##### ****What is LINQ?****

LINQ stands for Language Integrated Query, LINQ allows developers to write queries to retrieve, manipulate, and transform data from different data sources, such as databases, collections, XML, and In-Memory objects. It was introduced with .NET Framew**ork 3.5 & Visual Studio 2008.**

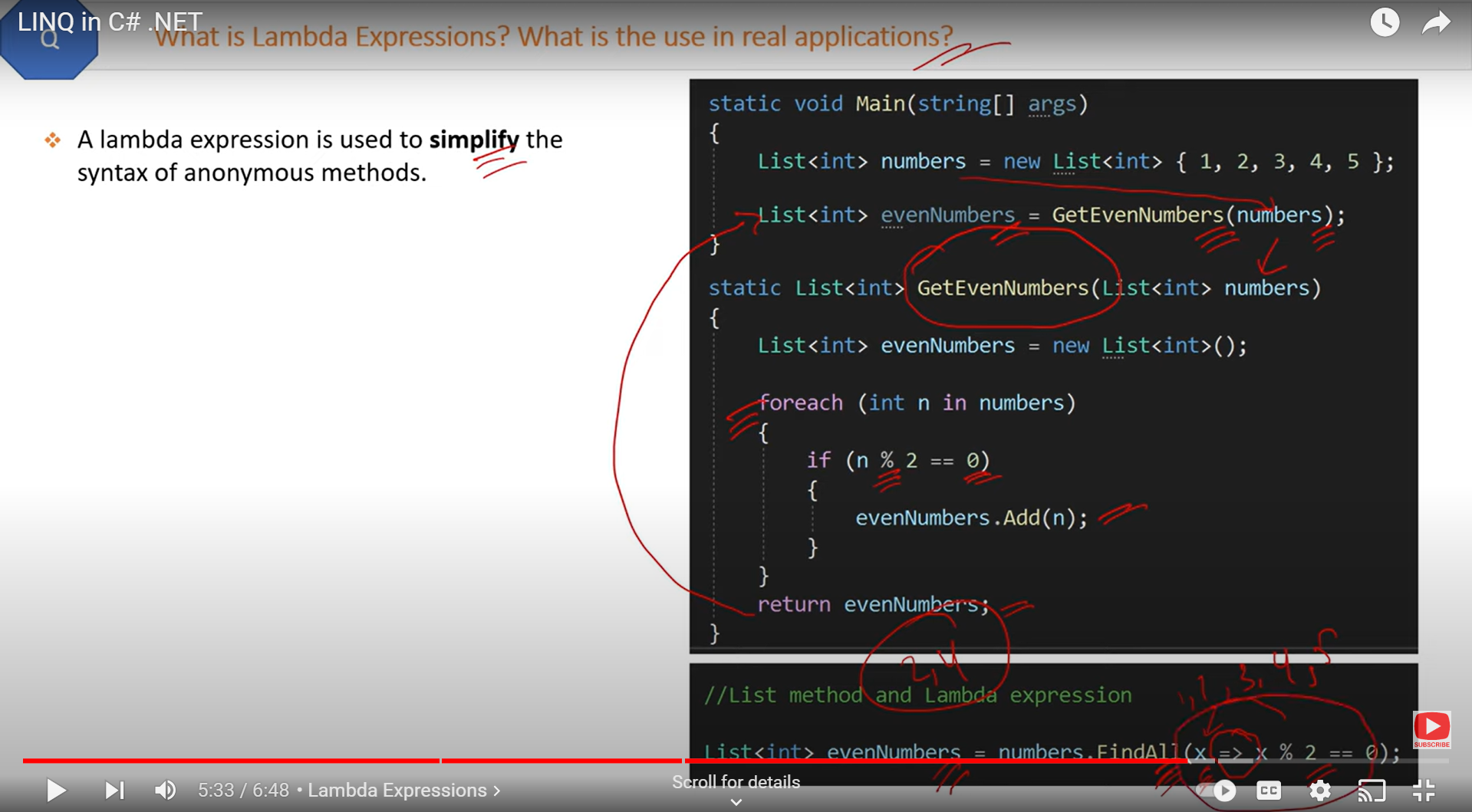
**What are the advantages and disadvantages of linQ?**

