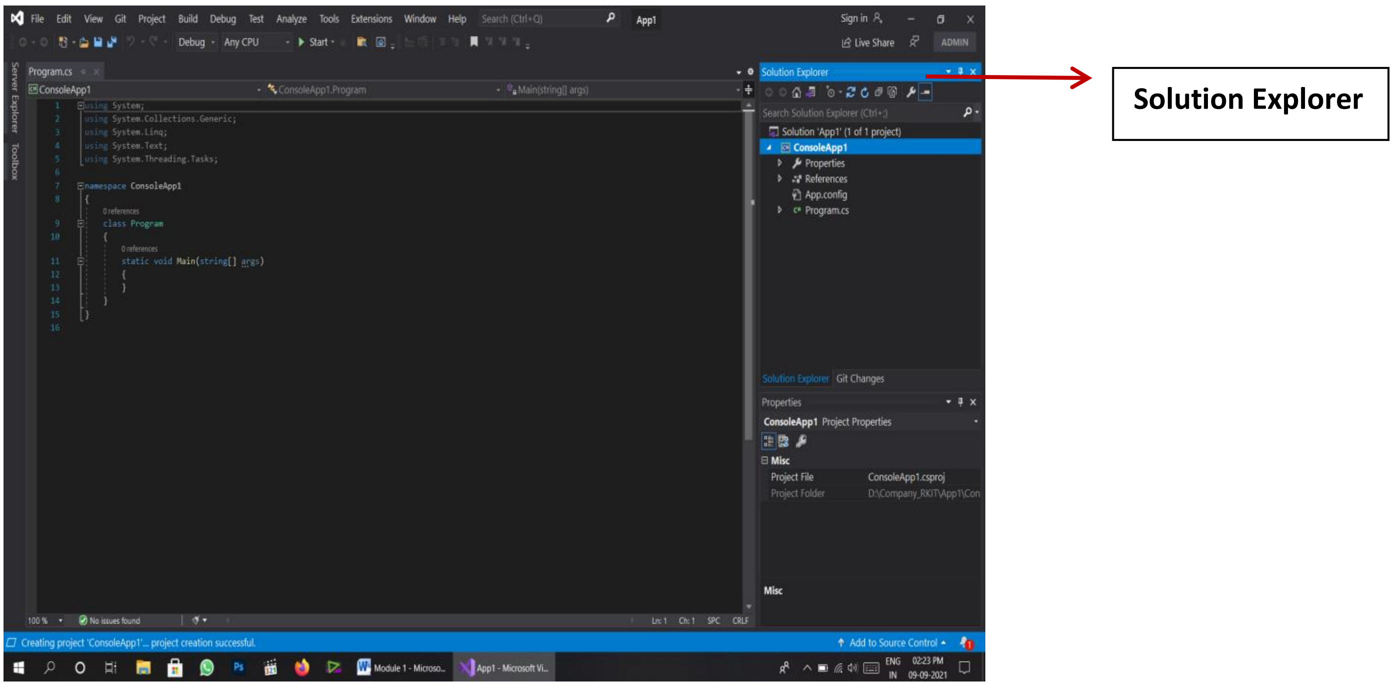
Module 1

1. Visual Studio 2019 IDE Overview

➢ Different types of windows

There are some main windows in visual studio IDE which are as shown below:

- **Solution Explorer**



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
        }
    }
}
```

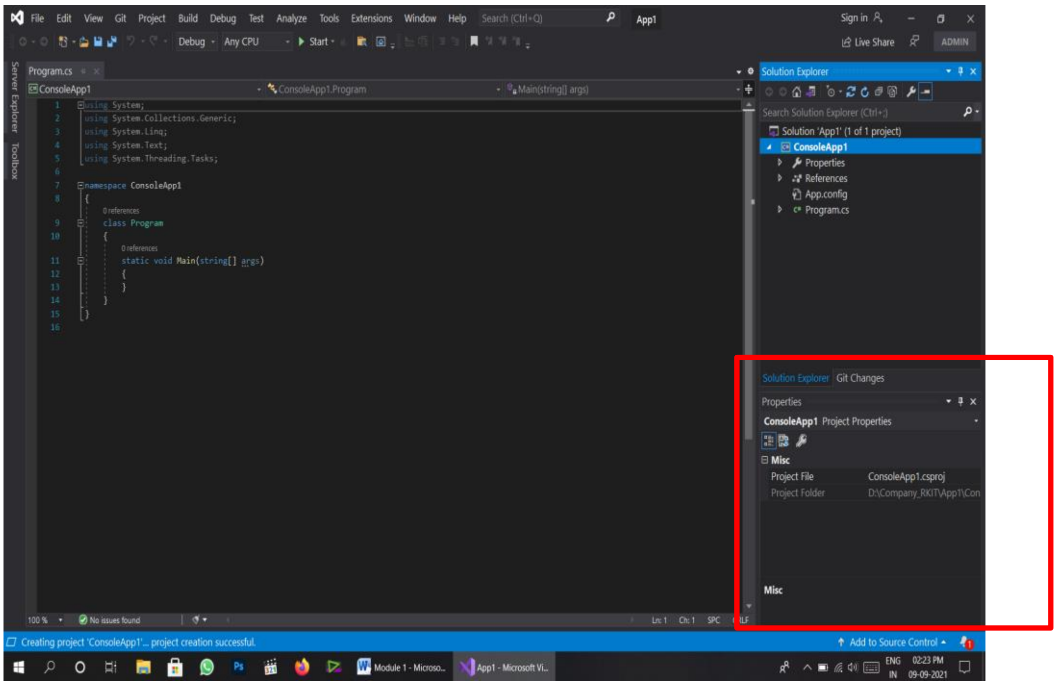
Project File	ConsoleApp1.csproj
Project Folder	D:\Company_RKIT\App1\Con

Creating project 'ConsoleApp1'... project creation successful.

Solution Explorer is a window in IDE as shown in the figure, that enables you to manage solutions, projects, and files. It provides a complete view of the files in a project, and it enables you to add or remove files and to organize files into subfolders. It helps in manage your code files and also provides navigation. It groups similar projects or project files under single solution.

- **Properties Window**

The Properties window displays different types of editing fields, depending on the needs of a particular property. These edit fields include edit boxes, drop-down lists, and links to custom editor dialog boxes. Properties shown in gray are read-only.



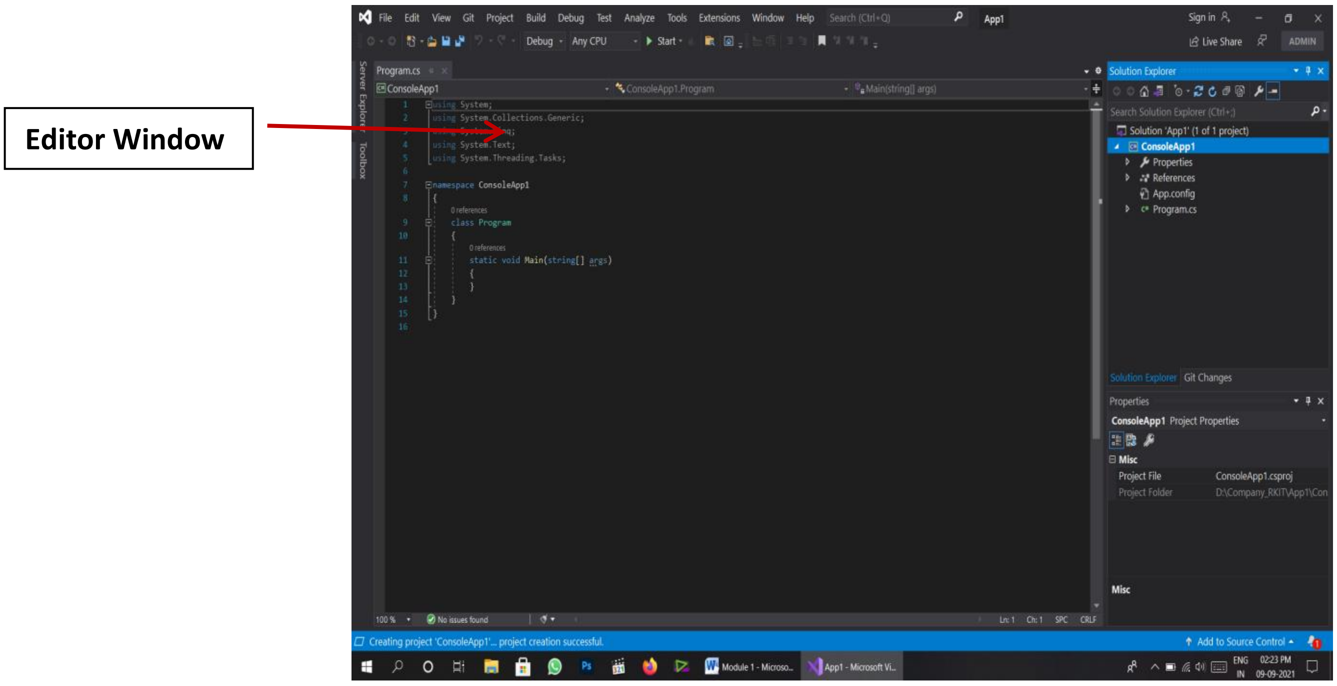
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
        }
    }
}
```

Project File	ConsoleApp1.csproj
Project Folder	D:\Company_RKIT\App1\Con

Creating project 'ConsoleApp1'... project creation successful.

