DOT NET PROGRAMMING

CHAPTER -2 :C# language fundament mentals

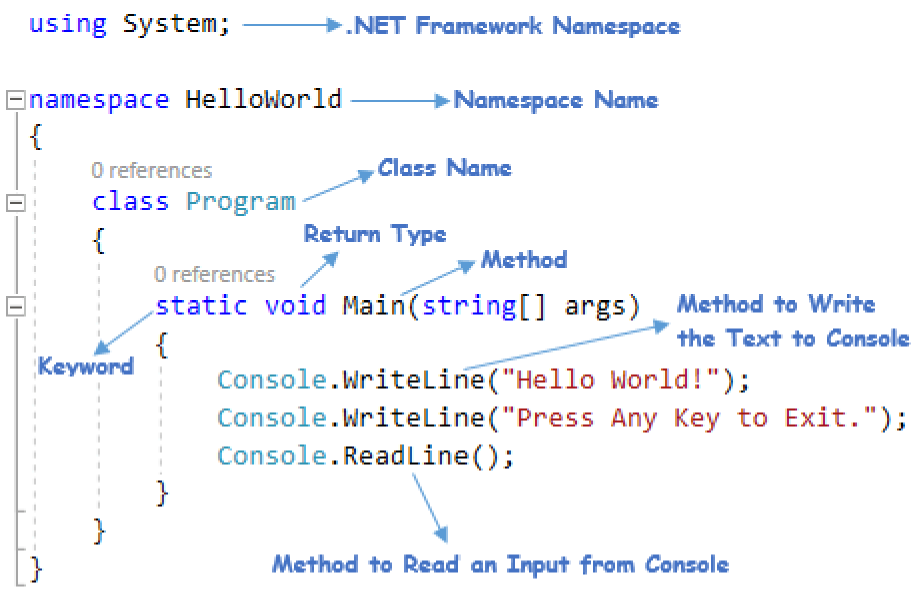
SESSION - 9: Anatomy of a basic C# class

C# demands that all program logic be contained within a type definition Unlike many other languages, in C# it is not possible to create global functions or globalpoints of data. Rather, all data members and all methods must be contained within a type definition.To get the ball rolling, create a new Console Application project named SimpleCSharpApp. You might agree that the code within the initial Program.cs file is rather uneventful.

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

namespace SimpleCSharpApp
{
   class Program {
      static void Main(string[] args)
      {
                     // Display a simple message to the user.
                   Console.WriteLine("***** My First C# App *****");
                   Console.WriteLine("Hello World!");
                   Console.WriteLine();
                            // Wait for Enter key to be pressed before shutting down.
                   Console.ReadLine();



      }
   }

}
```

Note C# is a case-sensitive programming language. Therefore, Main is not the same as main, and

Readline is not the same as ReadLine. Be aware that all C# keywords are lowercase (e.g., public,lock, class, dynamic), while namespaces, types, and member names begin (by convention) withan initial capital letter and have capitalized the first letter of any embedded words (e.g.,Console.WriteLine,System.Windows.MessageBox, System.Data.SqlClient).As a rule of thumb, whenever you receive a compiler error regarding “undefined symbols,” be sure to check



```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.WriteLine("Press Any Key to Exit.");
            Console.ReadLine();
        }
    }
}
```

your spelling and casing first!

using System;

Here, `using System` is the .NET Framework library namespaces, and we used `using` keyword to import `system` namespace to use existing class methods such as **WriteLine()**, **ReadLine()**, etc. By default, the **.NET Framework** provides a lot of namespaces to make the application implementation easy.The namespace is a collection of classes, and **classes** are the collection of objects and methods.

namespace HelloWorld

Here, `namespace HelloWorld` is the main namespace of our application, and by default, our application classes will be a part of it.

class Program

Here, `class Program` is used to define a class (**Program**) in the namespace (**Helloworld**). The class (**Program**) will contain all the variables, methods, etc., and we can define more than one class in same namespace based on our requirements.

static void Main(string[] args)

Here, `static void Main(string[] args)` is used to define a method in our class (**Program**).

- The keyword `static` tells us that the **main** method can be accessible without instantiating the class (**Program**).
- Another keyword `void` tells us that what this method should return.
- The name **Main** will refer to the name of our class method (**Program**). The **Main()** method is the entry point of our console application.
- After the name (**Main**) of a method, we defined a set of parameters within parentheses. Here our method takes only one parameter, called `args` and it is useful to send command-line arguments as text strings for our main method.

Console.WriteLine() / ReadLine()

Here, `Console.WriteLine()` and `Console.ReadLine()` methods are used to write a text to the console and read the input from the console.

The `Console` is a class of .NET Framework namespace `System` and WriteLine() and ReadLine() are the methods of `Console` class.

C# Class Members

As discussed, a **class** can contain multiple data members in the c# programming language. The following table lists a different type of data members that can be used in c# classes.

Member	Description
Fields	Variables of the class
Methods	Computations and actions that can be performed by the class
Properties	Actions associated with reading and writing named properties of the class
Events	Notifications that can be generated by the class
Constructors	Actions required to initialize instances of the class or the class itself
Operators	Conversions and expression operators supported by the class
Constants	Constant values associated with the class
Indexers	Actions associated with indexing instances of the class like an array
Finalizers	Actions to perform before instances of the class are permanently discarded
Types	Nested types declared by the class

Creating objects:

In c#, **Classes and Objects** are interrelated. The **class** in c# is nothing but a collection of various data members (fields, properties, etc.) and member functions. The **object** in c# is an instance of a **class** to access the defined properties and methods.

We will now learn the classes and objects in c# and how to use them in c# applications with examples.

C# Class

In c#, **Class** is a data structure, and it will combine various types of data members such as fields, properties, member functions, and events into a single unit.

Declaring a Class in C#

In c#, classes are declared by using `class` keyword. Following is the declaration of class in c# programming language.

```
public class users {

// Properties, Methods, Events, etc.

}
```

If you observe the above syntax, we defined a class "**users**" using `class` keyword with `public` access modifier. Here, the `public` access specifier will allow the users to create objects for this class, and inside of the body class, we can create required fields,properties,methods,andeventsto use in our applications.

Now we will see how to create a class in c# programming language with example.

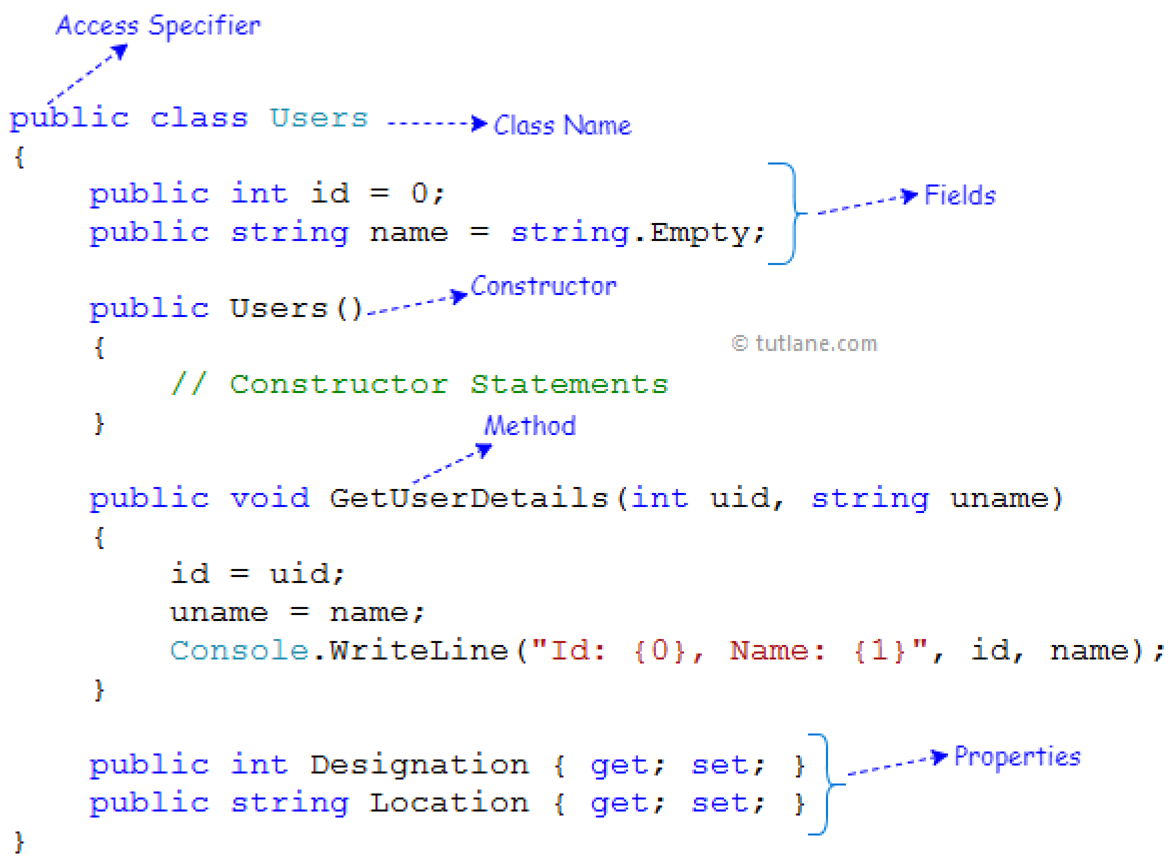
C# Class Example

Following is the example of creating a class in c# programming language with various data members and member functions.

```
public class Users
{
  public int id = 0;
  public string name = string.Empty;
            public Users()
              {
                          // Constructor Statements
             }
  public void GetUserDetails(int uid, string uname)
  {
    id = uid;
    uname = name;
    Console.WriteLine("Id: {0}, Name: {1}", id, name);
  }
  public int Designation { get; set; }
  public string Location { get; set; }
}
```

If you observe the above c# class example, we defined a class “**Users**” with various data members and member functions based on our requirements.

Following is the detailed description of various data members used in the above c# class example.



```
public class Users
{
    public int id = 0;
    public string name = string.Empty;

    public Users()
    {
        // Constructor Statements
    }

    public void GetUserDetails(int uid, string uname)
    {
        id = uid;
        uname = name;
        Console.WriteLine("Id: {0}, Name: {1}", id, name);
    }

    public int Designation { get; set; }
    public string Location { get; set; }
}
```

If you observe the above image, we used various data members likeaccess modifiers,fields,properties, methods, constructors, etc., in our c# class based on our requirements.

We will learn more about c#access modifiers,fields,properties,methods,constructors, etc. topics in the next chapters with examples.

C# Class Members

As discussed, a **class** can contain multiple data members in the c# programming language. The following table lists a different type of data members that can be used in c# classes.

Member	Description
Fields	Variables of the class
Methods	Computations and actions that can be performed by the class
Properties	Actions associated with reading and writing named properties of the class
Events	Notifications that can be generated by the class
Constructors	Actions required to initialize instances of the class or the class itself
Operators	Conversions and expression operators supported by the class
Constants	Constant values associated with the class
Indexers	Actions associated with indexing instances of the class like an array

Member	Description
Finalizers	Actions to perform before instances of the class are permanently discarded
Types	Nested types declared by the class

We can use the required data members while creating a class in c# programming language based on our requirements.

C# Object

In c#, **Object** is an instance of a **class**that can be used to access the data members and member functions of a **class**.

Creating Objects in C#

Generally, we can say that objects are the concrete entities of classes. In c#, we can create objects by using a **new** keyword followed by the class's name like as shown below.

```
Users user = new Users();
```

If you observe the above example, we created an instance (**user**) for the class (**Users**), which we created in the previous section. Now the instance “**user**” is a reference to an object that is based on **Users**. Using the object name “**user**” we can access all the data members and member functions of the **Users** class.

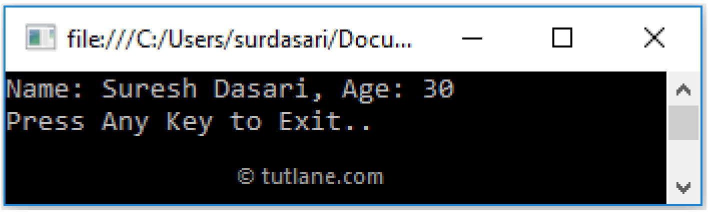
C# Objects Example

Following is the example of creating objects in the c# programming language.

```
using System;

namespace Tutlane
{
 class Program
 {
  static void Main(string[] args)
  {
    Users user = new Users("Suresh Dasari", 30);
    user.GetUserDetails();
    Console.WriteLine("Press Enter Key to Exit..");
    Console.ReadLine();
  }
 }
 public class Users
 {
  public string Name { get; set; }
  public int Age { get; set; }
  public Users(string name, int age)
  {
    Name = name;
    Age = age;
  }
  public void GetUserDetails()
  {
    Console.WriteLine("Name: {0}, Age: {1}", Name, Age);
  }
 }
}
```

If you observe the above example, we created a new class called “**Users**” with a required data members and member functions. To access **Users** class methods and properties, we created an object (**user**) for the**Users** class and performing the required operations.

When we execute the above c# program, we will get the result as shown below.



```
Name: Suresh Dasari, Age: 30
Press Any Key to Exit..
```

Constructor basics

Constructor

In c#, **Constructor** is a method that will invoke automatically whenever an instance of class or **struct** is created. The constructor will have the same name as the class or **struct,** and it useful to initialize and set default values for the data members of the new object.

If we create a class without having any constructor, then the compiler will automatically create a one default constructor for that class. So, there is always one constructor that will exist in every class.

In c#, a class can contain more than one constructor with different types of arguments. The constructors will never return anything, so we don't need to use any return type, not even **void,** while defining the constructor method in the class.

C# Constructor Syntax

As discussed, the constructor is a method, and it won't contain any return type. If you want to create a constructor in c#, then you need to create a method with the class name.

Following is the syntax of creating a constructor in the c# programming language.

```
public class User
{
// Constructor
public User()
{
// Your Custom Code
}
}
```

If you observe the above syntax, we created a class called "User" and a method whose name is same as the class name. Here the method User() will become a constructor of our class.

C# Constructor Types

In c#, we have a different type of constructors available; those are

- Default Constructor
- Parameterized Constructor
- Copy Constructor

- Static Constructor
- Private Constructor

Now we will learn about each constructor in a detailed manner with examples in the c# programming language.

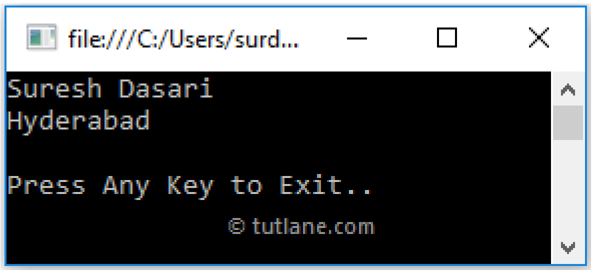
C# Default Constructor

In c#, if we create a constructor without having any parameters, then we will call it a **default constructor,** and every instance of the class will be initialized without any parameter values.

Following is the example of defining the default constructor in the c# programming language.

```
using System;
namespace Tutlane
{
  class User
  {
    public string name, location;
    // Default Constructor
    public User()
    {
      name = "Suresh Dasari";
      location = "Hyderabad";
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      // The constructor will be called automatically once the instance of the class created
      User user = new User();
      Console.WriteLine(user.name);
      Console.WriteLine(user.location);
      Console.WriteLine("\nPress Enter Key to Exit..");
      Console.ReadLine();
    }
  }
}
```

If you observe the above example, we created a class called "**User**" and the constructor method "**User()**" without having any parameters. When we create an instance of our class (**User**), automatically our constructor method will be called.



```
Suresh Dasari
Hyderabad

Press Any Key to Exit..
```

If you observe the above result, our constructor method has called automatically and initialized the parameter values after creating an instance of our class.

C# Parameterized Constructor

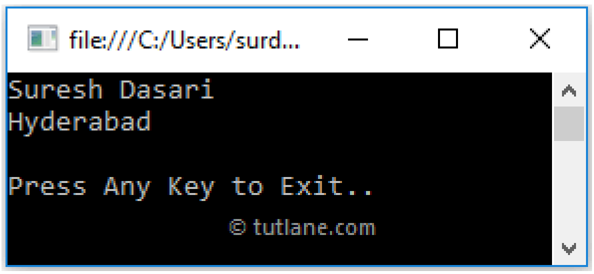
In c#, if we create a constructor with at least one parameter, then we will call it a **parameterized constructor,** and every instance of the class will be initialized with parameter values.

Following is the example of defining the parameterized constructor in the c# programming language.