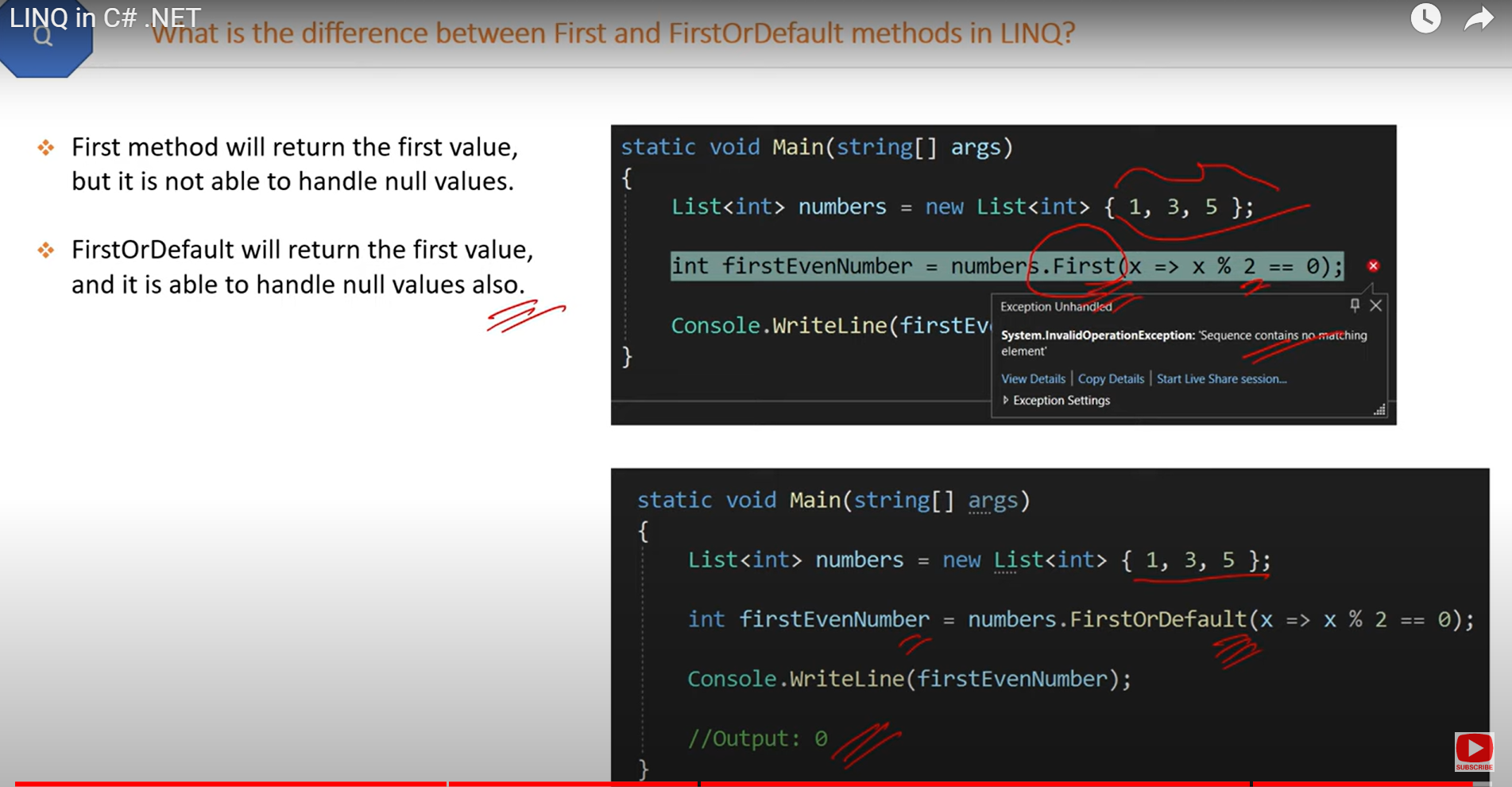
**Advantages :**

1. Easy and simple syntax to learn
2. Improve code readability
3. Improve performance
4. Type safety

**Disadvantages :**

1. limited support for some data sources
2. Difficult to maintain and debug

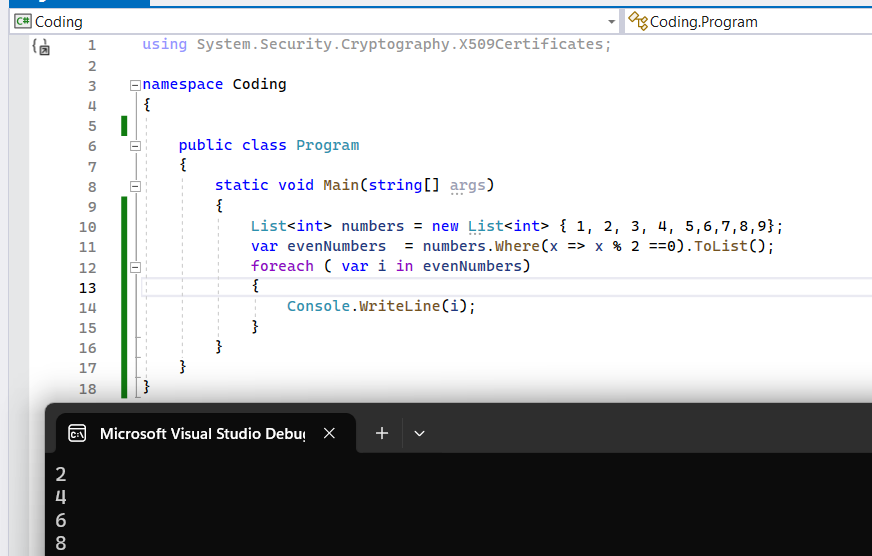


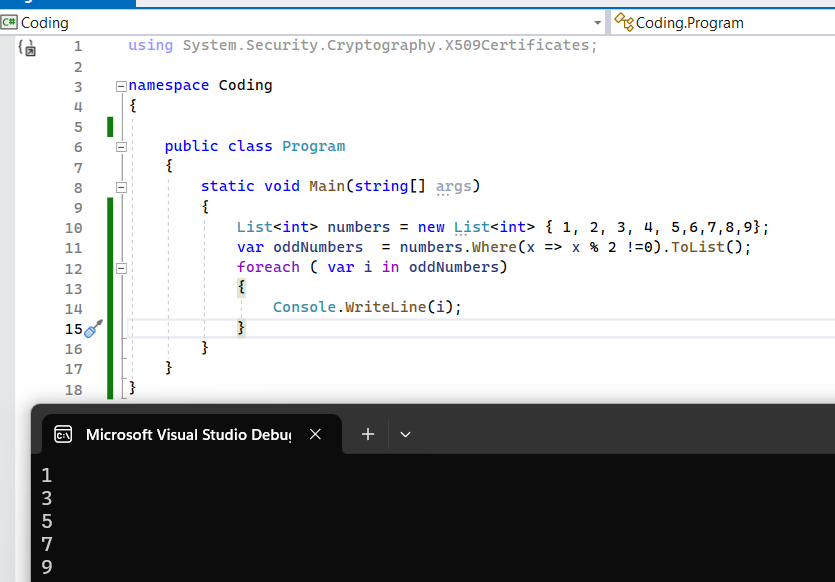


**how to print distinct numbers using Linq or remove duplicate numbers in to the list**

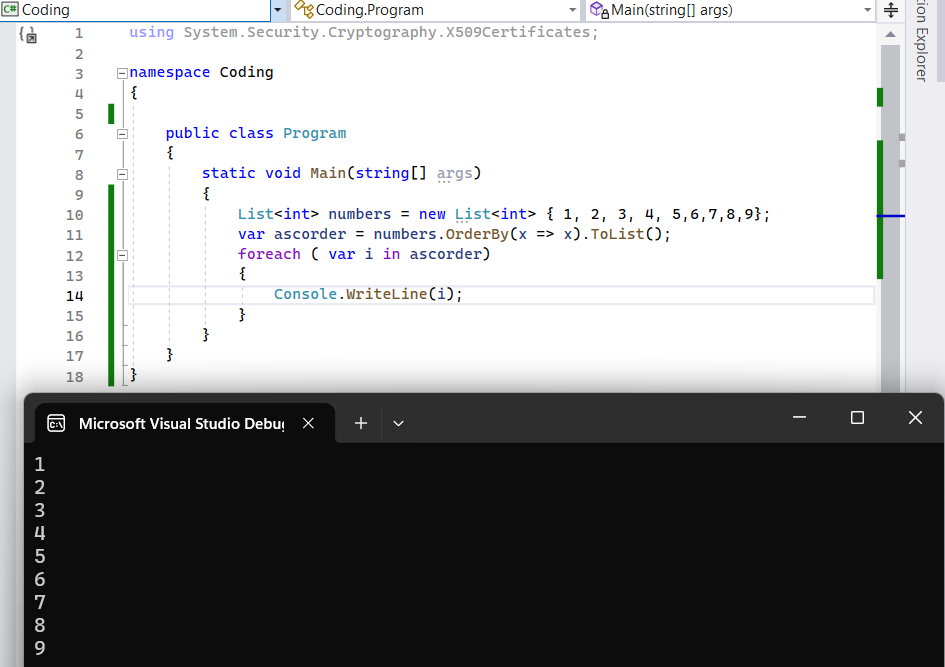
##### 

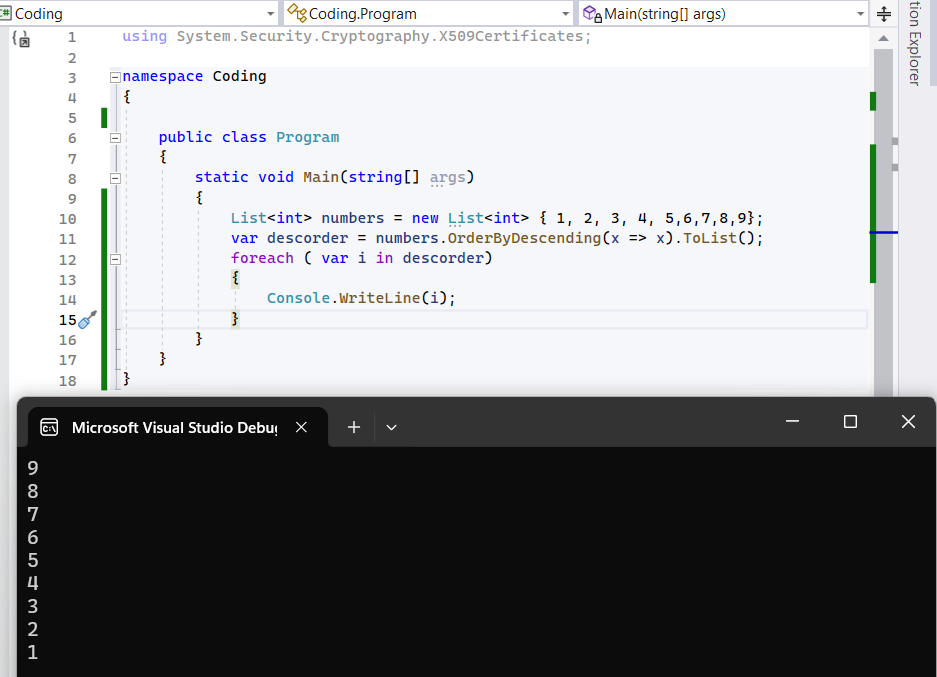
##### ****How to filter evennumbers and odd numbers from the list using LINQ ?****



