C# Interview Questions & Answers

By Shailendra Chauhan

Microsoft MVP, Founder & CEO - Dot Net Tricks

DotNetTricks

C# Interview Questions & Answers

Release History

- Initial Release 1.0 - 21st Nov 2018
- Second Release 1.1 - 4th Jan 2019
- Third Release 1.2 - 26th Jan 2019

About Dot Net Tricks

Dot Net Tricks is founded by Shailendra Chauhan (Microsoft MVP), in Jan 2010. Dot Net Tricks came into existence in form of a blog post over various technologies including .NET, C#, SQL Server, ASP.NET, ASP.NET MVC, JavaScript, Angular, Node.js and Visual Studio etc.

The company which is currently registered by a name of Dot Net Tricks Innovation Pvt. Ltd. came into the shape in 2015. Dot Net Tricks website has an average footfall on the tune of 300k+ per month. The site has become a cornerstone when it comes to getting skilled-up on .NET technologies and we want to gain the same level of trust in other technologies. This is what we are striving for.

We have a very large number of trainees who have received training from our platforms and immediately got placement in some of the reputed firms testifying our claims of providing quality training. The website offers you a variety of free study material in form of articles.

Dot Net Tricks Training Solutions

Dot Net Tricks provide you training in traditional as well as new age technologies, via various formats.

Master Courses (Instructor-led)

For a beginner who needs regular guidance, we have a fully packed Master Courses. They are almost equal to semester courses taught in engineering colleges when it comes to length, breadth of content delivery, the only difference instead of 5-6 months, they take approx. 16-weekend classes (2 months).

The detail about Master courses can be found here: https://www.dotnettricks.com/instructor-led-courses

Hands-On Learning (Learn to code)

Hands-On Learning courses give you the confidence to code and equally helpful to work in real-life scenarios. This course is composed of hands-on exercise using IDE or cloud labs so that you can practice each and everything by yourself. You can learn to code at your own pace, time and place.

The detail about Hands-On Learning courses can be found here: https://www.scholarhat.com

Skill Bootcamps (Instructor-led)

Professionals who don’t have two months’ time and want to get skilled up in least possible time due to some new project that their company has to take in very near future, we have designed Skill Bootcamps Concept, where you will get trained on consecutive days in a fast-paced manner, where our full focus is going to be on hands-on delivery of technological exercises.

The detail about Skill Bootcamps can be found here: https://www.dotnettricks.com/skill-bootcamp

Self-paced Courses

Self-paced courses give you the liberty to study at your own pace, time and place. We understand everyone has their own comfort zone, some of you can afford to dedicate 2 hours a day, some of you not. Keeping this thing in mind, we created these self-paced courses. While creating these courses we have ensured that quality of courses doesn't get compromise at any parameter, and they also will be able to produce the same results as our other course formats, given the fact you will be able to put your own honest effort.

The detail about Self-paced courses can be found here: https://www.dotnettricks.com/self-paced-courses

Corporate Training (Online and Classroom)

Dot Net Tricks having a pool of mentors who help the corporate to enhance their employment skills as per changing technology landscape. Dot Net Tricks offers customized training programs for new hires and experienced employees through online and classroom mode. As a trusted and resourceful training partner, Dot Net Tricks helps the corporate to achieve success with its industry-leading instructional design and customer training initiatives.

The detail about Corporate Training can be found here: https://www.dotnettricks.com/corporate-training

Learning Platform

We have a very robust technology platform to answer the needs of all our trainees, no matter which program they enrolled in. We have a very self-intuitive Learning Management System (LMS), which help you in remain focused and keeping an eye over your progress.

We offer two Learning Platforms as given below:

1. Dot Net Tricks: https://www.dotnettricks.com
2. Scholar Hat: https://www.scholarhat.com

Apart from these, we also provide on-demand Skill bootcamps and personalized project consultation.

Dedication

My mother Mrs Vriksha Devi and my wife Reshu Chauhan deserve to have their name on the cover as much as I do for all their support made this possible. I would like to say thanks to all my family members Virendra Singh(father), Jaishree and Jyoti(sisters), Saksham and Pranay(sons), friends, to you and to readers or followers of my articles at https://www.dotnettricks.com/mentors/shailendra-chauhan to encourage me to write this book.

-Shailendra Chauhan

Introduction

If you want to crack your C# interview, you've come across the right book. This is the book with which you will be confident to answers the questions on C# language. This book will teach you the C# from beginners to advanced.

This book is designed specifically to teach you the concepts of C# language that can be replaced by any other modern programming languages, such as Java, C++, PHP or Python.

If you are a working professional or having .NET experience, examine this book in details and see if you are familiar with all subjects the author has covered. In this book, you will find the answers of most complex questions the people ask in an interview.

So, what where my qualification to write this book? My qualification and inspiration come from my enthusiasm for and the experience with the technology and from my analytic and initiative nature. Being a consultant, corporate trainer, and blogger, I have thorough knowledge and understandings of .NET technologies. My inspiration and knowledge have also come from many years of my working experience and research over it.

So, the next question is who this book is for? This book is best suited for beginners and professionals. It is intended for anyone who so far has not engaged seriously in C# programming and would like to start a career in .NET. This book starts from scratch and teaches you all the core and advanced concepts of C# language in step by step way. It won't teach you absolutely everything to become a software developer and working at a software company, but it will help you to turn your programming into your profession.

This book is not only the C# interview book but it is more than that. This book helps you to get an in-depth knowledge of C# with a simple and elegant way. This book is updated to the latest version of C# 7.3.

I hope you will enjoy this book and find it useful. At the same time, I also encourage you to become a continuous reader of learning platform www.dotnettricks.com and be the part of the discussion. But most importantly practice a lot and enjoy the technology. That's what it's all about.

To get the latest information on C#, I encourage you to follow the official Microsoft document website at https://docs.microsoft.com/en-us/dotnet/csharp.

All the best for your interview and happy programming!

About the Author

Shailendra Chauhan - An Entrepreneur, Author, Architect, Corporate Trainer, and Microsoft MVP

He is the **Founder and CEO** of DotNetTricks which is a brand when it comes to e-Learning. DotNetTricks provides training and consultation over an array of technologies like **Cloud, .NET, Angular, React, Node and Mobile Apps development**. He has been awarded as **Microsoft MVP** three times in a row (2016-2018).

He has changed many lives from his writings and unique training programs. He has a number of most sought-after books to his name which have helped job aspirants in **cracking tough interviews** with ease.

Moreover, and to his credit, he has delivered **1000+ training sessions** to professionals worldwide in Microsoft .NET technologies and other technologies including JavaScript, AngularJS, Node.js, React and NoSQL Databases. In addition, he provides **Instructor-led online training**, **hands-on workshop** and **corporate training** programs.

Shailendra has a strong combination of **technical skills and solution development for complex application architecture with proven leadership and motivational skills** have elevated him to world-renowned status, placing him at the top of the list of most sought-after trainers.

"I always keep up with new technologies and learning new skills to deliver the best to my students," says Shailendra Chauhan, he goes on to acknowledge that the betterment of his followers and enabling his students to realize their goals are his prime objective and a great source of motivation and satisfaction.

Shailendra Chauhan - **"Follow me and you too will have the key that opens the door to success"**

How to Contact Us

Although the author of this book has tried to make this book as accurate as it possible but if there is something strikes you as odd, or you find an error in the book please drop a line via e-mail.

The e-mail addresses are listed as follows:

- mentor@dotnettricks.com
- info@dotnettricks.com

We always happy to hear from our readers. Please provide your valuable feedback and comments!

You can follow us on YouTube, Facebook, Twitter, LinkedIn and Google Plus or subscribe to RSS feed.

Table of Contents

1

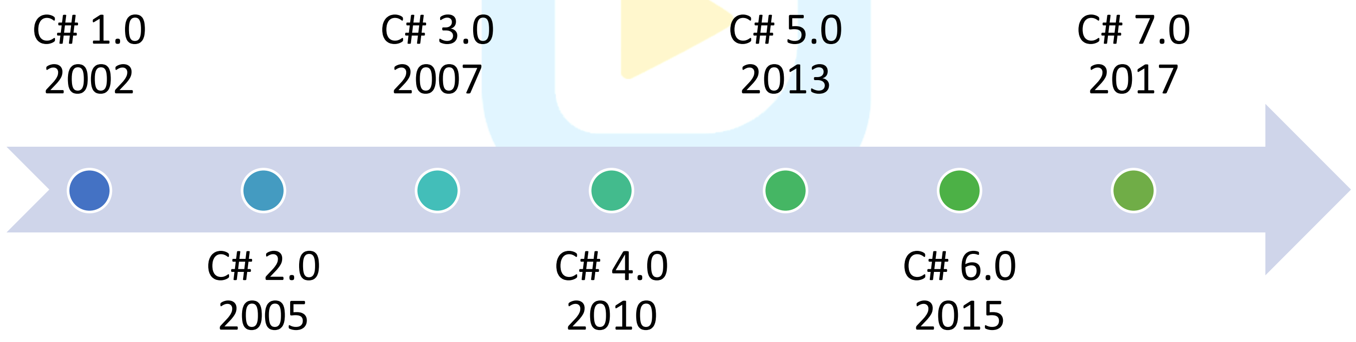
Introducing C#

Q1. What is C#?

Ans. C# pronounced as "See Sharp". It is an object-oriented programing language developed by Microsoft, which runs under the .NET platform. Its syntaxes are similar to C++ or Java. The most recent version of C# is 7.3 which is introduced with Visual Studio 2017 update 15.7. C# is widely used for developing web applications, desktop applications, mobile apps and games etc. Now, C# can be run on Mac, Linux/Unix and Windows using .NET Core.

Q2. Explain evolution history of C#?

Ans. The evolution history of C# is given below –



Q3. What features are added to different versions of C#?

C# 1.0

- Managed Code
- IDE - Visual Studio .NET 2002, 2003
- .NET Framework - 1.0, 1.1

C# 2.0

- Generics

- Static Classes
- Partial types
- Anonymous methods
- Iterators
- Nullable types
- Asymmetric Property and Indexer Accessors
- Delegate Inference
- Covariance and Contra-variance
- IDE - Visual Studio 2005
- .NET Framework - 2.0

C# 3.0

- Implicit types (var)
- Partial Methods
- Object and collection initializers
- Auto-Implemented properties
- Anonymous types
- Extension methods
- LINQ
- Query expressions
- Lambda expressions
- Expression trees
- IDE - Visual Studio 2008
- .NET Framework - 3.5

C# 4.0

- Dynamic binding
- Named arguments
- Optional Parameters
- Generic Covariance and Contra-variance
- COM Interop

- IDE -Visual Studio 2010
- .NET Framework - 4.0

C# 5.0

- Asynchronous methods
- Caller info attributes
- IDE - Visual Studio 2012, 2013
- .NET Framework - 4.5, 4.5.1

C# 6.0

- Auto Property Initializer
- Primary Constructors
- Dictionary Initializer
- Declaration Expressions
- Static Using
- await inside catch block
- Exception Filters
- Null-Conditional Operator
- IDE - Visual Studio 2015
- .NET Framework - 4.6

C# 7.0

- Local Functions
- Literal improvements
- Digit Separators
- Pattern matching
- ref returns and ref Locals
- Throw Expressions
- Expression Bodied Members
- Deconstruction
- IDE - Visual Studio 2017

Q4. What is Dynamic binding?

Ans. This feature is derived from dynamic languages such as Python, Ruby and JavaScript which resolve the types and members at runtime instead of compile time.

```
dynamic d = "hello c#";
Console.WriteLine(d.ToUpper()); // HELLO C#
Console.WriteLine(d.Show()); // Compiles OK, but gives error at runtime
```

Q5. What are Optional parameters?

Ans. In C# 4.0, methods parameters can be specified as optional by providing a default value. When the method having optional parameter is invoked, optional parameters can be omitted.

```
void Sum(int a, int b = 10)
{
  Console.WriteLine(a+b);
}
```

Above method can be called as:

```
Sum(5); // 15
```

Q6. What are Named arguments?

Ans. In C# 4.0, any argument can be passed by parameter name instead of parameter position. The Sum() can be called as :

```
Sum(b:5,a:10); // 15
```

Q7. What is Type variance?

Ans. C# 4.0, allows generic interfaces and generic delegates to mark their type parameters as covariant or contravariant. This enables the following code to work:

```
IEnumerable<string> str;
//TO DO: assign values to str

IEnumerable<object> obj = str;
```

COM-specific interoperability features

Dynamic binding, as well as named and optional arguments, help to make programming against COM less painful than today. On top of that, however, we are adding a number of other features that further improve the interoperability experience specifically with COM.

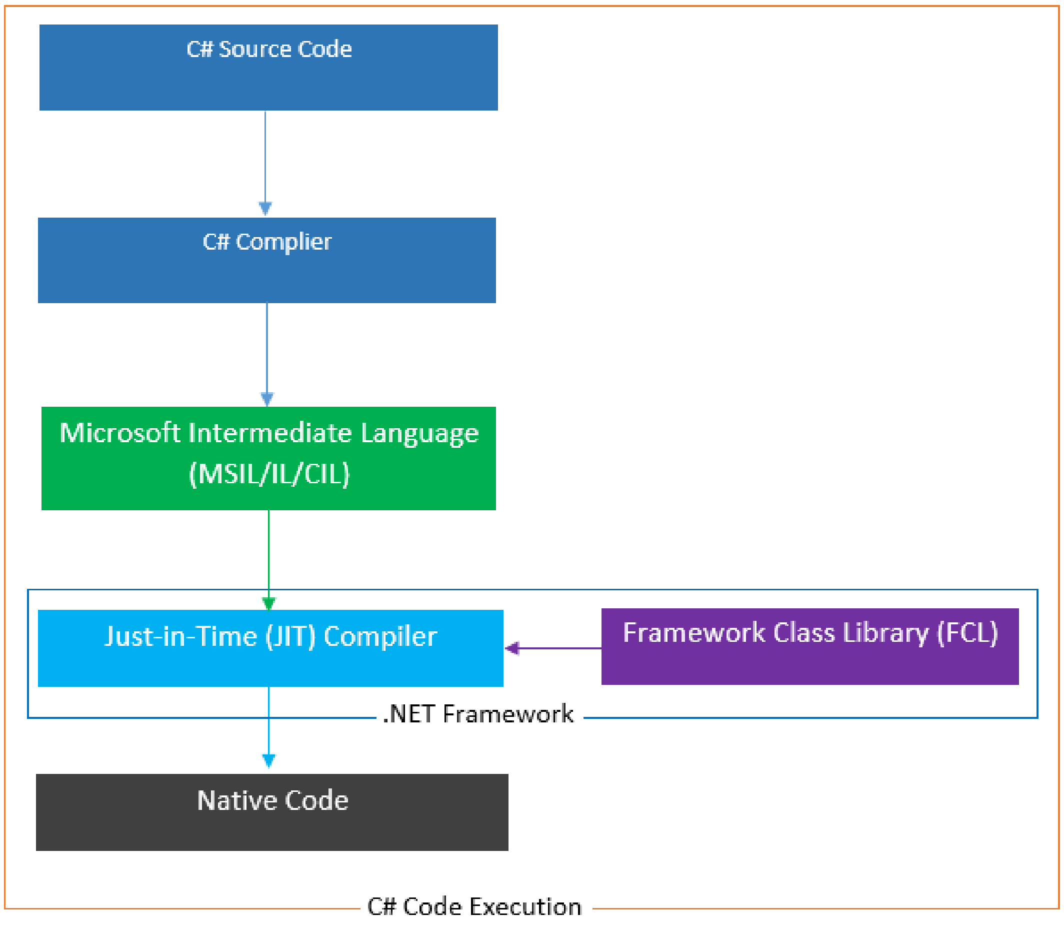
Q8. What are Asynchronous Methods?

Ans. In C# 5.0 two keywords async and await are introduced which allow you to write asynchronous code more easily and intuitively like as synchronous code. Before C# 5.0, for writing asynchronous code, you need to define callbacks to capture what happens after an asynchronous process finish. These keywords are used in a combination of each other and an await operator is applied to a one or more than one expression of an async method. An async method returns a Task or Task<TResult> that represents the ongoing work of the method.

```
public async Task<IEnumerable<Product>> GetProductList()
{
   HttpClient client = new HttpClient();
   Uri address = new Uri("http://dotnet-tricks.com/");
   client.BaseAddress = address;

   HttpResponseMessage response = await client.GetAsync("myservice/product/ProductList");

   if (response.IsSuccessStatusCode)
   {
      var list = await response.Content.ReadAsAsync<IEnumerable<Product>>();
      return list;
   }
   else
   {
     return null;
   }
}
```

Q9. Explain the C# code execution process?

Ans. C# code execution life cycle diagram is shown as follows-


C# Source Code
C# Complier
Microsoft Intermediate Language (MSIL/IL/CIL)
Just-in-Time (JIT) Compiler
Framework Class Library (FCL)
.NET Framework
Native Code
C# Code Execution

2

Data Type and Operators

Q1. What is Datatype?