```
using System;

namespace Tutlane
{
  class User
  {
    public string name, location;
    // Parameterized Constructor
    {
     name = a;
     location = b;
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      // The constructor will be called automatically once the instance of the class created
      User user = new User("Suresh Dasari", "Hyderabad");
      Console.WriteLine(user.name);
      Console.WriteLine(user.location);
      Console.WriteLine("\nPress Enter Key to Exit..");
      Console.ReadLine();
    }
  }
}
```

If you observe the above example, we created a class called “**User**” and the constructor method “**User(string, string)**” with parameters. When we create an instance of our class (**User**) with the required parameters, automatically our constructor method will be called.

When you execute the above c# program, you will get the result as shown below.



```
Suresh Dasari
Hyderabad

Press Any Key to Exit..
```

If you observe the above result, our constructor method has called automatically and initialized the parameter values after creating an instance of our class with the required parameters.

Static Constructors

A static constructor is used to initialize any static data, or to perform a particular action that needs to be performed only once. It is called automatically before the first instance is created or any static members are referenced.

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Static constructors have the following properties:

- A static constructor doesn't take access modifiers or have parameters.
- A class or struct can only have one static constructor.
- Static constructors cannot be inherited or overloaded.
- A static constructor cannot be called directly and is only meant to be called by the common language runtime (CLR). It is invoked automatically.
- The user has no control on when the static constructor is executed in the program.
- A static constructor is called automatically. It initializes the class before the first instance is created or any static members are referenced. A static constructor runs before an instance constructor. A type's static constructor is called when a static method assigned to an event or a delegate is invoked and not when it is assigned. If static field variable initializers are present in the class of the static constructor, they're executed in the textual order in which they appear in the class declaration. The initializers run immediately prior to the execution of the static constructor.

Private Constructors

A private constructor is a special instance constructor. It is generally used in classes that contain static members only. If a class has one or more private constructors and no public constructors, other classes (except nested classes) cannot create instances of this class. For example:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

The declaration of the empty constructor prevents the automatic generation of a parameterless constructor. Note that if you do not use an access modifier with the constructor it will still be private by default. However, the private modifier is usually used explicitly to make it clear that the class cannot be instantiated.

Private constructors are used to prevent creating instances of a class when there are no instance fields or methods, such as the Math class, or when a method is called to obtain an instance of a class. If all the methods in the class are static, consider making the complete class static

Example

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

    public static int currentCount;

    public static int IncrementCount()
    {
        return ++currentCount;
    }
}

class TestCounter
{
    static void Main()
    {

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

C# Copy Constructor

A constructor that creates an object by copying variables from another object or that copies the data of one object into another object is termed as the **Copy Constructor**. It is a parameterized constructor that contains a parameter of the same class type. The main use of copy constructor is to initialize a new instance to the values of an existing instance. Normally, C# does not provide a copy constructor for objects, but if you want to create a copy constructor in your program you can create according to your requirement.

Syntax:

```
class Class_Name {

        // Parameterized Constructor
        public Class_Name(parameter_list)
        {
            // code
        }

        // Copy Constructor
        public Class_Name(Class_Name instance_of_class)
        {
          // code
        }

    }
```

EXAMPLE

```
using System;

namespace simplecopyconstructor {

class technicalscripter {

// variables
private string topic_name;
private int article_no;


// copy constructor
public technicalscripter(technicalscripter tech)
{
topic_name = tech.topic_name;
article_no = tech.article_no;
}


public string Data
{

get
{
```

```
return "The name of topic is: " + topic_name +
" and number of published article is: " +
article_no.ToString();
} } }

public class GFG {

static public void Main()
{
technicalscripter t1 = new technicalscripter(" C# | Copy
Constructor", 38);

technicalscripter t2 = new technicalscripter(t1);

Console.WriteLine(t2.Data);
Console.ReadLine();
}
}
}
```

C# Constructor Overloading

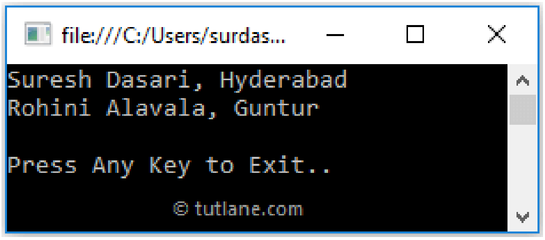
In c#, we can **overload** the constructor by creating another constructor with the same method name but with different parameters.

Following is the example of implementing a constructor overloading in the c# programming language.

```
 using System;
namespace Tutlane
{
  class User
  {
    public string name, location;
    // Default Constructor
    public User() {
      name = "Suresh Dasari";
      location = "Hyderabad";
    }
    // Parameterized Constructor
    public User(string a, string b)
    {
      name = a;
      location = b;
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
     User user = new User(); // Default Constructor will be called
     User user1 = new User("Rohini Alavala", "Guntur"); // Parameterized Constructor will be called
     Console.WriteLine(user.name + ", " + user.location);
     Console.WriteLine(user1.name + ", " + user1.location);
     Console.WriteLine("\nPress Enter Key to Exit..");
     Console.ReadLine();
    }
  }
}
```

If you observe the above example, we created a class called "**User**" and overloaded a constructor "**User()**" by creating another constructor "**User(string, string)**" with the same name but with different parameters.

When you execute the above c# program, you will get the result as shown below.



```
Suresh Dasari, Hyderabad
Rohini Alavala, Guntur

Press Any Key to Exit..
```

If you observe the above result, the respective constructor methods will be called automatically when we create an instance of our class with or without parameters based on our requirements.

C# Constructor Chaining

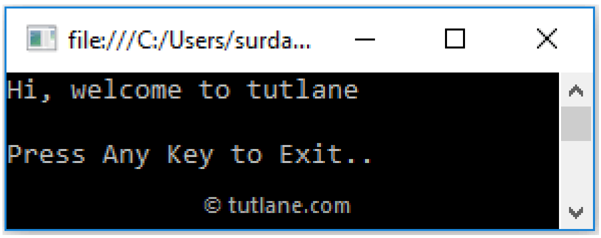
In c#, **Constructor Chaining** is an approach to invoke one constructor from another constructor. To achieve constructor chaining, we need to use `this` keyword after our constructor definition.

Following is the example of implementing a **constructor chaining** in c# programming language.

```
using System;

namespace Tutlane
{
        class User
        {
                public User()
        {
                Console.Write("Hi, ");
        }
                public User(string a): this()
        {
                Console.Write(a);
        }

        public User(string a, string b): this("welcome")
        {
                Console.Write(a + " " + b);
        }
        }

        class Program
        {
         static void Main(string[] args)
                {
         User user1 = new User(" to", "tutlane");
                Console.WriteLine();
                Console.WriteLine("\nPress Enter Key to Exit..");
                 Console.ReadLine();
        }
    }
}
```

If you observe the above example, we created different constructors with different parameters, and we are calling one constructor from another constructor using `this` keyword.

When you execute the above c# program, you will get the result as shown below.



```
Hi, welcome to tutlane

Press Any Key to Exit..
```

If you observe the above result, we are able to call one constructor from another constructor to achieve constructor chaining in the c# programming language.

This is how we can achieve constructor chaining in our applications using the c# programming language.

C# Destructor with Examples

In c#, **Destructor** is a special method of aclass,and it is used in aclassto destroy the object or instances ofclasses. The destructor in c# will invoke automatically whenever the class instances become unreachable.

Following are the properties of destructor in c# programming language.

- In c#, destructors can be used only in classes,and a class can contain only one destructor.
- The destructor inclasscan be represented by using the**tilde** (~) operator
- The destructor in c# won't accept any parameters and access modifiers.
- The destructor will invoke automatically whenever an instance of aclassis no longer needed.
- The destructor is automatically invoked by the garbage collector whenever theclassobjects are no longer needed in the application.

C# Destructor Syntax

Following is the syntax of defining a destructor in the c# programming language.

```
class User
{
 // Destructor
 ~User()
 {
  // your code
 }
}
```

If you observe the above syntax, we created a destructor with the sameclassname using the**tilde** (~) operator. Here, you need to remember that the destructor name must be the same as theclassname in the c# programming language.

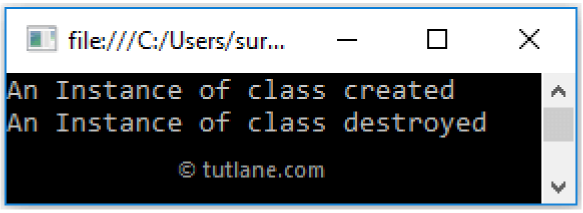
C# Destructor Example

Following is the example of using destructor in c# programming language to destruct the unused objects of a class.

```
using System;

namespace Tutlane
{
  class User
  {
   public User()
   {
     Console.WriteLine("An Instance of class created");
   }
   // Destructor
   ~User()
   {
     Console.WriteLine("An Instance of class destroyed");
   }
  }
  class Program
  {
   static void Main(string[] args)
   {
     Details();
     GC.Collect();
     Console.ReadLine();
   }
   public static void Details()
   {
     // Created instance of the class
     User user = new User();
   }
  }
}
```

If you observe the above example, we created a class with adefault constructor and **destructor**. Here we created an instance of class “**User**” in the**Details()** method, and whenever the **Details** function execution is done, then the garbage collector (**GC**) automatically will invoke a destructor in the**User** class to clear the object of a class.

When you execute the above c# program, you will get the result as shown below.



```
An Instance of class created
An Instance of class destroyed
```

This is how we can use destructor in c# programming language to clear or destruct unused objects based on our requirements.

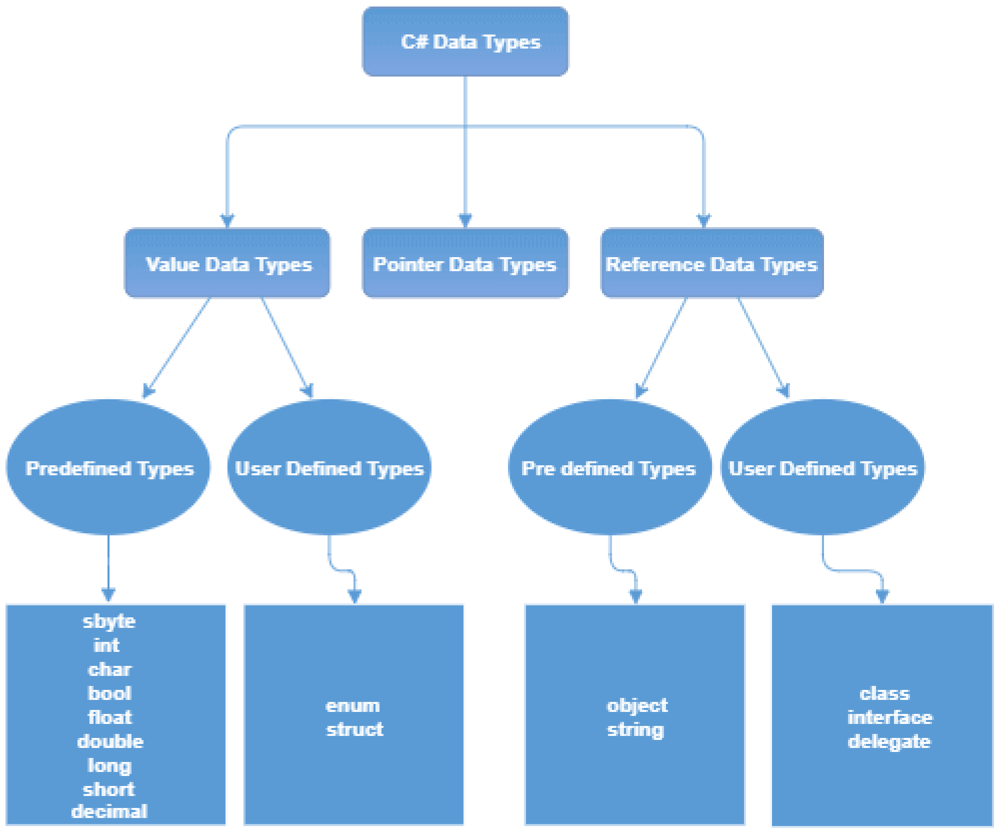
variables initialization syntax,Default assignments & variables scope,

Different Data Types in C#

In C# programming language, we have a 3 different type of data types, those are

Type	Data Types
Value Data Type	int, bool, char, double, float, etc.
Reference Data Type	string, class, object, interface, delegate, etc.
Pointer Data Type	Pointers.

The following diagram will illustrate more detail about different data types in the c# programming language.



C# Value Data Types

In c#, the **Value Data Types** will directly store the variable value in memory. In c#, the value data types will accept both signed and unsigned literals.

The following table lists the value data types in c# programming language with memory size and range of values.

Data Type	.NET Type	Size	Range
byte	Byte	8 bits	0 to 255
sbyte	SByte	8 bits	-128 to 127
int	Int32	32 bits	-2,147,483,648 to 2,147,483,647

Data Type	.NET Type	Size	Range
uint	UInt32	32 bits	0 to 4294967295
short	Int16	16 bits	-32,768 to 32,767
ushort	UInt16	16 bits	0 to 65,535
long	Int64	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	UInt64	64 bits	0 to 18,446,744,073,709,551,615
float	Single	32 bits	-3.402823e38 to 3.402823e38
double	Double	64 bits	-1.79769313486232e308 to 1.79769313486232e308
bool	Boolean	8 bits	True or False
decimal	Decimal	128 bits	(+ or -)1.0 x 10e-28 to 7.9 x 10e28
DateTime	DateTime	-	0:00:00am 1/1/01 to 11:59:59pm 12/31/9999

C# Reference Data Types

In c#, the **Reference Data Types** will contain a memory address of variable value because the reference types won't store the variable value directly in memory.

The following table lists the reference data types in c# programming language with memory size and range of values.

Data Type	.NET Type	Size	Range
string	String	Variable Length	0 to 2 billion Unicode characters
object	Object	-	-

C# Pointer Data Types

In c#, the **Pointer Data Types** will contain a memory address of thevariable value. To get the pointer details we have two symbols ampersand (&) and asterisk (*) in c# language.Following is the syntax of declaring the pointer type in the c# programming language.

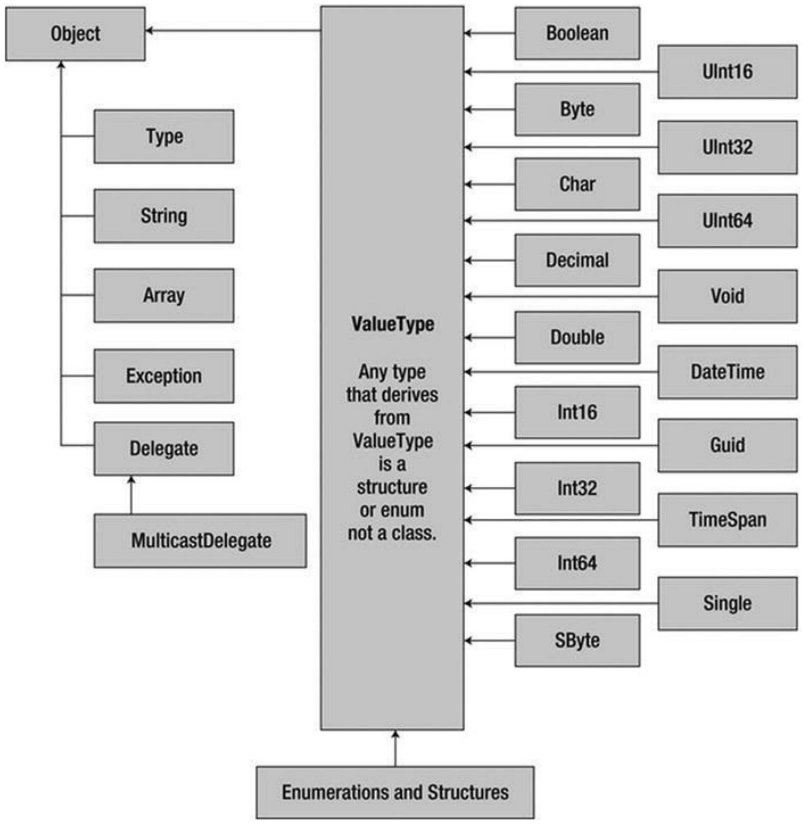
```
type* test;
```

Following is the example of defining the pointer type in the c# programming language.