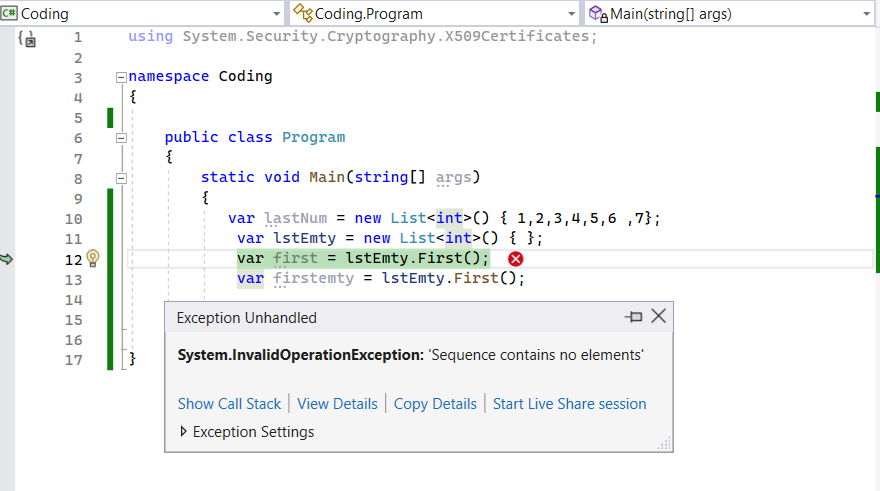


##### ****How to sorting numbers ascending order and descending order from the list using LINQ ?****

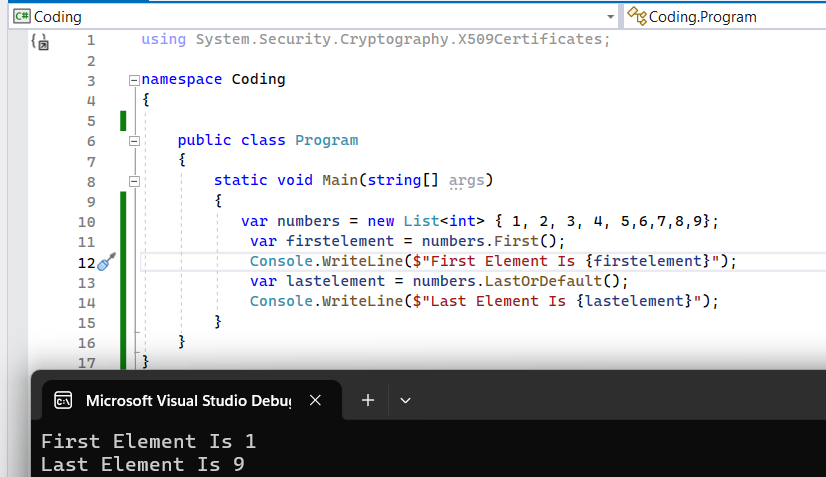


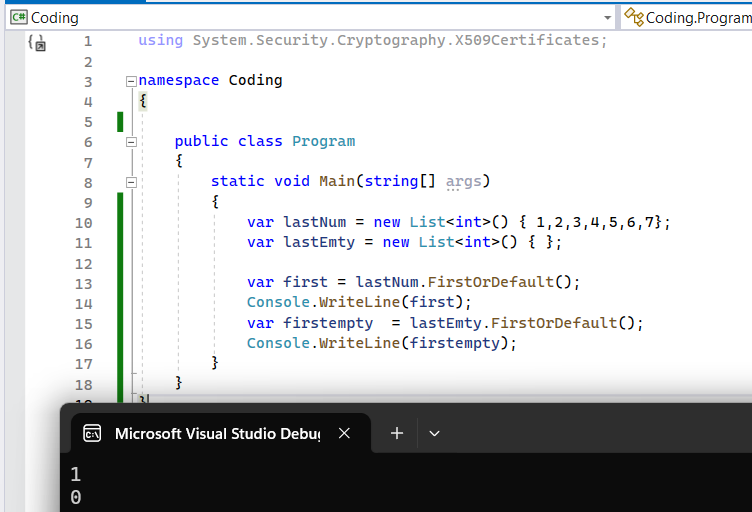


**How to print the first and last elements from the list using First and firstordefault and what is the difference between first and firstordefault?**



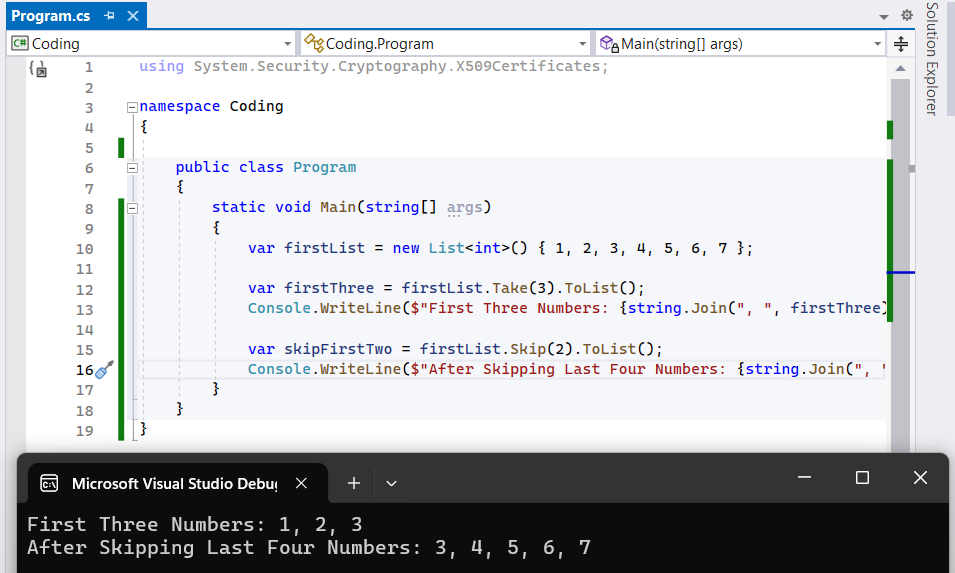
First Method will return the first value , but it is not able to handle null values





FirstOrDefault Method will return the first value , and it able to handle null values

**How to print first three numbers and last four numbers using Take and Skip Methods in linq?**



##### ****What are the 4 pillars of any object-oriented programming language?****

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

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

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

1. Abstract classes can have implementations for some of their members, but the interface can’t have the implementation for any of its members.
2. Interfaces cannot have fields where an abstract class can have fields.
3. An interface can inherit from another **[interface](https://dotnettutorials.net/lesson/interface-c-sharp/)**only and cannot inherit from an abstract class whereas an abstract class can inherit from another abstract class or another interface.
4. A class can inherit from **[multiple interfaces](https://dotnettutorials.net/lesson/multiple-inheritance-csharp/)** at the same time, whereas a class cannot inherit from multiple abstract classes at the same time.
5. Abstract class members can have access modifiers where as interface members cannot have access modifiers as they are by default public.

##### ****When to use Interface?****

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

##### ****When to use Abstract Classes in C#?****

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

##### ****What is an Abstract Class in C#?****

A class that is declared by using the keyword abstract is called an abstract class. An abstract class is a partially implemented class used for developing some of the operations which are common for all next level subclasses. So it contains both abstract methods, concrete methods including variables, properties, and indexers.

It is always created as a superclass next to the interface in the object inheritance hierarchy for implementing common operations from the interface.

An abstract class may or may not have abstract methods. But if a class contains an abstract method then it must be declared as abstract. The abstract class cannot be instantiated directly. It’s compulsory to create/derive a new class from an abstract class in order to provide the functionality to its abstract functions.

##### ****What is a Sealed Class?****

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

##### ****What is the abstract method?****

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

##### ****What is Encapsulation in C#?****

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). In C#, as with other OOP languages, encapsulation refers to bundling related data and behaviors into a single unit (typically a class) and restricting access to some of the object’s components. It helps protect the data’s integrity and expose only what’s necessary to the outside world.

