

F

[illegible]

UNIT III:**Working with data using Entity Framework core:**

Understanding modern databases: Understanding legacy Entity Framework, Understanding Entity Framework Core, Creating a console app for working with EF Core, Using a sample relational database, Using Microsoft SQL Server for Windows, Creating the Northwind sample database for SQL Server.

Setting up EF Core: Choosing an EF Core database provider, Connecting to a database, Defining the Northwind database context class

Defining EF Core models: Using EF Core conventions to define the model, Using EF Core annotation attributes to define the model, Using the EF Core Fluent API to define the model,

Building an EF Core model, Adding tables.

Querying EF Core models: Filtering included entities, Filtering and sorting pro, Getting the generated SQL, Logging EF Core using a custom logging provider, Pattern matching with Like.

Loading patterns with EF Core: Eager loading entities, Enabling lazy loading, Explicit loading entities.

Manipulating data with EF Core: Inserting entities, Updating entities, Deleting entities, Pooling database contexts

UNIT IV:**Building Websites Using ASP.NET Core Razor Pages:**

Understanding web development: Understanding HTTP, Using Google Chrome to make HTTP requests, Understanding client-side web development technologies

Understanding ASP.NET Core: Classic ASP.NET versus modern ASP.NET Core, Creating an empty ASP.NET Core project, Testing and securing the website, Controlling the hosting environment, Separating configuration for services and pipeline, Enabling a website to serve static content

Exploring ASP.NET Core Razor Pages: Enabling Razor Pages, Adding code to a Razor Page, Using shared layouts with Razor Pages, Using code-behind files with Razor Pages

Using Entity Framework Core with ASP.NET Core: Configure Entity Framework Core as a service, Manipulating data using Razor Pages, Injecting a dependency service into a Razor Page

Text books and Reference books	Text Book(s): [1]. Mark J Price, “C# 10 and .NET 6 – Modern Cross-Platform Development - Sixth Edition”, Oreilly publications. Nov 2021 [2]. Joseph Albahari, “C#10 in a Nut Shell”, Oreillym publications. Nov 2021 Reference Books: [1]. Kemal Birer, “ASP.NET Core for Jobseekers” bpb publications 2021. [2]. Kogent Learning Solutions, “ASP.NET4.5 PROGRAMMING” Black Book, dreamtech press, 2013.
E-resources and other digital material	[1]. Scott Hanselman, Maira Wenzel, Modern Web Development with .NET 6 Ep1: Create a web UI with ASP.NET Core, https://docs.microsoft.com/en-us/shows/learn-live/modern-web-development-net6-ep01-create-web-ui-aspnet-core (16-05-2022) [2]. Rehan Saeed, Upgrading ASP.NET Core to .NET 6 & C# 10, https://techcommunity.microsoft.com/t5/web-development/upgrading-asp-net-core-to-net-6-and-c-10/m-p/2927530 and https://www.youtube.com/watch?v=T6iP7QPWmPI (Microsoft) (16-05-2022) [3]. Cam Sopar, Getting Started with Entity Framework Core, https://docs.microsoft.com/en-us/shows/entity-framework-core-101/getting-started-with-entity-framework-core (16-05-2022) [4]. Kaushik Roy Chowdhury, ASP.NET 6.0 - Build Hands-On Web Projects, https://www.udemy.com/course/aspnetcore-31-build-hands-on-web-projects/ (16-05-22)

20IT5351B DOTNET TECHNOLOGIES LAB

Course Category:	Program Elective1	Credits:	3
Course Type:	Laboratory	Lecture-Tutorial-Practice:	0-0-3
Prerequisites:	20ES2103A: Object Oriented programming using Python	Continuous Evaluation:	30
		Semester end Evaluation:	70
		Total Marks:	100
Course Outcomes	Upon successful completion of the course, the student will be able to:		
	CO1	Implement the Console Applications in C#.	

	CO2	Implement the object oriented features of Dot Net frame work in solving Real-world Applications.														
	CO3	Design web application with variety of web controls and validation controls.														
	CO4	Develop dynamic web applications that include database interactivity.														
Contribution of Course Outcomes towards achievement of Program Outcomes(1-Low, 2- Medium, 3-High)																
CO	PO												PSO		BTL	POI
	1	2	3	4	5	6	7	8	9	10	11	12	1	2		
CO1		1													3	2.2.4
CO2												2	2		3	12.2.1
CO3		2			3									3	3	2.2.4
CO4		3			3				3						6	2.2.5,5.2.3, 9.1.2, 12.2.1
Course Content	Week 1: Classes and Objects <ol style="list-style-type: none"> Write a console application that performs type conversion. Understand the concept that performs boxing and unboxing of different types of variables. Implement arithmetic, logical and relational operators. Understand the concept of classes and objects. 															
	Week 2: Static Data members, static members functions and properties <ol style="list-style-type: none"> Identify the differences in the implementation of single and multiple objects. Understand the concept of static data members and member functions. Implement the static member functions in a class for the given application. Understand the concept of properties. 															

	<p>Week 3: Indexes and Structs</p> <ol style="list-style-type: none"> 1) Implement the concept of Indexers and identify the differences between Properties and Indexers. 2) Understand and implement the concept of Structs.
	<p>Week 4: Interfaces, Pointers, Delegates and Events</p> <ol style="list-style-type: none"> 1) Implement the concept for Interfaces. 2) Implement different types of Flow controls. 3) Implement the concept for Delegates. 4) Implement the concept for Events.
	<p>Week 5: Exception Handling</p> <ol style="list-style-type: none"> 1) Implement the concept for Exception Handling. 2) Create an application for performing Calculator Operations. 3) Design a Registration form with different types of controls using ASP.NET
	<p>Week 6: Data Access with Entity Framework</p> <ol style="list-style-type: none"> 1. Understand the concept of Data Access using Entity Framework 2. Create a website for a bank and include types of navigation. 3. Create a Web App to display all the Empname and Deptid of the employee from the database using SQL source control and bind it to GridView . Database fields are (DeptId, DeptName, EmpName, Salary).

	<p>Week 7: Data Access with Entity Framework-II</p> <ol style="list-style-type: none"> 1) Create a Login Module which adds Username and Password in the database. Username in the database should be a primary key. 2) Create a web application to insert 3 records inside the SQL database table having following fields (DeptId, DeptName, EmpName, Salary). Update the salary for any one employee and increment it to 15% of the present salary. Perform delete operation on 1 row of the database table.
	<p>Week 8: Dynamic Data Application using Razor Pages</p> <p>Develop a dynamic website for Hotel Management using ASP.NET Razor pages .</p>
	<p>Week 9: Dynamic Data Application using Razor Pages</p> <p>Develop a dynamic website for Bank Application using ASP.NET Razor pages</p>
	<p>Week 10: Dynamic Data Application using Razor Pages</p> <p>Case Study to develop a dynamic website using Dynamic controls using ASP.NET Razor Pages.</p>

Text books and Referenc e books	<p>Text Book(s):</p> <p>[1]Kogent Learning Solutions, “NET4.5 PROGRAMMING” Black Book, dreamtech press, 2013.</p> <p>[2]Kogent Learning Solutions, “ ASP.NET4.5 PROGRAMMING” Black Book, dreamtech press, 2013.</p> <p>Reference Books:</p> <p>[1] HerbertSchildt, “C# 4.0:complete reference”,McGrawHill,2010.</p> <p>[2]Matthew MacDonald, “ASP.NET: The complete Reference”, McGrawHill, 2002.</p> <p>[3] Chris Hart, John Kauffman, Dave Sussman, Chriss Ullman “ASP.Net 2.0 with c#” Wrox, 2006.</p>

Content

.NET Framework

.NET is a framework to develop software applications. It is designed and developed by Microsoft and the first beta version released in 2000.

It is used to develop applications for web, Windows, phone. Moreover, it provides a broad range of functionalities and support.

This framework contains a large number of class libraries known as Framework Class Library (FCL). The software programs written in .NET are executed in the execution environment, which is called CLR (Common Language Runtime). These are the core and essential parts of the .NET framework.

This framework provides various services like memory management, networking, security, memory management, and type-safety.

The .Net Framework supports more than 60 programming languages such as C#, F#, VB.NET, J#, VC++, JScript.NET, APL, COBOL, Perl, Oberon, ML, Pascal, Eiffel, Smalltalk, Python, Cobra, ADA, etc.

Following is the .NET framework Stack that shows the modules and components of the Framework.

The .NET Framework is composed of four main components:

1. Common Language Runtime (CLR)
2. Framework Class Library (FCL),
3. Core Languages (WinForms, ASP.NET, and ADO.NET), and
4. Other Modules (WCF, WPF, WF, Card Space, LINQ, Entity Framework, Parallel LINQ, Task Parallel Library, etc.)

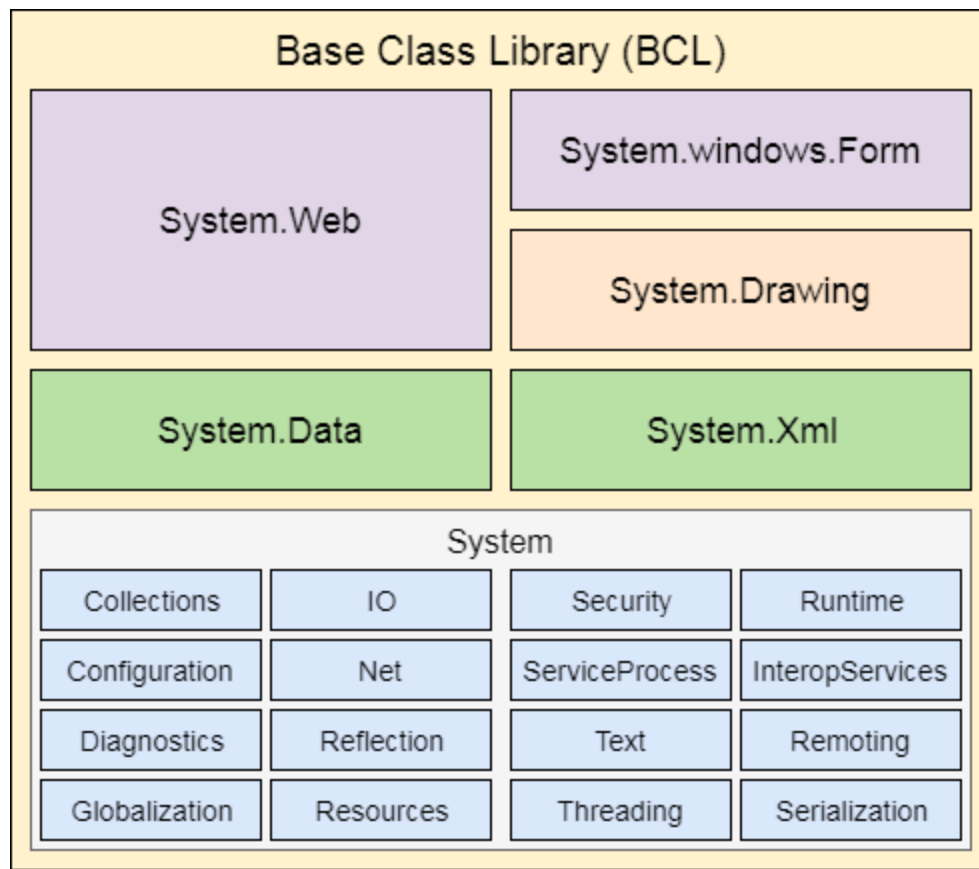
.NET APIs for Store/UWP apps		Task-Based Async Model		4.5 2012
Parallel LINQ		Task Parallel Library		4.0 2010
LINQ		Entity Framework		3.5 2007
WPF	WCF	WF	Card Space	3.0 2006
WinForms	ASP.NET	ADO.NET		.NET Framework 2.0 2005
Framework Class Library				
Common Language Runtime				

CLR (Common Language Runtime)

It is a program execution engine that loads and executes the program. It converts the program into native code. It acts as an interface between the framework and operating system. It does exception handling, memory management, and garbage collection. Moreover, it provides security, type-safety, interoperability, and portability.

FCL (Framework Class Library)

It is a standard library that is a collection of thousands of classes and used to build an application. The BCL (Base Class Library) is the core of the FCL and provides basic functionalities.



WinForms

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

ASP.NET

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

ADO.NET

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

WPF (Windows Presentation Foundation)

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

WCF (Windows Communication Foundation)

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

WF (Workflow Foundation)

Windows Workflow Foundation (WF) is a Microsoft technology that provides an API, an in-process workflow engine, and a rehostable designer to implement long-running processes as workflows within .NET applications.

LINQ (Language Integrated Query)

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

Entity Framework

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

Parallel LINQ

Parallel LINQ or PLINQ is a parallel implementation of LINQ to objects. It combines the simplicity and readability of LINQ and provides the power of parallel programming.

It can improve and provide fast speed to execute the LINQ query by using all available computer capabilities.

Apart from the above features and libraries, .NET includes other APIs and Model to improve and enhance the .NET framework.