```
int* a;
int* b;
```

The Data Type Class Hierarchy

It is interesting to note that even the primitive .NET data types are arranged in a class hierarchy. If you are new to the world of inheritance, you will discover the full details.the top of a class hierarchy provide some default behaviors that are granted to the derived types. The relationship between these core system types can be understood as
shown in Figure



C# Variables with Examples

In c#, **Variables** will represent storage locations, and each variable has a particular type that determines what type of values can be stored in the variable.

C# is a **Strongly Typed** programming language. Before we perform any operation on variables, it's mandatory to define a variable with the required data type to indicate what type of data that variable can hold in our application.

Syntax of C# Variables Declaration

Following is the syntax of declaring and initializing variables in the c# programming language.

```
[Data Type] [Variable Name];
[Data Type] [Variable Name] = [Value];
[Access Specifier] [Data Type] [Variable Name] = [Value];
```

If you observe the above syntax, we added a required data type before the variable name to tell the compiler about what type of data the variable can hold or which data type the variable belongs to.

- **[Data Type]** - It's a type of data the variable can hold, such as integer, string, decimal, etc.
- **[Variable Name]** - It's the name of the variable to hold the values in our application.
- **[Value]** - Assigning a required value to the variable.
- **[Access Specifier]** - It is used to define access permissions for the variable.

Now we will see how to define variables in our c# applications with examples.

C# Variables Declaration Example

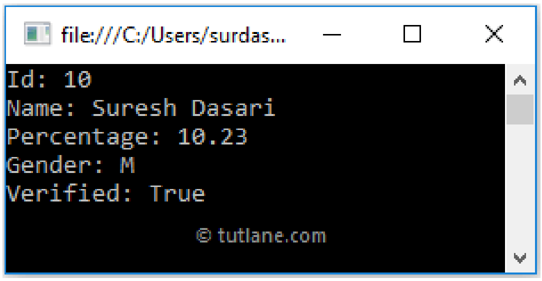
Following is the example of using the variables in the c# programming language.

```
 using System;
namespace Tutlane
{
class Program
            {
           static void Main(string[] args)
           {
           int number = 10;
           string name = "Suresh Dasari";
           double percentage = 10.23;
            char gender = 'M';
           bool isVerified = true;
           Console.WriteLine("Id: " + number);
           Console.WriteLine("Name: " + name);
            Console.WriteLine("Percentage: " + percentage);

           Console.WriteLine("Gender: " + gender);
           Console.WriteLine("Verified: " + isVerified);
            Console.ReadLine();
 }
}
}
```

If you observe the above c# variables example, we defined multiple variables with different data types and assigned values based on our requirements.

Output of C# Variables Declaration Example

When you execute the above program by pressing `Ctrl` + `F5` or clicking on the**Start** option in the menu bar, you will get the result shown below.



```
Id: 10
Name: Suresh Dasari
Percentage: 10.23
Gender: M
Verified: True
```

If you observe the above result, we are able to print the variables in our c# application based on our requirements.

Rules to Declare C# Variables

Before we declare and define variables in the c# programming language, we need to follow particular rules.

- define a variable name with a combination of alphabets, numbers, and underscore.
- A variable name must always start with either alphabet or underscore but not with numbers.
- While defining the variable, no white space is allowed within the variable name.
- Don't use any reserved keywords such as int, float, char, etc., for a variable name.
- In c#, once the variable is declared with a particular data type, it cannot be re-declared with a new type, and we shouldn’t assign a value that is not compatible with the declared type.

The following are some**valid** ways to define the variable names in the c# programming language.

```
int abc;
float a2b;
char _abc;
```

The following are some of the**Invalid** ways of defining the variable names in the c# programming language.

```
int a b c;
float 2abc;
char &abc;
double int;
```

C# Multiple and Multi-Line Variables Declaration

In c#, we can declare and initialize multiple variables of the same data type in a single line by separating with a comma.

Following is the example of defining the multiple variables of the same data type in a single line by separating with a comma in the c# programming language.

```
int a, b, c;
float x, y, z = 10.5;
```

While declaring the multiple variables of the same data type, we can arrange them in multiple lines to make them more readable. The compiler will treat it as a single statement until it encounters a **semicolon** (;).

Following is the simple of defining the multiple variables of the same data type in multiple lines in c# programming language.

```
int a,
  b,
  c;
float x,y,
   z = 10.5;
```

C# Variables Assignment

In c#, once we declare and assign a value to the variable that can be assigned to another variable of the same data type.

```
 int a = 123;
int b = a;
string name = "suresh";
string firstname = name;
```

In c#, it's mandatory to assign a value to the variable before we use it; otherwise, we will get a compile-time error.

If we try to assign a value of **string** data type to an **integer** data type or vice versa, as shown below, we will get an error like "**cannot implicitly convert type int to string**".

```
int a = 123;
string name = a;
```

basic inputs & output with the console class,

Console Class in C#

A console is an operating system window through which a user can communicate with the operating system or we can say a console is an application in which we can give text as an input from the keyboard and get the text as an output from the computer end. The command prompt is an example of a console in the windows and which accept MS-DOS commands. The console contains two attributes named as screen buffer and a console window.
In C#, the Console class is used to represent the standard input, output, and error streams for the console applications. You are not allowed to inherit Console class. This class is defined under

System namespace. This class does not contain any constructor. Instead of the constructor, this class provides different types of properties and methods to perform operations.

Properties

Property	Description
BackgroundColor	Gets or sets the background color of the console.
BufferHeight	Gets or sets the height of the buffer area.
BufferWidth	Gets or sets the width of the buffer area.
CapsLock	Gets a value indicating whether the CAPS LOCK keyboard toggle is turned on or turned off.
CursorLeft	Gets or sets the column position of the cursor within the buffer area.
CursorSize	Gets or sets the height of the cursor within a character cell.
CursorTop	Gets or sets the row position of the cursor within the buffer area.
CursorVisible	Gets or sets a value indicating whether the cursor is visible.
Error	Gets the standard error output stream.
ForegroundColor	Gets or sets the foreground color of the console.
In	Gets the standard input stream.
InputEncoding	Gets or sets the encoding the console uses to read input.
IsErrorRedirected	Gets a value that indicates whether the error output stream has been redirected from the standard error stream.
IsInputRedirected	Gets a value that indicates whether input has been redirected from the standard input stream.
IsOutputRedirected	Gets a value that indicates whether output has been redirected from the standard output stream.
KeyAvailable	Gets a value indicating whether a key press is available in the input stream.
LargestWindowHeight	Gets the largest possible number of console window rows, based on the current font and screen resolution.
LargestWindowWidth	Gets the largest possible number of console window columns, based on the current font and screen resolution.
NumberLock	Gets a value indicating whether the NUM LOCK keyboard toggle is turned on or turned off.
Out	Gets the standard output stream.
OutputEncoding	Gets or sets the encoding the console uses to write output.
Title	Gets or sets the title to display in the console title bar.
TreatControlCAsInput	Gets or sets a value indicating whether the combination of the Control modifier key and C console key (Ctrl+C) is treated as ordinary input or as an interruption that is handled by the operating system.
WindowHeight	Gets or sets the height of the console window area.
WindowLeft	Gets or sets the leftmost position of the console window area relative to the screen buffer.
WindowTop	Gets or sets the top position of the console window area relative to the screen buffer.
WindowWidth	Gets or sets the width of the console window.

Example:

```
// C# program to illustrate how to get
// Background and Foreground color
// of the console
using System;

public class GFG {

static public void Main()
{

// Get the Background and foreground
// color of Console Using BackgroundColor
// and ForegroundColor property of Console
Console.WriteLine("Background color:{0}",
Console.BackgroundColor);

Console.WriteLine("Foreground color : {0}",
Console.ForegroundColor);
}
}
```

Output:

```
Background color : Black
Foreground color : Black
```

Methods

Method	Description
Beep()	Plays the sound of a beep through the console speaker.
Clear()	Clears the console buffer and corresponding console window of display information.
MoveBufferArea()	Copies a specified source area of the screen buffer to a specified destination area.
OpenStandardError()	Acquires the standard error stream.
OpenStandardInput()	Acquires the standard input stream.
OpenStandardOutput()	Acquires the standard output stream.
Read()	Reads the next character from the standard input stream.
ReadKey()	Obtains the next character or function key pressed by the user. The pressed key is displayed in the console window.
ReadLine()	Reads the next line of characters from the standard input stream.
ResetColor()	Sets the foreground and background console colors to their defaults.
SetBufferSize(Int32, Int32)	Sets the height and width of the screen buffer area to the specified values.
SetCursorPosition(Int32,	Sets the position of the cursor.

Int32)	
SetError(TextWriter)	Sets the Error property to the specified TextWriter object.
SetIn(TextReader)	Sets the In property to the specified TextReader object.
SetOut(TextWriter)	Sets the Out property to the specified TextWriter object.
SetWindowPosition(Int32, Int32)	Sets the position of the console window relative to the screen buffer.
SetWindowSize(Int32, Int32)	Sets the height and width of the console window to the specified values.
Write()	Writes the text representation of the specified value or values to the standard output stream.
WriteLine()	Writes the specified data, followed by the current line terminator, to the standard output stream.

C# Basic Input and Output

In this tutorial, we will learn how to take input from user and and display output in C# using various methods

C# Output

In order to output something in C#, we can use

```
System.Console.WriteLine() OR
System.Console.Write()
```

Here, `System` is a namespace, `Console` is a class within namespace `System` and `WriteLine` and `Write` are methods of class `Console`.

Let's look at a simple example that prints a string to output screen.

Example 1: Printing String using WriteLine()

```
using System;

namespace Sample
{
	class Test
	{
		public static void Main(string[] args)
		{
			Console.WriteLine("C# is cool");
		}
	}
}
```

When we run the program, the output will be

```
C# is cool
```

Difference between WriteLine() and Write() method

The main difference between `WriteLine()` and `Write()` is that the `Write()` method only prints the string provided to it, while the `WriteLine()` method prints the string and moves to the start of next line as well.

Let's take at a look at the example below to understand the difference between these methods.

Example 2: How to use WriteLine() and Write() method?

```
using System;

namespace Sample
{
	class Test
	{
		public static void Main(string[] args)
		{
			Console.WriteLine("Prints on ");
			Console.WriteLine("New line");

			Console.Write("Prints on ");
			Console.Write("Same line");
		}
	}
}
```

When we run the program, the output will be

```
Prints on
New line
Prints on Same line
```

Printing Variables and Literals using WriteLine() and Write()

The `WriteLine()` and `Write()` method can be used to print variables and literals. Here's an example.

Example 3: Printing Variables and Literals

```
using System;

namespace Sample
{
	class Test
	{
		public static void Main(string[] args)
		{
			int value = 10;

			// Variable
			Console.WriteLine(value);
			// Literal
			Console.WriteLine(50.05);
		}
	}
}
```

When we run the program, the output will be

```
10
50.05
```

Combining (Concatenating) two strings using + operator and printing them

Strings can be combined/concatenated using the **+** operator while printing.

Example 4: Printing Concatenated String using + operator

```
using System;

namespace Sample
{
	class Test
	{
		public static void Main(string[] args)
		{
			int val = 55;
			Console.WriteLine("Hello " + "World");
			Console.WriteLine("Value = " + val);
		}
	}
}
```

When we run the program, the output will be

```
Hello World
Value = 55
```

Printing concatenated string using Formatted String [Better Alternative]

A better alternative for printing concatenated string is using formatted string. Formatted string allows programmer to use placeholders for variables. For example,

The following line,

```
Console.WriteLine("Value = " + val);
```

can be replaced by,

```
Console.WriteLine("Value = {0}", val);
```

`{0}` is the placeholder for variable *val* which will be replaced by value of *val*. Since only one variable is used so there is only one placeholder.

Multiple variables can be used in the formatted string. We will see that in the example below.

Example 5: Printing Concatenated string using String formatting

```
using System;

namespace Sample
{
	class Test
```

```
    {
        public static void Main(string[] args)
        {
            int firstNumber = 5, secondNumber = 10, result;
            result = firstNumber + secondNumber;
            Console.WriteLine("{0} + {1} = {2}", firstNumber, secondNumber, result);
        }
    }
}
```

When we run the program, the output will be

```
5 + 10 = 15
```

Here, `{0}` is replaced by *firstNumber*, `{1}` is replaced by *secondNumber* and `{2}` is replaced by *result*. This approach of printing output is more readable and less error prone than using `+` operator.

To know more about string formatting, visit *C# string formatting*.

C# Input

In C#, the simplest method to get input from the user is by using the `ReadLine()` method of the `Console` class. However, `Read()` and `ReadKey()` are also available for getting input from the user. They are also included in `Console` class.

Example 6: Get String Input From User

```
using System;

namespace Sample
{
	class Test
	{
		public static void Main(string[] args)
		{
			string testString;
			Console.Write("Enter a string - ");
			 testString = Console.ReadLine();
			Console.WriteLine("You entered '{0}'", testString);
		}
	}
}
```

When we run the program, the output will be:

```
Enter a string - Hello World
You entered 'Hello World'
```

Difference between ReadLine(), Read() and ReadKey() method:

The difference between `ReadLine()`, `Read()` and

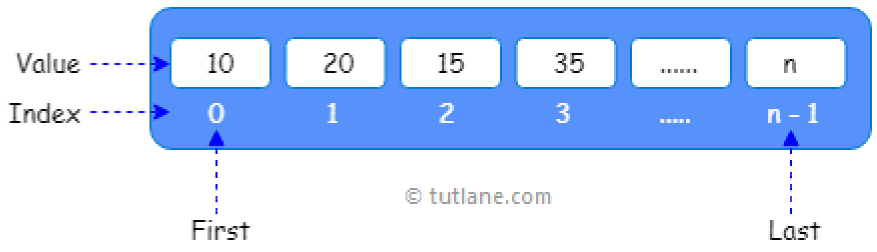
C# Arrays

In c#, **Arrays** are useful for storing multiple elements of the same data type at contiguous memory locations and arrays. It will store a fixed number of elements sequentially based on the predefined number of items.

In the previous chapter, we learned about variables in c#, which will help us hold a single value like **int x = 10;**.If we want to hold more than one value of the same data type, then an array came into the picture in c# to solve this problem.

An array can start storing the values from index **0**. If we have an array with n elements, it will start storing the elements from index **0** to **n-1**.

Following is the pictorial representation of storing the multiple values of the same type in the c# array data structure.



If you observe the above diagram, we are storing the values in an array starting from index **0,** and it will continue to store the values based on the defined number of elements.

C# Arrays Declaration

In c#, **Arrays** can be declared by specifying the type of elements followed by the square brackets **[]** like as shown below.

```
type[] array_name;
```

Here, the **type** is nothing but adata typeof elements to store in an array, and **array_name** represents an array's name.

For example, the following are the different ways of declaring an array with differentdata typesin the c# programming language.

```
// Store only int values
int[] numbers;
//Store only string values
string[] names;
//Store only double values
double[] ranges;
```

If you observe the above examples, we declared arrays with the required data type based on our requirements.

In c#, the array elements can be of any type, and by default, the values of numeric array elements are set to zero, and the reference elements are set to null.

C# Arrays Initialization

In c#, Arrays can be initialized by creating an instance of the array with a**new** keyword. Using a**new** keyword, we can declare and initialize an array at the same time based on our requirements.

Following are the different ways of declaring and initializing array elements by using the**new** keyword in the c# programming language.

```
// Declaring and Initializing an array with size of 5
int[] array = new int[5];
//Defining and assigning an elements at the same time
int[] array2 = new int[5]{1,2,3,4,5};
//Initialize with 5 elements will indicates the size of an array
int[] array3 = new int[] { 1, 2, 3, 4, 5 };
// Another way to initialize an array without size
int[] array4 = { 1, 2, 3, 4, 5 };
// Declare an array without initialization
int[] array5;
array5 = new int[]{ 1, 2, 3, 4, 5 };
```

In the first statement, we declared and initialized an integer array with the size of **5** to allow an array to store **5** integer values. The array can contain elements from **array[0]** to **array[4]**.

In the second statement, we declared and initialized an array same as the first statement and assigned values to each index followed by curly brackets **{ }**.

In a third or fourth statement, while declaration, we initialized an array with values without specifying any size. Here, the size of an array can be determined by the number of elements, so the size initializer is not required if we assign elements during the initialization.

In c#, we can declare an array variable without initialization, but we must use the**new** keyword to assign an array to the variable.

In the fifth statement, we declared an array without initialization, and we used a **new** keyword to assign array values to the variable.

In c#, after an array declaration, we can initialize array elements using index values. Following is an example of declaring and initializing array elements using individual index values in c#.