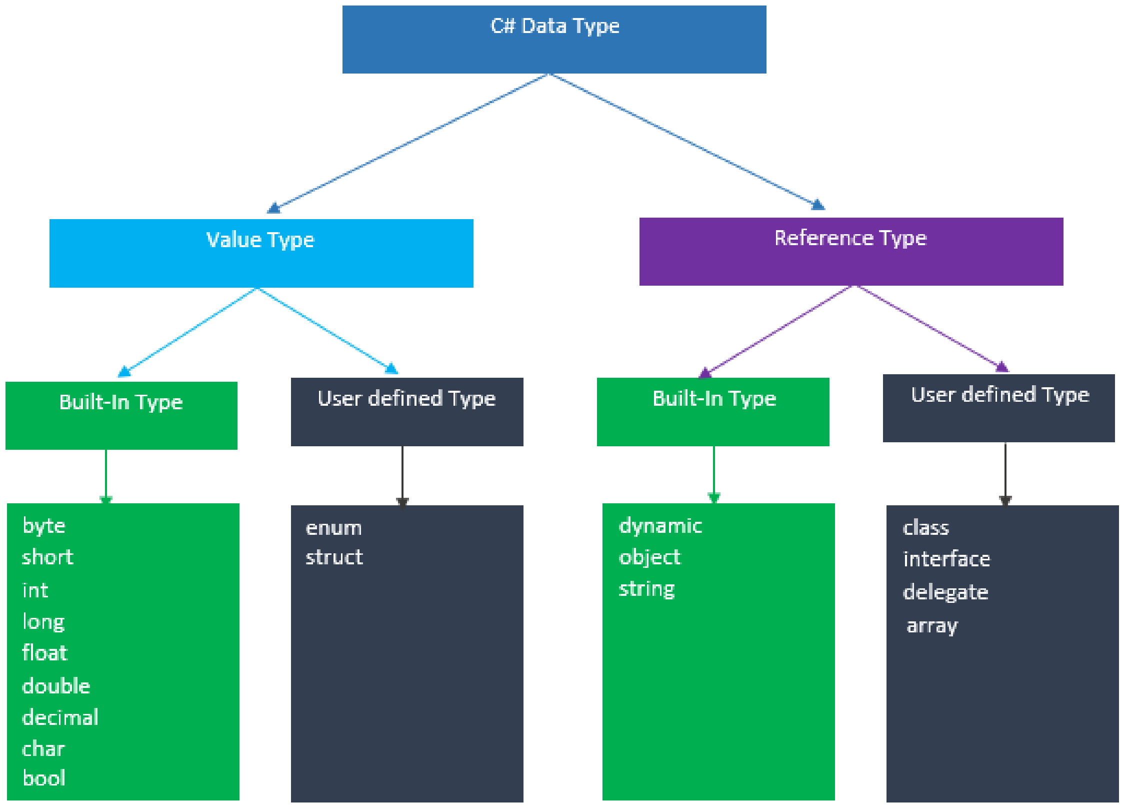
Ans. Data Type refers to the type of data that can be stored in a variable. It also specifies how much memory would be allocated to a variable and the operations that can be performed on that variable. C# is rich in data type which is broadly divided into two categories: Value Type and Reference Type.

Q2. What do you mean by value type and reference type?

Or

What is the difference between value types and reference types?

Ans. **Value Type** - A value type variable stores actual values. Values types are of two types - built-in and user-defined. Value types are stored in a stack and derived from System.ValueType class.



DotNetTricks

Reference Type - A reference type variable stores a reference to the actual value. Typically, a reference type contains a pointer to another memory location that stores the actual data. Reference types are of two types - built-in and user-defined. Reference types are stored in a heap and derived from System.Object class.

Q3. What are Nullable types? How to use Nullable types in .NET Framework?

Ans. Value types that can accept a normal value or a null value are referred to as Nullable types. These are instances of the Nullable struct.

For example, A Nullable<bool> pronounced as "Nullable of bool" and can have the values true, false or null. Nullable types can also be defined using **?** type modifier. This token is placed immediately after the value type being defined as nullable.

```
//assigning normal value
Nullable<bool> flag = true;
//Or You can also define as
bool? flag = true;

//assigning normal value
Nullable<int> x = 20;
//Or You can also define as
int? x = 20;
```

You can also assign null value to flag and d since both are nullable types.

```
//assigning null value
Nullable<bool> flag = null;
//Or You can also define as
bool? flag = null;

//assigning null value
Nullable<int> x = null;
//Or You can also define as
int? x = null;
```

Q4. How to check that a nullable variable is having value?

Ans. The HasValue property is used to check whether a variable having value or not. To get the value from a nullable type Value property as shown below –

```
int? x = 20;
int y;

if (x.HasValue)
{
  y = x.Value;
}
```

Note

You can't declare an implicitly nullable type variable as nullable like as string type cannot be declared as nullable type since it is implicitly nullable.

Q5. What are the differences between int, Int16, Int32 and Int64?

Ans. The differences between these are given below-

int

1. It is a primitive data type defined in C#.
2. It is mapped to Int32 of FCL (Framework class library) type.
3. It is a value type and represents System.Int32 struct.
4. It is signed and takes 32 bits always.
5. It has a minimum -2147483648 and maximum +2147483647 capacity.

Int16

1. It is an FCL (Framework class library) type.
2. In C#, short is mapped to Int16.
3. It is a value type and represents System.Int16 struct.
4. It is signed and takes 16 bits.
5. It has a minimum -32768 and maximum +32767 capacity.

Int32

1. It is an FCL (Framework class library) type.
2. In C#, int is mapped to Int32.
3. It is a value type and represents System.Int32 struct.
4. It is signed and takes 32 bits.
5. It has a minimum -2147483648 and maximum +2147483647 capacity.

Int64

1. It is an FCL (Framework class library) type.
2. In C#, long is mapped to Int64.
3. It is a value type and represents System.Int64 struct.
4. It is signed and takes 64 bits.
5. It has minimum –9,223,372,036,854,775,808 and maximum 9,223,372,036,854,775,807 capacity.

Q6. What is ?? Operator in C#?

Ans. The ?? operator is called the null-coalescing operator. It is used to define a default value for a nullable value type or reference type variable.

```
int? x = null;
int y = -1;

// when x is null, z = y
// when x is not null, z = x
int z = x ?? y; // where x and y are left and right operand
```

Q7. What are dynamic types in C#?

Ans. The dynamic type was introduced in C# 4.0. Dynamic type variables are declared using the "dynamic" keyword. A dynamic type variable bypasses the compile-time type checking and its operations are resolved at runtime. In dynamic type, if the operations are not valid then the exception would be thrown at runtime, not at compile time.

```
class Program
{
    public static void Main()
    {
        dynamic d = 1; //assigning integer
        Console.WriteLine(d);

        d = "Hi Dynamic"; //assigning string to the same variable d
        Console.WriteLine(d);

        d = TestData(); //assigning method result to the same variable d
        Console.WriteLine(d);

        // You can call anything on a dynamic variable,
        // but it may result in a runtime error
        Console.WriteLine(d.Error);
    }

    public static double TestData()
    {
        return 12.80;
    }
}

/* OutPut

  10
  Hi Dynamic
  12.80
  RuntimeBinderException was unhandled, 'double' does not contain a definition for 'Error'
*/
```

Q8. What are the differences between Object, Var and Dynamic type?

Ans. The differences between Object, Var and Dynamic type are given below:

Object	**Var**	**Dynamic**
The object was introduced with C# 1.0	Var was introduced with C# 3.0	Dynamic was introduced with C# 4.0
It can store any kind of value because the object is the base class of all type in .NET framework.	It can store any type of value but It is mandatory to initialize var types at the time of declaration.	It can store any type of the variable, similar to the old VB language variable.

The compiler has little information about the type. So, it can cause the problem at runtime while typecasting.	It is type-safe i.e. compiler has all the information about the stored value.	It is not type safe i.e. compiler doesn't have any information about the type of variable. So, it will cause a problem if the wrong properties or methods are accessed.
An object type can be passed as method argument and method also can return object type.	Var type cannot be passed as method argument and method cannot return object type. Var type work in the scope where it defined.	Dynamic type can be passed as method argument and method also can return dynamic type.
Need to cast object variable to original type to use it and performing desired operations.	No need to cast because the compiler has all information to perform operations.	Casting is not required but you need to know the properties and methods related to stored type.
Useful when you want to store multiple types of values to a single variable.	Useful when you don't know the type of assigned value i.e. type is anonymous.	Useful when you need to integrate your C# code with dynamic languages code or with the COM objects.

Q9. What is ref and out keyword in C#?

OR

What is the difference between ref and out keyword?

Ans. **Ref -** As per MSDN, the ref keyword causes an argument to be passed by reference, not by value. This means that when the value of that parameter is changed in the method, it gets reflected in the calling method.

Key Points

- An argument must be initialized before it is passed as a ref parameter.
- C# properties cannot be passed as a ref parameter since internally properties are functions and function can't be passed as a parameter.

Out - The out keyword also causes an argument to be passed by reference like "ref" keyword, but that argument can be passed without assigning any value to it.

Key Points

- An argument that is passed using an out keyword must be initialized in the method before it returns back to calling the method.
- Also, C# properties cannot be passed as an out parameter since internally properties are functions and function can't be passed as a parameter.

For example, A program uses ref and out keyword as a method argument.

```
public class Program
{
    public static void Main()
    {
        int arg1 = 0; //must be initialized
        int arg2;
        Method1(ref arg1);
        Console.WriteLine(arg1); // Now 2!

        Method2(out arg2);
        Console.WriteLine(arg2); // Now 3!
    }
    static void Method1 (ref int value)
    {
        value = 1;
    }
    static void Method2(out int value)
    {
        value = 2; //must be initialized
    }
}
/* Output
  1
  2
 */
```

Q10. Can you use out and ref for overloading as the different signature of method?

Ans. No, you cannot. Although, ref and out are treated differently at runtime they treated same at compile time. Hence a method cannot be overloaded with the same type of arguments.

Examples: These two methods are identical in terms of compilation.

```
class Class1
{
      public void Method1(out int a) // compiler error “cannot define overloaded”
      {
           // method that differ only on ref and out"
      }
      public void Method1(ref int a)
      {
           // method that differ only on ref and out"
      }
}
```

Q11. What is the params keyword in C#?

Ans. The params keyword enables a method parameter to receive n number of arguments. These n number of arguments are changed by the compiler into elements in a temporary array and actually, this array is received at receiving end. Also, the params arguments must be the last argument in the parameter list of the method.

```
public class Example
{
    static decimal Sum(decimal d1, params int[] values)
    {
        decimal total = d1;
        foreach (int value in values)
        {
            total += value;
        }
        return total;
    }

    static void Main()
    {
        decimal d1 = 10;
        int sum1 = Sum(d1, 1);
        int sum2 = Sum(d2, 1, 2);
        int sum3 = Sum(d3, 1, 2, 3);

        Console.WriteLine(sum1);
        Console.WriteLine(sum2);
        Console.WriteLine(sum3);
        Console.Read();
    }
}
/* Output
  11
  13
  16
 */
```

Q12. What do you mean by operators and what are different types of operators in c#?

Ans. An operator is a special symbol that tells to the compiler what operations can be performed on an operand or between two operands. Operators in programming languages are taken from mathematics.

C# supports the following types of operators as given below:

1. **Arithmetic Operators -** These operators are used to perform arithmetic operations. The following is a table of arithmetic operators in C#. Suppose you have two integer variables X, Y and having values 5, 2 respectively then

Operator	Name	C# Example
+	Addition	X + Y will give 7
-	Subtraction	X - Y will give 3
*	Multiplication	X * Y will give 10
/	Division	X / Y will give 2
%	Modulus	X % Y will give 1
++	Preincrement and Postincrement	X = ++Y, X = Y++ will give 3, 2
--	Predecrement and Postdecrement	X = --Y, X = Y-- will give 4, 2

2. **Relational Operators –** These operators are used to compare values and always result in boolean value (True or False). The following is a table of relational operators in C#. Suppose you have two integer variables X, Y and having values 5, 2 respectively then

Operator	Name	C# Example
>	Greater than	X > Y will give True
<	Less than	X < Y will give False
>=	Greater than or Equal to	X >= Y will give True
<=	Less than or Equal to	X <= Y will give False
==	Equal to	X == Y will give False
!=	Not Equal to	X != Y will give True

3. **Logical or Boolean Operators –** These operators are used to compare values and always result in boolean value (True or False). The following is a table of logical operators in C#. Suppose you have two boolean variables X, Y and having values True, False respectively then

Operator	Name	C# Example
||	Logical or	X || Y will give True
&&	Logical and	X && Y will give False
!	Logical not	!(X), !(Y) will give False, True

4. **Bitwise Operators –** These operators work on bits of a binary number and perform bit by bit operation. The following is a table of bitwise operators in C#. Suppose you have two integer variables X, Y and having values 4, 5 respectively and their binary equivalent would be 100, 101 then

Operator	Name	C# Example
|	Bitwise or	X | Y will give 5 i.e. 101
&	Bitwise and	X & Y will give 4 i.e. 100
~	Bitwise not	~(X), ~(Y) will give -5, -6 i.e. -101, -110
^	Bitwise exclusive-or	X^Y will give 1 i.e. 1

5. **Assignment Operators –** These operators are used to assign a new value to a variable. The following is a table of assignments operators in C#. Suppose you have two integer variables X, Y with values 5, 2 then

Operator	Name	C# Example
=	Equal to	X = Y will give 2
+=	Plus Equal to	X += Y will give 7
-=	Minus Equal to	X -= Y will give 3
*=	Multiply Equal to	X *= Y will give 10
/=	Divide Equal to	X /= Y will give 2
%=	Modulus Equal to	X %= Y will give 1

6. **Type Information Operators –** These operators are used to provides information about a particular type. The following is a table of type information operators in C#.

Operator	Description	C# Example
sizeof()	Returns the size of a value type.	sizeof(int) will give 4
typeof()	Returns the type of a class or value type.	typeof(bool) will give System.Boolean
Is	Determines whether an object is of a certain type or not	If (Renault is Car) // checks if Renault is an object of the Car class.

As	Cast an object to a certain type without raising an exception if the cast fails.	Object obj = new Object();
		// The cast fails: no exception is thrown and emp is set to null.
		Employee emp = obj as Employee;

7. **Misc. Operators** - There are some more important operators supported by C#. The following is a table of some Misc operators in C#. Suppose you have two integer variables X, Y and having values 5, 2 then

Operator	Description	C# Example
?:	Conditional Expression/Ternary Operator	X > Y ? X : Y will give 5
=>	Lambda Expression	X => { return X * Y; } will give 10
*	Pointer to a variable	*X will pointer to X variable.
&	Returns memory address of a variable	&X will give the address of X
?	Makes a value type to nullable type	bool? flag = true;

??	Provides default value for nullable value types or reference types	int? x = null;
		int y = -1;
		
		// when x is null, z = y
		// when x is not null, z = x
		int z = x ?? y;
		it results z = -1

Q13. What is the operator precedence in C#?

Ans. The following table lists grouped the C# operators in order of precedence. Operators within each group have equal precedence.

Category	**Operator**	**Associativity**
Primary	() [] -> . ++ - - new typeof sizeof checked unchecked default(T)	Left to right
Unary	+ - ! ~ ++ - - (type)* & await	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational and Type testing	<<= >>= is as	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right

Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Null-coalescing	??	Left to right
Ternary	?:	Right to left
Assignment and Lambda expression	= += -= *= /= %= >>= <<= &= ^= |= =>	Right to left

Q14. What is a ternary operator (?:) in C#?

Ans. A ternary or Conditional operator operates on a conditional expression that returns a Boolean value. If the conditional expression is true then the first expression would be evaluated otherwise the second expression would be evaluated. The syntax for the conditional operator is given below-

```
Conditional_expression ? First_expression : Second_expression;
```

This operator provides an elegant and equivalent syntax to the simple if-else statement.

```
int number = Convert.ToInt32(Console.ReadLine());
string msg;

// if-else statement
if (number < 0)
   msg = "negative";
else
   msg = "positive";

// ?: conditional operator equivalent to above if else statement
msg = number < 0 ? "negative" : "positive";
```

Also, conditional operator is right-associative. Hence, the expression a ? b : c ? d : e is evaluated as a ? b : (c ? d : e), not as (a ? b : c) ? d : e.

Q15. What are differences between Constant, ReadOnly and Static?

Ans. **Constant -** Constant fields or local variables must be assigned a value at the time of declaration and after that, they cannot be modified. By default constant is static, hence you cannot define a constant type as static.

```
public const int X = 10;
```

A const field is a compile-time constant. A constant field or local variable can be initialized with a constant expression which must be fully evaluated at compile time.

```
void Calculate(int Z)
{
   const int X = 10, X1 = 50;
   const int Y = X + X1; //no error, since its evaluated a compile time
   const int Y1 = X + Z; //gives error, since its evaluated at run time
}
```

You can apply const keyword to built-in value types (byte, short, int, long, char, float, double, decimal, bool), enum, a string literal, or a reference type which can be assigned with a value null.

```
const MyClass obj1 = null;//no error, since its evaluated a compile time
const MyClass obj2 = new MyClass();//gives error, since its evaluated at run time
```

Constants can be marked as public, private, protected, internal, or protected internal access modifiers.

Use the const modifier when you sure that the value a field or local variable would not be changed.

Readonly - A readonly field can be initialized either at the time of declaration or within the constructor of the same class. Therefore, readonly fields can be used for run-time constants.

```
class MyClass
{
    readonly int X = 10; // initialized at the time of declaration
    readonly int X1;

    public MyClass(int x1)
    {
      X1 = x1; // initialized at run time
    }
}
```

Explicitly, you can specify a readonly field as static since like constant by default it is not static. The readonly keyword can be applied to a value type and reference type (which initialized by using the new keyword) both. Also, delegate and event could not be readonly.

Use the readonly modifier when you want to make a field constant at runtime.

Static - The static keyword is used to specify a static member, which means that static members are common to all the objects and they do not tie to a specific object.

This keyword can be used with classes, fields, methods, properties, operators, events, and constructors, but it cannot be used with indexers, destructors, or types other than classes.

```
class MyClass
{
    static int X = 10;
    int Y = 20;
    public static void Show()
    {
        Console.WriteLine(X);
        Console.WriteLine(Y); //error, since you can access only static members
    }
}
```

<u>**Key points about static keyword:**</u>

1. If the static keyword is applied to a class, all the members of the class must be static.

2. Static methods can only access static members of the same class. Static properties are used to get or set the value of static fields of a class.
3. A static constructor can't be parameterized and public. A static constructor is always a private default constructor which is used to initialize static fields of the class.

Q16. What is a safe and unsafe code?

Ans. A C# code which runs under the management of CLR is called safe code. C# has an interesting feature which is unsafe code. The unsafe code does not run under the management of CLR. Sometimes, you need to access low-level functionality such as Win32 API calls or need to use the pointer in C#, and you take responsibility for ensuring such code operates correctly. Such code must be placed inside unsafe blocks in your source code.

Using *unsafe* keyword of C#, you can mark any of the following as unsafe:

- An entire method.
- A code block in braces.
- An individual statement.

For Example:

```
class TestUnsafe
{
    unsafe static void PointerMethod()
    {
        int i = 5;

        int* pt = &i;
        System.Console.WriteLine("*pt = " + *pt);
        System.Console.WriteLine("Address of pt = {0:X2}\n", (int)pt);
    }

    static void StillPointer()
    {
        int i = 7;

        unsafe
        {
            int* pt = &i;
            System.Console.WriteLine("*pt = " + *pt);
            System.Console.WriteLine("Address of pt = {0:X2}\n", (int)pt);
        }
    }

    static void Main()
    {
        PointerMethod();
        StillPointer();
    }
}
```

Q17. What is Type Casting or Type Conversions?

Ans. Type Casting is a mechanism to convert one type of value to another type. Typecasting is possible only when both data types are compatible with each other; otherwise, you will get an *InvalidCastException*.

Q18. What are different types of Type Casting supported in C#?