Registration page creation

https://www.youtube.com/watch?v=UfMbMm_lm1Q

What is C#

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

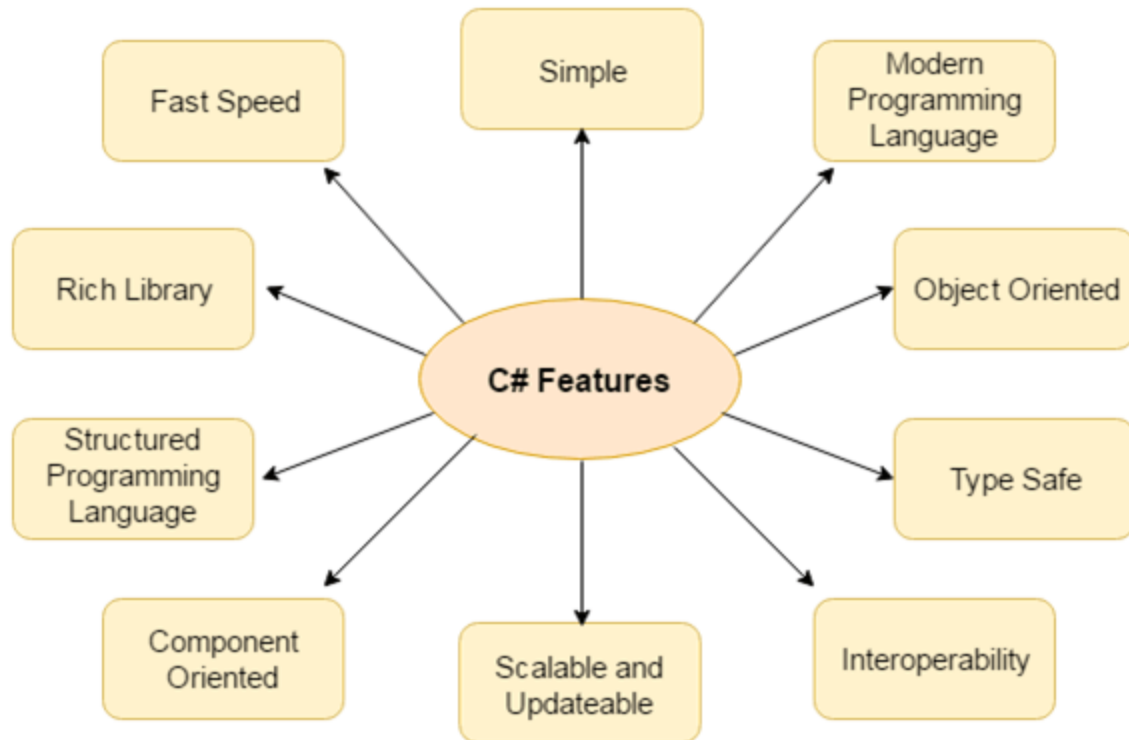
By the help of C# programming language, we can develop different types of secured and robust applications:

- Window applications
- Web applications
- Distributed applications
- Web service applications
- Database applications etc.

C# Features

C# is object oriented programming language. It provides a lot of **features** that are given below.

1. Simple
2. Modern programming language
3. Object oriented
4. Type safe
5. Interoperability
6. Scalable and Updateable
7. Component oriented
8. Structured programming language
9. Rich Library
10. Fast speed



1) Simple

C# is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Modern Programming Language

C# programming is based upon the current trend and it is very powerful and simple for building scalable, interoperable and robust applications.

3) Object Oriented

C# is object oriented programming language. OOPs makes development and maintenance easier where as in

Procedure-oriented programming language it is not easy to manage if code grows as project size grow.

4) Type Safe

C# type safe code can only access the memory location that it has permission to execute. Therefore it improves a security of the program.

5) Interoperability

Interoperability process enables the C# programs to do almost anything that a native C++ application can do.

6) Scalable and Updateable

C# is automatic scalable and updateable programming language. For updating our application we delete the old files and update them with new ones.

7) Component Oriented

C# is component oriented programming language. It is the predominant software development methodology used to develop more robust and highly scalable applications.

8) Structured Programming Language

C# is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

9) Rich Library

C# provides a lot of inbuilt functions that makes the development fast.

10) Fast Speed

The compilation and execution time of C# language is fast.

C# Example: Hello World

In C# programming language, a simple "hello world" program can be written by multiple ways. Let's see the top 4 ways to create a simple C# example:

- Simple Example
- Using System
- Using public modifier
- Using namespace

C# Simple Example

```
1. class Program
2. {
3.     static void Main(string[] args)
4.     {
5.         System.Console.WriteLine("Hello World!");
6.     }
7. }
```

Output:

Hello World!

C# Variable

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

The basic variable type available in C# can be categorized as:

Variable Type	Example
Decimal types	decimal
Boolean types	True or false value, as assigned
Integral types	int, char, byte, short, long
Floating point types	float and double
Nullable types	Nullable data types

Let's see the syntax to declare a variable:

1. type variable_list;

The example of declaring variable is given below:

1. int i, j;
2. double d;
3. float f;
4. char ch;

Here, i, j, d, f, ch are variables and int, double, float, char are data types.

We can also provide values while declaring the variables as given below:

1. int i=2,j=4; //declaring 2 variable of integer type
2. float f=40.2;

3. `char ch='B';`

C# Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character etc.

There are 3 types of data types in C# language.

Types	Data Types
Value Data Type	short, int, char, float, double etc
Reference Data Type	String, Class, Object and Interface
Pointer Data Type	Pointers

Value Data Type

The value data types are integer-based and floating-point based. C# language supports both signed and unsigned literals.

There are 2 types of value data type in C# language.

1) Predefined Data Types - such as Integer, Boolean, Float, etc.

2) User defined Data Types - such as Structure, Enumerations, etc.

The memory size of data types may change according to 32 or 64 bit operating system.

Let's see the value data types. It size is given according to 32 bit OS.

Data Types	Memory Size	Range
char	1 byte	-128 to 127
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 127
short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	4 byte	-2,147,483,648 to -2,147,483,647
signed int	4 byte	-2,147,483,648 to -2,147,483,647
unsigned int	4 byte	0 to 4,294,967,295

long	8 byte	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
signed long	8 byte	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long	8 byte	0 - 18,446,744,073,709,551,615
float	4 byte	1.5×10^{-45} - 3.4×10^{38} , 7-digit precision
double	8 byte	5.0×10^{-324} - 1.7×10^{308} , 15-digit precision
decimal	16 byte	at least -7.9×10^{28} - 7.9×10^{28} , with at least 28-digit precision

Reference Data Type

The reference data types do not contain the actual data stored in a variable, but they contain a reference to the variables.

If the data is changed by one of the variables, the other variable automatically reflects this change in value.

There are 2 types of reference data type in C# language.

1) Predefined Types - such as Objects, String.

2) User defined Types - such as Classes, Interface.

Pointer Data Type

The pointer in C# language is a variable, it is also known as locator or indicator that points to an address of a value.

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Declaring a pointer

The pointer in C# language can be declared using * (asterisk symbol).

1. `int * a; //pointer to int`
2. `char * c; //pointer to char`

C# Control Statement

Function

Function is a block of code that has a signature. Function is used to execute statements specified in the code block. A function consists of the following components:

Function name: It is a unique name that is used to make Function call.

Return type: It is used to specify the data type of function return value.

Body: It is a block that contains executable statements.

Access specifier: It is used to specify function accessibility in the application.

Parameters: It is a list of arguments that we can pass to the function during call.

C# Function Syntax

1. <access-specifier><return-type>FunctionName(<parameters>)
2. {
3. // function body
4. // return statement
5. }

Access-specifier, parameters and return statement are optional.

Let's see an example in which we have created a function that returns a string value and takes a string parameter.

C# Function: using no parameter and return type

A function that does not return any value specifies **void** type as a return type. In the following example, a function is created without return type.

```
1. using System;
2. namespace FunctionExample
3. {
4.     class Program
5.     {
6.         // User defined function without return type
7.         public void Show() // No Parameter
8.         {
9.             Console.WriteLine("This is non parameterized function");
10.            // No return statement
11.        }
12.        // Main function, execution entry point of the program
13.        static void Main(string[] args)
14.        {
15.            Program program = new Program(); // Creating Object
16.            program.Show(); // Calling Function
17.        }
18.    }
19. }
```

Output:

```
This is non parameterized function
```

C# Function: using parameter but no return type

```
1. using System;
2. namespace FunctionExample
3. {
4.     class Program
5.     {
6.         // User defined function without return type
7.         public void Show(string message)
8.         {
9.             Console.WriteLine("Hello " + message);
10.            // No return statement
11.        }
12.        // Main function, execution entry point of the program
13.        static void Main(string[] args)
14.        {
15.            Program program = new Program(); // Creating Object
16.            program.Show("Rahul Kumar"); // Calling Function
17.        }
18.    }
19. }
```

Output:

```
Hello Rahul Kumar
```

A function can have zero or any number of parameters to get data. In the following example, a function is created without parameters. A function without parameter is also known as **non-parameterized** function.

C# Function: using parameter and return type

```
1. using System;
2. namespace FunctionExample
3. {
4.     class Program
5.     {
6.         // User defined function
7.         public string Show(string message)
8.         {
9.             Console.WriteLine("Inside Show Function");
10.            return message;
11.        }
12.        // Main function, execution entry point of the program
13.        static void Main(string[] args)
14.        {
15.            Program program = new Program();
16.            string message = program.Show("Rahul Kumar");
17.            Console.WriteLine("Hello "+message);
18.        }
19.    }
20. }
```

Output:

Inside Show Function

Hello Rahul Kumar

C# Call By Value

In C#, value-type parameters are that pass a copy of original value to the function rather than reference. It does not modify the original value. A change made in passed value does not alter the actual value. In the following example, we have pass value during function call.

C# Call By Value Example

```
1. using System;
2. namespace CallByValue
3. {
4.     class Program
5.     {
6.         // User defined function
7.         public void Show(int val)
8.         {
9.             val *= val; // Manipulating value
10.            Console.WriteLine("Value inside the show function
            "+val);
11.            // No return statement
12.        }
13.        // Main function, execution entry point of the program
14.        static void Main(string[] args)
15.        {
16.            int val = 50;
17.            Program program = new Program(); // Creating
            Object
```

```
18.         Console.WriteLine("Value before calling the function
           "+val);
19.         program.Show(val); // Calling Function by passing
           value
20.         Console.WriteLine("Value after calling the function "
           + val);
21.     }
22. }
23. }
```

Output:

Value before calling the function 50

Value inside the show function 2500

Value after calling the function 50

C# Call By Reference

C# provides a **ref** keyword to pass argument as reference-type. It passes reference of arguments to the function rather than copy of original value. The changes in passed values are permanent and **modify** the original variable value.

C# Call By Reference Example

```
1. using System;
2. namespace CallByReference
3. {
4.     class Program
5.     {
6.         // User defined function
7.         public void Show(ref int val)
8.         {
9.             val *= val; // Manipulating value
10.            Console.WriteLine("Value inside the show function
            "+val);
11.            // No return statement
12.        }
13.        // Main function, execution entry point of the program
14.        static void Main(string[] args)
15.        {
16.            int val = 50;
17.            Program program = new Program(); // Creating
            Object
18.            Console.WriteLine("Value before calling the function
            "+val);
19.            program.Show(ref val); // Calling Function by passing
            reference
20.            Console.WriteLine("Value after calling the function "
            + val);
21.        }
22.    }
```

```
23. }
```

Output:

Value before calling the function 50

Value inside the show function 2500

Value after calling the function 2500

C# Out Parameter

C# provides **out** keyword to pass arguments as out-type. It is like reference-type, except that it does not require variable to initialize before passing. We must use **out** keyword to pass argument as out-type. It is useful when we want a function to return multiple values.

C# Out Parameter Example 1

```
1. using System;
2. namespace OutParameter
3. {
4.     class Program
5.     {
6.         // User defined function
7.         public void Show(out int val) // Out parameter
```

```

8.    {
9.        int square = 5;
10.        val = square;
11.        val *= val; // Manipulating value
12.    }
13.    // Main function, execution entry point of the program
14.    static void Main(string[] args)
15.    {
16.        int val = 50;
17.        Program program = new Program(); // Creating
        Object
18.        Console.WriteLine("Value before passing out variable
        " + val);
19.        program.Show(out val); // Passing out argument
20.        Console.WriteLine("Value after recieving the out
        variable " + val);
21.    }
22.    }
23. }

```

Output:

Value before passing out variable 50

Value after receiving the out variable 25

The following example demonstrates that how a function can return multiple values.

C# Out Parameter Example 2

```
1. using System;
2. namespace OutParameter
3. {
4.     class Program
5.     {
6.         // User defined function
7.         public void Show(out int a, out int b) // Out parameter
8.         {
9.             int square = 5;
10.            a = square;
11.            b = square;
12.            // Manipulating value
13.            a *= a;
14.            b *= b;
15.        }
16.        // Main function, execution entry point of the program
17.        static void Main(string[] args)
18.        {
19.            int val1 = 50, val2 = 100;
20.            Program program = new Program(); // Creating
            Object
21.            Console.WriteLine("Value before passing \n val1 = " +
                val1+" \n val2 = "+val2);
22.            program.Show(out val1, out val2); // Passing out
            argument
```

```
23.         Console.WriteLine("Value after passing \n val1 = " +  
        val1 + " \n val2 = " + val2);  
24.     }  
25. }  
26. }
```

Output:

Value before passing

val1 = 50

val2 = 100

Value after passing

val1 = 25

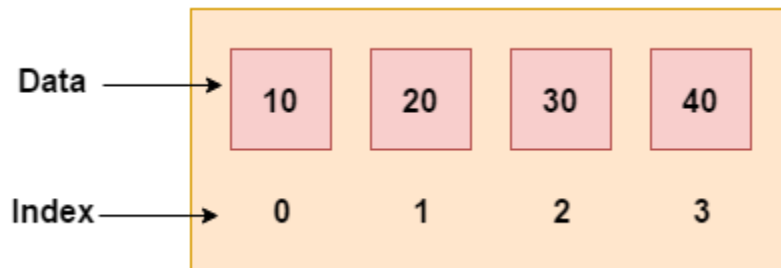
val2 = 25

--

C# Arrays

Like other programming languages, array in C# is a group of similar types of elements that have contiguous memory location. In C#,

array is an *object* of base type **System.Array**. In C#, array index starts from 0. We can store only fixed set of elements in C# array.



Advantages of C# Array

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.

