**C# Important Interview Questions**

## <https://www.c-sharpcorner.com/UploadFile/puranindia/C-Sharp-interview-questions/>

[https://www.interviewbit.com/web-api-interview-questions/#what-is-asp-net-web-api](https://www.interviewbit.com/web-api-interview-questions/%23what-is-asp-net-web-api)

<https://ankitsharmablogs.com/csharp-coding-questions-for-technical-interviews/>

# **JWT Authentication in ASP.NET Core Web API**

<https://code-maze.com/authentication-aspnetcore-jwt-1/>

**Instance:**  a copy of an entity in a programming language

1. What is C#?

C# is pronounced as "C-Sharp". It is an object-oriented programming language provided by Microsoft that runs on .Net Framework.

### 2. What is the difference between static, public, and void?

Public declared variables can be accessed from anywhere in the application. Static declared variables can be accessed globally without needing to create an instance of the class. Void is a type modifier which states the method and is used to specify the return type of a method in C#.

### 3. What is an object?

An object is a class instance that can be used to access class methods and properties. The "New" keyword can be used to construct an object.

In C#, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.

Let's see an example to create object using new keyword.

1. Student s1 = **new** Student();//creating an object of Student

### 4. Define Constructors.

A [constructor](https://www.simplilearn.com/tutorials/c-sharp-tutorial/c-sharp-constructor" \t "_blank" \o "constructor) is a member function with the same name as its class. The constructor is automatically invoked when an object is created. While the class is being initialized, it constructs all the values of data members.

In C#, constructor is a special method which is invoked automatically at the time of object creation. The main use of constructors is to initialize the private fields of the class while creating an instance for the class. When you have not created a constructor in the class, the compiler will automatically create a default constructor of the class. The default constructor initializes all numeric fields in the class to zero and all string and object fields to null.

Some of the key points regarding constructor are

* A class can have any number of constructors.
* A constructor doesn't have any return type, not even void.
* A static constructor cannot be a parameterized constructor.
* Within a class, you can create one static constructor only.

In C#, constructors can be divided into 5 types

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

## **1. Default Constructor**

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

### C# Default Constructor Example: Having Main() within class

1. **using** System;
2. **public** **class** Employee
3. {
4. **public** Employee()
5. {
6. Console.WriteLine("Default Constructor Invoked");
7. }
8. **public** **static** **void** Main(**string**[] args)
9. {
10. Employee e1 = **new** Employee();
11. Employee e2 = **new** Employee();
12. }
13. }

Output:

Default Constructor Invoked

Default Constructor Invoked

## **2. Parameterized Constructor**

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

1. **using** System;
2. **public** **class** Employee
3. {
4. **public** **int** id;
5. **public** String name;
6. **public** **float** salary;
7. **public** Employee(**int** i, String n, **float** s)
8. {
9. id = i;
10. name = n;
11. salary = s;
12. }
13. **public** **void** display()
14. {
15. Console.WriteLine(id + " " + name+" "+salary);
16. }
17. }
18. **class** TestEmployee{
19. **public** **static** **void** Main(**string**[] args)
20. {
21. Employee e1 = **new** Employee(101, "Sonoo", 890000f);
22. Employee e2 = **new** Employee(102, "Mahesh", 490000f);
23. e1.display();
24. e2.display();
26. }
27. }

Output:

101 Sonoo 890000

102 Mahesh 490000

## **3. Copy Constructor**

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

public employee(employee emp)

{

name=emp.name;

age=emp.age;

}

public employee(string name, int age) // Instance constructor.

{

this.name = name;

this.age = age;

}

The copy constructor is invoked by instantiating an object of type employee and bypassing it the object to be copied.

employee emp1 = new employee("Vithal", 23);

employee emp2 = new employee(emp1);

## **4. Static Constructor**

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

**Some key points of a static constructor are:**

1. A static constructor does not take access modifiers or have parameters.
2. A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.
3. A static constructor cannot be called directly.
4. The user has no control over when the static constructor is executed in the program.
5. 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.

## **5. Private Constructor**

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

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

### 5. What are Jagged Arrays?

The Array which comprises elements of type array is called Jagged Array. The elements in Jagged Arrays can be of various dimensions and sizes.

In C#, jagged array is also known as "array of arrays" because its elements are arrays. The element size of jagged array can be different.

### Declaration of Jagged array

Let's see an example to declare jagged array that has two elements.

1. **int**[][] arr = **new** **int**[2][];

### C# Jagged Array Example

Let's see a simple example of jagged array in C# which declares, initializes and traverse jagged arrays.

1. **public** **class** JaggedArrayTest
2. {
3. **public** **static** **void** Main()
4. {
5. **int**[][] arr = **new** **int**[2][];// Declare the array
7. arr[0] = **new** **int**[] { 11, 21, 56, 78 };// Initialize the array
8. arr[1] = **new** **int**[] { 42, 61, 37, 41, 59, 63 };
10. // Traverse array elements
11. **for** (**int** i = 0; i < arr.Length; i++)
12. {
13. **for** (**int** j = 0; j < arr[i].Length; j++)
14. {
15. System.Console.Write(arr[i][j]+" ");
16. }
17. System.Console.WriteLine();
18. }
19. }
20. }

Output:

11 21 56 78

42 61 37 41 59 63

### Initialization of Jagged array upon Declaration

Let's see an example to initialize the jagged array while declaration.

1. **int**[][] arr = **new** **int**[3][]{
2. **new** **int**[] { 11, 21, 56, 78 },
3. **new** **int**[] { 2, 5, 6, 7, 98, 5 },
4. **new** **int**[] { 2, 5 }
5. };

### 6. What is the difference between out and ref parameters?

When an argument is passed as a ref, it must be initialized before it can be passed to the method. An out parameter, on the other hand, need not to be initialized before passing to a method.

Both indicate that an argument/parameter is passed by reference. By default, parameters are passed to a method by value. By using these keywords (ref and out) we can pass a parameter by reference. Properties are not variables, therefore they cannot be passed as an out or ref parameter.

### 7. What is the benefit of ‘using’ statement in C#?

### When you use the using statement, C# will automatically call the Dispose method on the object when it's no longer needed. This means you don't have to manually call the Dispose method or worry about forgetting to do so. The using statement takes care of this for you!

### 8. What is serialization?

### In C#, serialization is the process of converting object into stream of bytes so that it can be saved to memory, file, send over network, or stored in a database. The reverse process of serialization is called deserialization.

### **JSON serialization (JavaScript Object Notation)** **:** It is process of converting custom .Net object to a json string.

### IMG_256

### 9. Can “this” command be used within a static method?

No. This is because only static variables/methods can be used in a static method.

### 10. Differentiate between Break and Continue Statement.

Continue statement - Used in jumping over a particular iteration and getting into the next iteration of the[loop.](https://www.simplilearn.com/tutorials/asp-dot-net-tutorial/for-each-loop" \t "_blank" \o "loop.)

Break statement - Used to skip the next statements of the current iteration and come out of the loop.

### 11. List the different types of comments in C#.

 The different types of comments in C# are:

* XML comments

Example -

/// example of XML comment

* Single Line comments

Example -

// example of single-line comment

* Multi-line comments

Example -

/\* example of an

multiline comment \*/

12. Explain the four steps involved in the C# code compilation.

Four steps of code compilation in C# include -

* Source code compilation in managed code.
* Newly created code is clubbed with assembly code.
* The Common Language Runtime (CLR) is loaded.
* Assembly execution is done through CLR.

13. Discuss the various methods to pass parameters in a method.

The various methods of passing parameters in a method include -

* Output parameters: Lets the method return more than one value.
* Value parameters: The formal value copies and stores the value of the actual argument, which enables the manipulation of the formal parameter without affecting the value of the actual parameter.
* Reference parameters: The memory address of the actual parameter is stored in the formal argument, which means any change to the formal parameter would reflect on the actual argument too.

14. Name all the C# access modifiers.

C# Access modifiers or specifiers are the keywords that are used to specify accessibility or scope of variables and functions in the C# application.

The C# access modifiers are -

* **Private Access Modifier -** A private attribute or method is one that can only be accessed from within the class.
* **Public Access Modifier -** When an attribute or method is declared public, it can be accessed from anywhere in the code. It is accessible by any other code in the same assembly or another assembly that references it.
* **Internal Access Modifier -** When a property or method is defined as internal, it can only be accessible from the current assembly point of that class but not from another assembly.
* **Protected Access Modifier -** When a user declares a method or attribute as protected, it can only be accessed by members of that class and those who inherit it.
* **Protected Internal Access Modifier -** Variable or function declared **protected internal** can be accessed in the assembly in which it is declared or by derived class in another assembly.

**Note**: Protected Internal member works as Internal within the same assembly and works as Protected for outside the assembly.

Example 1 - Protected Internal Access Modifier

1. **using** System;
2. **namespace** AccessSpecifiers
3. {
4. **class** InternalTest
5. {
6. **protected** **internal** **string** name = "Shantosh Kumar";
7. **protected** **internal** **void** Msg(**string** msg)
8. {
9. Console.WriteLine("Hello " + msg);
10. }
11. }
12. **class** Program
13. {
14. **static** **void** Main(**string**[] args)
15. {
16. InternalTest internalTest = **new** InternalTest();
17. // Accessing protected internal variable
18. Console.WriteLine("Hello " + internalTest.name);
19. // Accessing protected internal function
20. internalTest.Msg("Peter Decosta");
21. }
22. }
23. }

**Output:**

Hello Shantosh Kumar

Hello Peter Decosta

Example 2 - Protected Internal Access Modifier

// Assembly1.cs

// Compile with: /target:library

public class BaseClass

{

protected internal int myValue = 0;

}

class TestAccess

{

void Access()

{

var baseObject = new BaseClass();

baseObject.myValue = 5;

}

}

// Assembly2.cs

// Compile with: /reference:Assembly1.dll

class DerivedClass : BaseClass

{

static void Main()

{

var baseObject = new BaseClass();

var derivedObject = new DerivedClass();

// Error CS1540, because myValue can only be accessed by

// classes derived from BaseClass.

// baseObject.myValue = 10;

// OK, because this class derives from BaseClass.

derivedObject.myValue = 10;

}

}

### 15. Mention all the advantages of C#.

The following are the [advantages of C#](https://www.simplilearn.com/c-sharp-programming-for-beginners-article" \t "_blank" \o "advantages of C#) -

* C# is component-oriented.
* It is an object-oriented language.
* The syntax is really easy to grasp.
* It is easier to learn.
* C# is part of the framework called .NET

16. Why do we use C# language?

Below are the reasons why we use the C# language -

* C# is a component-oriented language.
* It is easy to pass parameters in the C# language.
* The C# language can be compiled on many platforms.
* The C# language follows a structured approach.
* It is easy to learn and pick up.
* The C# language produces really efficient and readable programmes.

17. Mention the features of C# briefly.

Some of the main features of C# are -

* C# is a safely typed and managed language.
* C# is object-oriented in nature.
* C# is a Cross-platform friendly language.
* C# is a platform-independent language when it comes to compilation.
* C# is general purpose in nature.
* C# is used in implementing Destructors and Constructors.
* C# is part of the .NET framework.
* C# is an easy-to-learn and easy-to-grasp language.
* C# is a structured language.

### 18. What is meant by Unmanaged or Managed Code?

Any language that is written in the .NET framework is managed code. Managed code use CLR, which looks after your applications by managing memory, handling security, allowing cross-language debugging, etc.

The code developed outside the .NET framework is known as unmanaged code. Applications that do not run under the control of the CLR are said to be unmanaged. This code is executed with the help of wrapper classes.