```
int[] array = new int[5];
array[0] = 1;
array[1] = 2;
array[2] = 3;
array[3] = 4;
array[4] = 5;
```

If you observe the above example, we are initializing an array of elements individually using individual index values.

Generally, in c# initializing an array without **size** or assigning values to an array without a**new** operator will throw compile-time errors. For example:

```
// Error. Initialize an array without size
int[] array = new int[];
// Error. Initialize an array without new keyword
int[] array1;
array1 = { 1, 2, 3, 4, 5 };
```

If you observe the above examples, we initialized an array without any **size in the first statement**. Inthe second statement, we declared and initializing array elements without using the**new** keyword. These two statements will throw compile-time errors in our c# applications.

C# Accessing an Array Elements

In c#, we can access array elements using for loop or foreach loop or with particular index numbers.

Following is the code snippet of accessing array elements by using particular index numbers.

```
int[] array = new int[5] { 1, 2, 3, 4, 5 };
int a = array[1]; // It will return 2
int b = array[4]; // It will return 5
```

If you observe the above code, we are trying to access an array of elements using index values in c#.

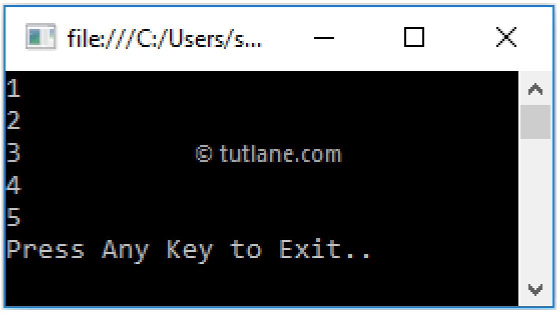
Following is the example of declaring, initializing, and accessing array elements with particular index numbers in the c# programming language.

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

```
 {
  static void Main(string[] args)
  {
    int[] array = new int[5] { 1, 2, 3, 4, 5 };
    Console.WriteLine(array[0]);
    Console.WriteLine(array[1]);
    Console.WriteLine(array[2]);
    Console.WriteLine(array[3]);
    Console.WriteLine(array[4]);
    Console.WriteLine("Press Enter Key to Exit..");
    Console.ReadLine();
  }
 }
}
```

If you observe the above example, we declared and initialized an array with 5 elements, and we are accessing an array of elements using index values.

When we execute the above c# program, we will get the result as shown below.



```
1
2
3
4
5
Press Any Key to Exit..
```

If you observe the above result, we are able to access array elements using index numbers based on our requirements.

C# Access Array Elements with For Loop

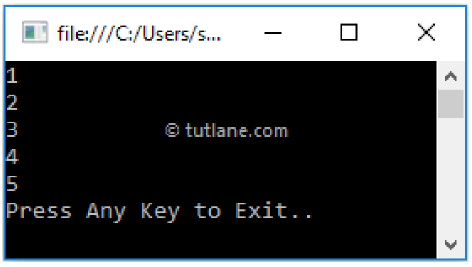
In c#, by usingfor loopwe can iterate through array elements and access the values of an array with length property.

Following is the example of accessing array elements using for loop in c# programming language.

```
using System;

namespace Tutlane
{
  class Program
  {
    static void Main(string[] args)
    {
      int[] array = new int[5] { 1, 2, 3, 4, 5 };
      for (int i = 0; i < array.Length; i++)
      {
        Console.WriteLine(array[i]);
      }
      Console.WriteLine("Press Enter Key to Exit..");
      Console.ReadLine();
    }
  }
}
```

If you observe the above example, we are looping through array elements withfor loopto access array elements based on our requirements.

When we execute the above c# program, we will get the result as shown below.



```
1
2
3
4
5
Press Any Key to Exit..
```

If you observe the above result, we are able to loop through elements in an array withfor loopand print array values based on our requirements.

C# Access Array Elements with Foreach Loop

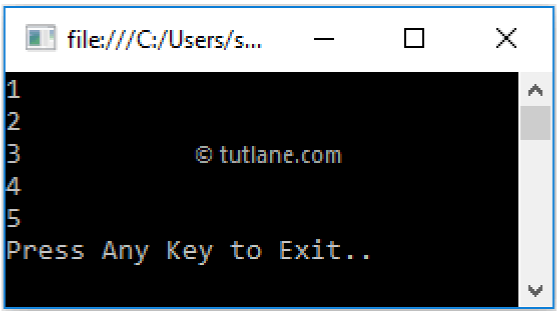
In c#, same asfor loop,we can use theforeach loopto iterate through array elements and access the values of an array based on our requirements.

Following is the example of accessing array elements using aforeach loopin the c# programming language.

```
using System;

namespace Tutlane
{
  class Program
  {
    static void Main(string[] args)
    {
      int[] array = new int[5] { 1, 2, 3, 4, 5 };
      foreach(int i in array)
      {
        Console.WriteLine(i);
      }
      Console.WriteLine("Press Enter Key to Exit..");
      Console.ReadLine();
    }
  }
}
```

If you observe the above example, we are looping through array elements with aforeach loopto access array elements based on our requirements.

When we execute the above c# program, we will get the result as shown below.



```
1
2
3
4
5
Press Any Key to Exit..
```

If you observe the above result, we are able to loop through elements in an array withforeach loopand print array values based on our requirements.

This is how we can access array elements in the c# programming language based on our requirements.

C# Array Types

In c#, we have a different type of arrays available; those are

- Single-Dimensional Arrays
- Multi-Dimensional Arrays
- Jagged Arrays

Jagged array is a **array** of **arrays** such that member **arrays** can be of different sizes. In other words, the length of each **array** index can differ.

C# Array Class

In c#, we have a class called **Array,** and it will act as a base class for all the arrays in the common language runtime (CLR). The Array class provides methods for creating, manipulating, searching, and sorting arrays.

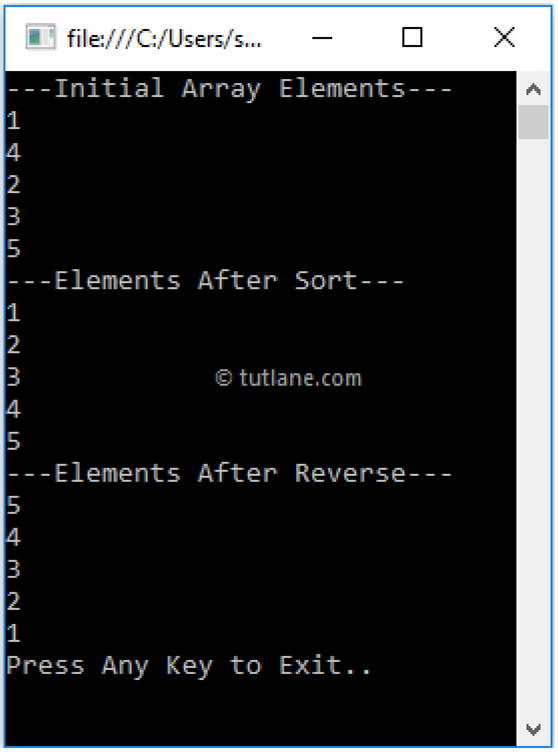
For example, by using the **Sort** or **Copy** methods of the **Array** class, we can sort the elements of an array and copy the elements of one array to another based on our requirements.

Following is the example of using an Array class to sort or filter or reverse array elements in the c# programming language.

```
using System;
namespace Tutlane
{
  class Program
  {
    static void Main(string[] args)
    {
      int[] array = new int[5] { 1, 4, 2, 3, 5 };
      Console.WriteLine("---Initial Array Elements---");
      foreach (int i in array)
      {
       Console.WriteLine(i);
      }
      Array.Sort(array);
      Console.WriteLine("---Elements After Sort---");
      foreach (int i in array)
      {
       Console.WriteLine(i);
      }
      Array.Reverse(array);
      Console.WriteLine("---Elements After Reverse---");
      foreach (int i in array)
      {
       Console.WriteLine(i);
      }
      Console.WriteLine("Press Enter Key to Exit..");
      Console.ReadLine();
    }
  }
}
```

If you observe the above example, we are sorting and changing the order of array elements using **Sort** and **Reverse** methods of an **Array** class.

When we execute the above c# program, we will get the result as shown below.



```
---Initial Array Elements---
1
4
2
3
5
---Elements After Sort---
1
2
3
4
5
---Elements After Reverse---
5
4
3
2
1
Press Any Key to Exit..
```

If you observe the above result, we sorted the array elements and changed the order of array elements using the Array class based on our requirements.

This is how we can use the required methods of **Array** class in our applications to implement the required functionality.

C# String with Examples

In c#, the**string** is a keyword that is useful to represent a sequential collection of characters called a text, and the string is an object of the**System.String** type.

In c#, we use **string** keyword to create string variablest o hold the particular text, which is a sequential collection of characters.

The stringvariablesin c# can hold any kind of text, and by using the **Length** property, we can know that the number of characters the string variableis holding based on our requirements.

C# String Declaration and Initialization

The following are the different ways of declaring and initializing string variablesusing **string** keyword in the c# programming language.

```
// Declare without initializing.
string str1;
// Declaring and Initializing
string str2 = "Welcome to Tutlane";
String str3 = "Hello World!";
// Initialize an empty string.
string str4 = String.Empty;
// Initialize to null.
String str5 = null;
// Creating a string from char
char[] letters = { 'A', 'B', 'C' };
string str6 = new string(letters);
```

If you observe the above code snippet, we created stringvariablesusing **string** and **String** keywords with or without initializing values based on our requirements.

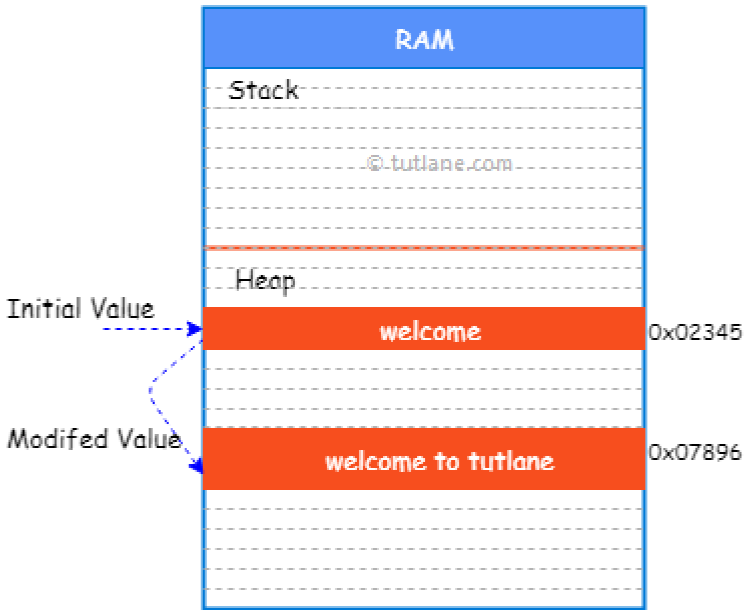
C# string vs. String

If you observe the above declarations, we used two keywords called **string** and **String** to declare string variables. In c#, the **string** keyword is just an alias for **String,** so both **string** and **String** are equivalent, and you can use whichever naming convention you prefer to define string variables.

C# String Immutability

In c#, the**string** is **immutable**, which means the string object cannot be modified once it is created. If any changes are made to the string object, like adding or modifying an existing value, it will simply discard the old instance in memory and create a new instance to hold the new value.

For example, when we create a new string variable "**msg**" with the text "**welcome**", a new instance will create a heap memory to hold this value. Now, if we make any changes to the**msg** variable, like changing the text from "**welcome**" to "**welcome to tutlane**", then the old instance on heap memory will be discarded, and another instance will create on heap memory to hold the variable value instead of modifying the old instance in the memory.



In c#, if we perform modifications like inserting, concatenating, removing, or replacing a value of the existing string multiple times, every time the new instance will create on heap memory to hold the new value, so automatically the performance of the application will be affected.

C# String Literals (Regular, Verbatim)

In c#, **string literal** is a sequence of characters enclosed in double quotation marks (**" "**). We have two kinds of string literals available in c#; those are **regular** and **verbatim**. The **regular** literals are useful when we want to embed escape characters like \n, \t, \', \", etc. in c#.

Regular :

```
string names = "Suresh\nRohini\nTrishika";
Console.WriteLine(names);
/*
Output:
Suresh
Rohini
Trishika
*/
string msg = "Welcome to \"tutlane\" world";
Console.WriteLine(msg);
// Output: Welcome to "tutlane" world
```

verbatim :

In c#, the special character @ will serve as **verbatim** literal, and it is useful to represent a multiline string or a string with backslash characters, for example, to represent file paths.

Following is an example of using verbatim literal @ in c# programming language to represent a multiline string and a file path.

```
string path = @"C:\Users\Tutlane\Documents\";
Console.WriteLine(path);
//Output: C:\Users\Tutlane\Documents\

string msg = @"Hi Guest,
Welcome to Tutlane World
Learning Made Easy";
Console.WriteLine(msg);
/* Output:
Hi Guest,
Welcome to Tutlane World
Learning Made Easy
*/

string msg1 = @"My daughter name was ""Trishika.""";
Console.WriteLine(msg1);
//Output: My daughter name was "Trishika."
```

If you observe the above examples, we used a verbatim character @ to represent multiline strings or backslash character strings, etc., based on our requirements.

C# Format Strings

In c#, the format string is a string whose contents can be determined dynamically at runtime. We can create a format string using the**Format** method and embedding placeholders in braces that will be replaced by other values at runtime.

Following is the example of using a format string to determine the string content dynamically at runtime in the c# programming language.

```
string name = "Suresh Dasari";
string location = "Hyderabad";
string user = string.Format("Name: {0}, Location: {1}", name, location);
Console.WriteLine(user);
// Output: Name: Suresh Dasari, Location: Hyderabad
```

If you observe the above code, we are formatting a string using the**Format** method and replacing the placeholders in braces with required values.

C# Access Individual Characters from Strings

In c#, we can access individual characters from string by using array notation with index values.

Following is the example of accessing individual characters from the string by specifying the index position.

```
string name = "Suresh Dasari";
for (int i = 0; i < name.Length; i++)
{
Console.Write(name[i]);
}
// Output: Suresh Dasari
```

If you observe the above example, we are looping through characters in a string using for loop and displaying each character of string based on the index position.

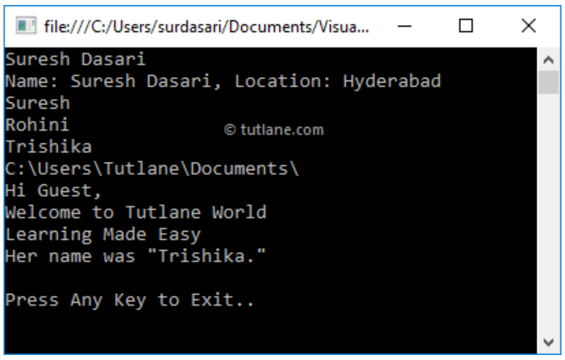
C# String Example

Following is the example declaring and initializing strings, formatting string value, and use string literals to represent data in the c# programming language.

```
using System;
namespace Tutlane
{
  class Program
  {
    static void Main(string[] args)
    {
      string firstname = "Suresh";
      string lastname = "Dasari";
      string location = "Hyderabad";
      string name = firstname + " " + lastname;
      Console.WriteLine(name);
      string userInfo = string.Format("Name: {0}, Location: {1}", name, location);
      Console.WriteLine(userInfo);
      string names = "Suresh\nRohini\nTrishika";
      Console.WriteLine(names);
      string path = @"C:\Users\Tutlane\Documents\";
      Console.WriteLine(path);
      string msg = @"Hi Guest,      Welcome to Tutlane World       Learning Made Easy";
      Console.WriteLine(msg);
      string msg1 = @"Her name was ""Trishika.""";
      Console.WriteLine(msg1);
      Console.WriteLine("\nPress Enter Key to Exit..");
      Console.ReadLine();
    }
  }
}
```

If you observe the above code, we declared and initialized string variables, formatted string using **Format** method, and used string literals to embed escape characters, etc., based on our requirements.

When you execute the above c# program, you will get the result as shown below.



```
Suresh Dasari
Name: Suresh Dasari, Location: Hyderabad
Suresh
Rohini
Trishika
C:\Users\Tutlane\Documents\
Hi Guest,
Welcome to Tutlane World
Learning Made Easy
Her name was "Trishika."

