### **The four basic principles of object-oriented programming are:**

### **Abstraction: -** Modeling the relevant attributes and interactions of entities as classes to define an abstract representation of a system.

### **Encapsulation: -** Hiding the internal state and functionality of an object and only allowing access through a public set of functions.

### **Inheritance: -** Ability to create new abstractions based on existing abstractions.

### **Polymorphism: -** Ability to implement inherited properties or methods in different ways across multiple abstractions.

**Encapsulation**

Encapsulation is sometimes referred to as the first pillar or principle of object-oriented programming. A class or struct can specify how accessible each of its members is to code outside of the class or struct. Methods and variables that aren't intended to be used from outside of the class or assembly can be hidden to limit the potential for coding errors or malicious exploits.

The members of a type include all methods, fields, constants, properties, and events. In C#, there are no global variables or methods as there are in some other languages. Even a program's entry point, the Main method, must be declared within a class or struct (implicitly when you use top-level statements).

The following list includes all the various kinds of members that may be declared in a class, struct, or record.

Fields

Constants

Properties

Methods

Constructors

Events

Finalizers

Indexers

Operators

Nested Types

Accessibility

Some methods and properties are meant to be called or accessed from code outside a class or struct, known as client code. Other methods and properties might be only for use in the class or struct itself. It's important to limit the accessibility of your code so that only the intended client code can reach it. You specify how accessible your types and their members are to client code by using the following access modifiers:

**Public:** Public access is the most permissive access level. There are no restrictions on accessing public members.

**Protected:** A protected member is accessible within its class and by derived class instances.

**Internal:** Internal types or members are accessible only within files in the same assembly.

**Protected internal:** A protected internal member of a base class is accessible from any type within its containing assembly. It is also accessible in a derived class located in another assembly only if the access occurs through a variable of the derived class type.

**Private:** Private access is the least permissive access level. Private members are accessible only within the body of the class or the struct in which they are declared

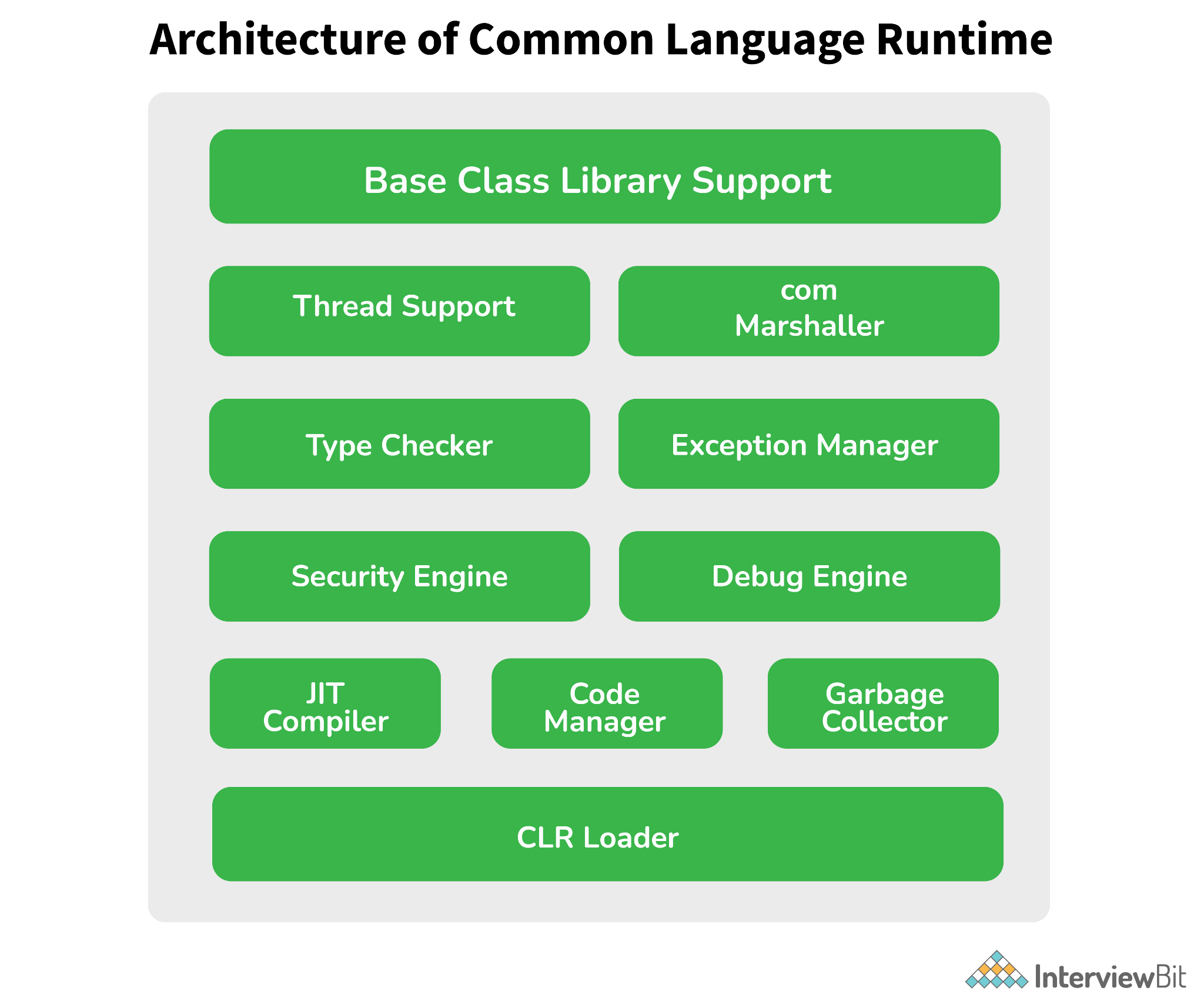
**Private protected:** A private protected member of a base class is accessible from derived types in its containing assembly only if the static type of the variable is the derived class type.