Ans. The following types of Typecasting are supported in C#:

1. **Implicit conversion –** Implicit conversion is being done automatically by the compiler and no data will be lost. It includes conversion of a smaller data type to a larger data types and conversion of derived classes to base class. This is safe type conversion.

```
int smallnum = 654667;
// Implicit conversion
long bigNum = smallnum;
```

```
class Base
{
    public int num1 { get; set; }
}

class Derived: Base
{
    public int num2 { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Derived d = new Derived();
        //Implicit Conversion
        Base b = d;
    }
}
```

2. **Explicit conversion -** Explicit conversion is being done by using a cast operator. It includes conversion of the larger data type to smaller data type and conversion of the base class to derived classes. In this conversion information might be lost or conversion might not be succeeding for some reasons. This is un-safe type conversion.

```
long bigNum = 654667;
// Explicit conversion
int smallnum = (int)bigNum;
```

```
class Base
{
    public int num1 { get; set; }
}

class Derived: Base
```

```
{
    public int num2 { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Base b = new Base();
         //Explicit Conversion
         Derived d = (Derived)b;
     }
}
```

3. **User-defined conversion -** User-defined conversion is performed by using special methods that you can define to enable explicit and implicit conversions. It includes conversion of class to a struct or basic data type and struct to class or basic data type. Also, all conversions methods must be declared as ***static***.

```
class RationalNumber
{
    int numerator;
    int denominator;

    public RationalNumber(int num, int den)
    {
        numerator = num;
        denominator = den;
    }

    public static implicit operator RationalNumber(int i)
    {
        // Rational Number equivalant of an int type has 1 as denominator
        RationalNumber rationalnum = new RationalNumber(i, 1);
        return rationalnum;
    }

    public static explicit operator float(RationalNumber r)
    {
        float result = ((float)r.numerator) / r.denominator;
        return result;
    }
}
class Program
{
    static void Main(string[] args)
    {
        // Implicit Conversion from int to rational number
        RationalNumber rational1 = 23;

        //Explicit Conversion from rational number to float
        RationalNumber rational2 = new RationalNumber(3, 2);
        float d = (float)rational2;
    }
}
```

Q19. How to do safe type casting in C#?
Or
What are the differences between IS and AS operators in C#?

Ans. Type Casting is the mechanism to convert one data type to another. And so, while explicit type casting of one data type to another, we get an exception if the previous one data type is not compatible with the new data type. To avoid this exception, we have IS and AS operator in C# for safe typecasting.

- **IS Operator -** The IS operator checks whether the type of a given object is compatible with the new object type. It returns Boolean type value: true if the given object is compatible with new one, else false. In this way IS operator help you to do safe typecasting.

```
class Employee
{
    public string Name { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        // Creates a new Object obj
        Object obj = new Object();
        // checking compatibility of obj object with other type

        // b1 is set to true.
        Boolean b1 = (obj is Object);
        // The cast fails: no exception is thrown, but b2 is set to false.
        Boolean b2 = (obj is Employee);

        //we can also use like this
        if (obj is Employee)
        {
            Employee emp = (Employee)obj;
            // TO DO:
        }
    }
}
```

In the above example, CLR is checking the type of **obj** object twice. First time within **IF Condition** and if it is true, then a second time within **IF Block**. Actually, this way affects the performance since each and every time CLR will walk the inheritance hierarchy, checking each base type against the specified type (Employee). To avoid this, we have AS operator.

- **AS Operator -** The AS operator also checks whether the type of a given object is compatible with the new object type. It returns non-null if the given object is compatible with new one, else null. In this way AS operator help you to do safe typecasting. The above code can be re-written by using AS operator in a better way.

```
class Employee
{
    public string Name { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        // Creates a new Object obj
        Object obj = new Object();
        // checking compatibility of obj object with other type

        // The cast fails: no exception is thrown, but emp is set to null.
        Employee emp = obj as Employee;
        if (emp != null)
        {
            // TO:DO
        }
    }
}
```

In the above example, CLR checks the obj object type only one time; if it is matched it returns non-null means Employee type otherwise returns null. Hence AS operator provide good performance over IS operator.

Note - AS operator used only for reference conversions, nullable conversions, and boxing conversions. This operator cannot perform other conversions like as user-defined conversions.

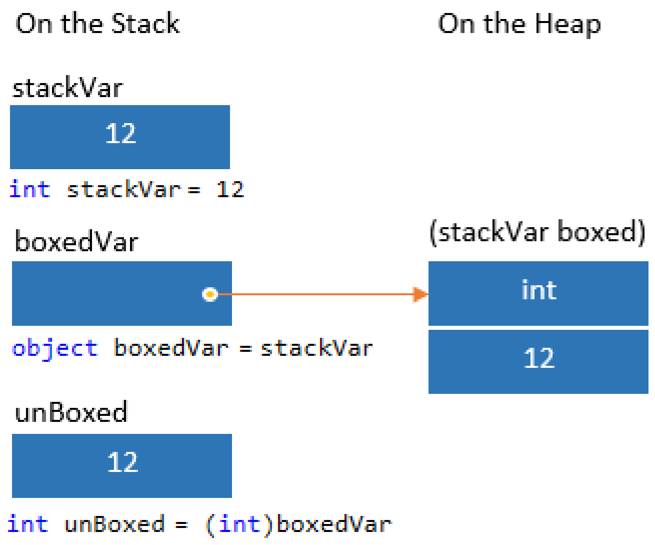
Q20. What are Upcasting and Downcasting?

Ans. Implicit conversion of derived classes to a base class is called **Upcasting** and explicit conversion of the base class to derived classes is called **Downcasting**.

```
class Base
{
    public int num1 { get; set; }
}

class Derived: Base
{
    public int num2 { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Derived d1 = new Derived();
        //Upcasting
        Base b1 = d1;

        Base b2 = new Base();
         //Downcasting
        Derived d2 = (Derived)b2;

    }
}
```

Q21. What are Boxing and Unboxing?

Ans. **Boxing -** Implicit conversion of a value type (int, char etc.) to a reference type (object) is known as boxing. In the boxing process, a value type is being allocated on the heap rather than the stack.



Boxing and unboxing

Unboxing - Explicit conversion of same reference type (which is being created by boxing process), back to a value type is known as unboxing. In the unboxing process, the boxed value type is unboxed from the heap and assigned to a value type which is being allocated on the stack.

For Example:

```
// int (value type) is created on the Stack
int stackVar = 12;

// Boxing = int is created on the Heap (reference type)
object boxedVar = stackVar;

// Unboxing = boxed int is unboxed from the heap and assigned to an int stack variable
int unBoxed = (int)boxedVar;
```

Real life example:

```
int i = 10;
ArrayList arrlst = new ArrayList();

//ArrayList contains object type value
//So, int i is being created on heap
arrlst.Add(i); // Boxing occurs automatically
```

```
int j = (int)arrlst[0]; // Unboxing occurs
```

Note:

- Sometimes boxing is necessary, but you should avoid it if possible since it will slow down the performance and increase memory requirements.
  For example, when a value type is boxed, a new reference type is created and the value is copied from the value type to the newly created reference type. This process takes time and required extra memory (around twice the memory of the original value type).
- Attempting to unbox null causes a *NullReferenceException*.

```
int? stackVar = null;
// Boxing= Integer is created on the Heap
object boxedVar = stackVar;

// NullReferenceException
int unBoxed = (int)boxedVar; //Object reference not set to an instance of an object.
```

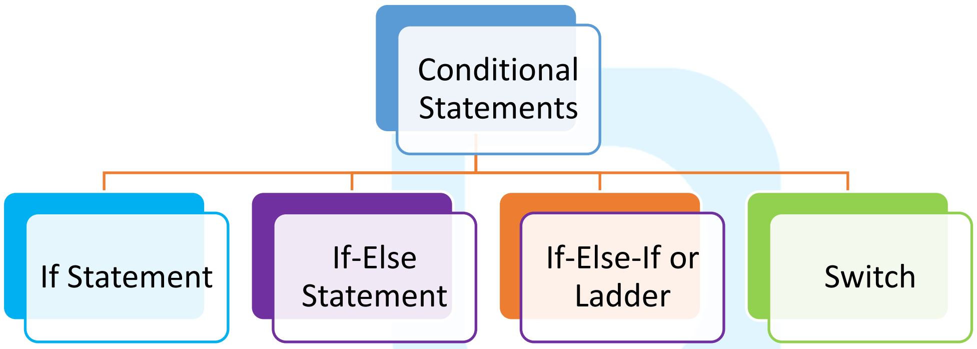
- Attempting to unbox a reference to an incompatible value type causes an *InvalidCastException.*

```
int stackVar = 12;
// Boxing= Integer is created on the Heap
object boxedVar = stackVar;

// InvalidCastException
float unBoxed = (float)boxedVar; //Specified cast is not valid.
```

3
Statements

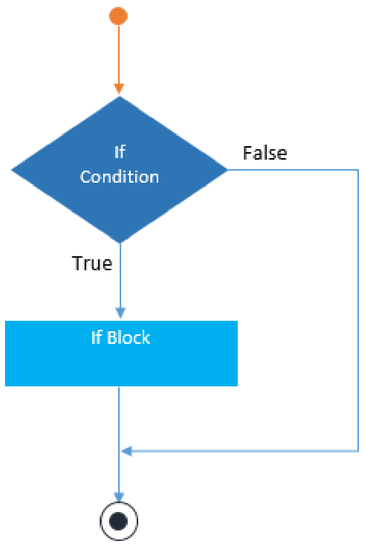
Q1. What are different types of decision-making statements in C#?

Ans. Decision-making statements help you to make a decision based on certain conditions. These conditions are specified by a set of decision-making statements having Boolean expressions which are evaluated to a Boolean value true or false. There are following types of decision-making statements in C#.



Q2. What is if statement in C#?

Ans. An if statement consists of a Boolean expression which is evaluated to a Boolean value. If the value is true, then if block is executed otherwise next statement(s), would be executed.



If statement

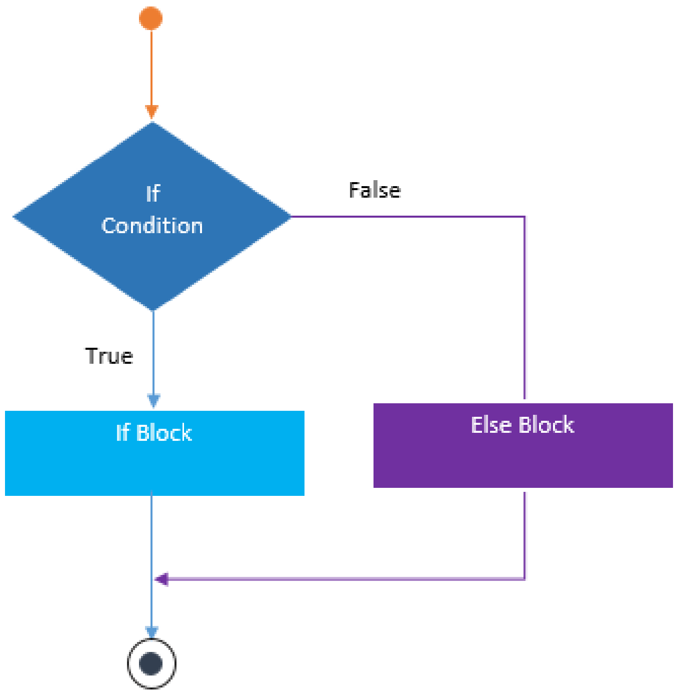
You can have multiple if statement as shown below-

```
public class Example
{
    static void Main()
    {
        int a = 5, b = 2;
        int result = a / b;

        if (result == 2)
        {
            Console.WriteLine("Result is 2");
        }
        if (result == 3)
        {
            Console.WriteLine("Result is 3");
        }
    }
}
/* Output
  Result is 2
 */
```

You can also do nesting of if statement means an if statement inside another if that is called nested if statement.

Q3. What is an if-else statement in C#?

Ans. An if-else statement consists of two statements – if statement and else statement. When the if statement expression is evaluated to true, then if block will be executed otherwise the else block will be executed.

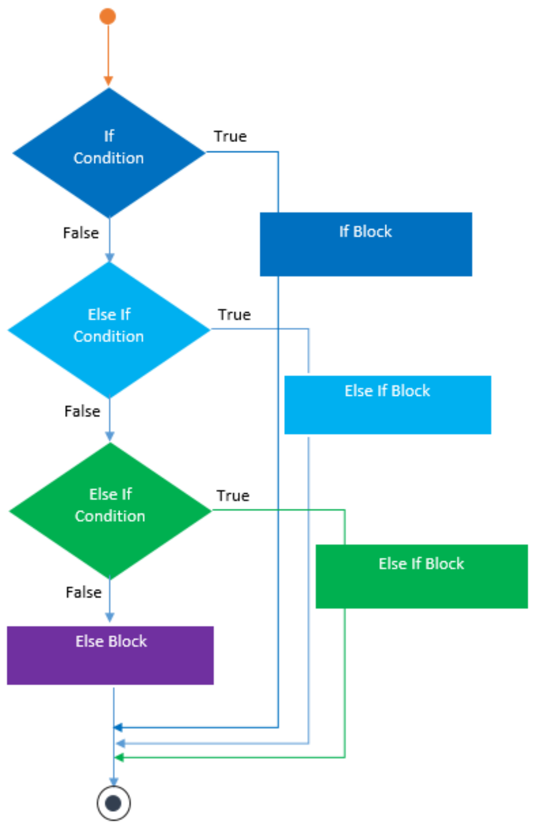


If-Else statement

```
public class Example
{
    static void Main()
    {
        int a = 5, b = 6;
        int result = a - b;
        if (result > 0)
        {
            Console.WriteLine("Result is greater than zero");
        }
        else
        {
            Console.WriteLine("Result is smaller than or equal to zero");
        }
    }
}
/* Output
Result is smaller than or equal to zero
 */
```

Q4. What is If-Else-If statement or ladder in C#?

Ans. The If-Else-If ladder is a set of statements that are used to test a series of conditions. If the first if statement met the result, the code within the if block executes. If not, control passes to the else statement, which contains a second "if" statement. If the second one met the result, code within the if block executes. This continues as a series of else if statements. A default else code block executes when no condition has been evaluated to true.



If-Else-If Ladder

If-Else-If ladder must contain more specific case at the top and generalize case at the bottom.

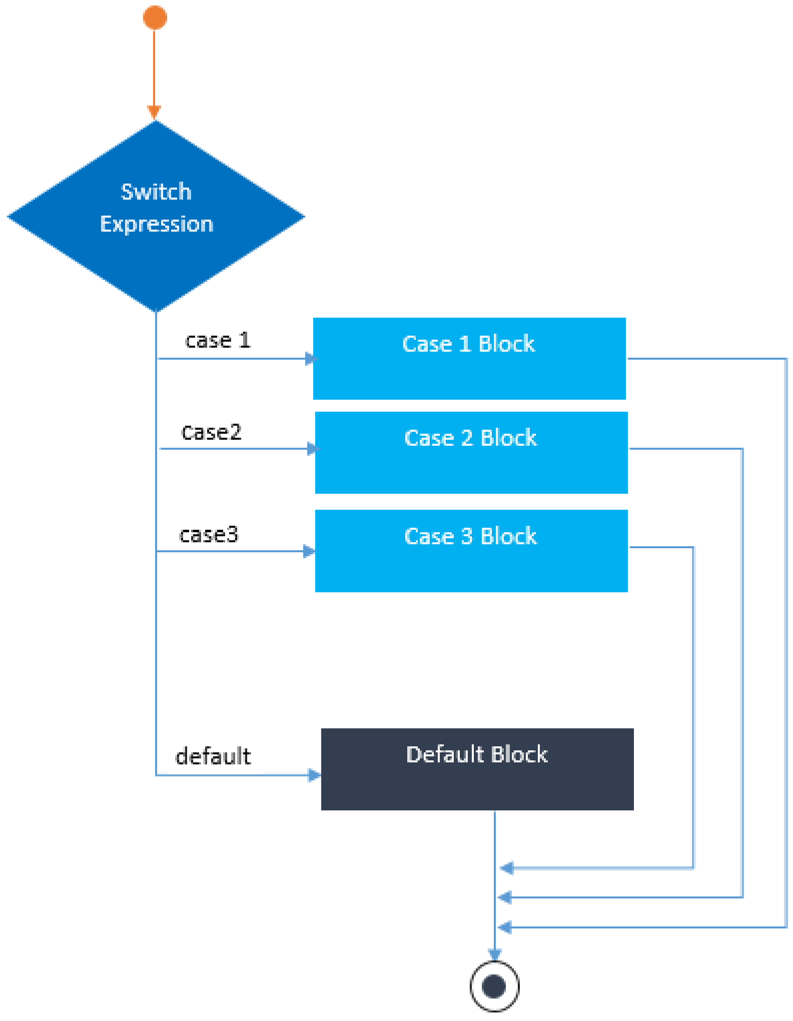
```
public class Example
{
    static void Main(string[] args)
    {
        char grade = 'B';

        if (grade == 'A')
        {
            Console.WriteLine("Excellent!");
        }
        else if (grade == 'B')
        {
            Console.WriteLine("Well done");
        }
        else if (grade == 'D')
        {
            Console.WriteLine("You passed");
        }
        else if (grade == 'F')
        {
            Console.WriteLine("Better try again");
        }
        else
        {
            Console.WriteLine("You Failed!");
        }
    }
}
/* Output
 Well done
*/
```

Q5. What is switch statement or ladder in C#?

Ans. The switch statement is an alternative for a long **If-Else-If** ladder that is used to test a series of conditions. A switch statement contains one or more case labels which are tested against the switch expression.

