# Basic Structure of C# Program

## 

1. **What is C#.NET?**
2. **Advantages of using the .NET Framework from the C# point of view.**
3. **Different Types of applications developed using C#.NET.**
4. **What is the visual studio?**
5. **What is a console application?**
6. **How to Create a console application using the visual studio?**
7. **Example of console application using C#.**
8. **Understanding the basic structure of a C# Program.**  
   **Importing section**  
   **Namespace Declaration**  
   **Class Declaration**  
   **Main() method**

So, here, first, we will understand what is C#.Net and Visual Studio and what type of applications we can develop using C#.Net. Then we will discuss the basic structure of a C# program using a console application.

##### ****What is C#.NET?****

1. C#.NET is one of the Microsoft programming languages.
2. It is the most powerful programming language among all programming languages available in the .NET framework because it contains all the features of C++, VB.NET, JAVA, and also some additional features.
3. C#.NET is a completely object-oriented programming language.
4. This programming language is intended to be a simple, modern, general-purpose object-oriented programming language.

##### ****Advantages of using the .NET framework from the C# point of view.****

1. It provides GUI features. Earlier programming languages like C, C++ do not support GUI features but C#.NET will provide complete GUI features. All GUI features are getting from the framework.
2. Any database will support and perform the operations. Using ADO.NET technology we can perform the operations with any DB server. ADO.NET is also a part of the framework.
3. It prepares WEB applications. Using ASP.NET technology we can implement web applications. ASP.NET is needed language support so we are using either VB or C# as language supporters. ASP.NET is also a part of the framework.

##### ****Different Types of applications developed using C#.NET.****

1. Windows applications
2. Web applications
3. Console applications
4. Class library:

##### ****What is the visual studio?****

Visual Studio is one of the Microsoft IDE tools. Using this tool we can implement applications with the .NET framework. This tool provides some features such as

1. Editor
2. Compiler
3. Interpreter

##### ****What is a console application?****

1. These applications contain a similar user interface to the OS like MS-DOS, UNIX, etc.
2. The console application is known as the CUI application because in this application we completely work with the CUI environment.
3. These applications are similar to c or c++ applications.
4. Console applications do not provide any GUI facilities like the mouse pointer, colors, buttons, menu bars, etc.

###### **Let’s First Understand the Basic Structure of C# Program using a console application.**

**<**span style="font-family: arial, helvetica, sans-serif;">//List of classlibraries /namespaces

**namespace** *namespacename*

**{**

**class** classname

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//statements

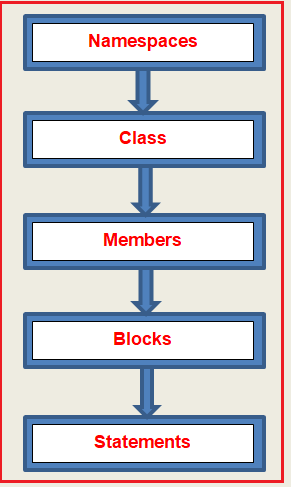
**}**

**}**

**}**

**<**/span**>**

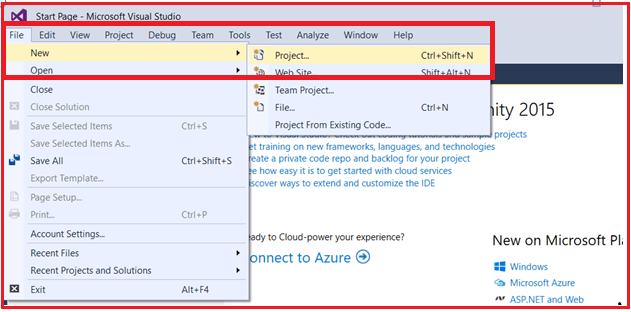
**The above process is shown in the below diagram.**



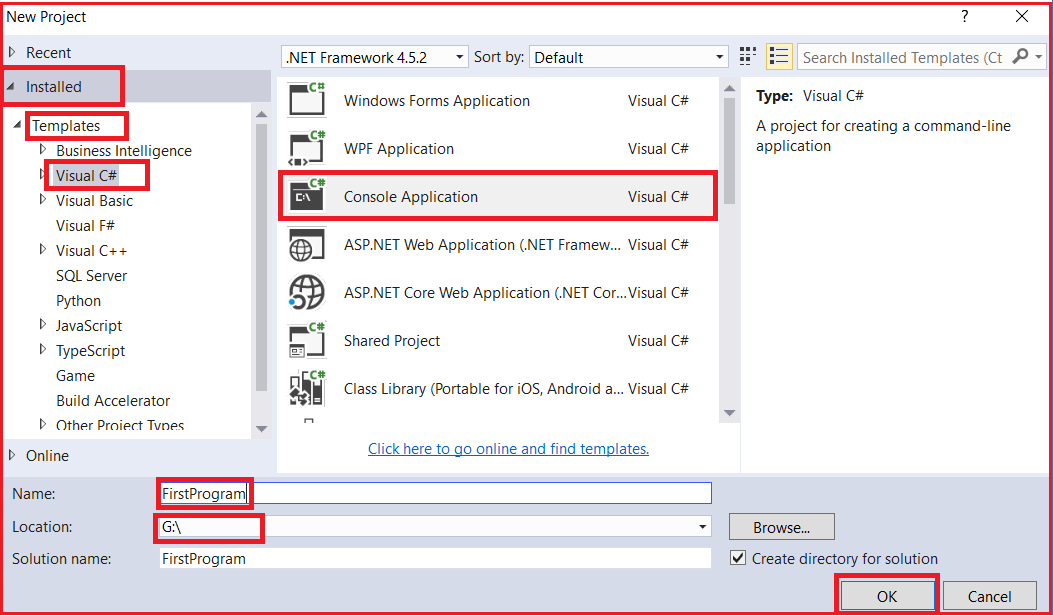
**NOTE:** C#.NET is a case-sensitive language and every statement in C# should end with a semicolon.

##### ****How to Create a console application using the visual studio?****

Open visual studio 2015 (You can use any lower or higher version). Go to the new project (**File => New => Project**) as shown in the below image.



In the next window, select **Installed => Templates => Visual C#** from the left pane. From the middle Panel, select **Console Application**. Then specify the application name and location and finally click on the **OK** button as shown in the below image.



Once you click on the OK button, it will take some time to create the project solution for us.

##### ****Let’s create one program to print a welcome message on the console.****

Whenever you create a console application, by default the .NET Framework creates a class (i.e. Program class) for us. Let’s modify the Program.cs file as shown below to print a welcome message on the console screen.

**<**span style="font-family: arial, helvetica, sans-serif;"**>using** System;

**namespace** *FirstProgram*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Console.WriteLine**(**"Welcome to C#.NET"**)**;

Console.ReadKey**()**;

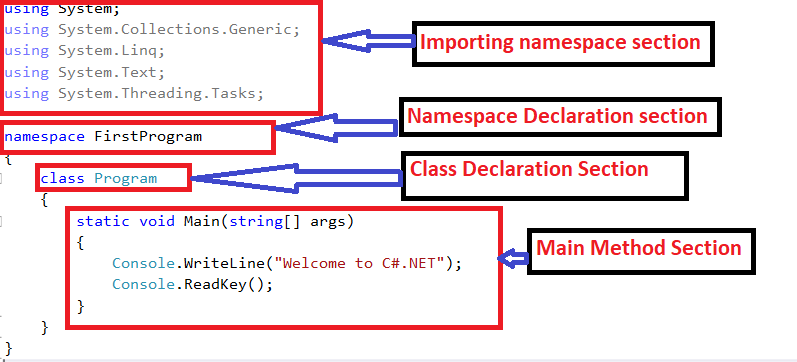
**}**

**}**

**}**

**<**/span**>**

Through visual studio whenever we are creating one console application, automatically we are getting four sections as shown in the image below.



**Let’s understand each of these sections in details.**

##### ****Importing section:****

This section contains importing statements that are used to import the BCL (Base Class Libraries). This is similar to the include statements in the C programming language.

**Syntax: using namespace;**  
**Example: using System;**

**Note:** If the required namespace is a member of another namespace we have to specify the parent and child namespaces separated by a dot.

**using System.Data;**  
**using System.IO;**

##### ****Namespace Declaration:****

Here a user-defined namespace is to be declared. In .NET applications, all classes related to the project should be declared inside one namespace.

**Syntax:**  
**namespace NamespaceName**  
**{**  
**}**

Generally, the namespace name will be the same as the project name.

##### ****Class Declaration:****

This is to declare the start-up class of the project. In every .NET applications like console and windows applications, there should be a start-up class. In this application, the start-up class name is “program”. A startup class is nothing but a class which contains a “Main()” method.

###### **Syntax:**

**class ClassName**  
**{**  
**}**

##### ****Main() method:****

The main() method is the starting execution point of the application. When the application is executed the main method will be executed first. This method contains the main logic of the application.

##### ****What is using?****

Using is a keyword. Using this keyword we can refer to .NET BCL in C# applications.

**Note:** In .NET the base class libraries are divided into a collection of namespaces. Each namespace contains a set of predefined classes and sub-namespaces. The namespace contains another namespace is called as sub-namespaces.

# Methods and Properties of Console Class in C#

## ****Methods and Properties of Console Class in C#****

In this article, I am going to discuss the **Methods and Properties of Console class in C#** with some examples. Please read our previous article where we discussed the [**basic structure of a C# program**](https://dotnettutorials.net/lesson/const-and-read-only-csharp/). As part of this article, I am going to discuss the following pointers related to the Console class in detail.

1. **What is Console Class in C#?**
2. **Properties of Console Class in C#.**
3. **Methods of Console class in C#.**
4. **Understanding the use of Write and WriteLine method in C#.**
5. **Program to show how to print the value of a variable in a console application.**
6. **Understanding the use of the ReadLine method in C#.**
7. **Program to show the use of BackgroundColor, ForegroundColor, and Title properties of Console class.**

#### ****What is Console Class in C#?****

In order to implement the user interface in console applications, Microsoft provided us with a class called **Console**. The **Console** class is available in the “**System**” namespace. This **Console** class provides some methods and properties using which we can implement the user interface in a console application.

All the properties and methods available in the console class are static. So we can access these members by using the Console class name i.e. we don’t require Console class instance.

##### ****Properties of Console Class in C#:****

|  |  |
| --- | --- |
| **Property** | **Description** |
| **Title** | Specifies the title of the console application |
| **Background color** | Specifies the background color of the text |
| **Foreground color** | Specifies the foreground color of the text |
| **Cursor size** | Specifies the height of the cursor in the console window “1 to 100” |

##### ****Methods of Console class in C#:****

|  |  |
| --- | --- |
| **Method** | **Description** |
| **Clear()** | To clear the screen |
| **Beep()** | Play a beep sound using a PC speaker at runtime |
| **Resetcolor()** | Reset the background and foreground color to its default state |
| **Write(“string”)** | Display the specified message on the console window |
| **WriteLine(“string”)** | Same as the write method but automatically moves the cursor to the next line after printing the message. |
| **Write(variable)** | Displays the value of the given variable |
| **WriteLine(variable)** | Displays the value of the given variable along with moving the cursor to the next line after printing the value of the variable. |
| **Read()** | Read a single character from the keyboard and returns its ASCII value. The Datatype should be int as it returns the ASCII value. |
| **ReadLine()**    **ReadKey()** | Reads a string value from the keyboard and returns the entered value only. As it returns the entered string value so the DataType is going to be a string.  This method reads a single character from the keyboard and returns that character. The Datatype should be int as it returns the ASCII value. It is a STRUCT Data type which is ConsoleKeyInfo. |

##### ****Example: Program to show the use of the Write and WriteLine method:****

**<**span style="font-family: arial, helvetica, sans-serif;"**>namespace** FirstProgram

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Console.WriteLine**(**"HELLO"**)**;

Console.Write**(**"WELCOME"**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**<**/span**>**

###### **Output:**

Console Class Methods and Properties in C#

###### **Example: Program to show how to print the value of a variable in a console application.**

**<**span style="font-family: arial, helvetica, sans-serif;"**>namespace** FirstProgram

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string name = "pranaya";

Console.WriteLine**(**name**)**;

Console.Write**(**"hello " + name**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**<**/span**>**

###### **Output:**

Console Class Methods and Properties in C#

**Note**: Read the values at runtime using the ReadLine() method.

###### **Example: Program to show how to read the value at runtime in a console application.**

**<**span style="font-family: arial, helvetica, sans-serif;"**>namespace** FirstProgram

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Console.WriteLine**(**"ENTER YOUR NAME"**)**;

String name = Console.ReadLine**()**;

Console.Write**(**"hello " + name**)**;

Console.ReadKey**()**;

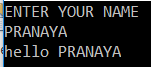
**}**

**}**

**}**

**<**/span**>**

###### **Output:**



###### **Example: Program to take two numbers as input from the console and then print the summation in the console.**

**<**span style="font-family: arial, helvetica, sans-serif;"**>namespace** FirstProgram

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** a, b, c;

Console.WriteLine**(**"ENTER TWO NUMBER"**)**;

a = **int**.Parse**(**Console.ReadLine**())**;

b = Convert.ToInt32**(**Console.ReadLine**())**;

c = a + b;

Console.WriteLine**(**"THE SUM IS :" + c**)**;

Console.WriteLine**(**"THE SUM IS : " + **(**a + b**))**;

Console.ReadKey**()**;

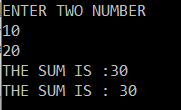
**}**

**}**

**}**

**<**/span**>**

###### **Output:**



**Note:** The ReadLine method always accepts the value in the form of a string. So we need to convert the values to the appropriate type. In the above example, we are converting the values to integer type by using **int.Parse** and **Convert.ToInt** methods. We will discuss these concepts in detail in a later article.