- **Editor Window**

This is where we write our code. It displays the file contents. We can edit the code or design a user interface like buttons or text boxes. It provides varieties of features which make it easier to write and manage the code. These features include syntax colouring, errors and warning marks, Brace matching, structure visualizer, line numbers, etc…



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
        }
    }
}
```

Project File	ConsoleApp1.csproj
Project Folder	D:\Company_RKIT\App1\Con

Creating project 'ConsoleApp1'... project creation successful.

➢ Solution, project

- **Solution**

  A Solution contains group of similar projects. It's simply a container for one or more related projects, along with build information, Visual Studio window settings, and any miscellaneous files that aren't associated with a particular project. A solution is described by a text file (extension .sln) with its own unique format; it's not intended to be edited by hand.

  Visual Studio uses two file types (.sln and .suo) to store settings for solutions:

  .sln - Visual Studio Solution-Organizes projects, project items, and solution items in the solution.
  .suo- Solution User Options- Stores user-level settings and customizations, such as breakpoints.

- **Project**

  We can create a new project from a project template for a particular type of application or website. A project template consists of a basic set of pre-generated code files, config files, assets, and settings. These templates are available in the dialog box where you create a new project. For more information, see Create a new project in Visual Studio and Create solutions and projects.
  For customizing our projects in a certain way, we can create a custom project template that can be used to create new projects from. For more information, see Create project and item templates.

➢ Code Editor features

Syntax Colouring: Some syntax elements of code and mark-up files are coloured differently to distinguish them. For example, keywords are one colour, but types with another colour.

Error and warning mark: As we add code and build your solution, we may see different-coloured wavy underlines or light bulbs appearing in the code. Red wavy underlines denote syntax errors, blue denotes compiler errors, green denotes warnings, and purple denotes other types of errors. Quick Actions suggest fixes for problems and make it easy to apply the fix.

Brace Matching: When the insertion point is placed on an open brace in a code file, both it and the closing brace are highlighted. This feature gives immediate feedback on misplaced or missing braces.

Structure Visualizer: Dotted lines connect matching braces in code files, making it easier to see opening and closing brace pairs.

Line Numbers: Line numbers can be displayed in the left margin of the code window. It can be useful in case of debugging code editing task becomes easier with line numbers.

Change Tracking: The colour of the left margin allows you to keep track of the changes you have made in a file. Changes you have made since the file was opened but not saved are denoted by a yellow bar on the left margin (known as the selection margin). After you have saved the changes (but before closing the file), the bar turns green. If you undo a change after you have saved the file, the bar turns orange.

Format Document: Sets the proper indentation of lines of code and moves curly braces to separate lines in the document.

Format Selection: Sets the proper indentation of lines of code and moves curly braces to separate lines in the selection.

Make Lowercase: Changes all characters in the selection to lowercase, or if there is no selection, changes the character at the insertion point to lowercase.

Word Wrap: Causes all the lines in a document to be visible in the code window.

Comment Selection: Adds comment characters to the selection or the current line.

Uncomment Selection: Removes comment characters from the selection or the current line.

➢ Shortcuts

We can navigate in Visual Studio more easily by using the shortcuts. These shortcuts include keyboard and mouse shortcuts as well as text you can enter to help accomplish a task more easily.

Shortcuts for some basic functioning are listed in table given below:

Commands	**Shortcut Keys**
Maximize floating window	Double-Click on title bar
Maximize/minimize windows	Win+Up arrow/Win+Down arrow
Close active document	Ctrl+F4
Show open file list	Ctrl+Alt+Down arrow
Show all floating windows	Ctrl+Shift+M
Switch between windows	Win+N
Solution Explorer search	Ctrl+;
Quick Find	Ctrl+F
Dismiss Find	Esc
Quick Replace	Ctrl+H
Go To All	Ctrl+T
Start debugging	F5
Stop debugging	Shift+F5
Restart debugging	Ctrl+Shift+F5
Toggle full screen	F11
New File	Ctrl+N
Open File	Ctrl+O

2. Project Types

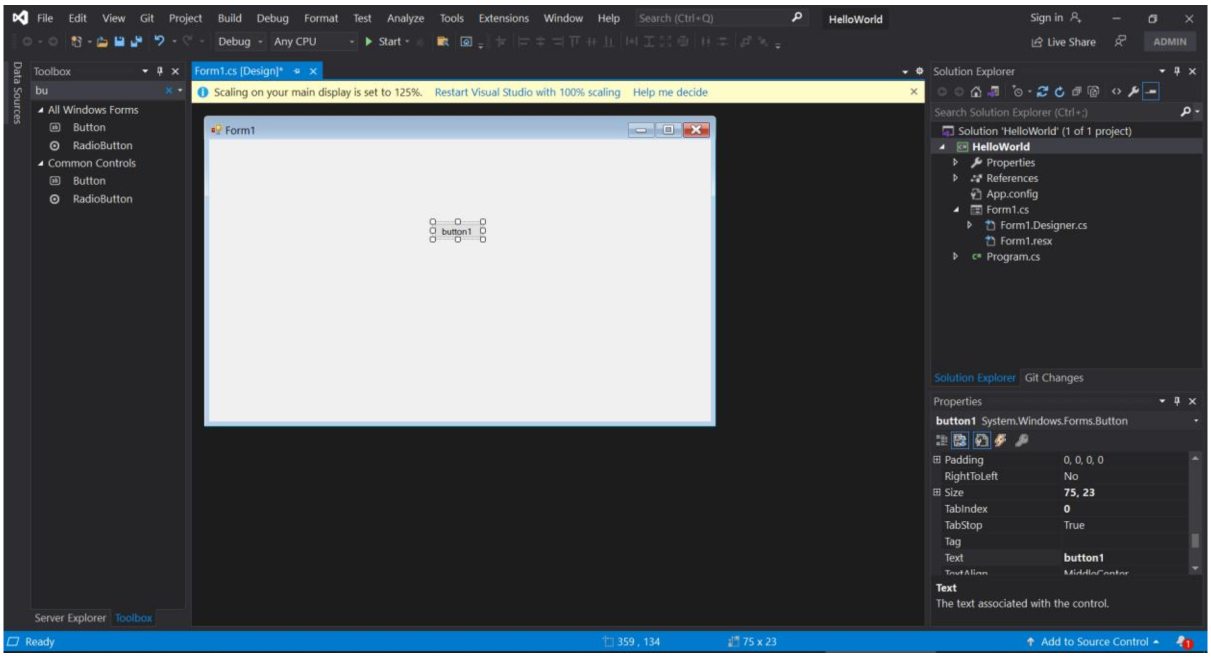
➢ Windows App

A program that is written to run under the Microsoft Windows operating system is called a "Windows app." Visual Studio 2019 IDE allows us to create windows app.

First, you'll create a Visual Basic application project. The project type comes with all the template files you'll need, before you've even added anything.

After you select your Visual Basic project template and name your file, Visual Studio opens a form for you. A form is a Windows user interface.

Click **Toolbox** to open the Toolbox fly-out window. Then Click the **Pin** icon to dock the **Toolbox** window. Click the **Button** control and then drag it onto the form.



Scaling on your main display is set to 125%. Restart Visual Studio with 100% scaling Help me decide

button1 System.Windows.Forms.Button

Padding	0, 0, 0, 0
RightToLeft	No
Size	75, 23
TabIndex	0
TabStop	True
Tag	
Text	button1

Text
The text associated with the control.

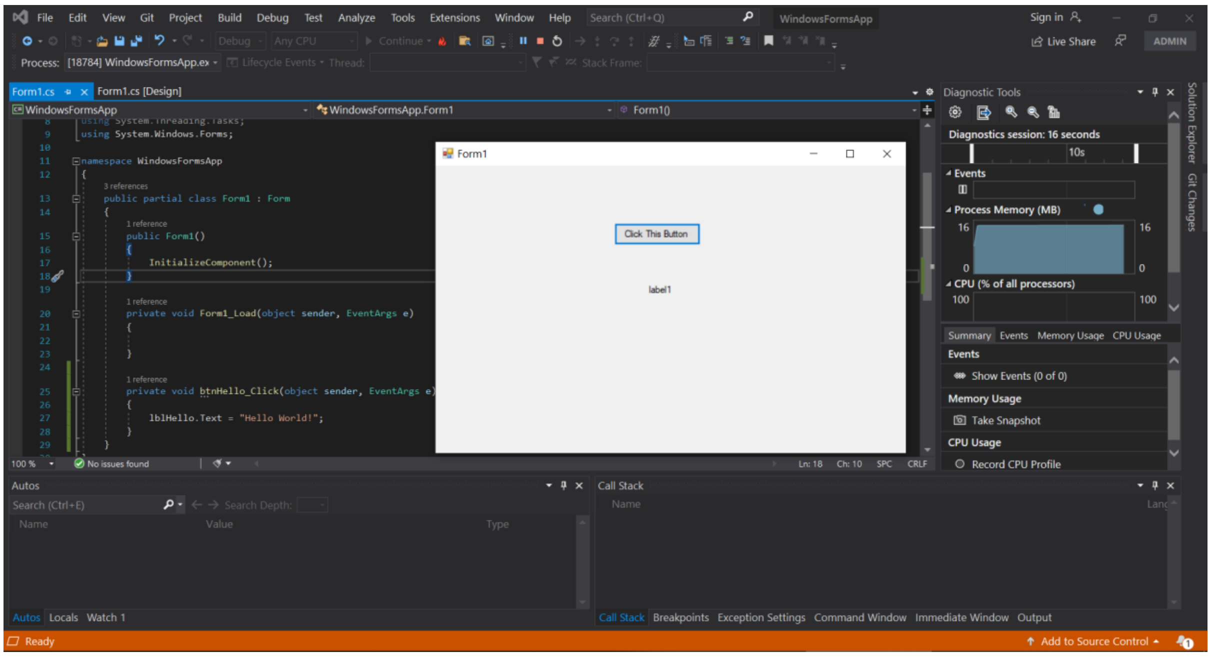
In the **Appearance** section (or the **Fonts** section) of the **Properties** window, type Click This Button, and then press **Enter**. In the **Design** section of the **Properties** window, change the name from **Button1** to btnHello, and then press **Enter**.

Select the **Label** control from the **Toolbox** window, and then drag it onto the form and drop it beneath the **Click This Button** button. Then change the name of **Label1** to lblHello, and then press **Enter**.

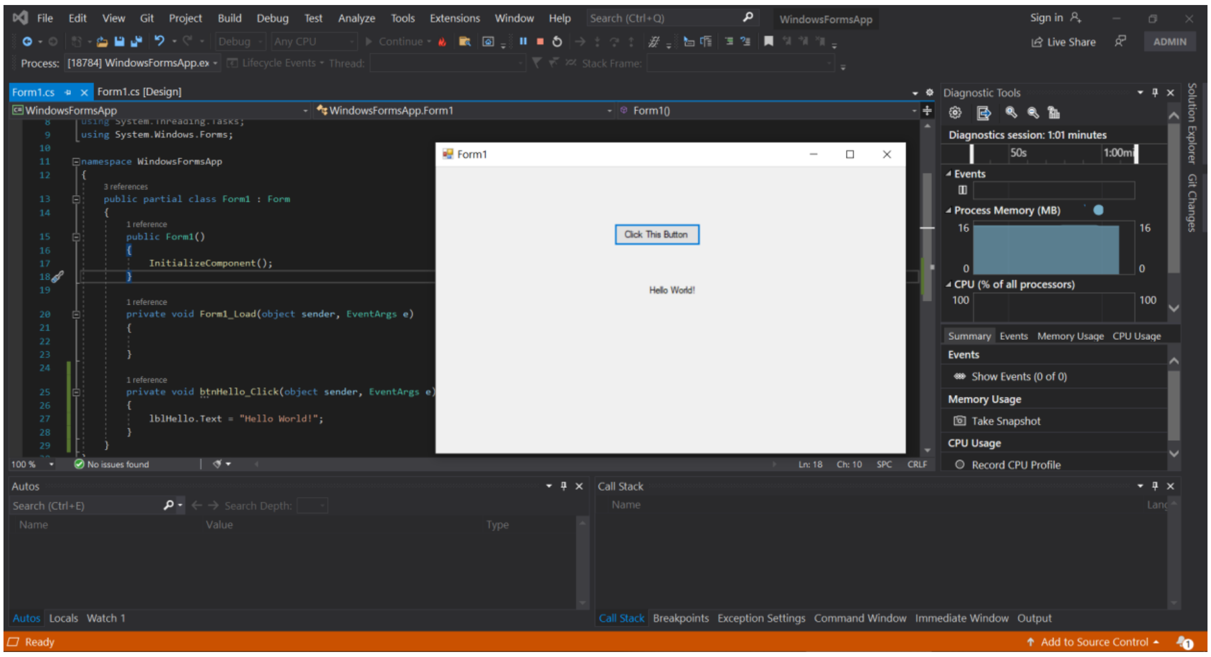
Then Write the code like this:

```
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;

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

      private void Form1_Load(object sender, EventArgs e)
      {


      }

      private void btnHello_Click(object sender, EventArgs e)
      {
         lblHello.Text = "Hello World!";
      }
   }
}
```

Output:



```
using System.Threading.Tasks;
using System.Windows.Forms;

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

        private void Form1_Load(object sender, EventArgs e)
        {

        }

        private void btnHello_Click(object sender, EventArgs e)
        {
            lblHello.Text = "Hello World!";
        }
    }
```



```
using System.Threading.Tasks;
using System.Windows.Forms;

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

        private void Form1_Load(object sender, EventArgs e)
        {

        }

        private void btnHello_Click(object sender, EventArgs e)
        {
            lblHello.Text = "Hello World!";
        }
    }
```

➢ Class Library

The Class Library contains program code, data, and resources that can be used by other programs and are easily implemented into other Visual Studio projects. In visual studio we can implement code in project type as class library and then it can be referred by other programs or even in same program.

Simple program that demonstrates use of class library is shown below. I have prepared one class library project and another console application which uses method specified in the class library.

Class library code:

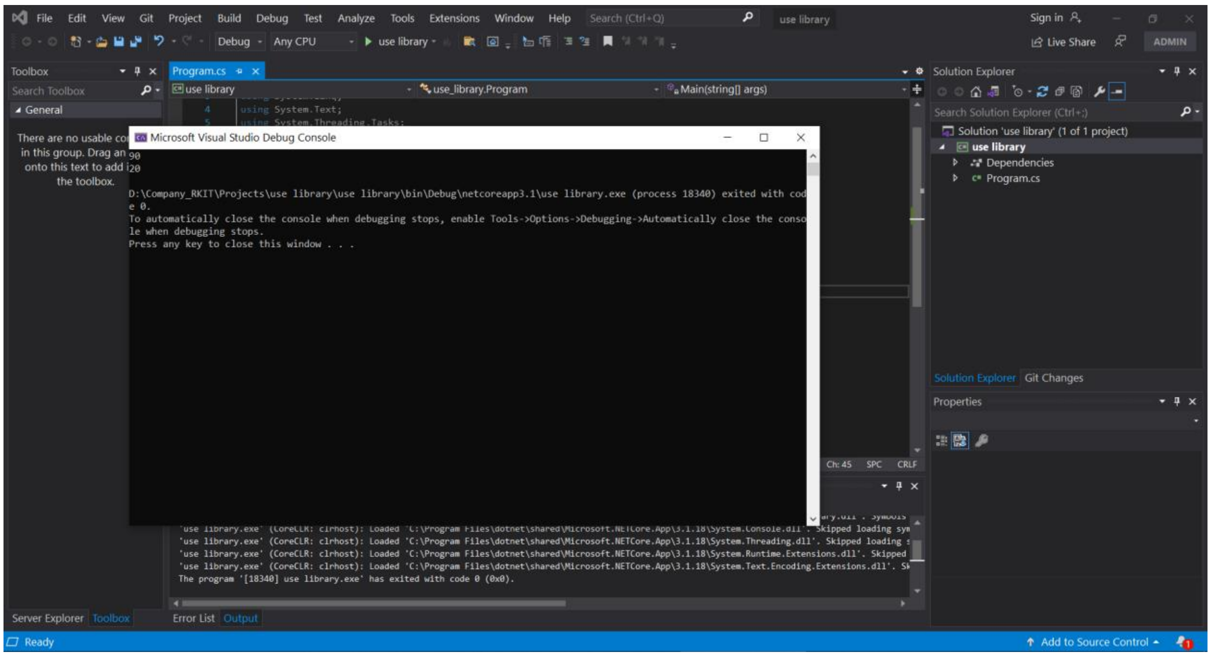
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ClassLibrary
{
   public sealed class Class1
   {
      public int add(int a, int b)
      {
         return a + b;
      }
      public int sub(int a, int b)
      {
         return a - b;
      }
   }
}
```

Console application which uses the class library method:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using ClassLibrary;

namespace use_library
{
   class Program
```

```
    {
        static void Main(string[] args)
        {
            Class1 c = new Class1();
            Console.WriteLine(c.add(15,75));
            Console.WriteLine(c.sub(95, 75));
        }
    }
}
```

Output:



```
90
20

D:\Company_RKIT\Projects\use library\use library\bin\Debug\netcoreapp3.1\use library.exe (process 18340) exited with cod
e 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

```
'use library.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\3.1.18\System.Console.dll'. Skipped loading sym
'use library.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\3.1.18\System.Threading.dll'. Skipped loading s
'use library.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\3.1.18\System.Runtime.Extensions.dll'. Skipped
'use library.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\3.1.18\System.Text.Encoding.Extensions.dll'. Sk
The program '[18340] use library.exe' has exited with code 0 (0x0).
```