When one case matches the value with the result of switch expression, the control continues executing the code from that label. When no case label contains a matching value, control is transferred to the default section, if it exists. When there is no default case, no action is taken and control is transferred outside to the switch statement.

```
public class Example
{
    static void Main(string[] args)
    {
        char grade = 'B';

        switch (grade)
        {
            case 'A':
                Console.WriteLine("Excellent!");
                break;
            case 'B':
            case 'C':
                Console.WriteLine("Well done");
                break;
            case 'D':
                Console.WriteLine("You passed");
                break;
            case 'F':
                Console.WriteLine("Better try again");
                break;
            default:
                Console.WriteLine("You Failed!");
                break;
        }
    }
}
/* Output
 Well done
*/
```



Switch Statement

The key point about the Switch statement

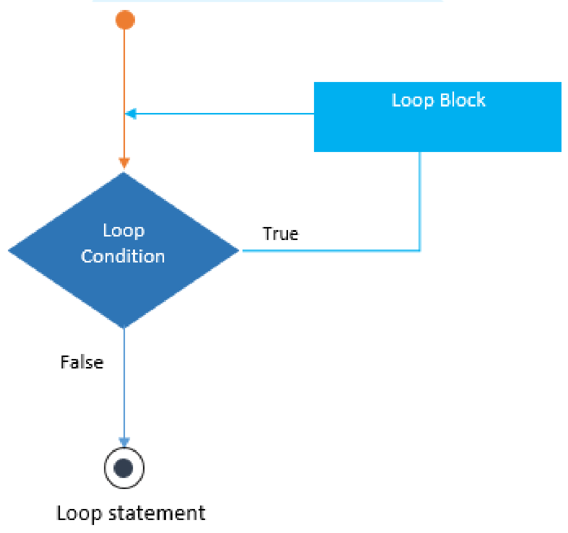
- Each case label specifies a constant value.
- A switch statement can have multiple switch sections, and each section can have one or more case labels.
- In C#, each switch section must be separated by a break or other jump statements such as goto, return and throw. Unlike C and C++, you cannot skip the jump statements.
- Unlike If-Else-If ladder, it is not mandatory to put the more specific case at the top and generalize case at the bottom since all the switch cases have equal precedence.

Q6. Which one is fast, switch or if-else-if ladder?

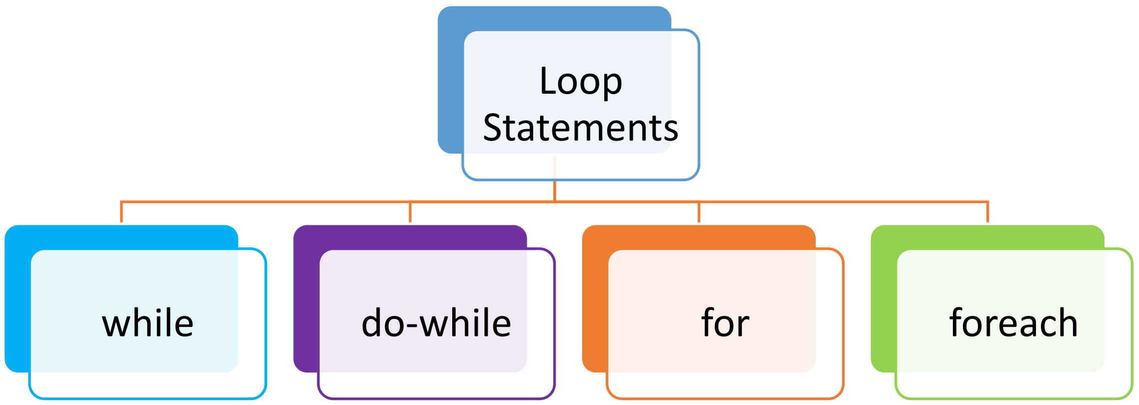
Ans. Typically, a switch statement is faster than an equivalent if-else-if statement. This happens because of the compiler's ability to optimise the switch statement. In case of an if-else-if ladder, the compiler needs to check each and every if condition in order to execute. While in case of a switch statement compiler does not need to check earlier cases, the compiler is able to find out the matching case which leads the fast execution.

Q7. What is a loop in C#?

Ans. A loops statement is used to repeat a block of statements for a certain time.



C# offers four types of loops– while, do-while, for and foreach.



Q8. What is while loop in C#?

Ans. This loop executes a statement or a block of statements until the specified expression evaluates to false. Typically, while loop is useful when you are not aware of an exact number of iterations.

```
Public class Example
{
    static void Main(string[] args)
    {
        int number = 5;
        while (number != 0) //specified boolean expression
        {
            Console.WriteLine("Current value of n is {0}", number);
            number--; //update statement
        }
    }
}
/* Output:
   Current value of n is 5
   Current value of n is 4
   Current value of n is 3
   Current value of n is 2
   Current value of n is 1
*/
```

Q9. What is do..while loop in C#?

Ans. This loop is similar to while loop except that it evaluates the expression after the loop has been executed once. Typically, a do-while loop is useful when you are sure that your loop block would be executed at least one time and further execution depends upon the specified expression evaluation.

```
public class Example
{
    static void Main(string[] args)
    {
        char choice = 'Y';
        do
        {
            int number = 5;

            Console.Write("\nPrint Number : {0}", number);
            Console.Write("\nDo you want to continue (y/n) :");
            choice = Convert.ToChar(Console.ReadLine());

        } while (choice == 'y'); //specified boolean expression
    }
}
/* Output:
   Print Number : 5
   Do you want to continue (Y/N) :y

   Print Number : 5
   Do you want to continue (Y/N) :n
*/
```

Q10. What is for loop in C#?

Ans. This loop has three sections - index declaration, condition (Boolean expression) and updation section. In each for loop iteration, the index is updated (incremented/decremented) by updation section and checked with the condition. If the condition is matched, it continues execution until the specified Boolean expression evaluates to false.

Typically, for loop is useful when you are aware of an exact number of iterations. Normally, it is used for iterating over arrays and for sequential processing.

```
public class Example
{
    static void Main(string[] args)
    {
        int[] arr = new int[5] { 5, 10, 12, 20, 30 };

        for (int index = 0; index < arr.Length; index++)
        {
            Console.WriteLine("Number is : {0}", arr[index]);
        }
    }
}
/* Output:
   Number is :5
   Number is :10
   Number is :12
   Number is :20
   Number is :30
*/
```

Q11. What is foreach loop in C#?

Ans. This loop operates on collection in .NET which implements *IEnumerable* and returns each element in order. For example, Array, ArrayList and other .NET collections.

Typically, *Foreach* is used in place of for loop where you need to retrieve the elements from the source collection, but it cannot be used to add or remove items from the source collection. For adding or removing items from the source collection, you should use *for* loop.

```
public class Example
{
    static void Main(string[] args)
    {
        int[] arr = new int[5] { 5, 10, 12, 20, 30 };

        foreach (var item in arr)
        {
            Console.WriteLine("Number is : {0}", item);
        }
    }
}
/* Output:
   Number is :5
   Number is :10
```

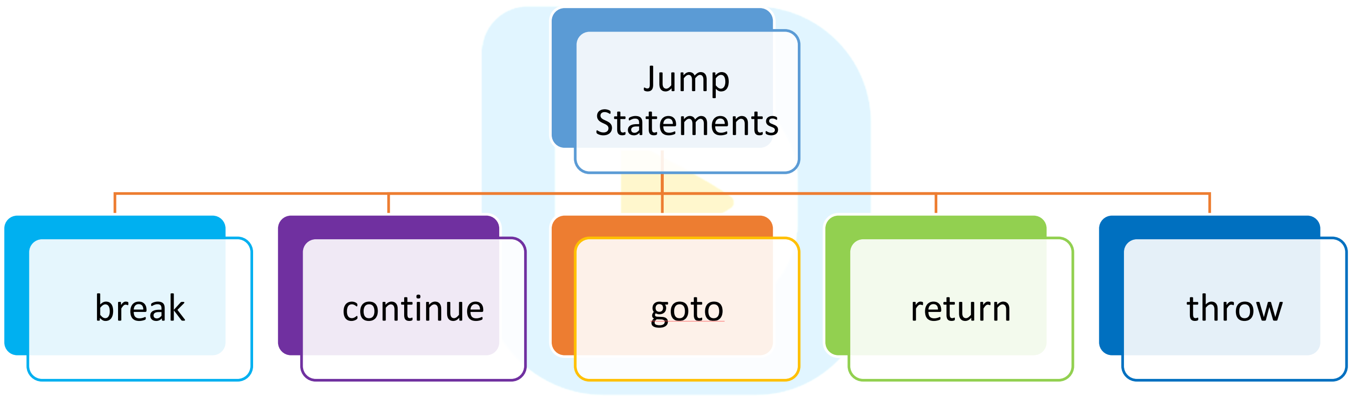
```
    Number is :12
    Number is :20
    Number is :30
*/
```

Above foreach loop statement is equivalent to the following while loop statement –

```
int[] arr = new int[5] { 5, 10, 12, 20, 30 };

var item = arr.GetEnumerator();
while (item.MoveNext())
{
   Console.WriteLine("Number is : {0}", item.Current);
}
```

Q12. What are jump statements?

Ans. The jump statements are used to transfer program control from one point in the program to another point, at any time. C# supports following types of jumps statements.



Q13. What is the break statement?

Ans. This statement terminates the execution of loop or switch in which it appears and transfers program control to the next statement which is placed immediately after the loop or switch.

```
public class Example
{
    static void Main(string[] args)
    {
        for (int i = 1; i <= 10; i++)
        {
            if (i == 5)
            {
                break;
            }
            Console.WriteLine(i);
        }
        Console.WriteLine("Next statement placed after loop");
    }
}
/* Output:
   1
   2
   3
   4
   Next statement placed after loop
*/
```

This statement is also used to terminates an inner nested loop and return control to the outer loop.

Q14. What is the goto statement?

Ans. This statement transfers program control to a labelled statement. The label statement must exist in the scope of the goto statement. More than one goto statement can transfer control to the same label. This statement can be used to get out from a loop or an inner nested loop to outer loop.

```
public class Example
{
    static void Main(string[] args)
    {
        for (int i = 1; i <= 10; i++)
        {
            if (i == 5)
            {
                goto Exitlabel;
            }
            Console.WriteLine(i);
        }
        Console.WriteLine("Next statement placed after loop");

        Exitlabel: //goto label
        Console.WriteLine("Labeled statement");
    }
}
/* Output:
   1
   2
   3
   4
   Labeled statement
*/
```

Note: Unlike the break statement, it does not transfer the program control to the next statement which is placed immediately after the loop or switch.

This statement is not recommended since it makes the program logic complex and difficult to understand. It also becomes difficult to trace the control flow of program execution.

Q15. What is the continue statement?

Ans. This statement skips the current iteration and passes the program control to the next iteration of the enclosing loop in which it appears.

```
public class Example
{
    static void Main(string[] args)
    {
        for (int i = 1; i <= 10; i++)
        {
            if (i <= 5)
            {
                continue;
            }
            Console.WriteLine(i);
        }
        Console.WriteLine("Next statement placed after loop");
    }
}
/* Output:
   6
   7
   8
   9
   10
   Next statement placed after loop
*/
```

Unlike break statement, it does not terminate the loop execution but it skips the current iteration of the loop and passes program control to the next iteration of the enclosing loop.

Q16. What is the return statement?

Ans. This statement terminates the execution of the method in which it appears and returns control to the calling method.

```
public class Example
{
    static void Main(string[] args)
    {
        double length = 5.0, width = 2.5;

        double result = CalculateArea(length, width);
        Console.WriteLine("The area is {0:0.00}", result);
    }
    public static double CalculateArea(double length, double width)
    {
        double area = length * width;
        return area;
    }
}
/* Output:
  The area is 12.50
*/
```

If the return statement appears in a try block and the finally block is also existing, then the finally block will be executed before control returns to the calling method.

Q17. What is throw statement?

Ans. This statement throws an exception which indicates that an error has occurred during the program execution. This statement is used with a combination of try-catch or try-finally statements.

```
public class Example
{
    static void Main(string[] args)
    {
        try
        {
            int number = 5, x = 0;

            double result = number / x;
            Console.WriteLine("Result is {0}", result);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Exception: "+ex.Message);
        }
    }
}
/* Output:
  Exception: Attempted to divide by zero.
*/
```

Q18. What are the different types of comment in C#?

Ans. There are 3 types of comments in C#.

- Single line (//)
- Multi (/* */)
- Page/XML Comments (///)

4

Array and Strings

Q1. What do you mean by an Array?

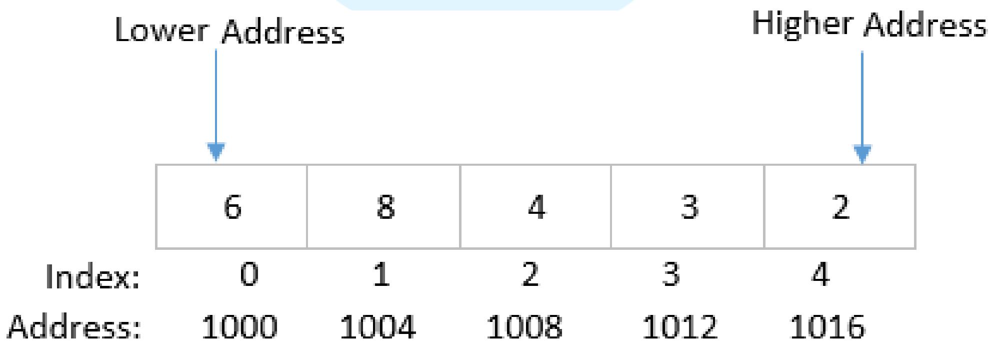
Ans. An array is a collection of homogenous or same types of elements which are accessible by a numeric index. The numeric index starts from 0 and goes to n-1, where n is the size of an array.

An array is stored in contiguous memory locations and the memory lowest address contains the first element and the highest memory address contains the last element in an array.

For example,

```
int[] arr = new int[5]; // declare arr as an int type array of size 5

arr[0] = 6;
arr[1] = 8;
arr[2] = 4;
arr[3] = 3;
arr[4] = 2;
```

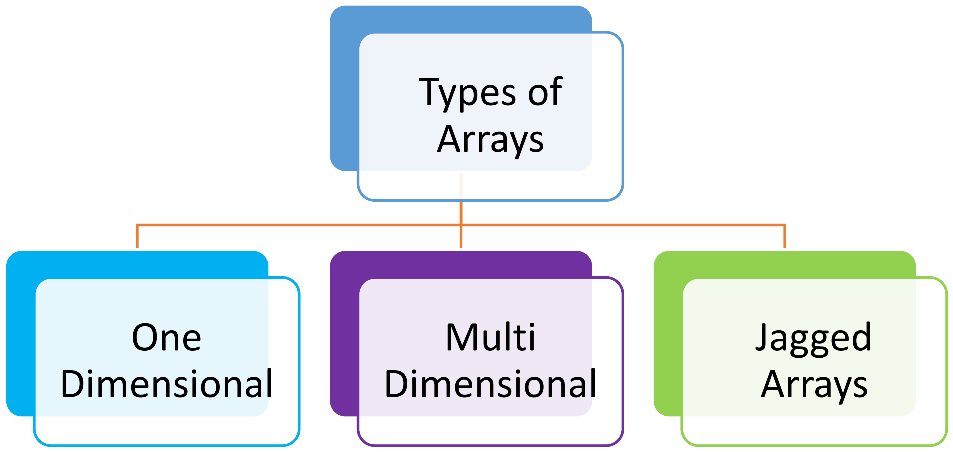
Memory allocation of the above array is given below-



Memory allocation in array

Q2. Explain different types of arrays in C#

Ans. There are three types of arrays in C# as given below-



Q3. What is Single-Dimensional Array?

Ans. This is the simplest form of an array in which elements are stored contiguously starting from 0 to n-1, where n is the size of the array. This has a single dimension.

For example,

```
int[] arr = new int[5]; // declare single dimensional array


arr[0] = 6;

arr[1] = 8;

arr[2] = 4;

arr[3] = 3;

arr[4] = 2;
```

Q4. What is Multi-Dimensional Array

Ans. This is also known as a rectangular array which has more than one dimension. 2D array, 3D array, 4D array are an example of a multi-dimensional array.

```
int[,] arr2D = new int[3, 3]; // declare a 2D array

arr2D[0, 0] = 1;
arr2D[0, 1] = 2;
arr2D[0, 2] = 3;

arr2D[1, 1] = 4;
arr2D[1, 2] = 5;
arr2D[1, 3] = 6;

arr2D[2, 1] = 7;
```

```
arr2D[2, 2] = 8;
arr2D[2, 3] = 9;

int[, ,] arr3D = new int[3, 3, 3]; // declare a 3D array

arr3D[0, 0, 0] = 1;
arr3D[0, 0, 1] = 2;
arr3D[0, 0, 2] = 3;

arr3D[0, 1, 0] = 1;
arr3D[0, 1, 1] = 2;
arr3D[0, 1, 2] = 3;

arr3D[0, 2, 0] = 1;
arr3D[0, 2, 1] = 2;
arr3D[0, 2, 2] = 3;

arr3D[1, 0, 0] = 1;
arr3D[1, 0, 1] = 2;
arr3D[1, 0, 2] = 3;

arr3D[1, 1, 0] = 1;
arr3D[1, 1, 1] = 2;
arr3D[1, 1, 2] = 3;

arr3D[1, 2, 0] = 1;
arr3D[1, 2, 1] = 2;
arr3D[1, 2, 2] = 3;

arr3D[2, 0, 0] = 1;
arr3D[2, 0, 1] = 2;
arr3D[2, 0, 2] = 3;

arr3D[2, 1, 0] = 1;
arr3D[2, 1, 1] = 2;
arr3D[2, 1, 2] = 3;

arr3D[2, 2, 0] = 1;
arr3D[2, 2, 1] = 2;
arr3D[2, 2, 2] = 3;
```

Q5. What is Jagged Array?

Ans. A jagged array is a special type of array whose elements are arrays itself of different dimensions and sizes. A jagged array is also called an "array of arrays." Also, a jagged array's elements must be initialized before its use.

```
int[][] jaggedArray = new int[3][]; //declare a jagged array

//initializing jagged array elements
jaggedArray[0] = new int[3]; //contains 3 elements
jaggedArray[1] = new int[2]; //contains 2 elements
jaggedArray[2] = new int[4]; //contains 4 elements
```

In the jagged array, it is necessary to specify the value in the first bracket [] since it specifies the jagged array size.

You can also mix jagged array and multi-dimensional array to declare an array as given below-

```
int[][,] mixedArray = new int[3][,] {
                                        new int[,] { {1,3}, {5,7} },
                                        new int[,] { {0,2}, {4,6}, {8,10} },
                                        new int[,] { {11,22}, {99,88}, {0,9} }
                                       };
```

Q6. What is the Array class in C#?

Ans. The Array is an abstract class, defined in System namespace. It acts as a base class for all arrays in C#. Array class provides functionality for creating, manipulating, searching, and sorting arrays in .NET Framework.

It provides various useful methods such as *CreateInstance, Copy, CopyTo, GetValue*, and *SetValue, Sort* and others properties such as *LongLength* and *GetLongLength* for manipulating an array.

```
int[] arr = new int[] { 4, 12, 6, 1 }; //declare a unorder array
Array.Sort(arr); //sorting unorder array arr
```

Q7. What is string array in C#?