##### ****Real-time Example of Encapsulation Principle in C#: Bank Account Management System****

One real-time example of encapsulation is a bank account management system. Let’s consider the scenario where a bank customer can deposit or withdraw money from their account. Still, certain rules and validations are applied, such as ensuring that the balance does not go below a minimum limit or the customer can’t withdraw more than what they have in the account. Let us see how we can implement this example using the Encapsulation Principle in C#:

**using** *System;*

**namespace** *EncapsulationPrincipleCSharp*

**{**

**public** **class** BankAccount

**{**

// This private field is encapsulated and can't be directly accessed from outside the class.

**private** **decimal** balance;

**public** **decimal** Balance

**{**

// Only provides a way to read the balance but not modify it directly.

**get** **{** **return** balance; **}**

**}**

**public** BankAccount**(decimal** initialBalance**)**

**{**

**if** **(**initialBalance **<** 0**)**

**{**

**throw** new ArgumentException**(**"Initial balance cannot be negative."**)**;

**}**

balance = initialBalance;

**}**

**public** **void** Deposit**(decimal** amount**)**

**{**

**if** **(**amount **<**= 0**)**

**{**

**throw** new ArgumentException**(**"Deposit amount should be positive."**)**;

**}**

balance += amount;

**}**

**public** **void** Withdraw**(decimal** amount**)**

**{**

**if** **(**amount **<**= 0**)**

**{**

**throw** new ArgumentException**(**"Withdrawal amount should be positive."**)**;

**}**

**if** **(**balance - amount **<** 0**)**

**{**

**throw** new InvalidOperationException**(**"Insufficient funds."**)**;

**}**

balance -= amount;

**}**

**}**

//Testing Encapsulation Principle

**public** **class** Program

**{**

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

**{**

// Starts with a balance of 500

BankAccount myAccount = new BankAccount**(**500**)**;

// Balance becomes 700

myAccount.Deposit**(**200**)**;

Console.WriteLine**(**myAccount.Balance**)**; // Outputs: 700

// Balance becomes 600

myAccount.Withdraw**(**100**)**;

Console.WriteLine**(**myAccount.Balance**)**; // Outputs: 600

// myAccount.balance = -1000; // This would be an error, as the balance field is private and inaccessible directly.

Console.Read**()**;

**}**

**}**

**}**

###### ****Advantages of Encapsulation Principle in C#:****

* ****Maintainability:**** Encapsulation helps in organizing code for maintainability. Since the internal workings are hidden, changes to the logic or data do not affect the external code that uses the object.
* ****Flexibility and Extensibility:**** You can change the internal implementation of a class without affecting the classes that use it.
* ****Control:**** By exposing only necessary parts of the object, you can have a high degree of control over the class’s functionality and data. For example, you can use validation within a class’s set method to enforce specific constraints.
* ****Increased Security:**** Encapsulation protects the integrity of the data by allowing only controlled and valid operations on it. It prevents unauthorized access and accidental modification of data.
* ****Abstraction:**** It provides a clear separation between what an object does and how it achieves what it does. This separation allows developers to work on complex systems by understanding the interfaces between objects without knowing the inner workings.
* ****Reduction of Side Effects:**** By restricting direct access to the object’s data, you can ensure that all operations on that data are done through well-defined interfaces, which reduces unforeseen errors or side effects.

##### ****What is Abstraction in C#?****

Abstraction is a foundational concept in object-oriented programming (OOP) and is not limited to just C#. In C#, like in other OOP languages, abstraction refers to hiding the complex implementation details and showing only the essential features of an object. Here’s a more detailed explanation:

* ****Hiding Complexity:**** Just like a car dashboard doesn’t show all the intricate details of how the engine works but provides you with a few essential controls (like the steering wheel, accelerator, brakes, etc.), abstraction in programming is about hiding the intricate details of how something works and providing only the essential operations to work with it.
* ****Facilitating Extensibility:**** Because abstraction allows you to hide implementation details behind a consistent interface, you can later change the underlying implementation without affecting the code that uses this abstraction.
* ****Using Classes and Interfaces:**** In C#, abstraction can be achieved using classes and interfaces. While you can’t create an instance of an abstract class, you can use its shape (properties and methods) in child classes. Interfaces go a step further by only defining the shape without any implementation, enforcing concrete classes to provide the specifics.

##### ****Real-Time Example of Abstraction Principle in C#: Vehicle System****

Let’s consider a Real-time Example Vehicle System. Suppose you want to model a basic vehicle system. All vehicles can be started and stopped, but the underlying details of how this happens can differ for each vehicle type. Let us see how we can implement this example using the Abstraction Principle in C#:

**using** *System;*

**namespace** *AbstractionPrincipleCSharp*

**{**

//Abstract Base Class (Abstraction)

**public** **abstract** **class** Vehicle

**{**

// These are abstract methods; the derived classes will provide the implementation.

**public** **abstract** **void** Start**()**;

**public** **abstract** **void** Stop**()**;

**}**

//Concrete Implementations

**public** **class** Car : Vehicle

**{**

**public** **override** **void** Start**()**

**{**

Console.WriteLine**(**"Car is starting with a key turn."**)**;

**}**

**public** **override** **void** Stop**()**

**{**

Console.WriteLine**(**"Car is stopping using its brakes."**)**;

**}**

**}**

**public** **class** ElectricTrain : Vehicle

**{**

**public** **override** **void** Start**()**

**{**

Console.WriteLine**(**"Electric train is starting by powering up."**)**;

**}**

**public** **override** **void** Stop**()**

**{**

Console.WriteLine**(**"Electric train is stopping by cutting off the power."**)**;

**}**

**}**

//Testing Abstraction Principle

**public** **class** Program

**{**

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

**{**

//Using the Abstraction

Vehicle myCar = new Car**()**;

Vehicle myTrain = new ElectricTrain**()**;

StartVehicle**(**myCar**)**; // Output: Car is starting with a key turn.

StartVehicle**(**myTrain**)**; // Output: Electric train is starting by powering up.

Console.Read**()**;

**}**

**static** **void** StartVehicle**(**Vehicle vehicle**)**

**{**

vehicle.Start**()**;

**}**

**}**

**}**

##### ****What is the Inheritance Principle in C#?****

Inheritance is one of the core concepts of object-oriented programming (OOP) and plays a significant role in C#. It allows a class (the “derived” or “child” class) to inherit members (like fields, properties, methods, and events) from another class (the “base” or “parent” class). Inheritance promotes code reusability and establishes a hierarchical relationship between classes.

So, inheritance is another core principle of object-oriented programming. It allows a class to inherit properties and methods from another, promoting code reuse and establishing a natural hierarchy between classes. Here’s a breakdown of the key concepts associated with inheritance in C#:

* ****Base Class (Parent Class):**** The class whose members are inherited by another class. It’s also known as the superclass or parent class.
* ****Derived Class (Child Class):**** The class that inherits members from the base class. It can also introduce additional members or override inherited ones. It’s also known as the subclass or child class.
* ****Inheritance Chain:**** Classes can inherit from a class that inherits from another class, leading to an inheritance “chain” or “hierarchy.”
* ****Access Modifiers:**** Only members with public, protected, or internal access modifiers in the base class are accessible from the derived class. Private members of the base class are not accessible in the derived class.
* ****Method Overriding:**** If the base class defines a method as virtual, the derived class can override this method using the override keyword. This allows the derived class to provide a specific implementation for that method.
* ****The base Keyword:**** Inside a derived class, you can use the base keyword to access members of the base class.
* ****Sealed Classes:**** In C#, if you want to prevent a class from being inherited, you can mark it with the sealed keyword.