###### **Example:**

###### **Program to accept employee details like empno, name, salary, address, job, and print the accepted information.**

**<**span style="font-family: arial, helvetica, sans-serif;"**>namespace** FirstProgram

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** EID, SALARY;

string ENAME, ADDRESS, JOB;

Console.WriteLine**(**"ENTER THE EMPLOYEE DTAILS"**)**;

Console.WriteLine**(**"ENTER THE EMPLOYEE ID"**)**;

EID = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"ENTER THE EMPLOYEE NAME"**)**;

ENAME = Console.ReadLine**()**;

Console.WriteLine**(**"ENTER THE EMPLOYEE SALARY"**)**;

SALARY = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"ENTER THE EMPLOYEE ADDRESS "**)**;

ADDRESS = Console.ReadLine**()**;

Console.WriteLine**(**"ENTER THE EMPLOYEE JOB"**)**;

JOB = Console.ReadLine**()**;

Console.WriteLine**(**"\n\n\nTHE EMPLOYEE DETAILS ARE GIVEN BELOW :"**)**;

Console.WriteLine**(**"THE EMPLOYEE ID IS: " + EID**)**;

Console.WriteLine**(**"THE EMPLOYEE NAME IS: " + ENAME**)**;

Console.WriteLine**(**"THE EMPLOYEE SALARY IS: " + SALARY**)**;

Console.WriteLine**(**"THE EMPLOYEE ADDRESS IS: " + ADDRESS**)**;

Console.WriteLine**(**"THE EMPLOYEE JOB IS: " + JOB**)**;

Console.ReadKey**()**;

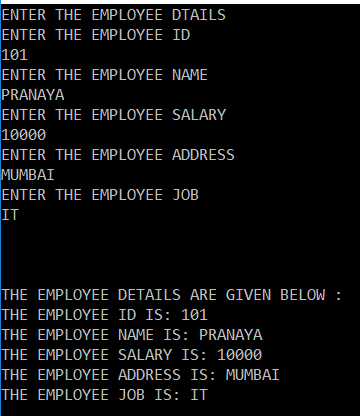
**}**

**}**

**}**

**<**/span**>**

###### **Output:**



###### **Example:**

Program to accept student no, student name, mark1, mark2, mark3 and calculate the total mark and average marks and printing accepted information.

**<**span style="font-family: arial, helvetica, sans-serif;"**>namespace** FirstProgram

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** SNO, MARK1, MARK2, MARK3, TOTAL, AVERAGE;

string SNAME;

Console.WriteLine**(**"ENTER THE STUDENT DETAILS"**)**;

Console.WriteLine**(**"ENTER THE STUDENT NO"**)**;

SNO = **int**.Parse**(**Console.ReadLine**())**;

Console.WriteLine**(**"ENTER THE STUDENT NAME"**)**;

SNAME = Console.ReadLine**()**;

Console.WriteLine**(**"ENTER THE MARKS OF 3 SUBJECTS"**)**;

MARK1 = **int**.Parse**(**Console.ReadLine**())**;

MARK2 = **int**.Parse**(**Console.ReadLine**())**;

MARK3 = **int**.Parse**(**Console.ReadLine**())**;

TOTAL = MARK1 + MARK2 + MARK3;

AVERAGE = TOTAL / 3;

Console.WriteLine**(**"\n\n\nTHE STUDENT DETAILS ARE GIVEN BELOW :"**)**;

Console.WriteLine**(**"THE STUDENT NO IS: " + SNO**)**;

Console.WriteLine**(**"THE STUUDENT NAME IS: " + SNAME**)**;

Console.WriteLine**(**"TOTAL MARKS IS : " + TOTAL**)**;

Console.WriteLine**(**"AVEARGE MAARK IS: " + AVERAGE**)**;

Console.ReadKey**()**;

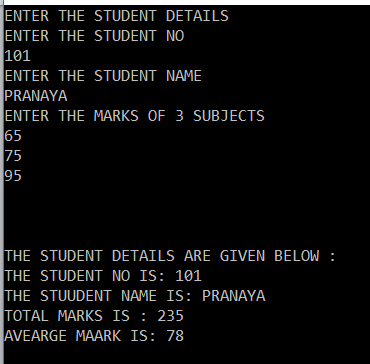
**}**

**}**

**}**

**<**/span**>**

###### **Output:**



###### **Example:**

Program to show the use of BackgroundColor, ForegroundColor, and Title properties of Console class in C#.

**<**span style="font-family: arial, helvetica, sans-serif;"**>namespace** FirstProgram

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Console.BackgroundColor = ConsoleColor.Blue;

Console.ForegroundColor = ConsoleColor.White;

Console.Title = "Understanding Console Class";

Console.WriteLine**(**"BackgroundColor Blue"**)**;

Console.WriteLine**(**"ForegroundColor White"**)**;

Console.WriteLine**(**"Title Understanding Console Class"**)**;

Console.ReadKey**()**;

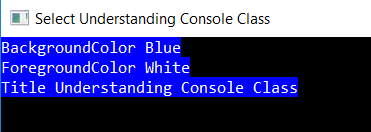
**}**

**}**

**}**

**<**/span**>**

###### **Output:**



# Data Types in C#

## ****Data Types in C# with Examples****

In this article, I am going to discuss the **Data Types in C#**with examples. Please read our previous article where we discuss the [**Console class Methods and Properties in C#**](https://dotnettutorials.net/lesson/console-class-methods-properties-csharp/) before proceeding to this article. As a developer, it is very important to understand the **Data Type in C#**. This is because you need to decide which data type to use for a specific type of value. As part of this article, we are going to discuss the following pointers related to C# data type in detail.

1. **Why we need data types in C#?**
2. **What is a data type in C#?**
3. **Different types of Data type in C#.**
4. **What is Value Data Type in C#?**
5. **What is Reference Data Type in C#?**
6. **Examples using Built-in Data Types in C#.**
7. **What is Pointer Type?**
8. **Examples using Escape Sequences in C#.**

##### ****Why we need data types in C#?****

The **Datatypes in C#** are basically used to store the data temporarily in the computer through a program. In the real world, we have different types of data like **integer, floating-point, character, string,** etc. To store all these different kinds of data in a program to perform business-related operations, we need the data types.

##### ****What is a data type in C#?****

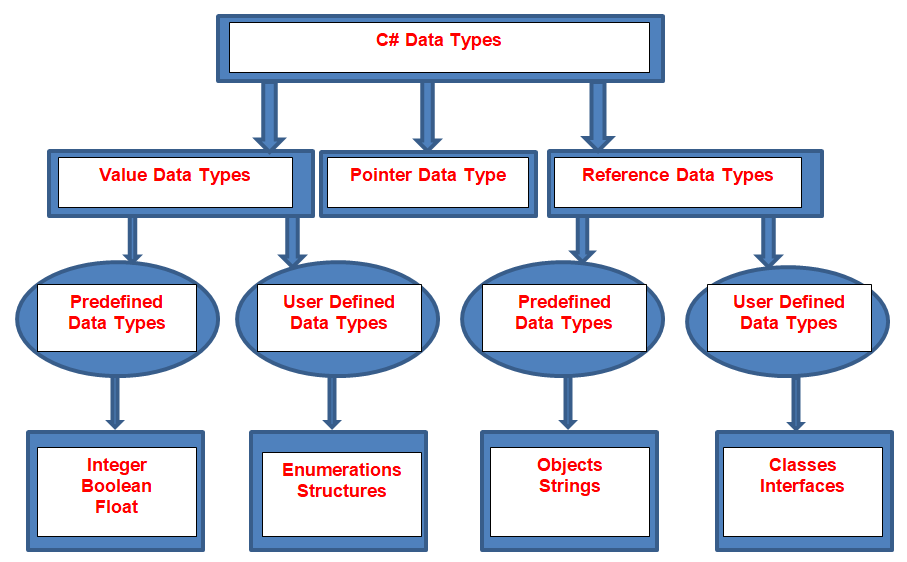
The Datatypes are something which gives information about

1. **Size** of the memory location.
2. The **range of data** that can be stored inside that memory location
3. Possible **legal operations** that can be performed on that memory location.
4. What **types of results** come out from an expression when these types are used inside that expression.

The keyword which gives all the above information is called the **data type**.

#### ****What are the different types of Data types available in C#?****

A **data type in C#** specifies the type of data that a variable can store such as **integer, floating, character, string**, etc. The following diagram shows the different types of data types available in C#.



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

1. Value Data Type
2. Reference Data Type
3. Pointer Data Type

Let us discuss each of these data types in detail

#### ****What is Value Data Type in C#?****

The data type which stores the value directly called the Value Data Type. They are derived from the class **System.ValueType**. The examples are **int, char,**and **float** which stores **numbers, alphabets,** and **floating-point** numbers respectively. The value data types in C# again classified into two types are as follows.

1. **Predefined Data Types** – Example includes **Integer, Boolean, Float,** etc.
2. **User-defined Data Types** – Example includes **Structure, Enumerations,** etc.

Based on the Operating system (32 or 64-bit), the size of the memory of the data types may change.  If you want to know the actual size of a type or a variable on a particular operating system, then you can make use of the **sizeof** method.

Let’s understand this with an example. The following example gets the size of int type on any platform.

**namespace** *FirstProgram*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Console.WriteLine**(**"Size of int: {0}", sizeof**(int))**;

Console.ReadKey**()**;

**}**

**}**

**}**

When we execute the above code, it gives the following output:

Data Types in Csharp

##### ****What is Reference Data Type in C#?****

The data type which is used to store the reference of a variable is called Reference Data Types. In other words, we can say that the reference types do not store the actual data stored in a variable, rather they store the reference to the variables. We will discuss this concept in a later article.

Again, the Reference Data Types are categorized into 2 types. They are as follows.

1. **Predefined Types** – Examples include Objects, String, and dynamic.
2. **User-defined Types** – Examples include Classes, Interface.

##### ****What is Pointer Type?****

The pointer in C# language is a variable, it is also known as a locator or indicator that points to an address of the value that means pointer type variables stores the memory address of another type. We will discuss this concept in detail in a later article.

##### ****Built-in Data Types in C#****

The built-in Data Types in C# are as follows

1. **Boolean type** – Only true or false
2. **Integral Types** – sbyte, byte, short, ushort, int, uint, long, ulong, char
3. **Floating Types** – float and double
4. **Decimal** Types
5. **String** Type

##### ****What is Escape Sequence in C#?****

Verbatim Literal is a string with an **@** symbol prefix, as in **@“Hello”**.  The Verbatim literals make escape sequences translate as normal printable characters to enhance readability.

**Without Verbatim Literal:** **“C:\\Pranaya\\DotNetTutorials\\Csharp”** – Less Readable

**With Verbatim Literal:** **@“C:\Pranaya\ DotNetTutorials\Csharp”** – Better Readable

**Let’s understand this with an example:**

**namespace** *FirstProgram*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

// Displaying double quotes in c#

string Name = "\"Dotnettutorials\"";

Console.WriteLine**(**Name**)**;

// Displaying new line character in c#

Name = "One\nTwo\nThree";

Console.WriteLine**(**Name**)**;

// Displaying new line character in c#

Name = "c:\\Pranaya\\Dotnettutorials\\Csharp";

Console.WriteLine**(**Name**)**;

// C# verbatim literal

Name = @"c:\Pranaya\Dotnettutorials\Csharp";

Console.WriteLine**(**Name**)**;

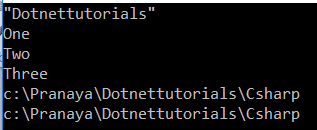
Console.ReadKey**()**;

**}**

**}**

**}**

###### **Output:**



# C# String in Depth

## ****C# String in Depth with Examples****

In this article, I am going to discuss **C# String in Depth**with examples. Please read our previous article, where we discussed [**C# Data Types**](https://dotnettutorials.net/lesson/data-types-in-csharp/) in detail. As a developer, it is very important to understand the concept of C# strings and I am also sure you are using the C# string in all of your projects. But there are many things that you should know from a performance point of view. So, as part of this article, we are going to discuss the following pointers in detail with examples.

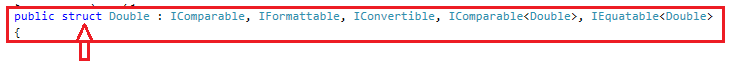
1. **Strings are reference types**
2. **Understanding the difference between string(small) vs String(Capital).**
3. **Why strings are immutable?**
4. **How we can improve performance using String intern?**
5. **StringBuilder for concatenation.**
6. **Why they made string is immutable.**

##### ****Strings are reference types in C#:****

C# strings are objects i.e. they are not normal data types. For example, if we define some variables using int or double data types as shown below,

Primitive Types are Struct in C#

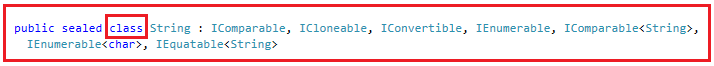
Then if you right-click on the data type and go to the definition then you will see that they are struct as shown in the below image. Struct means they are value type.