Disadvantages of C# Array

- Fixed size
-

C# Array Types

There are 3 types of arrays in C# programming:

1. Single Dimensional Array
2. Multidimensional Array
3. Jagged Array

C# Single Dimensional Array

To create single dimensional array, you need to use square brackets [] after the type.

```
1. int[] arr = new int[5]; //creating array
```

You cannot place square brackets after the identifier.

x

```
1. int arr[] = new int[5]; //compile time error
```

Let's see a simple example of C# array, where we are going to declare, initialize and traverse array.

```
1. using System;
2. public class ArrayExample
3. {
4.     public static void Main(string[] args)
5.     {
6.         int[] arr = new int[5]; //creating array
7.         arr[0] = 10; //initializing array
8.         arr[2] = 20;
9.         arr[4] = 30;
10.
11.         //traversing array
12.         for (int i = 0; i < arr.Length; i++)
13.         {
14.             Console.WriteLine(arr[i]);
15.         }
16.     }
```

```
17. }
```

Output:

10

0

20

0

30

C# Array Example: Declaration and Initialization at same time

There are 3 ways to initialize array at the time of declaration.

```
1. int[] arr = new int[5]{ 10, 20, 30, 40, 50 };
```

We can omit the size of array.

```
1. int[] arr = new int[] { 10, 20, 30, 40, 50 };
```

We can omit the new operator also.

```
1. int[] arr = { 10, 20, 30, 40, 50 };
```

Let's see the example of array where we are declaring and initializing array at the same time.

```
1. using System;  
2. public class ArrayExample
```

```
3. {
4.   public static void Main(string[] args)
5.   {
6.       int[] arr = { 10, 20, 30, 40, 50 };//Declaration and
       Initialization of array
7.
8.       //traversing array
9.       for (int i = 0; i < arr.Length; i++)
10.      {
11.          Console.WriteLine(arr[i]);
12.      }
13.  }
14. }
```

Output:

10

20

30

40

50

C# Array Example: Traversal using foreach loop

We can also traverse the array elements using foreach loop. It returns array element one by one.

```
1. using System;
2. public class ArrayExample
3. {
4.     public static void Main(string[] args)
5.     {
6.         int[] arr = { 10, 20, 30, 40, 50 };//creating and initializing
           array
7.
8.         //traversing array
9.         foreach (int i in arr)
10.        {
11.            Console.WriteLine(i);
12.        }
13.    }
14. }
```

Output:

10

20

30

40

50

C# Passing Array to Function

In C#, to reuse the array logic, we can create function. To pass array to function in C#, we need to provide only array name.

1. `functionname(arrayname);`//passing array

C# Passing Array to Function Example: print array elements

Let's see an example of C# function which prints the array elements.

```
1. using System;
2. public class ArrayExample
3. {
4.     static void printArray(int[] arr)
5.     {
6.         Console.WriteLine("Printing array elements:");
7.         for (int i = 0; i < arr.Length; i++)
8.         {
9.             Console.WriteLine(arr[i]);
10.        }
11.    }
12.    public static void Main(string[] args)
13.    {
14.        int[] arr1 = { 25, 10, 20, 15, 40, 50 };
15.        int[] arr2 = { 12, 23, 44, 11, 54 };
16.        printArray(arr1); //passing array to function
17.        printArray(arr2);
```

18. }

19. }

Output:

Printing array elements:

25

10

20

15

40

50

Printing array elements:

12

23

44

11

54

C# Passing Array to Function Example: Print minimum number

Let's see an example of C# array which prints minimum number in an array using function.

```
1. using System;
2. public class ArrayExample
3. {
4.     static void printMin(int[] arr)
5.     {
6.         int min = arr[0];
7.         for (int i = 1; i < arr.Length; i++)
8.         {
9.             if (min > arr[i])
10.            {
11.                min = arr[i];
12.            }
13.        }
14.        Console.WriteLine("Minimum element is: " + min);
15.    }
16.    public static void Main(string[] args)
17.    {
18.        int[] arr1 = { 25, 10, 20, 15, 40, 50 };
19.        int[] arr2 = { 12, 23, 44, 11, 54 };
20.
21.        printMin(arr1); //passing array to function
22.        printMin(arr2);
23.    }
24. }
```

Output:

Minimum element is: 10

Minimum element is: 11

C# Passing Array to Function Example: Print maximum number

Let's see an example of C# array which prints maximum number in an array using function.

```
1. using System;
2. public class ArrayExample
3. {
4.     static void printMax(int[] arr)
5.     {
6.         int max = arr[0];
7.         for (int i = 1; i < arr.Length; i++)
8.         {
9.             if (max < arr[i])
10.            {
11.                max = arr[i];
12.            }
13.        }
14.        Console.WriteLine("Maximum element is: " + max);
15.    }
16.    public static void Main(string[] args)
17.    {
```



```
18.      int[] arr1 = { 25, 10, 20, 15, 40, 50 };
19.      int[] arr2 = { 12, 23, 64, 11, 54 };
20.
21.      printMax(arr1); //passing array to function
22.      printMax(arr2);
23.  }
24. }
```

Output:

Maximum element is: 50

Maximum element is: 64

C# Multidimensional Arrays

The multidimensional array is also known as rectangular arrays in C#. It can be two dimensional or three dimensional. The data is stored in tabular form (row * column) which is also known as matrix.

To create multidimensional array, we need to use comma inside the square brackets. For example:

1. `int[,] arr=new int[3,3];`//declaration of 2D array
2. `int[,,] arr=new int[3,3,3];`//declaration of 3D array

C# Multidimensional Array Example

Let's see a simple example of multidimensional array in C# which declares, initializes and traverse two dimensional array.

```
1. using System;
2. public class MultiArrayExample
3. {
4.     public static void Main(string[] args)
5.     {
6.         int[,] arr=new int[3,3];//declaration of 2D array
7.         arr[0,1]=10;//initialization
8.         arr[1,2]=20;
9.         arr[2,0]=30;
10.
11.         //traversal
12.         for(int i=0;i<3;i++){
13.             for(int j=0;j<3;j++){
14.                 Console.Write(arr[i,j]+" ");
15.             }
16.             Console.WriteLine();//new line at each row
17.         }
18.     }
19. }
```

Output:

0 10 0

0 0 20

30 0 0

C# Multidimensional Array Example: Declaration and initialization at same time

There are 3 ways to initialize multidimensional array in C# while declaration.

```
1. int[,] arr = new int[3,3]= { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

We can omit the array size.

```
1. int[,] arr = new int[,] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

We can omit the new operator also.

```
1. int[,] arr = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

Let's see a simple example of multidimensional array which initializes array at the time of declaration.

```
1. using System;
2. public class MultiArrayExample
3. {
4.     public static void Main(string[] args)
5.     {
6.         int[,] arr = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };//declaration
           and initialization
7.
8.         //traversal
9.         for(int i=0;i<3;i++){
```

```
10.         for(int j=0;j<3;j++){
11.             Console.Write(arr[i,j]+" ");
12.         }
13.         Console.WriteLine();//new line at each row
14.     }
15. }
16. }
```

Output:

1 2 3

4 5 6

7 8 9

C# Jagged Arrays

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

Declaration of Jagged array

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

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

Initialization of Jagged array

Let's see an example to initialize jagged array. The size of elements can be different.

1. `arr[0] = new int[4];`
2. `arr[1] = new int[6];`

Initialization and filling elements in Jagged array

Let's see an example to initialize and fill elements in jagged array.

X

1. `arr[0] = new int[4] { 11, 21, 56, 78 };`
2. `arr[1] = new int[6] { 42, 61, 37, 41, 59, 63 };`

Here, size of elements in jagged array is optional. So, you can write above code as given below:

1. `arr[0] = new int[] { 11, 21, 56, 78 };`
2. `arr[1] = new int[] { 42, 61, 37, 41, 59, 63 };`

C# Jagged Array Example

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

1. `public class JaggedArrayTest`
2. `{`
3. `public static void Main()`
4. `{`
5. `int[][] arr = new int[2][]; // Declare the array`
6.
7. `arr[0] = new int[] { 11, 21, 56, 78 }; // Initialize the array`
8. `arr[1] = new int[] { 42, 61, 37, 41, 59, 63 };`
9.
10. `// Traverse array elements`
11. `for (int i = 0; i < arr.Length; i++)`

```

12.    {
13.        for (int j = 0; j < arr[i].Length; j++)
14.        {
15.            System.Console.Write(arr[i][j]+" ");
16.        }
17.        System.Console.WriteLine();
18.    }
19. }
20. }

```

Output:

```
11 21 56 78
```

```
42 61 37 41 59 63
```

Initialization of Jagged array upon Declaration

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

```

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

```

C# Jagged Array Example 2

Let's see a simple example of jagged array which initializes the jagged arrays upon declaration.

```

1. public class JaggedArrayTest
2. {
3.     public static void Main()
4.     {

```

```

5.    int[][] arr = new int[3][] {
6.        new int[] { 11, 21, 56, 78 },
7.        new int[] { 2, 5, 6, 7, 98, 5 },
8.        new int[] { 2, 5 }
9.    };
10.
11.    // Traverse array elements
12.    for (int i = 0; i < arr.Length; i++)
13.    {
14.        for (int j = 0; j < arr[i].Length; j++)
15.        {
16.            System.Console.Write(arr[i][j]+" ");
17.        }
18.        System.Console.WriteLine();
19.    }
20. }
21. }

```

Output:

```
11 21 56 78
```

```
2 5 6 7 98 5
```

```
2 5
```

C# Params

In C#, **params** is a keyword which is used to specify a parameter that takes variable number of arguments. It is useful when we don't know the number of arguments prior. Only one params keyword is allowed and no additional parameter is permitted after params keyword in a function declaration.

C# Params Example 1

```
1. using System;
2. namespace AccessSpecifiers
3. {
4.     class Program
5.     {
6.         // User defined function
7.         public void Show(params int[] val) // Params Paramater
8.         {
9.             for (int i=0; i<val.Length; i++)
10.            {
11.                Console.WriteLine(val[i]);
12.            }
13.        }
14.        // Main function, execution entry point of the program
15.        static void Main(string[] args)
16.        {
17.            Program program = new Program(); // Creating Object
18.            program.Show(2,4,6,8,10,12,14); // Passing arguments of variable length
19.        }
20.    }
21. }
```

Output:

2

4

6

8
10
12
14

C# Params Example 2

In this example, we are using object type params that allow entering any number of inputs of any type.

```
1. using System;
2. namespace AccessSpecifiers
3. {
4.     class Program
5.     {
6.         // User defined function
7.         public void Show(params object[] items) // Params Paramater
8.         {
9.             for (int i = 0; i < items.Length; i++)
10.            {
11.                Console.WriteLine(items[i]);
12.            }
13.        }
14.        // Main function, execution entry point of the program
15.        static void Main(string[] args)
```

```
16.      {
17.          Program program = new Program(); // Creating Object
18.          program.Show("Ramakrishnan Ayyer", "Ramesh", 101, 20.50, "Peter", 'A'); //
           Passing arguments of variable length
19.      }
20.  }
21. }
```

Output:

Ramakrishnan Ayyer

Ramesh

101

20.5

Peter

A

C# Array class

C# provides an Array class to deal with array related operations. It provides methods for creating, manipulating, searching, and sorting elements of an array. This class works as the base class for all arrays in the .NET programming environment.

C# Array Example

```
1. using System;
2. namespace CSharpProgram
3. {
4.     class Program
5.     {
6.         static void Main(string[] args)
7.         {
8.             // Creating an array
9.             int[] arr = new int[6] { 5, 8, 9, 25, 0, 7 };
10.            // Creating an empty array
11.            int[] arr2 = new int[6];
12.            // Displaying length of array
13.            Console.WriteLine("length of first array: "+arr.Length);
14.            // Sorting array
15.            Array.Sort(arr);
16.            Console.Write("First array elements: ");
17.            // Displaying sorted array
18.            PrintArray(arr);
19.            // Finding index of an array element
20.            Console.WriteLine("\nIndex position of 25 is "+Array.IndexOf(arr,25));
21.            // Coping first array to empty array
22.            Array.Copy(arr, arr2, arr.Length);
23.            Console.Write("Second array elements: ");
24.            // Displaying second array
25.            PrintArray(arr2);
26.            Array.Reverse(arr);
27.            Console.Write("\nFirst Array elements in reverse order: ");
```

```
28.         PrintArray(arr);
29.     }
30.     // User defined method for iterating array elements
31.     static void PrintArray(int[] arr)
32.     {
33.         foreach (Object elem in arr)
34.         {
35.             Console.Write(elem+" ");
36.         }
37.     }
38. }
39. }
```

Output:

length of first array: 6

First array elements: 0 5 7 8 9 25

Index position of 25 is 5

Second array elements: 0 5 7 8 9 25

First Array elements in reverse order: 25 9 8 7 5 0

lab programs

Week 1 (17/8/2022)

1. Write a C# program that performs implicit and explicit type conversions
2. Write a program to perform boxing and unboxing operations
3. Implement arithmetic and relational operators in C#
4. Create a class Student with the fields sno, sname, marks in three subjects. Create a method SetStudentDetails() which will read the information from the student. Create another method GetStudentDetails() which will display the information. Also, another method to compute the average mark of the student in three subjects. Display the information.

Week 2(2/9/2022)

1. Create C# class Employee with the fields eno, ename, address, designation, mobile number, salary, city, pincode. Create necessary methods to read the information and display the information. The employer contains 10 employees. The employer wants to know the total salary paid to its employees. As a programmer how do you suggest a solution to his problem?

2)

a) Create a class Emp with the fields eno, ename, company_name. Take the company_name as a static variable. Write a method set() to set the values and print the information using display() method

b) Create a class Emp with the fields eno, ename, company_name. Take the company_name as a static variable. Write a method set() to set the values and print the information using display() method. Modify company_name using a static method.

c) Create a class Reservation with the fields of username, email, no of tickets. Create a method to set the values to the username and email. Write a method to ask the user to enter the number of tickets and reduce that no of tickets. Print the total information

d) Create a class Reservation with the fields of username, email, no of tickets. Create a method to set the values to the username and email. Write a static method to ask the user to enter the number of tickets and reduce that no of tickets. Print the total information

3. Create a C# class to read the book information to a library which includes booknumber, name, pages, issn number, price, year_of_publication. Use C# Properties to store the information into these variables and retrieve the information.

Week 3: Indexes and Structs(9/9/2022)

- 1) Implement the concept of Indexers and identify the differences between Properties and Indexers.
- 2) Understand and implement the concept of Structs.

Properties

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

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

Usage of C# Properties

1. C# Properties can be read-only or write-only.
2. We can have logic while setting values in the C# Properties.

3. We make fields of the class private, so that fields can't be accessed from outside the class directly. Now we are forced to use C# properties for setting or getting values.

C# Properties Example

```
1. using System;
2.     public class Employee
3.     {
4.         private string name;
5.
6.         public string Name
7.         {
8.             get
9.             {
10.                 return name;
11.             }
12.             set
13.             {
14.                 name = value;
15.             }
16.         }
17.     }
18.     class TestEmployee{
19.         public static void Main(string[] args)
20.         {
21.             Employee e1 = new Employee();
22.             e1.Name = "Sonoo Jaiswal";
23.             Console.WriteLine("Employee Name: " + e1.Name);
24.
25.         }
26.     }
```

Output:

```
Employee Name: Sonoo Jaiswal
```


C# Properties Example 2: having logic while setting value

```
1. using System;
2. public class Employee
3. {
4.     private string name;
5.
6.     public string Name
7.     {
8.         get
9.         {
10.             return name;
11.         }
12.         set
13.         {
14.             name = value+" JavaTpoint";
15.
16.         }
17.     }
18. }
19. class TestEmployee{
20.     public static void Main(string[] args)
21.     {
22.         Employee e1 = new Employee();
23.         e1.Name = "Sonoo";
24.         Console.WriteLine("Employee Name: " + e1.Name);
25.     }
26. }
```

Output:

```
Employee Name: Sonoo JavaTpoint
```

C# Properties Example 3: read-only property

```
1. using System;
2. public class Employee
3. {
4.     private static int counter;
5.
6.     public Employee()
7.     {
8.         counter++;
9.     }
10.    public static int Counter
11.    {
12.        get
13.        {
14.            return counter;
15.        }
16.    }
17. }
18. class TestEmployee{
19.     public static void Main(string[] args)
20.     {
21.         Employee e1 = new Employee();
22.         Employee e2 = new Employee();
23.         Employee e3 = new Employee();
24.         //e1.Counter = 10;//Compile Time Error: Can't set value
25.
26.         Console.WriteLine("No. of Employees: " + Employee.Counter);
27.     }
28. }
```

Output:

```
No. of Employees: 3
```

An indexer is a special type of property that allows a class or a structure to be accessed like an array for its internal collection.

An **indexer** allows an object to be indexed such as an array. When you define an indexer for a class, this class behaves similar to a **virtual array**. You can then access the instance of this class using the array access operator ([]).

Syntax

A one dimensional indexer has the following syntax –

```
element-type this[int index] {  
  
    // The get accessor.  
  
    get {  
  
        // return the value specified by index  
  
    }  
  
    // The set accessor.  
  
    set {  
  
        // set the value specified by index  
  
    }  
}
```

Important points to remember on indexers:

- Indexers are always created with **this** keyword.
- Parameterized property are called indexer.
- Indexers are implemented through get and set accessors for the [] operator.
- ref and out parameter modifiers are not permitted in indexer.