### 

### **1. How is C# different from C?**

C being the procedural language while C# is a more object-oriented language. The biggest difference is that C# supports automatic garbage collection by Common Language Runtime (CLR) while C does not. C# primarily needs a .NET framework to execute while C is a platform-agnostic language.

### **2. What is Common Language Runtime (CLR)?**

CLR handles program execution for various languages including C#. The architecture of CLR handles memory management, garbage collection, security handling, and looks like:

### **3. What is garbage collection in C#?**

.NET's garbage collector manages the allocation and release of memory for your application. Each time you create a new object, the common language runtime allocates memory for the object from the managed heap. As long as address space is available in the managed heap, the runtime continues to allocate space for new objects. However, memory is not infinite. Eventually the garbage collector must perform a collection in order to free some memory. The garbage collector's optimizing engine determines the best time to perform a collection, based upon the allocations being made. When the garbage collector performs a collection, it checks for objects in the managed heap that are no longer being used by the application and performs the necessary operations to reclaim their memory.

Conditions for a garbage collection

Garbage collection occurs when one of the following conditions is true:

* The system has low physical memory. This is detected by either the low memory notification from the OS or low memory as indicated by the host.
* The memory that's used by allocated objects on the managed heap surpasses an acceptable threshold. This threshold is continuously adjusted as the process runs.
* The GC.Collect method is called. In almost all cases, you don't have to call this method, because the garbage collector runs continuously. This method is primarily used for unique situations and testing.

### **4. What are the types of classes in C#?**

Class is an entity that encapsulates all the properties of its objects and instances as a single unit. C# has four types of such classes:

* Static class: Static class, defined by the keyword ‘static’ does not allow inheritance. Therefore, you cannot create an object for a static class.
* Partial class: Partial class, defined by the keyword ‘partial’ allows its members to partially divide or share source (.cs) files.
* Abstract class: Abstract classes are classes that cannot be instantiated where you cannot create objects. Abstract classes work on the OOPS concept of abstraction. Abstraction helps to extract essential details and hide the unessential ones.
* Sealed class: Sealed classes are classes that cannot be inherited. Use the keyword sealed to restrict access to users to inherit that class.

### 

### **5. What is a managed and unmanaged code?**

Managed code lets you run the code on a managed CLR runtime environment in the .NET framework.Managed code runs on the managed runtime environment than the operating system itself.

Benefits: Provides various services like a garbage collector, exception handling, etc.

Unmanaged code is when the code doesn’t run on CLR, it is an unmanaged code that works outside the .NET framework.

They don’t provide services of the high-level languages and therefore, run without them. Such an example is C++.

### **6. What is the difference between an abstract class and an interface?**

## What is Abstract Class?

The special class which cannot be instantiated is known as an abstract class. These types of classes are either somewhat executed or not executed at all, and they can also be inherited by subclasses.

## What is Interface?

The interface enables us to determine the functionality or functions but cannot implement that. It can also contain properties, methods, events, and indexers just like the class. However, the execution part will depend on the derived class as the interface includes only the declaration of the members.