On the other hand, if you define a variable with string data type as shown below.

string in C#

Then if you right-click on the string data type and click on go to definition then you will see that it is a class. Class means reference data type.



So, the first point that you need to remember is strings are reference types while other primitive data types are struct types i.e. value type in C#.

##### ****What are the Differences between String(Capital) vs string(small) in C#?****

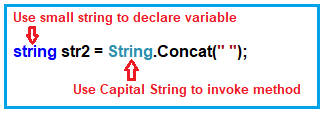
In C#, you can use the string in two ways i.e. you can use the string using capital S (i.e. String) or by using the small “s” (i.e. string) as shown in the below image.

String(Capital) vs string(small) in C#

Now the question that should come to your mind is what is the difference between these two (string vs String) in C#. Let’s understand this. The small string is actually an alias of String (Capital string). If you right-click on the small string and if you go to the definition then you will see that the actual class name is capital string i.e. String as shown in the below image.

what is the difference between string and String in C#

You can use any one of them i.e. either string or String. But as per the naming convention when you are creating a variable use the small string (i.e. string) and whenever you want to invoke methods on the string then use the capital string (i.e. String) as shown in the below image.



##### ****Strings are Immutable in C#:****

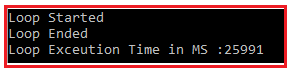
Before understanding strings are immutable, first, we need to understand two terms i.e. Mutable and Immutable. Mutable means can be changed whereas Immutable means can not be changed. C# strings are immutable means C# strings cannot be changed. Let us understand this with an example.

Please have a look at the below image. When the first statement is executed, it will create one object and assign the value DotNet. But when the second statement is executed, it will not override the first object, it lets the first object be there for garbage collection and creates a fresh object, and assign the value Tutorials.

Strings are immutable in C#

So, when the above two statements are executed, internally two memory locations are created. One with the value DotNet and the current one with the value Tutorials and the current one is going to be referred in the program. So, each time, we assign a new value to the string variable, a new object is created and this is the reason why strings are immutable in C#.

But this is not the case with a value type. For example, please have a look at the below two statements. When the first statement is executed one memory location is created and assign the value 100 and when the second statement is executed, it will not create a new memory location rather it will override the value of the same memory location.



##### Example to Proves C# strings are Immutable:

Let us see an example to understand C# strings are Immutable. Please copy and paste the following code. As you can see here we have a heavy loop. As part of the Loop, we assigning a value to the string str variable. Here, we are using GUID to generate a new value, and each time it will create a new value and assign it to the str variable. Again, we are using Stopwatch to check how much time it took to execute the loop.

**using** *System;*

**using** *System.Diagnostics;*

**namespace** *StringDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string str = "";

Console.WriteLine**(**"Loop Started"**)**;

var stopwatch = new Stopwatch**()**;

stopwatch.Start**()**;

**for** **(int** i = 0; i **<** 30000000; i++**)**

**{**

str = Guid.NewGuid**()**.ToString**()**;

**}**

stopwatch.Stop**()**;

Console.WriteLine**(**"Loop Ended"**)**;

Console.WriteLine**(**"Loop Exceution Time in MS :" + stopwatch.ElapsedMilliseconds**)**;

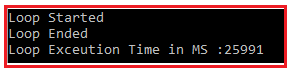
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:** When you execute the program, you will get the following output. The time may vary in your machine.



As you can see in the above output, it approximately took 26000 milliseconds to execute the loop. Each time the loop executes, it creates a fresh string object and assigns the new value to it. This is because strings are immutable in C#.

##### ****Example using Integer in C#:****

In the following C# example, instead of a string, we are using an integer variable. As integers are not immutable, so it will not create a fresh memory location each time the loop executes instead it will use the same memory location and update its value.

**using** *System;*

**using** *System.Diagnostics;*

**namespace** *StringDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** ctr =0;

Console.WriteLine**(**"Loop Started"**)**;

var stopwatch = new Stopwatch**()**;

stopwatch.Start**()**;

**for** **(int** i = 0; i **<** 30000000; i++**)**

**{**

ctr = ctr + 1;

**}**

stopwatch.Stop**()**;

Console.WriteLine**(**"Loop Ended"**)**;

Console.WriteLine**(**"Loop Exceution Time in MS :" + stopwatch.ElapsedMilliseconds**)**;

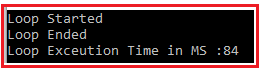
Console.ReadKey**()**;

**}**

**}**

**}**

###### **Output:**



As you can see in the above output, it only took 84 milliseconds to execute the loop.

##### ****Example: String with Same value in C#****

Let us understand what will happen if we assign the same value to the string variable again and again with an example in C#. As you can see in the below example, which is exactly the same as the first example, but here instead of using GUID, we are assigning a fixed value to the string str variable.

**using** *System;*

**using** *System.Diagnostics;*

**namespace** *StringDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string str = "";

Console.WriteLine**(**"Loop Started"**)**;

var stopwatch = new Stopwatch**()**;

stopwatch.Start**()**;

**for** **(int** i = 0; i **<** 30000000; i++**)**

**{**

str ="DotNet Tutorials";

**}**

stopwatch.Stop**()**;

Console.WriteLine**(**"Loop Ended"**)**;

Console.WriteLine**(**"Loop Exceution Time in MS :" + stopwatch.ElapsedMilliseconds**)**;

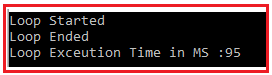
Console.ReadKey**()**;

**}**

**}**

**}**

###### **Output:**



As you can see in the above output it only took 95 milliseconds. This is because in this case fresh objects are not created each time the loop executes. Now, the question that should come to your mind is why? The answer is **String intern**. So, let us understand string interning in detail.

##### ****String Intern in C#:****

The **String Intern in C#** is a process that uses the same memory location if the value is the same. In our example, when the loop executes for the first time, it will create a fresh object and assign the value “**DotNet Tutorials**” to it. When the loop executes 2nd time, before creating a fresh object, it will check whether this “**DotNet Tutorials**” value is already there in the memory, if yes then it simply uses that memory location else it will create a new memory location. This is nothing but C# string interning.

So, if you are running a for loop and assigning the same value again and again, then it uses string interning to improve the performance. In this case, rather than creating a new object, it uses the same memory location. But when the value changes it will create a new fresh object and assign the value to the new object.

##### ****StringBuilder for Concatenation in C#:****

As we already discussed if the value changes then every time it will create a new fresh object in C# and this is because of the Immutability behavior of the string. The C# string immutability behavior can be very very dangerous when it comes to string concatenation. Let us understand string concatenation in C# with an example and understand the problem. In the below example, we are concatenating the string using the for loop.

**using** *System;*

**using** *System.Diagnostics;*

**namespace** *StringDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string str = "";

Console.WriteLine**(**"Loop Started"**)**;

var stopwatch = new Stopwatch**()**;

stopwatch.Start**()**;

**for** **(int** i = 0; i **<** 30000; i++**)**

**{**

str ="DotNet Tutorials" + str;

**}**

stopwatch.Stop**()**;

Console.WriteLine**(**"Loop Ended"**)**;

Console.WriteLine**(**"Loop Exceution Time in MS :" + stopwatch.ElapsedMilliseconds**)**;

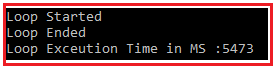
Console.ReadKey**()**;

**}**

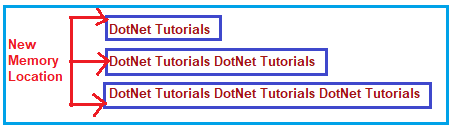
**}**

**}**

###### **Output:**



As you can see in the above image, it took approximately 5473 milliseconds to execute the loop. In order to understand how it executes the loop, please have a look at the below image. The loop executes the first time, it will create a new memory location and store the value “DotNet Tutorials”. For the second time, it creates another fresh memory location (fresh object) and stores the value “DotNet Tutorials DotNet Tutorials” and the first memory location will be going for garbage collection. And the same process will continue i.e. each time the loop executes a new memory location will be created and previous ones will be going for garbage collection.



In order to solve the above **String Concatenation Problem in C#**, the .NET Framework provides the **StringBuilder** class. As the name itself saying everything, the string builder class in C# is used to build a string. If you use string builder then fresh objects are not going to be created every time you concatenate something to the string variable in C#.

##### ****Example using StringBuilder in C#:****

Let us understand how to overcome the **String Concatenation Problem in C#** using the **StringBuilder** class. In the following example, we are using the StringBuilder class to concatenate strings. Here, first, we create an instance of the StringBuilder class and then use the **Append** method of the **StringBuilder** class to concatenate the string.

**using** *System;*

**using** *System.Diagnostics;*

**using** *System.Text;*

**namespace** *StringDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

StringBuilder stringBuilder = new StringBuilder**()**;

Console.WriteLine**(**"Loop Started"**)**;

var stopwatch = new Stopwatch**()**;

stopwatch.Start**()**;

**for** **(int** i = 0; i **<** 30000; i++**)**

**{**

stringBuilder.Append**(**"DotNet Tutorials"**)**;

**}**

stopwatch.Stop**()**;

Console.WriteLine**(**"Loop Ended"**)**;

Console.WriteLine**(**"Loop Exceution Time in MS :" + stopwatch.ElapsedMilliseconds**)**;

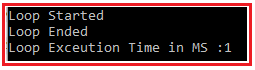
Console.ReadKey**()**;

**}**

**}**

**}**

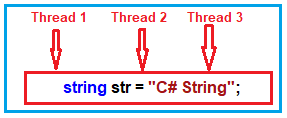
###### **Output:**



As you can see in the above output, it only took 1 millisecond to concatenate the string comparing to 5473 using string. This is because every time the for loop runs it will not create fresh objects rather than it will use the same memory location i.e. the same old object which drastically improves the application performance.

##### ****Why they made C# String Immutable?****

Now the question is why they made strings as Immutable in C#. **They made Strings as Immutable for Thread Safety**. Think of one situation where you have many threads and all the threads want to manipulate the same string object as shown in the below image. If strings are mutable then we have thread-safety issues.



In case if you are new to thread safety, I strongly recommended you to read the following article, where we discussed Thread and Thread Safety in detail.

[**https://dotnettutorials.net/lesson/multithreading-in-csharp/**](https://dotnettutorials.net/lesson/multithreading-in-csharp/)

# Static Keyword in C#

## ****Static Keyword in C# with Examples****

In this article, I am going to discuss **why do we need the keyword Static in C#** with examples. Please read our previous article, where we discussed [**C# String**](https://dotnettutorials.net/lesson/string-in-csharp/) in detail. At the end of this article, I am sure you will understand the need and use of Static Keyword in C# with examples.

##### ****Why do we need Static Keyword in C#?****

If you ask this question to any developers, they most probably answer you that the static keyword is used in Factory Design Pattern, Singleton Design Pattern as well as used for data sharing, etc. But I think, the static keyword is used for three basic purposes. And in this article, we will discuss these three purposes in detail. I hope you are going to enjoy this article.

##### ****Example to understand the Static Keyword in C#:****

Let us understand the need and use of the **C# Static Keyword** with an example. First, create a console application with the name StaticKeyowrdDemo.

##### ****CountryMaster.cs:****

Once you created the Console application, then create a class file with the name**CountryMaster.cs**and then copy and paste the following code into it.

**namespace** *StaticKeyowrdDemo*

**{**

**public** **class** CountryMaster

**{**

**public** string CountryCode **{** **get**; **set**; **}**

**public** string CountryName **{** **get**; **set**; **}**

**private** string ComputerName

**{**

**get**

**{**

**return** System.Environment.MachineName;

**}**

**}**

**public** **void** Insert**()**

**{**

**}**

**}**

**}**

##### ****Explanation of Above Class:****

Here we created the class with three properties and one method. The CountryCode property is going to hold the three-letter symbols of the country while the CountryName property holds the full country name. The ComputerName property has the logic to retrieve the current machine name.

The Insert Method inserts the country record into the database and while inserting it also uses the ComputerName property to tells that from which computer this record was inserted.

##### ****Customer.cs****

Now, create a new class file with the name **Customer.cs** and then copy and paste the following code in it.

**namespace** *StaticKeyowrdDemo*

**{**

**public** **class** Customer

**{**

**public** string CustomerCode **{** **get**; **set**; **}**

**public** string CustomerName **{** **get**; **set**; **}**

**private** string MachineName = "";

**private** **bool** IsEmpty**(**string **value)**

**{**

**if(value**.Length **>** 0**)**

**{**

**return** **true**;

**}**

**return** **false**;

**}**

**public** **void** Insert**()**

**{**

**if(**IsEmpty**(**CustomerCode**)** && IsEmpty**(**CustomerName**))**

**{**

//Insert the data

**}**

**}**

**}**

**}**

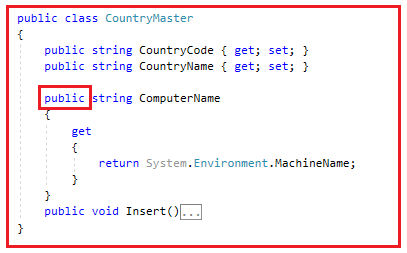
##### ****Explanation of Above Code:****

The CustomerCode property is going to hold the three-letter code of the customer while the CustomerName property holds the customer name. The IsEmpty method accepts one value and then check if the value is empty or not. If not empty then return true else return false. The Insert method simply checks if both CustomerCode and CustomerName are not empty then insert the customer record into the database.