### 19. What is meant by an Abstract Class?

In C#, abstract class is a class which is declared abstract. It can have abstract and non-abstract methods. It cannot be instantiated. Its implementation must be provided by derived classes. Here, derived class is forced to provide the implementation of all the abstract methods.

Abstract classes are the way to achieve abstraction in C#. Abstraction in C# is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1. Abstract class
2. Interface

Abstract class and interface both can have abstract methods which are necessary for abstraction.

## **Abstract Method**

A method which is declared abstract and has no body is called abstract method. It can be declared inside the abstract class only. Its implementation must be provided by derived classes. For example:

1. **public** **abstract** **void** draw();

#### **An abstract method in C# is internally a virtual method so it can be overridden by the derived class.**

You can't use static and virtual modifiers in abstract method declaration.

1. **using** System;
2. **public** **abstract** **class** Shape
3. {
4. **public** **abstract** **void** draw();
5. }
6. **public** **class** Rectangle : Shape
7. {
8. **public** **override** **void** draw()
9. {
10. Console.WriteLine("drawing rectangle...");
11. }
12. }
13. **public** **class** Circle : Shape
14. {
15. **public** **override** **void** draw()
16. {
17. Console.WriteLine("drawing circle...");
18. }
19. }
20. **public** **class** TestAbstract
21. {
22. **public** **static** **void** Main()
23. {
24. Shape s;
25. s = **new** Rectangle();
26. s.draw();
27. s = **new** Circle();
28. s.draw();
29. }
30. }

Output:

drawing ractangle...

drawing circle...

### 20. Differentiate between finalize blocks and finalize Method.

Once the try and catch blocks have been completed, the finalize block is called since it is used for exception handling. No matter if the exception has been captured, this block of code is run. In general, the code in this block is cleaner.

Just before garbage collection, the finalize method is called. The main priorities of the finalize method are to clean up unmanaged code, which is automatically triggered whenever an instance is not re-called.

### 21. What is meant by an Interface?

An interface is a class that does not have any implementation. Only the declarations of events, properties, and attributes are included.

Interface in C# is a blueprint of a class. It is like abstract class because all the methods which are declared inside the interface are abstract methods. It cannot have method body and cannot be instantiated.

It is used *to achieve multiple inheritance* which can't be achieved by class. It is used *to achieve fully abstraction* because it cannot have method body.

Its implementation must be provided by class or struct. The class or struct which implements the interface, must provide the implementation of all the methods declared inside the interface.

C# interface example

Let's see the example of interface in C# which has draw() method. Its implementation is provided by two classes: Rectangle and Circle.

1. **using** System;
2. **public** **interface** Drawable
3. {
4. **void** draw();
5. }
6. **public** **class** Rectangle : Drawable
7. {
8. **public** **void** draw()
9. {
10. Console.WriteLine("drawing rectangle...");
11. }
12. }
13. **public** **class** Circle : Drawable
14. {
15. **public** **void** draw()
16. {
17. Console.WriteLine("drawing circle...");
18. }
19. }
20. **public** **class** TestInterface
21. {
22. **public** **static** **void** Main()
23. {
24. Drawable d;
25. d = **new** Rectangle();
26. d.draw();
27. d = **new** Circle();
28. d.draw();
29. }
30. }

Output:

drawing ractangle...

drawing circle...

### 22. What is meant by a Partial Class?

It provides a special ability to implement the functionality of a single class into multiple files and all these files are combined into a single class file when the application is compiled. A partial class is created by using a ***partial***keyword. This keyword is also useful to split the functionality of methods, interfaces, or structure into multiple files.

A [partial class](https://www.simplilearn.com/tutorials/c-sharp-tutorial/partial-class-in-c-sharp" \t "_blank" \o "partial class) effectively breaks a class's definition into various classes in the same or other source code files. A class definition can be written in numerous files, but it is compiled as a single class at runtime, and when a class is formed, all methods from all source files can be accessed using the same object. The keyword 'partial' denotes this.

<https://www.geeksforgeeks.org/partial-classes-in-c-sharp/>

### 23. What is the difference between read-only and constants?

During the time of compilation, constant variables are declared as well as initialized. It’s not possible to change this particular value later. On the other hand, read-only is used after a value is assigned at run time.

### 24. What is an interface class?

An interface class is an abstract class with only public abstract methods. Only declaration is there in these methods, but not the implementation. They must be implemented in the inherited classes.

### 25. What are reference types and value types?

A value type holds a data value inside its memory space. Reference type, on the other hand, keeps the object’s address where the value is stored. It is, essentially, a pointer to a different memory location.

### 26. What are sealed classes in C#?

C# sealed keyword applies restrictions on the class and method. If you create a sealed class, it cannot be derived. If you create a sealed method, it cannot be overridden.

#### **Note: Structs are implicitly sealed therefore they can't be inherited.**

When a restriction needs to be placed on the class that needs to be inherited, sealed classes are created. In order to prevent any derivation from a class, a sealed modifier is used. Compile-time error occurs when a sealed class is forcefully specified as a base class.

#### **Note: Local variables can't be sealed.**

1. **using** System;
2. **public** **class** TestSealed
3. {
4. **public** **static** **void** Main()
5. {
6. **sealed** **int** x = 10;
7. x++;
8. Console.WriteLine(x);
9. }
10. }

Output:

Compile Time Error: Invalid expression term 'sealed'

### 27. What is method overloading?

Method overloading is the process of generating many methods in the same class with the same name but distinct signatures. The compiler utilizes overload resolution to identify which method to invoke when we compile.

Having two or more methods with same name but different in parameters in the same class, is known as method overloading in C#.

The **advantage** of method overloading is that it increases the readability of the program because you don't need to use different names for same action.

You can perform method overloading in C# by two ways:

1. By changing number of arguments
2. By changing data type of the arguments

### 28. What is the difference between Arraylist and Array?

An array only has items of the same type and its size is fixed. Arraylist can store items of different types and it does not have a fixed size.

The array is a fixed sized data structure thus, the array always needs to mention the size of the elements. On the other hand, ArrayList is not a fixed sized data structure, thus there is no need to mention the size of the ArrayList especially creating its [object.](https://www.simplilearn.com/tutorials/java-tutorial/java-classes-and-objects" \t "_blank" \o "object.) Also we can add more elements to Arraylist, though having some initial elements present in it.

### 29. Is it possible for a private virtual method to be overridden?

A private virtual method cannot be overridden as it can’t be accessed outside the class.

### 30. What are the differences between System.String and System.Text.StringBuilder classes?

System.String is absolute. When a string variable’s value is modified, a new memory is assigned to the new value. The previous memory allocation gets released. System.StringBuilder, on the other hand, is designed so it can have a mutable string in which a plethora of operations can be performed without the need for allocation of a separate memory location for the string that has been modified.

### 31. How can the Array elements be sorted in descending order?

You can use the Using Sort() methods and then Reverse() method.

### 32. What’s the difference between an abstract and interface class?

All methods in interfaces have only a declaration but no definition. We can have some strong methods in an abstract class. All methods in an interface class are public. Private methods may exist in an abstract class.

### 33. What is the difference between Dispose() and Finalize() methods?

Dispose() is used when an object is required to release any unmanaged resources in it. Finalize(), on the other hand, doesn’t assure the garbage collection of an object even though it is used for the same function.

### 34. What are circular references?

When two or more resources are dependent on each, it causes a lock condition, and the resources become unusable. This is called a circular reference.

### 35. What are generics in C# .NET?

In order to reduce code redundancy, raise type safety, and performance, generics can be used in order to make code classes that can be reused. Collection classes can be created using generics.

Generic is a concept that allows us to define classes and methods with placeholder. C# compiler replaces these placeholders with specified type at compile time. The concept of generics is used to create general purpose classes and methods.

To define generic class, we must use angle **<>** brackets. The angle brackets are used to declare a class or method as generic type. In the following example, we are creating generic class that can be used to deal with any type of data.

C# Generic class example

1. **using** System;
2. **namespace** CSharpProgram
3. {
4. **class** GenericClass<T>
5. {
6. **public** GenericClass(T msg)
7. {
8. Console.WriteLine(msg);
9. }
10. }
11. **class** Program
12. {
13. **static** **void** Main(**string**[] args)
14. {
15. GenericClass<**string**> gen   = **new** GenericClass<**string**> ("This is generic class");
16. GenericClass<**int**>    genI  = **new** GenericClass<**int**>(101);
17. GenericClass<**char**>   getCh = **new** GenericClass<**char**>('I');
18. }
19. }
20. }

Output:

This is generic class

101

I

C# allows us to create generic methods also. In the following example, we are creating generic method that can be called by passing any type of argument.

Generic Method Example

1. **using** System;
2. **namespace** CSharpProgram
3. {
4. **class** GenericClass
5. {
6. **public** **void** Show<T>(T msg)
7. {
8. Console.WriteLine(msg);
9. }
10. }
11. **class** Program
12. {
13. **static** **void** Main(**string**[] args)
14. {
15. GenericClass genC = **new** GenericClass();
16. genC.Show("This is generic method");
17. genC.Show(101);
18. genC.Show('I');
19. }
20. }
21. }

Output:

This is generic method

101

I

### 36. What is an object pool in .NET?

A container that has objects which are ready to be used is known as an object pool. It helps in tracking the object which is currently in use and the total number of objects present in the pool. This brings down the need for creating and re-creating objects.

### 37. List down the most commonly used types of exceptions in .NET

Commonly used types of exceptions in .NET are:

* ArgumentException
* ArithmeticException
* DivideByZeroException
* OverflowException
* InvalidCastException
* InvalidOperationException
* NullReferenceException
* OutOfMemoryException
* StackOverflowException

### 38. What are Custom Exceptions?

In some cases, errors have to be handled according to user requirements. Custom exceptions are used in such cases.

### 39. What are delegates?

In C#, delegate is a reference to the method. It works like function pointer in C and C++. But it is object-oriented, secured and type-safe than function pointer.

For static method, delegate encapsulates method only. But for instance method, it encapsulates method and instance both.

The best use of delegate is to use as event.

Internally a delegate declaration defines a class which is the derived class of **System.Delegate**.

## **C# Delegate Example**

Let's see a simple example of delegate in C# which calls add() and mul() methods.

1. **using** System;
2. **delegate** **int** Calculator(**int** n);//declaring delegate
4. **public** **class** DelegateExample
5. {
6. **static** **int** number = 100;
7. **public** **static** **int** add(**int** n)
8. {
9. number = number + n;
10. **return** number;
11. }
12. **public** **static** **int** mul(**int** n)
13. {
14. number = number \* n;
15. **return** number;
16. }
17. **public** **static** **int** getNumber()
18. {
19. **return** number;
20. }
21. **public** **static** **void** Main(**string**[] args)
22. {
23. Calculator c1 = **new** Calculator(add);//instantiating delegate
24. Calculator c2 = **new** Calculator(mul);
25. c1(20);//calling method using delegate
26. Console.WriteLine("After c1 delegate, Number is: " + getNumber());
27. c2(3);
28. Console.WriteLine("After c2 delegate, Number is: " + getNumber());
30. }
31. }

Output:

After c1 delegate, Number is: 120

After c2 delegate, Number is: 360

### 40. What is the difference between method overriding and method overloading?

In method overriding, the relevant method definition is replaced in the derived class, which changes the method behaviour. When it comes to method overloading, a method is created with the same name and is in the same class while having different signatures.

