freelance_Project available to buy contact on 8007592194

SR.NO	Project NAME	Technology
1	Online E-Learning Platform Hub	React+Springboot+MySql
2	PG Mates / RoomSharing / Flat Mates	React+Springboot+MySql
3	Tour and Travel management System	React+Springboot+MySql
4	Election commition of India (online Voting System)	React+Springboot+MySql
5	HomeRental Booking System	React+Springboot+MySql
6	Event Management System	React+Springboot+MySql
7	Hotel Management System	React+Springboot+MySql
8	Agriculture web Project	React+Springboot+MySql
9	AirLine Reservation System / Flight booking System	React+Springboot+MySql
10	E-commerce web Project	React+Springboot+MySql
11	Hospital Management System	React+Springboot+MySql
12	E-RTO Driving licence portal	React+Springboot+MySql
13	Transpotation Services portal	React+Springboot+MySql
14	Courier Services Portal / Courier Management System	React+Springboot+MySql
15	Online Food Delivery Portal	React+Springboot+MySql
16	Muncipal Corporation Management	React+Springboot+MySql
17	Gym Management System	React+Springboot+MySql
18	Bike/Car ental System Portal	React+Springboot+MySql
19	CharityDonation web project	React+Springboot+MySql
20	Movie Booking System	React+Springboot+MySql

freelance_Project available to buy contact on 8007592194

21	Job Portal web project	React+Springboot+MySql
22	LIC Insurance Portal	React+Springboot+MySql
23	Employee Management System	React+Springboot+MySql
24	Payroll Management System	React+Springboot+MySql
25	RealEstate Property Project	React+Springboot+MySql
26	Marriage Hall Booking Project	React+Springboot+MySql
27	Online Student Management portal	React+Springboot+MySql
28	Resturant management System	React+Springboot+MySql
29	Solar Management Project	React+Springboot+MySql
30	OneStepService LinkLabourContractor	React+Springboot+MySql
31	Vehical Service Center Portal	React+Springboot+MySql
32	E-wallet Banking Project	React+Springboot+MySql
33	Blogg Application Project	React+Springboot+MySql
34	Car Parking booking Project	React+Springboot+MySql
35	OLA Cab Booking Portal	React+Springboot+MySql
36	Society management Portal	React+Springboot+MySql
37	E-College Portal	React+Springboot+MySql
38	FoodWaste Management Donate System	React+Springboot+MySql
39	Sports Ground Booking	React+Springboot+MySql
40	BloodBank mangement System	React+Springboot+MySql
41	Bus Tickit Booking Project	React+Springboot+MySql
42	Fruite Delivery Project	React+Springboot+MySql
43	Woodworks Bed Shop	React+Springboot+MySql
44	Online Dairy Product sell Project	React+Springboot+MySql
45	Online E-Pharma medicine sell Project	React+Springboot+MySql
46	FarmerMarketplace Web Project	React+Springboot+MySql
47	Online Cloth Store Project	React+Springboot+MySql
48		React+Springboot+MySql
49		React+Springboot+MySql
50		React+Springboot+MySql

Introduction To C# (C-Sharp) Programming

- Microsoft introduced a new language called C# (pronounced C Sharp). C# is designed to be a simple, modern, general-purpose, object-oriented programming language, borrowing key concepts from several other languages, most notably Java.
- C# was developed by **Anders Hejlsberg** and his team during the development of **.Net Framework.**
- C# is a part of .NET Framework.
- Microsoft introduced C# as a new programming language to address the problems posed by traditional languages.

We can use C# for building variety of applications

- **WINDOWS APPLICATION**: using console application or winform application.
- **MOBILE APPLICATIONS:** for phones such as Nokia Lumia (built-in support) but we can use a third party tool or library called "**XAMARIN**" to create mobile applications for ANDROID and IOS as well.
- **WEB APPLICATON:** using ASP.NET web forms or ASP.NET MVC.
- **GAMING APPLICATION:** Unity.
- C# could theoretically be compiled to machine code, but in real life, it's always used in combination with the .NET framework. Therefore, applications written in C#, requires the .NET framework to be installed on the computer running the application. While the .NET framework makes it possible to use a wide range of languages, C# is sometimes referred to as THE .NET language, perhaps because it was designed together with the framework.
- **Language interoperability** is the ability of code to interact with code that is written by using a different programming language.

What is .Net Framework ?

- .NET is a programming framework created by Microsoft that developers can use to create applications more easily. A framework is just a bunch of code that the programmer can call without having to write it explicitly.
- It provides a controlled programming environment where software can be developed, installed and executed on Windows-based operating systems.
- It is basically a collection of libraries.
- Is a programming platform that is used for developing Windows, Web-based, and mobile software.
- It has a number of pre-coded solutions that manage the execution of programs written specifically for the framework.

- A programmer can develop applications using one of the languages supported by .NET.
- Microsoft introduced C# as a new programming language to address the problems posed by traditional languages.
- The .NET Framework is an infrastructure that is used to build, deploy, and run different types of applications and services using .NET technologies.
- The .NET Framework is an infrastructure that is used to Minimize software development, deployment, and versioning conflicts.

Microsoft C# Was Developed To

- Create a very simple and yet powerful tool for building interoperable and robust applications.
- Create a complete object-oriented architecture.
- Support powerful component-oriented development.
- Allow access to many features previously available only in C++ while retaining the ease-of-use of a rapid application development tool such as Visual Basic.
- Provide familiarity to programmers coming from C or C++ background.
- Allow to write applications that target both desktop and mobile devices.
- C# has features common to most object-oriented languages.
- Provide consistent object-oriented programming environment.
- Minimize software deployment and versioning conflicts by providing a code-execution environment.
- Promote safe execution of code by providing a code-execution environment.
- Provide a consistent developer experience across varying types of applications such as Windows-based applications and Web-based applications.
- It has language-specific features, such as:

o Type safety checking
o Generics
o Indexers

- These features make the C# as a preferred language to create a wide variety of applications.
- C# is a programming language designed for building a wide range of applications that run on the .NET Framework.

Purpose of C# Language

- Microsoft .NET was formerly known as Next Generation Windows Services (NGWS).

- It is a completely new platform for developing the next generation of Windows/Web applications.
- These applications transcend device boundaries and fully harness the power of the Internet.
- However, building the new platform required a language that could take full advantage.
- This is one of the factors that led to the development of C#.
- C# is an object-oriented language derived from C and C++.
- The motive of C# is to provide a simple, efficient, productive, and object-oriented language that is familiar and yet at the same time revolutionary.

Features of C#

- C# is a programming language designed for building a wide range of applications that run on the .NET Framework.
- Following are some basic key features of C#:
    - Type-safety Checking
    - Object-oriented Programming
    - Garbage Collection
    - Standardization by European Computer Manufacturers Association (ECMA)
    - Generic Types and Methods
    - Iterators
    - Methods with named Arguments
    - Methods with optional Arguments
    - Static Classes
    - Nullable Types
    - Auto-implemented Properties
    - Accessor Accessibility
    - Anonymous Methods
    - Parallel Computing
    - Auto-implemented Properties

- Partial Classes

- Object-oriented Programming: Focuses on objects so that code written once can be reused. This helps reduce time and effort on the part of developers.
- Type-safety Checking: Checked the overflow of types because uninitialized variables cannot be used in C# as C# is a case-sensitive language
- Garbage Collection: Performs automatic memory management from time to time and spares the programmer the task.
- Standardization by European Computer Manufacturers Association (ECMA): Specifies the syntax and constraints used to create standard C# programs.
- Generic Types and Methods: Are a type of data structure that contains code that remains the same throughout but the data type of the parameters can change with each use.
- Iterators: Enable looping (or iterations) on user-defined data types with the for each loop.
- Static Classes: Contain only static members and do not require instantiation.
- Partial Classes: Allow the user to split a single class into multiple source code (.cs) files.
- Anonymous Methods: Enable the user to specify a small block of code within the delegate declaration.
- Methods with named Arguments: Enable the user to associate a method argument with a name rather than its position in the argument list.
- Methods with optional Arguments: Allow the user to define a method with an optional argument with a default value.
- Nullable Types: Allow a variable to contain a value that is undefined.
- Accessor Accessibility: Allows the user to specify the accessibility levels of the get and set accessors.

- Auto-implemented Properties: Allow the user to create a property without explicitly providing the methods to get and set the value of the property.
- Parallel Computing: Support for parallel programming using which develop efficient, fine-grained, and scalable parallel code without working directly with threads or the thread pool.

There are Several Applications Of C#

C# is an object-oriented language that can be used in a number of applications.

- Gaming applications
- Web applications
- Mobile applications for pocket PCs, PDAs, and cell phones
- Web services
- Cloud applications
- Simple standalone desktop applications such as Library Management Systems, Student Mark Sheet generation, and so on
- Large-scale enterprise applications
- Complex distributed applications that can spread over a number of cities or countries

The CLR

- Is the foundation of the .NET Framework.
- Acts as an execution engine for the .NET Framework.
- Manages the execution of programs and provides a suitable environment for programs to run.
- Provides a multi-language execution environment.
- Is a backbone of .NET Framework
- Performs various functions such as:
    - Memory management
    - Code execution
    - Error handling
    - Code safety verification
    - Garbage collection

By Using DotNet Framework

- A programmer can develop applications using one of the languages supported by .NET.
- These applications make use of the base class libraries provided by the .NET Framework.
- The .NET Framework supports a number of development tools and language compilers in its Software Development Kit (SDK).

For Example

- To display a text message on the screen, the following command can be used:
- System.Console.WriteLine(".NET Architecture");
- The same WriteLine() method will be used across all .NET languages.
- This is done by making the Framework Class Library as a common class library for all .NET languages.

DotNet Framework Class Library (FCL)

- Is a comprehensive object-oriented collection of reusable types.
- Is used to develop applications ranging from traditional command-line to Graphical User Interface (GUI) applications that can be used on the Web.

DotNet Framework Components

- Common Language Specification (CLS)
- Common Type System (CTS)
- Base Framework Classes
- WPF
- Asp.Net
- WCF

- Ado.Net
- Entity Framework
- Task Parallel Library
- LINQ
- Parallel LINQ

The .NET Framework languages, such as C#, VB, and J# are statically typed languages.

CLS - Common Language Specification

- Is a set of rules that any .NET language should follow to create applications that are interoperable with other languages.

CTS - Common Type System

- Describes how data types are declared, used, and managed in the runtime and facilitates the use of types across various languages.

ADO.NET

- Provides classes to interact with databases.

ASP.Net

- Provides a set of classes to build Web applications. ASP.NET Web applications can be built using Web Forms, which is a set of classes to design forms for the Web pages similar to the HTML.
- Supports Web services that can be accessed using a standard set of protocols.

WCF

- Is a service-oriented messaging framework.
- Allows creating service endpoints and allows programs to asynchronously send and receive data from the service endpoint.

WPF

- Is a UI framework based on XML and vector graphics.
- Uses 3D computer graphics hardware and Direct3D technologies to create desktop applications with rich UI on the Windows platform.

LINQ

- Is a component that provides data querying capabilities to a .NET application.

Entity Framework

- Is a set of technologies built upon ADO.NET that enables creating data-centric applications in object-oriented manner.

Parallel LINQ

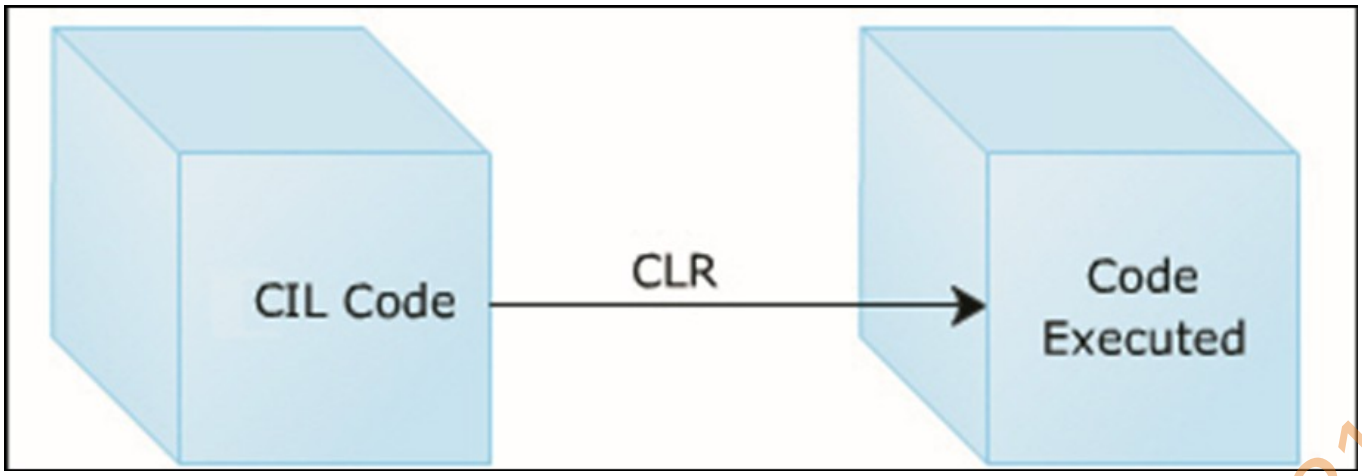
- Is a set of classes to support parallel programming using LINQ.

Task Parallel Library

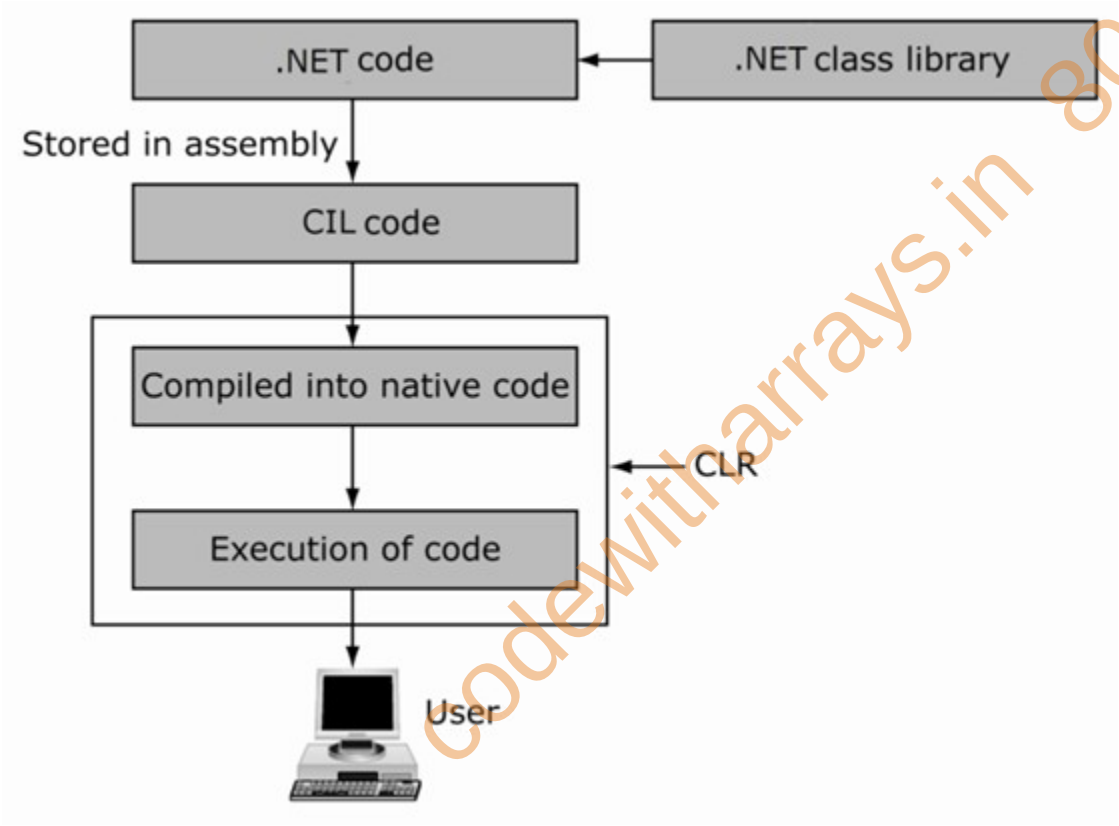
- Is a library that simplifies parallel and concurrent programming in a .NET application.

When a code is executed for the first time

- The CIL (COMMON INTERMEDIATE LANGUAGE) code is converted to a code native to the operating system.
- This is done at runtime by the Just-In-Time (JIT) compiler present in the CLR.
- The CLR converts the CIL code to the machine language code.
- Once this is done, the code can be directly executed by the CPU.