Difference between Abstract Class and Interface in C#.

1. The special class which cannot be instantiated is known as abstract class.

The interface enables us to determine the functionality or functions but cannot implement that.

2. Abstract classes have static members.

Interface does not have static members.

3. They have a constructor.

They don’t have a constructor.

4. It includes both a declaration and an explanation.

It includes only a declaration.

5. Here the performance is fast.

Here the performance is slow.

7. A class has the liberty to only utilize a single abstract class.

Here a class has the liberty to utilize multiple interfaces.

8. It is utilized to execute the core identity of class.

It is utilized to execute the peripheral skills of the class.

9. It includes methods, fields, constants, etc.

It only includes methods .

10. Abstract class can be fully, partially or not implemented.

Interfaces can be fully implemented.

### **7. What are the differences between ref and out keywords?**

Both of these are keywords used in the C# language that assists in passing any arguments. There is a crucial difference between Ref and Out keywords in C#. We use the ref keyword for passing data in a bi-directional manner, while the out keyword helps users get data in a unidirectional manner.

## What is Ref Keyword in C#?

It is a keyword that we use in the C# language for passing the available arguments by their references. In simpler words, if we make any changes in the given argument, then this method reflects these changes in the variable whenever the control returns to the calling method. The parameter of ref does not at all pass the *property*.

## What is Out Keyword in C#?

It is a keyword that we use in the C# language to pass the available arguments to the methods as a type of reference. We generally use this keyword when any method returns various different values. The parameter of out does not at all pass the *property*.

## Difference Between Ref and Out Keywords in C#

Here is a list of the differences between Ref and Out Keywords in C#.

|  |  |  |
| --- | --- | --- |
| **Parameters** | **Ref Keyword in C#** | **Out Keyword in C#** |
| Purpose of Keyword | We use the ref keyword when a called parameter needs to update the parameter (passed). | We use the out keyword when a called method needs to update multiple parameters (passed). |
| Direction | We use this keyword for passing data in a bi-directional manner. | We use this keyword for getting data in a unidirectional manner. |
| Need to Initialize | We need to initialize a variable before passing it as a ref. The compiler will throw an error if we don’t do this step. | We don’t need to initialize any given variable in case we use an out keyword. |
| Initialization | In this case, we need to initialize any passed parameter in the form of ref. | In this case, we need to initialize any passed parameter in the form of out. |

### **8. What are extension methods in C#?**

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are static methods, but they're called as if they were instance methods on the extended type. For client code written in C#, F# and Visual Basic, there's no apparent difference between calling an extension method and the methods defined in a type.

The most common extension methods are the LINQ standard query operators that add query functionality to the existing System.Collections.IEnumerable and System.Collections.Generic.IEnumerable<T> types. To use the standard query operators, first bring them into scope with a using System.Linq directive. Then any type that implements IEnumerable<T> appears to have instance methods such as GroupBy, OrderBy, Average, and so on. You can see these additional methods in IntelliSense statement completion when you type "dot" after an instance of an IEnumerable<T> type such as List<T> or Array.

Extension methods are defined as static methods but are called by using instance method syntax. Their first parameter specifies which type the method operates on. The parameter is preceded by the this modifier. Extension methods are only in scope when you explicitly import the namespace into your source code with a using directive.

**namespace ExtensionMethods**

**{**

**public static class MyExtensions**

**{**

**public static int WordCount(this String str)**

**{**

**return str.Split(new char[] { ' ', '.', '?' },**

**StringSplitOptions.RemoveEmptyEntries).Length;**

**}**

**}**

**}**

The WordCount extension method can be brought into scope with this using directive:

**using ExtensionMethods;**

**namespace Program**

**{**

**Public static void()**

**{**

**string s = "Hello Extension Methods";**

**int i = s.WordCount();**

**}**

**}**

**9. What are Generics in C#?**

Generics introduces the concept of type parameters to .NET, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. by using a generic type parameter T, you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations.

**// Declare the generic class.**

**public class GenericList<T>**

**{**

**public void Add(T input) { }**

**}**

**class TestGenericList**

**{**

**private class ExampleClass { }**

**static void Main()**

**{**

**// Declare a list of type int.**

**GenericList<int> list1 = new GenericList<int>();**

**list1.Add(1);**

**// Declare a list of type string.**

**GenericList<string> list2 = new GenericList<string>();**

**list2.Add("");**

**// Declare a list of type ExampleClass.**

**GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();**

**list3.Add(new ExampleClass());**

**}**

**}**

Generic classes and methods combine reusability, type safety, and efficiency in a way that their non-generic counterparts cannot. Generics are most frequently used with collections and the methods that operate on them. The System.Collections.Generic namespace contains several generic-based collection classes. The non-generic collections, such as ArrayList are not recommended and are maintained for compatibility purposes.