Ans. A string array is an array which is used to store string type values. It can be single-dimensional, multi-dimensional and jagged array.

```
string[] strarr = new string[3];
strarr[0] = "Hi";
strarr[0] = "Hello";
strarr[0] = "Array";
```

Q8. What is a string?

Ans. A string is a collection of Unicode characters. The string type is an alias for *System.String* class in the .NET Framework. It has the following characteristics-

1. It is a reference type.
2. It can contain a null value.
3. It is immutable means you cannot change the contents of a string.

For example,

```
string str = "Hello World";
str = str.Replace("World", "Dot Net Tricks");
```

In the above example, you are not changing the original string (*Hello World*) contents which *str* contains. But it's just setting the value of *str* to a new string which is a copy of the old string "World" is replaced by "Dot Net Tricks". The new string has been allocated to a new memory address that is different from old string address.

The String class provides many methods for creating, manipulating, and comparing strings. You should not use the new operator to create a string object except when initializing the string with an array of chars as given below-

```
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters); //initializing string from a char array
```

Q9. What are different types of strings?

Ans. There are two types of strings as given below-

1. **Mutable string:** Mutable strings are instances of *System.Text.StringBuilder* class. You can modify the contents of a mutable string i.e. whenever you perform any text manipulation on a mutable string, the original string is modified. It is extremely useful when you need to perform a lot of text manipulation on the string.

2. **Immutable string -** Immutable strings are instances of *System.String* class. You cannot modify the contents of a mutable string i.e. whenever you perform any text manipulation on an immutable string, a new string object is returned and original string remains same. It is extremely useful with regard to trust since a string would not change as long as you don't change your reference.

Example of Mutable and Immutable string:

```
using System;
using System.Text;

class Example
{
    static void Main(string[] args)
    {
        //original string is: Hello World
        string immutableStr = "Hello World";

        immutableStr.Replace("World", "Dot Net Tricks"); //modify string
        Console.WriteLine(immutableStr);    //no change here

        immutableStr = immutableStr.Replace("World", "Dot Net Tricks");//new string
        Console.WriteLine(immutableStr);    //changed string here

        //original string is: Hello World
        StringBuilder mutablestr = new StringBuilder("Hello World");

        mutablestr.Replace("World", "Dot Net Tricks");  //modify string

        Console.WriteLine(mutablestr); //changed string here
        Console.ReadLine();
    }
}

/* Output:

   Hello World
   Hello Dot Net Tricks
   Hello Dot Net Tricks
*/
```

Q10. What is the difference between System.String and System.Text.StringBuilder classes?

Ans. Differences between string and StringBuilder are given below-

System.String	System.Text.StringBuilder
Represents immutable string.	Represents mutable string.
It is unmodifiable in nature.	It is modifiable or dynamic in nature.
It is extremely useful with regard to trust since a string would not change as long as you don't change your reference.	It is extremely useful when you need to perform a lot of text manipulation on the string.
It is slow as compared to the StringBuilder object.	It is faster than string object.

Q11. What is a verbatim string?

Ans. A verbatim string is preceded by the @ character. The @ character specifies that no character interpretations should be applied on a string. A verbatim does not have escape sequences.

```
class Example
{
    static void Main(string[] args)
    {
        //non-verbatim string having escape sequences character '\' i.e. back slash
        string str = "c:\\Program Files\\Microsoft.NET";
        //verbatim string, no need of escape sequences character
        string strverbatim = @"c:\Program Files\Microsoft.NET";

        Console.WriteLine(str);
        Console.WriteLine(strverbatim);
    }
}
/* Output:
  c:\Program Files\Microsoft.NET
  c:\Program Files\Microsoft.NET
*/
```

Verbatim strings are used to specify the following types of strings:

- File path specifications string
- SQL strings
- XML strings

5

Class and Object

Q1. What do you mean by Class?

Ans. A Class is a user-defined data structure that contains data members (fields, properties, events, and indexer), member function and nested class type. Basically

- It is a user-defined data type.
- It is a reference type.
- In fact, a class is a tag or template for an object.

```
class Book
{
    //Attributes
    int BookId;
    string Title;
    string Author;
    string Subject;
    double Price;

    //Parametrized constructor to set book details
    public Book(int bookId, string title, string author, string subject, double price)
    {
        BookId = bookId;
        Title = title;
        Author = author;
        Subject = subject;
        Price = price;
    }

    //Behaviours
    public void showDetails()
    {
        Console.WriteLine("\n########## Book Details ##########");

        Console.WriteLine("BookId :{0}", BookId);
        Console.WriteLine("Title : {0}", Title);
        Console.WriteLine("Author : {0}", Author);
        Console.WriteLine("Subject : {0}", Subject);
        Console.WriteLine("Price : {0}", Price);
    }
}
```

```
public class Program
{
    public static void Main(string[] args)
    {
        //object creation and initialization
        Book objBook1 = new Book(1, "ASP.NET MVC Interview Questions & Answers",
"Shailendra Chauhan", ".NET ", 500);
        Book objBook2 = new Book(2, "Node.js Interview Questions & Answers ", "Shailendra
Chauhan", "JavaScript", 400);

        //calling class
        objBook1.showDetails();
        objBook2.showDetails();

        Console.ReadKey();
    }
}

/* Output

########## Book Details ##########
BookId :1
Title : ASP.NET MVC Interview Questions & Answers
Author : Shailendra Chauhan
Subject : .NET
Price : 500

########## Book Details ##########
BookId :2
Title : Node.js Interview Questions & Answers
Author : Shailendra Chauhan
Subject : JavaScript
Price : 400
*/
```

Q2. What do you mean by Object?

Ans. An object is a representative of the class and is responsible for memory allocation of its data members and member functions. An object is a real-world entity having attributes (data members) and behaviours (member functions).

For example, a student object can have attributes name, address and contact number. It performs activities attending class, giving exam etc.

Q3. What is Constructor?

Ans. A constructor is a special type of function/method which has the same name as its class. The constructor is invoked whenever an object of a class is created.

Key points

- It is a special function/method because its name is same as the class.
- It can be overloaded the same as normal methods.

- It is automatically invoked as soon as the object of the class is created.
- It does not return a value, not even a void type.

Q4. What are different types of constructors?

Ans. There are three types of the constructor – default constructor, parameterized constructor and copy constructor. But C# does not support copy constructor.

1. **Default Constructor -** The default constructor has no parameter. When a class has no constructor, the default constructor is served by the compiler to that class.
2. **Parameterized Constructor -** The parameterized constructor has one or more arguments and used to assign values to instance variables of the class.

```
class Example
{
    double num1, num2;

    #region Constructor Overloading

    //Default Constructor
    public Example()
    {
        Console.WriteLine("This is a default constructor!");
    }

    //Parameterized Constructor
    public Example(int a, int b)
    {
        num1 = a;
        num2 = b;
        Console.WriteLine("This is a paremeterized constructor!");
    }

    //Parameterized Constructor
    public Example(double a, double b)
    {
        num1 = a;
        num2 = b;
        Console.WriteLine("This is a paremeterized constructor!");
    }
    #endregion

    public void ShowResult()
    {
        double result = num1 + num2;
        Console.WriteLine("Result: {0}", result);
    }
}

class Program
{
    static void Main()
    {
        Example obj1 = new Example(); //default constructor called
        Example obj2 = new Example(4, 5); //parameterized constructor called
        Example obj3 = new Example(4.2, 5.2); //parameterized constructor called

        obj2.ShowResult();
        obj3.ShowResult();
        Console.Read();
    }
}

/* Output
This is a default constructor!
This is a parameterized constructor!
This is a parameterized constructor!
Result: 9
Result: 9.4
*/
```

3. **Copy Constructor –** This constructor is used to copy the entire values of an object to another object. But C# does not support copy constructor.

Q5. What is a static Constructor?

Ans. A static constructor is a special type of constructor that gets called before the first object of the class is created. Typically, it is useful to initialize the static fields or to perform a specific action that needs to be performed only once.

Key points

- Only one static constructor is allowed and it must be a default constructor having no access modifier i.e. private access. The class may have others non-static parameterized constructors.
- It can only access the static member.
- It is used to write the code that needs to be executed only once.
- A static constructor cannot be called directly.

```
class Example
{
    static int Count;
    double num1, num2;

    static Example()//static Constructor
    {
        Count++;
        Console.WriteLine("This is a static constructor!");
    }

    public Example(int a, int b)    //Parameterized Constructor

    {
        num1 = a;
        num2 = b;
        Console.WriteLine("This is a paremeterized constructor!");
    }

    public Example(double a, double b)    //Parameterized Constructor
    {
        num1 = a;
        num2 = b;
        Console.WriteLine("This is a paremeterized constructor!");
    }

    public void ShowResult()
    {
        double result = num1 + num2 + Count;
        Console.WriteLine("Result: {0}", result);
    }
}

class Program
{
    static void Main()
    {
        Example obj2 = new Example(4, 5); //parameterized constructor called
        Example obj3 = new Example(4.2, 5.2); //parameterized constructor called

        obj2.ShowResult();
        obj3.ShowResult();
        Console.Read();
    }
}

/* Output
This is a static constructor!
This is a parameterized constructor!
This is a parameterized constructor!
Result: 10
Result: 10.4
*/
```

Q6. Why static Constructor has no parameter?

Ans. Because it is going to be called by CLR and not by object then who will pass the parameter?

Q7. Why you can have only one static Constructor in a class?

Ans. To define multiple constructors for a class, you need to overload the constructors. It means you need to define parameterized constructors that accept parameters from outside but the static constructor is called by the CLR and CLR cannot pass parameters to the parameterized constructor. So, only one static constructor is possible in a class.

Q8. What is private Constructor?

Ans. A private constructor is a special type of constructor used in class that contains only static members.

Point of Interest:

- A private constructor cannot be externally called.
- If a class that has one or more private constructor and no public constructor, then other classes except nested classes are not allowed to create an instance of this class and it cannot be inherited.
- The private constructor restricts a class to be instantiated directly from external classes.
- It also provides an implementation of singleton class pattern.

```
public class Example
{
    private Example() { }

    private static int count;

    public static int Counter
    {
        get { return ++count; }
    }
}

class Program
{
    static void Main()
    {
        // Example obj = new Example(); //Error private constructor is inaccessible

        System.Console.WriteLine("Initial count = " + Example.Counter);
        System.Console.WriteLine("New count = " + Example.Counter);
    }
}
/*Out put

 Initial count = 1
 New count = 2

*/
```

Q9. When you create child class object, which class constructor will be called first?

Ans. When you create a child class object, first its base class constructor is called then its constructor is called.

```
Public class BaseClass
{
    public BaseClass()
    {
        Console.WriteLine("Base class constructor!");
    }

}

public class ChildClass : BaseClass
```

```
{
    public ChildClass()
    {
        Console.WriteLine("Child class constructor!");
    }
}

class ExecutionClass
{
    static void Main()
    {
        ChildClass objChild = new ChildClass();
        Console.ReadKey();
    }
}


/* Out put

Base class constructor!
Child class constructor!

*/
```

Q10. What is Destructor?

Ans. The destructor is a special type member function/method which has the same name as its class name preceded by a tilde (~) sign. The destructor is used to release unmanaged resources allocated by the object. It is called automatically before an object is destroyed. It cannot be called explicitly. A class can have only one destructor.

```
class Example
{
    public Example()
    {
        Console.WriteLine("Constructor called");
    }

    //destructor
    ~Example()
    {
        Console.WriteLine("Destructor called to destroy instance");
    }
}

class Program
{
    static void Main()
    {
        Example T = new Example();
        GC.Collect();
    }
}
/*Out Put
```

```
Constructor called.
Destructor called to destroy the instance

*/
```

Q11. What is finalize () method?

Ans. A *finalize* method belongs to the object class. It is used to free unmanaged resources like files, database connections, COM etc. held by an object before that object is destroyed. Internally, it is called by Garbage Collector and cannot be called by user code.

Implement it when you have unmanaged resources in your code and want to make sure that these resources are freed when the Garbage collection happens.

```
// Implementing Finalize method
class Example
{
    public Example()
    {
        Console.WriteLine("Constructor called");
    }

    //At runtime C# destructor is automatically Converted to Finalize method.
    ~Example()
    {
        //TO DO: clean up unmanaged objects
    }
}
```

Q12. What are the similarities and difference between finalize and destructor?

Ans. Finalize and destructor, both are used to release unmanaged resources allocated by the object. The differences between these two are given below-

- Finalize method corresponds to the .NET framework and it is a part of *System.Object* class whereas destructor is the C# implementation of the finalize method.
- In C#, destructors are converted to finalize method when the program is compiled.

Q13. What is the use of "using" statement in C#?

Ans. The using statement obtains the specified resources, uses it and then automatically calls the dispose method to clean up the specified resources when the execution of statement is completed.

```
using System;
using System.Data.SqlClient;

public class Program
{
    public static void Main(string[] args)
    {
        string connectionString = "Persist Security Info=False;Integrated
Security=SSPI;Initial Catalog=MyDatabase;Data Source=(local);";
```

```
        // using statement acquires the SqlConnection as a resource
        using (SqlConnection con = new SqlConnection(connectionString))
        {
            //using SqlConnection
            con.Open();

            SqlCommand command = new SqlCommand("SELECT TOP 2 * FROM Employee", con);
            SqlDataReader reader = command.ExecuteReader();
            while (reader.Read())
            {
                Console.WriteLine("Name: {0}, Address: {1}, Mobile: {2}",
                reader.GetInt32(0), reader.GetString(1), reader.GetString(2));
            }
        }//SqlConnection is automatically closed and released; as using statement is
finished
     }
}

/* Output

 Name: Mohan, Address: Noida, Mobile: 9500000000
 Name: Rohan, Address: Delhi, Mobile: 9400000000

*/
```

When you would not use the using statement, you need to write code to clean up the specified resources for releasing occupied memory as given below-

```
using System;
using System.Data.SqlClient;

public class Program
{
    public static void Main(string[] args)
    {
        string connectionString = "Persist Security Info=False;Integrated
Security=SSPI;Initial Catalog=MyDatabase;Data Source=(local);";
        SqlConnection con = new SqlConnection(connectionString);
        try
        {
            con.Open();

            SqlCommand command = new SqlCommand("SELECT TOP 2 * FROM Employee", con);
            SqlDataReader reader = command.ExecuteReader();
            while (reader.Read())
            {
                Console.WriteLine("Name: {0}, Address: {1}, Mobile: {2}",
                reader.GetInt32(0), reader.GetString(1), reader.GetString(2));
            }
        }
        finally
        {
            con.Close(); // Closing connection
```

```
                con.Dispose();// Releasing resource
            }
        }
}

/* Output
 Name: Mohan, Address: Noida, Mobile: 9500000000
 Name: Rohan, Address: Delhi, Mobile: 9400000000
*/
```

Q14. What is Access Modifier?

Ans. An access modifier is used to specify the accessibility of a class member. Access modifiers help to support abstraction and encapsulation by exposing only necessary members and hiding others.

Q15. Explain different types of access modifiers in C#?

Ans. C# supports four types of access modifiers as given below-

1. **Private –** A private member can only be accessed inside the same class or struct. By default, a class or a struct members have private access. Also, private members are not accessible outside the containing class or struct.

2. **Protected -** A protected member can only be accessed inside the same class or in a class that is derived from that class. A class member can be declared as protected but a struct members cannot be declared as protected since struct does not support inheritance.

3. **Public -** A public member can be accessed inside the same assembly or any other assembly that references it. A class or a struct members can be declared as public. By default, interface members are public and you cannot define any access modifier to interface members.

4. **Internal -** An internal member can be accessed inside the same assembly, but not within any other assembly. A class or a struct members can be declared as internal. By default, class, struct, interface, delegates itself have internal access when declared inside a namespace. A class, struct, interface, delegates itself have private access when nested.

5. **Protected Internal -** A protected member can be accessed inside the same assembly, or inside a derived class in another assembly. A class member can be declared as protected internal but a struct members cannot be declared as protected internal because structs do not support inheritance.

Q16. Can a child or derived type have higher accessibility form its parent type?

Ans. No, it cannot. You cannot have a public class B that has been derived from an internal class A. It would be a compile time error.

Q17. Can you declare struct members as protected?

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

Q18. Can you explain the member's Accessibility?

Ans. Access modifiers accessibly can be understood through the following table:

Modifiers	Same assembly			Different assembly	
	Same class	*Derived class*	*Other class*	*Derived class*	*Other Class*
Private	Yes	No	No	No	No
Protected	Yes	Yes	No	Yes	No
Internal	Yes	Yes	Yes	No	No
Protected Internal	Yes	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes	Yes

Q19. Can destructors have access modifiers?

Ans. No, destructors cannot have access modifiers since these are automatically called by the compiler.

Q20. While inheriting a class do the private members get inherited?

Ans. Yes, private members are also inherited but cannot be accessed directly by the derived class.

Q21. What is Enum?

Ans. In C#, an enum is a value type. An enum contains a list of named constants which are known as enumerators. By default, an enum has internal access but it can be declared as public.

By default, the first enumerator value starts from 0 and the successive enumerators values increased by 1.

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

In the above *Days* enum, Sat is 0, Sun is 1, Mon is 2, and so on. You can also change the default values for enumerators, as shown in the following example.

```
enum Days { Sat=2, Sun, Mon, Tue, Wed, Thu, Fri };
```

Now, the first enumerator value will start from 2 and each successive enumerator value will increase by 1.

Q22. When to use enum?

Ans. An enum can be used in the following two cases:

1. To define static constants.
2. To define constant flags.

Q23. What is a structure in C#?