### 41. How do you inherit a class into another class in C#?

In C#, colon can be used as an inheritance operator. You need to place a colon and follow it with the class name.

### 42. What are the various ways that a method can be overloaded??

**Different data types** can be used for a parameter in order for a method to be overloaded; **different orders of parameters** as well as **different numbers of parameters** can be used.

### 43. Why can't the accessibility modifier be specified for methods within the interface?

In an interface, there are virtual methods which do not come with method definition. All the methods present are to be overridden in the derived class. This is the reason they are all public.

### 44. How can we set the class to be inherited, but prevent the method from being overridden?

To set the class to be inherited, it needs to be declared as public. The method needs to be sealed to prevent any overrides.

### 45. What happens if the method names in the inherited interfaces conflict?

A problem could arise when the methods from various interfaces expect different data. But when it comes to the compiler itself, there shouldn’t be an issue.

### 46. What is the difference between a Struct and a Class?

Structs are essentially value-type variables, whereas classes would be reference types.

### 47. How to use nullable types in .Net?

When either normal values or a null value can be taken by value types, they are called nullable types.

There are two ways to declare Nullable types.

Nullable<int> Example;

OR

int? Example;

### 48. How can we make an array with non-standard values?

An array with non-default values can be created using Enumerable.Repeat.

### 49. What is the difference between “is” and “as” operators in c#?

An “is” operator can be used to check an object’s compatibility with respect to a given type, and the result is returned as a Boolean. An “as” operator can be used for casting an object to either a type or a class.

### 50. What is a multicast delegate?

Multicast delegate is when a single delegate comes with multiple handlers. Each handler is assigned to a method.

### 51. What are indexers in C# .NET?

In C#, indexers are called smart arrays. Indexers allow class instances to be indexed in the same way as arrays do.

### 52. What is the distinction between "throw" and "throw ex" in.NET?

The difference between them is that:

* ****“throw”**** preserves the stack trace (the stack trace will point to the method that caused the exception in the first place)
* ****“throw ex”**** does not preserve the stack trace (we will lose the information about the method that caused the exception in the first place. It will seem like the exception was thrown from the place of its catching and re-throwing)

 A stack trace is a record of all methods that have been called and resulted in a specific moment in code.

### 53. What are C# attributes and its significance?

C# gives developers an option to define declarative tags on a few entities. For instance, class and method are known as attributes. The information related to the attribute can be retrieved during runtime by taking the help of Reflection.

In C#, reflection is a *process to get metadata of a type at runtime*. The System.Reflection namespace contains required classes for reflection such as:

* Type
* MemberInfo
* ConstructorInfo
* MethodInfo
* FieldInfo
* PropertyInfo
* TypeInfo
* EventInfo
* Module
* Assembly
* AssemblyName
* Pointer etc.

The System.Reflection.Emit namespace contains classes to emit metadata.

## **C# Type class**

C# Type class represents type declarations for class types, interface types, enumeration types, array types, value types etc. It is found in System namespace. It inherits System.Reflection.MemberInfo class.

## **C# Reflection Example: Get Type**

1. **using** System;
2. **public** **class** ReflectionExample
3. {
4. **public** **static** **void** Main()
5. {
6. **int** a = 10;
7. Type type = a.GetType();
8. Console.WriteLine(type);
9. }
10. }

Output:

System.Int32

## **C# Reflection Example: Print Type Information**

1. **using** System;
2. **using** System.Reflection;
3. **public** **class** ReflectionExample
4. {
5. **public** **static** **void** Main()
6. {
7. Type t = **typeof**(System.String);
8. Console.WriteLine(t.FullName);
9. Console.WriteLine(t.BaseType);
10. Console.WriteLine(t.IsClass);
11. Console.WriteLine(t.IsEnum);
12. Console.WriteLine(t.IsInterface);
13. }
14. }

Output:

System.String

System.Object

true

false

false

### 54. In C#, how do you implement the singleton design pattern?

In a singleton pattern, a class is allowed to have only one instance, and an access point is provided to it globally.

### 56. Is C# code managed or unmanaged code?

C# is a managed code as the runtime of Common language can compile C# code to Intermediate language.

### 57. What is a Console application?

An application that is able to run in the command prompt window is called a console application.

### 58. What are namespaces in C#?

Namespaces allow you to keep one set of names that is different from others. A great advantage of namespace is that class names declared in one namespace don’t clash with those declared in another namespace.

Namespaces in C# are used to organize too many classes so that it can be easy to handle the application.

In a simple C# program, we use System.Console where System is the namespace and Console is the class. To access the class of a namespace, we need to use namespacename.classname. We can use **using** keyword so that we don't have to use complete name all the time.

In C#, global namespace is the root namespace. The global::System will always refer to the namespace "System" of .Net Framework.

### C# namespace example: by fully qualified name

Let's see an example of namespace in C# where one namespace program accesses another namespace program.

1. **using** System;
2. **namespace** First {
3. **public** **class** Hello
4. {
5. **public** **void** sayHello() { Console.WriteLine("Hello First Namespace"); }
6. }
7. }
8. **namespace** Second
9. {
10. **public** **class** Hello
11. {
12. **public** **void** sayHello() { Console.WriteLine("Hello Second Namespace"); }
13. }
14. }
15. **public** **class** TestNamespace
16. {
17. **public** **static** **void** Main()
18. {
19. First.Hello h1 = **new** First.Hello();
20. Second.Hello h2 = **new** Second.Hello();
21. h1.sayHello();
22. h2.sayHello();
23. }
24. }

Output:

Hello First Namespace

Hello Second Namespace

### C# namespace example: by using keyword

Let's see another example of namespace where we are using "using" keyword so that we don't have to use complete name for accessing a namespace program.

1. **using** System;
2. **using** First;
3. **using** Second;
4. **namespace** First {
5. **public** **class** Hello
6. {
7. **public** **void** sayHello() { Console.WriteLine("Hello Namespace"); }
8. }
9. }
10. **namespace** Second
11. {
12. **public** **class** Welcome
13. {
14. **public** **void** sayWelcome() { Console.WriteLine("Welcome Namespace"); }
15. }
16. }
17. **public** **class** TestNamespace
18. {
19. **public** **static** **void** Main()
20. {
21. Hello h1 = **new** Hello();
22. Welcome w1 = **new** Welcome();
23. h1.sayHello();
24. w1.sayWelcome();
25. }
26. }

Output:

Hello Namespace

Welcome Namespace

**59. Assembly vs Namespaces in C#**

Assemblies and namespaces play crucial roles in C# development. Assemblies provide a way to package and deploy code, while namespaces offer a mechanism to organize and structure code within those assemblies.

## **Assemblies :**

An assembly in C# is a logical unit of code that consists of one or more files and contains metadata for the code it contains. Assemblies provide a way to organize, deploy, and version code in a portable manner. They are the building blocks of .NET applications and can be either static or dynamic.

### 60. Difference between SortedList and SortedDictionary in C#.

In C#, [SortedList](https://www.geeksforgeeks.org/c-sharp-sortedlist-with-examples/" \t "_blank)is a collection of key/value pairs which are sorted according to keys. By default, this collection sort the key/value pairs in ascending order. It is of both generic and non-generic type of collection. The generic SortedList is defined in System.Collections.Generic namespace whereas non-generic SortedList is defined under System.Collections namespace.

**Example:**

|  |
| --- |
| // C# program to illustrate how  // to create a sortedlist  using System;  using System.Collections;  class GFG {      // Main Method      static public void Main()      {          // Creating a sortedlist          // Using SortedList class          SortedList my\_Slist = new SortedList();          // Adding key/value pairs in          // SortedList using Add() method          my\_Slist.Add(1.02, "Dog");          my\_Slist.Add(1.07, "Cat");          my\_Slist.Add(1.04, "Rat");          my\_Slist.Add(1.01, "Bird");          foreach(DictionaryEntry pair in my\_Slist)          {              Console.WriteLine("{0} and {1}",                        pair.Key, pair.Value);          }          Console.WriteLine();      }  } |

**Output:**

1.01 and Bird

1.02 and Dog

1.04 and Rat

1.07 and Cat

In C#, [SortedDictionary](https://www.geeksforgeeks.org/sorteddictionary-implementation-in-c-sharp/" \t "_blank)is a generic collection which is used to store the key/value pairs in the sorted form and the sorting is done on the key. SortedDictionary is defined under System.Collection.Generic namespace. It is dynamic in nature means the size of the sorted dictionary is growing according to the need.

**Example:**

|  |
| --- |
| // C# program to illustrate how  // to create a sorted dictionary  using System;  using System.Collections.Generic;  class GFG {      // Main Method      static public void Main()      {          // Creating sorted dictionary          // Using SortedDictionary class          SortedDictionary<int, string> My\_sdict =              new SortedDictionary<int, string>();          // Adding key/value pair in Sorted          // Dictionary Using Add() method          My\_sdict.Add(004, "Roscosmos");          My\_sdict.Add(003, "ESA");          My\_sdict.Add(001, "NASA");          My\_sdict.Add(005, "ISRO");          My\_sdict.Add(002, "CNSA");          Console.WriteLine("Top 5 space agencies 2018:");          // Accessing the key/value pair of the          // SortedDictionary Using foreach loop          foreach(KeyValuePair<int, string> pair in My\_sdict)          {              Console.WriteLine("Rank: {0} and Name: {1}",                                    pair.Key, pair.Value);          }      }  } |

**Output:**

Top 5 space agencies 2018:

Rank: 1 and Name: NASA

Rank: 2 and Name: CNSA

Rank: 3 and Name: ESA

Rank: 4 and Name: Roscosmos

Rank: 5 and Name: ISRO

Below are the some differences between SortedList and SortedDictionary:

|  |  |
| --- | --- |
| **SortedList** | **SortedDictionary** |
| The memory of SortedList is an overhead. | The memory of SortedDictionary is not bottlenecked. |
| In SortedList, the elements are stored in a continuous block in memory. | In SortedDictionary, the elements are stored in separate object that can spread all over the heap. |
| In SoterdList, the memory fragmentation is high. | In SoterdDictionary, the memory fragmentation is low. |
| It require less memory for storage. | It require more memory for storage. |
| In SortedList, less inserts and delete operations are required. | In SortedDictionary, more inserts and delete operations are required. |
| In SortedList, you can access elements using the index. | In SortedDictionary, you can access elements using index or key. Here key access is sufficient there is no need of accessing elements using index. |
| In SortedList, data are already in sorted form. | In SortedDictionary, data are in un-sorted form. |

### 61. What is Singleton design pattern in C#?

Singleton design pattern in C# has just one instance that gives global access to it.

### 62. What is tuple in C#?

Tuple is a data structure to represent a data set that has multiple values that could be related to each other.

The word Tuple means “a data structure which consists of the multiple parts”. So tuple is a data structure which gives you the easiest way to represent a data set which has multiple values that may/may not be related to each other. It ***introduced in .NET Framework 4.0***. In tuple, you can add elements from 1 to 8. If you try to add elements greater than eight, then the compiler will throw an error. Tuples are generally used when you want to create a data structure which contains objects with their properties and you don’t want to create a separate type for that.

**Features of Tuples:**

* It allows us to represent multiple data into a single data set.
* It allows us to create, manipulate, and access data set.
* It return multiple values from a method without using *out*parameter.
* It can also store duplicate elements.
* It allows us to pass multiple values to a method with the help of single parameters.

#### Creating a Tuple

In C#, there are mainly 2 ways to create the tuple which are as follows:

* **Using Constructor of Tuple Class:** You can create a tuple by using the constructor which is provided by **Tuple<T> class**. Where you can store elements starting from one to eight with their type. But you are not allowed to store elements greater than eight in a tuple. If you try to do so then the compiler will throw an error.