The following code example shows how client code uses the generic GenericList<T> class to create a list of integers. Simply by changing the type argument, the following code could easily be modified to create lists of strings or any other custom type:

class TestGenericList

{

static void Main()

{

// int is the type argument

GenericList<int> list = new GenericList<int>();

for (int x = 0; x < 10; x++)

{

list.AddHead(x);

}

foreach (int i in list)

{

System.Console.Write(i + " ");

}

System.Console.WriteLine("\nDone");

}

}

Generics overview

* Use generic types to maximize code reuse, type safety, and performance.
* The most common use of generics is to create collection classes.
* The .NET class library contains several generic collection classes in the System.Collections.Generic namespace. The generic collections should be used whenever possible instead of classes such as ArrayList in the System.Collections namespace.
* You can create your own generic interfaces, classes, methods, events, and delegates.
* Generic classes may be constrained to enable access to methods on particular data types.
* Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

### **10. What is the difference between an Array and ArrayList in C#?**

An array is a collection of similar variables clubbed together under one common name. While ArrayList is a collection of objects that can be indexed individually. With ArrayList you can access a number of features like dynamic memory allocation, adding, searching, and sorting items in the ArrayList.

* When declaring an array the size of the items is fixed therefore, the memory allocation is fixed. But with ArrayList, it can be increased or decreased dynamically.
* Array belongs to system.array namespace while ArrayList belongs to the system.collection namespace.
* All items in an array are of the same datatype while all the items in an ArrayList can be of the same or different data types.
* While arrays cannot accept null, ArrayList can accept null values.

### **11. What is inheritance? Does C# support multiple inheritance?**

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

The two functions are used for typecasting the data types:

Boxing: Boxing converts value type (int, char, etc.) to reference type (object) which is an implicit conversion process using object value.

*Example:*

int num = 23; // 23 will assigned to num

Object Obj = num; // Boxing

Unboxing: Unboxing converts reference type (object) to value type (int, char, etc.) using an explicit conversion process.

*Example:*

int num = 23; // value type is int and assigned value 23

Object Obj = num; // Boxing

int i = (int)Obj; // Unboxing

### **13. What are Properties in C#?**

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called accessors. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

Properties overview

* Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
* A get property accessor is used to return the property value, and a set property accessor is used to assign a new value. In C# 9 and later, an init property accessor is used to assign a new value only during object construction. These accessors can have different access levels. For more information, see Restricting Accessor Accessibility.
* The value keyword is used to define the value being assigned by the set or init accessor.
* Properties can be read-write (they have both a get and a set accessor), read-only (they have a get accessor but no set accessor), or write-only (they have a set accessor, but no get accessor). Write-only properties are rare and are most commonly used to restrict access to sensitive data.
* Simple properties that require no custom accessor code can be implemented either as expression body definitions or as auto-implemented properties.

Properties with backing fields

One basic pattern for implementing a property involves using a private backing field for setting and retrieving the property value. The get accessor returns the value of the private field, and the set accessor may perform some data validation before assigning a value to the private field. Both accessors may also perform some conversion or computation on the data before it is stored or returned.

The following example illustrates this pattern. In this example, the TimePeriod class represents an interval of time. Internally, the class stores the time interval in seconds in a private field named \_seconds. A read-write property named Hours allows the customer to specify the time interval in hours. Both the get and the set accessors perform the necessary conversion between hours and seconds. In addition, the set accessor validates the data and throws an ArgumentOutOfRangeException if the number of hours is invalid.

**using System;**

**class TimePeriod**

**{**

**private double \_seconds;**

**public double Hours**

**{**

**get { return \_seconds / 3600; }**

**set {**

**if (value < 0 || value > 24)**

**throw new ArgumentOutOfRangeException(**

**$"{nameof(value)} must be between 0 and 24.");**

**\_seconds = value \* 3600;**

**}**

**}**

**}**

**class Program**

**{**

**static void Main()**

**{**

**TimePeriod t = new TimePeriod();**

**// The property assignment causes the 'set' accessor to be called.**

**t.Hours = 24;**

**// Retrieving the property causes the 'get' accessor to be called.**

**Console.WriteLine($"Time in hours: {t.Hours}");**

**}**

**}**

**// The example displays the following output:**

**// Time in hours: 24**

### **14. What are partial classes in C#?**

Partial classes implement the functionality of a single class into multiple files. These multiple files are combined into one during compile time. The partial class can be created using the partial keyword.