➢ Web Application

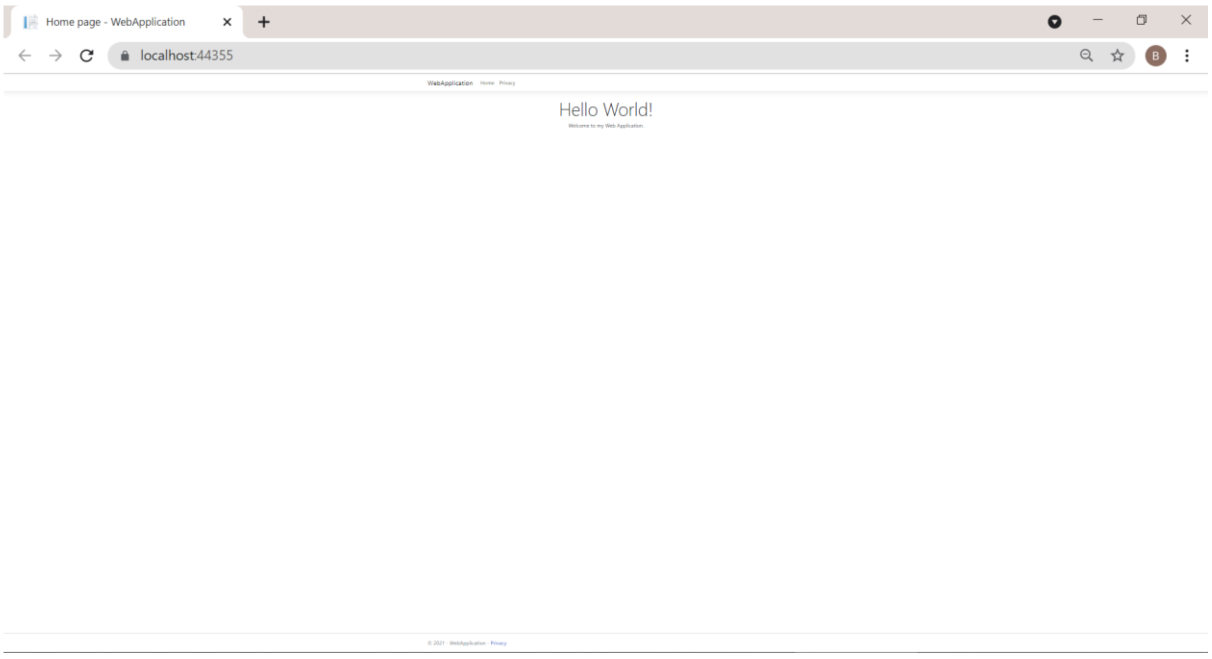
A web application is a computer program that utilizes web browsers and web technology to perform tasks over the Internet. Microsoft Visual Studio is an effective development environment for those who work with Microsoft development services. It's the best solution to create web apps with .NET or ASP.NET. If you would like to create a small website or build a basic web app, then you can choose Visual Studio Code.

Here is the code of small web app prepared by me. I have taken project type as ASP.Net core web app. Target framework is ASP.Net Core 3.1

Code for home page (index.cshtml)

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Hello World!</h1>
    <p>Welcome to my Web Application.</p>
</div>
```

Output:



Hello World!

Welcome to my Web Application.

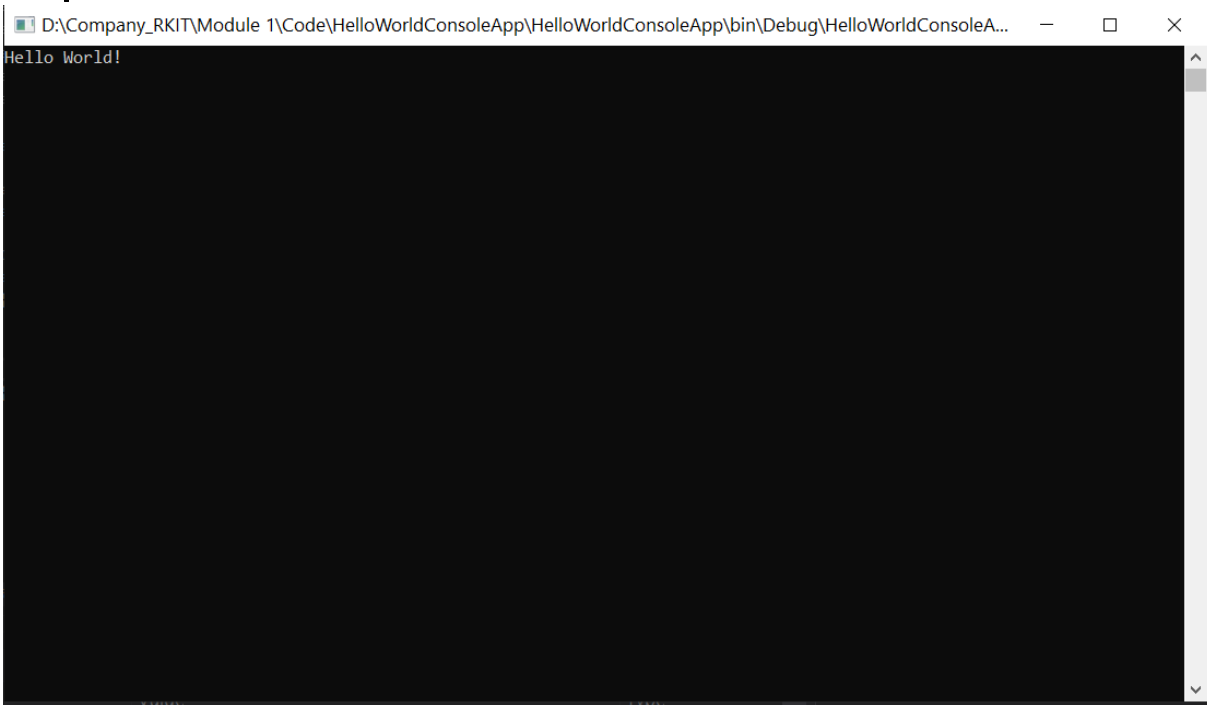
© 2021 - WebApplication - Privacy

3. Creating first C# program “Hello World”

A simple Console application for Hello World program in C# is as shown below:

```
using System;
namespace HelloWorldConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.ReadLine();
        }
    }
}
```

Output:



```
D:\Company_RKIT\Module 1\Code\HelloWorldConsoleApp\HelloWorldConsoleApp\bin\Debug\HelloWorldConsoleA...
Hello World!
```

➢ What is namespace?

Namespaces are used to organize the classes. It helps to control the scope of methods and classes in larger .Net programming projects. It provides a way to keep one set of names(like class names) different from other sets of names. The biggest advantage of using namespace is that the class names which are declared in one namespace will not clash with the same class names declared in another namespace. It is also referred as named group of classes having common features. The members of a namespace can be namespaces, interfaces, structures, and delegates.

To define a namespace in C#, we will use the namespace keyword followed by the name of the namespace and curly braces containing the body of the namespace as follows:

```
namespace namespace_name {
// code declarations
```

}

The members of a namespace are accessed by using dot(.) operator. A class in C# is fully known by its respective namespace.

Syntax:
[namespace_name].[member_name]

C# provides a keyword "using" which help the user to avoid writing fully qualified names again and again. The user just has to mention the namespace name at the starting of the program and then he can easily avoid the use of fully qualified names.
Syntax:
using [namespace_name][.][sub-namespace_name];

You can also define a namespace into another namespace which is termed as the nested namespace. To access the members of nested namespace user has to use the dot(.) operator.

For example, Generic is the nested namespace in the collections namespace as System.Collections.Generic

- **What is class?**

  Class is basically a blueprint for object. Objects are instances of a class. The methods and variables that constitute a class are called members of the class.

  Defining Class in C#:
  A class definition starts with the keyword class followed by the class name; and the class body enclosed by a pair of curly braces.

  For eg:
  class class_name{
  //members of class
  }

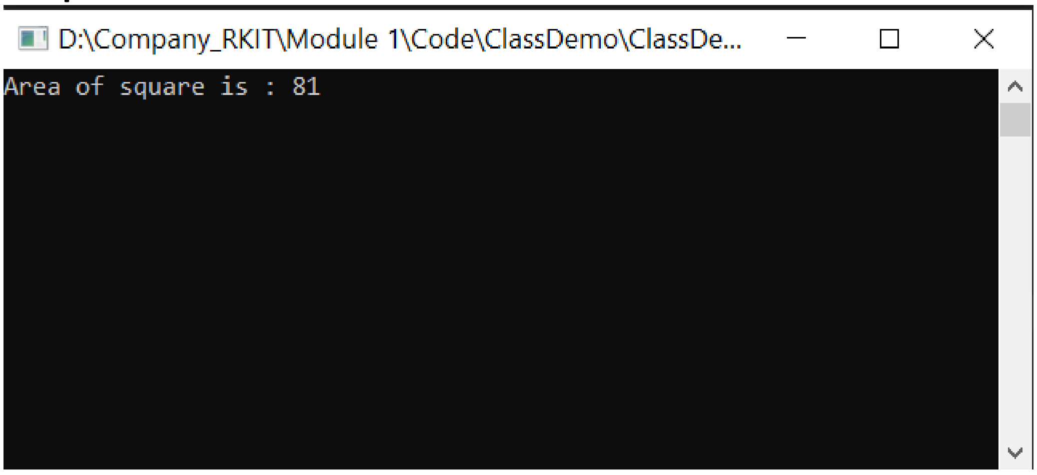
  For accessing members of class dot(.) operator is used.
  Default access modifier for class is internal. However, different access modifiers can be public, private, protected, protected internal & private protected.

Here, I have prepared a basic C# program that demonstrates usage and working of class.

C# Code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace classDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         Square s = new Square();
         s.length = 9;
         int area = s.length * s.length;
         Console.WriteLine("Area of square is : {0}", area);
         Console.ReadLine();
      }
   }
   class Square
   {
      public int length;
   }
}
```