**Syntax:**

// Constructor for single elements

Tuple <T1>(T1)

// Constructor for two elements

Tuple <T1, T2>(T1, T2)

// Constructor for eight elements

Tuple <T1, T2, T3, T4, T5, T6, T7, TRest>(T1, T2, T3, T4, T5, T6, T7, TRest)

**Example:**

// C# program to create tuple using tuple constructor.

using System;

public class GFG{

// Main method

static public void Main (){

// Tuple with one element

Tuple<string> My\_Tuple1 = new Tuple<string>("GeeksforGeeks");

// Tuple with three elements

Tuple<string, string, int>My\_Tuple2 = new Tuple<string, string, int>("Romil", "Python", 29);

// Tuple with eight elements

Tuple<int, int, int, int, int, int, int, Tuple<int>>My\_Tuple3 = new Tuple<int, int, int, int, int, int, int, Tuple<int>>(1,2,3,4,5,6,7, new Tuple<int>(8));

}

}

**Using Create Method:** When we use the tuple constructor to create a tuple we need to provide the type of each element stored in the tuple which makes your code cumbersome. So, C# provides another class that is Tuple class which contains the static methods for creating tuple object without providing the type of each element.

**Syntax:**

// Method for 1-tuple

Create(T1)

// Method for 2-tuple

Create(T1, T2)

// Method for 8-tuple

Create(T1, T2, T3, T4, T5, T6, T7, T8)

**Example:**

// C# program to create tuple

// using Create Method

using System;

public class GFG {

// Main method

static public void Main()

{

// Creating 1-tuple

// Using Create Method

var My\_Tuple1 = Tuple.Create("GeeksforGeeks");

// Creating 4-tuple

// Using Create Method

var My\_Tuple2 = Tuple.Create(12, 30, 40, 50);

// Creating 8-tuple

// Using Create Method

var My\_Tuple3 = Tuple.Create(13, "Geeks", 67,

89.90, 'g', 39939, "geek", 10);

}

}

### 63. What are Events?

An event is a notice that something has occurred.

### 64. What is the Constructor Chaining in C#?

### Constructor chaining is a technique in C# that allows one constructor to call another constructor of the same class or a base class.

With Constructor Chaining, an overloaded constructor can be called from another constructor. The constructor must belong to the same class.

### 65. What is a multicasting delegate in C#?

Multicasting of delegates helps users to point to more than one method in a single call.

### 66. What are Accessibility Modifiers in C#?

Access Modifiers are terms that specify a program's member, class, or datatype's accessibility.

### 67. What is a Virtual Method in C#?

In the parent class, a virtual method is declared that can be overridden in the child class. We construct a virtual method in the base class using the virtual keyword, and that function is overridden in the derived class with the Override keyword.

### 68. What is Multithreading with .NET?

Multi-threading refers to the use of multiple threads within a single process. Each thread here performs a different function.

Multithreading in C# is a process in which multiple threads work simultaneously. It is a process to achieve multitasking. It saves time because multiple tasks are being executed at a time. To create multithreaded application in C#, we need to use **System.Threding** namespace.

## **Process and Thread**

A process represents an application whereas a thread represents a module of the application. Process is heavyweight component whereas thread is lightweight. A thread can be termed as lightweight subprocess because it is executed inside a process.

Whenever you create a process, a separate memory area is occupied. But threads share a common memory area.

# **C# Thread Life Cycle**

In C#, each thread has a life cycle. The life cycle of a thread is started when instance of System.Threading.Thread class is created. When the task execution of the thread is completed, its life cycle is ended.

using System;

using System.Threading;

class Program

{

static void Main()

{

// Create and start two separate threads

Thread thread1 = new Thread(DoWork1);

Thread thread2 = new Thread(DoWork2);

thread1.Start();

thread2.Start();

// Wait for both threads to finish

thread1.Join();

thread2.Join();

Console.WriteLine("Both threads have completed.");

}

static void DoWork1()

{

for (int i = 0; i < 5; i++)

{

Console.WriteLine("Thread 1: Step " + i);

Thread.Sleep(1000); // Simulate some work

}

}

static void DoWork2()

{

for (int i = 0; i < 5; i++)

{

Console.WriteLine("Thread 2: Step " + i);

Thread.Sleep(800); // Simulate some work

}

}

}

### 69. In C#, what is a Hash table class?

The Hash table class represents a collection of key/value pairs that are organized based on the hash code of the key.

### 70. What is LINQ in C#?

LINQ refers to Language Integrated Query. It provides .NET languages (like C#) the ability to generate queries to retrieve data from the data source.

### 71. Why can't a private virtual procedure in C# be overridden?

Private virtual methods are not accessible outside of the class.

### 72. What is File Handling in C#?

File handling includes operations such as creating the file, reading from the file, and appending the file, among others.

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

In C#, Get and Set are termed accessors because they use properties. Such private fields are accessed via accessors.

C# Properites doesn't have storage location. C# Properties are extension of fields and accessed like fields.

The Properties have accessors that are used to set, get or compute their values.

A [get](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/get) property accessor is used to return the property value, and a [set](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/set) property accessor is used to assign a new value.

## **Usage of C# Properties**

1. C# Properties can be read-only or write-only.
2. We can have logic while setting values in the C# Properties.
3. We make fields of the class private, so that fields can't be accessed from outside the class directly. Now we are forced to use C# properties for setting or getting values.

C# Properties Example 2: having logic while setting value

1. **using** System;
2. **public** **class** Employee
3. {
4. **private** **string** name;
6. **public** **string** Name
7. {
8. **get**
9. {
10. **return** name;
11. }
12. **set**
13. {
14. name = value+" JavaTpoint";
16. }
17. }
18. }
19. **class** TestEmployee{
20. **public** **static** **void** Main(**string**[] args)
21. {
22. Employee e1 = **new** Employee();
23. e1.Name = "Sonoo";
24. Console.WriteLine("Employee Name: " + e1.Name);
25. }
26. }

Output:

Employee Name: Sonoo JavaTpoint

### 74. What is the Race condition in C#?

When 2 threads access the same resource and try to change it at the same time, we have a race condition.

### 75. Why are Async and Await used in C#?

Async and await keywords of C# were introduced in C# 5.0. They were designed to make it easier to write asynchronous code, which can run in the background while other code is executing.

The "async" keyword marks a method asynchronous, meaning it can be run in the background while another code executes. When you mark a method as async, you can use the "await" keyword to indicate that the method should wait for the result of an asynchronous operation before continuing.

## **Use of 'async' and 'await' in C#**

Asynchronous programming is a programming technique that allows code to be executed concurrently without blocking the execution of the calling thread. In other words, asynchronous code can run in the background while other code is executing. In synchronous programming, each line of code is executed sequentially, and the program waits for each operation to complete before moving on to the next one. This can lead to performance problems, particularly in programs that need to perform long-running operations like I/O or network requests.

Asynchronous programming can be implemented using various techniques, such as callbacks, events, and promises. In C#, the "async" and "await" keywords provide a convenient way to write asynchronous code that looks similar to synchronous code, making it easier to read and maintain

### 76. What is an Indexer in C#?

An indexer allows us to access instances of a class using an index just like an array.

## Define C# Indexer

In C#, we define an indexer just like properties using this keyword followed by [] index notation. For example,

public int this[int index]

{

get

{

return val[index];

}

set

{

val[index] = value;

}

}

* public - access modifier
* int - return type of indexer
* this - indicates we are defining indexer in current class
* int index - access values using integer index position
* get - method that returns values
* set - method that assigns values

## Example: C# indexer

using System;

class Program

{

// declare an array to store elements

private string[] studentName = new string[10];

// define an indexer

public string this[int index]

{

get

{

// return value of stored at studentName array

return studentName[index];

}

set

{

// assigns value to studentName

studentName[index] = value;

}

}

public static void Main()

{

// create instance of Program class

Program obj = new Program();

// insert values in obj[] using indexer i.e index position

obj[0] = "Harry";

obj[1] = "Ron";

obj[2] = "Hermoine";

Console.WriteLine("First element in obj: " + obj[0]);

Console.WriteLine("Second element in obj: " + obj[1]);

}

}

**Output**

First element in obj: Harry

Second element in obj: Ron

Here, the set method assigns values to studentName using index. And the get method returns values stored at studentName.

Here, we have used obj (instance of the Program class) like the studentName array.

**Note:** Without using indexer, we access the studentName array through obj as:

// insert value to studentName array when indexer is not used

obj.studentName[0] = "Harry";

Indexer helps to simplify the syntax.

### 77. What is Thread Pooling in C#?

In C#, a Thread Pool is a group of threads. These threads are used to do work without interfering with the principal thread's operation.

### 78. What information can you provide regarding the XSD file in C#?

XSD stands for XML Schema Definition. The XML file can have any attributes and elements if there is no XSD file associated with it.

### 79. What are I/O classes in C#?

In C#, the System.IO namespace contains multiple classes that are used to conduct different file operations such as creation, deletion, closure, and opening.

### 80. What exactly do you mean by regular expressions in C#?

We used the Regular Expression to check whether the given string matches the pattern or not. Regular Expression or Regex is a sequence of characters that defines the pattern. The pattern can contain numbers, literals, operators, characters, etc. We used the Patterns to search the strings or files. Here we will see if the matches are found or not.

Generally, the Regular Expressions are used for the parsing, finding the strings or validations, etc.

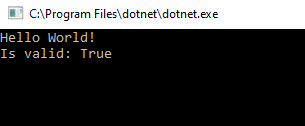
## **Regular Expression Example in C#**

Here we are taking an example to validate the text whether the email is in proper format or not. For this we will use Regex class.

1. using System;
2. using System.Text.RegularExpressions;
4. namespace ConsoleApp1
5. {
6. **class** Program
7. {
8. **static** **void** Main(string[] args)
9. {
10. Console.WriteLine("Hello World!");
11. string email = "support@javatpoint.com";
13. var result = Regex.IsMatch(email, @"^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$");
15. Console.Write("Is valid: {0} ", result);
17. Console.ReadLine();
18. }
19. }
20. }

From the above example, we are validating the input string, which is in a valid format or not. For this, we are using the Regex class. For the validation of the input text, we used the IsMatch method and the pattern of Regular Expression.

After the execution of the above code, we get the result, as shown below:



## **Regex Class method in C#**

To perform the various operations on the input string, the Regex class contains different methods. The table is having the list of various methods of Regex in C#.

|  |  |
| --- | --- |
| **Method** | **Description** |
| IsMatch | We used the IsMatch method for finding out whether the given input string matches the regular expression pattern or not. |
| Matches | Matches method is used to return the text, which matches with the regular expression pattern. |
| Replace | Replace method is used to replace the text, which matches the regular expression pattern. |
| Split | We used the Split method for splitting the string into the array of the substring at those positions, which matches with the regular expression pattern. |

The above method of Regex class is used for the validation, replacement, or splitting the values of the string with the regular expression pattern, which is based on the requirements.

**Q1: What is C# (C-Sharp), and what is its primary use?**

**Answer:** C# is a modern, object-oriented programming language developed by Microsoft. It is primarily used for building Windows desktop applications, web applications, and various types of software on the .NET platform.

**Q2: What is the difference between C# and .NET?**

**Answer:** C# is a programming language, while .NET is a framework. C# is one of the languages supported by the .NET framework. .NET provides libraries and runtime for developing and executing applications written in various languages, including C#.