```
1. using System;
2. namespace Indexer_example1
3. {
4.     class Program
5.     {
6.         class IndexerClass
7.         {
8.             private string[] names = new string[10];
9.             public string this[int i]
10.            {
11.                get
12.                {
13.                    return names[i];
14.                }
15.                set
16.                {
17.                    names[i] = value;
18.                }
19.            }
20.        }
21.        static void Main(string[] args)
22.        {
23.            IndexerClass Team = new IndexerClass();
24.            Team[0] = "Rocky";
25.            Team[1] = "Teena";
26.            Team[2] = "Ana";
27.            Team[3] = "Victoria";
28.            Team[4] = "Yani";
29.            Team[5] = "Mary";
30.            Team[6] = "Gomes";
31.            Team[7] = "Arnold";
32.            Team[8] = "Mike";
33.            Team[9] = "Peter";
34.            for (int i = 0; i < 10; i++)
35.            {
36.                Console.WriteLine(Team[i]);
37.            }
```

```
38.         Console.ReadKey();
39.     }
40. }
41. }
```

C# Structs

In C#, classes and structs are blueprints that are used to create instance of a class. Structs are used for lightweight objects such as Color, Rectangle, Point etc.

Unlike class, structs in C# are value type than reference type. It is useful if you have data that is not intended to be modified after creation of struct.

The struct (structure) is like a class in C# that is used to store data. However, unlike classes, a struct is a value type.

Suppose we want to store the name and age of a person. We can create two variables: name and age and store value.

However, suppose we want to store the same information of multiple people.

In this case, creating variables for an individual person might be a tedious task. To overcome this we can create a struct that stores name and age. Now, this struct can be used for every person.

Define struct in C#

In C#, we use the struct keyword to define a struct. For example,

```
struct Employee {
    public int id;
}
```

Here, id is a field inside the struct. A struct can include methods, indexers, etc as well.

Declare struct variable

Before we use a struct, we first need to create a struct variable. We use a struct name with a variable to declare a struct variable. For example,

```
struct Employee {  
    public int id;  
}  
...
```

```
// declare emp of struct Employee
```

```
Employee emp;
```

In the above example, we have created a struct named Employee. Here, we have declared a variable emp of the struct Employee.

Access C# struct

We use the struct variable along with the . operator to access members of a struct. For example,

```
struct Employee {  
    public int id;  
}  
...
```

```
// declare emp of struct Employee
```

```
Employee emp;
```

```
// access member of struct
```

```
emp.id = 1;
```

Here, we have used variable emp of a struct Employee with . operator to access members of the Employee.

```
emp.id = 1;
```

This accesses the id field of struct Employee.

Note: Primitive data types like int, bool, float are pre-defined structs in C#.

Example: C# Struct

```
using System;
```

```
namespace CsharpStruct {
```

```
// defining struct
```

```
struct Employee {
```

```
    public int id;
```

```
    public void getId(int id) {
```

```
        Console.WriteLine("Employee Id: " + id);
```

```
    }
```

```
}
```

```
class Program {  
    static void Main(string[] args) {  
  
        // declare emp of struct Employee  
  
        Employee emp;  
  
        // accesses and sets struct field  
        emp.id = 1;  
  
        // accesses struct methods  
        emp.getId(emp.id);  
  
        Console.ReadLine();  
    }  
}  
}
```

Output

Employee Id: 1

In the above program, we have created a struct named Employee. It contains a field id and a method getId().

C# Struct Example

Let's see a simple example of struct Rectangle which has two data members width and height.

```
1. using System;
2. public struct Rectangle
3. {
4.     public int width, height;
5.
6. }
7. public class TestStructs
8. {
9.     public static void Main()
10.    {
11.        Rectangle r = new Rectangle();
12.        r.width = 4;
13.        r.height = 5;
14.        Console.WriteLine("Area of Rectangle is: " + (r.width * r.height));
15.    }
16. }
```

Output:

X

Area of Rectangle is: 20

C# Struct Example: Using Constructor and Method

Let's see another example of struct where we are using constructor to initialize data and method to calculate area of rectangle.

```
1. using System;
2. public struct Rectangle
3. {
4.     public int width, height;
5.
6.     public Rectangle(int w, int h)
7.     {
8.         width = w;
```

```

9.     height = h;
10.  }
11.  public void areaOfRectangle() {
12.      Console.WriteLine("Area of Rectangle is: "+(width*height)); }
13.  }
14. public class TestStructs
15. {
16.     public static void Main()
17.     {
18.         Rectangle r = new Rectangle(5, 6);
19.         r.areaOfRectangle();
20.     }
21. }

```

Output:

Area of Rectangle is: 30

Note: Struct doesn't support inheritance. But it can implement interfaces.

C# static

In C#, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C#, static can be field, method, constructor, class, properties, operator and event.

Note: Indexers and destructors cannot be static.

Advantage of C# static keyword

Memory efficient: Now we don't need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

C# Static Field

A field which is declared as static, is called static field. Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects.

It is used to refer the common property of all objects such as `rateOfInterest` in case of `Account`, `companyName` in case of `Employee` etc.

C# static field example

Let's see the simple example of static field in C#.

```
1. using System;
2. public class Account
3. {
4.     public int accno;
5.     public String name;
6.     public static float rateOfInterest=8.8f;
7.     public Account(int accno, String name)
8.     {
9.         this.accno = accno;
10.        this.name = name;
11.    }
12.
13.    public void display()
14.    {
15.        Console.WriteLine(accno + " " + name + " " + rateOfInterest);
16.    }
17. }
18. class TestAccount{
19.     public static void Main(string[] args)
20.     {
21.         Account a1 = new Account(101, "Sonoo");
22.         Account a2 = new Account(102, "Mahesh");
23.         a1.display();
24.         a2.display();
25.
26.     }
27. }
```

Output:

101 Sonoo 8.8

102 Mahesh 8.8

C# static field example 2: changing static field

If you change the value of static field, it will be applied to all the objects.

```
1. using System;
2.     public class Account
3.     {
4.         public int accno;
5.         public String name;
6.         public static float rateOfInterest=8.8f;
7.         public Account(int accno, String name)
8.         {
9.             this.accno = accno;
10.            this.name = name;
11.        }
12.
13.        public void display()
14.        {
15.            Console.WriteLine(accno + " " + name + " " + rateOfInterest);
16.        }
17.    }
18.    class TestAccount{
19.        public static void Main(string[] args)
20.        {
21.            Account.rateOfInterest = 10.5f;//changing value
22.            Account a1 = new Account(101, "Sonoo");
23.            Account a2 = new Account(102, "Mahesh");
24.            a1.display();
25.            a2.display();
26.
27.        }
28.    }
```

Output:

101 Sonoo 10.5

102 Mahesh 10.5

C# static field example 3: Counting Objects

Let's see another example of static keyword in C# which counts the objects.

```
1. using System;
2. public class Account
3. {
4.     public int accno;
5.     public String name;
6.     public static int count=0;
7.     public Account(int accno, String name)
8.     {
9.         this.accno = accno;
10.        this.name = name;
11.        count++;
12.    }
13.
14.    public void display()
15.    {
16.        Console.WriteLine(accno + " " + name);
17.    }
18. }
19. class TestAccount{
20.     public static void Main(string[] args)
21.     {
22.         Account a1 = new Account(101, "Sonoo");
23.         Account a2 = new Account(102, "Mahesh");
24.         Account a3 = new Account(103, "Ajeet");
25.         a1.display();
26.         a2.display();
27.         a3.display();
28.         Console.WriteLine("Total Objects are: "+Account.count);
29.     }
30. }
```

Output:

101 Sonoo

102 Mahesh

103 Ajeet

Total Objects are: 3

C# static class

The C# static class is like the normal class but it cannot be instantiated. It can have only static members. The advantage of static class is that it provides you guarantee that instance of static class cannot be created.

Points to remember for C# static class

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

C# static class example

Let's see the example of static class that contains static field and static method.

```
1. using System;
2.     public static class MyMath
3.     {
4.         public static float PI=3.14f;
5.         public static int cube(int n){return n*n*n;}
6.     }
7.     class TestMyMath{
8.         public static void Main(string[] args)
9.         {
10.             Console.WriteLine("Value of PI is: "+MyMath.PI);
11.             Console.WriteLine("Cube of 3 is: " + MyMath.cube(3));
12.         }
```

```
13. }
```

Output:

Value of PI is: 3.14

Cube of 3 is: 27

C# static constructor

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

Points to remember for C# Static Constructor

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

C# Static Constructor example

Let's see the example of static constructor which initializes the static field `rateOfInterest` in `Account` class.

```
1. using System;
2. public class Account
3. {
4.     public int id;
5.     public String name;
6.     public static float rateOfInterest;
7.     public Account(int id, String name)
8.     {
9.         this.id = id;
10.        this.name = name;
11.    }
```

```
12.     static Account()
13.     {
14.         rateOfInterest = 9.5f;
15.     }
16.     public void display()
17.     {
18.         Console.WriteLine(id + " " + name+" "+rateOfInterest);
19.     }
20. }
21. class TestEmployee{
22.     public static void Main(string[] args)
23.     {
24.         Account a1 = new Account(101, "Sonoo");
25.         Account a2 = new Account(102, "Mahesh");
26.         a1.display();
27.         a2.display();
28.
29.     }
30. }
```

Output:

101 Sonoo 9.5

102 Mahesh 9.5

C# Constructor

In C#, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The

constructor in C# has the same name as class or struct.

There can be two types of constructors in C#.

- **Default constructor**
- **Parameterized constructor**

C# Default Constructor

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

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

```
1.using System;  
2. public class Employee  
3. {  
4.     public Employee()  
5.     {  
6.         Console.WriteLine("Default  
   Constructor Invoked");  
7.     }  
8.     public static void Main(string[] args)
```

```
9.      {  
10.          Employee e1 = new Employee();  
11.          Employee e2 = new Employee();  
12.      }  
13.  }
```

Output:

Default Constructor Invoked

Default Constructor Invoked

C# Default Constructor Example: Having Main() in another class

Let's see another example of default constructor where we are having Main() method in another class.

```
1.using System;  
2.  public class Employee  
3.  {  
4.      public Employee()
```

```
5.    {
6.        Console.WriteLine("Default
    Constructor Invoked");
7.    }
8. }
9. class TestEmployee{
10.     public static void Main(string[] args)
11.     {
12.         Employee e1 = new Employee();
13.         Employee e2 = new Employee();
14.     }
15. }
```

Output:

Default Constructor Invoked

Default Constructor Invoked

C# Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

```
1.using System;  
2. public class Employee  
3. {  
4.     public int id;  
5.     public String name;  
6.     public float salary;  
7.     public Employee(int i, String n,float s)  
8.     {  
9.         id = i;  
10.        name = n;  
11.        salary = s;  
12.    }  
13.    public void display()  
14.    {  
15.        Console.WriteLine(id + " " +  
        name+" "+salary);
```

```
16.     }
17. }
18. class TestEmployee{
19.     public static void Main(string[] args)
20.     {
21.         Employee e1 = new Employee(101,
22.             "Sonoo", 890000f);
23.         Employee e2 = new Employee(102,
24.             "Mahesh", 490000f);
25.         e1.display();
26.         e2.display();
27.     }
28. }
```

Output:

101 Sonoo 890000

102 Mahesh 490000

Destructor (Finalizer)

Destroyers are also known as Finalizers. A destructor is a very special member function of a class that is executed whenever an object of its class goes out of scope.

Destroyer is used to write the code that needs to be executed while an instance is destroyed i.e garbage collection process.

A destructor is a method with the same name as the name of the class but starting with the

character tilde (~) character. A destructor runs only after a class becomes unreachable.

Destructor Fundamental Points

- 1. Destructor has the same name as the class name with the "~" sign. The tilde sign distinguished it from the constructor.**
- 2. A destructor cannot be called explicitly, they invoked automatically after GC.**
- 3. A C# destructor automatically calls the destructor of its base class.**
- 4. Unlike constructors, the destructor of the child class called before the parent class, i.e.**

the destructors are called in the reverse order of the constructor calls.

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



```
16.      Employee e1 = new Employee();
17.      Employee e2 = new Employee();
18.  }
```

```
19.  }
```

Output:

Constructor Invoked

Constructor Invoked

Destructor Invoked

Destructor Invoked

C# this

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

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

C# this example

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