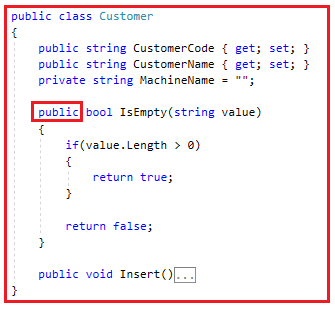
Here, the problem is with the **MachineName**variable. The **MachineName**should have the current computer name while inserting the customer data into the database so that we can track from which machine this customer data was inserted

If you remember, the **CountryMaster**class has the logic to retrieve the computer name. Rather than writing the duplicate logic here, we should go and use the logic which is already written in the **CountryMaster**class, so that we can avoid writing the repeating code or redundant code.

If you check the ComputerName property in the class **CountryMaster.cs**file, then you will see that, it is private, so in order to use that property in the Customer class, first, we need to change it to the public as shown in the below image.



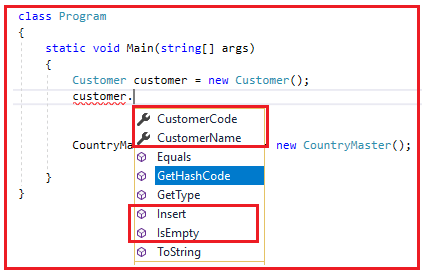
Again while inserting the CountryMaster record into the database, we also need to check both CountryCode and CountryName should not be empty. To check if empty or not, we also like to use the **IsEmpty**method which is defined in the Customer class rather than writing the complete logic here. Further, if you notice, the IsEmpty method of the Customer class is private, so in order to use that method in CountryMaster class, we need to change it to the public as shown in the below image.



The CountryMaster class has logic to retrieve the Computer name and we want to use that logic in the Customer class so we made the ComputerName property public. Similarly, the Customer class has logic check whether a value is empty or not and we also want that logic in the CountryMaster class, so we made the IsEmpty method as public. As long as we did this, we violate the encapsulation principle.

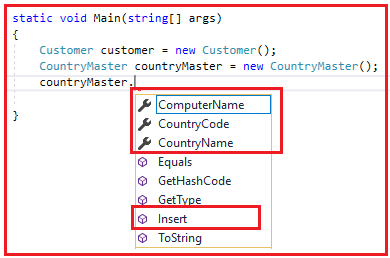
##### ****How we are violating the Encapsulation Principle?****

Let us understand how we are violating the encapsulation principle. Modify the Program class as shown below. Once you created the Customer class object, then you can see the public member of that class as shown in the below image.



As you can see, we have exposed the CustomerCode, CustomerName, Insert, and IsEmpty methods. There is a clear violation of abstraction. Abstraction means shows only what is necessary. So, the external person who is consuming your class, should see and consume the **CustomerCode**, **CustomerName,** and **Insert** method. But should not see the **IsEmpty** method. The **IsEmpty** method is for internal use i.e. use by other methods not by the consumer of the class. As we make the IsEmpty method as public, we are violating the Encapsulation principle.

In the same way, we also violating the abstraction principle with the **CountryMaster**object as we are exposing the **ComputerName**property to the external world that is going to consume the class as shown in the below image. The **ComputerName** property is for internal use only.



**Note:**With the above, we are achieving code reusability (reusing the ComputerName and IsEmpty method) but violating the encapsulation principle.

##### ****How to solve this problem?****

How to solve the above problem means how should we achieve code reusability but without violating the OOPs principles (i.e. Encapsulation Principle). In order to achieve both, let us add a new class and then move those two functions into that class. Create a class file with the name **CommonTask.cs**and then copy and paste the following code in it.

**namespace** *StaticKeyowrdDemo*

**{**

**public** **class** CommonTask

**{**

**public** **bool** IsEmpty**(**string **value)**

**{**

**if** **(value**.Length **>** 0**)**

**{**

**return** **true**;

**}**

**return** **false**;

**}**

**public** string GetComputerName**()**

**{**

**return** System.Environment.MachineName;

**}**

**}**

**}**

Please remove the IsEmpty() method from the Customer class and the ComputerName property from the CountryMaster class. Now both the logic which violates the OOPs principle has been moved to the **CommonTask**class.

##### ****Modifying Customer class:****

Now modify the Customer class as shown below. As you can see, in the constructor we set the value of the MachineName private variable and in the Insert method, we create an instance of **CommonTask** class and Invoke the **IsEmpty** method.

**namespace** *StaticKeyowrdDemo*

**{**

**public** **class** Customer

**{**

**public** string CustomerCode **{** **get**; **set**; **}**

**public** string CustomerName **{** **get**; **set**; **}**

**private** string MachineName = "";

**public** Customer**()**

**{**

CommonTask commonTask = new CommonTask**()**;

MachineName = commonTask.GetComputerName**()**;

**}**

**public** **void** Insert**()**

**{**

CommonTask commonTask = new CommonTask**()**;

**if(**!commonTask.IsEmpty**(**CustomerCode**)** && !commonTask.IsEmpty**(**CustomerName**))**

**{**

//Insert the data

**}**

**}**

**}**

**}**

##### ****Modifying the CountryMaster class:****

Please modify the **CountryMaster** class as shown below. Here, we created the instance of **CommonTask** and then Invoke the GetComputerName and IsEmpty methods.

**namespace** *StaticKeyowrdDemo*

**{**

**public** **class** CountryMaster

**{**

**public** string CountryCode **{** **get**; **set**; **}**

**public** string CountryName **{** **get**; **set**; **}**

**private** string ComputerName

**{**

**get**

**{**

CommonTask commonTask = new CommonTask**()**;

**return** commonTask.GetComputerName**()**;

**}**

**}**

**public** **void** Insert**()**

**{**

CommonTask commonTask = new CommonTask**()**;

**if** **(**!commonTask.IsEmpty**(**CountryCode**)** && !commonTask.IsEmpty**(**CountryName**))**

**{**

//Insert the data

**}**

**}**

**}**

**}**

As we centralized the **IsEmpty** and **GetComputerName**method in the **CommonTask** class, we can use these methods in both the **Customer** and **CountryMaster** classes. The above solution seems to be decent as it does not violate the **OOPs Principle** and also achieving code reusability and I hope many of you are also agree to it. But there is also some problem.

##### ****What is the problem in the above solution?****

In order to understand the problem, let us first analyze the **CommonTask** class in a great manner.

##### ****Note about CommonTask class:****

1. This **CommonTask** class is a collection of unrelated methods and properties that are not related to each other. Because it has unrelated methods, properties, or logic, it does not represent any real-world objects.
2. As it does not represent any real-world objects, so any kind of OOPs principles (inheritance, abstraction, polymorphism, encapsulation) should not be allowed to be applied to this CommonTask class.
3. So, in simple words, we can say that this is a fixed class i.e. a class with a fixed behavior. That is, its behavior can not be changed by inheritance, its behavior can not be polymorphised by using either static or dynamic polymorphism. So, we can say that this class is a fixed class or static class.

###### **How do we avoid Inheritance, how do we avoid abstract keywords, or how do we avoid the OOPs principle on a class?**

The answer is by using the **static** keyword. So, you need to mark the **CommonTask** class as **static** by using the static keyword. When you mark a class as **static**, everything inside the class should be static. That means, we also need to mark the **IsEmpty** and **GetComputerName** methods as static. So, modify the **CommonTask** class as shown below.

**namespace** *StaticKeyowrdDemo*

**{**

**public** **static** **class** CommonTask

**{**

**public** **static** **bool** IsEmpty**(**string **value)**

**{**

**if** **(value**.Length **>** 0**)**

**{**

**return** **true**;

**}**

**return** **false**;

**}**

**public** **static** string GetComputerName**()**

**{**

**return** System.Environment.MachineName;

**}**

**}**

**}**

Once you make the class static, then you cannot use the **new**keyword with the static class to create an instance, rather you need to invoke the **IsEmpty** and **GetComputerName** methods by using the class name. Internally only one instance of the static class gets created by CLR which serves all the clients.

##### ****Modify the Customer class:****

Now modify the Customer class as shown below. As you can see, now we are invoking the **GetComputerName** and **IsEmpty** method using the class name i.e. **CommonTask**.

**namespace** *StaticKeyowrdDemo*

**{**

**public** **class** Customer

**{**

**public** string CustomerCode **{** **get**; **set**; **}**

**public** string CustomerName **{** **get**; **set**; **}**

**private** string MachineName = "";

**public** Customer**()**

**{**

MachineName = CommonTask.GetComputerName**()**;

**}**

**public** **void** Insert**()**

**{**

**if(**!CommonTask.IsEmpty**(**CustomerCode**)** && !CommonTask.IsEmpty**(**CustomerName**))**

**{**

//Insert the data

**}**

**}**

**}**

**}**

##### ****Modify the CountryMaster class:****

Modify the **CountryMaster** class as shown below. As you can see in the below code, we are invoking the **GetComputerName** and **IsEmpty** method using the class name i.e. **CommonTask**.

**namespace** *StaticKeyowrdDemo*

**{**

**public** **class** CountryMaster

**{**

**public** string CountryCode **{** **get**; **set**; **}**

**public** string CountryName **{** **get**; **set**; **}**

**private** string ComputerName

**{**

**get**

**{**

**return** CommonTask.GetComputerName**()**;

**}**

**}**

**public** **void** Insert**()**

**{**

**if** **(**!CommonTask.IsEmpty**(**CountryCode**)** && !CommonTask.IsEmpty**(**CountryName**))**

**{**

//Insert the data

**}**

**}**

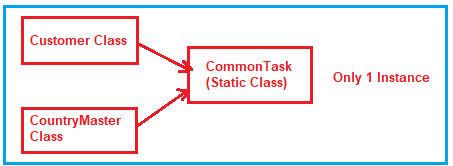
**}**

**}**

##### ****How is the static class instantiated?****

We cannot apply any OOPs principles to the static class like inheritance, polymorphism, encapsulation, and abstraction. But in the end, it is a class. And at least to use a class it has to be instantiated. If the static class is not instantiated then we cannot invoke the methods and properties that are present in the static class. Now let us see how does the instantiation takes place internally of a static class i.e. in our example, it is the **CommonTask** class.

The CLR (Common Language Runtime) will create only one instance of the **CommonTask** class irrespective of whether how many times they called from the **Customer** and **CountryMaster** class. For better understanding, please have a look at the below image.



Due to the single instance behavior, the static class is also going to be used to share the common data.

# Static and Non-Static Members in C#

## ****Static and Non-Static Members in C# with Examples****

In this article, I am going to discuss the **Static and Non-Static Members in C#** with some examples. Please read our previous article before proceeding to this article where we discussed the [**Data Type in C#**](https://dotnettutorials.net/lesson/data-types-in-csharp/) with examples. At the end of this article, you will be having a very good understanding of the following pointers.

1. **What are static and non-static members in C#?**
2. **When do we need to use static and non-static members in C#?**
3. **Static and Non-static variables in C#.**
4. **What is the scope of Non-Static variables in C#?**
5. **Static and Non-Static methods in C#.**
6. **What is Static and Non-Static Constructor in C#?**
7. **Understanding Static class in C#.**

##### ****What are static and non-static members in C#?****

The member of a class is divided into two categories

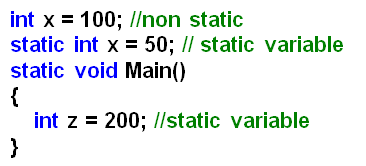
1. **Static members**
2. **Non-static members**

In simple words, we can define that, the members of a class that does not require an instance for initialization or execution are known as the static member. On the other hand, the members which require an instance of a class for both initialization and execution are known as non-static members.

##### ****Understanding the Static and Non-static Variables in C#****

Whenever we declare a variable by using the static modifier in C# or when we declare a variable inside of any static block then those variables are considered as static variables whereas the rest of the others are considered as non-static variables.

If you want a variable to have the same value throughout all instances of a class then you need to declare that variable as a static variable. So, the static variables are going to hold the application level data which is going to be the same for all the objects.  Have a look at the following example.



The static variable gets initialized immediately once the execution of the class starts whereas the non-static variables are initialized only after creating the object of the class and that is too for each time the object of the class is created.

A static variable gets initialized only once during the life cycle of a class whereas a non-static variable gets initialized either 0 or n number of times, depending on the number of objects created for that class.

If you want to access the static members of a class, then you need to access them using the class name whereas you need an instance of a class to access the non-static members.

##### ****Example: Static and Non-static Variables in C#****

Let us see an example for a better understanding of the static and non-static variables in C#. Please have a look at the below example. Here, we created two variables one is static (i.e. static int y = 200;) and one non-static variable (i.e. int x;). Then using the constructor of the class we initialize the non-static variable.