Output:



```
Area of square is : 81
```

➢ Variables and methods declaration

- **Variables in C#**

  A variable is a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and layout of the variable's memory the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

  Basic value types provided in C# are:
  Integral types - sbyte, byte, short, ushort, int, uint, long, ulong, and char
  Floating point types - float and double
  Decimal types - decimal
  Boolean types - true or false values, as assigned
  Nullable types - Nullable data types
  C# also allows defining other value types of variable such as enum and reference types of variables such as class

  How to define variable in C#?

  Syntax:
  <data_type><variable_list>;

  How to initialize variables in C#?
  Syntax:
  variable_name = value;

  However, Variables can be initialized in their declaration too. The initializer consists of an equal sign followed by a constant expression as –

  <data_type><variable_name>= value;

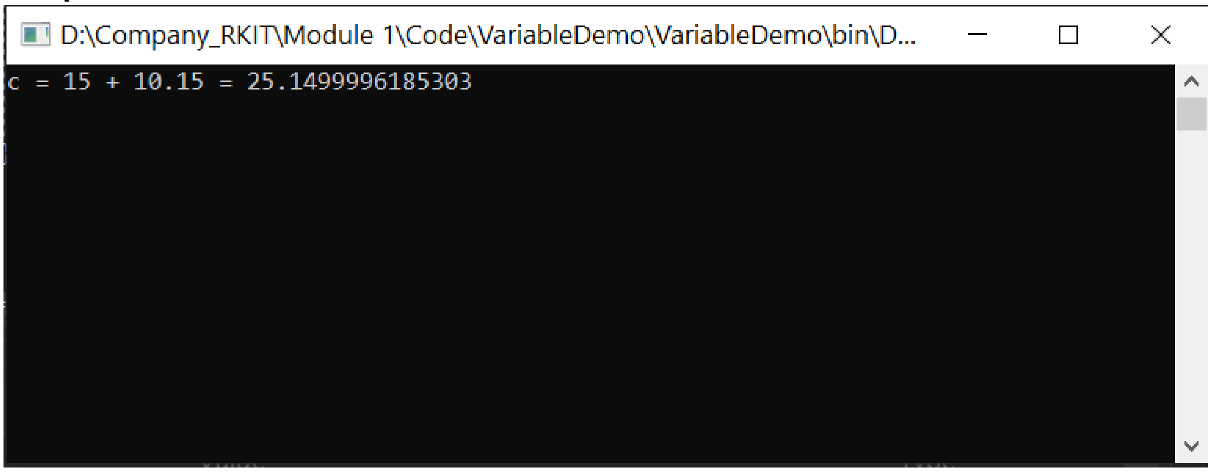
  Here I have performed a simple program that demonstrates use of variables in C#:

  C# Code:

```
using System;
namespace variableDemo
{
   class Program
```

```
    {
      static void Main(string[] args)
      {
        int a = 15;
        float b = 10.15F;
        double c;
        c = a + b;
        Console.WriteLine("c = {0} + {1} = {2}", a,b,c);
        Console.Read();
      }
    }
}
```

Output:


D:\Company_RKIT\Module 1\Code\VariableDemo\VariableDemo\bin\D...

```
c = 15 + 10.15 = 25.1499996185303
```

- **Methods declaration**

A method is a group of statements that together perform a task. Every C# program has at least one class with a method named Main.

How to define method in C#?

Syntax:
<Access Specifier><Return Type><Method Name>(Parameter List) {
 Method Body
}

Access Specifier – This determines the visibility of a variable or a method from another class.

Return type − A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is void.

Method name − Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.

Parameter list − Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.

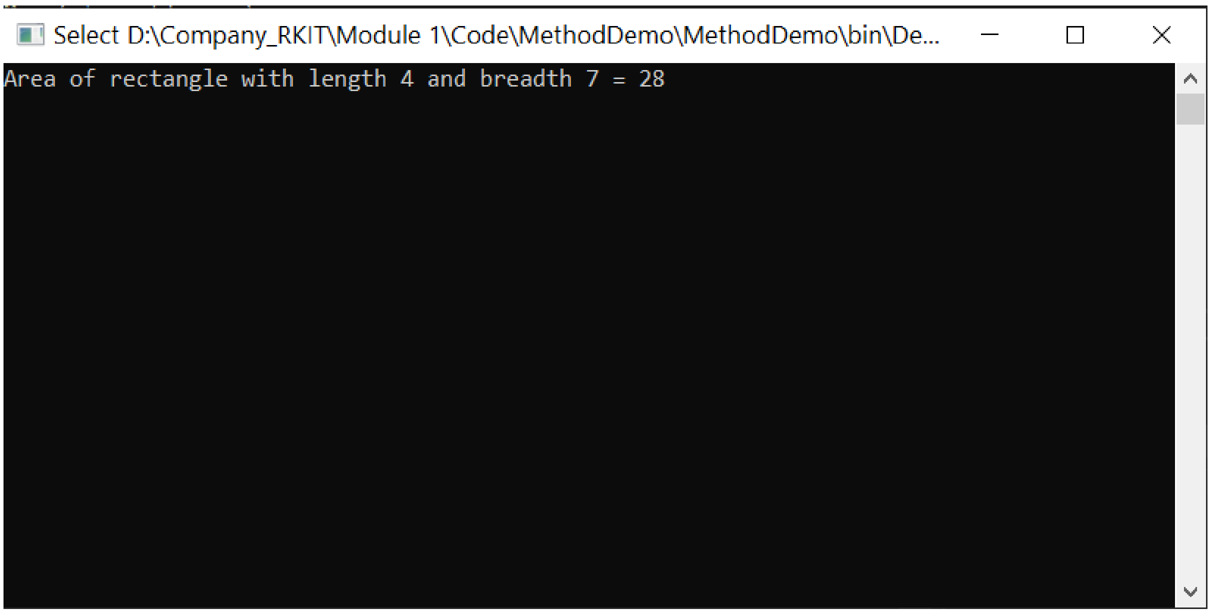
Method body − This contains the set of instructions needed to complete the required activity.

How to Call a Method in C#?
You can call a method using the name of the method.

Following code demonstrates how to define and call method:
C# Code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace methodDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            int l = 4;
            int b = 7;
            int area = r.Area(l, b);
            Console.WriteLine("Area of rectangle with length {0} and breadth {1} =
{2}", l,b,area);
            Console.Read();
        }
    }
    class Rectangle
    {
        public int Area (int length, int breadth)
        {
            int ans = length * breadth;
            return ans;
        }
    }
}
```

Output:



Select D:\Company_RKIT\Module 1\Code\MethodDemo\MethodDemo\bin\De...