##### ****Real-Time Example of Inheritance Principle in C#: Vehicle Management System****

Consider different types of vehicles, such as a basic vehicle, a car, and a motorcycle. All vehicles can be started and stopped and have a speed. However, cars and motorcycles have specific properties and behaviors. Let us see how we can implement this example using the Inheritance Principle in C#:

**using** *System;*

**namespace** *InheritancePrincipleCSharp*

**{**

//Base Class (Parent Class) - Vehicle

**public** **class** Vehicle

**{**

**public** **int** Speed **{** **get**; **protected** **set**; **}**

**public** **void** Start**()**

**{**

Console.WriteLine**(**"Vehicle started."**)**;

**}**

**public** **void** Stop**()**

**{**

Console.WriteLine**(**"Vehicle stopped."**)**;

**}**

**public** **virtual** **void** Accelerate**()**

**{**

Speed += 5;

Console.WriteLine**(**$"Vehicle accelerates. Current speed: {Speed} km/h."**)**;

**}**

**}**

//Derived Class (Child Class) - Car

**public** **class** Car : Vehicle

**{**

**public** **int** Doors **{** **get**; **set**; **}**

**public** **override** **void** Accelerate**()**

**{**

Speed += 10;

Console.WriteLine**(**$"Car accelerates. Current speed: {Speed} km/h."**)**;

**}**

**public** **void** OpenSunroof**()**

**{**

Console.WriteLine**(**"Sunroof opened."**)**;

**}**

**}**

//Derived Class (Child Class) - Motorcycle

**public** **class** Motorcycle : Vehicle

**{**

**public** **bool** HasSideCar **{** **get**; **set**; **}**

**public** **override** **void** Accelerate**()**

**{**

Speed += 7;

Console.WriteLine**(**$"Motorcycle accelerates. Current speed: {Speed} km/h."**)**;

**}**

**public** **void** UseKickstand**()**

**{**

Console.WriteLine**(**"Kickstand placed."**)**;

**}**

**}**

//Testing Inheritance Principle

**public** **class** Program

**{**

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

**{**

//Using the Inheritance

Car myCar = new Car **{** Doors = 4 **}**;

myCar.Start**()**;

myCar.Accelerate**()**;

myCar.OpenSunroof**()**;

myCar.Stop**()**;

Console.WriteLine**()**;

Motorcycle myBike = new Motorcycle **{** HasSideCar = **false** **}**;

myBike.Start**()**;

myBike.Accelerate**()**;

myBike.UseKickstand**()**;

myBike.Stop**()**;

Console.Read**()**;

**}**

**}**

**}**

1. ****When can a derived class override a base class member?****
2. ****What is the difference between a virtual method and an abstract method?****
3. ****What is the difference between a virtual method and an abstract method?****

##### ****What is Polymorphism in C#?****

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

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

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

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

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

1. Overloading
2. Overriding
3. Hiding

##### ****Overloading again is of three types****

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

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

There are two types of polymorphism

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

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

This is one of the frequently asked **[C# Polymorphism interview](https://dotnettutorials.net/lesson/polymorphism-csharp/)** questions. In the case of compile-time polymorphism, the object of the class recognizes which method to be executed for a particular method call at the time of program compilation and binds the method call with method definition.

This happens in the case of overloading because in the case of overloading each method will have a different signature and basing on the method call we can easily recognize the method which matches the method signature. It is also called static **[polymorphism](https://dotnettutorials.net/lesson/polymorphism-csharp/)**or early binding. Static polymorphism is achieved by using function overloading and operator overloading

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

This is also one of the frequently asked Polymorphism interview questions in C#. In the case of runtime **[polymorphism](https://dotnettutorials.net/lesson/polymorphism-csharp/)**for a given method call, we can recognize which method has to be executed exactly at runtime but not in compilation time because in the case of overriding and hiding we have multiple methods with the same signature. So which method to be given preference and executed that is identified at runtime and binds the method call with its suitable method. It is also called dynamic polymorphism or late binding. Dynamic polymorphism is achieved by using function overriding.

**using** *System;*

**namespace** *PolymorphismExample*

**{**

// Base class

**public** **abstract** **class** Animal

**{**

**public** **abstract** **void** MakeSound**()**;

**}**

// Derived class

**public** **class** Dog : Animal

**{**

**public** **override** **void** MakeSound**()**

**{**

Console.WriteLine**(**"The dog barks."**)**;

**}**

**}**

// Another derived class

**public** **class** Cat : Animal

**{**

**public** **override** **void** MakeSound**()**

**{**

Console.WriteLine**(**"The cat meows."**)**;

**}**

**}**

//Testing Polymorphism Principle

**public** **class** Program

**{**

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

**{**

Animal myDog = new Dog**()**;

Animal myCat = new Cat**()**;

MakeAnimalSound**(**myDog**)**; // Outputs: The dog barks.

MakeAnimalSound**(**myCat**)**; // Outputs: The cat meows.

Console.Read**()**;

**}**

// This function showcases polymorphism in action.

// Even though it accepts a parameter of type 'Animal',

// it's able to handle any derived type.

**static** **void** MakeAnimalSound**(**Animal animal**)**

**{**

animal.MakeSound**()**;

**}**

**}**

**}**

****In this Example:****

* We have an abstract Animal class with an abstract method, MakeSound().
* Derived classes (Dog and Cat) provide their own implementation of the MakeSound() method.
* In the Program class, even though we use the Animal type to hold references to the derived classes, we can still call the appropriate derived class’s MakeSound() method. This is the essence of polymorphism.

This allows for flexibility and makes adding more animal types in the future easier without making major changes to existing code. If you were to add a new animal, say Bird, you’d need to create a Bird class derived from Animal and provide its own implementation for the MakeSound() method.

##### ****Real-Time Example of Polymorphism Principle in C#: Graphics Program****

Let’s understand polymorphism using another real-world example: a graphics program that can draw different shapes. Each shape can be drawn on a canvas, but the way each shape is drawn might differ. Here’s how we can represent this using polymorphism in C#:

* ****Base Class: Shape****
* ****Derived Classes: Circle, Rectangle****

Let us see how we can implement this example using the Polymorphism Principle in C#:

**using** *System;*

**namespace** *PolymorphismExample*

**{**

// Base class

**public** **abstract** **class** Shape

**{**

**public** **abstract** **void** Draw**()**;

**}**

// Derived class

**public** **class** Circle : Shape

**{**

**public** **override** **void** Draw**()**

**{**

Console.WriteLine**(**"Drawing a circle on the canvas."**)**;

**}**

**}**

// Another derived class

**public** **class** Rectangle : Shape

**{**

**public** **override** **void** Draw**()**

**{**

Console.WriteLine**(**"Drawing a rectangle on the canvas."**)**;

**}**

**}**

**class** Program

**{**

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

**{**

Shape myCircle = new Circle**()**;

Shape myRectangle = new Rectangle**()**;

DrawShape**(**myCircle**)**; // Outputs: Drawing a circle on the canvas.

DrawShape**(**myRectangle**)**; // Outputs: Drawing a rectangle on the canvas.

Console.ReadKey**()**;

**}**

// This function showcases polymorphism.

// Even though it accepts a parameter of type 'Shape',

// it's able to handle any shape derived from it.

**static** **void** DrawShape**(**Shape shape**)**

**{**

shape.Draw**()**;

**}**

**}**

**}**

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

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

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

In **[Method overriding](https://dotnettutorials.net/lesson/function-overriding-csharp/)**, the parent class gives permission for its child class to override the method by declaring it as****virtual****. Now the child class can override the method using the ****Override**** keyword as it got permission from the parent. The parent class methods can be redefined under child classes even if they were not declared as ****Virtual****by using the ****‘new’**** keyword.

In **[method overriding](https://dotnettutorials.net/lesson/function-overriding-csharp/)** a base class reference variable pointing to a child class object will invoke the overridden method in the child class. In method hiding a base class reference variable pointing to a child class object will invoke the hidden method in the base class.

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

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

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

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

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

.NET Framework provides two things such as

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

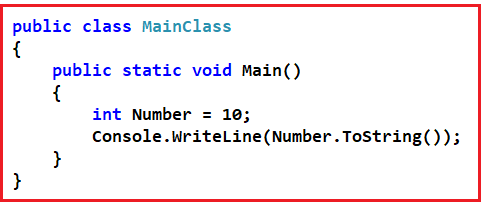
##### ****Explain BCL.****

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

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

##### ****Explain CLR and its Execution process.****

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



In .NET, the code is compiled twice.