The following figure represents the process of conversion of CIL code to the native code:



The CLR provides many features such as:

- Memory management
- Code execution
- Error handling

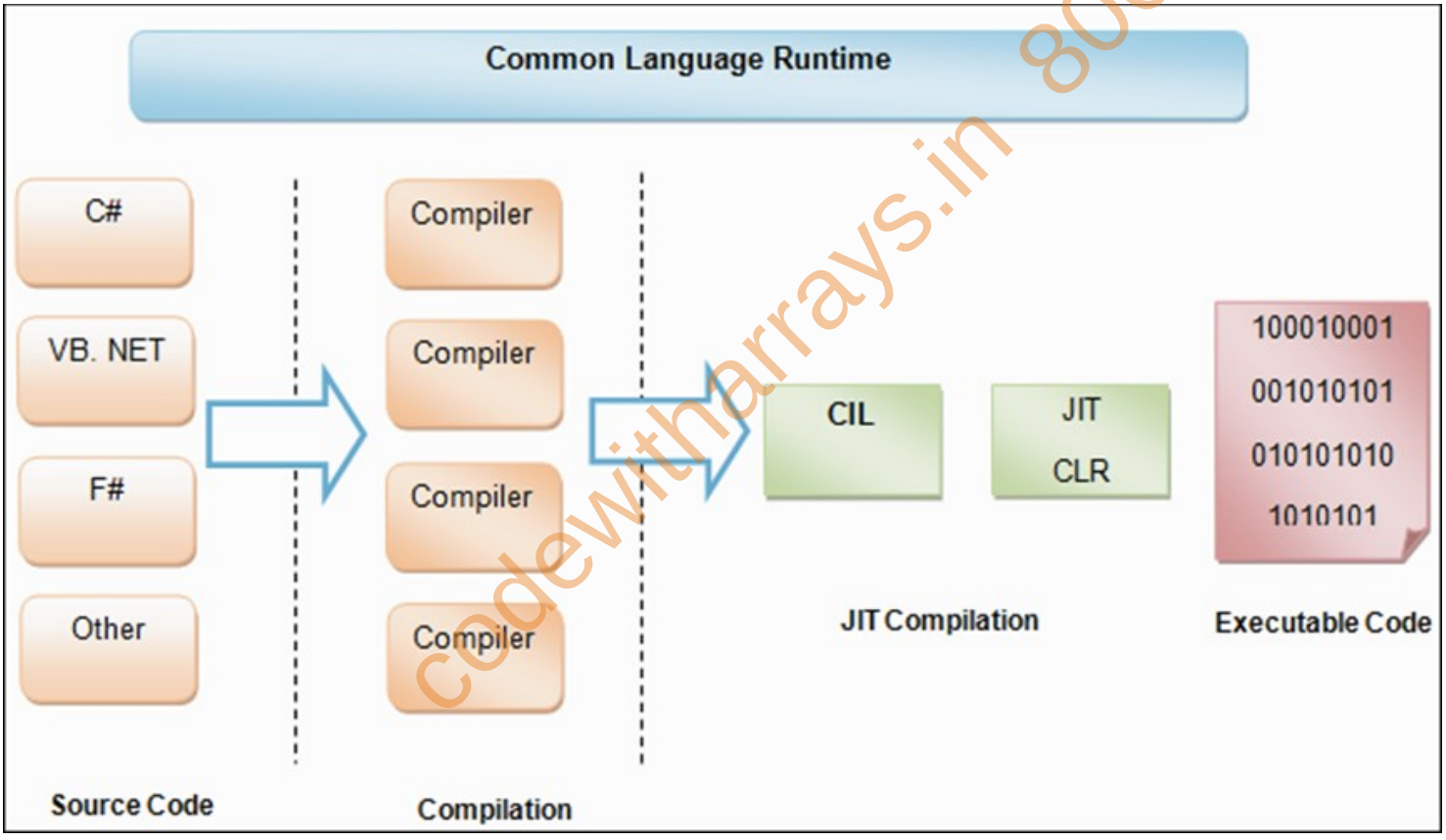
- Code safety verification
- Garbage collection

The applications that run under the CLR are called managed code.

The CLR manages code at execution time and performs operations such as:

- Thread management
- Remoting
- Memory management

Below figure shows a more detailed look at the working of the CLR:

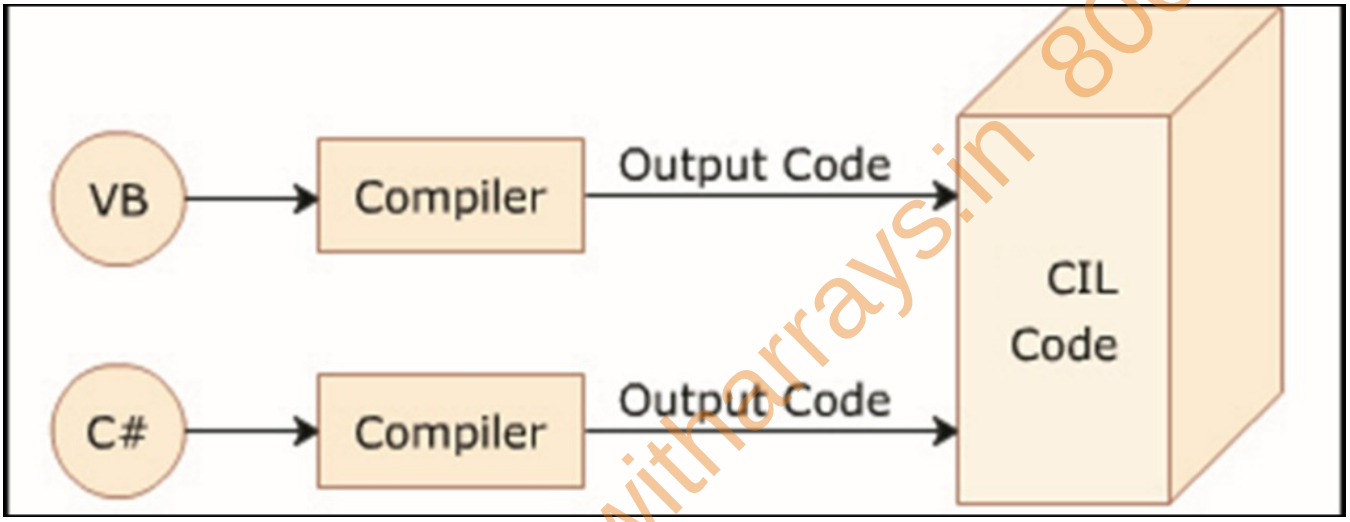


CIL - Common Intermediate Language

- Every .NET programming language generally has a compiler and a runtime environment of its own.

- The compiler converts the source code into executable code that can be run by the users.
- One of the primary goals of .NET Framework is to combine the runtime environments so that the developers can work with a single set of runtime services.
- When the code written in a .NET compatible language such as C# or VB is compiled, the output code is in the form of MSIL code.
- MSIL is composed of a specific set of instructions that indicate how the code should be executed.
- MSIL is now called as Common Intermediate Language (CIL).

The following figure depicts the concept of CIL:



What Is Memory Management ?

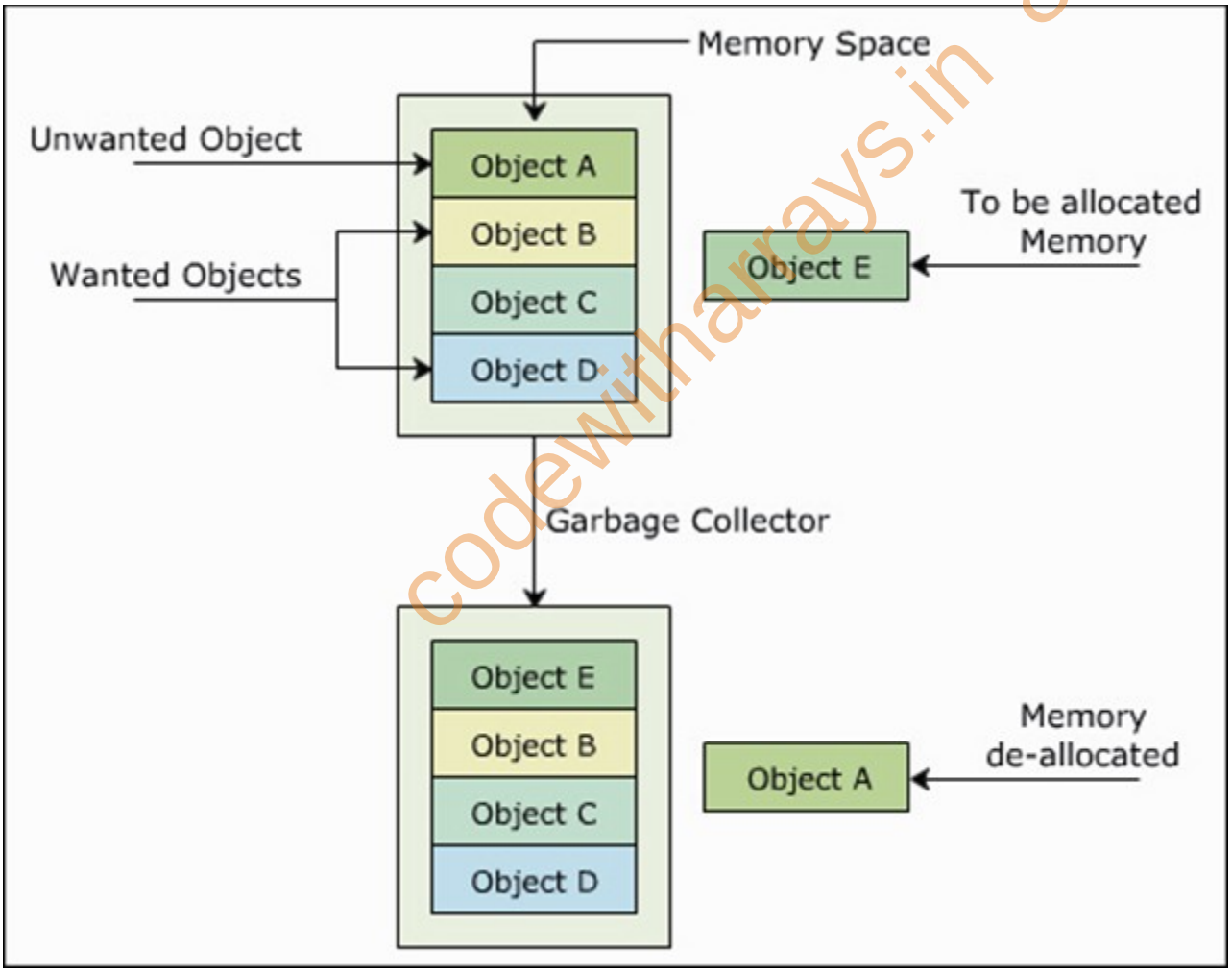
- In programming languages like C and C++, the allocation and de-allocation of memory is done manually.
- Performing these tasks manually is both, time-consuming and difficult.
- The C# language provides the feature of allocating and releasing memory using automatic memory management.

- This means that there is no need to write code to allocate memory when objects are created or to release memory when objects are not required in the application.
- Automatic memory management increases the code quality and enhances the performance and the productivity.

What Is Garbage Collection ?

- Garbage Collection is the process of automatic reclaiming of memory from objects that are no longer in scope.
- Garbage Collection helps the process of allocating and de-allocating memory using automatic memory management.

Below figure describes concept of garbage collection:



DLR - Dynamic Language Runtime

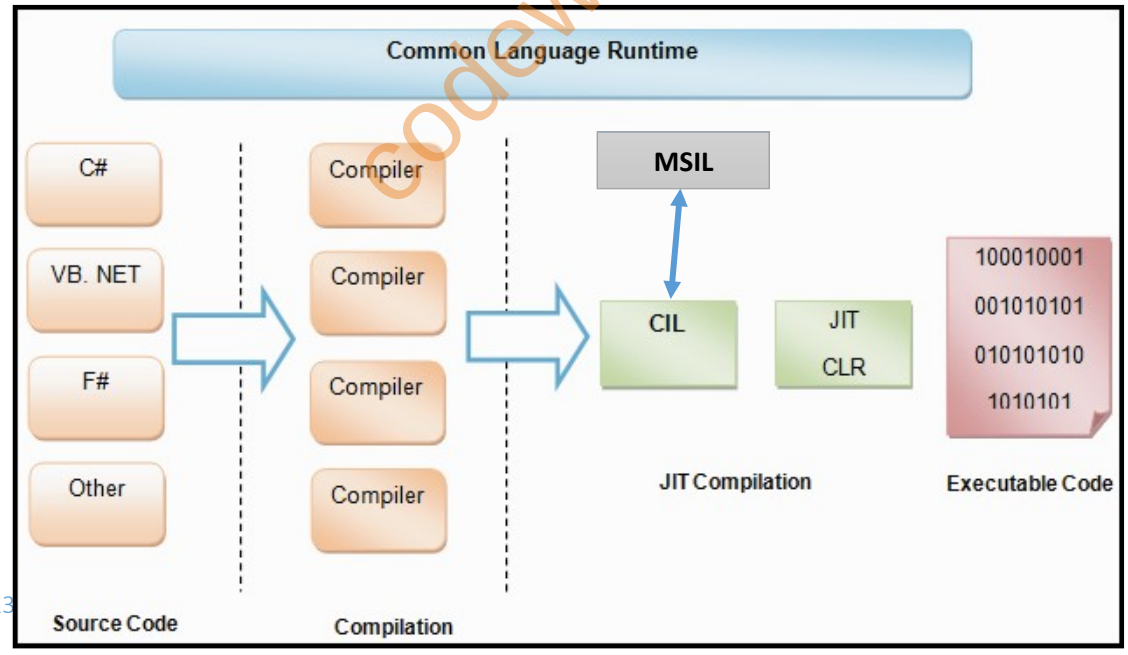
- Is a runtime environment built on top of the CLR to enable interoperability of dynamic languages such as Ruby and Python with the .NET Framework.
- Allows creating and porting dynamic languages to the .NET Framework.
- Provides dynamic features to the existing statically typed languages. For example, C# relies on the DLR to perform dynamic binding.

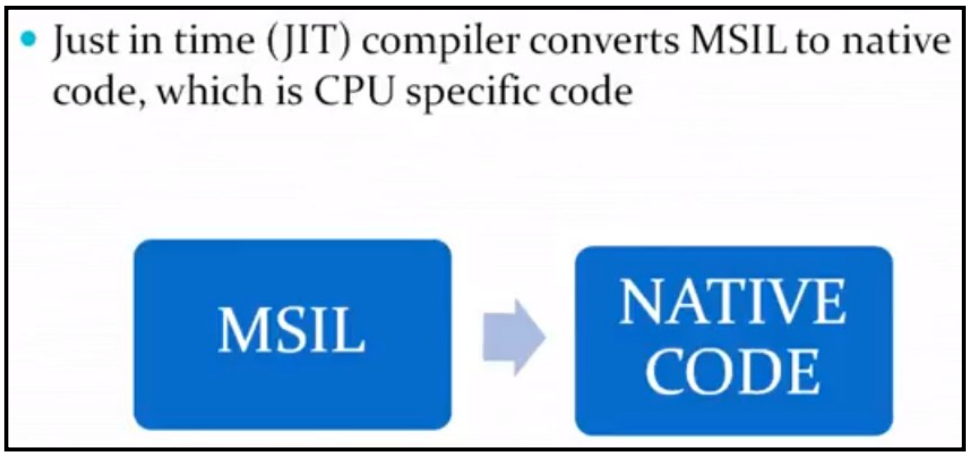
Common Language Runtime (Clr) In Dotnet Framework

→ **The CLR:**

- Is the foundation of the .NET Framework.
- Acts as an execution engine for the .NET Framework.
- Manages the execution of programs and provides a suitable environment for programs to run.
- Provides a multi-language execution environment.

The following figure shows a more detailed look at the working of the CLR:





When a code is executed for the first time;

→ The CIL (COMMON INTERMEDIATE LANGUAGE) code is converted to a code native to the operating system.

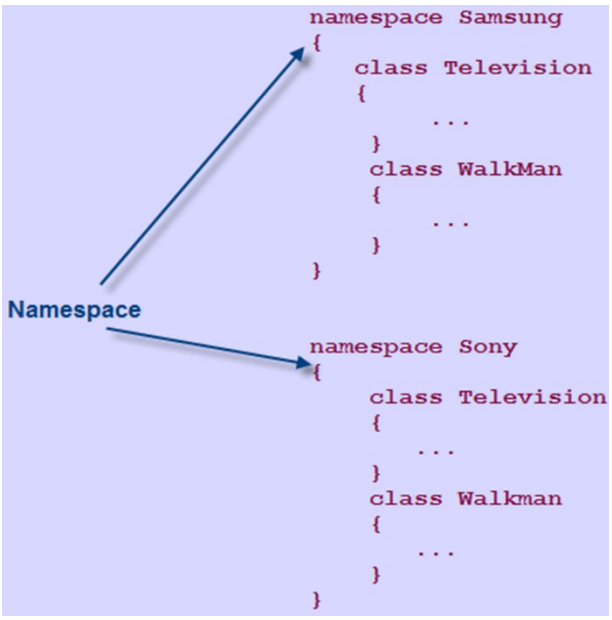
→ This is done at runtime by the Just-In-Time (JIT) compiler present in the CLR.

→ The CLR converts the CIL code to the machine language code.

Once this is done, the code can be directly executed by the CPU

What Is Namespace In C# ?

- A namespace is used in C# to group classes logically and prevent name clashes between classes with identical names.
- A namespace reduces any complexities when the same program is required in another application.
- A namespace is used to organize your code and is collection of classes, interfaces, structs, enums and delegates.
- **NOTE**: if u don't want to use namespace you can use **Fully Qualified Name (FQN)**.



Purpose Of Namespaces

Scenario

- Consider Venice, which is a city in the US as well as in Italy.
- You can easily distinguish between the two cities by associating them with their respective countries.
- Similarly, when working on a huge project, there may be situations where classes have identical names.
- This may result in name conflicts.
- This problem can be solved by having the individual modules of the project use separate namespaces to store their respective classes.
- By doing this, classes can have identical names without any resultant name clashes.

The following code renames identical classes by inserting a descriptive prefix:

```
class SamsungTelevision
{
...
}
```

```
class SamsungWalkMan
{
...
}
class SonyTelevision
{
...
}
class SonyWalkMan
{
...
}
```

In Above Code,

- The identical classes Television and WalkMan are prefixed with their respective company names to avoid any conflicts.
- There cannot be two classes with the same name.
- It is observed that the names of the classes get long and become difficult to maintain.

The following code demonstrates a solution to overcome this, by using namespaces:

```
namespace Samsung
{
class Television
{
...
}
class WalkMan
{
...
}
}
namespace Sony
{
class Television
{
...
}
class Walkman
{
...
}
}
```

In Above Code,

- Each of the identical classes is placed in their respective namespaces, which denote respective company names.
- It can be observed that this is a neater, better organized, and more structured way to handle naming conflicts.

Using Namespaces In C#

- C# allows you to specify a unique identifier for each namespace.
- This identifier helps you to access the classes within the namespace.
- Apart from classes, the following data structures can be declared in a namespace:

Interfaces

- An interface is a reference type that contains declarations of the events, indexers, methods, and properties.
- Interfaces are inherited by classes and structures and all the declarations are implemented in these classes and structures.

Structures

- A structure is a value type that can hold values of different data types.
- It can include fields, methods, constants, constructors, properties, indexers, operators, and other structures.

Enumeration

- An enumeration is a value type that consists of a list of named constants.
- This list of named constants is known as the enumerator list.

Delegate

- A delegate is a user-defined reference type that refers to one or more methods.

- It can be used to pass data as parameters to methods.