Press Any Key to Exit..
```

C# String Properties

The following table lists the available string properties in the c# programming language.

Property	Description
Chars	It helps us to get the characters from the current string object based on the specified position.
Length	It returns the number of characters in the current String object.

C# String Methods

In c#, the string class contains various methods to manipulate string objects based on our requirements.The following table lists important string methods available in the c# programming language.

Method	Description
Clone()	It returns a reference to this instance of String.
Compare(String, String)	It compares two specified String objects and returns an integer that indicates their relative position in the sort order.
Concat(String, String)	It concatenates two specified instances of String.
Contains(String)	It returns a value indicating whether a specified substring occurs within this string.
Copy(String)	It creates a new instance of String with the same value as a specified String.
Format(String, Object)	It replaces one or more format items in a specified string with the string representation of a specified object.
Trim()	It removes all leading and trailing white-space characters from the current String object.
ToLower()	It converts a given string to a lowercase.
ToUpper()	It converts a given string to uppercase.
Split(Char[])	It splits a string into substrings that are based on the characters in an array.
Substring(Int32)	It retrieves a substring from this instance. The substring starts at a specified character position and continues to the end of the string.

EXAMPLES

1. Splitting a String by another string

```
string str = "this—is--a--complete--sentence";
string[] tokens = str.Split(new[] { "--" }, StringSplitOptions.None);

Result:
[ "this", "is", "a", "complete", "sentence" ]
```

Getting Substrings of a given string

```
string helloWorld = "Hello World!";
string world = helloWorld.Substring(6); //world = "World!"
string hello = helloWorld.Substring(0,5); // hello = "Hello"
```

Replacing a string within a string
Using the **System.String.Replace method,**

```
string s = "Hello World";
s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

String Contains String

```
using System;

namespace StringApplication {

   class StringProg {

      static void Main(string[] args) {
         string str = "This is test";

         if (str.Contains("test")) {
            Console.WriteLine("The sequence 'test' was found.");
         }
         Console.ReadKey() ;
      }
   }
}
```

C# Static Keyword

In c#, **static** is a keyword or a modifier that is useful to make a class or methods or variable properties, not instantiable which means we cannot instantiate the items which we declared with a `static` modifier.

The **static** members which we declared can be accessed directly with a type name. Suppose if we apply a `static` modifier to a class property or a method or variable, we can access those static members directly with a class name instead of creating an object of a class to access those properties.

C# Static Variables

Following is the example of defining a class with static properties, and those can be accessed directly with the type instead of a specific object name.

```
class User
{
public static string name, location;
public static int age;
}
```

If you observe the above example, we defined variables with `static` keyword, and we can access those variables directly with a type name like **User.name** or **User.location** and **User.age**.

Following is the example of accessing the variables directly with a type name in the c# programming language.

```
Console.WriteLine(User.name);
Console.WriteLine(User.location);
Console.WriteLine(User.age);
```

If you observe the above statements, we are accessing our **static** properties directly using the class name instead of the class instance.

Generally, in c# the instance of a class will contain a separate copy of all instance fields so that the memory consumption will increase automatically, but if we use `static` modifier, there is only one copy of each field, so automatically, the memory will be managed efficiently.

In c#, we can use `static` modifier with classes, methods, properties, constructors, operators, fields, and events, but it cannot be used with indexers, **finalizers,** or types other than classes.

C# Static Keyword Example

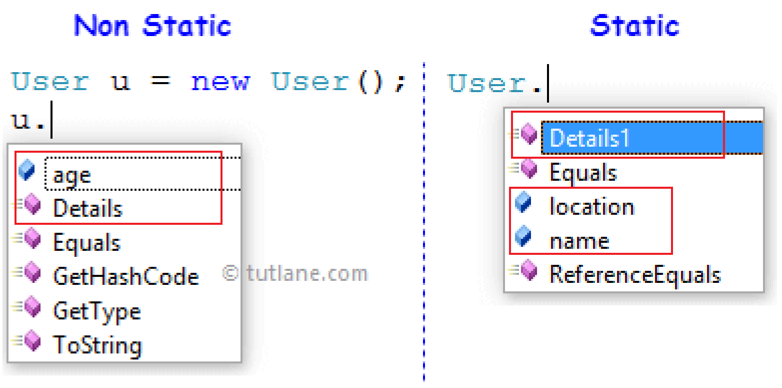
Following is the example of creating a class by including both static and non-static variables & methods. Here we can access non-static variables and methods by creating an instance of the class, but it won't allow us to access the static fields with an instance of the class so the static variables and methods can be accessed directly with the class name.

```
using System;

namespace Tutlane
{
  class User
  {
   // Static Variables
   public static string name, location;
   //Non Static Variable
   public int age;
   // Non Static Method
   public void Details()
   {
     Console.WriteLine("Non Static Method");
   }
   // Static Method
   public static void Details1()
   {
     Console.WriteLine("Static Method");
   }
  }
  class Program
  {
   static void Main(string[] args)
   {
     User u = new User();
     u.age = 32;
     u.Details();
     User.name = "Suresh Dasari";
     User.location = "Hyderabad";
     Console.WriteLine("Name: {0}, Location: {1}, Age: {2}", User.name, User.location, u.age);
     User.Details1();
     Console.WriteLine("\nPress Enter Key to Exit..");
     Console.ReadLine();
```

```
    }
  }
}
```

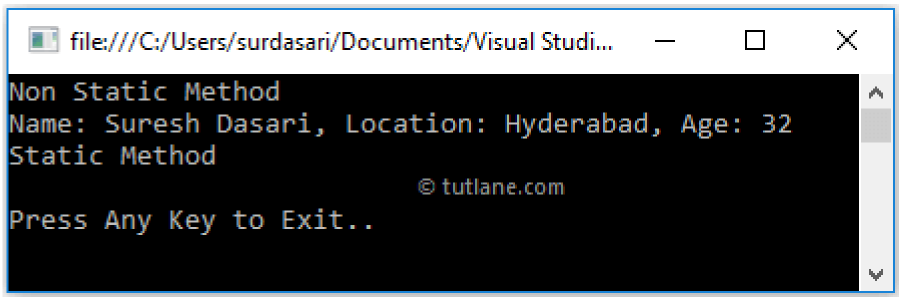
If you observe the above example, we created a class called "**User**" with **static** and **non-static** variables & methods. Here we are accessing **non-static** variables and methods with an instance of **User** class, and **static** fields & methods are able to access directly with the class name (**User**).

The following diagram will illustrate more details about how **static** and **non-static** variables & methods can be accessed in our c# application.



If you observe the above diagram, it clearly says that **non-static** fields and methods can be accessed only with an instance of the class, and the **static** fields & methods can be accessed directly with the class name.

When you run the above c# program, you will get the result as shown below.



```
Non Static Method
Name: Suresh Dasari, Location: Hyderabad, Age: 32
Static Method

Press Any Key to Exit..
```

This is how you can use `static` keyword in our c# applications to make aclassormethodsor variable properties as not instantiable based on our requirements.

C# Static Class with Examples

In c#, a **static class** can be created by using `static` modifier and the static class can contain only static members.

Generally, the **static class** is same as the **non-static class**, but the only difference is the **static class** cannot be instantiated. Suppose if we apply `static` modifier to a class, we don't require to use the **new** keyword to create a class type variable.

Another difference is the **static class** will contain only static members, but the **non-static class** can contain both static and non-static members.

C# Static Class Syntax

In c#, we can create a **static class** by applying `static` keyword to the class like as shown below.

```
static class sample
{
  //static data members
  //static methods
}
```

If you observe the above syntax, we applied`static`keyword to the class typeto create a static class called “**sample**”. The methods and data members that we are going to implement in the sample class must be**static**.

In c#, we can access members of a static class directly with the class name. For example, we have a static class called “**User**” with a method “**Details()**” that we can access like **User.Details()**.

C# Static Class Example

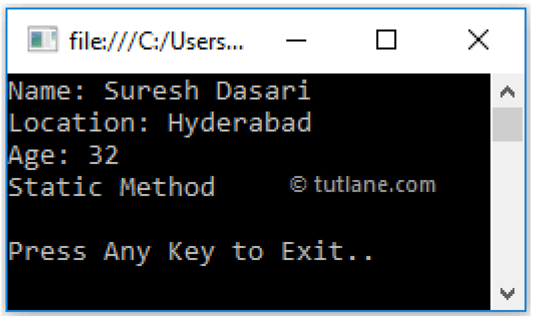
Following is the example of defining a **static class** to access data members and member functions without creating an instance of the class in the c# programming language.

```
using System;

namespace Tutlane
{
  static class User
  {
    // Static Variables
    public static string name;
    public static string location;
    public static int age;
    // Static Method
```

```
   public static void Details()
   {
     Console.WriteLine("Static Method");
   }
 }
 class Program
 {
   static void Main(string[] args)
   {
     User.name = "Suresh Dasari";
     User.location = "Hyderabad";
     User.age = 32;
     Console.WriteLine("Name: {0}", User.name);
     Console.WriteLine("Location: {0}", User.location);
     Console.WriteLine("Age: {0}", User.age);
     User.Details();
     Console.WriteLine("\nPress Enter Key to Exit..");
     Console.ReadLine();
   }
 }
}
```

If you observe the above example, we are accessing **static** class members and functions directly with the class name because we cannot instantiate the **static class**.

When you execute the above c# program, you will get the result as shown below.



```
Name: Suresh Dasari
Location: Hyderabad
Age: 32
Static Method

Press Any Key to Exit..
```

This is how we can create a static class and use it in our c# applications based on our requirements.

C# Static Class Features

Following are the main features of static class in c# programming language.

The static class in c# will contain only **static** members.

- In c#, the static classes cannot be instantiated.
- C# static classes are **sealed,**sothey cannot be inherited.
- The static classes in c# will not contain instance constructors.

As discussed in the static keywordarticle,we can use **static** members in **non-static** classes such as normalclasses. For normalclasses,you can create an instance of a class using the**new** keyword to access non-static members and functions, but it cannot access the **static** members and functions.

To know more about it, refer tostatic keyword in c# with examples.

The advantage of using static classes in c# applications will make sure that instances of classes cannot be created.

Static Methods

You can define one or more static methods in a non-static class. Static methods can be called without creating an object. You cannot call static methods using an object of the non-static class.

The static methods can only call other static methods and access static members. You cannot access non-static members of the class in the static methods.

Example: Static Method

```
class Program
{
   static int counter = 0;
   string name = "Demo Program";

   static void Main(string[] args)
   {
      counter++; // can access static fields
      Display("Hello World!"); // can call static methods

      name = "New Demo Program"; //Error: cannot access non-static members
      SetRootFolder("C:\MyProgram"); //Error: cannot call non-static method
   }

   static void Display(string text)
```

```
    {
        Console.WriteLine(text);
    }

    public void SetRootFolder(string path) {  }
}
```

Rules for Static Methods

1. Static methods can be defined using the `static` keyword before a return type and after an access modifier.
2. Static methods can be overloaded but cannot be overridden.
3. Static methods can contain local static variables.
4. Static methods cannot access or call non-static variables unless they are explicitly passed as parameters.

C# - Static Class, Methods, Constructors, Fields

In C#, static means something which cannot be instantiated. You cannot create an object of a static class and cannot access static members using an object.

C# classes, variables, methods, properties, operators, events, and constructors can be defined as static using the `static` modifier keyword.

Static Class

Apply the `static` modifier before the class name and after the access modifier to make a class static. The following defines a static class with static fields and methods.

Example: C# Static Class

```
public static class Calculator
{
    private static int _resultStorage = 0;

    public static string Type = "Arithmetic";

    public static int Sum(int num1, int num2)
    {
        return num1 + num2;
    }

    public static void Store(int result)
    {
        _resultStorage = result;
    }
}
```

Above, the `Calculator` class is a static. All the members of it are also static.

You cannot create an object of the static class; therefore the members of the static class can be accessed directly using a class name like `ClassName.MemberName`, as shown below.

Example: Accessing Static Members

```
class Program
{
    static void Main(string[] args)
    {
        var result = Calculator.Sum(10, 25); // calling static method
        Calculator.Store(result);

        var calcType = Calculator.Type; // accessing static variable
        Calculator.Type = "Scientific"; // assign value to static variable
    }
}
```

Rules for Static Class

1. Static classes cannot be instantiated.
2. All the members of a static class must be static; otherwise the compiler will give an error.
3. A static class can contain static variables, static methods, static properties, static operators, static events, and static constructors.
4. A static class cannot contain instance members and constructors.
5. Indexers and destructors cannot be static
6. `var` cannot be used to define static members. You must specify a type of member explicitly after the `static` keyword.
7. Static classes are sealed class and therefore, cannot be inherited.
8. A static class cannot inherit from other classes.
9. Static class members can be accessed using `ClassName.MemberName`.
10. A static class remains in memory for the lifetime of the application domain in which your program resides.

Static Members in Non-static Class

The normal class (non-static class) can contain one or more static methods, fields, properties, events and other non-static members.

It is more practical to define a non-static class with some static members, than to declare an entire class as static.

Static Fields

Static fields in a non-static class can be defined using the `static` keyword.

Static fields of a non-static class is shared across all the instances. So, changes done by one instance would reflect in others.

Example: Shared Static Fields

```
public class StopWatch
{
   public static int InstanceCounter = 0;
   // instance constructor
   public StopWatch()
   {
   }
}

class Program
{
   static void Main(string[] args)
   {
      StopWatch sw1 = new StopWatch();
      StopWatch sw2 = new StopWatch();
      Console.WriteLine(StopWatch.NoOfInstances); //2

      StopWatch sw3 = new StopWatch();
      StopWatch sw4 = new StopWatch();
      Console.WriteLine(StopWatch.NoOfInstances);//4
   }
}
```

Static Constructors

A non-static class can contain a parameterless static constructor. It can be defined with the static keyword and without access modifiers like public, private, and protected.

The following example demonstrates the difference between static constructor and instance constructor.

Example: Static Constructor vs Instance Constructor

```
public class StopWatch
{
    // static constructor
    static StopWatch()
    {
        Console.WriteLine("Static constructor called");
    }

    // instance constructor
    public StopWatch()
    {
        Console.WriteLine("Instance constructor called");
    }

    // static method
    public static void DisplayInfo()
    {
        Console.WriteLine("DisplayInfo called");
    }

    // instance method
    public void Start() { }

    // instance method
    public void Stop() {  }
}
```

Above, the non-static class `StopWatch` contains a static constructor and also a non-static constructor.

The static constructor is called only once whenever the static method is used or creating an instance for the first time. The following example shows that the static constructor gets called when the static method called for the first time. Calling the static method second time onwards won't call a static constructor.

Example: Static Constructor Execution

```
StopWatch.DisplayInfo(); // static constructor called here
StopWatch.DisplayInfo(); // none of the constructors called here
```

Output:

```
Static constructor called.
DisplayInfo called
DisplayInfo called
```

The following example shows that the static constructor gets called when you create an instance for the first time.

Example: Static Constructor Execution

```
StopWatch sw1 = new StopWatch(); // First static constructor and then instance
constructor called
StopWatch sw2 = new StopWatch();// only instance constructor called
StopWatch.DisplayInfo();
```

Output:

```
Static constructor called
instance constructor called
instance constructor called
DisplayInfo called
```

Rules for Static Constructors

1. The static constructor is defined using the `static` keyword and without using access modifiers public, private, or protected.
2. A non-static class can contain one parameterless static constructor. Parameterized static constructors are not allowed.

3. Static constructor will be executed only once in the lifetime. So, you cannot determine when it will get called in an application if a class is being used at multiple places.
4. A static constructor can only access static members. It cannot contain or access instance members.

Read and Write only Properties

C# properties are members of a C# class that provide a flexible mechanism to read, write or compute the values of private fields, in other words, by using properties, we can access private fields and set their values. Properties in C# are always public data members. C# properties use get and set methods, also known as accessors to access and assign values to private fields.

Now the question is what are accessors?

The get and set portions or blocks of a property are called accessors. These are useful to restrict the accessibility of a property, the set accessor specifies that we can assign a value to a private field in a property and without the set accessor property it is like a readonly field. By the get accessor we can access the value of the private field, in other words, it returns a single value. A Get accessor specifies that we can access the value of a field publically.

We have three types of properties: Read/Write, ReadOnly and WriteOnly. Let's see each one by one.