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

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

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

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

##### ****3.What is metadata?****

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

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

##### ****4.What is an assembly?****

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

##### ****5.What is the difference between an EXE and a DLL?****

This is one of the frequently asked C# Interview Questions and Answers. Let us understand the difference between **[Exe and DLL](https://dotnettutorials.net/lesson/assembly-dll-exe/)**.

EXE is an executable file and can run by itself as an application whereas DLL is usually consumed by an EXE or by another DLL and we cannot run or execute DLL directly.

For example in .NET compiling a Console Application or a Windows Application generates EXE, whereas compiling a Class Library Project or an ASP.NET web application generates DLL. In the .NET framework, both EXE and DLL are called assemblies.

A DLL can be reused in the application whereas an exe file can never be reused in an application. EXE stands for executable, and DLL stands for Dynamic Link Library

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

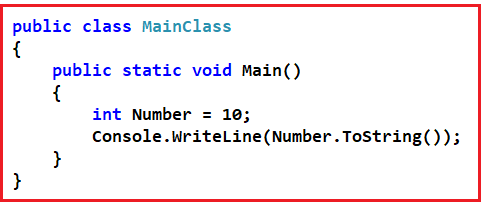
This is a very commonly asked C# interview question. This question is basically asked to check if you are passionate about catching up with the latest technological advancements. The list below shows the new features that are introduced in C# 7. Let’s have a look at the new features that are introduced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

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

##### ****7.Why should you override the ToString() method?****

This C# Interview Question is one of the most frequently asked .NET questions.

All types in .Net inherit from the ****System.Object**** class directly or indirectly. Because of this inheritance, every type in .Net inherits the ToString() method from System.Object class. To understand this better, please have a look at the example.



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

**public** **class** Customer

**{**

**public** **string** FirstName;

**public** **string** LastName;

**}**

**public** **class** MainClass

**{**

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

**{**

Customer C = new Customer**()**;

C.FirstName = "David";

C.LastName = "Boon";

Console.WriteLine**(**C.ToString**())**;

**}**

**}**

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

**public** **class** Customer

**{**

**public** **string** FirstName;

**public** **string** LastName;

**public** **override** **string** ToString**()**

**{**

**return** LastName + ", " + FirstName;

**}**

**}**

**public** **class** MainClass

**{**

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

**{**

Customer C = new Customer**()**;

C.FirstName = "David";

C.LastName = "Boon";

Console.WriteLine**(**C.ToString**())**;

**}**

**}**

##### ****8.What do you mean by String objects are immutable?****

This C# Interview question is frequently asked in .NET Interviews.

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

**string** s1 = "First String ";

**string** s2 = "Second String";

// Concatenate s1 and s2. This actually creates a new

// string object and stores it in s1, releasing the

// reference to the original object.

s1 += s2;

System.Console.WriteLine**(**s1**)**;

// Output: First String Second String

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

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

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

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

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

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

**class** Customer

**{**

// Private fileds not accessible outside the class.

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

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

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

// public FirstName property exposes \_firstName variable

**public** **string** FirstName

**{**

**get**

**{**

**return** \_firstName;

**}**

**set**

**{**

\_firstName = **value**;

**}**

**}**

// public LastName property exposes \_lastName variable

**public** **string** LastName

**{**

**get**

**{**

**return** \_lastName;

**}**

**set**

**{**

\_lastName = **value**;

**}**

**}**

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

**public** **string** FullName

**{**

**get**

**{**

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

**}**

**}**

//Country Property is Write Only

**public** **string** Country

**{**

**set**

**{**

\_coutry = **value**;

**}**

**}**

**}**

**class** MainClass

**{**

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

**{**

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

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

CustomerObject.FirstName = "David";

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

CustomerObject.LastName = "Boon";

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

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

**}**

**}**

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

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

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

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

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

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

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

**class** Customer

**{**

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

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

**public** **string** FirstName

**{**

**get**

**{**

**return** \_firstName;

**}**

**set**

**{**

\_firstName = **value**;

**}**

**}**

**public** **string** LastName

**{**

**get**

**{**

**return** \_lastName;

**}**

**set**

**{**

\_lastName = **value**;

**}**

**}**

// FullName is virtual

**public** **virtual** **string** FullName

**{**

**get**

**{**

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

**}**

**}**

**}**

**class** BankCustomer : Customer

**{**

// Overriding the FullName virtual property derived from customer class

**public** **override** **string** FullName

**{**

**get**

**{**

**return** "Mr. " + FirstName + " " + LastName;

**}**

**}**

**}**

**class** MainClass

**{**

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

**{**

BankCustomer BankCustomerObject = new BankCustomer**()**;

BankCustomerObject.FirstName = "David";

BankCustomerObject.LastName = "Boon";

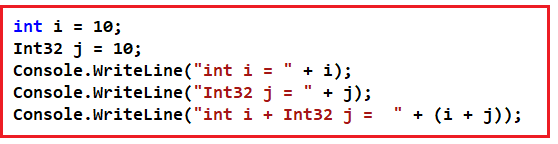
Console.WriteLine**(**"Customer Full Name is : " + BankCustomerObject.FullName**)**;

**}**

**}**

##### ****13.Difference between int and Int32 in C#****

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



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

enum Test : Int32

**{**

abc = 1

**}**

##### ****The following code will compile just fine****

enum Test: **int**

**{**

abc = 1

**}**

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

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

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

1. Value Types
2. Reference Types

##### ****15.What are the differences between value types and reference types?****

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

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

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

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

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

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

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

##### ****What are Boxing and Unboxing?****

****Boxing****– Converting a value type to a reference type is called boxing. An example is shown below.  
****int i = 101;****  
****object obj = (object)i; // Boxing****

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

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

##### ****17.What happens during the process of boxing?****

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

##### ****18.What are Access Modifiers in C#?****

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

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

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

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

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

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

**namespace** *PartialClass*

**{**

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

**{**

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

**{**

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

**}**

**}**

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

**{**

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

**{**

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

**}**

**}**

**public** **class** Demo

**{**

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

**{**

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

StudentObject.Study**()**;

StudentObject.Play**()**;

**}**

**}**

**}**

##### ****What is a Constructor in C#?****

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

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

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

##### ****Explain about Default constructor?****

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

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

##### ****Explain about parameterized constructor?****

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

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

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

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

##### ****Explain the static constructor in C#?****

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

We can create a constructor as static and when a constructor is created as static, it will be invoked only once. There is no matter how many numbers of instances (objects) of the class are created but it is going too invoked only once and that is during the creation of the first instance (object) of the class.

The static constructor is used to initialize static fields of the class and we can also write some code inside the static constructor that needs to be executed only once.

Static data fields are created only once in a class even though we are created any number of objects.

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

##### ****Explain Private constructor in C#?****

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

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

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

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

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

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

##### ****When to use a Private constructor in c#?****

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

There are several reasons for using private constructors

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

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

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

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

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

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

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

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

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

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

##### ****Types of Delegates in C#:****

****Delegates are classified into two types such as****

1. Single cast delegate
2. Multicast delegate

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

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

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

###### ****Tip to remember delegate syntax:****

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

##### ****Sample Delegate Program:****

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

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

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

###### ****Approach1:****

**namespace** *Sample*

**{**

**public** **delegate** **void** SampleDelegate**()**;

**public** **class** Sample

**{**

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

**{**

SampleDelegate del1 = new SampleDelegate**(**SampleMethodOne**)**;

SampleDelegate del2 = new SampleDelegate**(**SampleMethodTwo**)**;

SampleDelegate del3 = new SampleDelegate**(**SampleMethodThree**)**;

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

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

SampleDelegate del4 = del1 + del2 + del3 - del2;

del4**()**;

**}**

**public** **static** **void** SampleMethodOne**()**

**{**

Console.WriteLine**(**"SampleMethodOne Invoked"**)**;

**}**

**public** **static** **void** SampleMethodTwo**()**

**{**

Console.WriteLine**(**"SampleMethodTwo Invoked"**)**;

**}**

**public** **static** **void** SampleMethodThree**()**

**{**

Console.WriteLine**(**"SampleMethodThree Invoked"**)**;

**}**

**}**

**}**

###### ****Approach2:****

**namespace** *Sample*

**{**

**public** **delegate** **void** SampleDelegate**()**;

**public** **class** Sample

**{**

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

**{**

// In this example del is a multicast delegate. You use += operator

// to chain delegates together and -= operator to remove.