Ans. In C#, a structure is value type which contains data members and members function like class. The structure can contain fields, methods, constants, constructors, properties, indexers, operators and other nested structure types.

```
struct Book
{
    int BookId;
    string Title;
    string Author;
    string Subject;
    double Price;

    public void getDetails(int bookId, string title, string author, string subject,
double price)
    {
        BookId = bookId;
        Title = title;
        Author = author;
        Subject = subject;
        Price = price;
    }
    public void showDetails()
    {
        Console.WriteLine("\n########## Book Details ##########");

        Console.WriteLine("BookId :{0}", BookId);
        Console.WriteLine("Title : {0}", Title);
        Console.WriteLine("Author : {0}", Author);
        Console.WriteLine("Subject : {0}", Subject);
        Console.WriteLine("Price : {0}", Price);
    }
};

public class Program
{
    public static void Main(string[] args)
    {

        Book Book1 = new Book();
        Book Book2 = new Book();

        Book1.getDetails(1, "Node.js Interview Questions and Answers", "Shailendra
Chauhan", "JavaScript", 500);
        Book2.getDetails(2, "ASP.NET MVC Interview Questions and Answers", "Shailendra
Chauhan", ".NET", 400);

        Book1.showDetails();
        Book2.showDetails();

        Console.ReadKey();
    }
}

/* Output

########## Book Details ##########
BookId :1
Title : Node.js Interview Questions and Answers
Author : Shailendra Chauhan
```

```
Subject : .NET
Price : 500

########## Book Details ##########
BookId :2
Title : ASP.NET MVC Interview Questions and Answers
Author : Shailendra Chauhan
Subject : .NET
Price : 400
*/
```

Features of structure:

- Like a class, a struct can have methods, fields, indexers, properties, operator methods, events and nested structure.
- A struct can have only default constructors.
- A struct cannot have destructors. Since it is a value type and there is no need to call GC.
- Unlike a class, a struct does not support inheritance but it can implement one or more interfaces.
- By default, a struct is sealed; hence cannot be inherited by another type.
- A struct cannot have abstract, virtual or protected access modifiers. Since these access modifiers are used with the members of a base type but a *struct* cannot work as a base type.
- Unlike a class, a struct can be instantiated without using the *new* operator. In this case, struct fields will remain unassigned and the struct instance cannot be used until all the fields are initialized.

Q24. When to use structure?

Ans. Structures are useful for small data structures and provide better performance as compared to class.

Q25. What are the similarities between a class and a structure?

Ans. Following are the similarities between a class and a structure.

- Both can have constructors, methods, properties, fields, constants and enum,
- Both can implement the interface.
- Both can have delegates and events.

Q26. What is the difference between class and structure?

Ans. Differences between these two are given below-

Class	Structure
The class is a reference type. Hence stores on the heap.	The structure is a value type. Hence stores on the stack.
The class can have default and parameterized constructor	The structure can have only default constructor.
The class can have a destructor	The structure cannot have destructor since it is a value type and there is no need to call GC.

Supports inheritance	Does not support inheritance since by default structure is sealed.
Classes are slow as compared to the structure	Structures are fast as compared to classes.
The class can be declared as static	The structure cannot be declared as static.
The class can have abstract, virtual or protected members.	The structure cannot have abstract, virtual or protected members. Since it does not act as the base type.

Q27. What is a static class?

Ans. A static class is a special class which is loaded into memory automatically by the CLR at the time of code execution. Hence, there is no need to create an object of the static class.

Key points about static class

- It contains only static members.
- It cannot be instantiated.
- It is sealed, hence cannot be inherited but It can inherit object class.
- It can have only one static constructor (default constructor) which has private access.

```
static class Example
{
    //public Example();  Error, static classes cannot have instance constructors.
    //int a = 10; Error, static class cannot have non static instance members
    static Example()
    {
        Console.WriteLine("Hi, I am static constructor!");
    }

    public static void MyMethod()
    {
        Console.WriteLine("Hi, I am static method, called directly by class name!");
    }
}
class Program
{
   static void Main()
   {
      //Error, cannot create instance of static class.
      //Example obj = new Example();

      //Calling class members
      Example.MyMethod();
    }
}

/*Output

  Hi, I am static constructor!
  Hi, I am a static method called directly by class name!

*/
```

Q28. When to use a static class?

Ans. A static class is useful when you want to provide some common utilities like various configuration settings, driver manipulation etc. to your application.

Q29. What are static members?

Ans. Static class members (method, field, property, or event) are declared using the static keyword. These can be called only with the class name. In the memory, only one copy of static member exists and which is shared by all the instances of a class.

```
class Example
{
    static int count;
    public Example()
    {
        count++;
    }
    public static void MyMethod()
    {
        Console.WriteLine("Count ={0}", count);
        Console.WriteLine("Hi, I am static method, called directly by class name!");
    }

}

class Program
{
    static void Main()
    {
        Example obj1 = new Example();
        //obj1.MyMethod(); //error, MyMethod cannot be accessed by class instance, use
type name instead

        Example.MyMethod();
        Console.Read();
    }
}

/*Output

  Count =2
  Hi, I am a static method, called directly by class name!
*/
```

Q30. What is a Static Method?

Ans. You can declare a class member as static. Such methods are directly called by the class name and not by the object of that class. For example, see the above example.

Q31. What are the differences between static and instance methods?

Ans. **Static Method:** A static method is declared using a static modifier and can have access to the static members only. It is called by the class name and you cannot use *this* keyword with it.

Instance Method: Other methods in a class are instance methods. An instance method is called using the class object and can access both static and non-static members. It cannot be called directly by its class name and you can use *this* keyword with it.

Q32. What are sealed Classes in C#?

Ans. Sealed classes are special types of class that is being restricted to be inherited. Sealed modifier is used to prevent a class inheritance. If you try to inherit a sealed class then a compile-time error occurs.

```
Sealed class Example
{
    void Display()
    {
        Console.WriteLine("Sealed class method!");
    }
}
class MyExample : Example //Error, 'MyExample' cannot derive from sealed type 'Example'
{
    void Show()
    {
        Console.WriteLine("Show method!");
    }
}
```

Q33. What is the base class in .NET Framework from which all the classes are derived?

Ans. System.Object.

Q34. What is a partial class, interface or struct?

Ans. Partial class, interface and structure were introduced in C# 2.0. Now, it is possible to split the definition of a class, interface and structure over more than one source files. These source files must be defined in the same namespace or assembly and must have the partial keyword and same access modifiers.

```
// Partial Class
partial class Example
{
    void Test()
    { //write your code
    }
}
partial class Example
{
    void Test2()
    { //write your code
    }
}

// Partial Interface
partial interface IExample
{
    void ITest();
}
partial interface IExample
{
    void ITest2();
}

// Partial Structure
partial struct SExample
{
    void STest()
    {
        //write your code
    }
}
partial struct SExample
{
    void STest2()
    {
        //write your code
    }
}
```

Key points about partial Class, Interface or struct

1. During code compilation, all the files should be available to form the final class, interface or struct.
2. Any member declared in the one part/file will be available to all other parts.
3. If any part has Inheritance, then it applies to the entire class.
4. Different parts of a class or struct may inherit from different interfaces.
5. If any part is declared abstract, then the whole class, interface or struct is considered abstract.
6. If any part is declared sealed, then the whole class, interface or struct is considered sealed.

Q35. What are the advantages of partial?

Ans. There are following advantages of partial Class, Interface or structure:

1. Allow more than one developer to work simultaneously on the same class, struct or interface.
2. Partial class are particularly helpful for customizing auto-generated code by the IDE. Whenever the IDE generate the code then the tool may define some partial class, interface, methods and further customization of a partial class, the interface is done by the developers without messing with the system generated code.

Q36. What is the partial method?

Ans. A partial method is a special method that exists within a partial class or struct. One part of the partial class or struct have only partial method declaration means signature and another part of the same partial class or struct may have an implementation for that partial method. If the implementation is not provided for the declared partial method, the method and all calls to those partial methods will be removed at compile time.

Key points about the partial method

1. Partial methods can be declared or defined within the partial class or struct.

2. Partial methods are implicitly private and declarations must have the partial keyword.
3. Partial methods must return void.
4. Partial methods implementation is optional.
5. Partial methods can be static and unsafe and generic.
6. Partial methods can have ref parameters but not out parameters since these can't return value.
7. You cannot make a delegate to a partial method.
8. The signatures of the partial method will be the same in both parts of the partial class or struct.

Q37. When to use partial methods?

Ans. Partial methods are particularly helpful for customizing auto-generated code by the tool. Whenever the tool generates the code then the tool may declare some partial method and implementation of these methods is decided by the developers.

If you are using entity framework for making DAL then you have seen that the Visual Studio make a partial method OnContextCreated() as shown below. Now, the implementation of it depends on you whether you want to use it or not.

```
public partial class DALEntities : ObjectContext
{
    #region Constructors
    // Constructors for DALentities
    #endregion
    #region Partial Methods
    partial void OnContextCreated();
    #endregion
}
// This part can be put in the separate file
public partial class DALEntities : ObjectContext
{
    partial void OnContextCreated()
    {
        // put method implementation code
        Debug.WriteLine("OnContextCreated partial method");
    }
}
```

6
Exception Handling

Q1. What is Error?

Ans. Errors refer to the mistake or faults which occur during program development or execution. If you don't find them and correct them, they cause a program to produce wrong results.

Q2. Explain different types of errors?

Ans. In programming, language errors can be divided into three categories as given below-

1. **Syntax Errors -** Syntax errors occur during development when you make type mistake in the code. For example, instead of writing *while,* you write WHILE then, it will be a syntax error since C# is a case-sensitive language.

```
bool flag=true;

WHILE (flag) //syntax error, since c# is case sensitive
{
   //TO DO:
}
```

2. **Runtime Errors (Exceptions) -** Syntax errors occur during execution of the program. These are also called exceptions. This can be caused due to improper user inputs, improper system errors.

```
int a = 5, b = 0;
int result = a / b; // DivideByZeroException
```

Exceptions can be handled by using try-catch blocks.

3. **Logical Errors -** Logic error occurs when the program is written fine but it does not produce the desired result. Logic errors are difficult to find because you need to know for sure that the result is wrong

```
int a = 5, b = 6;
double avg = a + b / 2.0; // logical error, it should be (a + b) / 2.0
```

Q3. What are Exceptions?

Ans. Exceptions are defined as glitches, unexpected or unseen errors which occur during the execution of a program. These can be caused due to improper user inputs, improper design logic or system errors. In this case, if an application does not provide a mechanism to handle these exceptions then your application may crash.

Q4. What are different types of Exceptions?

Ans. Exceptions can be of following two types -

1. **Application Exception –** Such exceptions are generated while a program execution.
2. **System Exception –** Such exceptions are generated by the CLR.

Q5. What is the role of System.Exception class?

Ans. System.Exception class is provided by .NET Framework to handle any type of exception that occurs. The Exception class is the base class for all other exception classes provided by .NET Framework. The exception class instance has some important properties as given below:

Property	**Usage**
InnerException	Provides exception instance that caused the current exception.
Message	Provides detailed information about the error
StackTrace	Provides the function stack to show where the exception is thrown
Source	Provides the name of the application or the object that causes the error.
Targetsite	Shows which method throws the current exception

Q6. What are the various predefined exceptions classes?

Ans. In C#, exceptions are represented by various exceptions classes. Directly or indirectly, these exception classes are derived from the System.Exception class. There are some common predefined exceptions classes:

Exception Class	**Description**
System.ArithmeticException	A base class for handling exceptions that occur during arithmetic operations. System.DivideByZeroException and System.OverflowException classes inherit this class.
System.DivideByZeroException	Thrown when an attempt to divide a numeric value by zero.
System.IndexOutOfRangeException	Thrown when an attempt to access an array element via an index that is less than zero or outside the bounds of the array.
System.InvalidCastException	Thrown when an explicit conversion from a base type to a derived type fails at runtime.
System.NullReferenceException	Thrown when the referenced object is null but it is required.
System.OutOfMemoryException	Thrown when an attempt to allocate memory (via new) fails.
System.StackOverflowException	Thrown when the execution stack is exhausted by having too many pending method calls; typically, a very deep or recursion method calls.

Q7. What is Exception Handling?

Or

What are the ways in C# to handle exception?

Or
What is try, catch and finally block?

Ans. Exception handling is a mechanism to capture run-time errors and handle them correctly. It is achieved by using Try-Catch-Finally blocks and throw keyword.

Try Block - The *try* block encloses the statements that might throw an exception.

```
try
{
    // Statements that can cause exception.
}
```

Catch Block - The *catch* block handles any exception if one exists.

```
catch(ExceptionType e)
{
    // Statements to handle exception.
}
```

Finally Block - The *finally* block can be used for doing any clean-up process like releasing unused resources even if an exception is thrown. For example, Disposing database connection.

```
finally
{
   // Statement to clean up.
}
```

Throw Keyword - The *throw* keyword is used to throw an exception explicitly.

```
catch (Exception e)
{
   throw (e);
}
```

Q8. What is the order of try-catch-finally blocks?

Ans. In the order of exceptions handling blocks try block comes first, after that catch block(s) come and in the last finally block comes.

```
try
{
   // statements that can cause exception.
}
catch (MoreSpecificExceptionType e1)
{
   // error handling code
}
catch (SpecificExceptionType e2)
{
   // error handling code
}
catch (GeneralExceptionType eN)
{
   // error handling code
}
finally
{
   // statement to clean up.
}
```

You can also skip finally block if required.

```
try
{
   // statements that can cause exception.
}
catch (MoreSpecificExceptionType e1)
{
   // error handling code
}
catch (SpecificExceptionType e2)
{
   // error handling code
}
catch (GeneralExceptionType eN)
{
   // error handling code
}
```

You can also skip catch block(s) if required.

```
try
{
   // statements that can cause exception.
}
finally
{
   // statement to clean up.
}
```

Hence, a combination of try-catch-finally or try-catch or try-finally blocks is valid.

Q9. Can you have catch block without try block?

Ans. No, you cannot have catch blocks without try block since catch blocks are used to handle exceptions occur in the try block. Also, there may be multiple catch block corresponds to a try block.

Q10. Can you have a try block without catch block?

Ans. Yes, you can have a try block without catch blocks but you need to define one **finally** block corresponds to that try block. Also, there is only one finally block.

Q11. Can multiple catch blocks be executed?

Ans. However, you can write multiple catch blocks in our code. But multiple catch blocks cannot be executed. Only one catch block is executed which matched the occurred exceptions and then the control is transferred to the finally block if exist or to the next statement after try-catch blocks.

Q12. What are user-defined Exceptions in C#?

Ans. C# allows us to create a user-defined exception class that should be derived from the Exception class. In this user-defined exception class, you can write your own logic for handling exceptions. These exceptions are called user-defined exceptions.

```
 class UserDefinedException : Exception
{
    public UserDefinedException(string str)
    {
        Console.WriteLine(str);
    }
}
class Program
{
    public static void Main()
    {
        try
        {
            throw new UserDefinedException("User defined exception!");
        }
        catch (Exception e)
        {
            Console.WriteLine("Catching Exception: " + e.Message);
        }

        Console.WriteLine("Next statement to be executed");
        Console.Read();
    }
}

/* Output

 User-defined exception!
 Catching Exception: Exception of type 'UserDefinedException' was thrown.
 Next statement to be executed

*/
```

Q13. What do you understand by Custom Exceptions?

Ans. Custom exceptions are nothing but user-defined exceptions. The only difference is that here exceptions are handled as per user requirements.

7

Reflection and Serialization

Q1. What is Reflection?

Ans. Reflection is used to examine objects and their types. .NET Framework provides System.Type object, which allows you to access all the information about objects and their types. The System.Reflection namespace contains classes which allow you to obtain information about assemblies, modules and types.

Q2. How to use Reflection in C#?

Ans. In C#, you should use the typeof operator to get the object's type or use the GetType() method to get the type of the current instance.

```
class Program
{
    public static void Main()
    {
        // Create object
        Example obj = new Example();
        // Get the Type information.
        Type typeObj = obj.GetType();

        // Get Method Information.
        MethodInfo methodInfo = typeObj.GetMethod("Sum");
        object[] objParam = new object[] { 5, 10 };

        // Get and display the Invoke method.
        Console.WriteLine("Object assembly info: " + typeObj.Assembly);
        Console.WriteLine("Object method '" + methodInfo.Name + "' returns " +
methodInfo.Invoke(obj, objParam));
    }
}
/*Output
 Object assembly info: ConsoleApp, Version=1.0.0.0, Culture=neutral,PublicKeyToken=null
 Object method 'Sum' returns 15
*/
```

Q3. When to use Reflection?

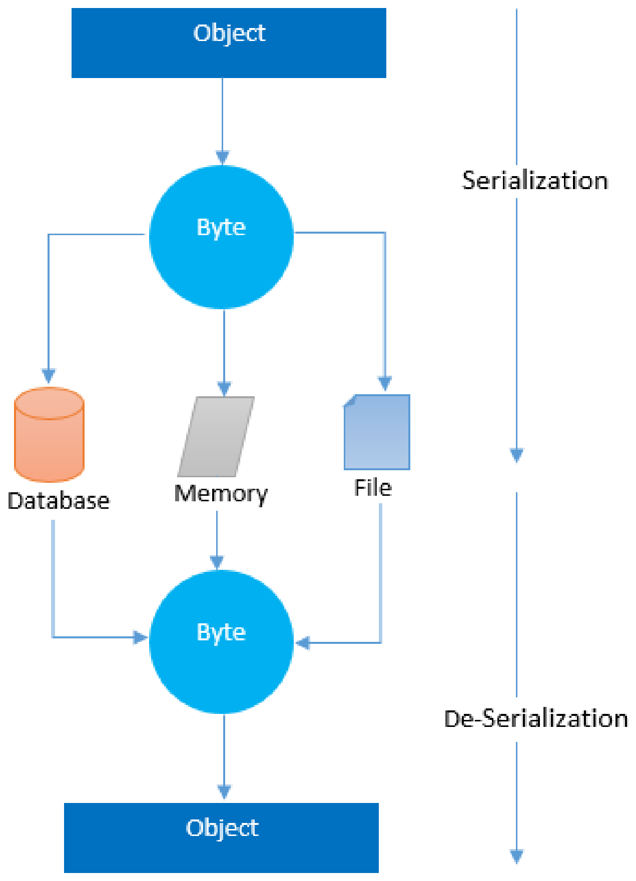
Ans. A reflection can be used in the following cases:

- To create an instance of a type or bind the type to an existing object dynamically.
- To get the type from an existing object and invoke its methods or access its fields and properties

- Need to perform late binding, accessing methods on types created at runtime.

Q4. What are Serialization and Deserialization?

Ans. Serialization is the process of converting an object into a stream of bytes. So that, you can store that object into memory, database or a file in form of XML or JSON or binary format. Its main purpose is to save the state of an object so that it can be recreated when it requires. This reverse process is called deserialization.



Q5. When to use serialization and deserialization?

Ans. Serialization and deserialization can be used in the following cases:

- Need to pass an object from an application to another remote application.
- Need to pass an object from one domain to another.
- Need to pass an object through a firewall as JSON/XML string.
- Need to maintain security or user-specific information across applications.

Q6. What namespace is used to support Serialization in .NET?

Ans. In .NET, serialization is provided by the System.Runtime.Serialization namespace. This namespace contains an interface IFormatter which provides *Serialize()* and *Deserialize()* methods in order to implement serialization.