```
1. using System;
2.     public class Employee
3.     {
4.         public int id;
5.         public String name;
6.         public float salary;
7.         public Employee(int id, String name,float salary)
8.         {
9.             this.id = id;
10.            this.name = name;
11.            this.salary = salary;
12.        }
13.        public void display()
14.        {
15.            Console.WriteLine(id + " " + name+" "+salary);
16.        }
17.    }
18.    class TestEmployee{
19.        public static void Main(string[] args)
20.        {
21.            Employee e1 = new Employee(101, "Sonoo", 890000f);
22.            Employee e2 = new Employee(102, "Mahesh", 490000f);
23.            e1.display();
24.            e2.display();
25.
26.        }
27.    }
```

Output:

101 Sonoo 890000

102 Mahesh 490000

C# Inheritance

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

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

Advantage of C# Inheritance

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

C# Single Level Inheritance Example: Inheriting Fields

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

```
1. using System;
2.     public class Employee
3.     {
4.         public float salary = 40000;
5.     }
6.     public class Programmer: Employee
7.     {
8.         public float bonus = 10000;
9.     }
```

```
10. class TestInheritance{
11.     public static void Main(string[] args)
12.     {
13.         Programmer p1 = new Programmer();
14.
15.         Console.WriteLine("Salary: " + p1.salary);
16.         Console.WriteLine("Bonus: " + p1.bonus);
17.
18.     }
19. }
```

Output:

Salary: 40000

Bonus: 10000

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

C# Single Level Inheritance Example: Inheriting Methods

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

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

Output:

Eating...

Barking...

C# Multi Level Inheritance Example

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

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

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

Output:

Eating...

Barking...

Weeping...

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

C# Aggregation (HAS-A Relationship)

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

C# Aggregation Example

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

```
1. using System;
2. public class Address
3. {
4.     public string addressLine, city, state;
5.     public Address(string addressLine, string city, string state)
6.     {
7.         this.addressLine = addressLine;
8.         this.city = city;
9.         this.state = state;
10.    }
11. }
12. public class Employee
13. {
14.     public int id;
15.     public string name;
16.     public Address address;//Employee HAS-A Address
17.     public Employee(int id, string name, Address address)
18.     {
19.         this.id = id;
20.         this.name = name;
21.         this.address = address;
```

```

22.     }
23.     public void display()
24.     {
25.         Console.WriteLine(id + " " + name + " " +
26.             address.addressLine + " " + address.city + " " + address.state);
27.     }
28. }
29. public class TestAggregation
30. {
31.     public static void Main(string[] args)
32.     {
33.         Address a1=new Address("G-13, Sec-3","Noida","UP");
34.         Employee e1 = new Employee(1,"Sonoo",a1);
35.         e1.display();
36.     }
37. }

```

Output:

1 Sonoo G-13 Sec-3 Noida UP

C# Base

In C#, base keyword is used to access fields, constructors and methods of base class.

You can use base keyword within instance method, constructor or instance property accessor only. You can't use it inside the static method.

C# base keyword: accessing base class field

We can use the base keyword to access the fields of the base class within derived class. It is useful if base and derived classes have the same fields. If derived class doesn't define same field, there is no need to use base keyword. Base class field can be directly accessed by the derived class.

Let's see the simple example of base keyword in C# which accesses the field of base class.

```

1. using System;
2. public class Animal{
3.     public string color = "white";
4. }
5. public class Dog: Animal
6. {
7.     string color = "black";
8.     public void showColor()
9.     {
10.         Console.WriteLine(base.color);
11.         Console.WriteLine(color);
12.     }
13.
14. }
15. public class TestBase
16. {
17.     public static void Main()
18.     {
19.         Dog d = new Dog();
20.         d.showColor();
21.     }
22. }

```

Output:

white

black

C# base keyword example: calling base class method

By the help of base keyword, we can call the base class method also. It is useful if base and derived classes defines same method. In other words, if method is overridden. If derived class doesn't define same method, there is no need to use base keyword. Base class method can be directly called by the derived class method.

Let's see the simple example of base keyword which calls the method of base class.

```

1. using System;
2. public class Animal{

```



```

3.     public virtual void eat(){
4.         Console.WriteLine("eating...");
5.     }
6. }
7. public class Dog: Animal
8. {
9.     public override void eat()
10.    {
11.        base.eat();
12.        Console.WriteLine("eating bread...");
13.    }
14.
15. }
16. public class TestBase
17. {
18.     public static void Main()
19.     {
20.         Dog d = new Dog();
21.         d.eat();
22.     }
23. }

```

Output:

eating...

eating bread...

C# inheritance: calling base class constructor internally

Whenever you inherit the base class, base class constructor is internally invoked. Let's see the example of calling base constructor.

```

1. using System;
2. public class Animal{
3.     public Animal(){
4.         Console.WriteLine("animal...");
5.     }
6. }

```

```
7. public class Dog: Animal
8. {
9.     public Dog()
10.    {
11.        Console.WriteLine("dog...");
12.    }
13.
14. }
15. public class TestOverriding
16. {
17.     public static void Main()
18.     {
19.         Dog d = new Dog();
20.
21.     }
22. }
```

Output:

animal...

dog...

Understanding virtual, override and new keyword in C#

Polymorphism is one of the main aspects of OOPS Principles which include method overriding and method overloading. Virtual and Override keywords are used for method overriding and the new keyword is used for method hiding. In this article, I am going to explain each keyword in details with the help of C# code.

Simple Class Inheritance

Consider the below class hierarchy with classes A, B, and C. A is the super/base class, B is derived from class A and C is derived from class B.

If a method Test() is declared in the base class A and classes B or C has no methods as shown below.

```
using System;

namespace Polymorphism

{

    class A

    {

        public void Test() { Console.WriteLine("A::Test()"); }

    }

    class B : A { }

    class C : B { }

    class Program

    {

        static void Main(string[] args)

        {

            A a = new A();

            a.Test(); // output --> "A::Test()"

            B b = new B();

            b.Test(); // output --> "A::Test()"

            C c = new C();

            c.Test(); // output --> "A::Test()"

        }

    }

}
```

```
Console.ReadKey();
```

```
}
```

```
}
```

```
}
```

Suppose you have Test() method in all the classes A, B, C as shown below:

```
using System;
```

```
namespace Polymorphism
```

```
{
```

```
class A
```

```
{
```

```
public void Test() { Console.WriteLine("A::Test()"); }
```

```
}
```

```
class B : A
```

```
{
```

```
public void Test() { Console.WriteLine("B::Test()"); }
```

```
}
```

```
class C : B
```

```
{
```

```
public void Test() { Console.WriteLine("C::Test()"); }
```

```
}
```

```
class Program
```

```

{
    static void Main(string[] args)
    {

        A a = new A();

        B b = new B();

        C c = new C();

        a.Test(); // output --> "A::Test()"
        b.Test(); // output --> "B::Test()"
        c.Test(); // output --> "C::Test()"

        a = new B();

        a.Test(); // output --> "A::Test()"

        b = new C();

        b.Test(); // output --> "B::Test()"

        Console.ReadKey();

    }

}

```

When you will run the above program, it will run successfully and give the O/P. But this program will show the two warnings as shown below:

1. 'Polymorphism.B.Test()' hides inherited member 'Polymorphism.A.Test()'. Use the new keyword if hiding was intended.

2. 'Polymorphism.C.Test()' hides inherited member 'Polymorphism.B.Test()'. Use the new keyword if hiding was intended.

Method Hiding (new keyword)

As you have seen in the above example the compiler generates the warnings since C# also supports method hiding. For hiding the base class method from the derived class simply declare the derived class method with a new keyword. Hence above code can be re-written as :

```
using System;

namespace Polymorphism
{
    class A
    {
        public void Test() { Console.WriteLine("A::Test()"); }
    }

    class B : A
    {
        public new void Test() { Console.WriteLine("B::Test()"); }
    }

    class C : B
    {
        public new void Test() { Console.WriteLine("C::Test()"); }
    }

    class Program
```

```

{

static void Main(string[] args)

{

A a = new A();

B b = new B();

C c = new C();


a.Test(); // output --> "A::Test()"

b.Test(); // output --> "B::Test()"

c.Test(); // output --> "C::Test()"


a = new B();

a.Test(); // output --> "A::Test()"


b = new C();

b.Test(); // output --> "B::Test()"


Console.ReadKey();

}

}

}

```

Moreover, if you are expecting the fourth and fifth output should be "B:: Test ()" and "C:: Test ()" since the objects a and b are referenced by the object of B and C respectively then you have to re-write the above code for Method Overriding.

Method Overriding (virtual and override keyword)

In C#, for overriding the base class method in a derived class, you have to declare a base class method as virtual and derived class method as override shown below:

```
using System;

namespace Polymorphism
{
    class A
    {
        public virtual void Test() { Console.WriteLine("A::Test()"); }
    }

    class B : A
    {
        public override void Test() { Console.WriteLine("B::Test()"); }
    }

    class C : B
    {
        public override void Test() { Console.WriteLine("C::Test()"); }
    }

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



```
A a = new A();

B b = new B();

C c = new C();

a.Test(); // output --> "A::Test()"

b.Test(); // output --> "B::Test()"

c.Test(); // output --> "C::Test()"

a = new B();

a.Test(); // output --> "B::Test()"

b = new C();

b.Test(); // output --> "C::Test()"
```

```
Console.ReadKey();

}

}

}
```

Mixing Method Overriding and Method Hiding

You can also mix the method hiding and method overriding by using virtual and new keyword since the method of a derived class can be virtual and new at the same time. This is required when you want to further override the derived class method into the next level as I am overriding Class B, Test() method in Class C as shown below:

```
using System;

namespace Polymorphism

{

class A
```

```
{  
public void Test() { Console.WriteLine("A::Test()"); }  
}
```

```
class B : A  
{  
public new virtual void Test() { Console.WriteLine("B::Test()"); }  
}
```

```
class C : B  
{  
public override void Test() { Console.WriteLine("C::Test()"); }  
}
```

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

```
A a = new A();
```

```
B b = new B();
```

```
C c = new C();
```

```
a.Test(); // output --> "A::Test()"
```

```
b.Test(); // output --> "B::Test()"
```

```
c.Test(); // output --> "C::Test()"
```

```
a = new B();
```

```
a.Test(); // output --> "A::Test()"
```

```
b = new C();
```

```
b.Test(); // output --> "C::Test()"
```

```
Console.ReadKey();
```

```
}
```

```
}
```

```
}
```

Note

1. The virtual keyword is used to modify a method, property, indexer, or event declared in the base class and allow it to be overridden in the derived class.
2. The override keyword is used to extend or modify a virtual/abstract method, property, indexer, or event of base class into a derived class.
3. The new keyword is used to hide a method, property, indexer, or event of base class into derived class.

Coming to the C# perspective, the virtual keyword is used to modify the declaration of any property, method or event to allow

overriding in a derived class. In simple terms, the virtual keyword implements the method overriding concept in C#.

C# Sealed

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

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

C# Sealed class

C# sealed class cannot be derived by any class. Let's see an example of sealed class in C#.

```
1. using System;
2. sealed public class Animal{
3.     public void eat() { Console.WriteLine("eating..."); }
4. }
5. public class Dog: Animal
6. {
7.     public void bark() { Console.WriteLine("barking..."); }
8. }
9. public class TestSealed
10. {
11.     public static void Main()
12.     {
```

```
13.     Dog d = new Dog();
14.     d.eat();
15.     d.bark();
16.
17.
18. }
19. }
```

Output:

Compile Time Error: 'Dog': cannot derive from sealed type 'Animal'

C# Sealed method

The sealed method in C# cannot be overridden further. It must be used with override keyword in method.

Let's see an example of sealed method in C#.

```
1. using System;
2. public class Animal{
3.     public virtual void eat() { Console.WriteLine("eating..."); }
4.     public virtual void run() { Console.WriteLine("running..."); }
5.
6. }
7. public class Dog: Animal
8. {
9.     public override void eat() { Console.WriteLine("eating bread..."); }
10.    public sealed override void run() {
11.        Console.WriteLine("running very fast...");
12.    }
13. }
14. public class BabyDog : Dog
15. {
16.     public override void eat() { Console.WriteLine("eating biscuits..."); }
17.     public override void run() { Console.WriteLine("running slowly..."); }
18. }
19. public class TestSealed
20. {
```

```
21. public static void Main()
22. {
23.     BabyDog d = new BabyDog();
24.     d.eat();
25.     d.run();
26. }
27. }
```

Output:

Compile Time Error: 'BabyDog.run()': cannot override inherited member 'Dog.run()' because it is sealed

Note: Local variables can't be sealed.

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

Output:

Compile Time Error: Invalid expression term 'sealed'

C# Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation and polymorphism.

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

C# Runtime Polymorphism Example

Let's see a simple example of runtime polymorphism in C#.

```
1. using System;
2. public class Animal{
3.     public virtual void eat(){
4.         Console.WriteLine("eating...");
5.     }
6. }
7. public class Dog: Animal
8. {
9.     public override void eat()
10.    {
11.        Console.WriteLine("eating bread...");
12.    }
13.
14. }
15. public class TestPolymorphism
16. {
17.     public static void Main()
18.     {
19.         Animal a= new Dog();
20.         a.eat();
21.     }
22. }
```

Output:

```
eating bread...
```

C# Runtime Polymorphism Example 2

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

```
1. using System;
2. public class Shape{
3.     public virtual void draw(){
4.         Console.WriteLine("drawing...");
5.     }
6. }
7. public class Rectangle: Shape
8. {
9.     public override void draw()
10.    {
11.        Console.WriteLine("drawing rectangle...");
12.    }
13.
14. }
15. public class Circle : Shape
16. {
17.     public override void draw()
18.     {
19.         Console.WriteLine("drawing circle...");
20.     }
21.
22. }
23. public class TestPolymorphism
24. {
25.     public static void Main()
26.     {
27.         Shape s;
28.         s = new Shape();
29.         s.draw();
30.         s = new Rectangle();
31.         s.draw();
32.         s = new Circle();
33.         s.draw();
34.
35.     }
36. }
```

Output:


```
drawing...
```

```
drawing rectangle...
```

```
drawing circle...
```

Runtime Polymorphism with Data Members

Runtime Polymorphism can't be achieved by data members in C#. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

```
1. using System;
2. public class Animal{
3.     public string color = "white";
4.
5. }
6. public class Dog: Animal
7. {
8.     public string color = "black";
9. }
10. public class TestSealed
11. {
12.     public static void Main()
13.     {
14.         Animal d = new Dog();
15.         Console.WriteLine(d.color);
16.
17.     }
18. }
```

Output:

```
white
```

C# Abstract

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

1. Abstract class
2. Interface

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

Abstract Method

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

1. `public abstract void draw();`

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

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

X

C# Abstract class

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

Let's see an example of abstract class in C# which has one abstract method `draw()`. Its implementation is provided by derived classes: `Rectangle` and `Circle`. Both classes have different implementation.