**Q3: What is an object-oriented programming (OOP) language, and how does C# support OOP?**

**Answer:** OOP is a programming paradigm that uses objects and classes to model real-world entities. C# supports OOP through features like classes, objects, inheritance, abstraction, encapsulation, and polymorphism.

**Q4: How do you declare and initialize variables in C#?**

**Answer:** Variables in C# are declared using a data type followed by the variable name. They can be initialized at the time of declaration or later. For example:

int age = 30; // Declaration and initialization

string name; // Declaration

name = "John"; // Initialization

**Q5: What is the difference between value types and reference types in C#?**

**Answer:** Value types (e.g., **int**, **float**, **structs**) store their data directly, while reference types (e.g., **class**, **string**, **objects**) store references to their data in memory. Value types are stored on the stack, while reference types are stored on the heap.

**Q6: How do you create and use a class in C#?**

**Answer:** You can create a class using the **class** keyword, define its properties and methods, and then instantiate it to create objects. For example:

class Person

{

public string Name { get; set; }

public int Age { get; set; }

}

// Creating an object

Person person = new Person();

person.Name = "John";

person.Age = 30;

**Q7: What is inheritance in C#, and how is it used to create derived classes?**

**Answer:** Inheritance is a mechanism in which a new class (derived or subclass) can inherit properties and methods from an existing class (base or superclass). It allows code reuse and the creation of a hierarchy of classes.

**Q8: What is method overloading in C#?**

**Answer:** Method overloading allows a class to have multiple methods with the same name but different parameters. It provides flexibility in calling methods with varying arguments while using a single method name.

**Q9: What is a constructor in C#, and how is it used to initialize objects?**

**Answer:** A constructor is a special method in a class that is called when an object of the class is created. It is used to initialize object properties and perform setup tasks. In C#, a constructor has the same name as the class.

**Q10: What is encapsulation, and how does it relate to access modifiers in C#?**

**Answer:** Encapsulation is the concept of bundling data (fields or properties) and methods (functions) that operate on the data into a single unit (a class). Access modifiers like **public**, **private**, and **protected** control the visibility and accessibility of members within and outside the class.

**Q11: What is exception handling in C#, and how do you catch and handle exceptions using try-catch blocks?**

**Answer:** Exception handling is a mechanism for dealing with runtime errors. You can catch and handle exceptions using **try-catch** blocks. The **try** block contains code that may throw an exception, and the **catch** block specifies how to handle the exception.

**Q12: What is the purpose of the using statement in C#, and how does it relate to resource management?**

**Answer:** The **using** statement is used for resource management, such as managing file streams or database connections. It ensures that resources are properly disposed of when they are no longer needed, even if an exception is thrown.

**Q13: How do you work with collections (e.g., lists, arrays) in C#?**

**Answer:** Collections are managed using classes like **List<T>**, **Array**, and **Dictionary<TKey, TValue>**. You can add, remove, and manipulate elements within collections.

**Q14: What is the purpose of the async and await keywords in C #, and how are they used for asynchronous programming?**

**Answer:** **async** and **await** are used for writing asynchronous code in C#. **async** is used to mark a method as asynchronous, and **await** keyword is used to indicate the method should wait for the result of an asynchronous operation before continuing.

**Q16: How do you work with files and directories in C#?**

**Answer:** You can work with files and directories using classes from the **System.IO** namespace, such as **File**, **Directory**, **Path**, and **FileInfo**. These classes allow you to create, read, write, move, and delete files and directories.

**Q17: What is garbage collection in C#, and how does it manage memory?**

**Answer:** Garbage collection is an automatic memory management process in C# that tracks and frees up memory that is no longer in use by the program. It helps prevent memory leaks and manages memory efficiently.

**Q18: What is polymorphism in C#, and how is it achieved through method overriding?**

Polymorphism is a Greek word, meaning "one name many forms". In other words, one object has many forms or has one name with multiple functionalities. "Poly" means many and "morph" means forms. Polymorphism provides the ability to a class to have multiple implementations with the same name. Object behaves differently in different-different situation. It is one of the core principles of Object Oriented Programming after encapsulation, abstraction and inheritance.

There are two types of polymorphism in C#: compile time polymorphism and runtime polymorphism. Compile time polymorphism is achieved by method overloading and operator overloading in C#. It is also known as static binding or early binding. Runtime polymorphism in achieved by method overriding which is also known as dynamic binding or late binding.

C# Runtime Polymorphism Example

Let's see an example of runtime polymorphism in C# where we are having two derived classes.

1. **using** System;
2. **public** **class** Shape{
3. **public** **virtual** **void** draw(){
4. Console.WriteLine("drawing...");
5. }
6. }
7. **public** **class** Rectangle: Shape
8. {
9. **public** **override** **void** draw()
10. {
11. Console.WriteLine("drawing rectangle...");
12. }
14. }
15. **public** **class** Circle : Shape
16. {
17. **public** **override** **void** draw()
18. {
19. Console.WriteLine("drawing circle...");
20. }
22. }
23. **public** **class** TestPolymorphism
24. {
25. **public** **static** **void** Main()
26. {
27. Shape s;
28. s = **new** Shape();
29. s.draw();
30. s = **new** Rectangle();
31. s.draw();
32. s = **new** Circle();
33. s.draw();
35. }
36. }

Output:

drawing...

drawing rectangle...

drawing circle...

**Q19: What is Encapsulation in C#?**

Encapsulation is the concept of wrapping data and methods into a single unit. It collects data members and member functions into a single unit called class. The purpose of encapsulation is to prevent alteration of data from outside. This data can only be accessed by getter functions of the class.

A fully encapsulated class has getter and setter functions that are used to read and write data. This class does not allow data access directly.

Here, we are creating an example in which we have a class that encapsulates properties and provides getter and setter functions.

### Example

1. **namespace** AccessSpecifiers
2. {
3. **class** Student
4. {
5. // Creating setter and getter for each property
6. **public** **string** ID { **get**; **set**; }
7. **public** **string** Name { **get**; **set**; }
8. **public** **string** Email { **get**; **set**; }
9. }
10. }
11. **using** System;
12. **namespace** AccessSpecifiers
13. {
14. **class** Program
15. {
16. **static** **void** Main(**string**[] args)
17. {
18. Student student = **new** Student();
19. // Setting values to the properties
20. student.ID = "101";
21. student.Name = "Mohan Ram";
22. student.Email = "mohan@example.com";
23. // getting values
24. Console.WriteLine("ID = "+student.ID);
25. Console.WriteLine("Name = "+student.Name);
26. Console.WriteLine("Email = "+student.Email);
27. }
28. }
29. }

**Output:**

ID = 101

Name = Mohan Ram

Email = mohan@example.com

**Q20: What is Inheritance in C#?**

In C#, inheritance is a process in which one class acquires all the properties and behaviors of its parent class automatically. In such way, you can reuse, extend or modify the attributes and behaviors which is defined in other class.

In C#, the class which inherits the members of another class is called **derived class** and the class whose members are inherited is called **base** class. The derived class is the specialized class for the base class.

### Advantage of C# Inheritance

**Code reusability:** Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

## **C# Single Level Inheritance Example: Inheriting Fields**

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

1. **using** System;
2. **public** **class** Employee
3. {
4. **public** **float** salary = 40000;
5. }
6. **public** **class** Programmer: Employee
7. {
8. **public** **float** bonus = 10000;
9. }
10. **class** TestInheritance{
11. **public** **static** **void** Main(**string**[] args)
12. {
13. Programmer p1 = **new** Programmer();
15. Console.WriteLine("Salary: " + p1.salary);
16. Console.WriteLine("Bonus: " + p1.bonus);
18. }
19. }

Output:

Salary: 40000

Bonus: 10000

In the above example, Employee is the **base** class and Programmer is the **derived** class.

## **C# Single Level Inheritance Example: Inheriting Methods**

Let's see another example of inheritance in C# which inherits methods only.

1. **using** System;
2. **public** **class** Animal
3. {
4. **public** **void** eat() { Console.WriteLine("Eating..."); }
5. }
6. **public** **class** Dog: Animal
7. {
8. **public** **void** bark() { Console.WriteLine("Barking..."); }
9. }
10. **class** TestInheritance2{
11. **public** **static** **void** Main(**string**[] args)
12. {
13. Dog d1 = **new** Dog();
14. d1.eat();
15. d1.bark();
16. }
17. }

Output:

Eating...

Barking...

## **C# Multi Level Inheritance Example**

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C#. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C#.

1. **using** System;
2. **public** **class** Animal
3. {
4. **public** **void** eat() { Console.WriteLine("Eating..."); }
5. }
6. **public** **class** Dog: Animal
7. {
8. **public** **void** bark() { Console.WriteLine("Barking..."); }
9. }
10. **public** **class** BabyDog : Dog
11. {
12. **public** **void** weep() { Console.WriteLine("Weeping..."); }
13. }
14. **class** TestInheritance2{
15. **public** **static** **void** Main(**string**[] args)
16. {
17. BabyDog d1 = **new** BabyDog();
18. d1.eat();
19. d1.bark();
20. d1.weep();
21. }
22. }

Output:

Eating...

Barking...

Weeping...

#### **Note: Multiple inheritance is not supported in C# through class.**

**Q21: What is C# Aggregation?**

# **C# Aggregation (HAS-A Relationship)**

In C#, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

### C# Aggregation Example

Let's see an example of aggregation where Employee class has the reference of Address class as data member. In such way, it can reuse the members of Address class.

1. **using** System;
2. **public** **class** Address
3. {
4. **public** **string** addressLine, city, state;
5. **public** Address(**string** addressLine, **string** city, **string** state)
6. {
7. **this**.addressLine = addressLine;
8. **this**.city = city;
9. **this**.state = state;
10. }
11. }
12. **public** **class** Employee
13. {
14. **public** **int** id;
15. **public** **string** name;
16. **public** Address address;//Employee HAS-A Address
17. **public** Employee(**int** id, **string** name, Address address)
18. {
19. **this**.id = id;
20. **this**.name = name;
21. **this**.address = address;
22. }
23. **public** **void** display()
24. {
25. Console.WriteLine(id + " " + name + " " +
26. address.addressLine + " " + address.city + " " + address.state);
27. }
28. }
29. **public** **class** TestAggregation
30. {
31. **public** **static** **void** Main(**string**[] args)
32. {
33. Address a1=**new** Address("G-13, Sec-3","Noida","UP");
34. Employee e1 = **new** Employee(1,"Sonoo",a1);
35. e1.display();
36. }
37. }

Output:

1 Sonoo G-13 Sec-3 Noida UP

**Q22: What is four fundamental principal of OOP?**

**Four fundamental principles of OOP:**

* Encapsulation
* Inheritance
* Polymorphism
* Abstraction

**1. Encapsulation:** Encapsulation is the procedure of encapsulating data and functions into a single unit. The unit can be a class, a record, or a struct.

Encapsulation is the technique used to implement abstraction in object oriented programming by using Access modifier private, protected or public keywords.

**2. Inheritance:** In C#, inheritance is a process in which one class acquires all the properties and behaviors of its parent class automatically. In such way, you can reuse, extend or modify the attributes and behaviors which is defined in other class.

In C#, the class which inherits the members of another class is called **derived class** and the class whose members are inherited is called **base** class.

**3. Polymorphism:** Polymorphism is the concept where an object behaves differently in different situations. There are two types of polymorphism –

**Compile time polymorphism** is achieved by method overloading.

**Runtime polymorphism** is implemented when we have “IS-A” relationship between objects and is achieved by method overriding.