Advantages & Characteristics Of Namespaces

- A namespace groups common and related classes, structures, or interfaces, which support OOP concepts of encapsulation and abstraction.
- A namespace provides a hierarchical structure that helps to identify the logic for grouping the classes.
- A namespace allows you to add more classes, structures, enumerations, delegates, and interfaces once the namespace is declared.
- A namespace includes classes with names that are unique within the namespace.

A namespace provides the following benefits:

- A namespace allows you to use multiple classes with same names by creating them in different namespaces.
- It makes the system modular.
- The .NET Framework comprises several built-in namespaces that contain:
    - Classes
    - Interfaces
    - Structures
    - Delegates
    - Enumerations
- These namespaces are referred to as system-defined namespaces.
- The most commonly used built-in namespace of the .NET Framework is System.
- The System namespace contains classes that:
    - Define value and reference data types, interfaces, and other namespaces.

- Allow you to interact with the system, including the standard input and output devices.

Pre-defined Namespaces

Some of the most widely used namespaces within the System namespace are as follows:

System.Collections

- The System.Collections namespace contains classes and interfaces that define complex data structures such as lists, queues, bit arrays, hash tables, and dictionaries.

System.Data

- The System.Data namespace contains classes that make up the ADO.NET architecture.
- The ADO.NET architecture allows you to build components that can be used to insert, modify, and delete data from multiple data sources.

System.Diagnostics

- The System.Diagnostics namespace contains classes that are used to interact with the system processes.
- This namespace also provides classes that are used to debug applications and trace the execution of the code.

System.IO

- The System.IO namespace contains classes that enable you to read from and write to data streams and files.

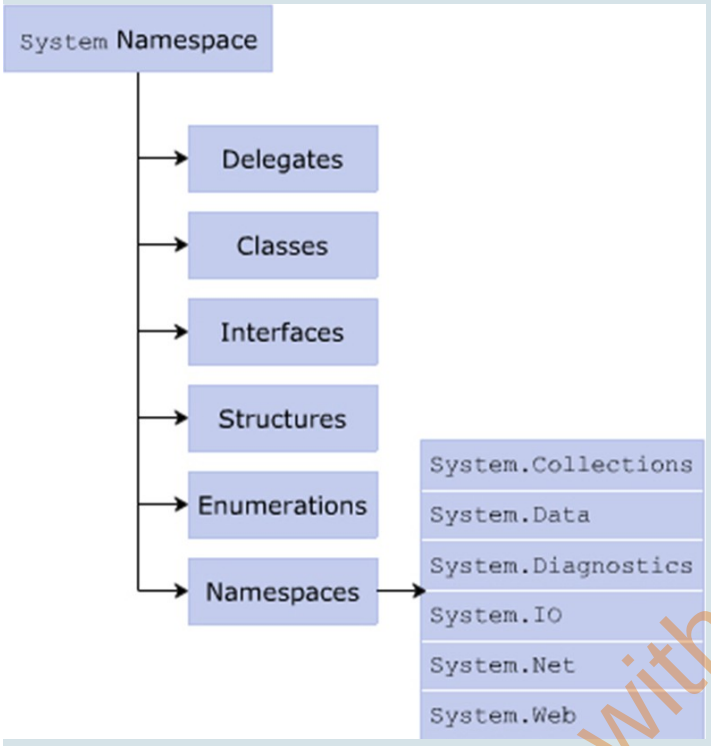
System.Net

- The System.Net namespace contains classes that allow you to create Web-based applications.

System.Web

- The System.Web namespace provides classes and interfaces that allow communication between the browser and the server.

The following figure displays some built-in namespaces:

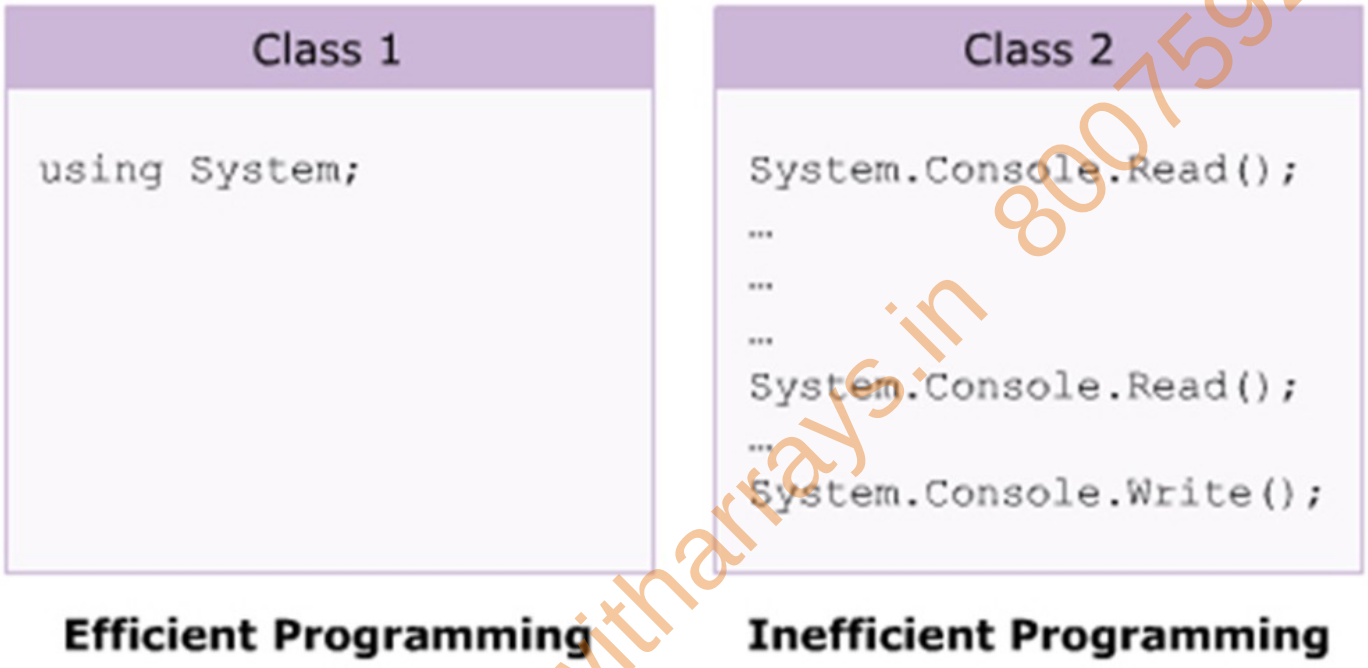


System Namespace in .Net Framework

- The System namespace is imported by default in the .NET Framework.
- It appears as the first line of the program along with the using keyword.
- For referring to classes within a built-in namespace, you need to explicitly refer to the required classes.

- It is done by specifying the namespace and the class name separated by the dot (.) operator after the using keyword at the beginning of the program.
- You can refer to classes within the namespaces in the same manner without the using keyword.
- However, this results in redundancy because you need to mention the whole declaration every time you refer to the class in the code.

The two approaches of referencing the System namespace are:



Though both are technically valid, the first approach is more recommended.

The following syntax is used to access a method in a system-defined namespace:

```
<NamespaceName>.<ClassName>.<MethodName>;
```

In Above Syntax,

- NamespaceName: Is the name of the namespace.

- ClassName: Is the name of the class that you want to access.
- MethodName: Is the name of the method within the class that is to be invoked.

The following syntax is used to access the system-defined namespaces with the using keyword:

```
using <NamespaceName>;
using <NamespaceName>.<ClassName>;
```

In Above Syntax,

- NamespaceName: Is the name of the namespace and it will refer to all classes, interfaces, structures, and enumerations.
- ClassName: Is the name of the specific class defined in the namespace that you want to access.

The following code demonstrates the use of the using keyword with namespaces:

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

In Above Code,

- The System namespace is imported within the program with the using keyword.
- If this were not done, the program would not even compile as the Console class exists in the System namespace.

Output

Hello World

The following code refers to the Console class of the System namespace multiple times:

```
class World
{
static void Main(string[] args)
{
System.Console.WriteLine("Hello World");
System.Console.WriteLine("This is C# Programming");
System.Console.WriteLine("You have executed a simple program of
C#");
}
}
```

In Above code, the class is not imported, but the System namespace members are used along with the statements.

Output

Hello World
This is C# Programming
You have executed a simple program of C#

Custom Namespaces

- C# allows you to create namespaces with appropriate names to organize structures, classes, interfaces, delegates, and enumerations that can be used across different C# applications.
- When using a custom namespace, you need not worry about name clashes with classes, interfaces, and so on in other namespaces.
- Custom namespaces:
  - Enable you to control the scope of a class by deciding the appropriate namespace for the class.
  - Declared using the namespace keyword and is accessed with the using keyword similar to any built-in namespace.

The following syntax is used to declare a custom namespace:

```
namespace <NamespaceName>
{
```

```
//type-declarations;
}
```

In Above Syntax:

- NamespaceName: Is the name of the custom namespace.
- type-declarations: Are the different types that can be declared. It can be a class, interface, struct, enum, delegate, or another namespace.

The following code creates a custom namespace named Department:

```
namespace Department
{
class Sales
{
static void Main(string [] args)
{
System.Console.WriteLine("You have created a custom namespace
named Department");
}
}
}
```

In Above code,

- Department is declared as the custom namespace.
- The class Sales is declared within this namespace.

Output

You have created a custom namespace named Department

- Once a namespace is created, C# allows:
  - Additional classes to be included later in that namespace. Hence, namespaces are additive.
  - A namespace to be declared more than once.
- These namespaces can be split and saved in separate files or in the same file.
- At the time of compilation, these namespaces are added together.

Important Guidelines for Creating Custom Namespaces

- While designing a large framework for a project, it is required to create namespaces to group the types into the appropriate namespaces such that the identical types do not collide.
- Therefore, the following guidelines must be considered for creating custom namespaces:
    - All similar elements such as classes and interfaces must be created into a single namespace. This will form a logical grouping of **similar** types and any programmer will be easily able to search for similar classes.
    - Creating deep hierarchies that are difficult to browse must be avoided.
    - Creating too many namespaces must be avoided for simplicity.

What Is Main Method ?

Main method is the entry point of your application.

Reading & Writing To The Console In C# Application

Input & Output In C# Program

- Programmers often need to display the output of a C# program to users.
- The programmer can use the command line interface to display the output.
- The programmer can similarly accept inputs from a user through the command line interface.
- Such input and output operations are also known as console operations.

Console Operations

Following are the features of the console operations:

- Console operations are tasks performed on the command line interface using executable commands.
- The console operations are used in software applications because these operations are easily controlled by the operating system.
- This is because console operations are dependent on the input and output devices of the computer system.
- A console application is one that performs operations at the command prompt.
- All console applications consist of three streams, which are a series of bytes. These streams are attached to the input and output devices of the computer system and they handle the input and output operations.

There are 3 streams:

1. Standard in
2. Standard out
3. Standard err

Standard in

The standard in stream takes the input and passes it to the console application for processing.

Standard out

The standard out stream displays the output on the monitor.

Standard err

The standard err stream displays error messages on the monitor.

Output Methods In C#

- In C#, all console operations are handled by the Console class of the System namespace.
- A namespace is a collection of classes having similar functionalities.
- To write data on the console, you need the standard output stream, provided by the Console class.
- There are two output methods that write to the standard output stream as follows:
    - Console.Write(): Writes any type of data.
    - Console.WriteLine(): Writes any type of data and this data ends with a new line character in the standard output stream. This means any data after this line will appear on the new line.

The following syntax is used for the Console.Write() method, which allows you to display the information on the console window:

```
Console.Write("<data>" + variables);
```

Where,

- data: Specifies strings or escape sequence characters enclosed in double quotes.
- variables: Specify variable names whose value should be displayed on the console.

The following syntax is used for the Console.WriteLine() method, which allows you to display the information on a new line in the console window:

```
Console.WriteLine("<data>" + variables);
```

The following code shows the difference between the Console.Write() and Console.WriteLine() methods:

```
Console.WriteLine("C# is a powerful programming language");
Console.WriteLine("C# is a powerful");
Console.WriteLine("programming language");
Console.Write("C# is a powerful");
Console.WriteLine(" programming language");
```

Output

C# is a powerful programming language
C# is a powerful
programming language
C# is a powerful programming language

What Are Placeholders?

- The WriteLine() and Write() methods accept a list of parameters to format text before displaying the output.
- The first parameter is a string containing markers in braces to indicate the position, where the values of the variables will be substituted. Each marker indicates a zero-based index based on the number of variables in the list.
- The following code uses placeholders in the Console.WriteLine() method to display the result of the multiplication operation:

```
int number, result;
number = 5;
result = 100 * number;
Console.WriteLine("Result is {0} when 100 is multiplied by {1}", result,number);
result = 150 / number;
Console.WriteLine("Result is {0} when 150 is divided by {1}", +result, number);
```


Output

Result is 500 when 100 is multiplied by 5
Result is 30 when 150 is divided by 5
Here, {0} is replaced with the value in result and {1} is replaced with the value in number.

Input Methods In C#

- In C#, to read data, you need the standard input stream. This stream is provided by the input methods of the Console class. There are two

input methods that enable the software to take in the input from the standard input stream.

- These methods are as follows:
  - Console.Read(): Reads a single character.
  - Console.ReadLine(): Reads a line of strings.

The following code reads the name using the ReadLine() method and displays the name on the console window:

```
string name;
Console.Write("Enter your name: ");
name = Console.ReadLine();
Console.WriteLine("You are {0}",name);
```

In Above Code,

- The ReadLine() method reads the name as a string and the string that is given is displayed as output using placeholders.

Output

Enter your name: David Blake
You are David Blake

The following code demonstrates the use of placeholders in the Console.WriteLine() method:

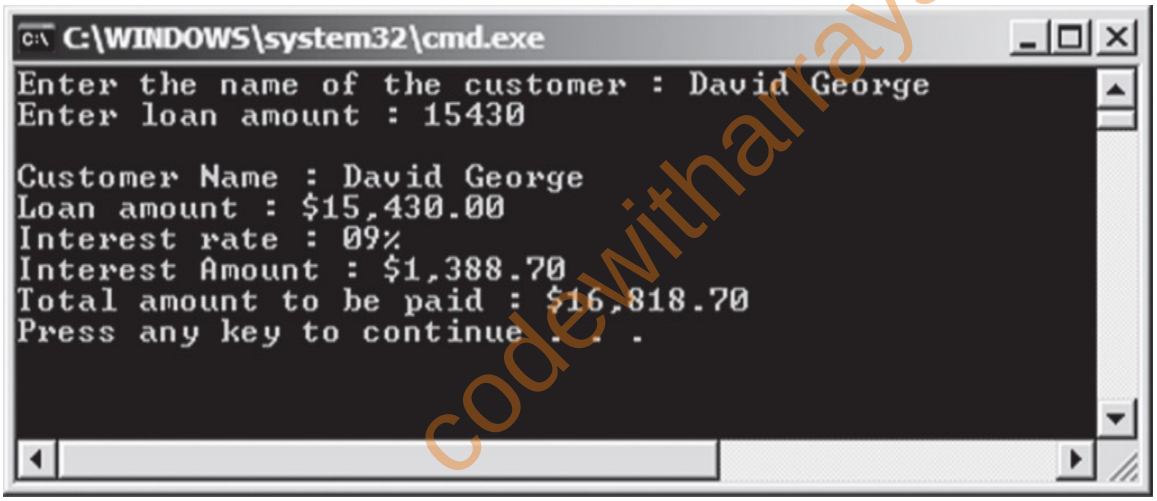
```
using System;
class Loan
{
static void Main(string[] args)
{
stringcustName;
doubleloanAmount;
float interest = 0.09F;
doubleinterestAmount = 0;
doubletotalAmount = 0;
Console.Write("Enter the name of the customer : ");
custName = Console.ReadLine();
Console.Write("Enter loan amount : ");
loanAmount = Convert.ToDouble(Console.ReadLine());
interestAmount = loanAmount * interest;
totalAmount = loanAmount + interestAmount;
Console.WriteLine("\nCustomer Name : {0}", custName);
```

```
Console.WriteLine("Loan amount : ${0:#,###.#0} \nInterest rate : {1:0#%} \nInterest Amount
: ${2:#,###.#0}",
loanAmount, interest, interestAmount );
Console.WriteLine("Total amount to be paid : ${0:#,###.#0} ", totalAmount);
}
}
```

In Above code,

- The name and loan amount are accepted from the user using the Console.ReadLine() method. The details are displayed on the console using the Console.Writeline() method.
- The placeholders {0}, {1}, and {2} indicate the position of the first, second, and third parameters respectively.
- The 0 specified before # pads the single digit value with a 0. The #option specifies the digit position.
- The % option multiplies the value by 100 and displays the value along with the percentage sign.

The following figure displays the example of placeholders:



Numeric Format Specifiers In C#

- Format specifiers are special characters that are used to display values of variables in a particular format. For example, you can display an octal value as decimal using format specifiers.

- In C#, you can convert numeric values in different formats. For example, you can display a big number in an exponential form.
- To convert numeric values using numeric format specifiers, you should enclose the specifier in curly braces. These curly braces must be enclosed in double quotes. This is done in the output methods of the Console class.

The following is the syntax for the numeric format specifier:

```
Console.WriteLine("{format specifier}", <variable name>);
```

Where,

- formatspecifier: Is the numeric format specifier.
- variable name: Is the name of the integer variable.

Numeric format specifiers work only with numeric data that can be suffixed with digits. The digits specify the number of zeros to be inserted, after the decimal location.

Example

- If you use a specifier such as C3, three zeros will be suffixed, after the decimal location of the given number. The following table lists some of the numeric format specifiers in C#:

Format Specifier	Name	Description
C or c	Currency	The number is converted to a string that represents a currency amount.
D or d	Decimal	The number is converted to a string of decimal digits 9), prefixed by a minus sign in case the number is negative. The precision specifier indicates the minim number of digits desired in the resulting string. This format is supported for fundamental types only.
E or e	Scientific (Exponential)	The number is converted to a string of the form '-d.ddd...E+ddd' or '-d.ddd...e+ddd', where each 'd' indicates a digit (0-9).