**namespace** *StaticNonStaticDemo*

**{**

**class** Example

**{**

**int** x; // Non statuc variable

**static** **int** y = 200; //Static Variable

**public** Example**(int** x**)**

**{**

this.x = x;

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

//Accessing the static variable using class name

//Before object creation

Console.WriteLine**(**"Static Variable Y = " + Example.y**)**;

//Creating object1

Example obj1 = new Example**(**50**)**;

//Creating object2

Example obj2 = new Example**(**100**)**;

Console.WriteLine**(**$"object1 x = {obj1.x} object2 x = {obj2.x}"**)**;

Console.WriteLine**(**"Press any key to exit."**)**;

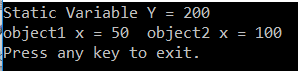
Console.ReadLine**()**;

**}**

**}**

**}**

###### **Output:**



##### ****What is the scope of Non-Static variables in C#?****

The Non Static variables are created when the object is created and are destroyed when the object is destroyed. The object is destroyed when its reference variable is destroyed or initialized with null. So we can say that the scope of the object is the scope of its referenced variables.

##### ****Static and Non-Static Methods in C#****

If we declare a method using the static modifier then it is called a static method else it is a non-static method. You cannot consume the non-static members directly within a static method. If you want to consume any non-static members with a static method then you need to create an object and then through the object, you can access the non-static members. On the other hand, you can directly consume the static members within a non-static method without any restriction.

##### ****Rules to follow while working with static and non-static members in c#:****

1. **Non-static to static:** Can be consumed only by using the object of that class.
2. **Static to static:** Can be consumed directly or by using the class name.
3. **Static to non-static:** Can be consumed directly or by using the class name.
4. **Non-static to non-static:** Can be consumed directly or by using the “**this**” keyword.

##### ****Example: Static and Non-static Methods in C#****

Let us see an example for a better understanding of the static and non-static Methods in C#. Please have a look at the below example. Here, we created two variables. One variable is static and another variable is non-static. Then we created two methods i.e. Add method which is a static method and the Mul method which is a non-static method. From the static method, we creating an instance of the Example class and calling the non-static variable and we can call the static variable directly or by using the class name. From the static method, we can call the non-static members directly and static members by using the class name.

**namespace** *StaticNonStaticDemo*

**{**

**class** Example

**{**

**int** x = 100;

**static** **int** y = 200;

**static** **void** Add**()**

**{**

//This is a static block

//we can access non static members X with the help of Example object

//We can access the static member directly or through class name

Example obj = new Example**()**;

//Console.WriteLine(obj.x + Example.y);

Console.WriteLine**(**"Sum of 100 and 200 is :" + **(**obj.x + y**))**;

**}**

**void** Mul**()**

**{**

//This is a non-static method

//we can access static members directly or through class name

//we can access the non-static members directly or through this keyword

Console.WriteLine**(**"Multiplication of 100 and 200 is :" + **(**this.x \* Example.y**))**;

Console.WriteLine**(**"Multiplication of 100 and 200 is :" + **(**x \* y**))**;

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

// Main method is a static method

// ADD() method is a static method

// Statid to Static

// we can call the add method directly or through class name

Example.Add**()**;

Add**()**;

// Mul() method is a non-static method

// we can call the non-static method using object only from a static method

// Static to non-static

Example obj = new Example**()**;

obj.Mul**()**;

Console.WriteLine**(**"Press any key to exit."**)**;

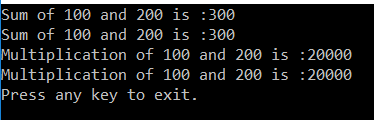
Console.ReadLine**()**;

**}**

**}**

**}**

###### **Output:**



##### ****Understanding Static and Non-Static Constructor in C#:****

If we create the constructor explicitly by the static modifier, then we call it a static constructor and the rest of the others are the non-static constructors.

The most important point that you need to remember is the static constructor is the fast block of code that gets executes under a class. No matter how many numbers of objects you created for the class the static constructor is executed only once. On the other hand, a non-static constructor gets executed only when we created the object of the class and that is too for each and every object of the class.

It is not possible to create a static constructor with parameters. This is because the static constructor is the first block of code that is going to execute under a class. And this static constructor is called implicitly, even if parameterized there is no chance of sending the parameter values.

##### ****Example: Static and Non-Static Constructor in C#****

Let us see an example for a better understanding of the static and non-static Constructors in C#. Please have a look at the below example. In the below example, we have created two constructors. Among the two constructors, one constructor is static and that static constructor is going to be executed first and that constructor is going to be executed only once in its lifetime. Once the static constructor is executed, then the main method starts its execution. Then we created two instances of the Example class and that means the non-static constructor is going to be executed two times.

**namespace** *StaticNonStaticDemo*

**{**

**class** Example

**{**

**static** Example**()**

**{**

Console.WriteLine**(**"static constructor is called"**)**;

**}**

**public** Example**()**

**{**

Console.WriteLine**(**"non-static constructor is called"**)**;

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

Console.WriteLine**(**"Main method is executed"**)**;

Example obj1 = new Example**()**;

Example obj2 = new Example**()**;

Console.WriteLine**(**"Press any key to exit."**)**;

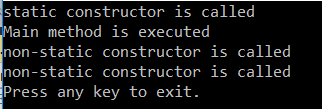
Console.ReadLine**()**;

**}**

**}**

**}**

###### **Output:**



##### ****Static class in C#:****

The class which is created by using the static modifier is called a static class in C#. A static class can contain only static members. It is not possible to create an instance of a static class. This is because it contains only static members. And we know we can access the static members of a class by using the class name.

##### ****Example: Static Class in C#****

Let us see an example for a better understanding of the static class in C#. Please have a look at the below example. As you can see in the below code, we have two classes. The first class TemperatureConverter is a static class and this class contains two static methods. As it is a static class, it can contain only static members. The TestTemperatureConverter is a normal class and from that class, we are calling the static methods by using the static class name.

**namespace** *StaticNonStaticDemo*

**{**

**public** **static** **class** TemperatureConverter

**{**

**public** **static** **double** CelsiusToFahrenheit**(**string temperatureCelsius**)**

**{**

// Convert argument to double for calculations.

**double** celsius = Double.Parse**(**temperatureCelsius**)**;

// Convert Celsius to Fahrenheit.

**double** fahrenheit = **(**celsius \* 9 / 5**)** + 32;

**return** fahrenheit;

**}**

**public** **static** **double** FahrenheitToCelsius**(**string temperatureFahrenheit**)**

**{**

// Convert argument to double for calculations.

**double** fahrenheit = Double.Parse**(**temperatureFahrenheit**)**;

// Convert Fahrenheit to Celsius.

**double** celsius = **(**fahrenheit - 32**)** \* 5 / 9;

**return** celsius;

**}**

**}**

**class** TestTemperatureConverter

**{**

**static** **void** Main**()**

**{**

Console.WriteLine**(**"Please select the convertor direction"**)**;

Console.WriteLine**(**"1. From Celsius to Fahrenheit."**)**;

Console.WriteLine**(**"2. From Fahrenheit to Celsius."**)**;

Console.Write**(**":"**)**;

string selection = Console.ReadLine**()**;

**double** F, C = 0;

**switch** **(**selection**)**

**{**

**case** "1":

Console.Write**(**"Please enter the Celsius temperature: "**)**;

F = TemperatureConverter.CelsiusToFahrenheit**(**Console.ReadLine**())**;

Console.WriteLine**(**"Temperature in Fahrenheit: {0:F2}", F**)**;

**break**;

**case** "2":

Console.Write**(**"Please enter the Fahrenheit temperature: "**)**;

C = TemperatureConverter.FahrenheitToCelsius**(**Console.ReadLine**())**;

Console.WriteLine**(**"Temperature in Celsius: {0:F2}", C**)**;

**break**;

**default**:

Console.WriteLine**(**"Please select a convertor."**)**;

**break**;

**}**

// Keep the console window open in debug mode.

Console.WriteLine**(**"Press any key to exit."**)**;

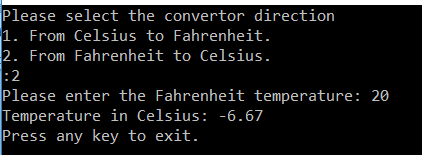
Console.ReadKey**()**;

**}**

**}**

**}**

###### **Output:**



# Const and Read-Only in C#

## ****Const and Read-Only in C# with Examples****

In this article, I am going to discuss the need and use of the keywords **Const and Read-only in C#**with examples. Please read our previous article where we discussed the [**Static and Non-Static Members in C#**](https://dotnettutorials.net/lesson/static-and-non-static-members-csharp/) with examples. The **const and read-only**are very two useful keywords in C# and also a little confusing to understand. So, as part of this article, we are going to discuss the following pointers in detail.

1. **Const Variable in C#.**
2. **Example using Const variable.**
3. **The read-Only variable in C#.**
4. **Example using the read-only variable.**
5. **Difference between Const, Readonly, Static and non-static variable in C#.**

##### ****According to MSDN****

Constants are the immutable values that are known at the time of program compilation and do not change their values for the lifetime of the program. The **Read-only** variables are also immutable values but these values are known at runtime and also do not change their values for the life of the program.

With the above definition in mind, let’s try to understand the const and readonly with some examples.

##### ****Const Variable in C#:****

1. The keyword **const** is used to create a “**constant**” variable. It means it will create a variable whose value is never going to be changed. In simple words, we can say that the variable whose value cannot be changed or modified once after its declaration is known as a constant variable.
2. Constants are static by default.
3. It is mandatory to initialize a constant variable at the time of its declaration.
4. The behavior of a constant variable is the same as the behavior of a static variable i.e. maintains only one copy in the life cycle of class and initialize immediately once the execution of the class start (object not required)
5. The only difference between a static and constant variable is that the static variable value can be modified but a constant variable value can never be modified.

**Note:**As we already discussed constants variable should be assigned a value at the time of variable declaration and hence these values are known at compile time. So, whenever we declare a constant variable, the C# compiler substitutes its value directly into the Intermediate Language (MSIL).

##### ****Example: Const Variable in C#****

Let us understand the Const keyword in C# with an example. Please have a look at the following example. As you can see in the below code, within the ConstExample class, we declare a const variable i.e. public const int number = 5; and within the program class Main method we access the const variable by using the class name. This is because const are static by default and as static it does not require an object instead it can be accessed by using the class name. It is also possible to declare local variables as const i.e. we can declare const variable within a method also. In our example, within the Main method we declare a const variable i.e. const int no = 10;. But once we declare the const variable, then we cannot change its value and if we try to change the value of the const variable in C#, then we will get a compile-time error.

**namespace** *ConstDemo*

**{**

**class** ConstExample

**{**

//we need to assign a value to the const variable

//at the time of const variable declaration else it will

//give compile time error

**public** const **int** number = 5;

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Const variables are static in nature

//so we can access them by using class name

Console.WriteLine**(**ConstExample.number**)**;

//We can also declare constant variable within a function

const **int** no = 10;

Console.WriteLine**(**no**)**;

//Once after declaration we cannot change the value

//of a constant variable. so the below live gives error

//no = 20;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

##### ****Read-only Variable in C#:****

1. The variable which is created by using the **readonly** keyword is known as a read-only variable in C#. The read-only variable’s value cannot be modified once after its initialization.
2. It is not mandatory or required to initialize the read-only variable at the time of its declaration like a constant. You can initialize the read-only variables under a constructor but the most important point is that once after initialization, you cannot modify the value.
3. The behavior of a read-only variable is similar to the behavior of a non-static variable. That is, it maintains a separate copy for each object. The only difference between these two is non-static variables value can be modified while the value of the read-only variable cannot be modified.
4. A constant variable is a fixed value for the complete class whereas a read-only variable is a fixed value but specific to one object of the class.

##### ****Example: Read-only Variable in C#****

Let us understand the readonly keyword in C# with an example. Please have a look at the following example. As you can see in the below code, within the ReadOnlyExample class, we declare a readonly variable i.e. public readonly int number = 5; and within the program class Main method we access the readonly variable by using the object of the ReadOnlyExample. This is because readonly variables are non-static by default and as non-static it requires an object instead.

**namespace** *ReadOnlyDemo*

**{**

**class** ReadOnlyExample

**{**

**public** **readonly** **int** number = 5;

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

ReadOnlyExample readOnlyInstance = new ReadOnlyExample**()**;

Console.WriteLine**(**readOnlyInstance.number**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

In the above example, the read-only variable is assigned with a value at the time of its declaration and is accessed using the instance of the class rather than using the class name as read-only variables are non-static in nature. Suppose, you may have another instance of the class, which might have the read-only number variable assigned to a different value based on some conditions. Can I do it? Yes, because the read-only variables are known at runtime.

##### ****Example: Read-only Variable Initialization through Constructor in C#****

In the below example, we are initializing the readonly variable through the class constructors. You can directly initialize the readonly variables at the time of declaration or you can initialize through class constructors. Once initialized then you cannot change the value of readonly variables in C#.

**namespace** *ReadOnlyDemo*

**{**

**class** ReadOnlyExample

**{**

//You can initialize at the time of declaration

**public** **readonly** **int** number = 5;

//You can also initialize through constructor

**public** ReadOnlyExample**()**

**{**

number = 20;

**}**

**public** ReadOnlyExample**(bool** IsAnotherInstance**)**

**{**

number = 100;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

ReadOnlyExample readOnlyInstance = new ReadOnlyExample**()**;

Console.WriteLine**(**readOnlyInstance.number**)**;

// You cannot change the value of a readonly variable once it is initialized

// readOnlyInstance.number = 20;

ReadOnlyExample readOnlyAnotherInstance = new ReadOnlyExample**(true)**;

Console.WriteLine**(**readOnlyAnotherInstance.number**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

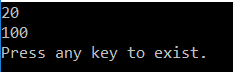
Console.ReadLine**()**;

**}**

**}**

**}**

###### **Output:**



As you can see in the above output, different values coming out of the program’s output for the two different instances of the class. Hence it proves that read-only variables are also immutable values that are known at runtime and do not change their values for the life of the program.