```
Area of rectangle with length 4 and breadth 7 = 28
```

4. Understanding C# Program

➢ Program Flow

Considering our first hello world program

```
1. /* First C# program */
2. using System;
3. namespace HelloWorldConsoleApp
4. {
```

```
5.     class Program
6.     {
7.        static void Main(string[] args)
8.        {
9.           Console.WriteLine("Hello World!");
10.          Console.ReadLine();
11.       }
12.   }
13.}
```

Line 1: It contains comment. Comments can be included in any part of a C# program. The comment begins with “/*” and ends with “*/”. The comment can span multiple lines.

Line 2: This line **using system;** the keyword **using** imports the **System** class file and makes the methods present within it as a part of the program. System is a namespace and it has to be imported for every c# program. System namespace contains the classes that most applications use for interacting with the operating system. The classes that are used more often are the ones required for basic Input/Output. Note that there is a semicolon at the end of this line. All lines of code in C# must end with a semicolon.

Line 3: namespace HelloWorldConsoleApp is the namespace which created when user defines project name. It has the same name as project name so all the related classes and methods of project are contained within this namespace.

Line 4: namespace are contained within curly braces. This line contains open curly braces for namespace.

Line 5: This line defines a class.

Line 7: Each class has one static void Main() function. This function is the entry point of a C# program. This means that the Main() function is the first function that is called when program execution begins. It is declared as public to make it accessible from just about anywhere in the program. The keyword public can be ignored, as by default, just as in C++, in C# also the members of a class are public. The Main() function is declared as a static member. Since in our program the Main() does not return any value its return type is declared as void. Do keep in mind the case of the keywords, all keywords on this line of code are in lower case except in case of the Main() function M is in upper case.

Line 9: Within the Main() function we call the WriteLine method of the Console class and pass the text “Hello World!” as its parameter. The WriteLine function displays text on the console or the DOS window. Note that the WriteLine method is a part of the Console class, which in turn is a part of the System namespace.

Line 10: We have used Console.Read() because otherwise the process would directly terminate and we won’t be able to see the output on console for a long time. Console.Read() reads the character from console so that we can view output till we close the output window.

➢ Understanding Syntax

First line of any program is **using System;** using is a keyword that is used to import a namespace. System is a namespace that all programs need for carrying out basic functionality of C# programs.

class keyword is used to define a class. Class is a blueprint for object.

Comments are used for explaining code. Compilers ignore the comment entries. The multiline comments in C# programs start with /* and terminates with the characters */. Single-line comments are indicated by the '//' symbol.

Variables are attributes or data members of a class, used for storing data.

Functions are set of statements that perform a specific task. The member functions of a class are declared within the class.

For **instantiating a class** we need to declare object of that class. It can be declared with new keyword.

An **identifier** is a name used to identify a class, variable, function, or any other user-defined item. The basic rules for naming classes in C# are as follows –

- A name must begin with a letter that could be followed by a sequence of letters, digits (0 - 9) or underscore. The first character in an identifier cannot be a digit.
- It must not contain any embedded space or symbol such as? - + ! @ # % ^ & * ( ) [ ] { } . ; : " ' / and \. However, an underscore (_) can be used.
- It should not be a C# keyword.

Keywords are reserved words predefined to the C# compiler. These keywords cannot be used as identifiers. However, if you want to use these keywords as identifiers, you may prefix the keyword with the @ character.

5. Working with code files, projects & solutions

➢ Understanding the structure of solution

A solution is a structure for organizing projects in Visual Studio. The solution maintains the state information for projects in two files:

.sln file (text-based, shared)

.suo file (binary, user-specific solution options)

The .sln file contains text-based information the environment uses to find and load the name-value parameters for the persisted data and the project VSPackages it references. When a user opens a solution, the environment cycles through the preSolution, Project, and postSolution information in the .sln file to load the solution, projects within the solution, and any persisted information attached to the solution. Each project's file contains additional information read by the environment to populate the hierarchy with that project's items. The hierarchy data persistence is controlled by the project. The data is not normally stored in the .sln file, although you can intentionally write project information to the .sln file if you choose to do so.

Header of .sln file(I have taken .sln file which has a project named HelloWorldConsoleApp)

Microsoft Visual Studio Solution File, Format Version 12.00

# Visual Studio Version 16

```
VisualStudioVersion = 16.0.31624.102

MinimumVisualStudioVersion = 10.0.40219.1

Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") =
"HelloWorldConsoleApp","HelloWorldConsoleApp\HelloWorldConsoleApp.csp
roj", "{DFADDF25-621A-43EB-B52E-12450D7836E5}"

EndProject

Global

	GlobalSection(SolutionConfigurationPlatforms) = preSolution

		Debug|Any CPU = Debug|Any CPU

		Release|Any CPU = Release|Any CPU

	EndGlobalSection

	GlobalSection(ProjectConfigurationPlatforms) = postSolution

		{DFADDF25-621A-43EB-B52E-12450D7836E5}.Debug|Any
CPU.ActiveCfg = Debug|Any CPU

		{DFADDF25-621A-43EB-B52E-12450D7836E5}.Debug|Any
CPU.Build.0 = Debug|Any CPU

		{DFADDF25-621A-43EB-B52E-12450D7836E5}.Release|Any
CPU.ActiveCfg = Release|Any CPU

		{DFADDF25-621A-43EB-B52E-12450D7836E5}.Release|Any
CPU.Build.0 = Release|Any CPU

	EndGlobalSection

	GlobalSection(SolutionProperties) = preSolution

		HideSolutionNode = FALSE

	EndGlobalSection

	GlobalSection(ExtensibilityGlobals) = postSolution
```

```
                SolutionGuid = {6D8BCC9F-82AB-47BC-B98A-65D41FB1F80B}
        EndGlobalSection
EndGlobal
```

The body of an .sln file consists of several sections labeled GlobalSection as shown above. The environment reads the Global section of the .sln file and processes all sections marked preSolution. In this example file, there is one such statement:

```
   GlobalSection(SolutionConfigurationPlatforms) = preSolution
    Debug|Any CPU = Debug|Any CPU
    Release|Any CPU = Release|Any CPU
    EndGlobalSection
```

The environment loads the VSPackage, calls QueryInterface on the VSPackage for the IVsPersistSolutionProps interface, and calls the ReadSolutionProps method with the data in the section so the VSPackage can store the data. The environment repeats this process for each preSolution section.

Based on the information contained in the project section of the .sln file, the environment loads each project file. The project itself is then responsible for populating the project hierarchy and loading any nested projects.

After all sections of the .sln file are processed, the solution is displayed in Solution Explorer and is ready for modification by the user.

➢ Understanding the structure of project

There are some basic files present in all project types which are described as below: From the solution explorer as seen above the project files and its structure can be viewed.

Program.cs file:

This is the file where actual code for Implementation is written. A console project always starts executing from the entry point public static void Main() in

Program class and so the name of file is Program.cs . Here is the console application project's program.cs file.

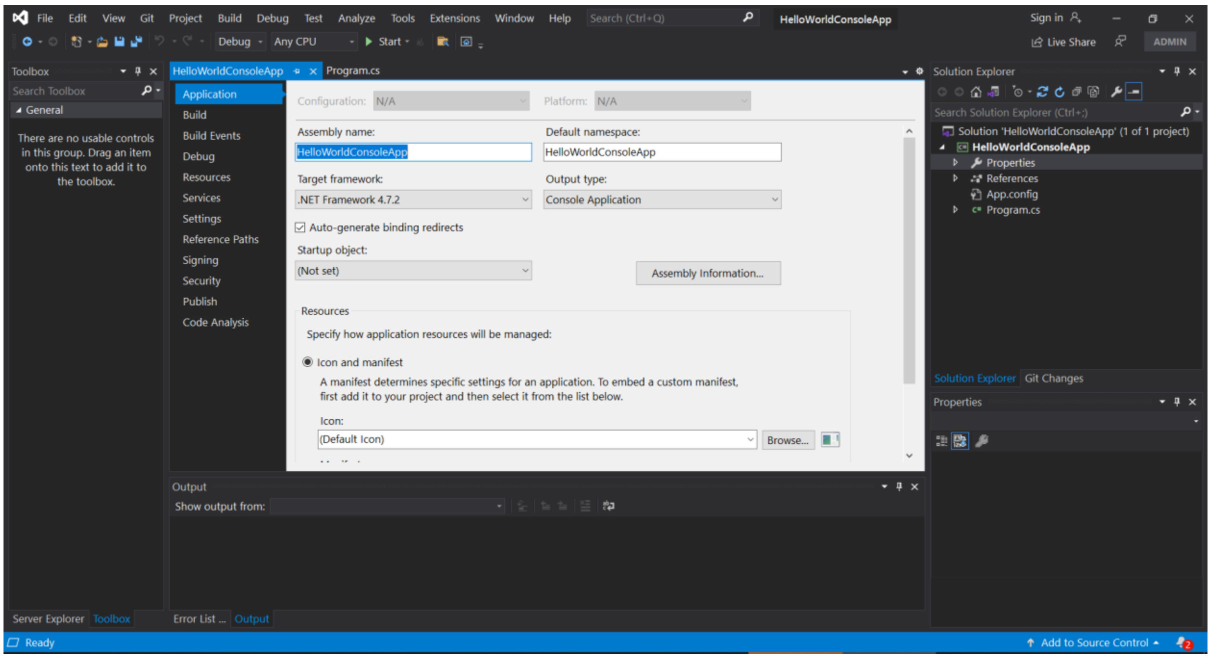
```
using System;
namespace variableDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         int a = 15;
         float b = 10.15F;
         double c;
         c = a + b;
         Console.WriteLine("c = {0} + {1} = {2}", a,b,c);
         Console.Read();
      }
   }
}
```

Dependencies file:
When building a solution that contains multiple projects, it can be necessary to build certain projects first, to generate code used by other projects. When a project consumes executable code generated by another project, the project that generates the code is referred to as a project dependency of the project that consumes the code. Such dependency relationships can be defined in the Project Dependencies dialog box.

Properties file:
Projects have properties that govern many aspects of compilation, debugging, testing and deploying. Some properties are common among all project types, and some are unique to specific languages or platforms. You access project properties by right-clicking the project node in Solution Explorer and choosing Properties or by typing properties into the search box on the menu bar and choosing Properties Window from the results.

.NET projects have a properties node in the project tree itself.

Project properties are organized into groups, and each group has its own property page. The pages might be different for different languages and project types.
For C# project it looks like:



Configuration: N/A Platform: N/A

Assembly name: HelloWorldConsoleApp

Default namespace: HelloWorldConsoleApp

Target framework: .NET Framework 4.7.2

Output type: Console Application

☑ Auto-generate binding redirects

Startup object: (Not set)

Resources

Specify how application resources will be managed:

● Icon and manifest

A manifest determines specific settings for an application. To embed a custom manifest, first add it to your project and then select it from the list below.

Icon: (Default Icon)

There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

Solution 'HelloWorldConsoleApp' (1 of 1 project)

As we can see there are different tabs in properties window like Application, Build, Build Events, Resources, services, etc. We can accordingly change the configuration as different tabs are selected.

To access properties on the solution, right click the solution node in Solution Explorer and choose Properties. In the dialog, you can set project configurations for Debug or Release builds, choose which projects should be the startup project when F5 is pressed, and set code analysis options.

Startup.cs file:

ASP.NET Core application must include Startup class. It is like Global.asax in the traditional .NET application. As the name suggests, it is executed first when the application starts. The startup class can be configured using UseStartup<T>() method at the time of configuring the host in the Main() method of Program as shown below:

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