- Custom numeric format strings contain more than one custom numeric format specifiers and define how data is formatted.
- A custom numeric format string is defined as any string that is not a standard numeric format string. The following table lists the custom numeric format specifiers and their description:

Format Specifier	Description
0	If the value being formatted contains a digit where '0' appears, then it is copied result string
#	If the value being formatted contains a digit where '#' appears, then it is copied result string
.	The first '.' character verifies the location of the decimal separator
,	The ',' character serves as a thousand separator specifier and a number scaling spec
%	The '%' character in a format string multiplies a number with 100 before it is format
E0, E+0,E-0, e0, e+0, e-0	If any of the given strings are present in the format string and they are followed least one '0' character, then the number is formatted using scientific notation
\	The backslash character causes the next character in the format string to be inter as an escape sequence
'ABC' "ABC"	The characters that are enclosed within single or double quotes are copied to the string
;	The ';' character separates a section into positive, negative, and zero numbers
Other	Any of the other characters are copied to the result string

The following code demonstrates the conversion of a numeric value using C, D, and E format specifiers:

```
int num = 456;
Console.WriteLine("{0:C}", num);
Console.WriteLine("{0:D}", num);
Console.WriteLine("{0:E}", num);
```

Output

$456.00
456
4.560000E+002

The following code demonstrates the use of custom numeric format specifiers:

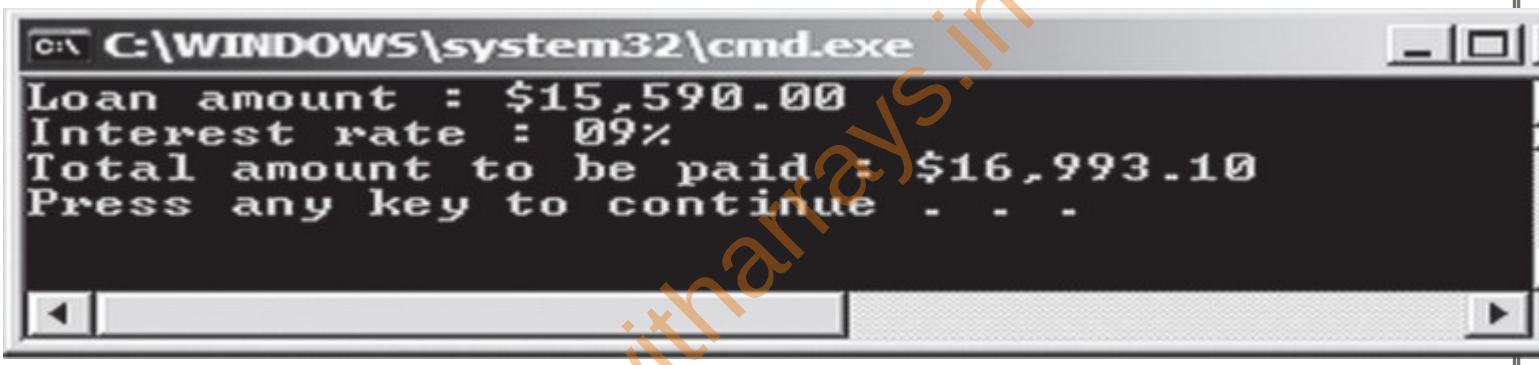
```
using System;
class Banking
{
```

```
static void Main(string[] args)
{
double loanAmount = 15590;
float interest = 0.09F;
double interestAmount = 0;
double totalAmount = 0;
interestAmount = loanAmount * interest ;
totalAmount = loanAmount + interestAmount;
Console.WriteLine("Loan amount : ${0:#,###.#0} ", loanAmount);
Console.WriteLine("Interest rate : {0:0#%} ", interest);
Console.WriteLine("Total amount to be paid :${0:#,###.#0}",totalAmount);
}
}
```

In Above Code,

- The #, %, ., and 0 custom numeric format specifiers are used to display the loan details of the customer in the desired format.

The following figure displays the example of custom numeric format specifiers:



Some More Number Format Specifiers

There are some additional number format specifiers that are described in the following table:

Format Specifier	Name	Description
F or f	Fixed-point	The number is converted to a string of the form '-ddd.ddd...' where each 'd' indicates a digit (0-9). If the number is negative, the string starts with a minus sign.
N or n	Number	The number is converted to a string of the form '-d,ddd,ddd.ddd...', where each 'd' indicates a digit (0-9). If the number is negative, the string starts with a minus sign.
X or x	Hexadecimal	The number is converted to a string of hexadecimal digits. Uses "X" to produce "ABCDEF", and "x" to produce "abcdef".

The following figure demonstrates the conversion of a numeric value using F, N, and X format specifiers:

```
int num = 456;
Console.WriteLine("{0:F}", num);
Console.WriteLine("{0:N}", num);
Console.WriteLine("{0:X}", num);
```

Output

456.00
456.00
1C8

Source Code

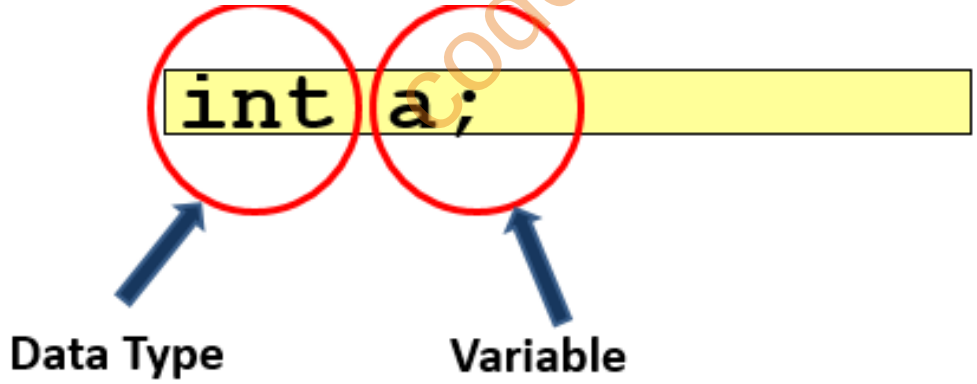
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

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

            Console.WriteLine("Enter first number");
            int num1 = int.Parse(Console.ReadLine());
```

```
            Console.WriteLine("Enter Second number");
            int num2 = int.Parse(Console.ReadLine());
            int sum = num1 + num2;

            Console.WriteLine("Addition result is: {0}", sum);
            Console.ReadLine();


            //Console.WriteLine("Enter your first Name ");
            //string fname = Console.ReadLine();
            //Console.WriteLine("Enter your Last Name ");
            //string lname = Console.ReadLine();
            //Console.WriteLine("Your Name is : " + fname + " " + lname); // concatenation
syntax
            //Console.WriteLine("Your Name is: {0} {1}", fname, lname); // placeholder
syntax
            //Console.ReadLine();
        }

    }
}
```

Variables & Data Types in C#

Variables In C#

- A variable is used to store data in a program and is declared with an associated data type.
- A variable has a name and may contain a value.
- A data type defines the type of data that can be stored in a variable.



A variable is an entity whose value can keep changing during the course of a program.

Example

- The age of a student, the address of a faculty member, and the salary of an employee are all examples of entities that can be represented by variables.

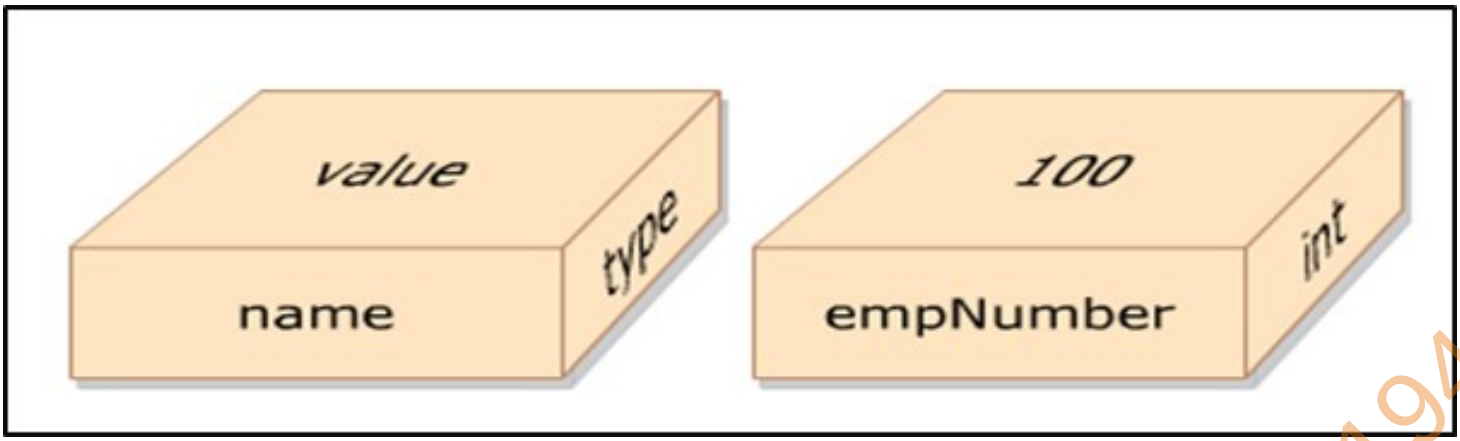
Variables

- In C#, a variable is a location in the computer's memory that is identified by a unique name and is used to store a value. The name of the variable is used to access and read the value stored in it.
- Different types of data such as a character, an integer, or a string can be stored in variables. Based on the type of data that needs to be stored in a variable, variables can be assigned different data types.

Using Variables

- In C#, memory is allocated to a variable at the time of its creation and a variable is given a name that uniquely identifies the variable within its scope.
- On initializing a variable, the value of a variable can be changed as required to keep track of data being used in a program. When referring to a variable, you are actually referring to the value stored in that variable.

The following figure illustrates the concept of a variable:



The following syntax is used to declare variables in C#:

```
<datatype><variableName>;
```

Where,

- datatype: Is a valid data type in C#.
- variableName: Is a valid variable name.

The following syntax is used to initialize variables in C#:

```
<variableName> = <value>;
```

Where,

- = Is the assignment operator used to assign values.
- value: Is the data that is stored in the variable.

The following code declares two variables, namely, empNumber and empName:

```
int empNumber;
string empName;
```

In Above Code,

- an integer variable declares empNumber, and a string variable, empName. Memory is allocated to hold data in each variable.

- Values can be assigned to variables by using the assignment operator (=), as follows:
  - empNumber = 100;
  - empName = “David Blake”;
- You can also assign a value to a variable upon creation, as follows:
  - int empNumber = 100;

Data Types

- Different types of values such as numbers, characters, or strings can be stored in different variables. To identify the type of data that can be stored in a variable, C# provides different data types.
- When a variable is declared, a data type is assigned to the variable. This allows the variable to store values of the assigned data type.
- In C# programming language, data types are divided into two categories:

1. Value Types

- Variables of value types store actual values that are stored in a stack that results in faster memory allocation to variables of value types.
- Most of the built-in data types are value types.
- The value type built-in data types are int, float, double, char, and bool. User-defined value types are created using the struct and enum keywords.

2. Reference Types

- Variables of reference type store the memory address of other variables in a heap.
- These values can either belong to a built-in data type or a user-defined data type.

Pre-defined Data Types

- The pre-defined data types are referred to as basic data types in C# that have a pre-defined range and size.
- The size of the data type helps the compiler to allocate memory space and the range helps the compiler to ensure that the value assigned, is within the range of the variable's data type.

Data Type	Size	Range
byte	Unsigned 8-bit integer	0 to 255
sbyte	Signed 8-bit integer	-128 to 127
short	Signed 16-bit integer	-32,768 to 32,767
ushort	Unsigned 16-bit integer	0 to 65,535
int	Signed 32-bit integer	-2,147,483,648 to 2,147,483,647
uint	Unsigned 32-bit integer	0 to 4,294,967,295
long	Signed 64-bit integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	Unsigned 64-bit integer	0 to 18,446,744,073,709,551,615
float	32-bit floating point with 7 digits precision	±1.5e−45 to ±3.4e38
double	64-bit floating point with 15-16 digits precision	±5.0e−324 to ±1.7e308
decimal	128-bit floating point with 28-29 digits precision	±1.0 × 10e−28 to ±7.9 × 10e28
char	Unicode 16-bit character	U+0000 to U+ffff
bool	Stores either true or false	true or false

Reference data types store the memory reference of other variables that hold the actual values that can be classified into the following types:

Object

- Object is a built-in reference data type that is a base class for all pre-defined and user-defined data types. A class is a logical structure that represents a real world entity. The pre-defined and user-defined data types are created based on the Object class.

String

- String is a built-in reference type that signifies Unicode character string values. It allows you to assign and manipulate string values. Once strings are created, they cannot be modified.

Class

- A class is a user-defined structure that contains variables and methods. For example, the Employee class can be a user-defined structure that can contain variables such as empSalary, empName, and empAddress. It can also contain methods such as CalculateSalary(), which returns the net salary of an employee.

Delegates

- A delegate is a user-defined reference type that stores the reference of one or more methods.

Interface

- An interface is a user-defined structure that groups related functionalities which may belong to any class or struct.

Array

- An array is a user-defined data structure that contains values of the same data type, such as marks of students.

INTEGRAL TYPE - Integer DataType

Signed Integers (Which Takes Negative And Positive Values)

Unsigned Integers (Which Only Takes Positive Values)

- **sbyte**
- **byte**
- **short**

- **ushort**
- **int**
- **uint**
- **long**
- **ulong**

DataType, Its Range & Its Size

Type	Range	Size
sbyte	-128 to 127	Signed 8-bit integer
byte	0 to 255	Unsigned 8-bit integer
char	U+0000 to U+ffff	Unicode 16-bit character
short	-32,768 to 32,767	Signed 16-bit integer
ushort	0 to 65,535	Unsigned 16-bit integer
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint	0 to 4,294,967,295	Unsigned 32-bit integer
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

2 Properties to get maximum & minimum value of data type

- **MINVALUE** PROPERTY
- **MAXVALUE** PROPERTY

Boolean Data Type

Bool keyword is used for Boolean data type which only stores **TRUE** or **FALSE**.

Float Double And Decimal Data Type

C# Alias	.NET Type	Size	Precision
float	System.Single	4 bytes	7 digits
double	System.Double	8 bytes	15-16 digits
decimal	System.Decimal	16 bytes	28-29 decimal places

String And Character Data Type

- **String** stores multiple characters in a single variable.
- Double quotes will be used with string data type.
- **Char** stores single character at a time in a variable.
- Single quotes will be used for char data type.

Verbatim Literal

- Verbatim literal is a string with an @ symbol.
- Verbatim literal make escape sequences translate as normal printable characters to enhance readability.

Practical Example:

- **WITHOUT VERBATIM LITERAL**: "D:\\Adil\\Csharp\\Tutorials" – Less readable
- **WITH VERBATIM LITERAL**: @"D:\Adil\Csharp\Tutorials" – More readable

Rules For Defining Variables

A variable needs to be declared before it can be referenced by following certain rules as follows:

- A variable name can begin with an uppercase or a lowercase letter. The name can contain letters, digits, and the underscore character (_).
- The first character of the variable name must be a letter and not a digit. The underscore is also a legal first character, but it is not recommended at the beginning of a name.
- C# is a case-sensitive language; hence, variable names count and Count refer to two different variables.
- C# keywords cannot be used as variable names. If you still need to use a C# keyword, prefix it with the '@' symbol.
- It is always advisable to give meaningful names to variables such that the name gives an idea about the content that is stored in the variable.

The following table displays a list of valid and invalid variable names in C#:

Variable Name	Valid/Invalid
Employee	Valid
student	Valid
_Name	Valid
Emp_Name	Valid
@goto	Valid
static	Invalid as it is a keyword
4myclass	Invalid as a variable cannot start with a digit
Student&Faculty	Invalid as a variable cannot have the special character &

Variable Declaration

- In C#, you can declare multiple variables at the same time in the same way you declare a single variable.
- After declaring variables, you need to assign values to them.
- Assigning a value to a variable is called initialization.
- You can assign a value to a variable while declaring it or at a later time.

The following is the syntax to declare and initialize a single variable:

```
<data type><variable name> = <value>;
```

Where,

- data type: Is a valid variable type.
- variable name: Is a valid variable name or identifier.
- value: Is the value assigned to the variable.

The following is the syntax to declare multiple variables:

```
<data type><variable name1>, <variable name2>,..,<variable nameN>;
```

Where,

- data type: Is a valid variable type.
- variable name1, variable name2, variable nameN: Are valid variable names or identifiers.

The following is the syntax to declare and initialize multiple variables:

```
<data <data type><variable name1> = <value1>, <variable name2> = <value2>;
```

The following code demonstrates how to declare and initialize variables in C#:

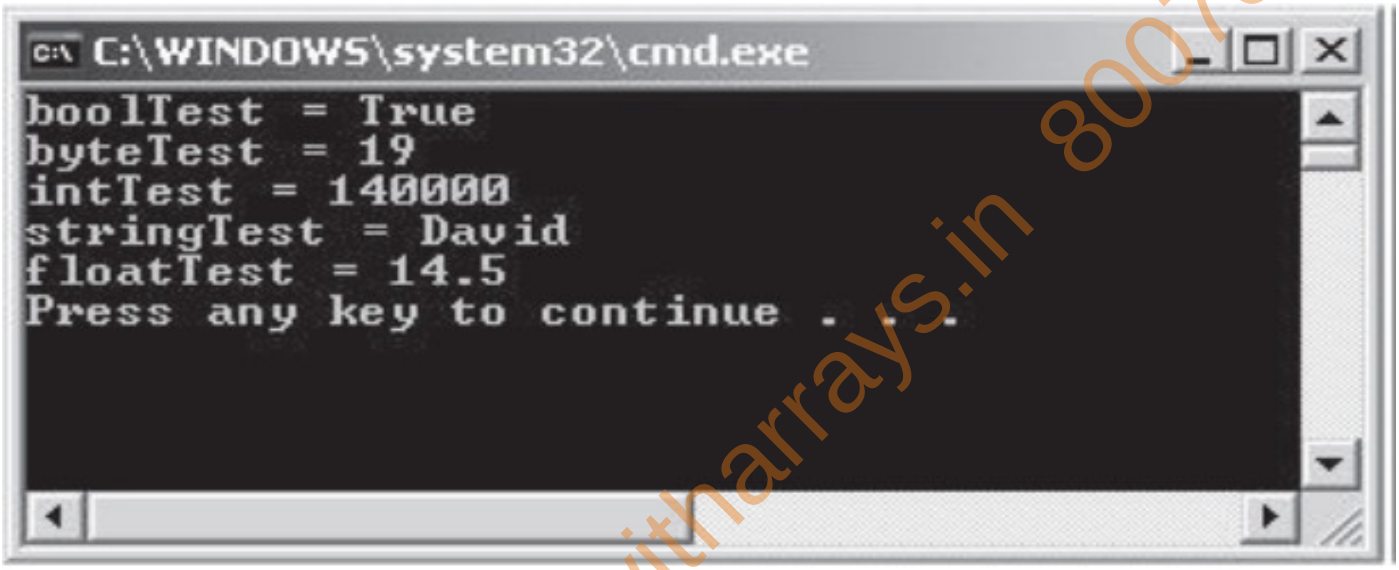
```
bool boolTest = true;
short byteTest = 19;
int intTest;
string stringTest = "David";
float floatTest;
int Test = 140000;
```