##### ****Let us understand Const, Readonly, Static and non-static variables with one example:****

**namespace** *ReadOnlyConstDemo*

**{**

**class** Example

**{**

**int** x; //Non-static variable

**static** **int** y = 200; //Static Variable

const **float** PI = 3.14f; //Const Variable

**readonly** **bool** flag; //Readonly Variable

**public** Example**(int** x, **bool** flag**)**

**{**

this.x = x;

this.flag = flag;

**}**

**static** **void** Main**(**string**[]** args**)**

**{**

Console.WriteLine**(**Example.y**)**;

Console.WriteLine**(**Example.PI**)**;

Example obj1 = new Example**(**50, **true)**;

Example obj2 = new Example**(**100, **false)**;

Console.WriteLine**(**obj1.x + " " + obj1.x**)**;

Console.WriteLine**(**obj2.flag + " " + obj2.flag**)**;

Console.WriteLine**(**"Press any key to exist."**)**;

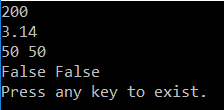
Console.ReadLine**()**;

**}**

**}**

**}**

###### **OUTPUT:**



# Properties in C#

## ****Properties in C# with Examples****

In this article, I am going to discuss the **Properties in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**Const and Read-Only Variables**](https://dotnettutorials.net/lesson/const-and-read-only-csharp/) in C#. As part of this article, we are going to discuss the following pointers related to properties in detail.

1. **Why do we need properties in C#?**
2. **What is a Property in C#?**
3. **What are Accessors in C#?**
4. **What is Set Accessor?**
5. **What is Get Accessor?**
6. **What are the different types of properties supported by C#.NET?**
7. **What is Read-only Property?**
8. **What is Write only property?**
9. **What is Read Write property?**
10. **What are the advantages of using Properties in C#?**
11. **What is the default accessibility modifier of Accessors in C#?**
12. **What are symmetric and asymmetric accessors in C#?**
13. **What are Auto-Implemented Properties in C#?**
14. **Why do we need Properties in real-time applications with an example?**

##### ****Why do we need Properties in C#?****

In order to encapsulate and protect the data members (i.e. fields), we use properties in C#. The Properties in C# are used as a mechanism to set and get the values of a class outside of that class. If a class contains any value in it and if we want to access those values outside of that class, then you can provide access to those values in 2 different ways

1. By storing the value under a public variable we can give access to the value outside of the class.
2. By storing that value in a private variable we can also give access to that value outside of the class by defining a property for that variable,

##### ****What is a Property in C#?****

A property in C# is a member of a class which is used to set and get the data from a data field of a class. The most important point that you need to remember is. a property in C# is never used to store data, it just acts as an interface to transfer the data. We use the Properties as they are the public data members of a class, but they are actually special **methods**called **accessors**.

##### ****What are Accessors in C#?****

The Assessors are nothing but special methods which are used to set and get the values from the underlying data member. Assessors are of two types such as

1. **set accessor**
2. **get accessor**

##### ****What is set Accessor?****

The **set** accessor is used to set the data (i.e. value) into a data field. This set accessor contains a fixed variable named “**value**”. Whenever we call the property to set the data, whatever data (value) we are supplying that will come and store in the variable “**value**” by default.

**Syntax: set { Data Field Name = value; }**

##### ****What is Get Accessor?****

The get accessor is used to get the data from the data field. Using this get accessor you cannot set the data.

**Syntax: get { return Data Field Name; }**

##### ****What are the different types of properties supported by C#.NET?****

The C#.NET supports four types of properties. They are as follows

1. **Read-only property**
2. **Write only property**
3. **Read Write property**
4. **Auto-implemented property**

Let us understand each of the above properties in details with examples.

##### ****What is Read-only Property in C#?****

The Read-only property is used to read the data from the data field. Using this property you cannot set the data into the data field. This property will contain only one accessor i.e. “get” accessor.

**Syntax: AccessModifier Datatype PropertyName { get { return DataFieldName; } }**

##### ****What is Write only Property in C#?****

The write-only property is used to write the data into the data field of a class. Using this property you cannot read the data from the data field. This property will contain only one accessor i.e. set accessor.

**Syntax: AccessModifier Datatype PropertyName {  set { DataFieldName = value; } }**

##### ****What is Read Write Property in C#?****

The Read-Write property is used for both read the data from the data field as well as write the data into the data field. This property will contain two accessor i.e. set and get.

**Syntax:**

AccessModifier DataType PropertyName

**{**

**set** **{** DataFieldName = **value**; **}**

**get** **{** Return DataFieldName; **}**

**}**

**NOTE:** Whenever we create a property, the data type of the property must be the same as the data type of the data field for which we create the property. A property can never accept any arguments.

###### **Let’s see an example to understand the Read and Write property.**

**namespace** *PropertyDemo*

**{**

**public** **class** Example

**{**

**private** **int** \_empid, \_eage;

**private** string \_ename, \_eaddress;

**public** **int** empid

**{**

**set**

**{**

\_empid = **value**;

**}**

**get**

**{**

**return** \_empid;

**}**

**}**

**public** **int** eage

**{**

**set**

**{**

\_eage = **value**;

**}**

**get**

**{**

**return** \_eage;

**}**

**}**

**public** string ename

**{**

**set**

**{**

\_ename = **value**;

**}**

**get**

**{**

**return** \_ename;

**}**

**}**

**public** string eaddress

**{**

**set**

**{**

\_eaddress = **value**;

**}**

**get**

**{**

**return** \_eaddress;

**}**

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Example obj1 = new Example**()**;

obj1.empid = 101;

obj1.ename = "pranaya";

obj1.eage = 27;

obj1.eaddress = "bbsr";

Console.WriteLine**(**"Employee details are:"**)**;

Console.WriteLine**(**"employee id:" + obj1.empid**)**;

Console.WriteLine**(**"employee name:" + obj1.ename**)**;

Console.WriteLine**(**"employee age:" + obj1.eage**)**;

Console.WriteLine**(**"employee address:" + obj1.eaddress**)**;

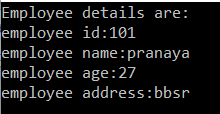
Console.ReadKey**()**;

**}**

**}**

**}**

**OUTPUT:**



In the above example, we declare the data fields of class Example as private. As a result, these data fields are not accessible directly from outside the class Example. So, here from the Program class, we transferred the data into the data field with the help of properties. As we are providing the abstraction to the data fields, this is known as data abstraction. So properties will provide data abstraction.

##### ****Let us see an example to understand the Read-Only and Write-Only Properties in C#.****

**namespace** *PropertyDemo*

**{**

**public** **class** Example

**{**

**int** num1, num2, result;

**public** **int** setnum1

**{**

**set**

**{**

num1 = **value**;

**}**

**}**

**public** **int** setnum2

**{**

**set**

**{**

num2 = **value**;

**}**

**}**

**public** **int** getresult

**{**

**get**

**{**

**return** result;

**}**

**}**

**public** **void** **add()**

**{**

result = num1 + num2;

**}**

**public** **void** sub**()**

**{**

result = num1 - num2;

**}**

**public** **void** mul**()**

**{**

result = num1 \* num2;

**}**

**public** **void** div**()**

**{**

result = num1 / num2;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Example obj1 = new Example**()**;

Console.WriteLine**(**"ENTER ANY TWO NUMBERS"**)**;

obj1.setnum1 = **int**.Parse**(**Console.ReadLine**())**;

obj1.setnum2 = **int**.Parse**(**Console.ReadLine**())**;

obj1.add**()**;

Console.WriteLine**(**"the sum is:" + obj1.getresult**)**;

obj1.sub**()**;

Console.WriteLine**(**"the sub is:" + obj1.getresult**)**;

obj1.mul**()**;

Console.WriteLine**(**"the mul is:" + obj1.getresult**)**;

obj1.div**()**;

Console.WriteLine**(**"the div is:" + obj1.getresult**)**;

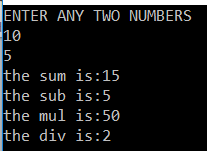
Console.ReadKey**()**;

**}**

**}**

**}**

**OUTPUT:**



##### ****What are the advantages of using Properties in C#?****

1. Properties will provide the abstraction to the data fields.
2. They also provide security to the data fields.
3. Properties can also validate the data before storing into the data fields.

##### ****What is the default accessibility modifier of Accessors in C#?****

The default accessibility modifier of the accessor is same as the accessibility modifier of property.

**For example:**

**public** **int** empid

**{**

**set** **{** \_empid = **value**; **}**

**get** **{** **return** \_empid; **}**

**}**

In the above example, property empid is declared as public. So, the set and get accessor will be public. If the property is private then the set and get will be private.

##### ****What are symmetric and asymmetric accessors in C#?****

If the accessibility modifier of the accessors (both get and set) are the same within a property then the accessors are known as symmetric accessors. On the other hand, if the accessibility modifier of the accessors is not the same within a property then the accessors are known as asymmetric accessors.

**For example:**

**public** **int** empid

**{**

**protected** **set** **{** \_empid = **value**; **}**

**get** **{** **return** \_empid; **}**

**}**

In the above example, the set accessor is declared as protected while the get is public, so they are known as asymmetric. In general asymmetric accessors are used in the inheritance process.

**We can also write the Read-only property using two accessors like**

**public** **int** empid

**{**

**private** **set** **{** \_empid = **value**; **}**

**get** **{** **return** \_empid; **}**

**}**

###### **We can also write the Write only property using two accessors like**

**public** **int** empid

**{**

**set** **{** \_empid = **value**; **}**

**private** **get** **{** **return** \_empid; **}**

**}**

##### ****What are Auto-Implemented Properties in C#?****

If you do not have any additional logic while setting and getting the data from a data field then you can make use of the auto-implemented properties which was introduced in C# 3.0

The Auto-implemented property reduces the amount of code that we have to write. When we use auto-implemented properties, the C# compiler implicitly creates a **private**, **anonymous field** behind the scene which is going to hold the data.

**Syntax: Access specifier Datatype Property name { get; set; }**

**Example: public int A { Get; Set; }**

##### ****Let’s see an example to understand auto implemented properties in C#.****

**namespace** *PropertyDemo*

**{**

**class** Test

**{**

**public** **int** A **{get**; **set**;**}**

**public** **int** B **{** **get**; **set**; **}**

**public** **int** Add**()**

**{**

**return** A + B;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Test obj = new Test**()**;

obj.A = 100;

obj.B = 200;

Console.WriteLine**(**obj.Add**())**;

Console.ReadKey**()**;

**}**

**}**

**}**

**OUTPUT: 300**

##### ****Why do we need properties in real-time applications?****

Declaring the class fields as public and exposing those fields to the outside class is bad as we do not have any control over what gets assigned and returned.

###### **Let’s understand this with one example.**

**namespace** *PropertyDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID;

**public** string Name;

**public** **int** PassMark;

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Student s = new Student**()**;

s.ID = -100;

s.Name = **null**;

s.PassMark = 0;

Console.WriteLine**(**"ID = {0} && Name = {1} && PassMark = {2}", s.ID, s.Name, s.PassMark**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**OUTPUT:**

properties in real-time applications

##### ****Problems with the above public fields are as follows****

1. An ID value should always be a non-negative number.
2. The name cannot be set to NULL.
3. If a student name is missing then we should return “No Name”.
4. The PassMark value should always be read-only.

Programming languages that do not have properties use getter and setter methods to encapsulate and protect fields.

###### **Let’s rewrite the same program using setter and getter methods to achieve the above requirements.**

**namespace** *PropertyDemo*

**{**

**public** **class** Student

**{**

**private** **int** \_iD;

**private** string \_name;

**private** **int** \_passMark = 35;

**public** **void** SetID**(int** ID**)**

**{**

**if** **(**ID **<** 0**)**

**{**

**throw** new Exception**(**"ID value should be greater than zero"**)**;

**}**

this.\_iD = ID;

**}**

**public** **int** GetID**()**

**{**

**return** this.\_iD;

**}**

**public** **void** SetName**(**string Name**)**

**{**

**if** **(**string.IsNullOrEmpty**(**Name**))**

**{**

**throw** new Exception**(**"Name should not be empty"**)**;

**}**

this.\_name = Name;

**}**

**public** string GetName**()**

**{**

**if** **(**string.IsNullOrEmpty**(**\_name**))**

**{**

**return** "No Name";

**}**

**return** this.\_name;

**}**

**public** **int** GetPassMark**()**

**{**

**return** this.\_passMark;

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Student S = new Student**()**;

S.SetID**(**101**)**;

S.SetName**(**"Pranaya"**)**;

Console.WriteLine**(**"Student ID = {0}", S.GetID**())**;

Console.WriteLine**(**"Student Name = {0}", S.GetName**())**;

Console.WriteLine**(**"Studenr Pass Mark = {0}", S.GetPassMark**())**;

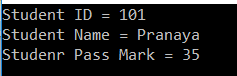
Console.ReadKey**()**;

**}**

**}**

**}**

**OUTPUT:**



**Note:** The advantage of properties over traditional getter() and setter() method is that we can access them as they are public fields.