SampleDelegate del = new SampleDelegate**(**SampleMethodOne**)**;

del += SampleMethodTwo;

del += SampleMethodThree;

del -= SampleMethodTwo;

del**()**;

**}**

**public** **static** **void** SampleMethodOne**()**

**{**

Console.WriteLine**(**"SampleMethodOne Invoked"**)**;

**}**

**public** **static** **void** SampleMethodTwo**()**

**{**

Console.WriteLine**(**"SampleMethodTwo Invoked"**)**;

**}**

**public** **static** **void** SampleMethodThree**()**

**{**

Console.WriteLine**(**"SampleMethodThree Invoked"**)**;

**}**

**}**

**}**

****Note***:*** A multicast delegate invokes the methods in the invocation list, in the same order in which they are added.

If the delegate has a return type other than void and if the delegate is a multicast delegate, only the value of the last invoked method will be returned. Along the same lines, if the delegate has an out parameter, the value of the output parameter will be the value assigned by the last method.

##### ****Where do you use multicast delegates?****

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

##### ****Where did you use delegates in your project? Or how did you use delegates in your project?****

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

###### ****Employee Class****

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** **string** Name **{** **get**; **set**; **}**

**public** **int** Experience **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

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

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

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

##### ****How to achieve?****

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

**namespace** *DelegateDemo*

**{**

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

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** **string** Name **{** **get**; **set**; **}**

**public** **int** Experience **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**public** **static** **void** PromoteEmployee**(**List**<**Employee**>** lstEmployees, EligibleToPromotion IsEmployeeEligible**)**

**{**

**foreach** **(**Employee employee in lstEmployees**)**

**{**

**if** **(**IsEmployeeEligible**(**employee**))**

**{**

Console.WriteLine**(**"Employee {0} Promoted", employee.Name**)**;

**}**

**}**

**}**

**}**

**}**

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

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

**namespace** *DelegateDemo*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** **string** Name **{** **get**; **set**; **}**

**public** **int** Experience **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**public** **static** **void** PromoteEmployee**(**List**<**Employee**>** lstEmployees, EligibleToPromotion IsEmployeeEligible**)**

**{**

**foreach** **(**Employee employee in lstEmployees**)**

**{**

**if** **(**IsEmployeeEligible**(**employee**))**

**{**

Console.WriteLine**(**"Employee {0} Promoted", employee.Name**)**;

**}**

**}**

**}**

**}**

**class** Program

**{**

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

**{**

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Experience = 5,

Salary = 10000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Kumar",

Experience = 10,

Salary = 20000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Rout",

Experience = 20,

Salary = 30000

**}**;

List**<**Employee**>** lstEmployess = new List**<**Employee**>()**;

lstEmployess.Add**(**emp1**)**;

lstEmployess.Add**(**emp2**)**;

lstEmployess.Add**(**emp3**)**;

EligibleToPromotion eligibleTopromote = new EligibleToPromotion**(**Program.Promote**)**;

Employee.PromoteEmployee**(**lstEmployess, eligibleTopromote**)**;

Console.ReadKey**()**;

**}**

**public** **static** **bool** Promote**(**Employee employee**)**

**{**

**if** **(**employee.Salary **>** 10000**)**

**{**

**return** **true**;

**}**

**else**

**{**

**return** **false**;

**}**

**}**

**}**

**}**

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

##### ****Using Lambda expressions****

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

**class** Program

**{**

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

**{**

Employee emp1 = new Employee**()**

**{**

ID = 101,

Name = "Pranaya",

Experience = 5,

Salary = 10000

**}**;

Employee emp2 = new Employee**()**

**{**

ID = 102,

Name = "Kumar",

Experience = 10,

Salary = 20000

**}**;

Employee emp3 = new Employee**()**

**{**

ID = 103,

Name = "Rout",

Experience = 20,

Salary = 30000

**}**;

List**<**Employee**>** lstEmployess = new List**<**Employee**>()**;

lstEmployess.Add**(**emp1**)**;

lstEmployess.Add**(**emp2**)**;

lstEmployess.Add**(**emp3**)**;

Employee.PromoteEmployee**(**lstEmployess, x =**>** x.Experience **>** 5**)**;

**}**

**}**

##### ****Why do we need Multi-threading in our project?****

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

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

##### ****What are the advantages and disadvantages of multithreading?****

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

###### ****Advantages of multithreading:****

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

###### ****Disadvantages of multithreading:****

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

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

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

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

**namespace** *ThreadingExample*

**{**

**class** Program

**{**

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

**{**

// Prompt the user for the target number

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

// Read from the console and store it in target variable

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

// Create an instance of the Number class, passing it

// the target number that was read from the console

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

// Specify the Thread function

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

// Alternatively we can just use Thread class constructor as shown below

// Thread T1 = new Thread(number.PrintNumbers);

T1.Start**()**;

**}**

**}**

// Number class also contains the data it needs to print the numbers

**class** Number

**{**

**int** \_target;

// When an instance is created, the target number needs to be specified

**public** Number**(int** target**)**

**{**

// The targer number is then stored in the class private variable \_target

this.\_target = target;

**}**

// Function prints the numbers from 1 to the traget number that the user provided

**public** **void** PrintNumbers**()**

**{**

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

**{**

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

**}**

**}**

**}**

**}**

##### ****What is the difference between Threads and Tasks?****

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

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

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

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

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

###### ****Program code used in the demo:****

**namespace** *MultithreadingDemo*

**{**

**class** Program

**{**

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

**{**

Console.WriteLine**(**"Main Thread Started" + Thread.CurrentThread.Name**)**;

Thread T1 = new Thread**(**Program.Thread1Function**)**;

T1.Start**()**;

Thread T2 = new Thread**(**Program.Thread2Function**)**;

T2.Start**()**;

//if (T1.Join(1000))

//{

// Console.WriteLine("Thread1Function completed");

//}

//else

//{

// Console.WriteLine("Thread1Function hot not completed in 1 second");

//}

T1.Join**()**;

T2.Join**()**;

Console.WriteLine**(**"Thread2Function completed"**)**;

**for** **(int** i = 1; i **<**= 10; i++**)**

**{**

**if** **(**T1.IsAlive**)**

**{**

Console.WriteLine**(**"Thread1Function is still doing it's work"**)**;

Thread.Sleep**(**500**)**;

**}**

**else**

**{**

Console.WriteLine**(**"Thread1Function Completed"**)**;

**break**;

**}**

**}**

Console.WriteLine**(**"Main Thread Completed"**)**;

Console.ReadLine**()**;

**}**

**public** **static** **void** Thread1Function**()**

**{**

Console.WriteLine**(**"Thread1Function started"**)**;

Thread.Sleep**(**5000**)**;

Console.WriteLine**(**"Thread1Function is about to return"**)**;

**}**

**public** **static** **void** Thread2Function**()**

**{**

Console.WriteLine**(**"Thread2Function started"**)**;

**}**

**}**

**}**

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

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

**namespace** *MultithreadingDemo*

**{**

**class** Program

**{**

**static** **int** Total = 0;

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

**{**

AddOneMillion**()**;

AddOneMillion**()**;

AddOneMillion**()**;

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

Console.ReadLine**()**;

**}**

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

**{**

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

**{**

Total++;

**}**

**}**

**}**

**}**

****Output:****Total = 3000000

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

###### ****Now, let’s rewrite the program using multiple threads.****

**namespace** *MultithreadingDemo*

**{**

**class** Program

**{**

**static** **int** Total = 0;

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

**{**

Thread thread1 = new Thread**(**Program.AddOneMillion**)**;

Thread thread2 = new Thread**(**Program.AddOneMillion**)**;