You can easily split the functionalities of methods, interfaces, or structures into multiple files. You can even add nested partial classes.

public partial Clas\_name

{

// code

}

### **15. What is the difference between late binding and early binding in C#?**

When an object is assigned to an object variable of the specific type, then the C# compiler performs the binding with the help of .NET Framework. C# performs two different types of bindings which are:

Early Binding or Static Binding

Late Binding or Dynamic Binding

Early Binding

It recognizes and checks the methods, or properties during compile time. In this binding, the compiler already knows about what kind of object it is and what are the methods or properties it holds, here the objects are static objects. The performance of early binding is fast and it is easy to code. It decreases the number of run-time errors.

Explanation: In the below example, we have a class named as Geeks. This class contains details() method. Here, the compiler already knows about the properties and methods present in Geeks. But when we try to call mymethod() then it will throw an error because this method is not known by the compiler.

// C# program to illustrate the

// concept of early binding

using System;

class Geeks {

// data members

public string name;

public string subject;

// public method

public void details(string name, string subject)

{

this.name = name;

this.subject = subject;

Console.WriteLine("Myself: " + name);

Console.WriteLine("My Favorite Subject is: " + subject);

}

}

// Driver class

class GFG {

// Main Method

static void Main(string[] args)

{

// creating object of Geeks class

Geeks g = new Geeks();

// Calling the method of Geeks class

g.details("Ankita", "C#");

// Calling "mymethod()" gives error

// because this method does not

// belong to class Geeks or compiler

// does not know mymethod() at compile time

g.mymethod();

}

}

### **16. What are the Arrays in C#?**

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements. If you want the array to store elements of any type, you can specify object as its type. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from Object.

class TestArraysClass

{

static void Main()

{

// Declare a single-dimensional array of 5 integers.

int[] array1 = new int[5];

// Declare and set array element values.

int[] array2 = new int[] { 1, 3, 5, 7, 9 };

// Alternative syntax.

int[] array3 = { 1, 2, 3, 4, 5, 6 };

// Declare a two dimensional array.

int[,] multiDimensionalArray1 = new int[2, 3];

// Declare and set array element values.

int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

// Declare a jagged array.

int[][] jaggedArray = new int[6][];

// Set the values of the first array in the jagged array structure.

jaggedArray[0] = new int[4] { 1, 2, 3, 4 };

}

}

An array has the following properties:

* An array can be single-dimensional, multidimensional or jagged.
* The number of dimensions and the length of each dimension are established when the array instance is created. These values can't be changed during the lifetime of the instance.
* The default values of numeric array elements are set to zero, and reference elements are set to null.
* A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to null.
* Arrays are zero indexed: an array with n elements is indexed from 0 to n-1.
* Array elements can be of any type, including an array type.
* Array types are reference types derived from the abstract base type Array. All arrays implement IList, and IEnumerable. You can use the foreach statement to iterate through an array. Single-dimensional arrays also implement IList<T> and IEnumerable<T>.

### **17. What are Indexers in C#?**

Indexers allow instances of a class or struct to be indexed just like arrays. The indexed value can be set or retrieved without explicitly specifying a type or instance member. Indexers resemble properties except that their accessors take parameters.

The following example defines a generic class with simple get and set accessor methods to assign and retrieve values. The Program class creates an instance of this class for storing strings.

using System;

class SampleCollection<T>

{

// Declare an array to store the data elements.

private T[] arr = new T[100];

// Define the indexer to allow client code to use [] notation.

public T this[int i]

{

get { return arr[i]; }

set { arr[i] = value; }

}

}

class Program

{

static void Main()

{

var stringCollection = new SampleCollection<string>();

stringCollection[0] = "Hello, World";

Console.WriteLine(stringCollection[0]);

}

}

// The example displays the following output:

// Hello, World.

* Indexers enable objects to be indexed in a similar manner to arrays.
* A get accessor returns a value. A set accessor assigns a value.
* The “this” keyword is used to define the indexer.
* The value keyword is used to define the value being assigned by the set accessor.
* Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.
* Indexers can be overloaded.
* Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.

### **18. Difference between the Equality Operator (==) and Equals() Method in C#?**

Equality operator (==) is a reference type which means that if equality operator is used, it will return true only if both the references point to the same object.