###### **Let’s rewrite the same program using properties to achieve the above requirements.**

**namespace** *PropertyDemo*

**{**

**public** **class** Student

**{**

**private** **int** \_id;

**private** string \_name;

**private** **int** \_passMark = 35;

**public** **int** ID

**{**

**set**

**{**

**if** **(value** **<** 0**)**

**{**

**throw** new Exception**(**"ID value should be greater than zero"**)**;

**}**

this.\_id = **value**;

**}**

**get**

**{**

**return** this.\_id;

**}**

**}**

**public** string Name

**{**

**set**

**{**

**if** **(**string.IsNullOrEmpty**(value))**

**{**

**throw** new Exception**(**"Name should not be empty"**)**;

**}**

this.\_name = **value**;

**}**

**get**

**{**

**return** string.IsNullOrEmpty**(**this.\_name**)** ? "No Name" : this.\_name;

**}**

**}**

**public** **int** PassMark

**{**

**get**

**{**

**return** this.\_passMark;

**}**

**}**

**}**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Student S = new Student**()**;

S.ID = 101;

S.Name = "Pranaya";

Console.WriteLine**(**"Student ID = {0}", S.ID**)**;

Console.WriteLine**(**"Student Name = {0}", S.Name**)**;

Console.WriteLine**(**"Studenr Pass Mark = {0}", S.PassMark**)**;

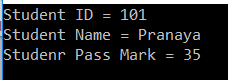
Console.ReadKey**()**;

**}**

**}**

**}**

**OUTPUT:**



**Why we should override the ToString method in C#**

**Why we should override the ToString method in C#**

In this article, I am going to discuss **why we should override the ToString method in C#** with an example and also we will discuss **how to override the ToString()** method. Please read our previous article where we discussed the [**Properties in C#**](https://dotnettutorials.net/lesson/properties-csharp/) with examples. As part of this article, we are going to discuss the following two pointers.

1. **Understanding the Object class.**
2. **Why we should override the ToString Methos?**
3. **How to Override the tostring method in C#?**

**Understanding the Object class.**

The **System.Object** class is the Superclass of all dot net types. That means, all the types in .NET Framework are inherited directly or indirectly from the **Object** class. Because of this inheritance, every type in .NET inherits the **ToString()** method from the **Object**class.

**Consider the following example.**

**namespace** *UnderstandingObjectClassMethods*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

**int** Number = 100;

Console.WriteLine**(**Number.ToString**())**;

**}**

**}**

**}**

In the above example the **Number.ToString()** method will give you the string representation of the integer 100. If you have a complex type let say Employee class as shown in the example below and when you call the **ToString()** method, then you will not get the output as expected. Hence we have to override the **ToString()** method, which is inherited from the **Object** class.

**namespace** *UnderstandingObjectClassMethods*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Employee emp = new Employee**()**;

emp.FirstName = "Pranaya";

emp.LastName = "Rout";

Console.WriteLine**(**emp.ToString**())**;

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** string FirstName;

**public** string LastName;

**}**

**}**

**When you run the above code it will give us the below output**

why we should override the ToString method in C#

Our requirement is when we call the **ToString()** method it should display the **First Name** and **Last Name** of the Employee. To achieve this we need to **override** the **ToString()** method which is provided by the Object class.

**Modifying the ToString() Method:**

Please modify the code as shown below to override the ToString() method.

**namespace** *UnderstandingObjectClassMethods*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Employee emp = new Employee**()**;

emp.FirstName = "Pranaya";

emp.LastName = "Rout";

Console.WriteLine**(**emp.ToString**())**;

Console.ReadKey**()**;

**}**

**}**

**public** **class** Employee

**{**

**public** string FirstName;

**public** string LastName;

**public** **override** string ToString**()**

**{**

**return** FirstName + ", " + LastName;

**}**

**}**

**}**

Now run the application and you will see the First Name and Last Name of the employee as expected as shown below.

why we should override the ToString method in C#

**Override Equals Method in C#**

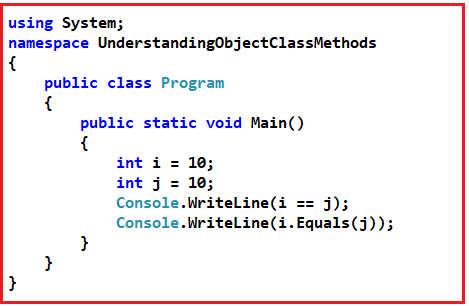
**Override Equals Method in C#**

In this article, I am going to discuss the **Why we need to Override Equals method in C#** with example. Please read our previous article before proceeding to this article where we discussed [**why and how to override the ToString() method**](https://dotnettutorials.net/lesson/why-we-should-override-the-tostring-method/) in C#. As part of this article, we are going to discuss the following pointers.

1. **Understanding the Equals method?**
2. **Understanding the difference between the “==” operator and the Equals() method in C#?**
3. **Why do we need to override the Equals() method in C#?**
4. **How we can override the Equals Method?**

**Understanding the difference between the “==” operator and the Equals() method in C#?**

In C#, as we already discussed every type directly or indirectly inherits from the **Object** class. So, the **Equals()** virtual method, which has a default implementation within the Object class is also available in every type via inheritance. In the following example, the variables **i** and **j** are integers. So, both the **==** and **Equals()** method returns true, since i and j, both variables have the same value i.e. 10.



In the following example, we compare 2 enums, and both the **==** operator and **Equals()** method returns **true** since both direction1 and direction2 enums have the same underlying integer value of 1.

**namespace** *UnderstandingObjectClassMethods*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Direction direction1 = Direction.East;

Direction direction2 = Direction.East;

Console.WriteLine**(**direction1 == direction2**)**;

Console.WriteLine**(**direction1.Equals**(**direction2**))**;

**}**

**}**

**public** enum Direction

**{**

East = 1,

West = 2,

North = 3,

South = 4

**}**

**}**

**Working with Reference Type in C#:**

If the type is a reference type, then by default the **“==”** operator checks for **reference equality** whereas the **Equals()** method checks for **value equality**. If this is not clear at the moment, don’t worry let us understand this with an example,

In the following example, C1 and C2 are 2 different object reference variables. But both are pointing to the same object. The most important point that you need to keep in mind is reference variables are different from objects. Reference variables are created on the stack and they point to the actual objects which are stored in the heap.

Since, C1 and C2 both refer to the same object, the reference equality, and the value equality is true. Value equality means that two objects contain the same values. In this example, the actual object is only one, so obviously, the values are also equal. If two objects have reference equality, then they also have value equality, but value equality does not guarantee reference equality.

**namespace** *UnderstandingObjectClassMethods*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Customer C1 = new Customer**()**;

C1.FirstName = "Pranaya";

C1.LastName = "Rout";

Customer C2 = C1;

Console.WriteLine**(**C1 == C2**)**;

Console.WriteLine**(**C1.Equals**(**C2**))**;

Console.ReadKey**()**;

**}**

**}**

**public** **class** Customer

**{**

**public** string FirstName **{** **get**; **set**; **}**

**public** string LastName **{** **get**; **set**; **}**

**}**

**}**

In the following example, the **==** operator returns **False**. This makes sense because C1 and C2 are referring to different objects. However, the **Equals()** method returns false, in spite of the values across C1 and C2 being the same. Hence, it makes sense to override, the **Equals()** method to return **true** when the values across the objects are the same.

**namespace** *UnderstandingObjectClassMethods*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Customer C1 = new Customer**()**;

C1.FirstName = "Pranaya";

C1.LastName = "Rout";

Customer C2 = new Customer**()**;

C2.FirstName = "Pranaya";

C2.LastName = "Rout";

Console.WriteLine**(**C1 == C2**)**;

Console.WriteLine**(**C1.Equals**(**C2**))**;

Console.ReadKey**()**;

**}**

**}**

**public** **class** Customer

**{**

**public** string FirstName **{** **get**; **set**; **}**

**public** string LastName **{** **get**; **set**; **}**

**}**

**}**

**Overriding the Equals Method in C#:**

In the following example, we override the Equals() method. When overriding the **Equals()** method, make sure the passed-in object is not null and can be cast to the type you are comparing. When overriding **Equals()**, you also need to override GetHashCode(), otherwise you get a compiler warning.

**namespace** *UnderstandingObjectClassMethods*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Customer C1 = new Customer**()**;

C1.FirstName = "Pranaya";

C1.LastName = "Rout";

Customer C2 = new Customer**()**;

C2.FirstName = "Pranaya";

C2.LastName = "Rout";

Console.WriteLine**(**C1 == C2**)**;

Console.WriteLine**(**C1.Equals**(**C2**))**;

**}**

**}**

**public** **class** Customer

**{**

**public** string FirstName **{** **get**; **set**; **}**

**public** string LastName **{** **get**; **set**; **}**

**public** **override** **bool** Equals**(object** obj**)**

**{**

// If the passed object is null

**if** **(**obj == **null)**

**{**

**return** **false**;

**}**

**if** **(**!**(**obj **is** Customer**))**

**{**

**return** **false**;

**}**

**return** **(**this.FirstName == **((**Customer**)**obj**)**.FirstName**)**

&& **(**this.LastName == **((**Customer**)**obj**)**.LastName**)**;

**}**

**public** **override** **int** GetHashCode**()**

**{**

**return** FirstName.GetHashCode**()** ^ LastName.GetHashCode**()**;

**}**

**}**

**}**

# Difference Between Convert.ToString and ToString Method in c#

## ****Difference Between Convert.ToString and ToString Method in c#****

In this article, I am going to discuss the **Difference Between Convert.ToString and ToString Method in C#** with Examples. Please read our previous article before proceeding to this article where we discussed why and how to override the [**Equals method in C#**](https://dotnettutorials.net/lesson/why-we-should-override-equals-method/) with examples.

##### ****Convert.ToString and ToString Method in C#****

Both these methods are used to convert a value to a string. The difference is **Convert.ToString()** method handles null whereas the **ToString()** doesn’t handle null in C#.

In C# if you declare a **string variable** and if you **don’t assign any value** to that variable, then by default that variable takes a **null** value. In such a case, if you use the **ToString()**method then your program will throw the **null reference exception.** On the other hand, if you use the **Convert.ToString()** method then your program will not throw an exception.

###### **Let us understand the Difference Between these two methods with an example.**

**namespace** *UnderstandingToStringMethod*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Customer C1 = **null**;

C1.ToString**()**;

Console.ReadLine**()**;

**}**

**}**

**public** **class** Customer

**{**

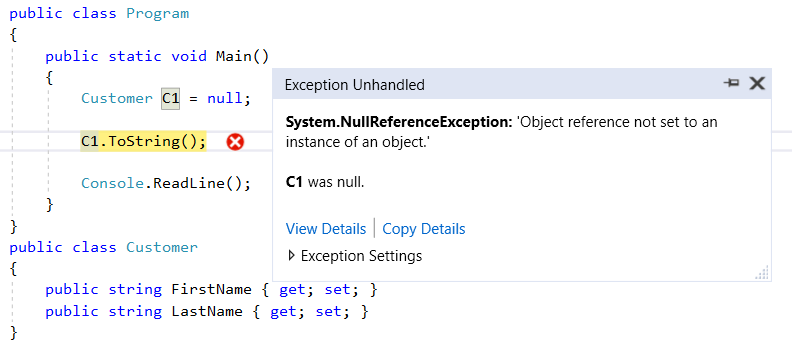
**public** string FirstName **{** **get**; **set**; **}**

**public** string LastName **{** **get**; **set**; **}**

**}**

**}**

**When you run the application, it will give you the following error**



This is because the **ToString()** method in C# expects the object to be not NULL on which it is invoking. In our example the object **C1 is Null** and we are invoking the **ToString()** on the **NULL object**, so it gives **NULL Reference exception**.

##### ****Let see another example.****

**namespace** *UnderstandingToStringMethod*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

String Name = **null**;

Name.ToString**()**;

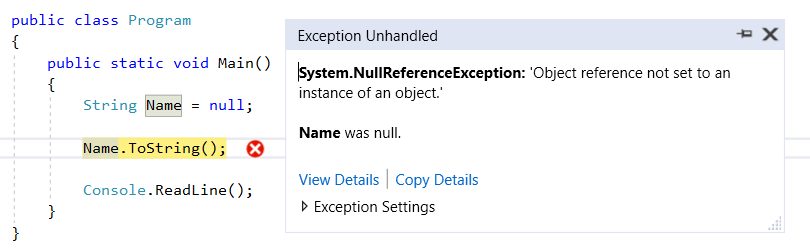
Console.ReadLine**()**;

**}**

**}**

**}**

When we execute the above program, it also gives us the same **NULL Reference exception** as shown below.



This is because the Name variable is Null and we are invoking the ToString() method. Let see what happens when we use the **Convert.Tostring()** method with the above two examples.

**namespace** *UnderstandingObjectClassMethods*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**()**

**{**

Customer C1 = **null**;

Convert.ToString**(**C1**)**;

String Name = **null**;

Convert.ToString**(**Name**)**;

Console.WriteLine**(**"No Error"**)**;

Console.ReadLine**()**;

**}**

**}**

**public** **class** Customer

**{**

**public** string FirstName **{** **get**; **set**; **}**

**public** string LastName **{** **get**; **set**; **}**

**}**

**}**

Now, with the above changes, run the application and it should be executed without any error. So in short, the **Convert.ToString()** method handles null, while the **ToString()** method doesn’t handle the Null and throws an exception. So it’s always a good programming practice to use the **Convert.ToString()** method will take care of the Null values and it is also safe to use.