1. `using System;`
2. `public abstract class Shape`
3. `{`
4. `public abstract void draw();`
5. `}`
6. `public class Rectangle : Shape`

```

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

```

Output:

drawing ractangle...

drawing circle...

C# Abstract Class Features

1. An abstract class can inherit from a class and one or more interfaces.
2. An abstract class can implement code with non-Abstract methods.
3. An Abstract class can have modifiers for methods, properties etc.
4. An Abstract class can have constants and fields.
5. An abstract class can implement a property.
6. An abstract class can have constructors or destructors.

7. An abstract class cannot be inherited from by structures.
8. An abstract class cannot support multiple inheritance.

Example 1

```
1. #region
2.
3. //An abstract class can inherit from a class and one or more interfaces.
4.
5. interface IVendorTransDetails
6. {
7.     void getVendorID();
8. }
9. interface IClaimsTracker
10. {
11.     void getSeqID();
12. }
13. class ClaimsMaster
14. {
15.     string getDCNNO()
16.     {
17.         return "PC20100308A00005";
18.     }
19. }
```

Example 2

```
1. abstract class Abstract : ClaimsMaster, IClaimsTracker, IVendorTransDetails
2. {
3.     //Here we should implement modifiers otherwise it throws compile-time error
4.     public void getVendorID()
5.     {
6.         int s = new int();
7.         s = 001;
8.         Console.Write(s);
9.     }
10.
11.     public void getSeqID()
12.     {
13.         int SeqID = new int();
14.         SeqID = 001;
```

```

15.     Console.Write(SeqID);
16. }
17. }
18. #endregion

```

Example 3

```

1.  #region
2.
3.  //An abstract class can implement code with non-Abstract methods.
4.
5.  abstract class NonAbstractMethod
6.  {
7.      //It is a Non-abstract method we should implement code into the non-abstract method
      on the class.
8.      public string getDcn()
9.      {
10.         return "PS20100301A0012";
11.     }
12.     public abstract void getSeqID();
13. }
14. class Utilize : NonAbstractMethod
15. {
16.     public override void getSeqID()
17.     {
18.     }
19.
20. }
21. #endregion

```

Example 4

```

1.  #region
2.  //Abstract class can have modifiers for methods,properties and An abstract class can
    implement a property
3.
4.  public abstract class abstractModifier
5.  {
6.      private int id;
7.      public int ID
8.      {

```

```

9.      get { return id; }
10.     set { id = value; }
11.    }
12.    internal abstract void Add();
13. }
14. #endregion

```

Example 5

```

1. #region
2. //Abstract class can have constant and fields
3. public abstract class ConstantFields
4. {
5.     public int no;
6.     private const int id = 10;
7. }
8. #endregion

```

Example 6

```

1. #region
2. //An abstract class can have constructors or destructors
3. abstract class ConsDes
4. {
5.     ConsDes()
6.     {
7.     }
8.     ~ConsDes()
9.     {
10.    }
11. }
12. #endregion

```

Example 7

```

1. #region
2. //An abstract class cannot be inherited from by structures
3. public struct test
4. {
5. }
6. //We can't inheritance the struct class on the abstract class
7. abstract class NotInheritanceStruct

```

```
8. {  
9. }  
10. #endregion
```

Example 8

```
1. #region  
2. //An abstract class cannot support multiple inheritance  
3. class A  
4. {  
5. }  
6. class B : A  
7. {  
8. }  
9. abstract class Container : B //But we can't inherit like this : A,B  
10. {  
11. }  
12. #endregion
```

C# Interface

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

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

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

C# interface example

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

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

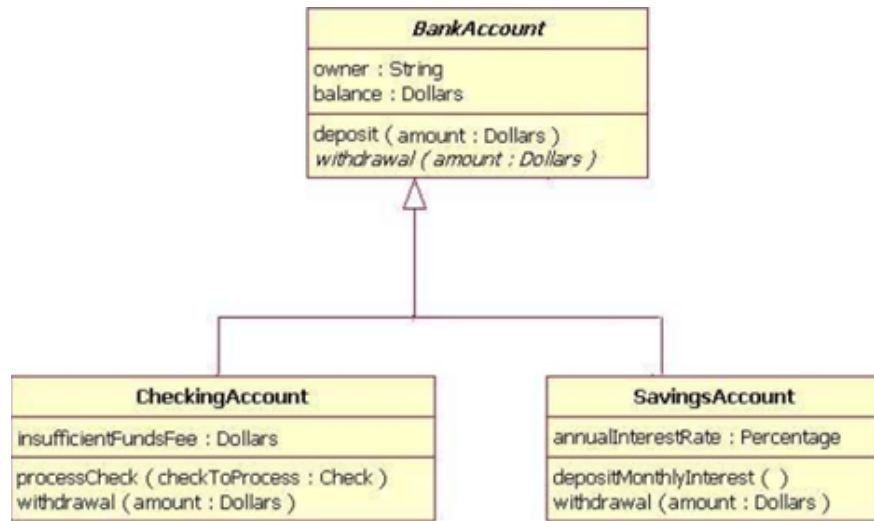
Output:

drawing ractangle...

drawing circle...

Lab On Inheritance, Abstract Classes

1. Implement the following inheritance:



2. Create a Class Polygon with three variables l,b,h. Create a parameterised method to set the values to l,b,h. Create an abstract method compute().

Create a subclass of Polygon as Cuboid and implement the method compute(). Create an another subclass of Polygon as Cube and implement compute(). Write a method to display the computed information in both the classes

3. Create a class A with the two variables x and y. Create a parameterised method to set the values. Write abstract methods sum() and display().

Create two subclasses B and C with a variable in each of the class. Create constructors in the subclasses. implement sum of three numbers in each class and display the information.

Create an abstract class A with an abstract method display(). Create a subclass B and implement display() function and print "this is it". Create a subclass C and override display() and print "this is in class c". Create a subclass D and override display() and print "this is in subclass d". Create a main function to call all the display methods.(Do not use super keyword, use Dynamic Method Dispatch or runtime polymorphism)

4. Create a class Library with the variables bno,bname and bprice. Set the values to the variables using a parameterised method. Create an abstract method updatePrice() and abstract method display().

Create a subclass ItLibrary with the variable deptname. Create a parameterised constructor to set the deptname. Create a getBookDetails() to display the information.

Create a main function to print this.

5. Create a class A with a method show(). Create a subclass B and implement show() method. Create a subclass C to the class B and implement show(). Create another subclass C and Implement show().

Create a main method to access all the show() methods.

(One solution is to use base.method() in all the subclasses

Other solution is do it the following in the main method.

Create a variable to the base class as A object1;

Create objects to all the classes A ob1=new A(),B ob2=new B(),C ob3=new C().

Now assign object1=ob1 and call object1.show().

Now assign object1=ob2 and call object1.show().

Now assign object1=ob3 and call object1.show().)

6. Create an abstract class Accounts with balance, accountnumber, accountholdername, address. Create abstract methods deposit() and withdraw(). Create a method display() to show the balance of the account number.

Create a subclass of this class SavingsAccount with a variable rateofinterest. Create claculateAmount(). Create display() to display the rate of interest with new balance and full account holder details

Create another subclass CurrentAccount with overdraftlimit. Crate a display() method to show overdraft limit along with the full account holder details.

Create object of the two classes and call their methods.

7. Create a program to implement virtual functions

Lab List on Interfaces

1. Create an interface called Vehicle with the methods setVehicle(int, String, String,double),display(). Create a subclass Veh with the members vehno,vehname,vehprice. Implement the interface to the class. Create three objects to the class.
2. Modify the program 1 to create necessary method in the Veh class which is useful for find out the total cost of three vehicles.
3. Create an interface called Calculator with abstract methods int add(int,int), int subtract(int,int), int multiply(int,int), int divide(int, int).

Create a class Calc which implements Calculator and implement all the methods in it.

4. create an interface A with the methods sum(int,int), mul(double,double,double). Create a subclass B which implements only sum(int,int). Create a subclass C which implements mul(double,double,double). Display the sum and multiplication values.
5. Create a class which can take the variables in various access specifiers(public, protected, private, default). Access these variables in a class, in a subclass, other package. Observer the results.
6. Create an interface Sort with the methods: getData(), doSort(), displaySort(). Implement this interface to a class which implements all the three methods.
7. Create an interface with a constant pi 3.14 and a method double volume() and setDim(double). Create a subclass to find the volume of a sphere.

Delegates

A delegate is an object which refers to a method or you can say it is a reference type variable that can hold a reference to the methods. It provides a way which tells which method is to be called when an event is triggered.

For example, if you click on a *Button* on a form (Windows Form application), the program would call a specific method. In simple words, it is a type that represents references to methods with a particular parameter list and return type and then calls the method in a program for execution when it is needed.

Important Points About Delegates:

- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.
- Delegates can be chained together; for example, multiple methods can be called on a single event.

Delegate is one of the base types in .NET. Delegate is a class, which is used to create and invoke delegates at runtime.

A delegate in C# is similar to a function pointer in C or C++.

delegate just contains the details of a method.

2. Why do we need delegates in C#?

Programmers often need to pass a method as a parameter of other methods. For this purpose we create and use delegates.

Example:

```
1. public delegate int DelegateMethod(int x, int y);
```

Any method that matches the delegate's signature, which consists of the return type and parameters, can be assigned to the delegate.

What are the benefits of delegates?

Delegates make event handling simple and easy.

What are types of delegates in C#?

There are two types of delegates, singlecast delegates, and multicast delegates.

Singlecast delegate

Singlecast delegates point to a single method at a time. In this the delegate is assigned to a single method at a time. They are derived from `System.Delegate` class.

Multicast Delegate

When a delegate is wrapped with more than one method that is known as a multicast delegate.

In C#, delegates are multicast, which means that they can point to more than one function at a time. They are derived from System.MulticastDelegate class.

How to define a delegates in C#?

There are three steps in defining and using delegates:

1. Declaration

To create a delegate, you use the delegate keyword.

1. **[attributes] [modifiers] delegate Return Type Name ([formal-parameters]);**
 - The attributes factor can be a normal C# attribute.
 - The modifier can be one or an appropriate combination of the following keywords: new, public, private, protected, or internal.
 - The Return Type can be any of the data types we have used so far. It can also be a type void or the name of a class.
 - The Name must be a valid C# name.

Because a delegate is a definituon for a method, you must use parentheses, required for every method. If this method will not take any argument, leave the parentheses empty.

Example:

1. **public delegate void DelegateExample();**

The above code is how a delegate with no papameters is defined.

2. Instantiation

1. **DelegateExample d1 = new DelegateExample(Display);**

The above code shows how a delegate is initiated.

3. *Invocation*

```
1. d1();
```

The above code piece invokes a delegate d1().

6. What is a Singlecast delegate in C#?

Here is a sample code that demonstrates how to create and use a singlecast delegate.

```
1. using System;
2. namespace ConsoleApplication5
3. {
4.     class Program
5.     {
6.         public delegate void delmethod();
7.
8.         public class P
9.         {
10.
11.             public static void display()
12.             {
13.                 Console.WriteLine("Hello!");
14.             }
15.
16.             public static void show()
17.             {
18.                 Console.WriteLine("Hi!");
19.             }
20.
21.             public void print()
22.             {
23.                 Console.WriteLine("Print");
24.             }

```

```

25.
26. }
27.
28. static void Main(string[] args)
29. {
30. // here we have assigned static method show() of class P to delegate delmethod()
31. delmethod del1 = P.show;
32.
33. // here we have assigned static method display() of class P to delegate delmethod() using
    new operator
34. // you can use both ways to assign the delegate
35. delmethod del2 = new delmethod(P.display);
36. P obj = new P();
37.
38. // here first we have create instance of class P and assigned the method print() to the
    delegate i.e. delegate with class
39. delmethod del3 = obj.print;
40.
41. del1();
42. del2();
43. del3();
44. Console.ReadLine();
45. }
46. }
47. }

```

What is a Multicast delegate in C#?

Multicasting of a Delegate

Multicasting of delegate is an extension of the normal delegate(sometimes termed as Single Cast Delegate). It helps the user to point more than one method in a single call.

Properties:

- Delegates are combined and when you call a delegate then a complete list of methods is called.
- All methods are called in First in First Out(FIFO) order.
- '+' or '+=' Operator is used to add the methods to delegates.

- ‘-’ or ‘-=’ Operator is used to remove the methods from the delegates list.