Thread thread3 = new Thread**(**Program.AddOneMillion**)**;

thread1.Start**()**;

thread2.Start**()**;

thread3.Start**()**;

thread1.Join**()**;

thread2.Join**()**;

thread3.Join**()**;

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

Console.ReadLine**()**;

**}**

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

**{**

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

**{**

Total++;

**}**

**}**

**}**

**}**

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

##### ****How do protect shared resources from concurrent access?****

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

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

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

**{**

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

**{**

Interlocked.Increment**(ref** Total**)**;

**}**

**}**

###### ****The other option is to use a lock.****

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

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

**{**

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

**{**

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

**{**

Total++;

**}**

**}**

**}**

##### ****Which option is better?****

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

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

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

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

**{**

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

Thread thread1 = new Thread**(**Program.AddOneMillion**)**;

Thread thread2 = new Thread**(**Program.AddOneMillion**)**;

Thread thread3 = new Thread**(**Program.AddOneMillion**)**;

thread1.Start**()**;

thread2.Start**()**;

thread3.Start**()**;

thread1.Join**()**;

thread2.Join**()**;

thread3.Join**()**;

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

stopwatch.Stop**()**;

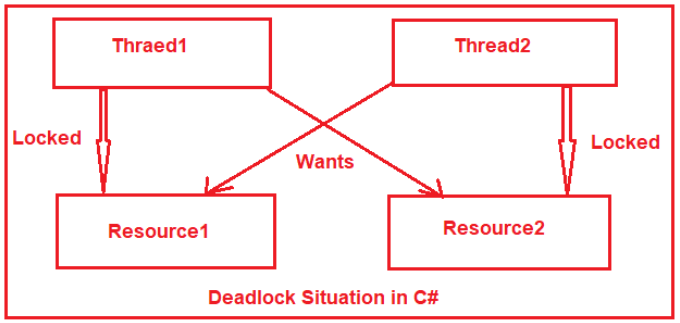
Console.WriteLine**(**"Time Taken in Ticks = " + stopwatch.ElapsedTicks**)**;

**}**

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

##### ****Explain why and how a deadlock can occur in multithreading with an example?****

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



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

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

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

##### ****What is AutoResetEvent and how it is different from ManualResetEvent?****

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

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

##### ****What is the Semaphore?****

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

**class** Program

**{**

**static** Semaphore semaphore = new Semaphore**(**5, 5**)**;

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

**{**

Task.Factory.StartNew**(()** =**>**

**{**

**for** **(int** i = 1; i **<**= 15; ++i**)**

**{**

PrintSomething**(**i**)**;

**if** **(**i % 5 == 0**)**

**{**

Thread.Sleep**(**2000**)**;

**}**

**}**

**})**;

Console.ReadLine**()**;

**}**

**public** **static** **void** PrintSomething**(int** number**)**

**{**

semaphore.WaitOne**()**;

Console.WriteLine**(**number**)**;

semaphore.Release**()**;

**}**

**}**

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

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

##### ****What is Mutex and how it is different from other Synchronization mechanisms?****

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

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

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

##### ****What is synchronization and why it is important?****

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

To stop the dirty reads we use synchronization primitives.

##### ****What is LiveLock?****

This is one of the most frequently asked Deadlock Interview Questions in C#. A livelock is very similar to a deadlock except for involved threads states are continually changing their state but still, they cannot complete their work.

A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

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

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

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

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

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

**namespace** *ExceptionHandlingDemo*

**{**

**class** Program

**{**

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

**{**

**int** a = 20;

**int** b = 10;

**int** c;

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

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

c = a / b;

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

Console.ReadKey**()**;

**}**

**}**

**}**

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

**namespace** *ExceptionHandlingDemo*

**{**

**class** Program

**{**

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

**{**

**int** a = 20;

**int** b = 0;

**int** c;

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

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

c = a / b;

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

Console.ReadKey**()**;

**}**

**}**

**}**

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

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

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

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

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

##### ****Explain the difference between Error and Exception in C#?****

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

Exceptions are those which can be handled at the runtime whereas errors cannot be handled.

An exception is an object of a type deriving from the System.Exception class. The exception is thrown by the CLR (Common Language Runtime) when errors occur that are nonfatal and recoverable by user programs. It is meant to give you an opportunity to do something with a throw statement to transfer control to a catch clause in a try block.

The error is something that most of the time we cannot handle it. Errors are the unchecked exception and the developer is not required to do anything with these. Errors normally tend to signal the end of our program, it typically cannot be recovered from and should cause us to exit from the current program. It should not be caught or handled.

All the Errors are Exceptions but the reverse is not true. In general, Errors are which nobody can control or guess when it happened on the other hand Exception can be guessed and can be handled.

##### ****Explain about try-catch implementation.****

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

1. Try
2. Catch
3. finally

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

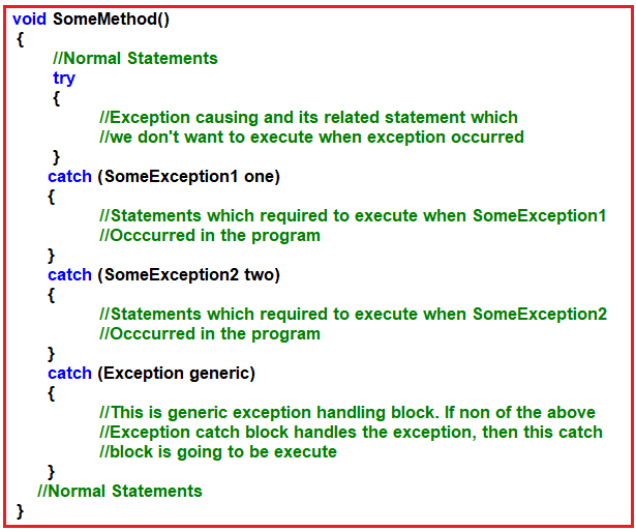
****Catch:****The catch block is used to catch the exception that is thrown from its corresponding try block. It has logic to take necessary actions on that caught exception.

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

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

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

****SYNTAX TO USE TRY CATCH:****



##### ****What are SOLID Design Principles?****

SOLID Design Principles represent five Design Principles used to make software designs more understandable, flexible, and maintainable. The Five SOLID Design Principles are as follows:

1. ****S****stands for the **[Single Responsibility Principle](https://dotnettutorials.net/lesson/single-responsibility-principle/" \t "https://dotnettutorials.net/course/solid-design-principles/_blank)**, also known as ****SRP.**** This Principle states that each software module or class should have only one reason to change. In other words, each module or class should have only one responsibility.
2. ****O****stands for the **[Open-Closed Principle](https://dotnettutorials.net/lesson/open-closed-principle/" \t "https://dotnettutorials.net/course/solid-design-principles/_blank)**, also known as ****OSP.**** The Open-Closed Principle states that software entities, such as modules, classes, functions, etc., should be open for extension but closed for modification.
3. ****L****stands for the **[Liskov Substitution Principle](https://dotnettutorials.net/lesson/liskov-substitution-principle/" \t "https://dotnettutorials.net/course/solid-design-principles/_blank)**, also known as ****LSP****. This Principle states that the object of a derived class should be able to replace an object of the base class without causing any errors in the system or modifying the behavior of the base class. That means the child class objects should be able to replace parent class objects without changing the correctness or behavior of the program.
4. ****I****stand for the **[Interface Segregation Principle](https://dotnettutorials.net/lesson/interface-segregation-principle/" \t "https://dotnettutorials.net/course/solid-design-principles/_blank)**,****also known as ****ISP****. This Principle states that Clients should not be forced to implement any methods they don’t use. Rather than one fat interface, numerous little interfaces are preferred based on groups of methods, with each interface serving one submodule.
5. ****D****stands for **[Dependency Inversion Principle](https://dotnettutorials.net/lesson/dependency-inversion-principle/" \t "https://dotnettutorials.net/course/solid-design-principles/_blank)**, also known as ****DIP****. This Principle states that high-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions. Secondly, abstractions should not depend upon details. Details should depend upon abstractions.