# Checked and unchecked keyword in C#

## ****Checked and unchecked keyword in C#****

In this article, I am going to discuss the need and use of **Checked and unchecked keyword in C#** with examples. Please read our previous article, where we discussed the [**difference between Convert.ToString and ToString method**](https://dotnettutorials.net/lesson/difference-between-convert-tostring-and-tostring-csharp/) in C#. The C# provides checked and unchecked keyword which are used to handle integral type exceptions.

##### ****Why do we need Checked and unchecked keyword in C#?****

According to MSDN, The **checked keyword in C#** is used to explicitly enable overflow checking for integral-type arithmetic operations and conversions.

The **unchecked keyword in C#** is used to suppress overflow-checking for integral-type arithmetic operations and conversions.

Here, overflow checking means when the value of any integral-type exceeds its range, it does not raises any exception, instead it will give us unexpected or garbage results. If this is not clear at the moment, then don’t worry we will try to understand the above two points with examples.

##### ****Example:****

First, create a console application. Now, for demonstration purpose let’s take “**int**” data type and see what the maximum value it can hold. To do so, please modify the Program class as shown below.

**using** *System;*

**namespace** *CheckedUncheckedDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Console.WriteLine**(int**.MaxValue**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: 2147483647**

##### ****Example without Checked Keyword:****

Now, let’s see where the checked keyword can help us in making your code more useful. In the following example, you can see that we have three integer variables. The integer variables a and b holds the maximum value that an integer can hold. Then we just simply add the integer a and b and stored it in the third integer variable i.e. c.

**using** *System;*

**namespace** *CheckedUncheckedDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** a = 2147483647;

**int** b= 2147483647;

**int** c = a + b;

Console.WriteLine**(**c**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

Now run the application and see the output.

**Output: -2**

As you can see it display -2, but this is we were not expecting. What we except is some error (overflow) or exception. This is where the Checked keyword helps us to achieve and throws overflow exception.

##### ****Example: Using Checked Keyword****

The code example uses checked keyword. As we use the checked keyword, it should throws runtime exception rather than displaying -2.

**using** *System;*

**namespace** *CheckedUncheckedDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** a = 2147483647;

**int** b= 2147483647;

**int** c = **checked(**a + b**)**;

Console.WriteLine**(**c**)**;

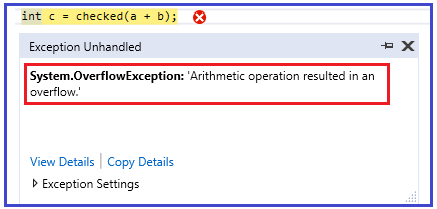
Console.ReadLine**()**;

**}**

**}**

**}**

Now, when you run the application, you should get the following OverflowException as expected.



In simple words, we can say that the checked keyword is used in scenarios where you want to ensure that your left hand side data type is not getting overflow

##### ****Unchecked keyword in C#:****

Let us understand the need and use of unchecked keyword in C#. The unchecked keyword behaves almost the same way as the default behavior of the compiler.

Let’s prove that the above point. So, modify the Program class as shown below and then see the output.

**using** *System;*

**namespace** *CheckedUncheckedDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** a = 2147483647;

**int** b= 2147483647;

**int** c = **unchecked(**a + b**)**;

Console.WriteLine**(**c**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

As shown in the above code, we have have just added the unchecked keyword in front of the arithmetic expression of the c variable. Now, run your application and you should get the following output..

###### **Output: -2**

So this proves that the unchecked keyword works almost the same way as the default compiler works. Now the question that should comes to your mind is, when the default compiler works same as unchecked keyword then what is the exact use of it.

Now, let’s see a simple example to understand the exact need of unchecked keyword in C#. Please modify the Program class as shown below.

**using** *System;*

**namespace** *CheckedUncheckedDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

const **int** a = 2147483647;

const **int** b= 2147483647;

**int** c = a + b;

Console.WriteLine**(**c**)**;

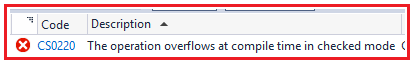
Console.ReadLine**()**;

**}**

**}**

**}**

As you can see in the above code that we have declared variable a and b as const int. now, when you try to compile the project, you should get the following compile time error.



If you want to bypass this behavior then you need to use the unchecked keyword in C#. Please modify the Program class as shown below which will help you to achieve this task.

**using** *System;*

**namespace** *CheckedUncheckedDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

const **int** a = 2147483647;

const **int** b= 2147483647;

**int** c = **unchecked(**a + b**)**;

Console.WriteLine**(**c**)**;

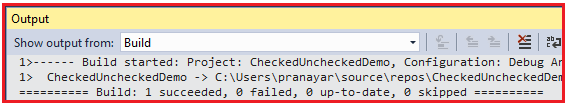
Console.ReadLine**()**;

**}**

**}**

**}**

Now, when you compile this code you will see that the compiler doesn’t throw any error like below diagram.



**Stack and Heap in C#**

**Stack and Heap in C# with Examples**

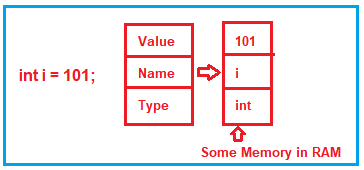
In this article, I am going to discuss **Stack and Heap in C# Application** with examples. Please read our previous article, where we discussed the [**Checked and unchecked keywords in C#**](https://dotnettutorials.net/lesson/checked-and-unchecked-keyword/). As part of this article, first, we will discuss what happens internally when we declare a variable of value types as well as reference types. Then we will move forward and learn two important concepts i.e. stack and heap memory as well as we will talk about value types and reference types.

**What happens internally when we declare a variable in .NET Application?**

When we declare a variable in a .NET application, it allocates some memory in the RAM. The memory which it allocates in RAM has three things are as follows:

1. **Name of the variable,**
2. **The data type of the variable, and**
3. **Value of the variable.**

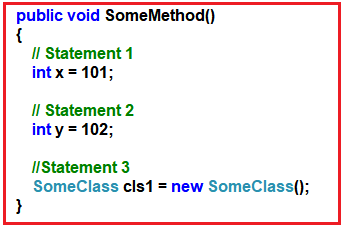
For better understanding, please have a look at the following image. Here, we declare a variable of type int and assign a value 101.



The above image shows a high-level overview of what happening in the memory. But depending on the data type (i.e. depending on the value type and reference type ), the memory may be allocated either in the stack or in the heap memory.

**Understanding Stack and Heap Memory in C#:**

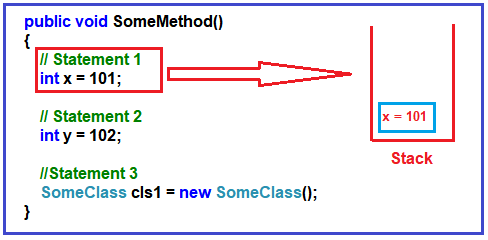
There are two types of memory allocation for the variables that we created in the .NET Application i.e. stack memory and heap memory. Let us understand the stack and heap memory with an example. In order to understand stack and heap, please have a look at the following code, and let’s understand what actually happens in the below code internally.



As you can see in the above image, the SomeMethod having three statements, let’s understand statement by statement how things are executed internally.

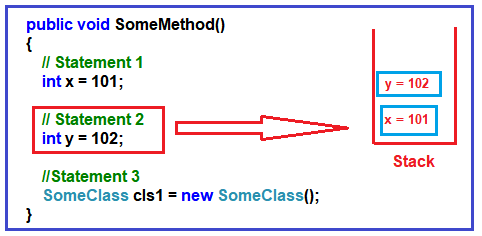
**Statement1:**

When the first statement is executed, the compiler allocates some memory in the stack. The stack memory is responsible for keeping track of the running memory needed in your application. For better understanding, please have a look at the following image.



**Statement2:**

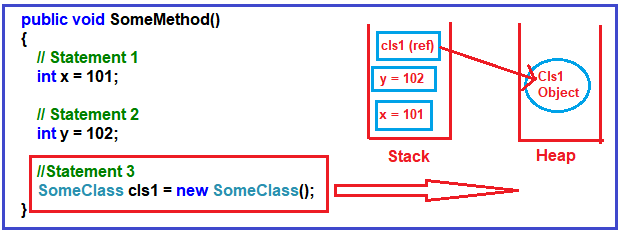
When the second statement is executed, it stacks this memory allocation (memory allocation for variable y) on top of the first memory allocation (memory allocation for variable x). You can think about the stack as a series of plates or dishes put on top of each other. Please have a look at the following diagram for a better understanding.



The Stack Memory allocation and de-allocation in .NET are done using the Last In First Out principle. In other words, we can say that the memory allocation and de-allocation are done only at one end of the memory, i.e., the top of the stack.

**Statement3:**

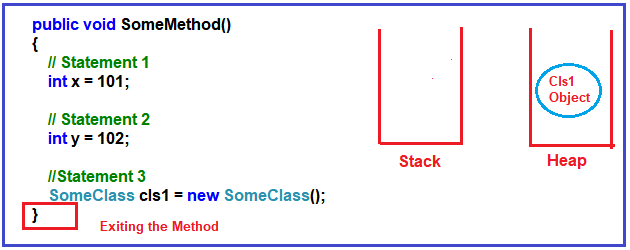
In the 3rd statement, we have created an object of SomeClass. When the 3rd statement is executed, it internally creates a pointer on the stack memory and the actual object is stored in a different memory location called Heap memory. The heap memory location does not track running memory. Heap is used for dynamic memory allocation. For better understanding please have a look at the below image.



Note: The reference pointers are allocated on the stack. The statement, **SomeClass cls1** does not allocate any memory for an instance of **SomeClass**, it only allocates a variable with the name cls1 in the stack and sets its value to null. The time it hits the new keyword, it allocates on the memory in the heap.

**What happens when the method completes its execution?**

When the three statements are executed, then the control will exist from the method. When it passes the end control i.e. the end curly brace “}”, it will clear all the memory variables which are created on the stack. It will de-allocate the memory in ‘LIFO’ fashion from the stack. For better understanding please have a look at the below image.

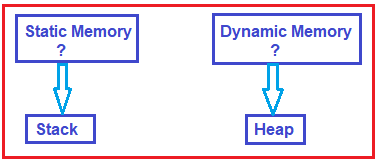


It will not de-allocate the heap memory. Later, the heap memory will be de-allocated by the garbage collector. Now you may have one question in your mind is why two types of memory, can’t we just allocate everything on just one memory type?

**Why do we have two types of memory?**

As we know, in C#, the primitive data types such as int, double, bool, etc. they just hold a single value. On the other hand, the reference data types or object data types are complex i.e. an object data type or reference data type can have reference to other objects as well as other primitive data types.

So, the reference data type holds references to other multiple values, and each one of them must be stored in memory. Object types need dynamic memory while primitive data types need static memory. Please have a look at the following image for a better understanding.

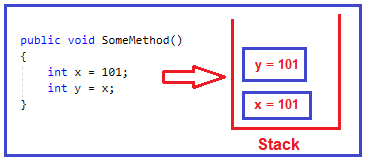


**Value types and reference types in .NET**

As we understood the concept of Stack and Heap, Now, let us move forward and understand the concept value types and reference types in detail. The Value types are the types that hold both data and memory in the same location. On the other hand, a reference type is a type that has a pointer that points to the actual memory location.

**Understanding Value Type:**

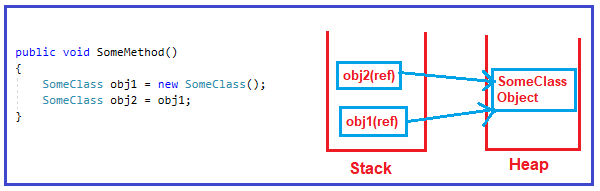
Let us understand value type with an example. Please have a look at the following image. As you can see in the image, first we create an integer variable with the name x and then we assign this x integer value to another integer variable whose name is y. In both these memory values are allocated on the stack.



In .NET, when we assign one integer variable value to another integer variable, then it creates a completely different copy in the stack memory that’s what you can see in the above image. So, if you change one variable value, then the other variable will not be affected. In .NET these kinds of data types are called ‘Value types’. So, bool, byte, char, decimal, double, enum, float, long, sbyte, int, short, ulong, struct, uint, ushort are examples of value types.

**Understanding Reference Type in C#:**

Let us understand reference type with an example. Please have a look at the following image. Here, first, we create an object i.e. obj1) and then assign this object to another object i.e. obj2. In this case, both reference variables (obj1 and obj2) will point to the same memory location.



In this case, when you change one of them, the other object is also gets affected. These kinds of data types are termed as ‘Reference types’ in .NET. So, class, interface, object, string, and delegate are the example of Reference Types.

**How is the heap memory freed up?**

The memory allocation which is done on the stack is gone when the control moves out from the method i.e once the method completes its execution. On the other hand, the memory allocation which is done on the heap needs to be de-allocated by the garbage collector.

When an object stored on the heap is no longer use, that means the object does not have any reference pointing, then the object is eligible for garbage collection. At some point in time, the garbage collector will de-allocate this object from the heap.