```
floatTest = 14.5f;
Console.WriteLine("boolTest = {0}", boolTest);
Console.WriteLine("byteTest = " + byteTest);
Console.WriteLine("intTest = " + intTest);
Console.WriteLine("stringTest = " + stringTest);
Console.WriteLine("floatTest = " + floatTest);
```

In Above Code,

- Variables of type bool, byte, int, string, and float are declared. Values are assigned to each of these variables and are displayed using the WriteLine() method of the Console class.

The code displays the following output:



Data types conversion in c# programming

There are two types of conversions **implicit** and **explicit** conversion.

Implicit conversion is done by the compiler.

1. When there is no loss of information if the conversion is done.
2. If there is no possibility of throwing exception during the conversion.

- Example: converting an int to a float will not loose any data and no exception will be thrown, hence an implicit conversion can be done.
- Where as when converting a float to an int, we loose the fractional part and also a possibility of overflow exception. Hence, in this case an explicit conversion is required.
- For explicit conversion we can use the **convert class** in c#.
- **Parse function** of int, float. Strings

Program:-

```
//   int x = 100;

 // float y = x; // implicit conversion of data type

  string a = "100";
  string b = "200";

  //int num1 = int.Parse(a);
  //int num2 = int.Parse(b);

  int num1 = Convert.ToInt32(a);
  //int num2 = Convert.ToInt32(b);
  //int c = num1 + num2;

  Console.WriteLine(a);

  //float a = 34.657f;
  //int b = (int)a; // explicit conversion of data type
  //int b = Convert.ToInt32(a);
  //Console.WriteLine(b);
  Console.ReadLine();
```

Constants & Literals In C# Programming

A constant has a fixed value that remains unchanged throughout the program while a literal provides a mean of expressing specific values in a program.

Example

- Consider a code that calculates the area of the circle.
- To calculate the area of the circle, the value of pi and radius must be provided in the formula.
- The value of pi is a constant value.
- This value will remain unchanged irrespective of the value of the radius provided.
- Similarly, constants in C# are fixed values assigned to identifiers that are not modified throughout the execution of the code.
- They are defined when you want to preserve values to reuse them later or to prevent any modification to the values.

Constants In C#

- A constant has a fixed value that remains unchanged throughout the program.
- In C#, you can declare constants for all data types.
- You have to initialize a constant at the time of its declaration.
- Constants are declared for **value types** rather than for **reference types**.
- To declare an identifier as a constant, the **"const"** keyword is used in the identifier declaration. The compiler can identify constants at the time of compilation, because of the **"const"** keyword.

The following syntax is used to initialize a constant:

```
const<data type><identifier name> = <value>;
```

where,

- **const**: Keyword denoting that the identifier is declared as constant.
- **data type:** Data type of constant.
- **identifier name:** Name of the identifier that will hold the constant.
- **value**: Fixed value that remains unchanged throughout the execution of the code.

The following code declares a constant named _pi and a variable named radius to calculate the area of the circle:

```
const float _pi = 3.14F;
float radius = 5;
float area = _pi * radius * radius;
Console.WriteLine("Area of the circle is " + area);
```

In Above Code,

- A constant called _pi is assigned the value 3.14, which is a fixed value. The variable, radius, stores the radius of the circle. The code calculates the area of the circle and displays it as the output.

Literals In C# Programming

- A literal is a static value assigned to variables and constants.
- Numeric literals might suffix with a letter of the alphabet to indicate the data type of the literal.
- This letter can be either in upper or lowercase.

Example

For example, in the following declaration,

```
string bookName = "Csharp"
```

Csharp is a literal assigned to the variable bookName of type string.

Types Of Literals

- Boolean Literal
- Integer Literal
- Real Literal
- Character Literal
- String Literal
- Null Literal

Boolean Literal

- Boolean Literal: Boolean literals have two values, true or false. For example,
  - bool val = true;
- where,
  - true: Is a Boolean literal assigned to the variable val.

Integer Literal

- Integer Literal: An integer literal can be assigned to int, uint, long, or ulongdata types. Suffixes for integer literals include U, L, UL, or LU. U denotes uint or ulong, L denotes long. UL and LU denote ulong. For example,
  - long val = 53L;
- Where,
  - 53L: Is an integer literal assigned to the variable val.

Real Literal

- Real Literal: A real literal is assigned to float, double (default), and decimal data types. This is indicated by the suffix letter appearing after the assigned value. A real literal can be suffixed by F, D, or M. F denotes float, D denotes double, and M denotes decimal. For example,
  - float val = 1.66F;
- Where,

- 1.66F: Is a real literal assigned to the variable val.

Character Literal

- Character Literal: A character literal is assigned to a char data type. A character literal is always enclosed in single quotes. For example,
  - char val = 'A';
- Where,
  - A: Is a character literal assigned to the variable val.

String Literal

- String Literal: There are two types of string literals in C#, regular and verbatim. A regular string literal is a standard string. A verbatim string literal is similar to a regular string literal but is prefixed by the '@' character. A string literal is always enclosed in double quotes. For example,
  - string mailDomain = "@gmail.com";
- Where,
  - @gmail.com: Is a verbatim string literal.

Null Literal

- Null Literal: The null literal has only one value, null. For example,
  - string email = null;
- Where,
  - null: Specifies that e-mail does not refer to any objects (reference).

Date And Time Format Specifiers In C#

A date and time format specifier is a special character that enables you to display the date and time values in different formats.

Example

- You can display a date in mm-dd-yyyy format and time in hh:mm format.
- If you are displaying GMT time as the output, you can display the GMT time along with the abbreviation GMT using date and time format specifiers.
- The date and time format specifiers allow you to display the date and time in 12-hour and 24-hour formats.
- The following is the syntax for date and time format specifiers:

```
Console.WriteLine("{format specifier}", <datetime object>);
```

Where,

- formatspecifier: Is the date and time format specifier.
- datetime object: Is the object of the DateTime class.

Format Specifier	Name
d	Short date
D	Long date
f	Full date/time (short time)
F	Full date/time (long time)
g	•General date/time (short time)

Syntax

```
DateTime dt = DateTime.Now;
Console.WriteLine("{0:D}", dt);
Console.WriteLine("{0:f}", dt);
Console.WriteLine("{0:F}", dt);
```

Console.WriteLine("{0:g}", dt);
Console.WriteLine("{0:d} {1:D}", dt, dt);

Time Formats

Format Specifier	Name
G	General date/time (long time)
m or M	Month day
t	Short time
T	Long time
y or Y	Year month pattern
ddd	Represents the abbreviated name of the day of the week.
dddd	Represents the full name of the day of the week.
FF	Represents the two digits of the seconds fraction
HH	Represents the hour from 00 to 23
MM	Represents the month as a number from 01 to 12
MMM	Represents the abbreviated name of the month
ss	•Represents the seconds as a number from 0 to 59

Full Date & Time Example

Console.WriteLine("{0:HH:mm:ss tt}", dt);
Console.WriteLine("{0:dd-MM-yyyy}", dt);

Using Standard Date and Time Format Specifiers

The following code demonstrates the conversion of a specified date and time using the d, D, f, F, and g date and time format specifiers:

```
DateTimedt = DateTime.Now;
// Returns short date (MM/DD/YYYY)
Console.WriteLine("Short date format(d): {0:d}", dt);
// Returns long date (Day, Month Date, Year)
Console.WriteLine("Long date format (D): {0:D}", dt);
// Returns full date with time without seconds
Console.WriteLine("Full date with time without seconds (f):{0:f}", dt);
// Returns full date with time with seconds
Console.WriteLine("Full date with time with seconds (F):{0:F}", dt);
// Returns short date and short time without seconds
Console.WriteLine("Short date and short time without seconds (g):{0:g}", dt);
```

Output

Short date format(d): 23/04/2007
Long date format (D): Monday, April 23, 2007
Full date with time without seconds (f):Monday, April 23, 2007 12:58 PM
Full date with time with seconds (F):Monday, April 23, 2007 12:58:43 PM
Short date and short time without seconds (g):23/04/2007 12:58 PM

The following code demonstrates the conversion of a specified date and time using the G, m, t, T, and y date and time format specifiers:

```
DateTimedt = DateTime.Now;
// Returns short date and short time with seconds
Console.WriteLine("Short date and short time with seconds (G):{0:G}", dt);
// Returns month and day - M can also be used
Console.WriteLine("Month and day (m):{0:m}", dt);
// Returns short time
Console.WriteLine("Short time (t):{0:t}", dt);
// Returns short time with seconds
Console.WriteLine("Short time with seconds (T):{0:T}", dt);
// Returns year and month - Y also can be used
Console.WriteLine("Year and Month (y):{0:y}", dt);
```

Output

Short date and short time with seconds (G):23/04/2007 12:58:43 PM
Month and day (m):April 23
Short time (t):12:58 PM
Short time with seconds (T):12:58:43 PM
Year and Month (y):April, 2007

Custom DateTime Format Strings

Any non-standard DateTime format string is referred to as a custom DateTime format string. Custom DateTime format strings consist of more than one custom DateTime format specifiers. The following table lists some of the custom DateTime format specifiers:

Format Specifier	Name
ddd	Represents the abbreviated name of the day of the week
dddd	Represents the full name of the day of the week
FF	Represents the two digits of the seconds fraction
H	Represents the hour from 0 to 23
HH	Represents the hour from 00 to 23
MM	Represents the month as a number from 01 to 12
MMM	Represents the abbreviated name of the month
s	Represents the seconds as a number from 0 to 59

The following code demonstrates the use of custom DateTime format specifiers:

```
using System;
class DateTimeFormat
{
public static void Main(string[] args)
{
DateTime date = DateTime.Now;
Console.WriteLine("Date is {0:ddd MMM dd, yyyy}", date);
Console.WriteLine("Time is {0:hh:mm tt}", date);
Console.WriteLine("24 hour time is {0:HH:mm}", date);
Console.WriteLine("Time with seconds: {0:HH:mm:ss tt}", date);
Console.WriteLine("Day of month: {0:m}", date);
Console.WriteLine("Year: {0:yyyy}", date);
}
}
```


In Above Code,

- The date and time is displayed using the different DateTime format specifiers.

The following figure displays the output of the code:

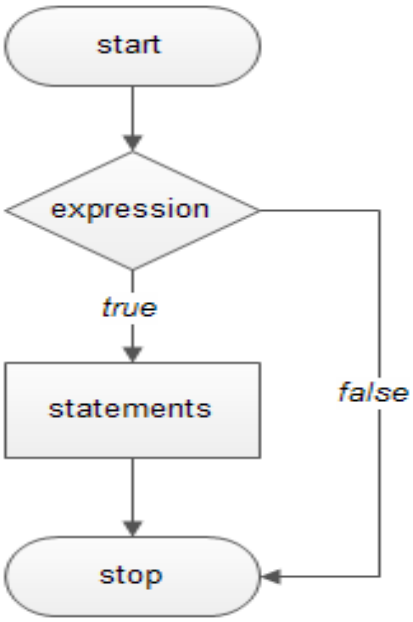
Decision Making Statements In C#

- Is a programming construct supported by C# that controls the flow of a program.
- Executes a particular block of statements based on a boolean condition, which is an expression returning true or false.
- Is referred to as a **decision-making construct**.
- Allow you to take logical decisions about executing different blocks of a program to achieve the required logical output.
- **C# supports the following decision-making constructs:**
- if…else
- if…else…if
- Nested if
- switch…case
- Nested Switch Case

The If Statement

- The **if statement** allows you to execute a block of statements after evaluating the specified logical condition.
- The if statement starts with the **if keyword** and is followed by the **condition**.
- If the condition evaluates to **true**, the block of statements following the if statement is executed.

- If the condition evaluates to **false**, the block of statements following the if statement is ignored and the statement after the block is executed.



The If-else Statement

- In some situations, it is required to define an action for a false condition by using an **if...else construct.**
- The if...else construct starts with the if block followed by an else block and the else block starts with the else keyword followed by a block of statements.
- If the condition specified in the if statement evaluates to false, the statements in the else block are executed.
- The **if…else…if** construct allows you to check multiple conditions to execute a different block of code for each condition.
- It is also referred to as **if-else–if ladder**.

- The construct starts with the if statement followed by multiple else if statements followed by an optional else block.
- The conditions specified in the if…else…if construct are evaluated sequentially.
- The execution starts from the if statement. If a condition evaluates to false, the condition specified in the following else…if statement is evaluated.

Nested If Construct

- The **nested if construct** consists of multiple if statements.
- The nested if construct starts with the if statement, which is called the **outer if** statement, and contains multiple if statements, which are called **inner if** statements.
- In the nested if construct, the outer if condition controls the execution of the inner if statements. The compiler executes the inner if statements only if the condition in the outer if statement is true.
- In addition, each inner if statement is executed only if the condition in its previous inner if statement is true.

Switch…case Construct

- A program is difficult to comprehend when there are too many if statements representing multiple selection constructs.
- To avoid using multiple if statements, in certain cases, the switch…case approach can be used as an alternative.

- The switch…case statement is used when a variable needs to be compared against different values.
- In C#, the flow of execution from one case statement is not allowed to continue to the next case statement and is referred to as the **'no-fall-through'** rule of C#.
- In C#, the flow of execution from one case statement is not allowed to continue to the next case statement and is referred to as the **'no-fall-through'** rule of C#.

Nested-switch…case Construct

- C# allows the switch…case construct to be **nested**. That is, a case block of a switch…case construct can contain another switch… case construct.
- Also, the case constants of the inner switch…case construct can have values that are identical to the case constants of the outer construct.

Arithmetic Operators

- Arithmetic operators are **binary operators** because they work with **two operands**, with the operator being placed in between the operands.
- These operators allow you to perform computations on numeric or string data.
- +, - , / , * , % .

Relational Or Comparison Operators

- Relational operators make a comparison between two **operands** and return a **boolean** value, true, or false.
- **==**
- **!=**
- **>**
- **<**
- **>=**
- **<=**

Logical OR CONDITIONAL Operators

OPERATOR	CONDITION 1	CONDITION 2	RESULT
OR ||	TRUE	TRUE	TRUE
	TRUE	FALSE	TRUE
	FALSE	TRUE	TRUE
	FALSE	FALSE	FALSE

BUILT IN DATA TYPES IN C#

→ INTEGRAL TYPE

- **Signed Integers (Which Takes Negative And Positive Values)**
- **Unsigned Integers (Which Only Takes Positive Values)**
  - **sbyte**
  - **byte**
  - **short**
  - **ushort**
  - **int**
  - **uint**
  - **long**
  - **ulong**

Type	Range	Size
sbyte	-128 to 127	Signed 8-bit integer
byte	0 to 255	Unsigned 8-bit integer
char	U+0000 to U+ffff	Unicode 16-bit character
short	-32,768 to 32,767	Signed 16-bit integer
ushort	0 to 65,535	Unsigned 16-bit integer
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint	0 to 4,294,967,295	Unsigned 32-bit integer
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

→ **MINVALUE** PROPERTY
→ **MAXVALUE** PROPERTY

Boolean Data Type

Bool keyword is used for Boolean data type which only stores **TRUE** or **FALSE**.

Float Double And Decimal Data Type

C# Alias	.NET Type	Size	Precision
float	System.Single	4 bytes	7 digits
double	System.Double	8 bytes	15-16 digits
decimal	System.Decimal	16 bytes	28-29 decimal places

String And Character Data Type

→ **String** stores multiple characters in a single variable.
→ Double quotes will be used with string data type.
→ **Char** stores single character at a time in a variable.
→ Single quotes will be used for char data type.

➢ ESCAPE SEQUENCE

➢ Verbatim Literal

- Verbatim literal is a string with an **@** symbol.
- Verbatim literal make escape sequences translate as normal printable characters to enhance readability.

Practical Example:

→ WITHOUT VERBATIM LITERAL: "D:\\Adil\\Csharp\\Tutorials" – Less readable
→ WITH VERBATIM LITERAL: **@**"D:\Adil\Csharp\Tutorials" – More readable

Data types conversion in c# programming

There are two types of conversions **implicit** and **explicit** conversion.

Implicit conversion is done by the compiler.

1. When there is no loss of information if the conversion is done.
2. If there is no possibility of throwing exception during the conversion.

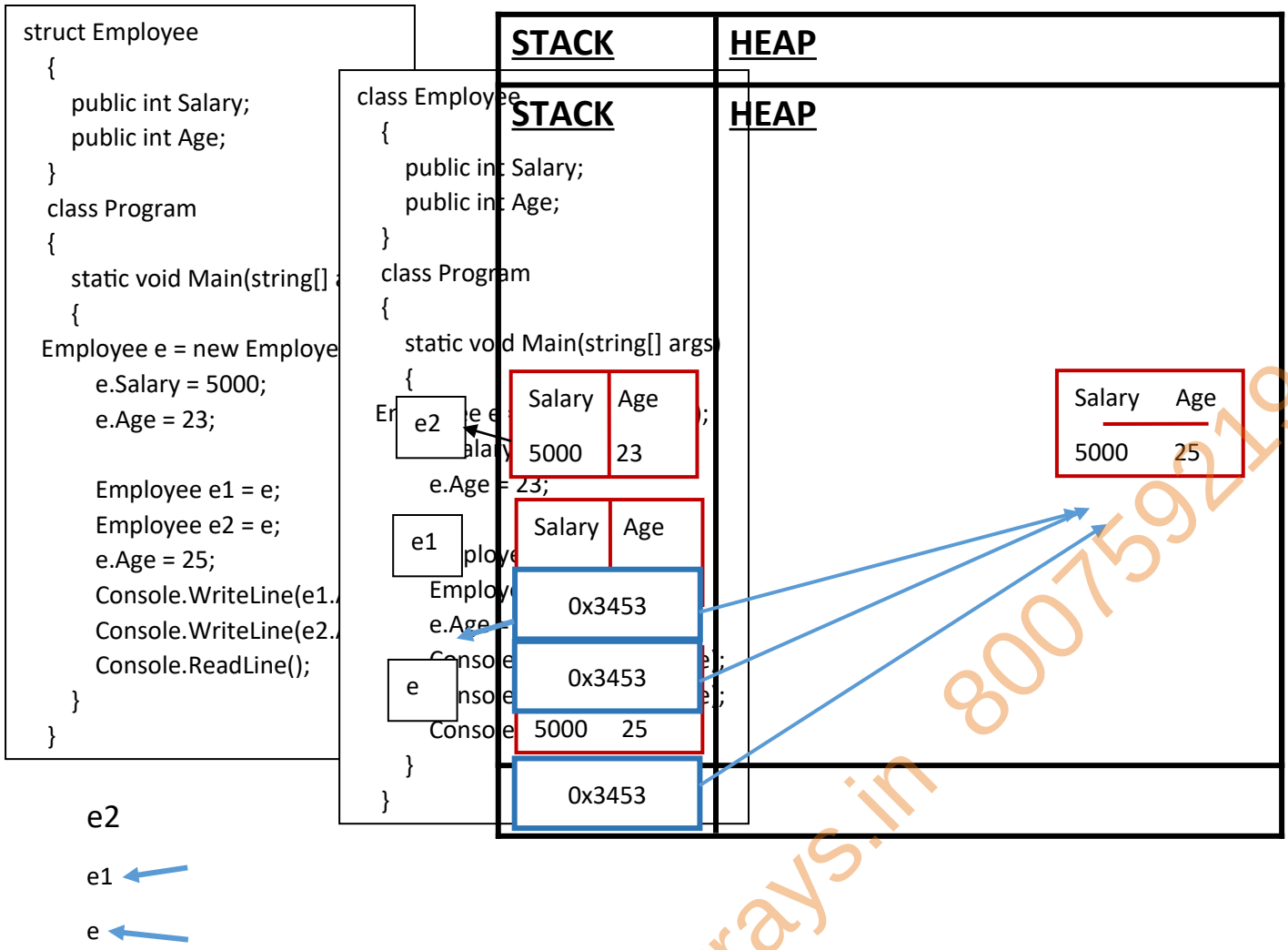
Example: converting an int to a float will not loose any data and no exception will be thrown, hence an implicit conversion can be done.

Where as when converting a float to an int, we loose the fractional part and also a possibility of overflow exception. Hence, in this case an explicit conversion is required.

For explicit conversion we can use the **convert class** in c#.

Parse function of int, float. Strings

Value Type Vs Reference Type In C#



VALUE TYPE:

A data type is a value type if it holds a data value within its own memory space. It means variables of these data types directly contain their values.

The following data types are all of value type:

→ bool
→ byte
→ char
→ decimal
→ double
→ enum
→ float

→ int
→ long
→ sbyte
→ short
→ struct
→ uint
→ ulong
→ ushort

REFERENCE TYPE:

Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

The following data types are of reference type:

→ String
→ All arrays, even if their elements are value types
→ Class
→ object
→ Delegates
→ Interface

→ **Difference between value type and reference types ?**

Value Type	Reference Type
They are stored on stack memory	They are stored on heap memory
Contains actual value	Contains reference to a value
Cannot contain null values. However this can be achieved by nullable types	Can contain null values.
Memory is allocated at compile time	Memory is allocated at run time

→ **Diff bt stack and heap**

Stack	Heap
Values are stored on one another like a stack.	Values are stored in random order like dumped into a huge space

Used for value type	Used for reference types

Var And Dynamic Keywords In C#

Var Keyword:

→ Var was introduced in c# 3.0.
→ Var keyword is used to store any type of data in its variable.
→ Value of var variable is decided at compile time.
→ We have to initialize the variable with **var** keyword.
→ If we want to check the type of value which is stored in var variable then we can use **gettype()** method with the var variable.
→ When we initialize the var variable with some value then we cannot change the value of var variable with some other data type value.
→ We can use all the methods of particular type value which is stored in var variable.
→ Intellisense help is available for the var type of variables. This is because, its type is inferred by the compiler from the type of value it is assigned and as a result, the compiler has all the information related to the type.
→ Var variables cannot be used for property or return values from a function. They can only be used as local variable in a function.
→ We cannot use var variable as a function parameter.
→ Var keyword is of value type.

Dynamic Keyword:

→ Dynamic was introduced in c# 4.0
→ Dynamic keyword is also used to store any type of data in its variable.
→ Value of dynamic variable is decided at run time.
→ Initialization is not mandatory when we declare a variable with dynamic keyword.
→ If we want to check the type of value which is stored in dynamic variable then we can use **gettype()** method with the dynamic variable.
→ When we initialize the dynamic variable with some value then we can change the value of dynamic variable with some other data type value.
→ Intellisense help is not available for dynamic type of variables since their type is unknown until run time. So intellisense help is not available.
→ Dynamic variables can be used to create properties and return values from a function.
→ We can use dynamic variable as a function parameter.
→ Dynamic keyword is of reference type.

BOXING AND UN-BOXING IN C#

- As we all know, C# is a strongly-typed language. That means we have to declare the type of a variable that indicates the kind of values it is going to store, such as integer, double, float, boolean decimal, text, etc.
- In C#, data types are categorized into two types: Value types and Reference types. Value types include simple types (such as int, double, float, bool, and char), enum types, struct types, and Nullable value types. Reference types include class types, interface types, delegate types, and array types.
- In C#, Boxing and unboxing allows developers to convert .NET data types from value type to reference type and vice versa.
- In C#, Converting a value type to a reference type is called called boxing in C# and converting a reference type to a value type is called unboxing in C#.
- Mainly C# has two kinds of data types, value types and reference types. Value type stores the value itself, on the other hand reference type stores the address of the value where it is stored.

- In C#, Some predefined data types such as int, float, double, decimal, bool, char, etc. are value types and object, string, and array are reference types.

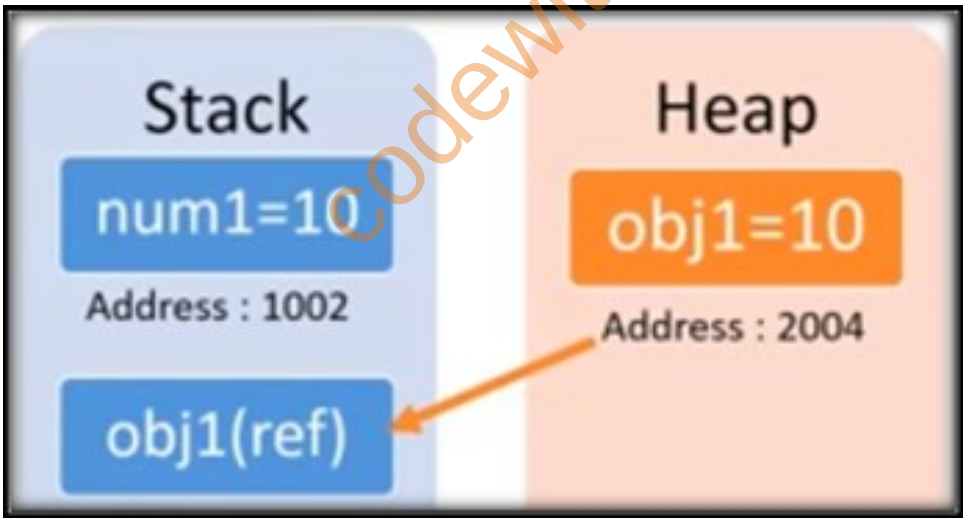
What Is Boxing In C# ?

- Implicit (automatic) conversion of a value type to a reference type is called "Boxing".

Example:

```
int num1 = 10; // int is value type
object obj1 = num1; // Implicit conversion of value type into reference type.
```

- In Boxing process, a value type is being allocated on the heap rather than the stack.



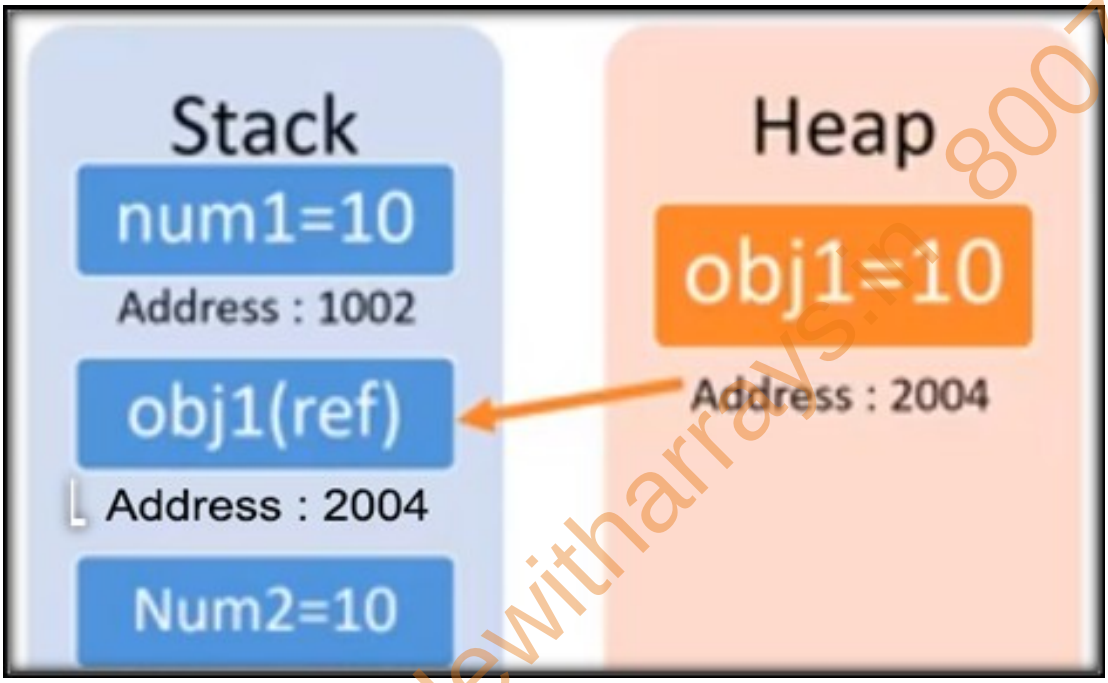
What Is Un-Boxing In C# ?

- Explicit (manual) conversion of the same reference type (which is being created by boxing), back to a value type.

Example:

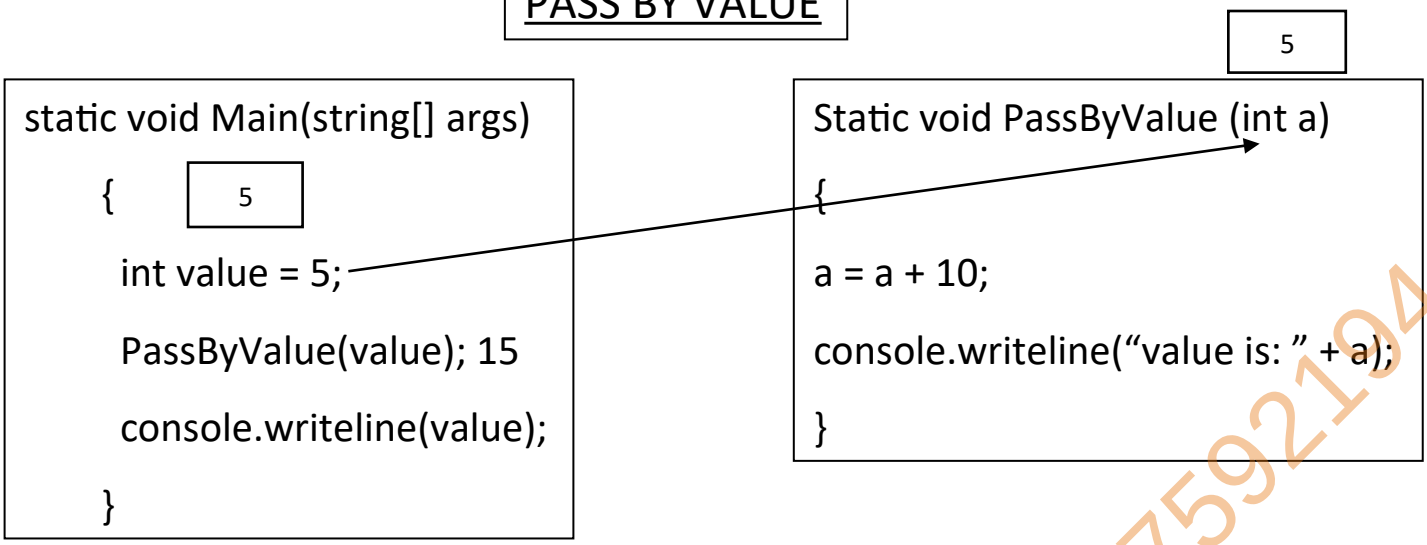
```
int num1 = 10; // int is value type
object obj1 = num1; // Implicit conversion of value type into reference type
(Boxing)
int num2 = (int) obj1; // Explicit conversion of reference type into value
type (Unboxing)
```

- In unboxing process, an unboxed value is being allocated to a variable on the stack rather than the heap.

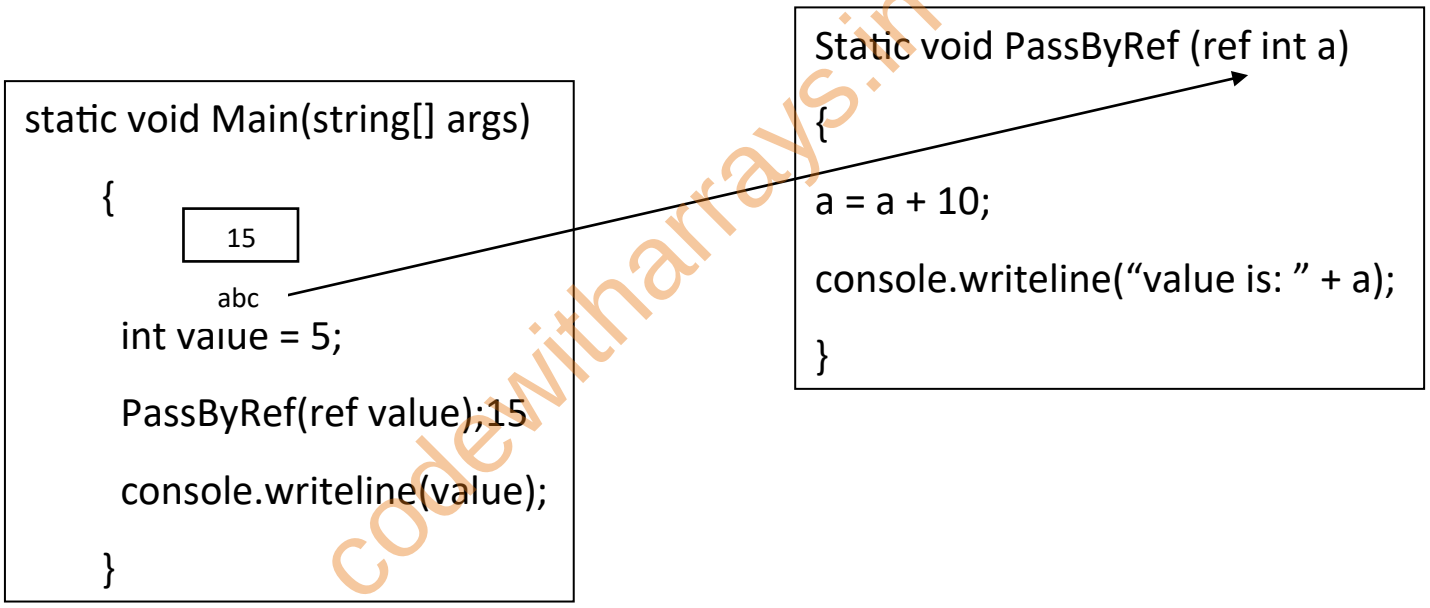


Pass By Value And Pass By Reference In C#
(Ref And Out Keywords)

PASS BY VALUE



PASS BY REFERENCE



PASS BY OUT

```
static void Main(string[] args)
    {
      int value;
      PassByOut(out value);
      console.writeline(value);
    }
```