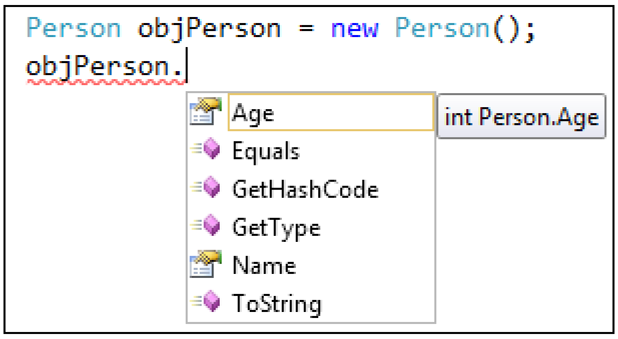
Declare and read/write example of property

We create an example that is used for the Name and Age properties for a Person. So first of all, create a Person class then we will use this class in an executable program.

Person.cs

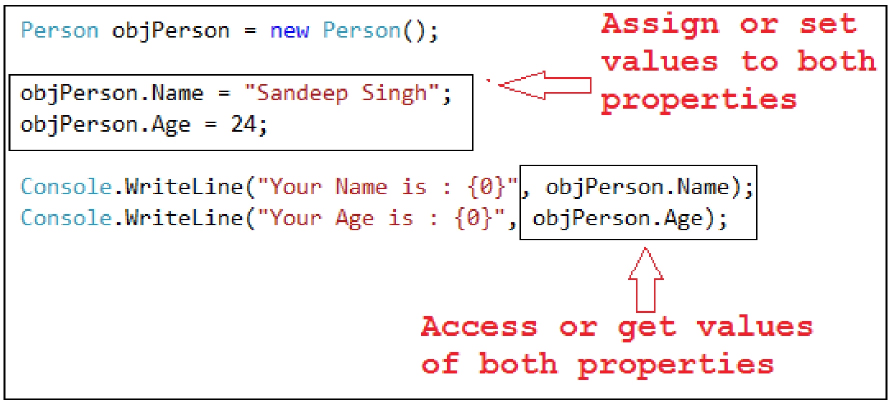
```
namespace PropertyExample
{
public class Person
{
private string mName=string.Empty;
private int mAge=0;

public string Name
{
    get
    {
        return mName;
    }
    set
    {
       mName=value;
    }
}

 public int Age
{
    get
    {
       return mAge;
    }
    set
    {
    mAge=value;
    }
 }} }
```

Now, we use this class in an executable program by creating an object of the Person class. It is Visual Studio IntelliSense that automatically shows object properties when we enter the dot (.) operator after an object. In the following figure we can see Age and Name properties of a Person.



```
Person objPerson = new Person();
objPerson.
```

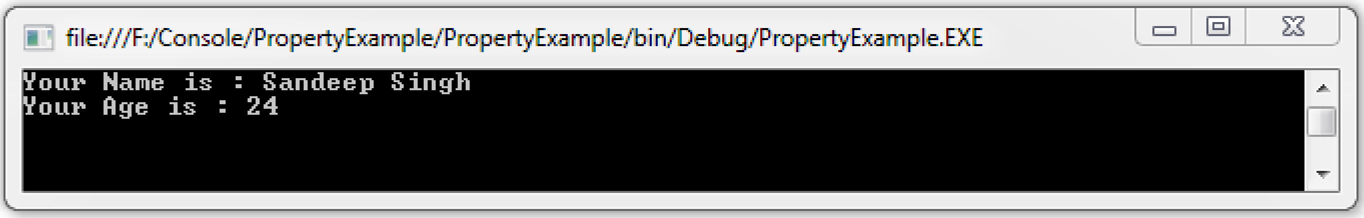
Now we read and write values for the property.



```
Person objPerson = new Person();

objPerson.Name = "Sandeep Singh";
objPerson.Age = 24;

Console.WriteLine("Your Name is : {0}", objPerson.Name);
Console.WriteLine("Your Age is : {0}", objPerson.Age);
```

Assign or set values to both properties
Access or get values of both properties

Finally we get the output of this program:



```
file:///F:/Console/PropertyExample/PropertyExample/bin/Debug/PropertyExample.EXE
Your Name is : Sandeep Singh
Your Age is : 24
```

Let's see the line of code:

1. objPerson.Name="SandeepSingh";

This line of code is called the set accessor of the Name property in the Person class where we are using the private field mName in the set block, so this line of code actually assigns a value in the mName field, in other words we can assign a value to the private field by property.

1. Console.WriteLine("YourNameis:{0}",objPerson.Name);

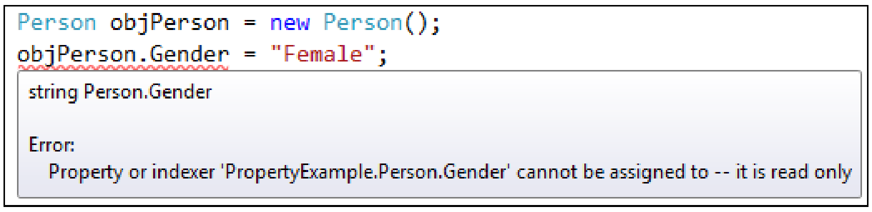
This line of code is called a get accessor of the Name Property in the Person class; in other words, we can access the mName private field by the Name Property because the Name property get accessor returns a value of the private field mName. So the private field is accessible by the property.

Create Readonly Property

We can also create a read only property. Read only means that we can access the valueof aproperty but we can't assign a value to it. When a property does not have a set accessor then it is a read only property. For example in the person class we have a Gender property that has only a get accessor and doesn't have a set accessor. The Person class is:

```
public class Person
{
public string Gender
{
    get
    {
            return"Male";
    }
}
}
```

When we assign a value to the Gender Property of the Person class object then we get an error that it is a readonly property and can't assign a value to it.



```
Person objPerson = new Person();
objPerson.Gender = "Female";
```

```
string Person.Gender

Error:
   Property or indexer 'PropertyExample.Person.Gender' cannot be assigned to -- it is read only
```

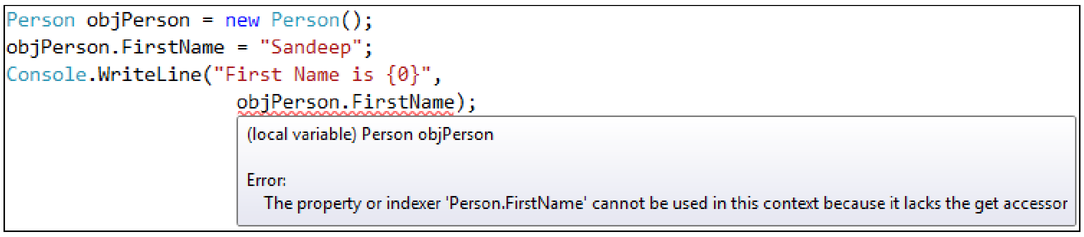
So the Gender property of the Person class always returns a value and we can't assign a value to it.

Create WriteOnly Property

We can also create a write only property. A write only property is a property that we can assign a value to but can't get that value because that property doesn't have a get accessor. For example we have a Person class that has the property FirstName that has a set accessor but doesn't have a get accessor so it is a write only property.

```
Public class Person
{
private string mFirstName=string.Empty;
public string FirstName
{
set{
    mFirstName=value;
    }
}}
```

When we access the value of the FirstName property then we get an error like:



```
Person objPerson = new Person();
objPerson.FirstName = "Sandeep";
Console.WriteLine("First Name is {0}",
                    objPerson.FirstName);
```

(local variable) Person objPerson

Error:
The property or indexer 'Person.FirstName' cannot be used in this context because it lacks the get accessor

We can create a write only property when we don't define a get accessor.

Assign Values to Properties on Object Creation

We can also assign values to properties when we are creating an object of the class. For example when we create a Person class object then we can assign a Name and Age property to the person object.

```
Person  objPerson= new    Person()
{
Name="SandeepSingh",
Age=24
};
```

Properties of objects are a block defined bycurly braces and in the block each property will be separated by a comma.

Validate Property Value

We can validate a value of a property before it's set to a variable; in other words, we can check a value to be assigned to a private field and if it is correct then it will be assigned to the private field, otherwise it will give an error.

Suppose you are a citizen of India and participate as a voter in a parliament election so your age should be greater than or equal to 18 otherwise you can't vote. To implement this function we have a Voter class and that class has an Age property. The Age property can have a value greater than or equal to 18 otherwise it will show 0 for the age with a message. Our Voter class is:

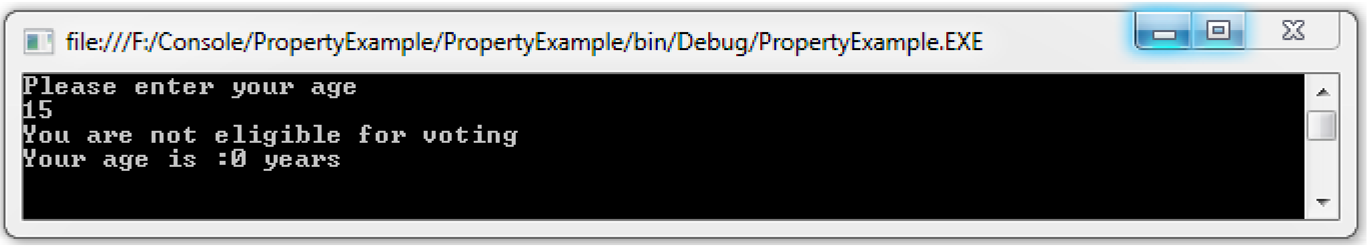
```
using System;
namespace PropertyExample
{
public class Voter
{
    private int mAge=0;
    public int Age
{
      get
     {
                returnmAge;
    }
    set
    {

    if(value>=18)
    {
                mAge=value;
    }
else
{
    Console.WriteLine("Youarenoteligibleforvoting");
}}
}}}
```

Now create an executable program by which we assign an age value of a Voter.

```
Using System;
```

```
namespace PropertyExample
{
class Program
{
static void Main(string[]args)
{
    Voter objVoter=new Voter();
    Console.WriteLine("Pleaseenteryourage");
    objVoter.Age=Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Yourageis:{0}years",objVoter.Age);
    Console.ReadKey();
}
}
}
```

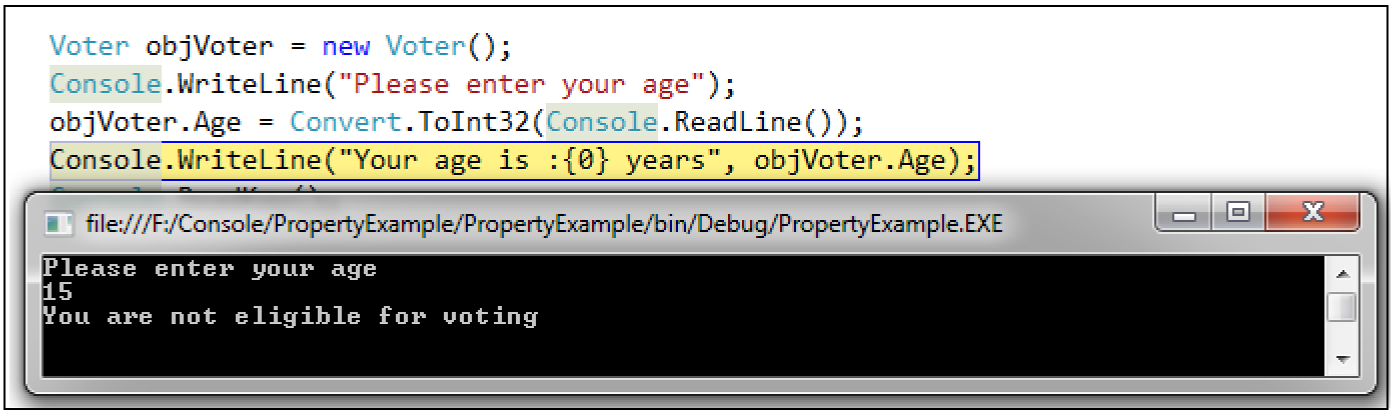
Now we enter a value less than 18 and then we get a message with 0 years.



file:///F:/Console/PropertyExample/PropertyExample/bin/Debug/PropertyExample.EXE

```
Please enter your age
15
You are not eligible for voting
Your age is :0 years
```

In the output screen above we get a message from the set accessor and the value isn't set in the private field so we get a default value of the private field that we already set to 0. Don't be confusedby the message and return value. Here set is not returning a value. It is just printing a message on the console before the get accessor calls the Age property. The following picture shows the message shown before calling the get accessor.

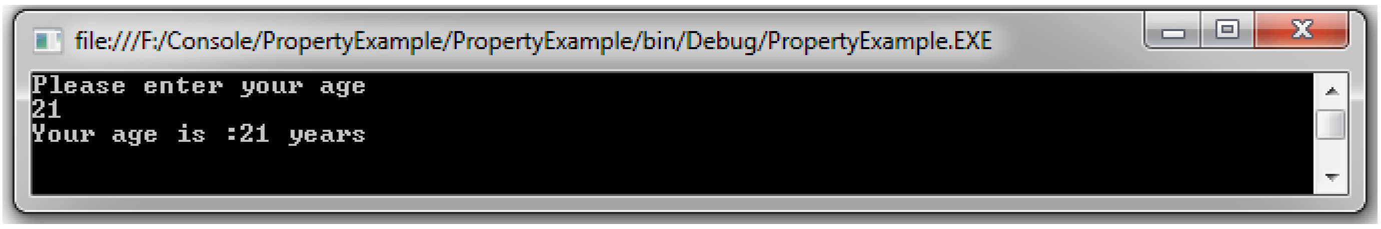


```
Voter objVoter = new Voter();
Console.WriteLine("Please enter your age");
objVoter.Age = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Your age is :{0} years", objVoter.Age);
```

file:///F:/Console/PropertyExample/PropertyExample/bin/Debug/PropertyExample.EXE

```
Please enter your age
15
You are not eligible for voting
```

If weuse a value greater than or equal to 18 then we get the private field value without a message.


file:///F:/Console/PropertyExample/PropertyExample/bin/Debug/PropertyExample.EXE

```
Please enter your age
21
Your age is :21 years
```

C# Encapsulation with Examples

In c#, **Encapsulation** is a process of binding the data members and member functions into a single unit. In c#, the class is the real-time example for encapsulation because it will combine various types ofdata membersandmember functionsinto a single unit.

Generally, in c# the encapsulation is used to prevent alteration of code (data) accidentally from the outsidefunctions. In c#, by defining the class fields with properties, we can protect the data from accidental corruption.

If we define class fields withproperties, then the encapsulated class won’t allow us to access the fields directly. Instead, we need to use getter and setterfunctionsto read or write data based on our requirements.

Following is the example of defining an **encapsulation** class using properties with **get** and **set** accessors.

```
class User
{
 private string location;
 private string name;
 public string Location
 {
  get
  {
   return location;
  }
```

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

If you observe the above code, we defined variableswith private access modifiers and exposed those variables in a public way usingproperties**get** and **set** accessors. If you want to make any modifications to the defined variables, then we can make it by usingpropertieswith **get** and **set** accessors.

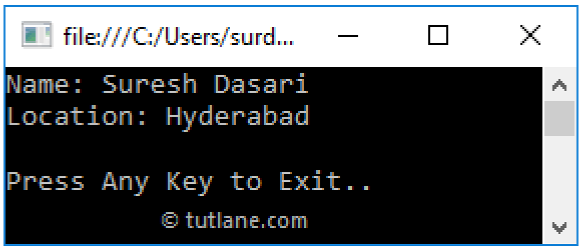
C# Encapsulation Example

Following is the example of defining an encapsulated class in c# programming language.

```
using System;

using System.Text;
namespace Tutlane
{
  class User
  {
   private string location;
   private string name;
   public string Location
   {
     get
     {
      return location;
     }
     set
     {
      location = value;
     }
   }
   public string Name
   {
    get
    {
      return name;
    }
    set
    {
      name = value;
    }
   }
  }
  class Program
  {
   static void Main(string[] args)
   {
     User u = new User();
     // set accessor will invoke
     u.Name = "Suresh Dasari";
     // set accessor will invoke
     u.Location = "Hyderabad";
     // get accessor will invoke
     Console.WriteLine("Name: " + u.Name);
     // get accessor will invoke
     Console.WriteLine("Location: " + u.Location);
     Console.WriteLine("\nPress Enter Key to Exit..");
     Console.ReadLine();
   }
  }
}
```

If you observe the above example, we definedfields in encapsulatedclassusingproperties,and we are able to manipulatefieldvalues using get and set accessors ofproperties.