Equals() method: Equals method is used to compare the values carried by the objects. int x=10, int y=10. If x==y is compared then, the values carried by x and y are compared which is equal and therefore they return true.

Equality operator: Compares by reference

Equals(): Compares by value

### **19. What are the different ways in which a method can be Overloaded in C#?**

Overloading means when a method has the same name but carries different values to use in a different context. Only the main() method cannot be overloaded.

In order to overload methods in C#,

* Change the number of parameters in a method, or
* Change the order of parameters in a method, or
* Use different data types for parameters  
  In these ways, you can overload a method multiple times.

**public** **class** **Area** {

**public** double **area**(double x) {

double area = x \* x;

**return** area;

}

**public** double **area**(double a, double b) {

double area = a \* b;

**return** area;

}

}

### 

### **20. What is Reflection in C#?**

Reflection provides objects (of type Type) that describe assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you are using attributes in your code, reflection enables you to access them. For more information, see Attributes.

Here's a simple example of reflection using the GetType() method - inherited by all types from the Object base class - to obtain the type of a variable:

// Using GetType to obtain type information:

int i = 42;

Type type = i.GetType();

Console.WriteLine(type);

The output is: System.Int32.

The following example uses reflection to obtain the full name of the loaded assembly.

// Using Reflection to get information of an Assembly:

Assembly info = typeof(int).Assembly;

Console.WriteLine(info);

The output is: System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e.

### **21. What is the difference between constant and readonly in C#?**

The readonly keyword is different from the const keyword. A const field can only be initialized at the declaration of the field. A readonly field can be assigned multiple times in the field declaration and in any constructor. Therefore, readonly fields can have different values depending on the constructor used. Also, while a const field is a compile-time constant, the readonly field can be used for run-time constants

**Readonly**

In a field declaration, readonly indicates that assignment to the field can only occur as part of the declaration or in a constructor in the same class. A readonly field can be assigned and reassigned multiple times within the field declaration and constructor.

A readonly field can't be assigned after the constructor exits. This rule has different implications for value types and reference types:

* Because value types directly contain their data, a field that is a readonly value type is immutable.
* Because reference types contain a reference to their data, a field that is a readonly reference type must always refer to the same object. That object isn't immutable. The readonly modifier prevents the field from being replaced by a different instance of the reference type. However, the modifier doesn't prevent the instance data of the field from being modified through the read-only field.
* In a readonly struct type definition, readonly indicates that the structure type is immutable. For more information, see the readonly struct section of the Structure types article.
* In an instance member declaration within a structure type, readonly indicates that an instance member doesn't modify the state of the structure. For more information, see the readonly instance members section of the Structure types article.
* In a ref readonly method return, the readonly modifier indicates that method returns a reference and writes aren't allowed to that reference.

The readonly struct and ref readonly contexts were added in C# 7.2. readonly struct members were added in C# 8.0

Readonly field example

In this example, the value of the field year can't be changed in the method ChangeYear, even though it's assigned a value in the class constructor:

class Age

{

private readonly int \_year;

Age(int year)

{

\_year = year;

}

void ChangeYear()

{

//\_year = 1967; // Compile error if uncommented.

}

}

### 

### **22. What is the difference between String and StringBuilder in C#?**

The major difference between String and StringBuilder is that String objects are immutable while StringBuilder creates a mutable string of characters. StringBuilder will make the changes to the existing object rather than creating a new object.

StringBuilder simplifies the entire process of making changes to the existing string object. Since the String class is immutable, it is costlier to create a new object every time we need to make a change. So, the StringBuilder class comes into picture which can be evoked using the System.Text namespace.

In case, a string object will not change throughout the entire program, then use String class or else StringBuilder.

For ex:

string s = string.Empty;

**for** (i = 0; i < 1000; i++)

{

s += i.ToString() + " ";

}

Here, you’ll need to create 2001 objects out of which 2000 will be of no use.

The same can be applied using StringBuilder:

StringBuilder sb = **new** StringBuilder();

**for** (i = 0; i < 1000; i++)

{

sb.Append(i); sb.Append(' ');

}

By using StringBuilder here, you also de-stress the memory allocator.

### **23. Delegates in C#**

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates. You create a custom method, and a class such as a windows control can call your method when a certain event occurs. The following example shows a delegate declaration:

public delegate int PerformCalculation(int x, int y);

Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either static or an instance method. This flexibility means you can programmatically change method calls, or plug new code into existing classes.