**4. Abstraction:** Abstraction in C# is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1. Abstract class
2. Interface

**Reusability:**

Once a class has been written, created, and debugged, it can be distributed to other programmers for use in their own programs. This is called reusability; in .NET terminology, this concept is called a component or a DLL. In OOP, however, inheritance provides an essential extension of the idea of reusability. A programmer can use an existing class and, add additional features to it, without modifying it.

## **Sealed Classes**

Sealed classes are the reverse of abstract classes. While abstract classes are inherited and refined in the derived class, sealed classes cannot be inherited. Instead, you can create an instance of a sealed class. A sealed class is used to prevent further refinement through inheritance.

## **Interface**

An interface is a set of related functions that must be implemented in a derived class. Members of an interface are implicitly public and abstract. Interfaces are similar to abstract classes. First, both types must be inherited; second, you cannot create an instance of either. Although there are several differences as in the following;

* An Abstract class can contain some implementations, but an interface can't.
* An Interface can only inherit other interfaces, but abstract classes can inherit from other classes and interfaces.
* An Abstract class can contain constructors and destructors, but an interface can't.
* An Abstract class contains fields, but interfaces don't.

So the question is, which of these to choose? Select interfaces because, with an interface, the derived type can still inherit from another type, and interfaces are more straightforward than abstract classes.

# **C# Destructor**

A destructor works opposite to constructor, it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

#### **Note: C# destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.**

### C# Constructor and Destructor Example

Let's see an example of constructor and destructor in C# which is called automatically.

1. **using** System;
2. **public** **class** Employee
3. {
4. **public** Employee()
5. {
6. Console.WriteLine("Constructor Invoked");
7. }
8. ~Employee()
9. {
10. Console.WriteLine("Destructor Invoked");
11. }
12. }
13. **class** TestEmployee{
14. **public** **static** **void** Main(**string**[] args)
15. {
16. Employee e1 = **new** Employee();
17. Employee e2 = **new** Employee();
18. }
19. }

Output:

Constructor Invoked

Constructor Invoked

Destructor Invoked

Destructor Invoked

#### **Note: Destructor can't be public. We can't apply any modifier on destructors.**

# **C# this**

In c# programming, this is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C#.

* It can be used **to refer current class instance variable**. It is used if field names (instance variables) and parameter names are same, that is why both can be distinguish easily.
* It can be used **to pass current object as a parameter to another method**.
* It can be used **to declare indexers**.

### C# this example

Let's see the example of this keyword in C# that refers to the fields of current class.

1. **using** System;
2. **public** **class** Employee
3. {
4. **public** **int** id;   // fields
5. **public** String name;
6. **public** **float** salary;
7. **public** Employee(**int** id, String name,**float** salary)
8. {
9. **this**.id = id;
10. **this**.name = name;
11. **this**.salary = salary;
12. }
13. **public** **void** display()
14. {
15. Console.WriteLine(id + " " + name+" "+salary);
16. }
17. }
18. **class** TestEmployee{
19. **public** **static** **void** Main(**string**[] args)
20. {
21. Employee e1 = **new** Employee(101, "Sonoo", 890000f);
22. Employee e2 = **new** Employee(102, "Mahesh", 490000f);
23. e1.display();
24. e2.display();
26. }
27. }

Output:

101 Sonoo 890000

102 Mahesh 490000

**Q23: Difference between string and string builder?**

**string** and **StringBuilder** are both classes in C# used to work with text, but they have different characteristics and are suitable for different scenarios. Here are the key differences between **string** and **StringBuilder**:

1. **Mutability:**
   * **String:** **string** objects in C# are immutable, meaning that once a **string** is created, it cannot be modified. Any operation that appears to modify a **string** actually creates a new **string** object, which can lead to performance overhead when performing multiple string manipulation operations.
   * **StringBuilder:** **StringBuilder** objects are mutable, allowing you to efficiently modify the contents of a string without creating new objects for each operation. This makes **StringBuilder** more suitable for scenarios where you need to perform multiple string concatenations or modifications.
2. **Performance:**
   * **string:** When you perform string concatenation or modification operations on a **string**, such as using the **+** operator or the **string.Concat** method, new string objects are created, which can be inefficient in terms of memory and performance, especially when dealing with large strings or many operations.
   * **StringBuilder:** **StringBuilder** is optimized for string manipulation operations. It uses a resizable buffer internally to store the character data, which means that appending or modifying strings with **StringBuilder** is more efficient and consumes less memory compared to using **string**.
3. **Memory Usage:**
   * **string:** Because **string** objects are immutable, when you modify a **string**, the old **string** objects are not immediately garbage collected. This can lead to increased memory usage as multiple **string** objects are created and retained in memory until they are no longer referenced.
   * **StringBuilder:** **StringBuilder** avoids the memory overhead associated with creating multiple string objects because it modifies its internal buffer directly. It helps reduce memory consumption when performing extensive string manipulation operations.
4. **Usage Scenarios:**
   * **string:** Use **string** when you have a fixed, unchanging text value or when you don't need to perform extensive string manipulations. **string** is suitable for representing text literals, constants, and read-only text.
   * **StringBuilder:** Use **StringBuilder** when you need to concatenate or modify strings in a loop, build complex strings incrementally, or perform multiple string manipulation operations. **StringBuilder** is well-suited for scenarios like building SQL queries, processing large text files, or constructing long strings dynamically.
5. **Thread Safety:**
   * **string:** **string** objects are inherently thread-safe because they are immutable. Multiple threads can read and use the same **string** without issues. However, if you modify a **string** concurrently, you may need to implement synchronization mechanisms.
   * **StringBuilder:** **StringBuilder** is not inherently thread-safe. If you need to modify a **StringBuilder** from multiple threads concurrently, you should use appropriate synchronization mechanisms (e.g., locks) to ensure thread safety.

In summary, **string** is suitable for representing fixed, unchanging text, while **StringBuilder** is designed for efficient string concatenation and modification operations, especially when dealing with dynamic or frequently changing strings. Choosing the right class for your scenario can significantly impact the performance and memory efficiency of your code.

**Q24: What is GAC?**

GAC stands for Global Assembly Cache, and it is a special repository in the Microsoft .NET Framework where strongly named assemblies are stored and shared globally across the system. The Global Assembly Cache is a central location for managing and storing .NET assemblies that are intended to be shared among multiple applications or across the entire machine.

The Global Assembly Cache plays a crucial role in ensuring that shared .NET assemblies are managed, versioned, and secured effectively, making it easier to maintain and update applications that depend on these assemblies.

**Q25: Difference between Entity Framework and ADO.Net?**

Entity Framework (EF) and ADO.NET are both technologies used in .NET for database access, but they differ significantly in their approach and functionality. Here are the key differences between Entity Framework and ADO.NET:

**1. Abstraction Level:**

**- Entity Framework:** EF provides a higher level of abstraction and is considered an Object-Relational Mapping (ORM) framework. It allows developers to work with databases using object-oriented concepts like classes, objects, and LINQ queries. EF abstracts away many low-level database operations.

**- ADO.NET:** ADO.NET is a lower-level data access technology that provides a more direct and fine-grained control over database interactions. It involves writing SQL queries and working with data in the form of datasets, data readers, and data tables.

**2. Data Model:**

**- Entity Framework:** EF uses a model-first or code-first approach, where developers define data models using C# classes or an Entity Data Model (EDM). EF generates the database schema based on these models or vice versa.

**- ADO.NET:** ADO.NET typically does not provide an abstraction for data models. Developers work with raw data and SQL, and they are responsible for defining the database schema and mapping data to objects themselves.

**3. Querying:**

**- Entity Framework:** EF allows developers to write queries using LINQ (Language Integrated Query) or query builder methods, making queries more type-safe and readable. Developers don't need to write raw SQL queries in most cases.

**- ADO.NET:** ADO.NET involves writing SQL queries manually. Developers use SqlCommand or other SQL-related classes to execute queries and retrieve data.

**4. Ease of Use:**

**- Entity Framework:** EF is known for its ease of use and developer-friendly features. It reduces the amount of boilerplate code required for common database operations and simplifies data access code.

**- ADO.NET:** ADO.NET can be more verbose and requires developers to handle many aspects of database interaction manually, which can lead to more code and potential error-prone code.

**5. Automatic Change Tracking:**

**- Entity Framework:** EF includes built-in change tracking, which automatically detects changes made to objects in memory and generates SQL statements to persist those changes to the database when needed.

**- ADO.NET:** ADO.NET does not provide automatic change tracking, and developers are responsible for tracking changes and writing appropriate update statements.

**6. Support for Different Database Providers:**

**- Entity Framework:** EF is designed to work with multiple database providers, allowing developers to switch databases relatively easily by changing the database connection string.

**- ADO.NET:** ADO.NET can also work with different database providers, but developers may need to write more provider-specific code.

**7. Development Speed:**

**- Entity Framework:** EF can accelerate development by providing high-level abstractions and reducing the need for boilerplate code.

**- ADO.NET:** ADO.NET offers more control but may require more development time due to lower-level operations.

**8. Learning Curve:**

**- Entity Framework:** EF has a shorter learning curve for developers familiar with object-oriented programming and LINQ.

**- ADO.NET:** ADO.NET may have a steeper learning curve, especially for developers who are new to database programming.

In summary, Entity Framework is a higher-level, ORM-based approach that simplifies database access and is suitable for most applications. ADO.NET offers more control but requires more manual coding and is often used when fine-grained control over database interactions is necessary or when working with legacy codebases. The choice between EF and ADO.NET depends on the specific requirements of the project and the developer's familiarity with the technologies.

**Q26: what is Entity Framework and why we use it?**

Entity Framework (EF) is an object-relational mapping (ORM) framework developed by Microsoft for the .NET platform. It provides a set of tools and libraries for developers to work with relational databases using an object-oriented approach. The primary goal of Entity Framework is to simplify and streamline database interactions in .NET applications, making it easier to develop database-driven applications.

Here are the key aspects and reasons why Entity Framework is used:

1. **Object-Relational Mapping (ORM):** Entity Framework bridges the gap between the object-oriented programming (OOP) world and relational databases. It allows developers to work with database data as if they were working with regular .NET objects, abstracting away the complexities of SQL and database schema management.
2. **Abstraction of Database Operations:** EF abstracts common database operations (such as querying, inserting, updating, and deleting records) into a set of object-oriented methods and LINQ (Language Integrated Query) queries. Developers can express these operations in C# or VB.NET code without writing raw SQL queries.
3. **Database Independence:** Entity Framework supports multiple database providers, including SQL Server, MySQL, PostgreSQL, SQLite, and Oracle. Developers can write database-agnostic code, and EF takes care of generating the appropriate SQL for the chosen database system.
4. **Code-First and Database-First Approaches:** EF supports both Code-First and Database-First approaches. In the Code-First approach, developers define the data model using C# classes, and EF generates the database schema based on those classes. In the Database-First approach, EF generates C# classes and entities based on an existing database schema.
5. **Automatic Change Tracking:** Entity Framework includes a change tracking mechanism that monitors changes made to objects in memory. It can automatically generate SQL statements to persist those changes to the database when required.
6. **Relationships and Navigation Properties:** EF supports defining and managing relationships between entities (tables) in the database. Navigation properties allow developers to easily traverse and manipulate related data.
7. **LINQ Integration:** Entity Framework seamlessly integrates with LINQ, allowing developers to write strongly typed queries using a familiar syntax. This makes it easy to compose complex queries in a type-safe manner.
8. **Optimistic Concurrency Control:** EF provides mechanisms for handling concurrent updates to the database. It can detect conflicts when multiple users attempt to modify the same data simultaneously and apply appropriate strategies for resolving conflicts.
9. **Security and Validation:** Entity Framework includes features for managing security and validating data before it's persisted to the database. You can implement custom validation logic within your entities.
10. **Migrations:** EF supports database schema migrations, allowing developers to evolve the database schema as the application evolves. It helps manage changes to the database structure over time without data loss.
11. **Performance Optimization:** While EF simplifies development, it also provides ways to optimize performance, including techniques for eager loading, lazy loading, and query optimization.