When you execute the above c# program, you will get the result as shown below.



```
Name: Suresh Dasari
Location: Hyderabad

Press Any Key to Exit..
```

Inheritance Is As keyword Usage,

In c#, **Inheritance** is one of the primary concepts of object-oriented programming (OOP), and it is used to inherit the properties from one class (base) to another (child) class.

The inheritance will enable us to create a new class by inheriting the properties from otherclassesto reuse, extend, and modify other class members' behaviorbased on our requirements.

In c# inheritance, theclasswhose members are inherited is called a **base** (**parent**)class,and theclassthat inherits the members of the**base** (**parent**) class is called a **derived** (**child**)class.

C# Inheritance Syntax

Following is the syntax of implementing an inheritance to define a derived class that inherits the base class's properties in the c# programming language.

```
<access_modifier> class <base_class_name>
{
// Base class Implementation
}

<access_modifier> class <derived_class_name> : <base_class_name>
{
// Derived class implementation
}
```

If you observe the above syntax, we are inheriting the base class's propertiesinto the**child**classto improve code reusability.

Following is the simple example of implementing inheritance in the c# programming language.

```
public class X
{
  public void GetDetails()
  {
   // Method implementation
  }
}
public class Y: X
{
  // your class implementation
}
class Program
{
 static void Main(string[] args)
 {
   Y y = new Y();
   y.GetDetails();
 }
}
```

If you observe the above example, we defined a class "**X**" with the method called "**GetDetails**" and the class "**Y**" is inheriting from class "**X**". After that, we call a "**GetDetails**" method by using an instance of derived class "**Y**".

In c#, it's not possible to inherit the base class constructors in the derived class. The accessibility of other base class membersalso depends on the access modifiers that we used to define those members in a baseclass.

C# Inheritance Example

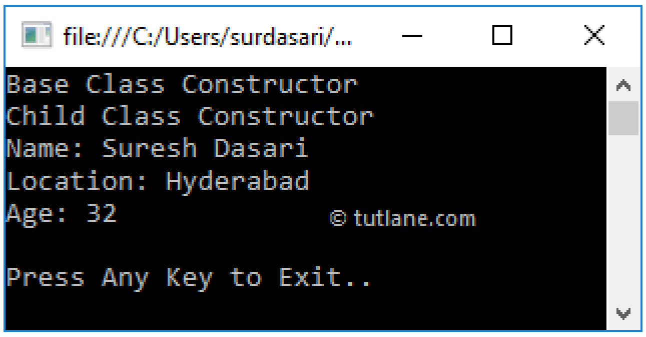
Following is the example of implementing an **inheritance** by defining two classes in the c# programming language.

```
using System;

namespace Tutlane
{
  public class User
  {
   public string Name;
   private string Location;
   public User()
   {
     Console.WriteLine("Base Class Constructor");
   }
   public void GetUserInfo(string loc)
   {
     Location = loc;
     Console.WriteLine("Name: {0}", Name);
     Console.WriteLine("Location: {0}", Location);
   }
  }
  public class Details: User
  {
   public int Age;
   public Details()
   {
     Console.WriteLine("Child Class Constructor");
   }
   public void GetAge()
   {
     Console.WriteLine("Age: {0}", Age);
   }
  }
  class Program
  {
   static void Main(string[] args)
   {
     Details d = new Details();
     d.Name = "Suresh Dasari";
     // Compile Time Error
     //d.Location = "Hyderabad";
     d.Age = 32;
     d.GetUserInfo("Hyderabad");
     d.GetAge();
     Console.WriteLine("\nPress Any Key to Exit..");
     Console.ReadLine();
   }
  }
}
```

If you observe the above example, we defined a base class called “**User**” and inheriting all the user class propertiesinto a derived class called “**Details**” and we are accessing all the members of the**User** class with an instance of the**Details** class.

If we uncomment the commented code, we will get a compile-time error because the **Location** property in the**User** class is defined with a private access modifier. The**private** members can be accessed only within the class.

When you execute the above c# program, you will get the result as shown below.



```
Base Class Constructor
Child Class Constructor
Name: Suresh Dasari
Location: Hyderabad
Age: 32

Press Any Key to Exit..
```

If you observe the above result, we are able to access all thepropertiesof base class into child class based on our requirements.

In c#, Structures won't support inheritance, but they can implement **interfaces**.

C# Multi-Level Inheritance

Generally, c# supports only **single inheritance** that means aclasscan only inherit from one baseclass. However, in c# the inheritance is transitive, and it allows you to define a hierarchical inheritance for a set of types, and it is called a multi-level inheritance.

For example, if class **C** is derived from class **B**, and class **B** is derived from class **A**, then class **C** inherits the members declared in both class **B** and class **A**.

```
public class A
{
// Implementation
}
public class B: A
{
// Implementation
}
public class C: B
{
// Implementation
}
```

If you observe the above code snippet, class **C** is derived from class **B**, and class **B** is derived from class **A**, then class **C** inherits the members declared in both class **B** and class **A**. This is how we can implement **multi-level** inheritance in our applications.

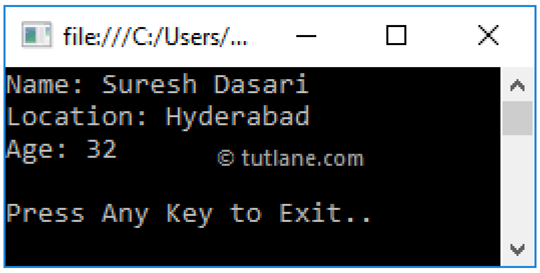
C# Multi-Level Inheritance Example

```
using System;
namespace Tutlane
{
  public class A
  {
   public string Name;
   public void GetName()
   {
     Console.WriteLine("Name: {0}", Name);
   }
  }
  public class B: A
  {
   public string Location;
   public void GetLocation()
   {
     Console.WriteLine("Location: {0}", Location);
   }
  }
  public class C: B
  {
   public int Age;
   public void GetAge()
   {
     Console.WriteLine("Age: {0}", Age);
   }
  }
  class Program
  {
   static void Main(string[] args)
   {
     C c = new C();
     c.Name = "Suresh Dasari";
     c.Location = "Hyderabad";
     c.Age = 32;
     c.GetName();
     c.GetLocation();
     c.GetAge();
     Console.WriteLine("\nPress Any Key to Exit..");
     Console.ReadLine();
   }
  }
}
```

If you observe the above example, we implemented three classes (**A**, **B**, **C**), and class **C** is derived from class **B**, and class **B** is derived from class **A**.

By implementing a multi-level inheritance, class **C** can inherit the members declared in class **B** and class **A**.

When you execute the above c# program, you will get the result as shown below.



```
Name: Suresh Dasari
Location: Hyderabad
Age: 32

Press Any Key to Exit..
```

C# Multiple Inheritance

As discussed, c# supports only **single inheritance** that means a class can only inherit from one baseclass. If we try to inherit a class from multiple base classes, then we will get compile-time errors.

```
public class A
{
// Implementation
}
public class B
{
// Implementation
}
public class C: A, B
{
// Implementation
}
```

If you observe the above code snippet, class **C** is trying to inheritpropertiesfrom both classes **A** and **B** simultaneously, which will lead to compile-time errors like "**Class C cannot have multiple classes: A and B**".

As discussed, **multi-level** inheritance is supported in c#, but **multiple** inheritance is not supported. If you want to implement **multiple** inheritance in c#, we can achieve this by using interfaces. In the next chapters, we will learn how to useinterfacesto achieve multiple inheritance in a detailed manner.

n c#, **sealed** is a keyword used to stop inheriting the particular class from other classes. We can also prevent overriding the particular properties or methods based on our requirements.

Generally, when we create a particular class we can inherit all the properties and methods in any class. If you want to restrict access to a defined class and its members, then by using a **sealed** keyword, we can prevent other classes from inheriting the defined class.

C# Sealed Class

In c#, a sealed class can define by using a **sealed** keyword. As discussed, when we define a class with the **sealed** keyword, then we don't have a chance to inherit that particular class.

The following are the various ways of defining **sealed classes** in the c# programming language.

```
//TYPE 1

sealed class Test
{
        public string Name;
        public string Location;
}

//TYPE 2
public sealed class Test1
        {
                public int Age;
        }


//TYPE 3
sealed public class Test2
{
// Implementation
}
```

If you observe the above code snippet, we defined various classes with a **sealed** keyword to ensure that the defined classes are not inheritable to any class. In c#, we can use a **sealed** keyword before or after the access modifier to define sealed classes.

C# Sealed Class Example

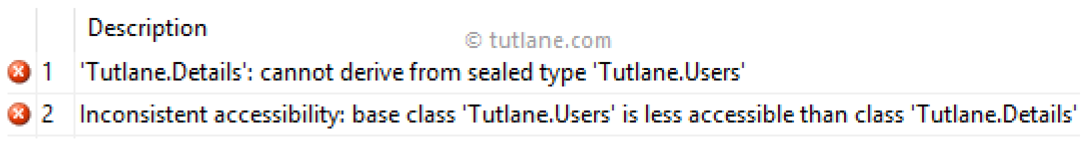
Following is the example of using a **sealed** keyword to define sealed classes in the c# programming language.

```
 using System;
namespace Tutlane
```

```
{
   // Base Class
   sealed class Users
   {
      public string name = "Suresh Dasari";
      public string location = "Hyderabad";
      public void GetInfo()
      {
         Console.WriteLine("Name: {0}", name);
         Console.WriteLine("Location: {0}", location);
      }
   }
   // Derived Class
   public class Details : Users
   {
      public int age = 32;
      public void GetAge()
      {
         Console.WriteLine("Age: {0}", age);
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         Details d = new Details();
         d.GetAge();
         d.GetInfo();
         Console.WriteLine("\nPress Enter Key to Exit..");
         Console.ReadLine();
      }
   }
}
```

If you observe the above example, we defined a class “**Users**” with a **sealed** keyword, and we are trying to inherit the properties from the **Users** class using the **Details** class.

When you execute the above c# program, we will get the result as shown below.



```
     Description
1    'Tutlane.Details': cannot derive from sealed type 'Tutlane.Users'
2    Inconsistent accessibility: base class 'Tutlane.Users' is less accessible than class 'Tutlane.Details'
```

If you observe the above result, we are getting compile-time errors because we tried to inherit sealed class properties in another class.

C# Sealed Method or Property

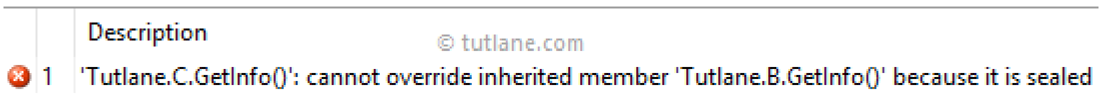
In c#, we can also use the **sealed** keyword on a method or property that overrides a virtual method or property in a base class to allow other classes to derive from the base class and prevent them from overriding specific virtual methods or properties.

Following is the example of using a **sealed** keyword on a method that overrides a virtual method in a base class.

```
using System;
namespace Tutlane
{
    public class A
    {
        public virtual void GetInfo()
        {
            Console.WriteLine("Base Class A Method");
        }
        public virtual void Test()
        {
            Console.WriteLine("Base Class A Test Method");
        }
    }
    public class B: A
    {
        public sealed override void GetInfo()
        {
             Console.WriteLine("Derived Class B Method");
        }
        public override void Test()
        {
             Console.WriteLine("Derived Class B Test Method");
        }
    }
    public class C: B
    {
        // Compile time error
        public override void GetInfo()
        {
            Console.WriteLine("Age: {0}", base.age);
        }
        public override void Test()
        {
            Console.WriteLine("Derived Class C Test Method");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            C c = new C();
            c.GetInfo();
            c.Test();
            Console.WriteLine("\nPress Enter Key to Exit..");
            Console.ReadLine();
        }
    }
}
```

If you observe the above example, we used a **sealed** keyword on the method (**GetInfo**) that overrides a base class virtual method. In the above example, class **C** is inherited from class **B,** but class **C** cannot override a virtual method **GetInfo** that is declared in class **A** because that method is sealed in class **C**.

When you execute the above c# program, you will get a result, as shown below.



```
   Description
1  'Tutlane.C.GetInfo()': cannot override inherited member 'Tutlane.B.GetInfo()' because it is sealed
```

This is how you can use the **sealed** keyword on a method or property that overrides a base class virtual method or property.

Instead of using **sealed** keyword, we can prevent derived classes to override base class methods or properties by not declaring them as virtual.

C# Sealed Keyword Features

The following are the important points which we need to remember about the **sealed** keyword in the c# programming language.

- In c#, to apply a **sealed** keyword on a method or property, it must always use with override.
- In c#, we should not use an **abstract** modifier with a **sealed** class because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.
- In c#, the local variables cannot be sealed.
- In c#, structs are implicitly sealed; they cannot be inherited.

Delegation

n c#, the **delegate** is a type that defines a method signature, and it is useful to hold the reference of one or more methods which are having the same signatures.By using delegates, you can invoke the methods and send methods as an argument to other methods.In c#, the delegate is a reference type, and it's type-safe and secure. The delegates are similar to function pointers in c++.

C# Delegate Declaration Syntax

In c#, the declaration of delegate will be same as method signature, but the only difference is we will use a `delegate` keyword to define delegates.Following is the syntax of defining a delegate using `delegate` keyword in c# programming language.

```
<access_modifier> delegate <return_type> <delegate_name>()
```

Following is the example of declaring a delegate using `delegate` keyword in c#.

```
public delegate void UserDetails(string name);
```

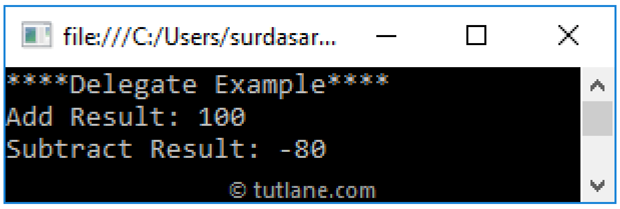
If you observe the above example, a delegate's declaration is same as a method declaration with required parameter types and return value.The above defined method "**UserDetails**" can be pointed to any other method with the same parameters and return type.In c#, the delegates must be instantiated with a method or expression with the same return type and parameters. We can invoke a method through the delegate instance.

C# Delegate Example

```
using System;
namespace Tutlane
{
   // Declare Delegate
   public delegate void SampleDelegate(int a, int b);
   class MathOperations
   {
      public void Add(int a, int b)
      {
        Console.WriteLine("Add Result: {0}", a + b);
      }
      public void Subtract(int x, int y)
      {
        Console.WriteLine("Subtract Result: {0}", x - y);
      }
   }
   class Program
   {
     static void Main(string[] args)
     {
       Console.WriteLine("****Delegate Example****");
       MathOperations m = new MathOperations();
       // Instantiate delegate with add method
       SampleDelegate dlgt = m.Add;
       dlgt(10, 90);
       // Instantiate delegate with subtract method
       dlgt = m.Subtract;
       dlgt(10, 90);
       Console.ReadLine();
     }
   }
}
```

If you observe the above example, we created a delegate called “**SampleDelegate**” and the delegate has been instantiated by using defined methods.

When you execute the above c# program, you will get the result as shown below.



```
****Delegate Example****
Add Result: 100
Subtract Result: -80
```

This is how we can use delegates in our applications to call all the methods which are having the same signatures with a single object.In c#, you can invoke the delegates same as method or by using the **Invoke** method like as shown below.

```
SampleDelegate dlgt = m.Add;
dlgt(10, 90);
// or
dlgt.Invoke(10, 90);
```

Types of Delegates in C#

In c#, we have two different types of delegates available. Those are

- Single cast Delegates
- Multicast Delegates

Single Cast Delegates in C#

In c#, a delegate that points to a single method is called a single cast delegate, and it is used to hold the reference of a single method as explained in the above example.

Multicast Delegates in C#

In c#, a delegate that points to multiple methods is called a multicast delegate, and it is used to hold the reference of multiple methods with a single delegate.

By using `"+"` operator, we can add the multiple method references to the delegate object. Same way, by using `"-"` operator we can remove the method references from the delegate object.

In c#, Multicast delegates will work with only the methods that are having `void` as return type. If we want to create a multicast delegate with the return type, then we will get a return type of the last method in the invoking list.