Note

In the context of method overloading, the signature of a method does not include the return value. But in the context of delegates, the signature does include the return value. In other words, a method must have the same return type as the delegate.

Delegates have the following properties:

* Delegates are similar to C++ function pointers, but delegates are fully object-oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.
* Delegates allow methods to be passed as parameters.
* Delegates can be used to define callback methods.
* Delegates can be chained together; for example, multiple methods can be called on a single event.
* Methods don't have to match the delegate type exactly. For more information, see Using Variance in Delegates.
* Lambda expressions are a more concise way of writing inline code blocks. Lambda expressions (in certain contexts) are compiled to delegate types. For more information about lambda expressions, see Lambda expressions.

Using Delegate **EX1:**

**public delegate void Del(string message);**

**// Create a method for a delegate.**

**public static void DelegateMethod(string message)**

**{**

**Console.WriteLine(message);**

**}**

// Instantiate the delegate.

**Del handler = DelegateMethod;**

**// Call the delegate.**

**handler("Hello World");**

**EX2:**

The following example method uses the Del type as a parameter:

**public static void MethodWithCallback(int param1, int param2, Del callback)**

**{**

**callback("The number is: " + (param1 + param2).ToString());**

**}**

You can then pass the delegate created above to that method:

**MethodWithCallback(1, 2, handler);**

and receive the following output to the console:

**The number is: 3**

**EX3:**

**public class MethodClass**

**{**

**public void Method1(string message) { }**

**public void Method2(string message) { }**

**}**

Along with the static DelegateMethod shown previously, we now have three methods that can be wrapped by a Del instance.

A delegate can call more than one method when invoked. This is referred to as multicasting. To add an extra method to the delegate's list of methods—the invocation list—simply requires adding two delegates using the addition or addition assignment operators ('+' or '+=').

**var obj = new MethodClass();**

**Del d1 = obj.Method1;**

**Del d2 = obj.Method2;**

**Del d3 = DelegateMethod;**

//Both types of assignment are valid.

**Del allMethodsDelegate = d1 + d2;**

**allMethodsDelegate += d3;**

To remove a method from the invocation list, use the subtraction or subtraction assignment operators (- or -=).

//remove Method1

**allMethodsDelegate -= d1;**

// copy AllMethodsDelegate while removing d2

**Del oneMethodDelegate = allMethodsDelegate - d2;**

to find the number of methods in a delegate's invocation list, you may write:

**int invocationCount = d1.GetInvocationList().GetLength(0);**

### **24. Structs In C#**

Structs are similar to classes in that they represent data structures that can contain data members and function members. However, unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. Key to these data structures is that they have few data members, that they do not require use of inheritance or referential identity, and that they can be conveniently implemented using value semantics where assignment copies the value instead of the reference.

Structs differ from classes in several important ways:

* Structs are value types (Value semantics).
* All struct types implicitly inherit from the class System.ValueType (Inheritance).
* Assignment to a variable of a struct type creates a copy of the value being assigned (Assignment).
* The default value of a struct is the value produced by setting all value type fields to their default value and all reference type fields to null (Default values).
* Boxing and unboxing operations are used to convert between a struct type and object (Boxing and unboxing).
* The meaning of this is different for structs (This access).
* Instance field declarations for a struct are not permitted to include variable initializers (Field initializers).
* A struct is not permitted to declare a parameterless instance constructor (Constructors).
* A struct is not permitted to declare a destructor (Destructors).

### **25. Abstract In C#**

The abstract modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes, not instantiated on its own. Members marked as abstract must be implemented by non-abstract classes that derive from the abstract class.

abstract class Shape

{

public abstract int GetArea();

}

class Square : Shape

{

private int \_side;

public Square(int n) => \_side = n;

// GetArea method is required to avoid a compile-time error.

public override int GetArea() => \_side \* \_side;

static void Main()

{

var sq = new Square(12);

Console.WriteLine($"Area of the square = {sq.GetArea()}");

}

}

// Output: Area of the square = 144

Abstract classes have the following features:

An abstract class cannot be instantiated.

An abstract class may contain abstract methods and accessors.

It is not possible to modify an abstract class with the sealed modifier because the two modifiers have opposite meanings. The sealed modifier prevents a class from being inherited and the abstract modifier requires a class to be inherited.

A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

Use the abstract modifier in a method or property declaration to indicate that the method or property does not contain implementation.

Abstract methods have the following features:

An abstract method is implicitly a virtual method.

Abstract method declarations are only permitted in abstract classes.

Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no curly braces ({ }) following the signature. For example:

public abstract void MyMethod();

The implementation is provided by a method override, which is a member of a non-abstract class.

It is an error to use the static or virtual modifiers in an abstract method declaration.

Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.

It is an error to use the abstract modifier on a static property.

An abstract inherited property can be overridden in a derived class by including a property declaration that uses the override modifier.

An abstract class must provide implementation for all interface members.

An abstract class that implements an interface might map the interface methods onto abstract methods. For example:

### **26. Virtual In C#**

The virtual keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it:

public virtual double Area()

{

return x \* y;

}

The implementation of a virtual member can be changed by an overriding member in a derived class.

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

By default, methods are non-virtual. You cannot override a non-virtual method.

You cannot use the virtual modifier with the static, abstract, private, or override modifiers. The following example shows a virtual property:

class MyBaseClass

{

// virtual auto-implemented property. Overrides can only

// provide specialized behavior if they implement get and set accessors.

public virtual string Name { get; set; }

// ordinary virtual property with backing field

private int \_num;

public virtual int Number

{

get { return \_num; }

set { \_num = value; }

}

}

class MyDerivedClass : MyBaseClass

{

private string \_name;

// Override auto-implemented property with ordinary property

// to provide specialized accessor behavior.

public override string Name

{

get

{

return \_name;

}

set

{

if (!string.IsNullOrEmpty(value))

{

\_name = value;

}

else

{

\_name = "Unknown";

}

}

}

}

### 

### 

### 

### **27. Object-Oriented programming (C#)**

C# is an object-oriented programming language. The four basic principles of object-oriented programming are:

* Abstraction:- Modeling the relevant attributes and interactions of entities as classes to define an abstract representation of a system.
* Encapsulation:- Hiding the internal state and functionality of an object and only allowing access through a public set of functions.
* Inheritance:- Ability to create new abstractions based on existing abstractions.
* Polymorphism:- Ability to implement inherited properties or methods in different ways across multiple abstractions.

### **28.** What is not true of Dependency Injection?

A Easier to create unit tests by mocking dependencies

B Higher maintainability by having loosely couple classes

C  **Higher run-time performance due to injection**

### **29. What is a dependency injection container?**

A Virtualize your application to better isolate from crashes

B **A framework to create dependencies and automatically inject them**

C A box of Craftsman injectors

### **30. Why is the interface used for Dependency Injection?**

A **To abstract out the implementation, thus decouples the dependency of implementation.**

B Because it supports multiple inheritance

C Reflection can be leveraged to instantiate interface

### **31. What is the purpose of mocking a dependent object?**

A **To control the dependent object behavior so it's easier to test the depending class**

B Because it bullied other classes

C It allows you to test the mocked class behavior

### **32. SOLID design principle.**

1. **The Single Responsibility Principle.:** - It is often defined as: An object should only have one reason to change; the longer the file or class, the more difficult it will be to achieve this.

### **33. Design patterns.:-** Design patterns are solutions to software design problems you find again and again in real-world application development. Patterns are about reusable designs and interactions of objects. They are categorized in three groups: Creational, Structural, and Behavioral.

Creational

1. Factory Method: The Factory Method design pattern defines an interface for creating an object, but let subclasses decide which class to instantiate. This pattern lets a class defer instantiation to subclasses.

Frequency of use: high

Structural

Behavioral

1. Strategy: - The Strategy design pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern lets the algorithm vary independently from clients that use it.

Frequency of use: high

1. Observer:- The Observer design pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Frequency of use: high

1. Factory Method: The Factory Method design pattern defines an interface for creating an object, but let subclasses decide which class to instantiate. This pattern lets a class defer instantiation to subclasses.

Frequency of use: high

### **34. Types:-**

### **Type parameters:-** Generic classes define type parameters. Type parameters are a list of type parameter names enclosed in angle brackets. Type parameters follow the class name. The type parameters can then be used in the body of the class declarations to define the members of the class. In the following example, the type parameters of Pair are TFirst and TSecond:

public class Pair<TFirst, TSecond>

{

public TFirst First { get; }

public TSecond Second { get; }

public Pair(TFirst first, TSecond second) =>

(First, Second) = (first, second);

}

1. **Base classes:-** A class declaration may specify a base class. Follow the class name and type parameters with a colon and the name of the base class. Omitting a base class specification is the same as deriving from type object. In the following example, the base class of Point3D is Point

public class Point3D : Point

{

public int Z { get; set; }

public Point3D(int x, int y, int z) : base(x, y)

{

Z = z;

}

}