Here is sample code that demonstrates how to create and use a multicast delegate.

```
1. using System;
2. namespace delegate_Example4
3. {
4.
5. class Program
6. {
7. public delegate void delmethod(int x, int y);
8.
9. public class TestMultipleDelegate
10. {
11. public void plus_Method1(int x, int y)
12. {
13. Console.WriteLine("You are in plus_Method");
14. Console.WriteLine(x + y);
15. }
16.
17. public void subtract_Method2(int x, int y)
18. {
19. Console.WriteLine("You are in subtract_Method");
20. Console.WriteLine(x - y);
21. }
22. }
23.
24. static void Main(string[] args)
25. {
26.
27. TestMultipleDelegate obj = new TestMultipleDelegate();
28. delmethod del = new delmethod(obj.plus_Method1);
29.
30. // Here we have multicast
31. del += new delmethod(obj.subtract_Method2);
32. // plus_Method1 and subtract_Method2 are called
33. del(50, 10);
34. Console.WriteLine();
35. //Here again we have multicast
36. del -= new delmethod(obj.plus_Method1);
37. //Only subtract_Method2 is called
```

```
38. del(20, 10);
39. Console.ReadLine();
40. }
41. }
42. }
```

Example:

```
1. using System;
2. delegate int Calculator(int n); //declaring delegate
3.
4. public class DelegateExample
5. {
6.     static int number = 100;
7.     public static int add(int n)
8.     {
9.         number = number + n;
10.        return number;
11.    }
12.    public static int mul(int n)
13.    {
14.        number = number * n;
15.        return number;
16.    }
17.    public static int getNumber()
18.    {
19.        return number;
20.    }
21.    public static void Main(string[] args)
22.    {
23.        Calculator c1 = new Calculator(add); //instantiating delegate
24.        Calculator c2 = new Calculator(mul);
25.        c1(20); //calling method using delegate
26.        Console.WriteLine("After c1 delegate, Number is: " + getNumber());
27.        c2(3);
28.        Console.WriteLine("After c2 delegate, Number is: " + getNumber());
29.
30.    }
31. }
```

Output:

After c1 delegate, Number is: 120

After c2 delegate, Number is: 360

Example 2:

```
// C# program to illustrate the use of Delegates

using System;

namespace GeeksForGeeks {

    class Geeks {

        public delegate void addnum(int a, int b);

        public delegate void subnum(int a, int b);

        public void sum(int a, int b)
        {
            Console.WriteLine("(100 + 40) = {0}", a + b);
        }

        public void subtract(int a, int b)
        {
            Console.WriteLine("(100 - 60) = {0}", a - b);
        }

        // Main Method

        public static void Main(String []args)
        {
```

```

// creating object "obj" of class "Geeks"

Geeks obj = new Geeks();

addnum del_obj1 = new addnum(obj.sum);

subnum del_obj2 = new subnum(obj.subtract);

// pass the values to the methods by delegate object

del_obj1(100, 40);

del_obj2(100, 60);

// These can be written as using

// "Invoke" method

// del_obj1.Invoke(100, 40);

// del_obj2.Invoke(100, 60);

}

}

}

```

Output:

(100 + 40) = 140

(100 - 60) = 40

Example:

```
// C# program to illustrate the
// Multicasting of Delegates
using System;

class rectangle {

    // declaring delegate
    public delegate void rectDelegate(double height,
                                      double width);

    // "area" method
    public void area(double height, double width)
    {
        Console.WriteLine("Area is: {0}", (width * height));
    }

    // "perimeter" method
    public void perimeter(double height, double width)
    {
        Console.WriteLine("Perimeter is: {0} ", 2 * (width + height));
    }

    // Main Method
    public static void Main(String []args)
    {

        rectangle rect = new rectangle();

        rectDelegate rectdele = new rectDelegate(rect.area);

        // also can be written as
        // rectDelegate rectdele = rect.area;

        // call 2nd method "perimeter"
        // Multicasting
        rectdele += rect.perimeter;

        // pass the values in two method
```

```
        // by using "Invoke" method
        rectdele.Invoke(6.3, 4.2);
        Console.WriteLine();

        // call the methods with
        // different values
        rectdele.Invoke(16.3, 10.3);
    }
}
```

Output:

Area is: 26.46

Perimeter is: 21

Area is: 167.89

Perimeter is: 53.2

using System;

delegate int NumberChanger(int n);

namespace DelegateAppl {

class TestDelegate {

static int num = 10;

public static int AddNum(int p) {

num += p;


```
        return num;
    }

    public static int MultNum(int q) {

        num *= q;

        return num;

    }

    public static int getNum() {

        return num;

    }

    static void Main(string[] args) {

        //create delegate instances

        NumberChanger nc1 = new NumberChanger(AddNum);

        NumberChanger nc2 = new NumberChanger(MultNum);


        //calling the methods using the delegate objects

        nc1(25);

        Console.WriteLine("Value of Num: {0}", getNum());

        nc2(5);

        Console.WriteLine("Value of Num: {0}", getNum());

        Console.ReadKey();

    }

}

}
```

When the above code is compiled and executed, it produces the following result –

Value of Num: 35

Value of Num: 175

```
using System;
```

```
delegate int NumberChanger(int n);
```

```
namespace DelegateAppl {
```

```
    class TestDelegate {
```

```
        static int num = 10;
```

```
        public static int AddNum(int p) {
```

```
            num += p;
```

```
            return num;
```

```
        }
```

```
        public static int MultNum(int q) {
```

```
            num *= q;
```

```
            return num;
```

```
        }
```

```
        public static int getNum() {
```

```
            return num;
```

```
        }
```

```
static void Main(string[] args) {  
    //create delegate instances  
  
    NumberChanger nc;  
  
    NumberChanger nc1 = new NumberChanger(AddNum);  
    NumberChanger nc2 = new NumberChanger(MultNum);  
  
  
    nc = nc1;  
    nc += nc2;  
  
  
    //calling multicast  
    nc(5);  
  
    Console.WriteLine("Value of Num: {0}", getNum());  
  
    Console.ReadKey();  
  
}  
  
}  
  
}
```

Lambda expressions

Lambda expressions are anonymous functions that contain expressions or sequence of operators. All lambda expressions use the lambda operator `=>`, The left side of the lambda operator

specifies the input parameters and the right side holds an expression

Expression Lambdas

Parameter => expression

Parameter-list => expression

Count => count + 2;

Sum => sum + 2;

n => n % 2 == 0

The lambda operator => divides a lambda expression into two parts. The left side is the input parameter and the right side is the lambda body.

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

```
public static class demo
```

```
{
```

```
    public static void Main()
```

```
    {
```

```
        List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 };
```

```

List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);

foreach (var num in evenNumbers)
{
    Console.Write("{0} ", num);
}

Console.WriteLine();

Console.Read();

}
}

```

```

1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. class Dog
5. {
6.     public string Name { get; set; }
7.     public int Age { get; set; }
8. }
9. class demo{
10.     static void Main()
11.     {
12.         List<Dog> dogs = new List<Dog>() {
13.             new Dog { Name = "Rex", Age = 4 },
14.             new Dog { Name = "Sean", Age = 0 },
15.             new Dog { Name = "Stacy", Age = 3 }
16.         };

```

```

17.     var names = dogs.Select(x => x.Name);
18.     foreach (var name in names)
19.     {
20.         Console.WriteLine(name);
21.
22.     }
23.     Console.Read();
24. }
25. }

```

Output

Rex

Sean

Stacy

Using Lambda Expressions with Anonymous Types

```

1.  using System;
2.  using System.Collections.Generic;
3.  using System.Linq;
4.  class Dog
5.  {
6.      public string Name { get; set; }
7.      public int Age { get; set; }
8.  }
9.  class demo{
10.     static void Main()
11.     {
12.         List<Dog> dogs = new List<Dog>() {
13.             new Dog { Name = "Rex", Age = 4 },
14.             new Dog { Name = "Sean", Age = 0 },
15.             new Dog { Name = "Stacy", Age = 3 }
16.         };
17.         var newDogsList = dogs.Select(x => new { Age = x.Age, FirstLetter = x.Name[0] });
18.         foreach (var item in newDogsList)
19.         {
20.             Console.WriteLine(item);
21.         }

```

```
22. Console.Read();
23. }
24. }
```

Output

{ Age = 4, FirstLetter = R }

{ Age = 0, FirstLetter = S }

{ Age = 3, FirstLetter = S }

Sorting using a lambda expression

The following is an example of sorting with a lambda expression:

```
1. var sortedDogs = dogs.OrderByDescending(x => x.Age);
2. foreach (var dog in sortedDogs)
3. {
4.     Console.WriteLine(string.Format("Dog {0} is {1} years old.", dog.Name, dog.Age));
5. }
```

Output

Dog Rex is 4 years old.

Dog Stacy is 3 years old.

Dog Sean is 0 years old.

Evaluating Expressions using Lambda Expressions

using System;

using System.Collections.Generic;

using System.Linq;

class Dog

{

```

    public string Name { get; set; }

    public int Age { get; set; }
}

class demo{

    static void Main()

    {

        List<Dog> dogs = new List<Dog>() {

            new Dog { Name = "Rex", Age = 4 },

            new Dog { Name = "Sean", Age = 0 },

            new Dog { Name = "Stacy", Age = 3 }

        };

        var newDogsList = dogs.Select(x => new { Age = x.Age, FirstLetter = x.Name[0] });

        foreach (var item in newDogsList)

        {

            Console.WriteLine(item);

        }

        Console.Read();

    }

}

```

Output:

```

{ Age = 4, FirstLetter = R }

{ Age = 0, FirstLetter = S }

{ Age = 3, FirstLetter = S }

```


The anonymous type items in the newly constructed collection **newDogsList** have the values *Age* and **FirstLetter** as Arguments.

How to use Attributes with C# Lambda Expressions?

A Lambda Expression and its parameters can now have properties starting with **C# 10**. The example below demonstrates how to add characteristics to a Lambda Expression:

```
Func<string, int> parse = [Example(1)] (s) => int.Parse(s);
```

```
var choose = [Example(2)][Example(3)] object (bool b) => b ? 1 : "two";
```

As shown in the following example, you **may additionally add properties** to the Input Parameters or Return Value:

```
var sum = ([Example(1)] int a, [Example(2), Example(3)] int b) => a + b;
```

```
var inc = [return: Example(1)] (int s) => s++;
```

C# Exception Handling

Exception Handling in C# is a process to handle runtime errors. We perform exception handling so that normal flow of the application can be maintained even after runtime errors.

In C#, exception is an event or object which is thrown at runtime. All exceptions are derived from *System.Exception* class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Advantage

It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

C# Exception Classes

All the exception classes in C# are derived from **System.Exception** class. Let's see the list of C# common exception classes.

Exception	Description
System.DivideByZeroException	handles the error generated by dividing a number with zero.
System.NullReferenceException	handles the error generated by referencing the null object.
System.InvalidCastException	handles the error generated by invalid typecasting.
System.IO.IOException	handles the Input Output errors.
System.FieldAccessException	handles the error generated by invalid private or protected field access.

C# Exception Handling Keywords

In C#, we use 4 keywords to perform exception handling:

- try
- catch
- finally, and
- throw

C# try/catch

In C# programming, exception handling is performed by try/catch statement. The **try block** in C# is used to place the code that may throw exception. The **catch block** is used to handle the exception. The catch block must be preceded by try block.

C# example without try/catch

1. using System;

```

2. public class ExExample
3. {
4.     public static void Main(string[] args)
5.     {
6.         int a = 10;
7.         int b = 0;
8.         int x = a/b;
9.         Console.WriteLine("Rest of the code");
10.    }
11. }

```

Output:

```

Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.

```

C# try/catch example

```

1. using System;
2. public class ExExample
3. {
4.     public static void Main(string[] args)
5.     {
6.         try
7.         {
8.             int a = 10;
9.             int b = 0;
10.            int x = a / b;
11.        }
12.        catch (Exception e) { Console.WriteLine(e); }
13.
14.        Console.WriteLine("Rest of the code");
15.    }
16. }

```

Output:

```

System.DivideByZeroException: Attempted to divide by zero.

```

```

Rest of the code

```

C# finally

C# finally block is used to execute important code which is to be executed whether exception is handled or not. It must be preceded by catch or try block.

C# finally example if exception is handled

```
1. using System;
2. public class ExExample
3. {
4.     public static void Main(string[] args)
5.     {
6.         try
7.         {
8.             int a = 10;
9.             int b = 0;
10.            int x = a / b;
11.        }
12.        catch (Exception e) { Console.WriteLine(e); }
13.        finally { Console.WriteLine("Finally block is executed"); }
14.        Console.WriteLine("Rest of the code");
15.    }
16. }
```

Output:

```
System.DivideByZeroException: Attempted to divide by zero.
```

```
Finally block is executed
```

```
Rest of the code
```

C# finally example if exception is not handled

```
1. using System;
2. public class ExExample
3. {
4.     public static void Main(string[] args)
5.     {
6.         try
7.         {
8.             int a = 10;
9.             int b = 0;
10.            int x = a / b;
11.        }
```

```
12.     catch (NullReferenceException e) { Console.WriteLine(e); }
13.     finally { Console.WriteLine("Finally block is executed"); }
14.     Console.WriteLine("Rest of the code");
15. }
16. }
```

Output:

```
Unhandled Exception: System.DivideBy
```

C# User-Defined Exceptions

C# allows us to create user-defined or custom exception. It is used to make the meaningful exception. To do this, we need to inherit Exception class.

C# user-defined exception example

```
1. using System;
2. public class InvalidAgeException : Exception
3. {
4.     public InvalidAgeException(String message)
5.         : base(message)
6.     {
7.
8.     }
9. }
10. public class TestUserDefinedException
11. {
12.     static void validate(int age)
13.     {
14.         if (age < 18)
15.         {
16.             throw new InvalidAgeException("Sorry, Age must be greater than 18");
17.         }
18.     }
19.     public static void Main(string[] args)
20.     {
21.         try
22.         {
23.             validate(12);
24.         }
```

```
25.     catch (InvalidAgeException e) { Console.WriteLine(e); }
26.     Console.WriteLine("Rest of the code");
27. }
28. }
```

Output:

```
InvalidAgeException: Sorry, Age must be greater than 18
```

```
Rest of the code
```

C# Checked and Unchecked

C# provides checked and unchecked keyword to handle integral type exceptions. Checked and unchecked keywords specify checked context and unchecked context respectively. In checked context, arithmetic overflow raises an exception whereas, in an unchecked context, arithmetic overflow is ignored and result is truncated.

C# Checked

The checked keyword is used to explicitly check overflow and conversion of integral type values at compile time.

Let's first see an example that does not use checked keyword.

C# Checked Example without using checked

```
1. using System;
2. namespace CSharpProgram
3. {
4.     class Program
5.     {
6.         static void Main(string[] args)
7.         {
8.             int val = int.MaxValue;
9.             Console.WriteLine(val + 2);
10.        }
11.    }
12. }
```

Output:

-2147483647

See, the above program produces the wrong result and does not throw any overflow exception.

C# Checked Example using checked

This program throws an exception and stops program execution.

```
1. using System;
2. namespace CSharpProgram
3. {
4.     class Program
5.     {
6.         static void Main(string[] args)
7.         {
8.             checked
9.             {
10.                int val = int.MaxValue;
11.                Console.WriteLine(val + 2);
12.            }
13.        }
14.    }
15. }
```

Output:

Unhandled Exception: System.OverflowException: Arithmetic operation resulted in an overflow.

C# Unchecked

The Unchecked keyword ignores the integral type arithmetic exceptions. It does not check explicitly and produce result that may be truncated or wrong.