C# Multicast Delegate Example

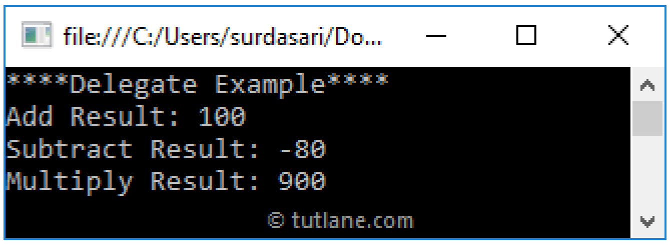
Following is the example of implementing a multicast delegate to hold the reference of multiple methods with `"+"` operator in c# programming language.

```
using System;
namespace Tutlane
{
   // Declare Delegate
   public delegate void SampleDelegate(int a, int b);
   class MathOperations
   {
      public void Add(int a, int b)
      {
         Console.WriteLine("Add Result: {0}", a + b);
      }
      public void Subtract(int x, int y)
```

```
        {
          Console.WriteLine("Subtract Result: {0}", x - y);
        }
        public void Multiply(int x, int y)
        {
          Console.WriteLine("Multiply Result: {0}", x * y);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("****Delegate Example****");
            MathOperations m = new MathOperations();
            // Instantiate delegate with add method
            SampleDelegate dlgt = m.Add;
            dlgt += m.Subtract;
            dlgt += m.Multiply;
            dlgt(10, 90);
            Console.ReadLine();
        }
    }
}
```

If you observe the above example, we created a delegate called “**SampleDelegate**” and holding the reference of multiple methods using "+" operator. Our delegate becomes a multicast delegate, and invoking a dlgt instance will invoke all the methods sequentially.

When you execute the above c# program, you will get the result as shown below.



```
****Delegate Example****
Add Result: 100
Subtract Result: -80
Multiply Result: 900
```

This is how we can use multicast delegates in c# to hold the reference of multiple methods based on our requirements.

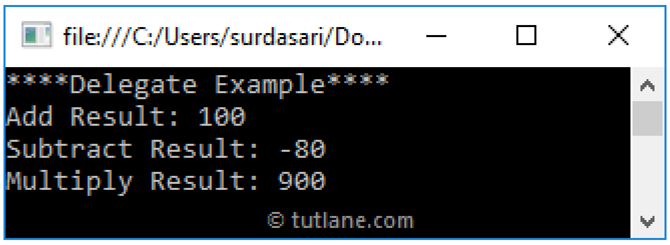
Pass Method as Parameter using Delegate

By using delegates in c#, we can pass methods as the parameter. Following is the example of sending methods as a parameter using a delegate.

```
using System;
namespace Tutlane
{
    // Declare Delegate
    public delegate void SampleDelegate(int a, int b);
    class MathOperations
    {
      public void Add(int a, int b)
      {
        Console.WriteLine("Add Result: {0}", a + b);
      }
      public void Subtract(int x, int y)
      {
        Console.WriteLine("Subtract Result: {0}", x - y);
      }
      public void Multiply(int x, int y)
      {
        Console.WriteLine("Multiply Result: {0}", x * y);
      }
    }
    class Program
    {
      static void Main(string[] args)
      {
         Console.WriteLine("****Delegate Example****");
         MathOperations m = new MathOperations();
         SampleMethod(m.Add, 10, 90);
         SampleMethod(m.Subtract, 10, 90);
         SampleMethod(m.Multiply, 10, 90);
         Console.ReadLine();
      }
      static void SampleMethod(SampleDelegate dlgt, int a, int b)
      {
         dlgt(a, b);
      }
    }
}
```

If you observe the above example, we created a “**SampleMethod**” with a **delegate** as a parameter type. By declaring like this, you can pass the method as a parameter to the newly created method (**SampleMethod**).

When you execute the above c# program, you will get the result as shown below.



```
****Delegate Example****
Add Result: 100
Subtract Result: -80
Multiply Result: 900
```

C# Delegates Overview

The following are the important properties of delegate in c# programming language.

- We need to use a `delegate` keyword to define delegates.
- In c#, delegates are used to hold the reference of one or more methods that have the same signature as delegates.
- In c#, delegates are like function pointers in C++, but these are type-safe and secure.
- By using delegates, you can pass methods as a parameter to the other methods.
- In c#, we can invoke delegates as normal methods or by using **Invoke** property.
- By using `"+"` operator, we can add multiple methods to delegates.
- By using delegates, we can call multiple methods with a single event.

Polymorphism

The word **polymorphism** means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.

Polymorphism can be static or dynamic. In **static polymorphism**, the response to a function is determined at the compile time. In **dynamic polymorphism**, it is decided at run-time.

Static Polymorphism

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are −

- Function overloading
- Operator overloading

We discuss operator overloading in next chapter.

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

The following example shows using function **print()** to print different data types −

```
using System;

namespace PolymorphismApplication {
   class Printdata {
      void print(int i) {
         Console.WriteLine("Printing int: {0}", i );
      }
      void print(double f) {
         Console.WriteLine("Printing float: {0}" , f);
      }
      void print(string s) {
         Console.WriteLine("Printing string: {0}", s);
      }
      static void Main(string[] args) {
         Printdata p = new Printdata();

         // Call print to print integer
         p.print(5);

         // Call print to print float
         p.print(500.263);

         // Call print to print string
         p.print("Hello C++");
         Console.ReadKey();
      }
   }
}
```

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

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C++
```

Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it

. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

```
 using System;

namespace PolymorphismApplication {
   abstract class Shape {
      public abstract int area();
   }

   class Rectangle:  Shape {
      private int length;
      private int width;

      public Rectangle( int a = 0, int b = 0) {
         length = a;
         width = b;
      }
      public override int area () {
         Console.WriteLine("Rectangle class area :");
         return (width * length);
      }
   }
   class RectangleTester {
      static void Main(string[] args) {
         Rectangle r = new Rectangle(10, 7);
         double a = r.area();
         Console.WriteLine("Area: {0}",a);
         Console.ReadKey();
      }
   }
}
```

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

```
Rectangle class area :
Area: 70
```

When you have a function defined in a class that you want to be implemented in an inherited class(es), you use **virtual** functions. The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

The following program demonstrates this −

```
using System;

namespace PolymorphismApplication {
   class Shape {
      protected int width, height;

      public Shape( int a = 0, int b = 0) {
         width = a;
         height = b;
      }
      public virtual int area() {
         Console.WriteLine("Parent class area :");
         return 0;
      }
   }
   class Rectangle: Shape {
      public Rectangle( int a = 0, int b = 0): base(a, b) {

      }
      public override int area () {
         Console.WriteLine("Rectangle class area :");
         return (width * height);
      }
   }
   class Triangle: Shape {
      public Triangle(int a = 0, int b = 0): base(a, b) {
      }
      public override int area() {
         Console.WriteLine("Triangle class area :");
         return (width * height / 2);
      }
   }
   class Caller {
      public void CallArea(Shape sh) {
         int a;
         a = sh.area();
         Console.WriteLine("Area: {0}", a);
      }
   }
   class Tester {
      static void Main(string[] args) {
         Caller c = new Caller();
         Rectangle r = new Rectangle(10, 7);
         Triangle t = new Triangle(10, 5);

         c.CallArea(r);
         c.CallArea(t);
         Console.ReadKey();
      }
   }
}
```

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

```
Rectangle class area:
Area: 70
Triangle class area:
Area: 25
```

The Virtual and Override Keywords

In c#, the **virtual** keyword is useful to **override** base class members such as properties, methods, etc., in the derived class to modify it based on our requirements.

Following is the simple example of defining a method with the **virtual** keyword in the c# programming language.

```
public class Users
{
  public virtual void GetInfo()
  {
    Console.WriteLine("Base Class");
  }
}
```

If you observe the above code snippet, we defined a **GetInfo** method with the **virtual** keyword. The **GetInfo** method can be overridden by any derived class that inherits properties from a base class called **Users**.

Generally, whenever the virtual method is invoked, the run-time object will check for an overriding member in the derived class. If no derived class has overridden the member, then the virtual method will be treated as an original member.

In c#, by default all the methods are **non-virtual,** and we cannot override **non-virtual** methods. If you want to override a method, you need to define it with the **virtual** keyword.

The **virtual** keyword in c# cannot be used with static, abstract, private, or override modifiers. In c#, the virtual inherited properties can be overridden in a derived class by including a property declaration that uses the override modifier.

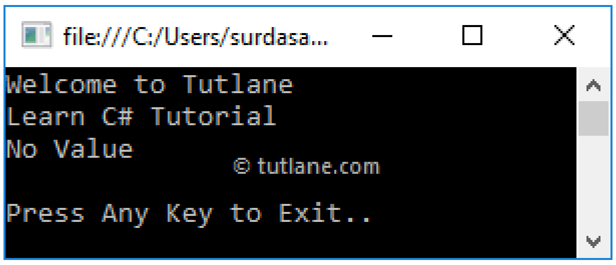
C# Virtual Keyword Example

Following is the example of using a virtual keyword to allow class members to be overridden in a derived class in the c# programming language.

```
using System;

namespace Tutlane
{
   // Base Class
   public class BClass
   {
      public virtual string Name { get; set; }
      public virtual void GetInfo()
      {
         Console.WriteLine("Learn C# Tutorial");
      }
   }
   // Derived Class
   public class DClass : BClass
   {
      private string name;
      public override string Name
      {
         get
         {
            return name;
         }
         set
         {
            if (!string.IsNullOrEmpty(value))
            {
               name = value;
            }
            else
            {
               name = "No Value";
            }
         }
      }
      public override void GetInfo()
      {
         Console.WriteLine("Welcome to Tutlane");
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         DClass d = new DClass();
         d.GetInfo();
         BClass b = new BClass();
         b.GetInfo();
         d.Name = string.Empty;
         Console.WriteLine(d.Name);
         Console.WriteLine("\nPress Enter Key to Exit..");
         Console.ReadLine();
      }
   }
}
```

If you observe the above example, we are overriding a base class (**BClass**) methods and properties which we defined with a **virtual** keyword in a derived class (**DClass**) using override keyword.

When you execute the above c# program, you will get the result as shown below.



```
Welcome to Tutlane
Learn C# Tutorial
No Value

Press Any Key to Exit..
```

This is how we can use **virtual** keyword in c# to allow class members, such as methods, properties, etc., to be overridden in a derived class based on our requirements.

Abstract Classes, Abstract Methods

n c#, **abstract** is a keyword, and it is useful to define classes and class members that are needed to be implemented or overridden in a derived class.

In c#, you can use abstract modifiers with classes, methods, properties, events, and indexers based on our requirements. The members we defined as abstract or included in an abstract class must be implemented by classes derived from an abstract class.

Now we will see how to use abstract modifier in classes and methods with examples.

C# Abstract Class

In c#, **abstract class** is a class that is declared with a `abstract` modifier. If we define a class with `abstract` modifier, then that class is intended only to be used as a base class for other classes.

The **abstract class** cannot instantiate, and it can contain both abstract and non-abstract members. The class that is derived from the abstract class must implement all the inherited abstract methods and accessors.

In c#, you can define an abstract class by using `abstract` keyword. Following is the example of defining an abstract class using `abstract` keyword.

```
abstract class Info
{
abstract public void GetDetails();
}
```

If you observe the above code snippet, we defined an abstract class (**Info**) using `abstract` keyword with **GetDetails** method signature.

If we define a method with `abstract` modifier, then that method implementation must be done in a derived class.

Following is the example of implementing a class by deriving from the abstract class.

```
class User: Info
{
   public override void GetDetails()
   {
     // Method Implementation
   }
}
```

If you observe the code snippet, we inherited an abstract class (**Info**) in the **User** class and implemented a defined abstract method in the **User** class using the override keyword. In c#, abstract methods are internally treated as virtual methods, so those methods need to be overridden by the derived class.

In c#, we should not use a sealed keyword with an abstract class because the sealed keyword will make a class not inheritable but `abstract` modifier requires a class to be inherited.

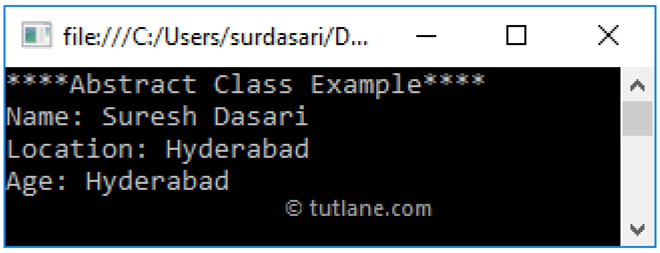
C# Abstract Class Example

Following is the example of defining an abstract class using `abstract` modifier in c# programming language.

```
using System;

namespace Tutlane
{
   abstract class Info
   {
      abstract public void GetDetails(string x, string y, int z);
   }
   class User: Info
   {
      public override void GetDetails(string a, string b, int c)
      {
         Console.WriteLine("Name: {0}", a);
         Console.WriteLine("Location: {0}", b);
         Console.WriteLine("Age: {0}", b);
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         User u = new User();
         Console.WriteLine("****Abstract Class Example****");
         u.GetDetails("Suresh Dasari", "Hyderabad", 32);
         Console.ReadLine();
      }
   }
}
```

If you observe the above example, we defined an abstract class called “**Info**” with required methods, and the derived class “**User**” has implemented all the inherited abstract methods and accessors.

When you execute the above c# program, you will get the result as shown below.



```
****Abstract Class Example****
Name: Suresh Dasari
Location: Hyderabad
Age: Hyderabad
```

This is how we can use abstract classes in our applications based on our requirements.

C# Abstract Class Features

The following are important features of abstract class in c# programming language.

- In c#, abstract classes cannot be instantiated.
- The abstract classes can contain both abstract and non-abstract methods and accessors.
- In c#, we should not use a sealed keyword with abstract class because the sealed keyword will make a class not inheritable, but `abstract` modifier requires a class to be inherited.
- A class that is derived from an abstract class must include all the implementations of inherited abstract methods and accessors.

C# Abstract Method

In c#, the **abstract method** is a method that is declared with a `abstract` modifier. If we define a method with `abstract` modifier, then that method doesn't contain any implementation, and method declaration ends with a semicolon.

Following is the example of defining an abstract method in the c# programming language.

```
public abstract void GetDetails();
```

The abstract methods in c# are permitted to declare only in abstract classes, and the class that is derived from an abstract class must provide an implementation for defined abstract methods.

In c#, abstract methods are internally treated as virtual methods. Hence, those methods need to be overridden in the derived class, and we should not static or virtual modifiers during abstract method declaration.

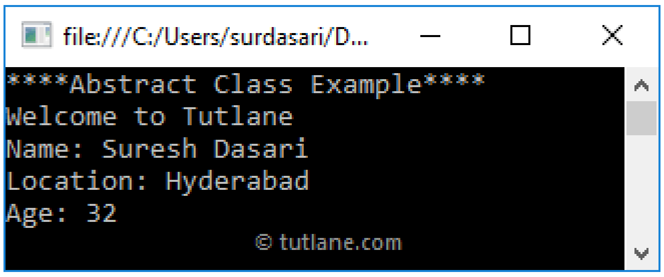
C# Abstract Method Example

Following is the example of declaring an abstraction method in an abstract class in the c# programming language.

```
using System;

namespace Tutlane
{
    abstract class Info
    {
        public void Welcome()
        {
            Console.WriteLine("Welcome to Tutlane");
        }
        public int age = 32;
        public abstract void GetDetails(string x, string y);
    }
    class User: Info
    {
        public override void GetDetails(string a, string b)
        {
            Welcome();
            Console.WriteLine("Name: {0}", a);
            Console.WriteLine("Location: {0}", b);
            Console.WriteLine("Age: {0}", age);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            User u = new User();
            Console.WriteLine("****Abstract Class Example****");
            u.GetDetails("Suresh Dasari", "Hyderabad");
            Console.ReadLine();
        }
    }
}
```

If you observe the above example, we defined an abstract class called “**Info**” with required abstract and non-abstract methods. The derived class “**User**” has implemented all the inherited abstract methods and accessors.

When you execute the above c# program, we will get the result as shown below.



```
****Abstract Class Example****
Welcome to Tutlane
Name: Suresh Dasari
Location: Hyderabad
Age: 32
```

This is how we can use abstract methods in c# abstract classes based on our requirements.

C# Abstract Method Features

The following are the important features of the abstract method in the c# programming language.

- In c#, abstract methods are permitted to declare only within abstract classes.
- The abstract method declaration will not contain any implementation; only the derived classes will provide an actual implementation for abstract methods.
- In c#, abstract methods are internally treated as virtual methods, so they need to be overridden in the derived class.
- We should not use static or virtual modifiers during the abstract method declaration.

In c#, abstract properties will act the same as abstract methods, but the only difference is declaration and invocation syntax.