealed class cannot be a base class and an abstract class can only be a base class.

**C# Interview Questions on Inheritance**

**Inheritance - Video**  
   
[**Click here for video on Method Hiding**](http://csharp-video-tutorials.blogspot.com/2012/06/part-22-c-tutorial-method-hiding-in-c.html)  
**What are the 4 pillars of any object oriented programming language?  
1.** Abstraction  
**2.** Inheritance  
**3.** Encapsulation  
**4.** Polymorphism  
  
**Do structs support inheritance?**No, structs do not support inheritance, but they can implement interfaces.  
  
**What is the main advantage of using inheritance?**  
Code reuse  
  
**Is the following code legal?***class ChildClass : ParentClassA, ParentClassB  
{  
}*  
No, a child class can have only one base class. You cannot specify 2 base classes at the same time. C# supports single class inheritance only. Therefore, you can specify only one base class to inherit from. However, it does allow multiple interface inheritance.  
  
**What will be the output of the following code?**  
*using System;  
public class BaseClass  
{  
public BaseClass()  
{  
Console.WriteLine("I am a base class");  
}  
}  
public class ChildClass : BaseClass  
{  
public ChildClass()  
{  
Console.WriteLine("I am a child class");  
}  
static void Main()  
{  
ChildClass CC = new ChildClass();  
}  
}***Output:**  
I am a base class  
I am a child class  
This is because base classes are automatically instantiated before derived classes. Notice the output, The BaseClass constructor executed before the ChildClass constructor.  
  
**Does C# support multiple class inheritance?**No, C# supports single class inheritance only. However classes can implement multiple interfaces at the same time.

**C# Interview Questions on structs**

**Structs - Video**  
   
**Will the following code compile?**  
*using System;  
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"  
  
**Can a struct have a default constructor (a constructor without parameters) or a destructor in C#?**  
No  
  
**Can you instantiate a struct without using a new operator in C#?**Yes, you can instantiate a struct without using a new operator  
  
**Can a struct inherit from another struct or class in C#?**  
No, a struct cannot inherit from another struct or class, and it cannot be the base of a class.  
  
**Can a struct inherit from an interface in C#?**Yes  
  
**Are structs value types or reference types?**  
Structs are value types.  
  
**What is the base type from which all structs inherit directly?**All structs inherit directly from System.ValueType, which inherits from System.Object.

**Basic C# Interview Questions on classes and structs**

**Difference between classes and structs - Video**  
   
**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 is created, the variable to which the struct is assigned holds 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 donot 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 donot 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.  
  
**Can you create an instance for an abstract class?**  
No, you cannot create an instance for an abstract class.  
  
**How do you prevent a class from being inherited by another class?**Use the sealed keyword to prevent a class from being inherited by another class.  
  
**Classes and structs can be declared as static, Is this statement true or false?**  
False, only classes can be declared as static and not structs.  
  
**Can you create an instance of a static class?**No, you cannot create an instance of a static class.  
  
**Can a static class contain non static members?**  
No, a static class can contain only static members.

**C# Interview Questions on Data Types**

**What are the 3 types of comments in C#?**  
**1.** Single Line Comments. You define single line comments with // as shown below.  
//This is an example for single line comment  
**2.** Multi line comments. You define multi line comments with /\* \*/ as shown below.  
/\*This is an example for  
Multi Line comments\*/  
**3.** XML Comments. You define XML comments with /// as shown below.  
///This is an example for defining XML comments.  
  
**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 for built in datatypes 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.

**C# Interview Questions on value types and reference types**

**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 a value type or reference type?**  
Value Type  
  
**If you define a user defined data type by using the class keyword, Is it a 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 for value types?**Enum  
Struct  
  
**Give examples for reference types?**  
Class  
Delegate  
Array  
Interface  
  
**What are the differences between value types and reference types?  
1.** Value types are stored on the stack where as reference types are stored on the managed heap.  
**2.** Value type variables directly contain their values where as reference variables holds only a reference to the location of the object that is created on the managed heap.  
**3.** 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 over head of object allocation and garbage collection.  
**4.**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.

### C# Interview Questions on data type casting

**Data type Conversions - Video**  
   
  
**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 as data type conversion.  
  
**What are the 2 kinds of data type conversions in C#?  
Implicit conversions:**No special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.  
  
**Explicit conversions:**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?**  
**1.** Explicit conversions require a cast operator where as an implicit converstion is done automatically.  
**2.** Explicit conversion can lead to data loss where as 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 integer. An implicit conversion is not done automatically bcos 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 shown below.  
//Create a new derived type.  
Car C1 = new Car();  
// Implicit conversion to base type is safe.  
Vehicle V = C1;  
  
// Explicit conversion is required to cast back to derived type. The code below will compile but throw an exception at run time 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

**C# Interview questions on Boxing and Unboxing**

**What is 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 typpe 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?**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 performance impact.

**Basic C# Interview Questions on arrays**

**What is an array?**  
An array is a data structure that contains several variables of the same type.  
  
**What are the 3 different types of arrays?  
1.** Single-Dimensional  
**2.** Multidimensional  
**3.** Jagged  
   
**Arrays - Video**  
   
  
**What is Jagged Array?**  
A jagged array is an array of arrays.  
  
**Are arrays value types or reference types?**Arrays are reference types.  
  
**What is the base class for Array types?**  
System.Array  
  
**Can you use foreach iteration on arrays in C#?**Yes,Since array type implements **IEnumerable**, you can use **foreach** iteration on all arrays in C#.

**Basic C# Interview Questions on strings**

**What is the difference between string keyword and System.String class?**  
**string**keyword is an alias for **Syste.String**class. Therefore, System.String and string keyword are the same, and you can use whichever naming convention you 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?**String objects are immutable means, they cannot be changed after they have been 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 = s1;  
str1 = str1 + "C#";  
System.Console.WriteLine(s2);*The output of the above code is "Hello" and not "Hello C#". This is bcos, 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 new line 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 ""Venkat.""";  
//Output: My Name is "Venkat."

**More C# interview questions on strings**

**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?  
1.** Objects of type StringBuilder are mutable where as objects of type System.String are immutable.   
**2.** As StringBuilder objects are mutable, they offer better performance than string objects of type System.String.  
**3.** 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 use TryParse method as shown in the example below. If the string contains nonnumeric 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?**  
Parse method throws an exception if the string you are trying to parse is not a valid number where as TryParse returns false and does not throw an exception if parsing fails. Hence TryParse is more efficient than Parse.