Entity Framework is commonly used in a wide range of .NET applications, including web applications (ASP.NET), desktop applications (WPF and Windows Forms), and backend services. It offers a higher level of abstraction for database operations, reducing the amount of boilerplate code required to interact with the database and enhancing productivity. However, it's essential to have a good understanding of database design principles, SQL, and EF's features to use it effectively in application development.

**Q27: what is ORM?**

ORM stands for Object-Relational Mapping. It is a programming technique used in software development to bridge the gap between the object-oriented programming (OOP) paradigm and relational databases. The primary goal of ORM is to simplify the process of storing, retrieving, and manipulating data in a database using object-oriented languages like Java, C#, or Python.

In a traditional application, data is stored in relational databases, which use tables with rows and columns to organize and represent data. On the other hand, object-oriented programming deals with objects, classes, and their relationships. ORM serves as an intermediary layer that allows developers to work with database records as if they were ordinary objects, abstracting away the complexities of SQL queries and database-specific operations.

Here are some key aspects of ORM:

1. **Object-Relational Mapping:** ORM maps database tables to classes and their rows to objects. Each table in the database corresponds to a class, and each row in the table corresponds to an object instance of that class.
2. **Abstraction:** ORM abstracts the underlying database operations, including CRUD (Create, Read, Update, Delete) operations, into object-oriented methods and functions. This abstraction simplifies the code and makes it more readable and maintainable.
3. **Cross-Platform Compatibility:** ORM frameworks are designed to work with multiple database management systems (DBMS) such as MySQL, PostgreSQL, SQL Server, and SQLite. This means you can write database-agnostic code, and the ORM framework handles the database-specific details.
4. **Automatic SQL Generation:** ORM frameworks generate SQL queries and statements on behalf of developers. Developers interact with the ORM API, and the ORM framework translates these high-level operations into SQL queries that are executed against the database.
5. **Object Relationships:** ORM allows developers to define and manage relationships between objects and classes, such as one-to-one, one-to-many, and many-to-many relationships. These relationships are often represented as attributes or properties in object classes.
6. **Caching and Optimization:** ORM frameworks often provide caching mechanisms to improve performance. They can cache frequently accessed data to reduce database queries and optimize data retrieval.

Popular ORM frameworks and libraries in various programming languages include:

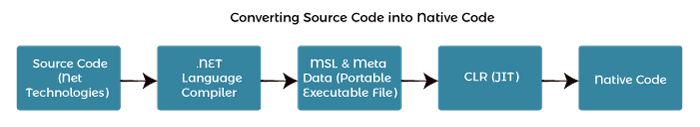
* **Entity Framework (EF):** A popular ORM framework for .NET applications, including ASP.NET and .NET Core.
* **Hibernate:** A widely used ORM framework for Java applications.
* **Django ORM:** Built into the Django web framework for Python, it provides ORM capabilities for Python developers.
* **SQLAlchemy:** A popular ORM library for Python that supports multiple database backends.
* **Ruby on Rails' ActiveRecord:** Part of the Ruby on Rails web framework, it offers ORM features for Ruby developers.

ORM simplifies database interaction in application development and reduces the need to write custom SQL queries, making it easier to work with databases in an object-oriented manner. However, it's essential to understand the underlying database and SQL concepts, as well as the performance implications of ORM operations, to use ORM effectively in software projects.

**Q28: what is CLR?**

.NET CLR is a runtime environment that manages and executes the code written in any .NET programming language. CLR is the virtual machine component of the .NET framework. The language's compiler compiles the source code of applications developed using .NET compliant languages into CLR's intermediate language called MSIL, i.e., Microsoft intermediate language code. This code is platform-independent. It is comparable to byte code in java. Metadata is also generated during compilation and MSIL code and stored in a file known as the Manifest file. This metadata is generally about members and types required by CLR to execute MSIL code. A just-in-time compiler component of CLR converts MSIL code into native code of the machine. This code is platform-dependent. CLR manages memory, threads, exceptions, code execution, code safety, verification, and compilation.

**The following figure shows the conversion of source code into native code.**



### The main components of CLR are:

* Common type system
* Common language speciation
* Garbage Collector
* Just in Time Compiler
* Metadata and Assemblies

### 1. Common type system:

CTS provides guidelines for declaring, using, and managing data types at runtime. It offers cross-language communication. For example, VB.NET has an integer data type, and C# has an int data type for managing integers. After compilation, Int32 is used by both data types. So, CTS provides the data types using managed code. A common type system helps in writing language-independent code.

### 2. Common Language Specification (CLS):

Common Language Specification (CLS) contains a set of rules to be followed by all NET-supported languages. The common rules make it easy to implement language integration and help in cross-language inheritance and debugging. Each language supported by NET Framework has its own syntax rules. But CLS ensures interoperability among applications developed using NET languages.

### 3. Garbage Collection:

Garbage Collector is a component of CLR that works as an automatic memory manager. It helps manage memory by automatically allocating memory according to the requirement. It allocates heap memory to objects. When objects are not in use, it reclaims the memory allocated to them for future use. It also ensures the safety of objects by not allowing one object to use the content of another object.

### 4. Just in Time (JIT) Compiler:

JIT Compiler is an important component of CLR. It converts the MSIL code into native code (i.e., machine-specific code). The .NET program is compiled either explicitly or implicitly. The developer or programmer calls a particular compiler to compile the program in the explicit compilation. In implicit compilation, the program is compiled twice. The source code is compiled into Microsoft Intermediate Language (MSIL) during the first compilation process. The MSIL code is converted into native code in the second compilation process. This process is called JIT compilation. There are three types of JIT compilers -Pre, Econo, and Normal. Pre JIT Compiler compiles entire MSIL code into native code before execution. Econo JIT Compiler compiles only those parts of MSIL code required during execution and removes those parts that are not required anymore. Normal JIT Compiler also compiles only those parts of MSIL code required during execution but places them in cache for future use. It does not require recompilations of already used parts as they have been placed in cache memory.

### 5. Metadata:

A Metadata is a binary information about the program, either stored in a CLR Portable Executable file (PE) along with MSIL code or in the memory. During the execution of MSIL, metadata is also loaded into memory for proper interpretation of classes and related. Information used in code. So, metadata helps implement code in a language-neutral manner or achieve language interoperability.

### 6. Assemblies:

An assembly is a fundamental unit of physical code grouping. It consists of the assembly manifest, metadata, MSIL code, and a set of resources like image files. It is also considered a basic deployment unit, version control, reuse, security permissions, etc.

## **.NET CLR Functions**

Following are the functions of the CLR.

* It converts the program into native code.
* Handles Exceptions
* Provides type-safety
* Memory management
* Provides security
* Improved performance
* Language independent
* Platform independent
* Garbage collection
* Provides language features such as inheritance, interfaces, and overloading for object-oriented programs.

## **WinForms**

Windows Forms is a smart client technology for the .NET Framework, a set of managed libraries that simplify common application tasks such as reading and writing to the file system.

## **ASP.NET**

ASP.NET is a web framework designed and developed by Microsoft. It is used to develop websites, web applications, and web services. It provides a fantastic integration of HTML, CSS, and JavaScript. It was first released in January 2002.

## **ADO.NET**

ADO.NET is a module of .Net Framework, which is used to establish a connection between application and data sources. Data sources can be such as SQL Server and XML. ADO .NET consists of classes that can be used to connect, retrieve, insert, and delete data.

## **WPF (Windows Presentation Foundation)**

Windows Presentation Foundation (WPF) is a graphical subsystem by Microsoft for rendering user interfaces in Windows-based applications. WPF, previously known as "Avalon", was initially released as part of .NET Framework 3.0 in 2006. WPF uses DirectX.

## **WCF (Windows Communication Foundation)**

It is a framework for building service-oriented applications. Using WCF, you can send data as asynchronous messages from one service endpoint to another.

## **LINQ (Language Integrated Query)**

It is a query language, introduced in .NET 3.5 framework. It is used to make the query for data sources with C# or Visual Basics programming languages.

## **Entity Framework**

It is an ORM based open source framework which is used to work with a database using .NET objects. It eliminates a lot of developers effort to handle the database. It is Microsoft's recommended technology to deal with the database.

## **29. What is Boxing and Unboxing in C#?**

Boxing and Unboxing are both used for type conversions.

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

// Boxing

int i = 100;

Object obj = i; // Boxing Process

i = 200;

Console.WriteLine(i); // Output -- 200

Console.WriteLine(obj); // Output -- 100

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

// Unboxing

Object obj2 = 123;

int anum2 = (int)obj; // Unboxing Process

obj2 = 150;

Console.WriteLine(anum2); // Output -- 123

Console.WriteLine(obj2); // Output -- 150

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

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

**Struct**

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

**Class**

* The class is a reference type in C#, and it inherits from the System.Object Type.
* Classes are usually used for large amounts of data.
* Classes can be inherited from other classes.
* A class can be an abstract type.
* We can create a default constructor.

## **31. What is an enum in C#?**

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

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

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

The underlying default type of the enumeration element is int. By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1.

enum Day {Sat, Sun, Mon, Tue, Wed, Thu, Fri};

Some points about enum,

* Enums are enumerated data types in c#.
* Enums are not for the end-user. They are meant for developers.
* Enums are strongly typed constant. They are strongly typed, i.e., an enum of one type may not be implicitly assigned to an enum of another type even though the underlying value of their members is the same.
* Enumerations (enums) make your code much more readable and understandable.
* Enum values are fixed. Enum can be displayed as a string and processed as an integer.
* The default type is int, and the approved types are byte, sbyte, short, ushort, uint, long, and ulong.
* Every enum type automatically derives from System.Enum, and thus, we can use System.Enum methods on enums.
* Enums are value types created on the stack, not the heap.

**Key Differences Between For Loop and While Loop**

# **C# For Loop**

The C# for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops.

**Syntax:**

1. **for**(initialization; condition; incr/decr){
2. //code to be executed
3. }

## **C# Nested For Loop**

In C#, we can use for loop inside another for loop, it is known as nested for loop. The inner loop is executed fully when outer loop is executed one time. So if outer loop and inner loop are executed 3 times, inner loop will be executed 3 times for each outer loop i.e. total 9 times.

# **C# While Loop**

In C#, while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop than for loop.

**Syntax:**

1. **while**(condition){
2. //code to be executed
3. }

### C# While Loop Example

using System;

**public** **class** WhileExample

    {

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

      {

**int** i=1;

**while**(i<=10)

          {

              Console.WriteLine(i);

              i++;

          }

     }

   }

# **C# Do-While Loop**

The C# do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The C# do-while loop is executed at least once because condition is checked after loop body.

**Syntax:**

1. **do**{
2. //code to be executed
3. }**while**(condition);

### C# do-while Loop Example

Let's see a simple example of C# do-while loop to print the table of 1.