```
    {
        BuildWebHost(args).Run();
    }
    public static IWebHost BuildWebHost(string[] args)
    {
        WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .Build();
    }
}
```

The name "Startup" is by ASP.NET Core convention. However, we can give any name to the Startup class, just specify it as the generic parameter in the UseStartup() method.

Startup.cs file of a WebApplication project is as mentioned below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace WebApplication
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }
        public IConfiguration Configuration { get; }
        // This method gets called by the runtime. Use this method to add
services to the container.
     public void ConfigureServices(IServiceCollection services)
        {
            services.AddRazorPages();
        }
        // This method gets called by the runtime. Use this method to configure
the HTTP request pipeline.
     public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }
            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseAuthorization();
            app.UseEndpoints(endpoints =>
            {
                endpoints.MapRazorPages();
            });
        }
    }
}
```

As you can see, Startup class includes two public methods: ConfigureServices and Configure.

The Startup class must include a Configure method and can optionally include ConfigureService method.

The **ConfigureServices** method is a place where you can register your dependent classes with the built-in IoC(Inversion Of Control) container. After registering dependent class, it can be used anywhere in the application. You just need to include it in the parameter of the constructor of a class where you want to use it. The IoC container will inject it automatically.

The **Configure** method is a place where you can configure application request pipeline for your application using IApplicationBuilder instance that is provided by the built-in IoC container.

➢ Familiar with different type of file extensions

Some basic and most used file extensions are listed below

.ADDIN	Visual Studio Add-in Definition File
.APPX	Windows App Package
.APPXMANIFEST	Windows 10 App Manifest
ASAX	ASP.NET Server Application File
.ASCX	ASP.NET User Control File
.ASHX	ASP.NET Web Handler File
.ASM	Visual Studio Assembler Source Code File
.IDL	Interface Definition Language File
.ASP	Active Server Page
.ASPX	Active Server Page Extended File
.AXD	ASP.NET Web Handler File
CS	C# Source Code File
.CSHTML	ASP.NET Razor Web Page
.CSPROJ	Visual Studio C# Project
.CSS	Cascading Style Sheet
.CSX	Visual C# Script
.DBPROJ	Visual Studio Database Project File
.DEF	Module-Definition File
.DTD	Document Type Definition File
HTML	Hypertext Markup Language File
.IDB	Visual Studio Intermediate Debug File
.XML	XML File
.XSS	XML Style Sheet
.JS	JavaScript File

6. Understanding data types & variables with conversion

➢ Base datatype

The variables in C#, are categorized into the following types –

- Value types
- Reference types
- Pointer types

Value type: Value type variables can be assigned a value directly. They are derived from the class System.Value Type.

The value types directly contain data. Some examples are int, char, and float, which stores numbers, alphabets, and floating point numbers, respectively. When you declare an int type, the system allocates memory to store the value.

Type	Represents	Default value
bool	Boolean value	False
byte	8-bit unsigned integer	0
char	16-bit Unicode character	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	0.0M
double	64-bit double-precision floating point type	0.0D
float	32-bit double-precision floating point type	0.0F
int	32-bit signed integer type	0
long	64-bit signed integer type	0L
sbyte	8-bit signed integer type	0
short	16-bit signed integer type	0
uint	32-bit unsigned integer type	0
ulong	64-bit unsigned integer type	0
ushort	16-bit unsigned integer type	0

To get the exact size of a type or a variable on a particular platform, you can use the sizeof method. The expression sizeof(type) yields the storage size of the object or type in bytes.

Reference type: The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables. In other words, they refer to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Example of built-in reference types are: object, dynamic, and string.

The **Object Type** is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types.

Eg: object obj;
obj = “ABCD”;

You can store any type of value in the **dynamic data** type variable. Type checking for these types of variables takes place at run-time.

Eg: dynamic d = 45;

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

The **String Type** allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.

Eg: String s = "abcd";

A @quoted string literal looks as follows –
@"abcd";

The user-defined reference types are: class, interface, or delegate.

Pointer Type: Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as the pointers in C or C++.

Eg:
char* c;
int* i;

➢ Datatype Conversion

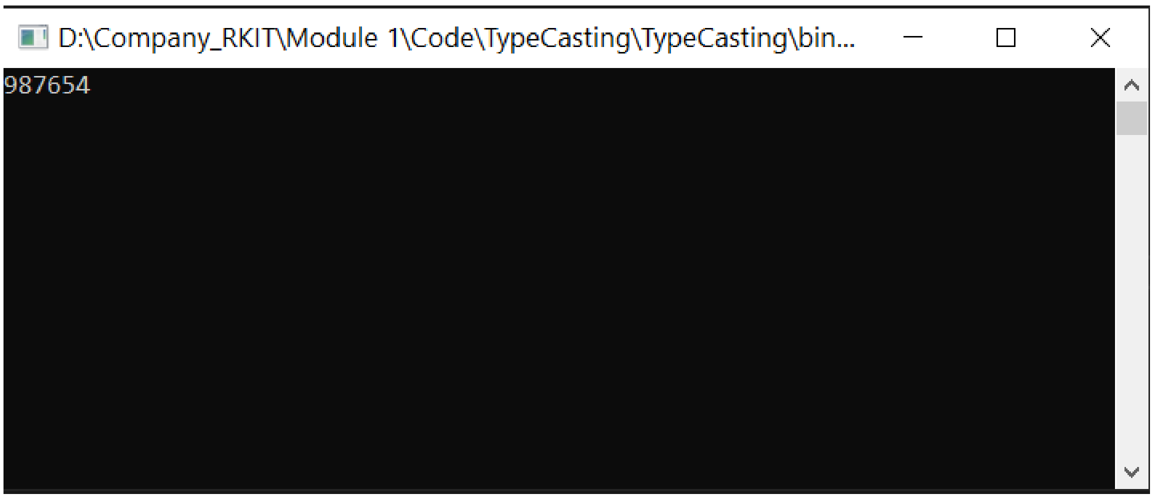
Type conversion is converting one type of data to another type. It is also known as Type Casting.

Implicit conversions: No special syntax is required because the conversion always succeeds and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.
Eg:
int num = 2147483365;
long bigNum = num;

Explicit type conversion: These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.
I have prepared a small code that demonstrates explicit type conversion.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TypeCasting
{
   class Program
   {
        static void Main(string[] args)
        {
          double d = 987654.321;
          int i;
          i = (int)d;
          Console.WriteLine(i);
          Console.Read();

        }
    }
}
```

Output:


D:\Company_RKIT\Module 1\Code\TypeCasting\TypeCasting\bin...