```
Static void PassByOut (out int a)
{
a = 20;
console.writeline("value is: " + a);
}
```

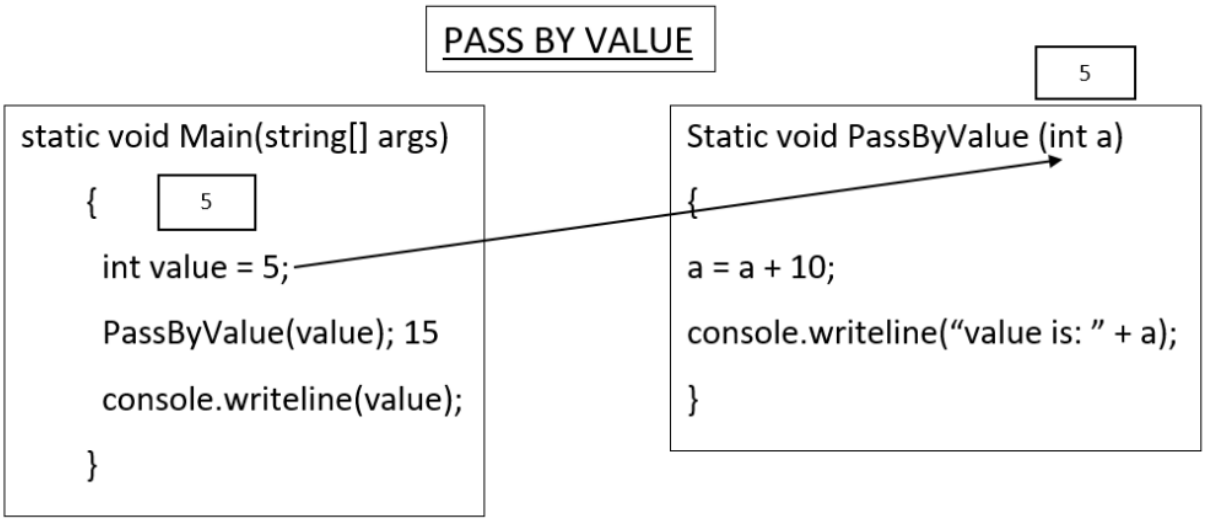
Pass By Reference (Ref Keyword)

→ The ref keyword causes arguments to be passed in a method by reference.
→ In call by reference, the called method changes the value of the parameters passed to it from the calling method.
→ Any changes made to the parameters in the called method will be reflected in the parameters passed from the calling method when control passes back to the calling method.
→ It is necessary that both the called method and the calling method must explicitly specify the **ref** keyword before the required parameters.
→ The variables passed by reference from the calling method must be first initialized.

Out Keyword

→ The out keyword is similar to the ref keyword and causes arguments to be passed by reference.
→ The only difference between the two is that the out keyword does not require the variables that are passed by reference to be initialized.
→ Both the called method and the calling method must explicitly use the **out** keyword.

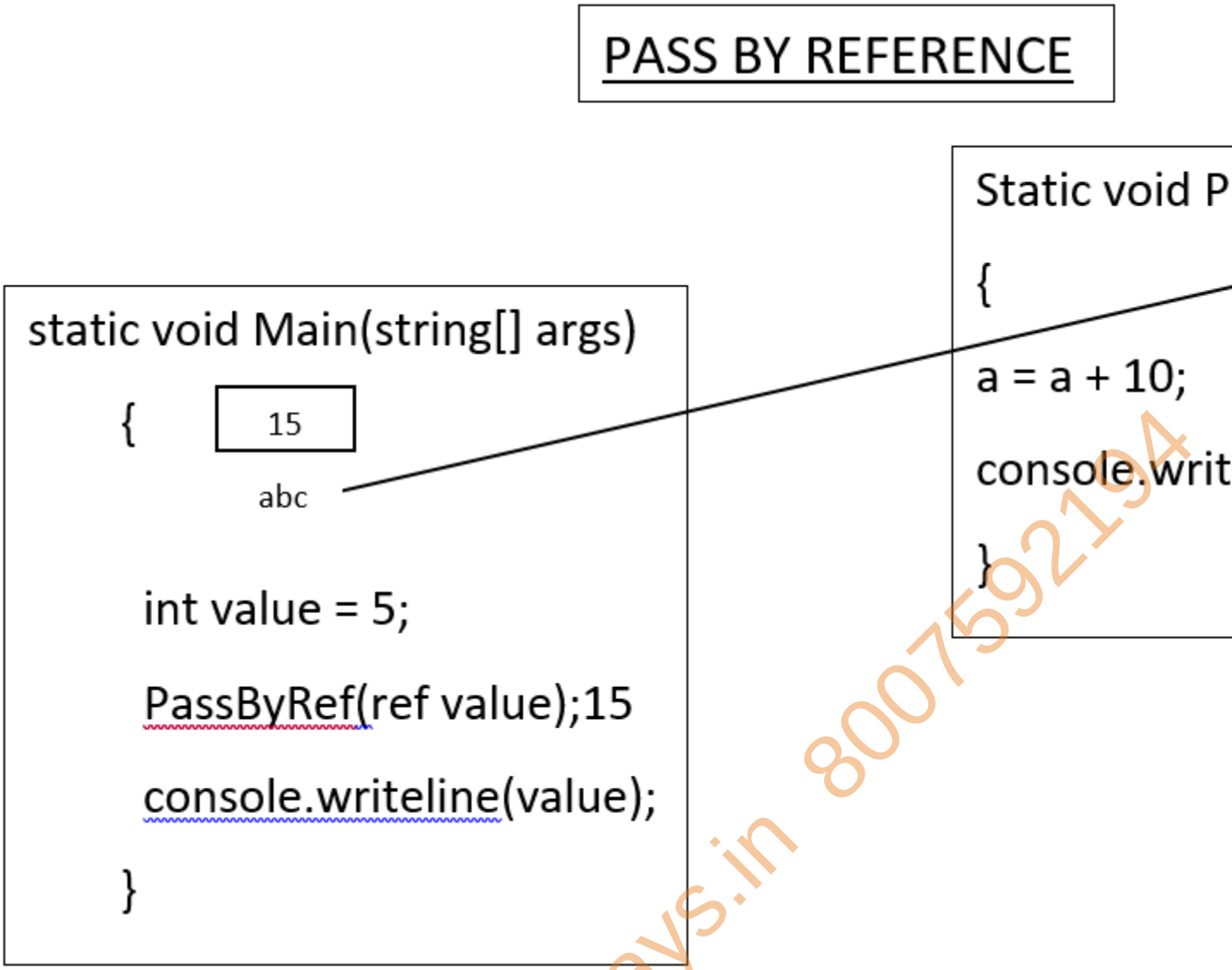
Pass By Value



Pass By Value

Pass By Reference (Ref Keyword)

- The ref keyword causes arguments to be passed in a method by reference.
- In call by reference, the called method changes the value of the parameters passed to it from the calling method.
- Any changes made to the parameters in the called method will be reflected in the parameters passed from the calling method when control passes back to the calling method.
- It is necessary that both the called method and the calling method must explicitly specify the ref keyword before the required parameters.
- The variables passed by reference from the calling method must be first initialized.



Pass By Reference

The following syntax is used to pass values by reference using the ref keyword.

```
<access_modifier><return_type><MethodName> (ref parameter1, ref parameter2,
parameter3, parameter4, ..parameterN)
{
// actions to be performed
}
```

where,

parameter 1…parameterN: Specifies that there can be any number of parameters and it is not necessary for all the parameters to be ref parameters.

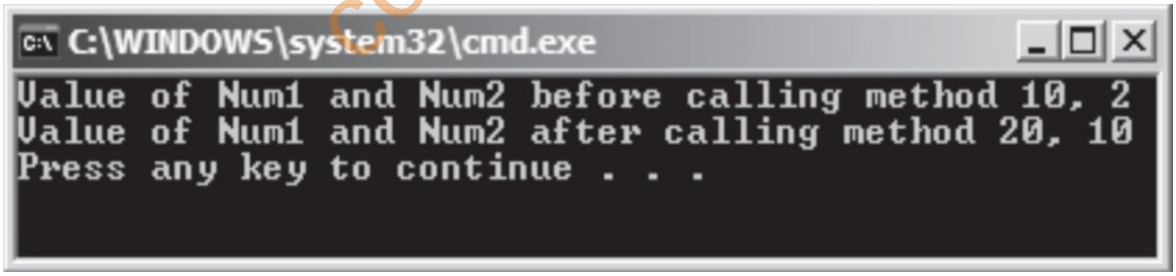
```
using System;
```

```
class RefParameters
{
static void Calculate(ref intnumValueOne, ref int
numValueTwo)
{
numValueOne = numValueOne * 2;
numValueTwo = numValueTwo / 2;
]
}
static void Main(string[] args)
{
intnumOne = 10;
intnumTwo = 20;
Console.WriteLine("Value of Num1 and Num2 before calling method " +numOne + ",
" + numTwo);
Calculate(ref numOne, ref numTwo);
Console.WriteLine("Value of Num1 and Num2 after calling method " +numOne + ",
" + numTwo);
}
}
```

In Above Code,

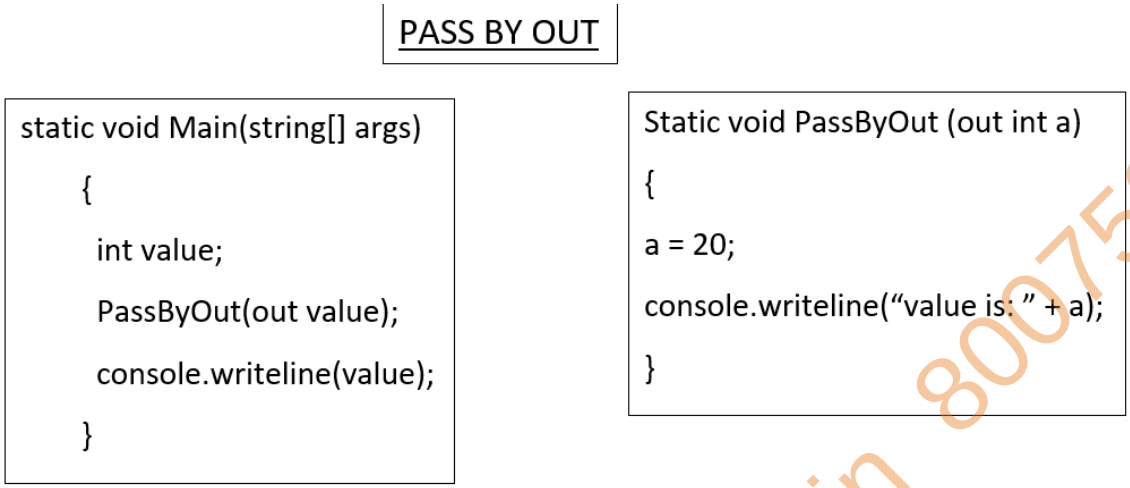
- The Calculate() method is called from the Main() method, which takes the parameters prefixed with the ref keyword.
- The same keyword is also used in the Calculate() method before the variables numValueOne and numValueTwo.
- In the Calculate() method, the multiplication and division operations are performed on the values passed as parameters and the results are stored in the numValueOne and numValueTwovariables respectively.
- The resultant values stored in these variables are also reflected in the numOne and numTwo variables respectively as the values are passed by reference to the method Calculate().

The following figure displays the use of ref keyword: Output



Out Keyword

- The out keyword is similar to the ref keyword and causes arguments to be passed by reference.
- The only difference between the two is that the out keyword does not require the variables that are passed by reference to be initialized.
- Both the called method and the calling method must explicitly use the out keyword.



Out Keyword C#

The following syntax is used to pass values by reference using the out keyword:

```
<access_modifier><return_type><MethodName> (out parameter1, out
parameter2, ...parameterN)
{
// actions to be performed
}
```

where,

parameter 1…parameterN: Specifies that there can be any number of parameters and it is not necessary for all the parameters to be
out parameters.

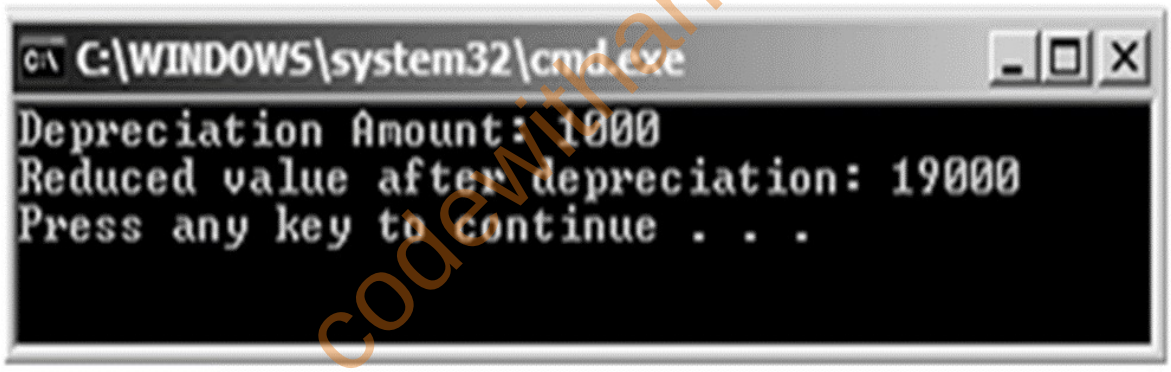
The following code uses the out keyword to pass the parameters by reference:

```
using System;
class OutParameters
{
static void Depreciation(out intval)
{
{
val = 20000;
intdep = val * 5/100;
intamt = val - dep;
Console.WriteLine("Depreciation Amount: " + dep);
Console.WriteLine("Reduced value after depreciation: " +
amt);
}
}
static void Main(string[] args)
{
int value;
Depreciation(out value);
}
}
```

In Above Code,

- the Depreciation() method is invoked from the Main() method passing the val parameter using the out keyword. In the Depreciation() method, the depreciation is calculated and the resultant depreciated amount is deducted from the val variable. The final value in the amt variable is displayed as the output.

The following figure shows the use of out keyword:



Source Code Of Pass By Value, Pass By Reference & Out Keywords

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

```
using System.Text;
using System.Threading.Tasks;

namespace REF_AND_OUT_KEYWORDS
{
    class Program
    {
        //public static void PassByValue(int a)
        //{
        //    a = a + 10;
        //    Console.WriteLine("Value is: {0}", a);
        //}

        //public static void PassByRef(ref int a)
        //{
        //    a = a + 10;
        //    Console.WriteLine("Value is: {0}", a);
        //}

        public static void PassByOut(out int a)
        {
            a = 20;
            Console.WriteLine("Value is: {0}", a);
        }

        static void Main(string[] args)
        {
            int value;
            PassByOut(out value); // 20
            Console.WriteLine(value); // 20
            Console.ReadLine();
        }
    }
}
```

Access Modifiers In C# Programming

Access Modifiers

C# provides you with access modifiers that allow you to specify which classes can access the data members of a particular class.

In C#, there are four commonly used access modifiers.

These are described as follows:

public: The public access modifier provides the most permissive access level.
The members declared as public can be accessed anywhere in the class as well as from other classes.

The following code declares a public string variable called Name to store the name of the person which can be publicly accessed by any other class:

```
class Employee
{
// No access restrictions.
public string Name = "Adil";
}
```

private: The private access modifier provides the least permissive access level. Private members are accessible only within the class in which they are declared.

Following code declares a variable called _salary as private, which means it cannot be accessed by any other class except for the Employee class.

```
class Employee
{
// Accessible only within the class
private float _salary;
}
```

protected:
The protected access modifier allows the class members to be accessible within the class as well as within the derived classes.

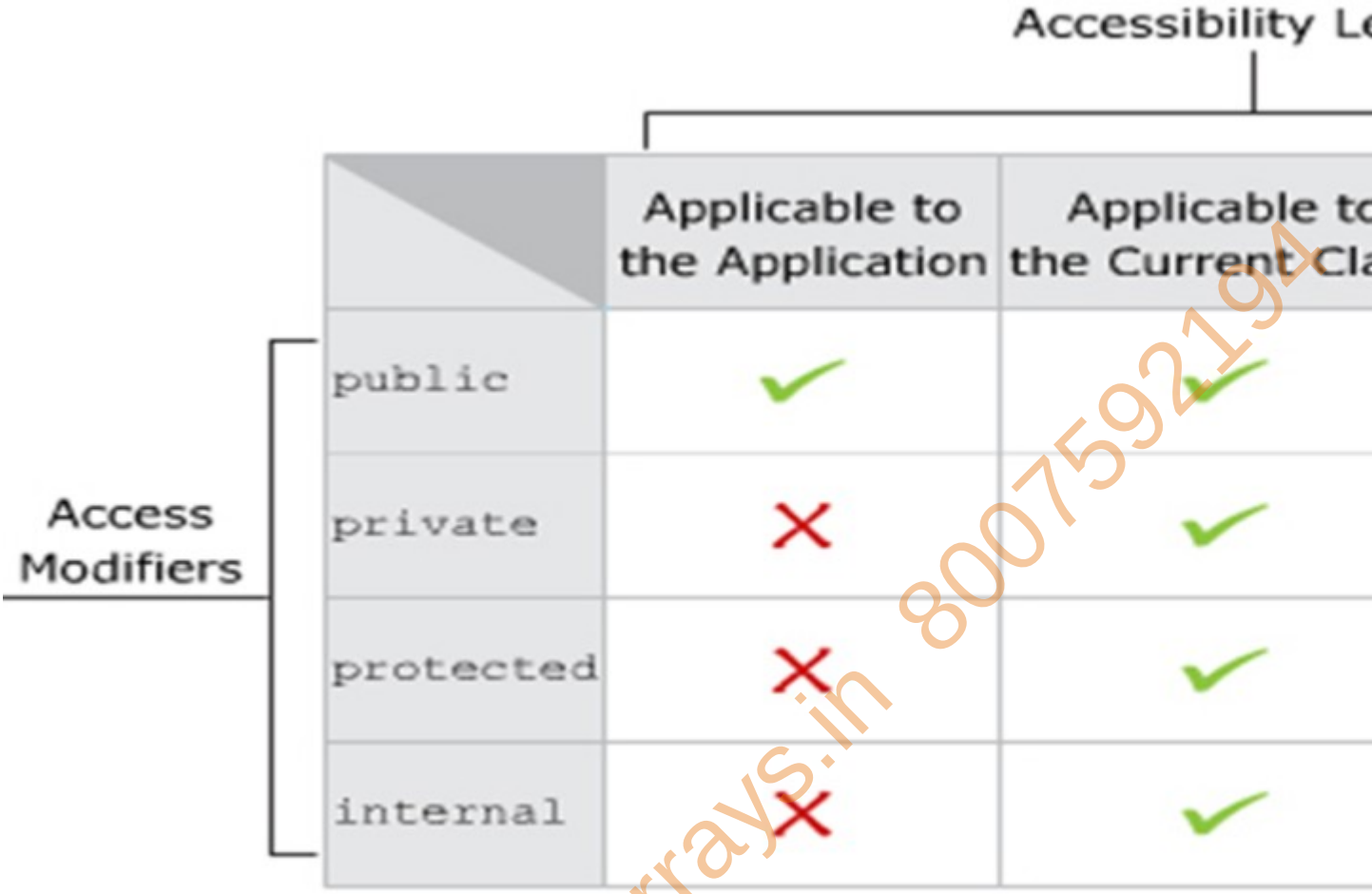
The following code declares a variable called Salary as protected, which means it can be accessed only by the Employee class and its derived classes:

```
class Employee
{
// Protected access
protected float Salary;
}
```

internal: The internal access modifier allows the class members to be accessible only within the classes of the same assembly. An assembly is a file that is automatically generated by the compiler upon successful compilation of a .NET application. The code declares a variable called NumOne as internal, which means it has only assembly-level access.