1. **using** System;
2. **public** **class** DoWhileExample
3. {
4. **public** **static** **void** Main(**string**[] args)
5. {
6. **int** i = 1;
8. **do**{
9. Console.WriteLine(i);
10. i++;
11. } **while** (i <= 10) ;
13. }
14. }

## **Difference between Convert.ToString and ToString Method in C#**

Both these methods are used to convert a value to a string. The difference is **Convert.ToString()** method handles null whereas the **ToString()** doesn’t handle null in C#.

In C# if you declare a **string variable** and if you **don’t assign any value** to that variable, then by default that variable takes a **null** value. In such a case, if you use the **ToString()**method then your program will throw the **Null Reference Exception.** On the other hand, if you use the **Convert.ToString()** method then your program will not throw an exception.

**What is the Difference Between int and double?**

The **main difference** between int and double is that int**is used to store 32 bit two’s complement integer while double is used to store 64 bit double precision floating point value.**

The int and double are two main primitive data types. Usually, int allocates 4 bytes for data whereas double allocates 8 bytes for data.

1 byte = 8 bit

# **C# static class**

The C# static class is like the normal class but it cannot be instantiated. It can have only static members and static methods. The advantage of static class is that it provides you guarantee that instance of static class cannot be created. Static classes are **[sealed](https://www.geeksforgeeks.org/c-sealed-class/)**, means ***you cannot inherit a static class from another class***.

Points to remember for C# static class

* C# static class contains only static members.
* C# static class cannot be instantiated.
* C# static class is sealed.
* C# static class cannot contain instance constructors.

**Example:**

**using** System;

**public** **static** **class** MyMath

    {

**public** **static** **float** PI=3.14f;

**public** **static** **int** cube(**int** n){**return** n\*n\*n;}

    }

**class** TestMyMath{

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

        {

            Console.WriteLine("Value of PI is: "+MyMath.PI);

            Console.WriteLine("Cube of 3 is: " + MyMath.cube(3));

        }

    }

#### **Difference between static and non-static class**

| **Static Class** | **Non-Static Class** |
| --- | --- |
| Static class is defined using static keyword. | Non-Static class is not defined by using static keyword. |
| In static class, you are not allowed to create objects. | In non-static class, you are allowed to create objects using new keyword. |
| The data members of static class can be directly accessed by its class name. | The data members of non-static class is not directly accessed by its class name. |
| Static class always contains static members. | Non-static class may contain both static and non-static methods. |
| Static class does not contain an instance constructor. | Non-static class contains an instance constructor. |
| Static class cannot inherit from another class. | Non-static class can be inherited from another class. |

# **C# static constructor**

C# static constructor is used to initialize static fields. It can also be used to perform any action that is to be performed only once. It is invoked automatically before first instance is created or any static member is referenced.

### Points to remember for C# Static Constructor

* C# static constructor cannot have any modifier or parameter.
* C# static constructor is invoked implicitly. It can't be called explicitly.

## **C# Static Constructor example**

**using** System;

**public** **class** Account

    {

**public** **int** id;

**public** String name;

**public** **static** **float** rateOfInterest;

**public** Account(**int** id, String name)

        {

**this**.id = id;

**this**.name = name;

        }

**static** Account()

        {

            rateOfInterest = 9.5f;

        }

**public** **void** display()

        {

            Console.WriteLine(id + " " + name+" "+rateOfInterest);

        }

   }

**class** TestEmployee{

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

        {

            Account a1 = **new** Account(101, "Sonoo");

            Account a2 = **new** Account(102, "Mahesh");

            a1.display();

            a2.display();

        }

    }

Output:

101 Sonoo 9.5

102 Mahesh 9.5

# **Difference between readonly and const keyword in C#**

In C#, a ***const***keyword is used to declare constant fields and constant local. The value of the constant field is the same throughout the program or in other words, once the constant field is assigned the value of this field is not be changed. In C#, constant fields and locals are not variables, a constant is a number, string, null reference, boolean values.

In C#, you can use a ***readonly***keyword to declare a readonly variable. This readonly keyword shows that you can assign the variable only when you declare a variable or in a constructor of the same class in which it is declared.

**Example:**

// C# program to illustrate the use

// of the readonly keyword

using System;

class GFG {

// readonly variables

public readonly int myvar1;

public readonly int myvar2;

// Values of the readonly

// variables are assigned

// Using constructor

public GFG(int b, int c)

{

myvar1 = b;

myvar2 = c;

Console.WriteLine("Display value of myvar1 {0}, "+

"and myvar2 {1}", myvar1, myvar2);

}

// Main method

static public void Main()

{

GFG obj1 = new GFG(100, 200);

}

}

|  |  |
| --- | --- |
| **ReadOnly Keyword** | **Const Keyword** |
| In C#, readonly fields can be created using readonly keyword | In C#, constant fields are created using const keyword. |
| ReadOnly is a runtime constant. | Const is a compile time constant. |
| The value of readonly field can be changed. | The value of the const field cannot be changed. |
| It cannot be declared inside the method. | It can be declared inside the method. |
| In readonly fields, we can assign values in declaration and in the constructor part. | In const fields, we can only assign values in declaration part. |

**Solid Principles**

<https://www.educative.io/blog/solid-principles-oop-c-sharp?utm_campaign=brand_educative&utm_source=google&utm_medium=ppc&utm_content=performance_max_india&eid=5082902844932096&utm_term=&utm_campaign=%5BNew%5D+Performance+Max&utm_source=adwords&utm_medium=ppc&hsa_acc=5451446008&hsa_cam=18931439518&hsa_grp=&hsa_ad=&hsa_src=x&hsa_tgt=&hsa_kw=&hsa_mt=&hsa_net=adwords&hsa_ver=3&gad_source=1&gclid=CjwKCAiA0PuuBhBsEiwAS7fsNXjwFEODkGJInx9cAia2Bc-aFXnQFtcNIIyd3t4jTHPnULhgTqPhzRoCT2QQAvD_BwE>

The SOLID principles are a set of golden rules that aim to improve the design and maintainability of software. These principles were first introduced in the early 2000s and have since become widely accepted as best practices for developers working with **object-oriented programming languages.** SOLID principles are particularly relevant for **agile development,** as they help create flexible, scalable, and easy to modify code.

There are several advantages to following the SOLID principles when designing and building software, including:

* **Improved maintainability:** You can create code that is easier to maintain and modify over time because the SOLID principles encourage the creation of modular, flexible code that is less prone to errors and more resistant to changes in requirements.
* **Reduced complexity:** The SOLID principles help to reduce the complexity of software by promoting the use of abstraction and encapsulation, which can make it easier to understand and work with the code.
* **Enhanced flexibility:** These principles encourage the creation of flexible code that is open to extension but closed to modification, which encourages flexibility without breaking existing functionality.
* **Increased scalability:** The SOLID principles can help to [make software more scalable](https://www.educative.io/blog/scalable-systems-101" \t "_blank), as they encourage the use of abstractions and decoupled dependencies, which can help to prevent codebase from becoming overly complex and difficult to manage.

SOLID is a **mnemonic device for 5 design principles** of [object-oriented programs](https://www.educative.io/blog/object-oriented-programming" \t "_blank) (OOP) that result in readable, adaptable, and scalable code. SOLID can be applied to any OOP program.

The 5 principles of SOLID are:

* **S**ingle-responsibility principle
* **O**pen-closed principle
* **L**iskov substitution principle
* **I**nterface segregation principle
* **D**ependency inversion principle

SOLID principles were developed by computer science instructor and author Robert C. Martin (sometimes called “Uncle Bob”) in 2000 and quickly became a staple of [modern object-oriented design (OOD)](https://www.educative.io/blog/interview-guide-low-level-object-oriented-advanced-system-design" \t "_blank). The SOLID acronym became commonplace when these principles gained widespread popularity in the programming world.

Now, SOLID has also been adopted in both [agile development](https://www.educative.io/blog/java-agile-development" \t "_blank) and adaptive software development.

## **S:** Single-responsibility principle

### “A class should only have a single responsibility, that is, only changes to one part of the software’s specification should be able to affect the specification of the class.” -Robert C. Martin

The **single-responsibility principle** (SRP) states that each class, module, or function in your program should only do one job. In other words, each should have full responsibility for a single functionality of the program. The class should contain only variables and methods relevant to its functionality.

## ****O**:** Open-closed principle

### “Software entities … should be open for extension, but closed for modification.” Robert C. Martin

OCP implementations often rely on polymorphism and abstraction to code behaviour at a class level rather than hard-coding for certain situations.

## **L:** Liskov substitution principle

### "Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." Robert c. Martin

## **I:** Interface segregation principle

### "Many client-specific interfaces are better than one general-purpose interface." Robert C. Martin

The **interface segregation principle** (ISP) requires that classes only be able to perform behaviors that are useful to achieve its end functionality. In other words, classes do not include behaviors they do not use.

## ****D**:** Dependency inversion principle

### “One should depend upon abstractions, [not] concretions.” Robert C. Martin

The **dependency inversion principle** (DIP) has two parts:

1. High-level modules (Parent Class) should not depend on low-level modules (Child Class). Instead, both should depend on abstractions (interfaces)
2. Abstractions should not depend on details. Details (like concrete implementations) should depend on abstractions.

The first part of this principle **reverses traditional OOP software design**. Without DIP, programmers often construct programs to have high-level (less detail, more abstract) components explicitly connected with low-level (specific) components to complete tasks.

DIP decouples high and low-level components and instead connects both to abstractions. High and low-level components can still benefit from each other, but a change in one should not directly break the other.

## **What is an extension method in C#?**

In C#, an extension method is a static method used to extend the functionality of an existing type without modifying the original type or creating a new derived type. Extension methods allow developers to add methods to existing types, such as classes, structs, interfaces, enums, etc., not originally defined in those types.

Extension methods are declared in a static class and are defined as static methods with a special first parameter called the "this" parameter. The "this" parameter specifies the type being extended and allows the extension method to be called as if it were an instance method of that type.

For example, consider the following extension method that extends the string type by providing a method to capitalize the first letter of the string:

public static class StringExtensions

{

public static string CapitalizeFirstLetter(this string str)

{

if (string.IsNullOrEmpty(str))

return str;

return char.ToUpper(str[0]) + str.Substring(1);

}

}

With this extension method, the CapitalizeFirstLetter method can be called on any string object like this:

string s = "hello world";

string capitalized = s.CapitalizeFirstLetter(); // "Hello world"

## **What is the difference between Dispose and Finalize in C#?**

In C#, both the Dispose and Finalize methods are used to release resources, but they serve different purposes and behaviors.

The Dispose method releases unmanaged resources, such as file handles or database connections, not automatically managed by the .NET runtime. It is typically implemented in a class that implements the IDisposable interface, which defines the Dispose method.

The Dispose method is called explicitly by client code to release resources when they are no longer needed. It can be called implicitly using the statement, which ensures that the Dispose method is called when the object goes out of scope.

On the other hand, the Finalize method is used to perform cleanup operations on an object just before it is garbage collected. Therefore, it is typically implemented in a class that overrides the Object.Finalize method.

The garbage collector calls the Finalize method, which automatically manages the memory of .NET objects, to release unmanaged resources that have not been explicitly released by the Dispose method.

The main difference between the two methods is that the Dispose method is deterministic and can be explicitly called by client code. In contrast, the Finalize method is non-deterministic and is called by the garbage collector at an undetermined time.

It is important to note that objects that implement the Dispose method should also implement the Finalize method as a backup mechanism in case the client code does not call the Dispose method.

In summary, the Dispose method is used to release unmanaged resources deterministically. In contrast, the Finalize method is used as a backup mechanism to release unmanaged resources when the object is garbage collected.

## **What are the differences between IEnumerable and IQueryable?**