Example

```
1. using System;
2. namespace CSharpProgram
3. {
```

```
4.  class Program
5.  {
6.      static void Main(string[] args)
7.      {
8.          unchecked
9.          {
10.             int val = int.MaxValue;
11.             Console.WriteLine(val + 2);
12.         }
13.     }
14. }
15. }
```

Output:

-2147483647

DATABASE Links

<https://www.youtube.com/watch?v=tcmmCcMs8yU>

<https://www.youtube.com/watch?v=nh-fleqcds4>

code:

```
private void button1_Click(object sender, EventArgs e)
{
    this.Close();
}
```



```

private void button2_Click(object sender, EventArgs e)

{

    SqlConnection con = new SqlConnection(@"Data
Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename=C:\Users\itadmin\Documents\user
db.mdf;Integrated Security=True;Connect Timeout=30");

    SqlDataAdapter sda = new SqlDataAdapter("Select Count(*) From Login where
username='" + textBox1.Text + "' and password='" + textBox2.Text + "'",con);

    DataTable dt = new DataTable();

    sda.Fill(dt);

    if(dt.Rows[0][0].ToString() == "1")
    {

        this.Hide();

        Main ss = new Main();

        ss.Show();

    }
    else

    {

        MessageBox.Show(" Please Check Your Username and Password ");

    }

}

```

insert query link

<https://www.youtube.com/watch?v=dUntK3EGJcY>

<https://www.youtube.com/watch?v=aer8S1fFbNc>

code:

```
using System;  
using System.Collections.Generic;  
using System.ComponentModel;  
using System.Data;  
using System.Drawing;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
using System.Windows.Forms;  
using System.Data.SqlClient; // add this line
```

```
namespace Database
```

```
{
```

```
public partial class Form1 : Form
{
    //SqlCommand com = new SqlCommand();

    public Form1()
    {
        InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {

    }

    private void button1_Click(object sender, EventArgs e)
    {
        this.Close();
    }

    private void button2_Click(object sender, EventArgs e)
    {

        SqlConnection con = new SqlConnection(@"Data Source=(LocalDB)\MSSQLLocalDB;
AttachDbFilename=C:\Users\itadmin\Documents\userdb.mdf;Integrated Security=True;Connect
Timeout=30");
    }
}
```

```
SqlCommand com = new SqlCommand();

con.Open();

com.Connection = con;

com.CommandText = "Insert into Login(username,password)
values('" + textBox1.Text + "','" + textBox2.Text + "')";
```

```
com.ExecuteNonQuery();
```

```
con.Close();
```

```
MessageBox.Show(" inserted sucessfully ");
```

```
}
```

```
}
```

```
}
```

Dot Net Technologies

Employee Database

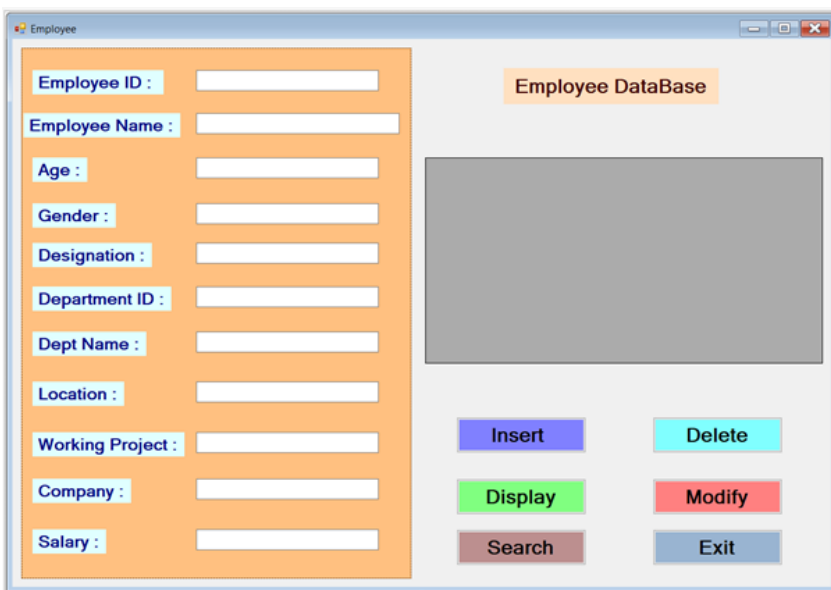
1. First Create a table with in the data base with below query

Query :

```
CREATE TABLE EmployeeData
(
    [Emp_ID] INT NOT NULL PRIMARY KEY
    DEFAULT 0,
    [Emp_Name] VARCHAR(50) NULL,
    [Age] INT NULL,
    [Gender] VARCHAR(50) NULL,
    [Designation] VARCHAR(50) NULL,
    [Dept_ID] INT NULL,
    [Dept_Name] VARCHAR(50) NULL,
    [Location] VARCHAR(50) NULL,
    [Work_Project] VARCHAR(50) NULL,
    [Company] VARCHAR(50) NULL,
    [Salary] FLOAT NULL
)
```

2. After creating the table then save it with name EmployeeData

3. Now, Create a windows form in the visual studio just like the below one



The image shows a Windows Form application titled "Employee". The form is divided into two main sections. On the left, there is a sidebar with an orange background containing eleven input fields, each with a label: "Employee ID :", "Employee Name :", "Age :", "Gender :", "Designation :", "Department ID :", "Dept Name :", "Location :", "Working Project :", "Company :", and "Salary :". On the right, the main area has a light gray background. At the top right of this area is a label "Employee DataBase". Below this label is a large, empty gray rectangular box. At the bottom of the main area, there are six buttons arranged in two columns: "Insert" (blue), "Delete" (cyan), "Display" (green), "Modify" (red), "Search" (brown), and "Exit" (gray).

4. After creating the windows form now, write the code for each button
5. The code is just below down there.

Code For Each button :

```
using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Data.SqlClient;

namespace Employee_CRUD
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            //Insert Button
        }
    }
}
```

```
SqlConnection con = new SqlConnection("Data
Source=(LocalDB)\\MSSQLLocalDB;AttachDbFilename=C:\\Users\\SHREEE\\OneDrive\\Documents\\MyDataBase.
mdf;Integrated Security=True;Connect Timeout=30");

con.Open();

SqlCommand cmd = new SqlCommand("insert into EmployeeData
values(@Emp_ID,@Emp_Name,@Age,@Gender,@Designation,@Dept_ID,@Dept_Name,@Location,@Work_Projec
t,@Company,@Salary)",con);

cmd.Parameters.AddWithValue("@Emp_ID", int.Parse(textBox1.Text));
cmd.Parameters.AddWithValue("@Emp_Name", textBox2.Text);
cmd.Parameters.AddWithValue("@Age", int.Parse(textBox3.Text));
cmd.Parameters.AddWithValue("@Gender", textBox4.Text);
cmd.Parameters.AddWithValue("@Designation", textBox5.Text);
cmd.Parameters.AddWithValue("@Dept_ID", int.Parse(textBox6.Text));
cmd.Parameters.AddWithValue("@Dept_Name", textBox7.Text);
cmd.Parameters.AddWithValue("@Location", textBox8.Text);
cmd.Parameters.AddWithValue("@Work_Project", textBox9.Text);
cmd.Parameters.AddWithValue("@Company", textBox10.Text);
cmd.Parameters.AddWithValue("@Salary", double.Parse(textBox11.Text));

cmd.ExecuteNonQuery();

con.Close();

MessageBox.Show(" Sucessfully Inserted the Data into Table.. ");

}
```



```

private void button4_Click(object sender, EventArgs e)

{

    // Update Button

    SqlConnection con = new SqlConnection("Data
Source=(LocalDB)\\MSSQLLocalDB;AttachDbFilename=C:\\Users\\SHREEE\\OneDrive\\Documents\\MyDataBase.
mdf;Integrated Security=True;Connect Timeout=30");

    con.Open();

    SqlCommand cmd = new SqlCommand();

    cmd.Connection = con;

    cmd.CommandText = "Update EmployeeData set Emp_Name=" + textBox2.Text + ", Age=" + textBox3.Text
+ ", Gender=" + textBox4.Text + ", Designation=" + textBox5.Text + ", Dept_ID=" + textBox6.Text + ",
Dept_Name=" + textBox7.Text + ", Location=" + textBox8.Text + ", Work_Project=" + textBox9.Text + ",
Company=" + textBox10.Text + ", Salary=" + textBox11.Text + " where Emp_ID=" + textBox1.Text + """;

    cmd.ExecuteNonQuery();

    con.Close();

    MessageBox.Show(" Sucessfully Updated the Data into Table");

}

private void button2_Click(object sender, EventArgs e)

{

    // Delete Button

    SqlConnection con = new SqlConnection("Data
Source=(LocalDB)\\MSSQLLocalDB;AttachDbFilename=C:\\Users\\SHREEE\\OneDrive\\Documents\\MyDataBase.
mdf;Integrated Security=True;Connect Timeout=30");

```

```
con.Open();
```

```
SqlCommand cmd = new SqlCommand("Delete EmployeeData where Emp_ID = @Emp_ID", con);
```

```
cmd.Parameters.AddWithValue("@Emp_ID", int.Parse(textBox1.Text));
```

```
cmd.ExecuteNonQuery();
```

```
con.Close();
```

```
MessageBox.Show(" Sucessfully Deleted the Data into Table.. ");
```

```
}
```

```
private void button3_Click(object sender, EventArgs e)
```

```
{
```

```
// Display Button
```

```
SqlConnection con = new SqlConnection("Data  
Source=(LocalDB)\\MSSQLLocalDB;AttachDbFilename=C:\\Users\\SHREEE\\OneDrive\\Documents\\MyDataBase.  
mdf;Integrated Security=True;Connect Timeout=30");
```

```
con.Open();
```

```
SqlCommand cmd = new SqlCommand("Select * from EmployeeData",con);
```

```
SqlDataAdapter da = new SqlDataAdapter(cmd);
```

```

        DataTable dt = new DataTable();

        da.Fill(dt);

        dataGridView1.DataSource = dt;

    }

    private void button5_Click(object sender, EventArgs e)
    {
        // Exit Button

        this.Close();
    }

    private void button6_Click(object sender, EventArgs e)
    {
        // Search Button

        SqlConnection con = new SqlConnection("Data
Source=(LocalDB)\\MSSQLLocalDB;AttachDbFilename=C:\\Users\\SHREEE\\OneDrive\\Documents\\MyDataBase.
mdf;Integrated Security=True;Connect Timeout=30");

        con.Open();

        SqlCommand cmd = new SqlCommand("Select * from EmployeeData where Emp_ID=@Emp_ID", con);

        cmd.Parameters.AddWithValue("Emp_ID", int.Parse(textBox1.Text));

```

```
SqlDataAdapter da = new SqlDataAdapter(cmd);  
  
DataTable dt = new DataTable();  
  
da.Fill(dt);  
  
dataGridView1.DataSource = dt;  
  
}  
  
}  
  
}
```

6. Make Sure That Use the Connection String of Your Database.

7. Save All Changes.

8. Now, Finally You can Run the Windows Form

Windows Form :

The screenshot shows a Windows application window titled "Employee". The window is divided into two main sections. The left section, with an orange background, contains a vertical list of input fields for employee data: "Employee ID :", "Employee Name :", "Age :", "Gender :", "Designation :", "Department ID :", "Dept Name :", "Location :", "Working Project :", "Company :", and "Salary :". Each label is in a light blue box, and each input field is a white rectangle. The right section, with a light gray background, is titled "Employee DataBase" in an orange box. Below the title is a large gray rectangular area, likely a placeholder for a data grid or list. At the bottom of the right section are six buttons arranged in two columns: "Insert" (blue), "Delete" (cyan), "Display" (green), "Modify" (red), "Search" (brown), and "Exit" (blue-gray).

Insert Query :

Employee

Employee ID : 999

Employee Name : Kolluri Mounav

Age : 20

Gender : Male

Designation : Student

Department ID : 110

Dept Name : Information Technology

Location : Kanuru

Working Project : Eye Cataract Detection

Company : VRSEC

Salary : 45000.00

Employee DataBase

	Emp_ID	Emp_Name	Age	Gender
▶	1	Ajay	22	Male
	2	Rizwan	19	Male
	4	M.J.N.V.Sai	20	Male
	5	Srinivas	150	Male
	999	Kolluri Mounav	20	Male
*				

Insert **Delete**

Display **Modify**

Search **Exit**

Modify Query :

Employee

Employee ID : 999

Employee Name : Mounav

Age : 20

Gender : Male

Designation : Manager

Department ID : 110

Dept Name : Information Technology

Location : Kanuru

Working Project : Eye Cataract Detection

Company : VRSEC

Salary : 45000.00

Employee DataBase

Emp_ID	Emp_Name	Age	Gender
1	Ajay	22	Male
2	Rizwan	19	Male
4	M.J.N.V Sai	20	Male
5	Srinivas	150	Male
999	Mounav	20	Male
*			

Insert **Delete**

Display **Modify**

Search **Exit**

Delete Query :

Employee

Employee ID : 2345

Employee Name : Mounav kolluri

Age : 20

Gender : Male

Designation : Manager

Department ID : 110

Dept Name : Information Technology

Location : Kanuru

Working Project : Eye Cataract Detection

Company : VRSEC

Salary : 45000.00

Employee DataBase

	Emp_ID	Emp_Name	Age	Gender
▶	1	Ajay	22	Male
	2	Rizwan	19	Male
	4	M.J.N.V.Sai	20	Male
	5	Srinivas	150	Male
	999	Mounav	20	Male
	2345	Mounav kolluri	20	Male
*				

Insert **Delete**

Display **Modify**

Search **Exit**

Employee

Employee ID : 2345

Employee Name : Mounav kolluri

Age : 20

Gender : Male

Designation : Manager

Department ID : 110

Dept Name : Information Technology

Location : Kanuru

Working Project : Eye Cataract Detection

Company : VRSEC

Salary : 45000.00

Employee DataBase

	Emp_ID	Emp_Name	Age	Gender
▶	1	Ajay	22	Male
	2	Rizwan	19	Male
	4	M.J.N.V.Sai	20	Male
	5	Srinivas	150	Male
	999	Mounav	20	Male
*				

Insert **Delete**

Display **Modify**

Search **Exit**

Search Query :

Emp_ID	Emp_Name	Age	Gender
4	M.J.N.V.Sai	20	Male

Display Query :

When we click on that button it displays the table from the database

Exit :

When we click on the exit button it exits the total application.

End Of The Task