```
987654
```

➢ Boxing/Unboxing

Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type. When the common language runtime (CLR) boxes a value type, it wraps the value inside a System.Object instance and stores it on the managed heap. Boxing a value type allocates an object instance on the heap and copies the value into the new object.

Eg:

int i = 65;

object o = i;

Unboxing is an explicit conversion from the type object to a value type or from an interface type to a value type that implements the interface.

Eg:

int i = 1234;

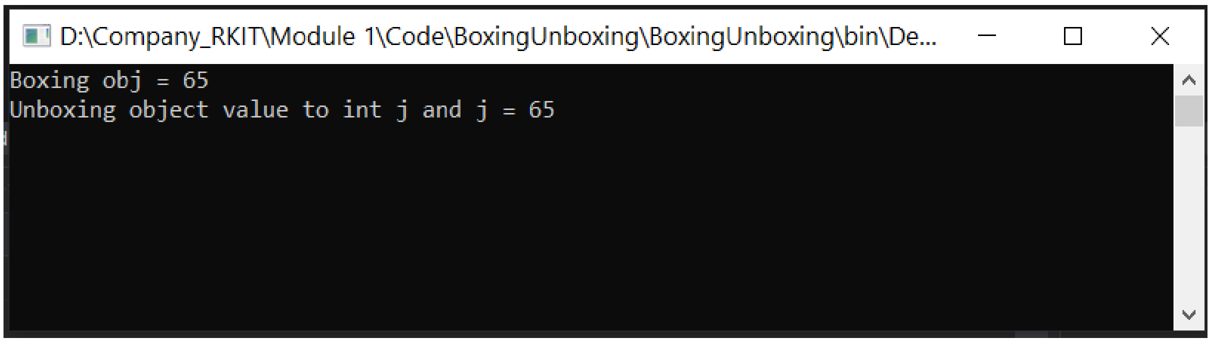
object o = i;

int j = (int)o;

A code that demonstrates both boxing and unboxing is shown as below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BoxingUnboxing
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 65;
            Object obj = i; //Boxing
            Console.WriteLine("Boxing obj = {0}", obj);
            int j = (int)obj; //Unboxing
            Console.WriteLine("Unboxing object value to int j and j = {0}", j);
            Console.Read();

        }
    }
}
```

Output:



D:\Company_RKIT\Module 1\Code\BoxingUnboxing\BoxingUnboxing\bin\De...

```
Boxing obj = 65
Unboxing object value to int j and j = 65
```

7. Understanding Decision making & statements

In Decision Making in programming, a certain block of code needs to be executed when some condition is fulfilled.

➢ If else

Types of if statements are:

- if
- if-else
- if-else-if
- Nested if

The **if statement** checks the given condition. If the condition evaluates to be true then the block of code/statemnts will execute otherwise not.

Syntax:

```
if(condition)
{
//code to be executed if condition is true
}
```

In **if-else**, the if statement evaluates the code if the condition is true but what if the condition is not true, here comes the else statement. It tells the code what to do when the if condition is false.

Syntax:

```
if(condition)
{
// code to be executed if condition is true
}
else
{
// code to be executed if condition is false
}
```

The **if-else-if ladder** statement executes one condition from multiple statements. The execution starts from top and checked for each if condition. The statement of if block will be executed which evaluates to be true. If none of the if condition evaluates to be true then the last else block is evaluated.

Syntax:

```
if(condition1)
{
// code to be executed if condition1 is true
}
else if(condition2)
{
// code to be executed if condition2 is true
}
else if(condition3)
{
// code to be executed if condition3 is true
}
...
else
{
// code to be executed if all the conditions are false
}
```

if statement inside an if statement is known as **nested if**. if statement in this case is the target of another if or else statement. When more than one condition needs to be true and one of the condition is the sub-condition of parent condition, nested if can be used.

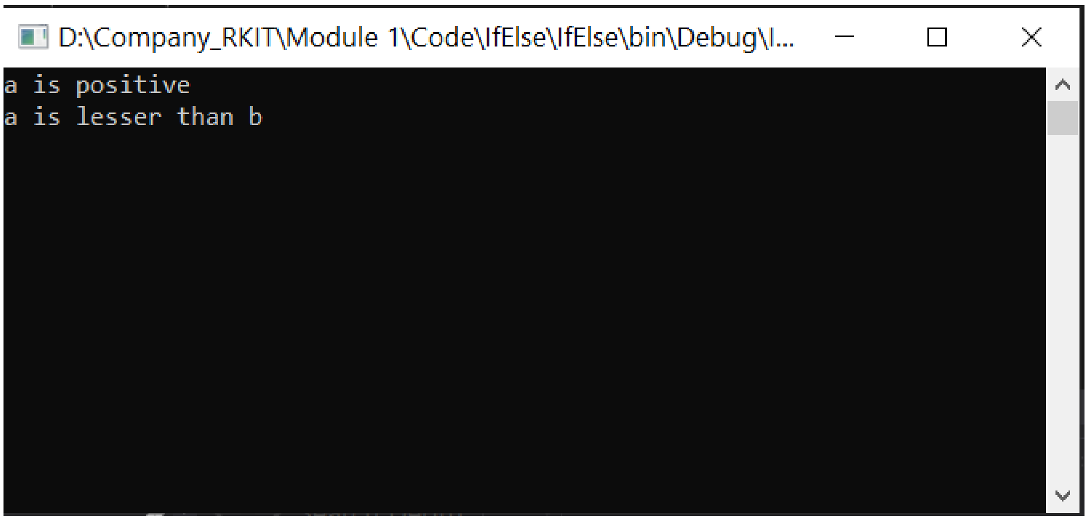
Syntax:

```
if (condition1)
{
// code to be executed
// if condition1 is true
if (condition2)
{
// code to be executed
// if condition2 is true
}
}
```

A code that demonstrates use of all types of if statements is prepared by me and is as shown below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace IfElse
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 9;
            int b = 15;
            if (a > 0)
            {
                Console.WriteLine("a is positive");
                if (b > 10)
                {
                    if (a > b)
                    {
                        Console.WriteLine("a is greater than b");
                    }
                    else
                        Console.WriteLine("a is lesser than b");
                }
            }
            else if (a < 0)
                Console.WriteLine("a is negative");
        else
            Console.WriteLine("a = 0");
        Console.Read();
    }
  }
}
```

Output:



D:\Company_RKIT\Module 1\Code\IfElse\IfElse\bin\Debug\I...

```
a is positive
a is lesser than b
```

➢ Switch Case

Switch statement is an alternative to long if-else-if ladders. The expression is checked for different cases and the one match is executed. break statement is used to move out of the switch. If the break is not used, the control will flow to all cases below it until break is found or switch comes to an end. There is default case (optional) at the end of switch, if none of the case matches then default case is executed.

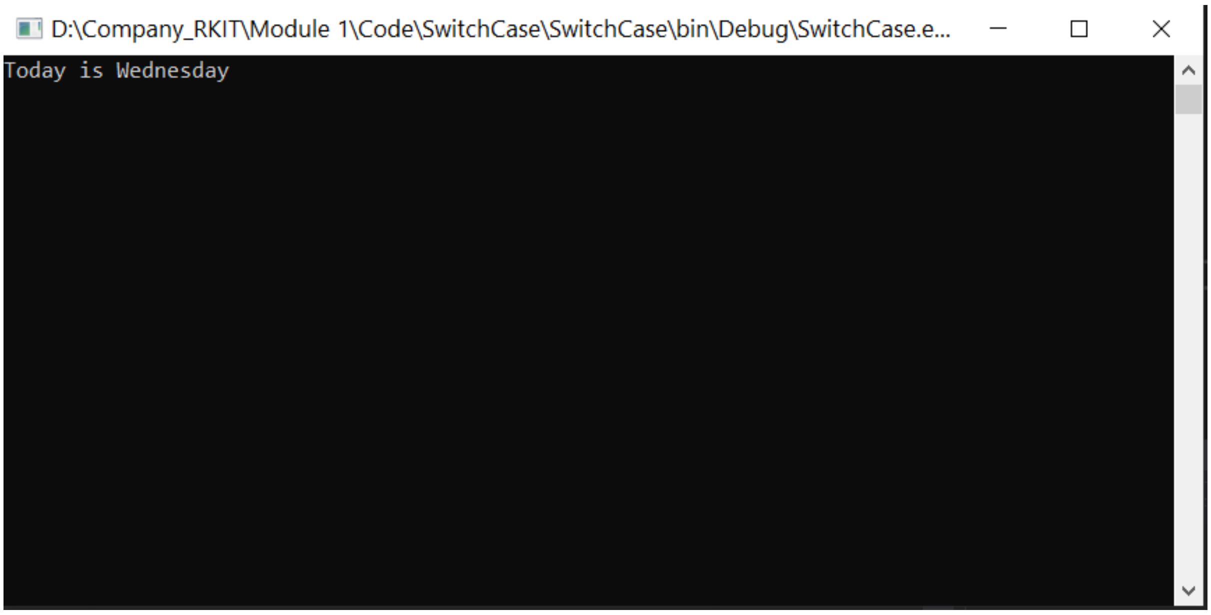
Syntax:

```
switch (expression)
{
case value1: // statement sequence
break;
case value2: // statement sequence
break;
.
.
.
case valueN: // statement sequence
break;
default: // default statement sequence
}
```

Code for switch case is as shown below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SwitchCase
{
    class Program
    {
        static void Main(string[] args)
        {
            int day = 3;
            switch (day)
            {
                case 1:
                    Console.WriteLine("Today is Monday");
                    break;
                case 2:
                    Console.WriteLine("Today is Tuesday");
                    break;
                case 3:
                    Console.WriteLine("Today is Wednesday");
                    break;
                case 4:
                    Console.WriteLine("Today is Thursday");
                    break;
                case 5:
                    Console.WriteLine("Today is Friday");
                    break;
                case 6:
                    Console.WriteLine("Today is Saturday");
                break;
            case 7:
                Console.WriteLine("Today is Sunday");
                break;
            default:
                Console.WriteLine("There is a Weekend.");
                break;
        }
      Console.Read();
     }
   }
}
```

Output:



D:\Company_RKIT\Module 1\Code\SwitchCase\SwitchCase\bin\Debug\SwitchCase.e...

```
Today is Wednesday
```