Q7. What are Serializable and NonSerialized attributes?

Ans. In .NET Framework, *SerializableAttribute* attribute to serialize all the members of a class. If you do not want to serialize a member, decorate that member with *NonSerializedAttribute.* Let's understand these things with the help of the following examples:

```
Using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

// An Employee object that needs to be serialized.
[Serializable()]
public class Employee
{
    public int EmpId;
    public string Name;

    // A field that is not serialized.
    [NonSerialized()]
    public int Salary;
}
```

Q8. What are different types of Serialization?

Ans. There are three types of serialization supported by .NET Framework as given below-

1. **XML Serialization -** This serialization serializes an object into an XML stream. This is best for cross-platform transfer since XML is supported by every platform. This generates a human-readable and editable data file.
2. **Binary Serialization -** This serialization uses binary encoding. It is faster and secure than XML serialization but it is not easily portable to another platform except .NET.
3. **SOAP Serialization -** This serialization serializes the object into XML stream which follows the SOAP specification. A SOAP is an XML based protocol to transport procedure calls using XML.

Q9. What namespace is used for SOAP Serialization?

Ans. In .NET Framework, SOAP serialization is provided by System.Runtime.Serialization.Formatters.Soap namespace.

Q10. What is Custom Serialization?

Ans. You can also, write your own custom class for serialization and deserialization process. The custom class must be marked as *SerializableAttribute* and implement the *ISerializable* interface.

8 Property and Indexer

Q1. What is Property?

Ans. A Property acts as a wrapper around a field. It is used to assign and read the value from that field by using set and get accessors. The get accessor is executed when the property is read and the set accessor is executed when the property is assigned a new value. A property can be created for a public, private, protected and internal field.

Unlike fields, properties do not denote storage locations and you cannot pass a property as a ref or out parameter.

```
using System;

class Example
{
    string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}

class Program
{
    static void Main()
    {
        Example obj = new Example();
        obj.Name = "Dot Net Tricks"; // called set { }
        Console.WriteLine(obj.Name); // called get { }
    }
}
```

Q2. When to use the property?

Ans. There are following cases where you should use properties:

1. Need to validate data before assigning it to a field
2. Need to do intermediate computation on data before assigning or retrieving it to a field
3. Need to log all access for a field
4. Need to protect a field by reading and writing

Q3. What are different types of properties?

Ans. Properties can be divided into three categories read-only, write-only, and read-write properties.

1. **Read-Only Property -** A read-only property allows you to only retrieve the value of a field. To create a read-only property, you should define the ***get*** accessor.

```
class User
{
    string name;
    public string Name
    {
        get { return name; }
    }
}
```

2. **Write-Only Property -** A write-only property allows you to only change the value of a field. To create a write-only property, you should define the ***set*** accessor.

```
class User
{
    string name;
    public string Name
    {
        set { name = value; }
    }
}
```

3. **Read-Write Property -** A read-write property allows you to assign and read the value of a field. To create a read-write property, you should define the ***set*** and ***get*** accessors.

```
class User
{
    string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

Q4. What is auto-implemented property?

Ans. Auto-implemented properties were introduced with C# 3.0, which make property declaration more concise. Unlike standard property, in auto-implemented property, wrapped or backing field is automatically created by the compiler but it is not available for use by the class members.

```
public int Name { get; set; }
```

To make an auto-implemented property read-only or write-only, you need to specify both get and set accessors.

```
public int ReadOnly { get; private set; }
public int WriteOnly { private get; set; }
```

Q5. When to use auto-implemented property?

Ans. It is useful when you want to store a default value and you don't want to add any additional functionality to either of the accessors.

Q6. What is static Property?

Ans. You can also declare a property static. To make a static property you must ensure that the backing store field is also static. Typically, a static property is used to make a singleton class.

```
public class MySingleton
{
    private static MySingleton instance = new MySingleton();
    private MySingleton() { }

    public static MySingleton GetInstance
    {
        get { return instance; }
    }
}
```

Q7. What is abstract Property?

Ans. An abstract property declaration does not have any implementation for the property accessors. The accessors would be implemented into derived classes. Abstract properties can be declared with an abstract class and an interface.

```
Public abstract class Person
{
    public abstract string Name { get; set; }

}
class Student: Person
{
    private string name;

    // Override Name property
    public override string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

```
public interface IPerson
{
    string Name { get; set; }

}

class Student: IPerson
{
    private string name;

    // implement Name property
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

Q8. What is Indexer in C#?

Ans. An indexer enables a class or struct instances to be indexed in the same way as an array. Unlike, property an indexer is defined using this keyword.

Just like methods, an indexer can be overloaded and can multiple parameters with a different type. Unlike an array, an indexer cannot be passed as a ref or out parameter.

```
class Program
{
    static void Main(string[] args)
    {
        Example names = new Example();
        names[0] = "Shailendra";
        names[1] = "Aman";
        names[2] = "Pavan";
        names[3] = "Mohan";
        names[4] = "Deepak";

        for (int i = 0; i < Example.size; i++)
        {
            Console.WriteLine(names[i]);
        }

        Console.ReadKey();
    }

}

class Example
{
    static public int size = 10;
    private string[] namelist = new string[size];

    public string this[int index]
    {
        get
        {
            string tmp;

            if (index >= 0 && index <= size - 1)
            {
                tmp = namelist[index];
            }
            else
            {
                tmp = "";
            }

            return (tmp);
        }
        set
        {
            if (index >= 0 && index <= size - 1)
            {
                namelist[index] = value;
            }
        }
    }
}
/* Output
   Shailendra
   Aman
   Pavan
   Mohan
   Deepak
*/
```

Q9. Which one to choose between Property and Method?

Ans. Typically, properties represent data and methods represent actions. Properties are used like fields, but they can be used to validate data before assigning to a class field or doing any manipulation before accessing its value.

Methods are preferable to properties in the following situations:

- The operations which need to access the network resources or the file system or database operations.
- The operations return an array.
- The operation is a type conversion, such as *ToString()* method.

Q10. What are the differences between Property and Indexer?

Ans. The differences between property and indexer are given below:

Property	Indexer
It is identified by its name.	It is identified by its signature.
Can be a static or an instance member.	Must be an instance member.
Accessed through a simple name.	Accessed through an index.
A *get* accessor of a property has no parameters.	A *get* accessor of an indexer has the same parameters as it has.
A set accessor of property has an implicit value parameter.	A set accessor of an indexer has the same parameters as it has, along with the value parameter.
Supports shortened syntax using Auto-Implemented Properties.	Does not support shortened syntax.

9

Abstract Class and Interface

Q1. What is an abstract Class?

Ans. An abstract class is a special type of class which cannot be instantiated and acts as a base class for other classes. Abstract class members declared as abstract must be implemented by the derived classes.

Key points about abstract class

1. An abstract class cannot be instantiated.
2. An abstract class contains abstract members as well as non-abstract members.
3. An abstract class cannot be a sealed class because the sealed modifier prevents a class from being inherited and the abstract modifier requires a class to be inherited.
4. A non-abstract class which is derived from an abstract class must include actual implementations of all the abstract members of parent abstract class.
5. An abstract class can be inherited from a class and one or more interfaces.
6. An Abstract class can have **access modifiers like private, protected, internal with class members**. But abstract members cannot have a private access modifier.
7. An Abstract class can have instance variables (like constants and fields).
8. An abstract class can have constructor and destructor (only default and not parameterized).
9. An abstract method is implicitly a virtual method.
10. Abstract properties behave like abstract methods.
11. An abstract class cannot be inherited by structures.
12. An abstract class cannot support multiple inheritance or implementation.

Q2. How to create an abstract Class in C#?

Ans. An abstract class can be created using abstract keyword and future it can be derived by Square and Rectangle as given below:

For example,

```
abstract class ShapesClass
{
    abstract public int Area();
}
class Square : ShapesClass
{
    int side = 0;
    public Square(int n)
    {
        side = n;
    }
    // Override Area method
    public override int Area()
    {
        return side * side;
    }
}

class Rectangle: ShapesClass
{
    int length = 0, width = 0;

    public Rectangle(int length, int width)
    {
        this.length = length;
        this.width = width;
    }
    // Override Area method
    public override int Area()
    {
        return length * width;
    }
}
```

Q3. When to create an abstract class?

Ans. The purpose of an abstract class is to provide either basic or default functionality or common functionalities that multiple derived classes can share and override.

Another need of abstract is to create multiple versions of your component since versioning is not a problem with the abstract class. You can add properties or methods to an abstract class without breaking the code and all inheriting classes are automatically updated with the change.

Q4. What are common design guidelines for an abstract class?

Ans. The common design guidelines for the abstract class are given below-

1. Don't define public constructors inside the abstract class. Since an abstract class cannot be instantiated.
2. Define a protected or an internal constructor inside an abstract class. Since a protected constructor allows the base class to do its own initialization when derived classes are instantiated. An internal constructor will limit the implementation of the abstract class to the same assembly which contains that class.

Q5. What is an interface and how to use it?

Ans. An interface acts as a contract between itself and class or a structure which implements it. It means a class that implements an interface is bound to implements all its members. The interface has only the member's declaration or signature and implicitly every member of an interface is public and abstract.

For example, the most common use of interfaces is, within SOA (Service Oriented Architecture). In SOA (WCF), a service is exposed through interfaces to different clients. Typically, an interface is exposed to a group of clients which needs to use common functionalities.

```
interface IStore
{
    void Read();
    void Write();
}
interface ICompress
{
    void Compress();
    void Decompress();
}

public class Document : IStore, ICompress
{
    #region IStore

    public void Read()
    {
        Console.WriteLine("Executing Document's Read Method for IStore");
    }

    public void Write()
    {
        Console.WriteLine("Executing Document's Write Method for IStore");
    }

    #endregion  // IStore

    #region ICompress

    public void Compress()
    {
        Console.WriteLine("Executing Document's Compress Method for ICompress");
    }
    public void Decompress()
    {
        Console.WriteLine("Executing Document's Decompress Method for ICompress");
    }
    #endregion  // ICompress
}
```

Key points about the interface

1. An interface doesn't provide inheritance like a class or abstract class but it only declares members which an implementing class need to be implemented.
2. It cannot be instantiated but it can be referenced by the class object which implements it. Also, Interface reference works just like object reference and behave like as object.

```
IStore IObjStore = new Document();
ICompress IObjCompress = new Document();
```

3. It contains only properties, indexers, methods, delegates and events signature.
4. It cannot contain **constants members, constructors, destructor, instance variables, static members or nested interfaces**.
5. Members of interfaces cannot have any access modifiers not even public.
6. Implicitly, every member of an interface is public and abstract. Also, you are not allowed to specify the members of an interface public and abstract or virtual.
7. An interface can be inherited from one or more interfaces.
8. An interface can extend another interface.
9. A class or structure can implement more than one interface.
10. A class that implements an interface can mark any method of the interface as virtual and this method can be overridden by derived classes.
11. Implementing multiple interfaces by a class, sometimes result in a conflict between member signatures. You can resolve such conflicts by explicitly implementing an interface member.
12. It is a good practice to start all interface names with a capital 'I' letter.

Q6. When to use an interface?

Ans. The use cases of an interface are given below:

1. Need to provide common functionality to unrelated classes.
2. Need to group objects based on common behaviours.
3. Need to introduce polymorphic behaviour to classes since a class can implement more than one interface.
4. Need to provide a more abstract view to a model which is unchangeable.
5. Need to create loosely coupled components, easily maintainable and pluggable components because the implementation of an interface is separated from itself.

Q7. What are the disadvantages of an interface?

Ans. There are following disadvantages of interfaces:

1. The main issue with an interface is that when you add a new member to its, then you must implement those members within all of the classes which implement that interface.
2. Interfaces are slow as these required extra in-directions to find the corresponding method in the actual class.

Q8. What are common design guidelines for the interface?

Ans. The common design guidelines for an interface are given below-

1. Keep your interfaces focused on the problem you are trying to solve and keep related tasks (methods) into an interface. Interfaces that have multiple unrelated tasks tend to be very difficult to implement in a class. Split up interfaces that contain unrelated functionality.
2. Make sure your interface does not contain too many methods. Since too many methods make implementing the interface difficult as the implementing class has to implement each and every method in the interface.

3. Don't make interfaces for specific functionality. An interface should define the common functionality that can be implemented by the classes of different modules or subsystems.

Q9. How can you achieve multiple inheritances (implementation) using interfaces?

Ans. In the interface, you implement rather inherit and which is possible as shown in the following the example.

```
interface I1
{
    void Show();
}
interface I2
{
    void Hide();
}
class MyClass : I1, I2
{
    // implementing interface method
    void Show()
    {
        Console.WriteLine("Show() from I1 is implemented");
    }
    void Hide()
    {
        Console.WriteLine("Hide() from I2 is implemented");
    }
    public static void Main()
    {
        System.Console.WriteLine("Multiple interfaces implement Example\n");

        MyClass obj = new MyClass();

        I1 i1 = (I1)obj;
        i1.Show();

        I2 i2 = (I2)obj;
        i2.Hide();

        Console.ReadKey();
    }
}

/*
 Output

Multiple interfaces implement Example

Show() from I1 is implemented
Hide() from I2 is implemented

 */
```

Q10. What is the difference between an interface and an abstract class?

Ans. The differences between an interface and an abstract class are given below-

Interface	**Abstract Class**
The interface has only the member's declaration or signature and implicitly every member of an interface is public and abstract. It contains only properties, indexers, methods, delegates and events signature	An abstract class contains at least one abstract member and others, non-abstract members. Like a simple class, an abstract class can have data members and member functions.
It cannot contains constants members, constructors, destructors, instance variables, static members or nested interfaces.	It can contains constants members, constructors, destructor, instance variables, static members or nested class.
It cannot have field member.	It can have field member.
It can have only public member.	It can have public, protected and private member.
Methods have only declaration.	It can have non abstract method with body definition.
It provides full abstraction.	It provides partial abstraction.
Static and Instance constants are possible.	Only static constants are possible.
It supports multiple inheritances and so a class can inherit more than one interface.	It does not support multiple inheritances and so a class can inherit only one abstract class.
Use an interface when you need to provide common functionality to unrelated classes and to create loosely coupled components, easily maintainable and pluggable components because the implementation of an interface is separated from itself.	Use an abstract class when you need to create multiple versions of your component and to provide default behaviours as well as common behaviours that multiple derived classes can share and override.

10
Delegates and Events

Q1. What are Delegates?

Ans. A delegate is a reference type that holds the reference to a class method. Any method which has the same signature as a delegate can be assigned to delegate. It is very similar to the function pointer but with a difference that delegates are type-safe. We can say that it is the object-oriented implementation of function pointers.

Key points about delegates

- Delegates are like C++ function pointers but they are type safe.
- Delegates allow methods to be passed as parameters.
- Delegates are used in event handling for defining callback methods.
- Delegates can be chained together i.e. these allow defining a set of methods that can be executed as a single unit.
- Once a delegate is created, the method it is associated will never change because delegates are immutable in nature.
- Delegates provide a way to execute methods at runtime.
- All delegates are implicitly derived from System.MulticastDelegate, class which is inheriting from System.Delegate class.
- Delegate types are incompatible with each other, even if their signatures are the same. These are considered equal if they have the reference of the same method.

Q2. What are the steps to create a Delegate?

Ans. There are three steps for defining and using delegates:

1. **Declaration -** A delegate is declared by using the keyword delegate, otherwise it resembles a method declaration.
2. **Instantiation** - To create a delegate instance, we need to assign a method (which has the same signature as a delegate) to delegate.
3. **Invocation -** Invoking a delegate is like as invoking a regular method.

```
//1. Declaration
public delegate int MyDelagate(int a, int b); //delegates having the same signature as a
method

public class Example
```

```
{
   // methods to be assigned and called by the delegate
    public int Sum(int a, int b)
    {
        return a + b;
    }

    public int Difference(int a, int b)
    {
        return a - b;
    }
 }
    class Program
    {
        static void Main()
        {
            Example obj = new Example();

            // 2. Instantiation : As a single cast delegate
            MyDelagate sum = new MyDelagate(obj.Sum);
            MyDelagate diff = new MyDelagate(obj.Difference);

            // 3.Invocation
            Console.WriteLine("Sum of two integer is = " + sum(10, 20));
            Console.WriteLine("Difference of two integer is = " + diff(20, 10));
        }
    }

/* Output
 Sum of two integer is = 30
 Difference of two integer is = 10
*/
```

Q3. What are the different types of delegates?

Ans. There are two types of delegates as given below-

1. **Single cast delegate -** A single cast delegate holds the reference of the only a single method. In the previous example, the created delegate is a single cast delegate.
2. **Multicast delegate -** A delegate which holds the reference of more than one method is called the multicast delegate. A multicast delegate only contains the reference of methods which return type is void. The + and += operators are used to combine delegate instances. Multicast delegates are considered equal if they reference the same methods in the same order.

```
//1. Declaration
public delegate void MyDelagate(int a, int b);
public class Example
{
   // methods to be assigned and called by the delegate
    public void Sum(int a, int b)
    {
        Console.WriteLine("Sum of integers is = " + (a + b));
    }
    public void Difference(int a, int b)
    {
        Console.WriteLine("Difference of integer is = " + (a - b));
    }
}
class Program
{
    static void Main()
    {
        Example obj = new Example();
        // 2. Instantiation
        MyDelagate multicastdel = new MyDelagate(obj.Sum);
        multicastdel += new MyDelagate(obj.Difference);

        // 3. Invocation
        multicastdel (50, 20);
    }
}

/* Output
 Sum of integers is = 70
 Difference of integer is = 30

*/
```

Q4. What are Events?

Ans. An event enables a class or object to provide notifications in your application. An event is declared like a field except that the declaration includes an event keyword and the type must be a delegate type. An event can have many handlers. A method that handles an event is called an event handler.

Q5. How to define an Event or Event Handler?

Ans. You can declare your event handlers as you define any other delegate. By convention, event handlers in the .NET Framework always return void and take two parameters. The first parameter is the "source" of the event and a second parameter is an object derived from EventArgs. Your event handlers will need to follow this design pattern. EventArgs is the base class for all event data. The EventArgs class inherits all its methods from Object class.

```
Class MyEventClass
{
    public delegate void EventHandler();

    static void VB()
    {
        Console.WriteLine("VB.NET");
    }
    static void C()
    {
        Console.WriteLine("C#.NET");
    }
    static void SQL()
    {
        Console.WriteLine("MS-SQL");
    }
    static void AJAX()
    {
        Console.WriteLine("AJAX.NET");
    }

   public static event EventHandler show;

    static void Main()
    {
        // Add event handlers to Show event.
        show += new EventHandler(VB);
        show += new EventHandler(C);
        show += new EventHandler(SQL);
        show += new EventHandler(AJAX);

        // invoke the event.
        show.Invoke();
    }

}
/*Output
 VB.NET
 C#.NET
 MS-SQL
*/
```

```
class MyEventHandler
{
    public event EventHandler EventRaised;  //declaration of event handler
    public void RaiseAnEvent()
    {
        EventHandler handler = EventRaised;  // instance
        if (handler != null)
        {
            handler(this, EventArgs.Empty);
        }
    }
}
class MyEventObserver
{
    public void MyMethodHandleEvent(object sender, EventArgs args) // event
    {
        Console.WriteLine("Event occurred at " + sender);
    }
}
class ExecutionClass
{
    static void Main()
    {
        MyEventHandler objHandler = new MyEventHandler();
        MyEventObserver objObserver = new MyEventObserver();
        objHandler.EventRaised += objObserver.MyMethodHandleEvent;    //invoke
        objHandler.RaiseAnEvent();
    }
}
/* Output
 Event occurred at TestApplication.MyEventHandler.
*/
```

Q6. What are the differences between Events and Delegates?

Ans. The differences between events and delegates are given below:

Events	Delegates
While defining interface we can use events	We can't use delegate in the interface definition.
An event can only be invoked from the class that declares it.	Delegates can be invoked from derived classes and clients.
The event comes with its pair of assessors i.e. Add and Remove. Which is assigned and unassigned with a += and -= operator.	No such accessors are required here.
The event has a restrictive signature and must always be of the form `Event(object sender, EventArgs args)`	Delegates are not bounded of such restrictive signature.

Q7. Do Events have a return type?

Ans. No, events have not any return type, not even void type.

Q8. Can Events have access modifiers?

Ans. Yes, you can have all access modifiers in events. You can have events with the protected keyword, which will be accessible only to inherited classes. You can have private events only for objects in that class.

Q9. What is Attribute?

Ans. An attribute provides declarative information about types, class, methods, properties, fields etc. Attributes are accessed at compile-time or runtime through the metadata or reflection.

Q10. Can you explain some built-in attributes?

Ans. The list of some inbuilt attributes is given below-

Inbuilt Attribute	Targets	Description
DllImport	Method	Specifies the DLL location that contains the implementation of an external method.
Serializable	Field, class	Applies to a class or field which specifies that class or fields will be serialized.
NonSerialized	Class, struct, enum, delegate	Specifies that all public and private fields of this type can be serialized.

Obsolete	All, except Assembly, Parameter, and Return Type	Specify a type or method obsolete i.e. it informs to the user that this one will be removed in future versions of the code.

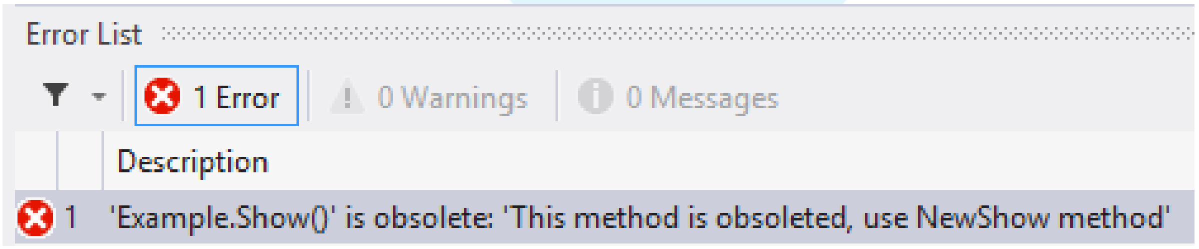
Q11. How to show a message in your latest version of code about your old method removal?

Ans. This can be handled using Obsolete attribute as given below:

For example,

```
public class Example
{
    [Obsolete("This method is obsoleted, use NewShow method", true)]
    public void Show()
    {
        //To DO:
    }
    public void NewShow()
    {
        //To DO:
    }
}
class Program
{
    static void Main(string[] args)
    {
        Example obj = new Example();
        obj.Show();
    }
}
```

When you will run this program, you will get the following error as given below-


Error List

1 Error | 0 Warnings | 0 Messages

```
   Description
1  'Example.Show()' is obsolete: 'This method is obsoleted, use NewShow method'
```

Q12. How to create a custom attribute class?

Ans. You can also create your own custom attributes by defining a custom attribute class that must be derived directly or indirectly from *Attribute* class, which makes identifying attribute definitions in metadata fast and easy.

```
[System.AttributeUsage(System.AttributeTargets.Class)]
public class AuthorInfoAttribute : System.Attribute
{
    private string name;
    public string address;

    public AuthorInfoAttribute(string name)
    {
        this.name = name;
```

```
            address = "Delhi";
        }
}
[AuthorInfo("Shailendra Chauhan", address = "Noida")]
class Author
{
    // TO DO:
}
```

The custom attribute class name may or may not have attribute suffix; it is optional.

11

Collections and Generics

Q1. What are Collections?

Ans. A collection is a set of related objects. Unlike arrays, a collection can grow and shrink dynamically as the number of objects added or deleted. A collection is a class, so you must declare a new collection before you can add elements to that collection.

Q2. What are various collections in C#?

Ans. The .NET Framework provides various collections like ArrayList, HashTable, SortedList, Stack and Queue etc. All these collections exist in *System.Collections* namespace.

Class	Description
ArrayList	Represents an array of objects whose size is dynamically increased or decreased as required. It is an alternative to an array. It supports add and remove methods for adding and removing objects from the collection.
Hashtable	Represents a collection of objects which are stored in key/value pair's fashion, where the key is a hash code and value is an object. The key is used to access or manipulates the objects in the collection. It supports add and remove methods for adding and removing objects from the collection.
SortedList	Represents a collection of objects which are stored in key/value pairs fashion like HashTable and can be sorted by the keys. It can accessible by key or by index number. Typically, it a combination of ArrayList and HashTable. It supports add and remove methods for adding and removing objects from the collection.
Stack	Represents a last in, first out (LIFO) collection of objects. It supports push and pop methods for adding and removing objects from the collection.
Queue	Represents a first in, first out (FIFO) collection of objects. It supports Enqueue and Dequeue methods for adding and removing objects from the collection.

Q3. What are Generics in C#?

Ans. In C# 2.0**,** Generics were introduced which enables you to define type-safe classes or interfaces or methods or events or collections without compromising type safety, performance, or productivity.

In generics, a generic type parameter is supplied between the open (<) and close (>) brackets and which makes it strongly typed collections i.e. generics collections contain only similar types of objects.

Q4. What are the advantages of generics?

Ans. There are following advantages of generics:

- Generics provide code re-usability.
- Generics enforce type safety. So, retrieving the data from generics doesn't require any type casting which makes your code clean and easier to write and maintain.
- Generics provide better performance since they don't require to do type casting.
- Generics decreases the number of run-time errors due to typecasting or boxing/unboxing because the types used in generic operations are evaluated at compile time.

Q5. What are various Generic Collections in C#?

Ans. The .NET Framework provides various generics collections like List<T>, Dictionary<TKey, TValue>, SortedList<T>, Stack<T> and Queue<T> etc.

Class	**Description**
List<T>	Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.
Dictionary<TKey, TValue>	Represents a collection of keys and values. It supports add and remove methods for adding and removing objects from the collection.
SortedList<TKey, TValue>	Represents a collection of key/value pairs that are sorted by key based. It supports add and remove methods for adding and removing objects from the collection.
Stack<T>	Represents a last in, first out (LIFO) strongly typed collection of objects. It supports push and pop methods for adding and removing objects from the collection.
Queue<T>	Represents a first in, first out (FIFO) strongly typed collection of objects. It supports Enqueue and Dequeue methods for adding and removing objects from the collection.

Q6. What is the name of the namespace for collections in C#?

Ans. The collections are the part of *System.Collections.Generic* namespace.

Q7. What are the differences between Generic Collections and Non-Generic Collections in C#?

Ans. The differences between generic and non-generic collections are given below-

Generics Collections	**No-Generic Collections**
Store elements of same data types.	Can store elements of different data types.
There is no need of typecasting or boxing/unboxing while retrieving elements from collections.	Need type casting or boxing/unboxing while retrieving elements from collections.

Provides code reusability.	Do not provide code reusability.
Provide type safety hence, incompatible type object cannot be stored in a generics collections.	Do not provide type safety which causes an error at runtime if the incompatible type object is stored.
Generics are fast as compared to collections.	Collections are slow as compared to generics.

Q8. What are Concurrent Collections?

Ans. The concurrent collections were introduced with C# 4.0 in .NET Framework 4.0. Concurrent collections are thread-safe and useful where multi-threaded read and write access to a collection is desired. These collections are the key to parallel programming that was introduced in .NET Framework 4.0.

The .NET Framework provides various concurrent collections like ConcurrentDictionary<Key, Value>, ConcurrentStack<T>, ConcurrentQueue<T>, ConcurrentBag<T> etc.

Class	**Description**
ConcurrentDictionary< Key , Value>	Represents a thread-safe collection of keys and values.
ConcurrentStack<T>	Represents a last in, first out (LIFO) thread safe and strongly typed collection of objects.
ConcurrentQueue<T>	Represents a first in, first out (FIFO) thread safe and strongly typed collection of objects
ConcurrentBag<T>	Represents a thread-safe collection of unordered elements.

Q9. What is the name of the namespace for concurrent collections in C#?

Ans. The concurrent collections exist in *System.Collections.Concurrent namespace*.

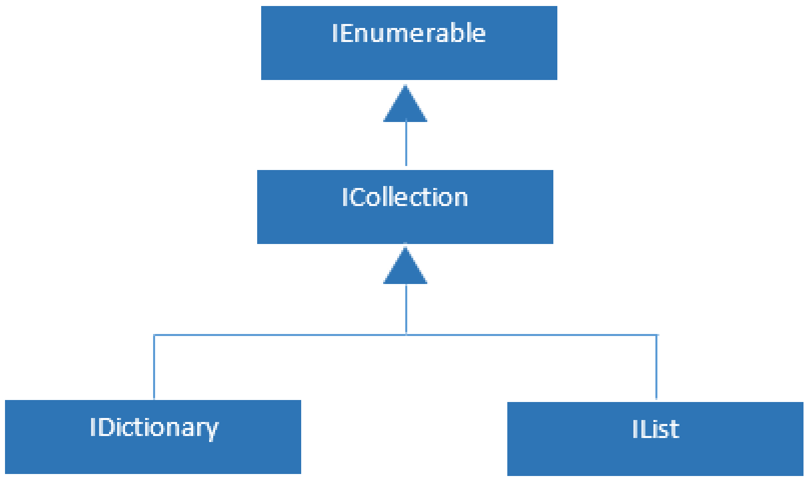
Q10. What is a base interface for all the collection?

Ans. IEnumerable is the base interface for all the collection in .NET.

Q11. What are collection interfaces in C#?

Ans. All of the collection types use some common interfaces. These common interfaces define the basic functionality of each collection class. The key collections interfaces are – IEnumerable, ICollection, IDictionary and IList.

IEnumerable acts as a base interface for all the collection types that is extended by ICollection. ICollection is further extended by IDictionary and IList.



Interface	Description
IEnumerable	Provides an enumerator which supports a simple iteration over a non-generic collection.
ICollection	Defines size, enumerators and synchronization methods for all non-generic collections.
IDictionary	Represents a non-generic collection of key/value pairs.
IList	Represents a non-generic collection of objects that can be individually accessed by index.

All collections interfaces are not implemented by all the collections classes. It depends on the collection class behaviours. **For example**, the IDictionary interface would be implemented by only those collection classes which support key/value pairs, like HasTable and SortedList etc.

12 Threading

Q1. What are the thread and process?

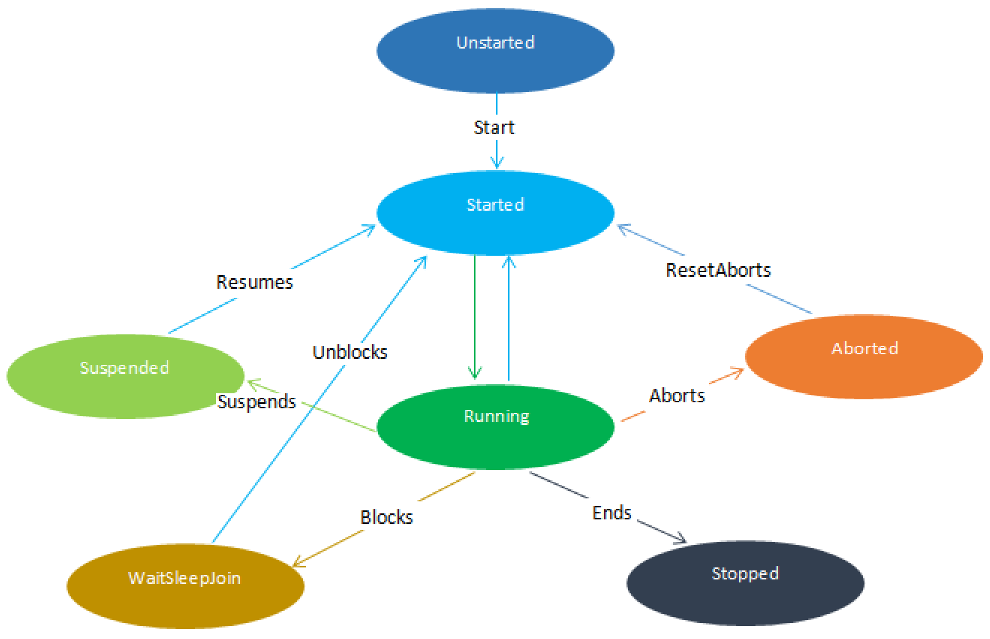
Ans. As per MSDN “A thread is a basic unit to which the operating system allocates processor time”. A thread can execute any part of the executing program which is known as a process. One or more threads can be used to execute a process and share a process virtual address space and system resources.

Q2. What is Thread Synchronization?

Ans. Synchronization is useful when multiple threads access the same shared resources or data. You can synchronize access to shared resources using the lock keyword. By doing so, only one thread at a time accesses and changes the shared resources.

Q3. Explain Thread life cycle?

Ans. The most common thread states are illustrated in the following diagram, and what happens when a thread moves into each state:

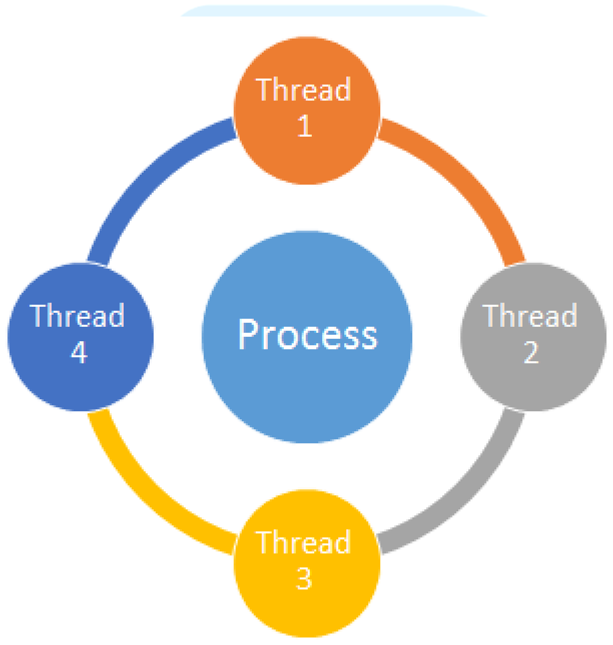


Thread Life Cycle

State	Description
Unstarted	Represents initial state and thread is not started yet.
Started	Represents a thread has been started by invoking Thread.Start() method.
Running	Represents a thread is running.
Stopped	Represents a thread has been finished.
WaitSleepJoin	Represents a thread is blocked. This might be because of calling Thread.Sleep() or Thread.Join() methods, or requesting a lock.
Suspended	Represents a thread has been suspended.
Aborted	Represents a thread has been aborted i.e. it is dead. But its state is changed to Stopped.

Q4. What is Multithreading in C#?

Ans. Multithreading allows you to perform more than one operation concurrently. The .NET Framework *System.Threading* namespace is used to perform threading in C#.

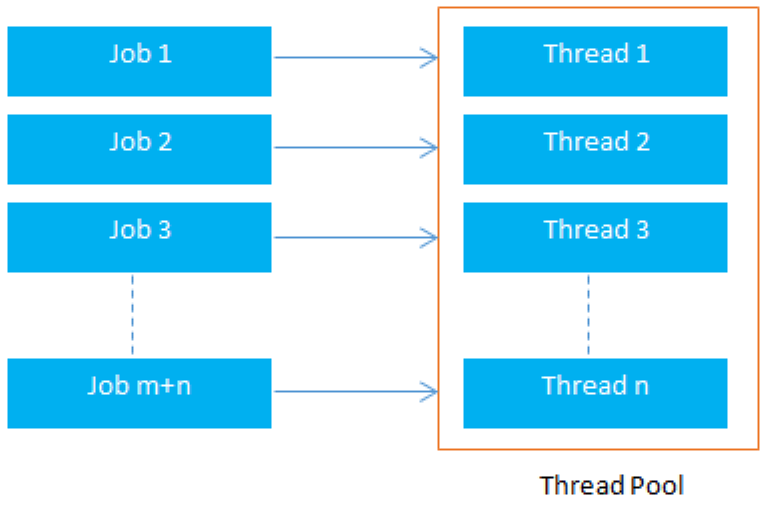


For example, you can use one thread to monitor input from the user, second thread to perform background tasks, third thread to process user input and fourth thread to show the output of the third thread processing.

Q5. What is a Thread Pool?

Ans. A thread pool is a collection of threads which are re-used when required, instead of creating a new thread every time. The thread pool is used to decrease the number of threads within an application and provide a better threads management.

The .NET Framework *ThreadPool* class is used to implement a thread pool. This is a static class that you can directly access.



Thread pooling provides you better performance and better system stability when you need to create or destroy too many threads for your applications.

Q6. What namespace is used for Threading?

Ans. System.Threading namespace is used for threading.

References

This book has been written by referring to the following sites:

1. https://docs.microsoft.com/en-us/aspnet/overview - Microsoft Docs - ASP.NET
2. https://stackoverflow.com/questions/tagged/c# - Stack Overflow - C#
3. https://www.dotnettricks.com/learn/csharp - Dot Net Tricks - C#