```
public class Sample
{
// Only accessible within the same assembly
internal static intNumOne = 3;
}
```

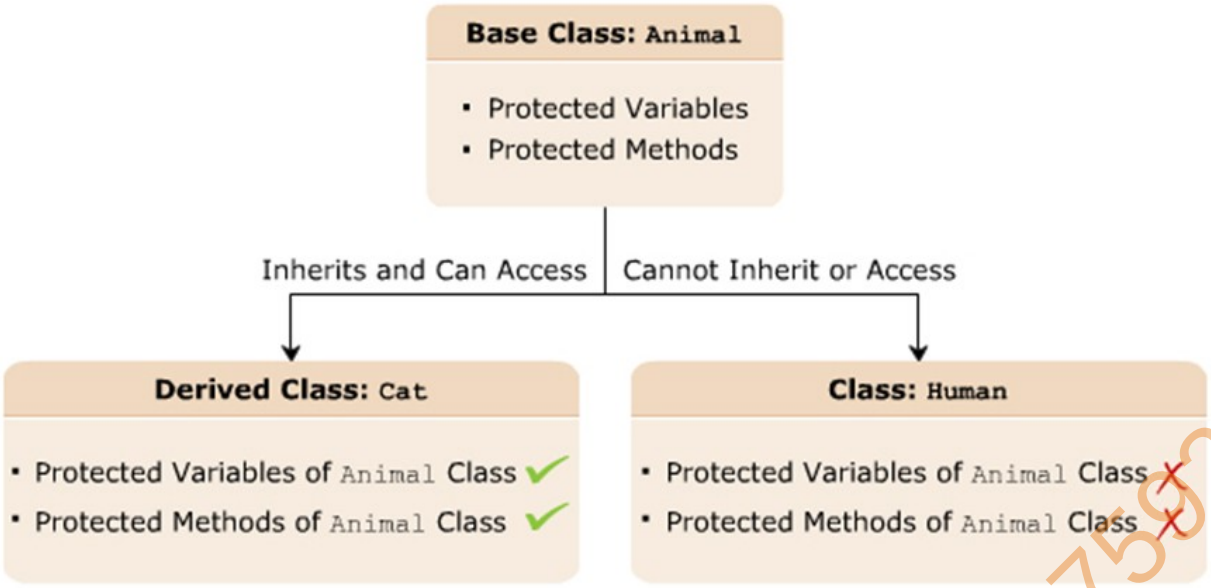
The following figure displays the various accessibility levels:



Protected Access Modifier

- The **protected** access modifier protects the data members that are declared using this modifier.
- The **protected** access modifier is specified using the protected keyword.
- Variables or methods that are declared as protected are accessed only by the class in which they are declared or by a class that is derived from this class.

The following figure displays an example of using the protected access modifier:



Protected Access Modifier

The following syntax declares a protected variable:

```
protected <data_type> <VariableName>;
```

where,
data_type: Is the data type of the data member.
VariableName: Is the name of the variable.

The following syntax declares a protected method:

```
protected <return_type> <MethodName>(argument_list);
```

where,
return_type: Is the type of value the method will return.
MethodName: Is the name of the method.
argument_list: Is the list of parameters.

The following code demonstrates the use of the protected access modifier:

```
class Animal
{
```

```
        protected string Food;
        protected string Activity;
}
class Cat:Animal
{
   static voidMain(String[]args)
   {
cat objCat=newCat();
objCat.Food="Mouse";
objCat.Activity="lazearound";
Console.WriteLine("The Cat loves to eat"+objCat.Food+".");
Console.WriteLine("The Catloves to"+objCat.Activity+".");
}
}
```

In Above code:

- Two variables are created in the class Animal with the protected keyword.
- The class Cat is inherited from the class Animal.
- The instance of the class Cat is created that is referring the two variables defined in the class Animal using the dot (.) operator.
- The protected access modifier allows the variables declared in the class Animal to be accessed by the derived class Cat.

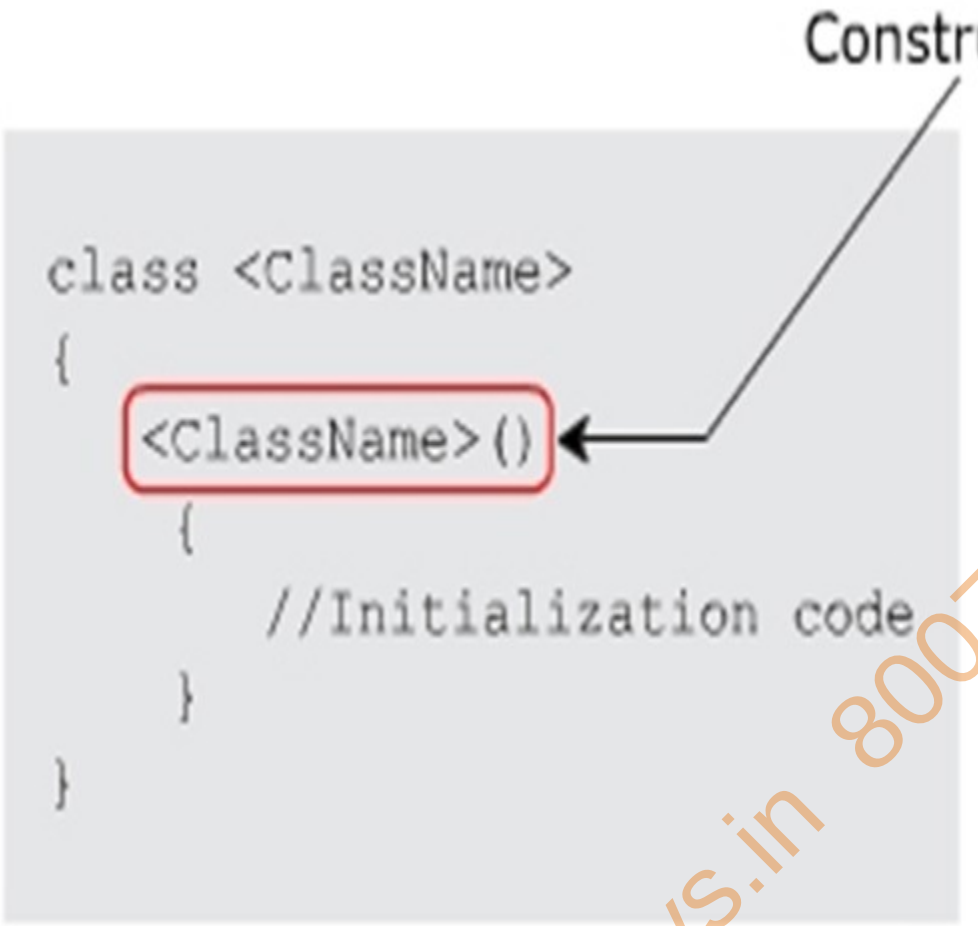
Output

```
Cat loves to eat Mouse.
The Cat loves to laze around
```

What Is Constructor In C# ?

- A Constructor is used to initialize objects.
- Constructors can initialize the variables of a class or perform startup operations only once when the object of the class is instantiated.
- A class constructor is a special member function of a class that is executed whenever we create new objects of that class.
- They are automatically executed whenever an instance of a class is created.
- A constructor has exactly the same name as that of class and it does not have any return type.
- A C# class can contain one or more special member functions having the same name as the class, called constructors.
- A constructor is a method having the same name as that of the class.

The following figure shows the constructor declaration:



It is possible to specify the accessibility level of constructors within an application by using access modifiers such as:

- public: Specifies that the constructor will be called whenever a class is instantiated. This instantiation can be done from anywhere and from any assembly.
- private: Specifies that this constructor cannot be invoked by an instance of a class.
- protected: Specifies that the base class will initialize on its own whenever its derived classes are created. Here, the class object can only be created in the derived classes.
- internal: Specifies that the constructor has its access limited to the current assembly. It cannot be accessed outside the assembly.

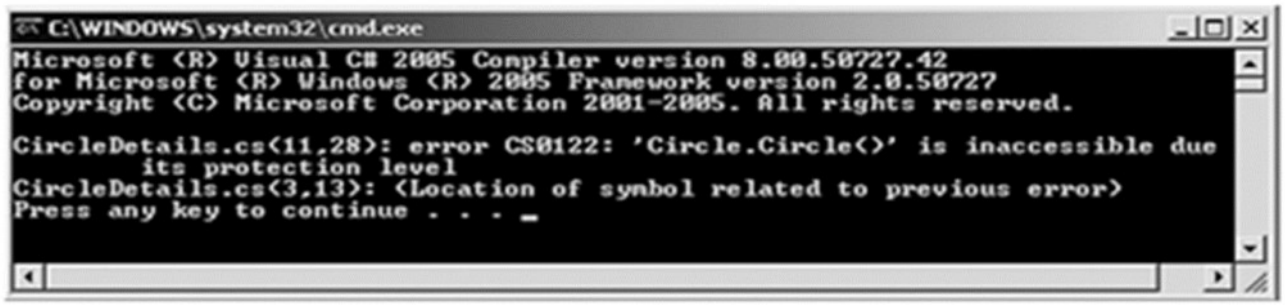
The following code creates a class Circle with a private constructor:

```
using System;
 public class Circle
{
private Circle()
{
}
}
class CircleDetails
{
public static void Main(string[] args)
{
Circle objCircle = new Circle();
}
}
```

In Above Code,

- The program will generate a compile-time error because an instance of the Circle class attempts to invoke the constructor which is declared as private. This is an illegal attempt.
- Private constructors are used to prevent class instantiation.
- If a class has defined only private constructors, the new keyword cannot be used to instantiate the object of the class.
- This means no other class can use the data members of the class that has only private constructors.
- Therefore, private constructors are only used if a class contains only static data members.
- This is because static members are invoked using the class name.

The following figure shows the output for creating a class Circle with a private constructor:



The following code used to initialize the values of _empName, _empAge, and _deptName with the help of a constructor:

```
using System;

class Employees
{
string _empName;
int _empAge;
string _deptName;
Employees(string name, intnum)
{
_empName = name;
_empAge = num;
_deptName = "Research & Development";
}
static void Main(string[] args)
{
Employees objEmp = new Employees("John", 10);
Console.WriteLine(objEmp._deptName);
}
}
```

In Above Code,

- A constructor is created for the class Employees. When the class is instantiated, the constructor is invoked with the parameters John and 10.
- These values are stored in the class variables empName and empAge respectively. The department of the employee is then displayed in the console window.

Source Code Of Constructor

```
using System;
```

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

namespace ConstructorsInCsharp
{
    class Employees
    {

        int EmpId;
        string EmpName;
        int EmpAge;

        public Employees(int EmpId, string EmpName, int EmpAge)
        {
            this.EmpId = EmpId;
            this.EmpName = EmpName;
            this.EmpAge = EmpAge;
        }

        public int getId()
        {
            return this.EmpId;
        }
        public string getName()
        {
            return this.EmpName;
        }
        public int getAge()
        {
            return this.EmpAge;
        }
        static void Main(string[] args)
        {
            Employees Ali = new Employees(11,"Ali Khan",22);
            Employees Zain = new Employees(12, "Zain Ali", 21);

            Console.WriteLine("Employee Id is {0}",Ali.getId());
            Console.WriteLine("Employee Name is {0}", Ali.getName());
            Console.WriteLine("Employee Age is {0}", Ali.getAge());
            Console.WriteLine("--------------------");
            Console.WriteLine("Employee Id is {0}", Zain.getId());
            Console.WriteLine("Employee Name is {0}", Zain.getName());
            Console.WriteLine("Employee Age is {0}", Zain.getAge());
            Console.ReadLine();

        }
    }
}
```

TYPES OF CONSTRUCTORS IN C#

Constructors generally following types:

- Default Constructor
- Parameterized constructor
- Private Constructor
- Static Constructor
- Copy Constructor

STATIC CONSTRUCTOR

→ A static constructor is used to initialize static variables of the class and to perform a particular action only once.
→ Static constructor is called only once, no matter how many objects you create.
→ Static constructor is called before instance constructor.
→ A static constructor does not take any parameters and does not use any access modifiers.

Key points of static constructor

→ Only one static constructor can be created in the class.
→ It is called automatically before the first instance of the class created.
→ We cannot call static constructor directly.

CONSTRUCTORS IN C#

→ A Constructor is used to initialize objects.
→ A class constructor is a special member function of a class that is executed whenever we create new objects of that class.
→ A constructor has exactly the same name as that of class and it does not have any return type.

Default Constructor

A constructor which has not defined any parameters or we can say without any parameters is called default constructor. It initializes the same value of every instance of class.

Parameterized Constructor

A constructor which has at least one parameter is called **Parameterized** Constructor. Using this type of constructor we can initialize each **instance** of the class to different values.

COPY CONSTRUCTOR IN C#

The constructor which creates an object by copying variables from another object is called a copy constructor. The purpose of a copy constructor is to initialize a new instance to the values of an existing instance.

In c#, Copy Constructor is a parameterized constructor which contains a parameter of same class type. The copy constructor in C# is useful whenever we want to initialize a new instance to the values of an existing instance.

In simple words, we can say copy constructor is a constructor which copies a data of one object into another object.

Private Constructor In C#

When a constructor is created with a private specifier, it is not possible for other classes to derive from this class, neither is it possible to create an instance of this class. They are usually used in classes that contain static members only. Some key points of a private constructor are:

- One use of a private constructor is when we have only static members.
- Once we provide a constructor that is either private or public or any, the compiler will not add the parameter-less public constructor to the class.
- In the presence of parameterless private constructor you cannot create a default constructor.
- We cannot inherit the class in which we have a private constructor.
- We can have parameters in private constructor. YES

Destructors In C# Programming

A destructor is a special method which has the same name as the class but starts with the character ~ before the class name and immediately de-allocates memory of objects that are no longer required.

Following are the features of destructors:

- Destructors cannot be overloaded or inherited.
- Destructors cannot be explicitly invoked.
- Destructors cannot specify access modifiers and cannot take parameters.

A destructor is a special method which has the same name as the class but starts with the character ~ before the class name and immediately de-allocates memory of objects that are no longer required.

A C# class can also have a destructor (only one is allowed per class), which is a special method and also has the same name as the class but prefixed with a special symbol ~.

A destructor of an object is executed when the object is no longer required in order to de-allocate memory of the object.

Following are the features of destructors:

- Destructors cannot be overloaded or inherited.
- Destructors cannot be explicitly invoked.
- Destructors cannot specify access modifiers and cannot take parameters.

The following code demonstrates the use of destructors:

```
using System;

class Employee
{
privateint _empId;
private string _empName;
privateint _age;
```

```
private double _salary;
Employee(int id, string name, int age, double sal)
{
Console.WriteLine("Constructor for Employee called");
_empId = id;
_empName = name;
_age = age;
_salary = sal;

 }
~Employee()
using System;

class Employee
{
privateint _empId;
private string _empName;
privateint _age;
private double _salary;
Employee(int id, string name, int age, double sal)
{
Console.WriteLine("Constructor for Employee called");

}
static void Main(string[] args)
{
Employee objEmp = new Employee(1, "John", 45, 35000);
Console.WriteLine("Employee ID: " + objEmp._empId);
Console.WriteLine("Employee Name: " + objEmp._empName);
Console.WriteLine("Age: " + objEmp._age);
Console.WriteLine("Salary: " + objEmp._salary);
}
}
```

In Above Code,

- The destructor ~Employee is created having the same name as that of the class and the constructor.
- The destructor is automatically called when the object objEmp is no longer needed to be used.
- However, you have no control on when the destructor is going to be executed.

Source Code Of Destructors In C#

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

namespace DESTRUCTORS
{
```

```
    class person
    {
        public string Name;
        public int Age;

        public person(string Name, int Age)
        {
            this.Name = Name;
            this.Age = Age;
        }

        public string getName()
        {
            return this.Name;
        }
        public int getAge()
        {
            return this.Age;
        }

        ~person()
        {
            Console.WriteLine("Destructor has been invoked !!");
        }

    }
    class Program
    {
        static void Main(string[] args)
        {
            person Ali = new person("Ali",22);
            person Anas = new person("Anas", 23);
            Console.WriteLine(Ali.getName());
            Console.WriteLine(Ali.getAge());
            Console.WriteLine("--------------");
            Console.WriteLine(Anas.getName());
            Console.WriteLine(Anas.getAge());
            //Console.ReadLine();

        }
    }
}
```

TYPES OF DELEGATES IN C#

→ MULTIPLE DELEGATES
→ SINGLE CAST DELEGATES
→ MULTI CAST DELEGATES

1. MULTIPLE DELEGATES:

In C#, a user can invoke multiple delegates within a single program. Depending on the delegate name or the type of parameters passed to the delegate, the appropriate delegate is invoked.

2. SINGLE CAST DELEGATES:

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

3. MULTI CAST DELEGATES:

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

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

We can use += and -= assignment operators to implement multi cast delegates.

What Are Properties In C#

→ Properties allow you to control the accessibility of a class variables, and are the recommended way to access variables from the outside in c#.

→ A property is much like a combination of a variable and a method - it can't take any parameters, but you are able to process the value before it's assigned to our returned.

→ Properties are like data fields (variables), but have logic behind them.

→ From the outside, they look like any other member variable.
- But they act like a member function.

→ Defined like a field, with “get” and “set” accessors code added.
→ Properties are also used for encapsulation.

Types Of Properties In C#

→ Read / write properties

→ Read only properties

→ Write only properties

→ Auto implemented properties

Properties In C#

- Properties allow you to control the accessibility of a class variables, and are the recommended way to access variables from the outside in c#.
- Properties in C# allow you to set and retrieve values of fields declared with any access modifier in a secured manner.
- A property is much like a combination of a variable and a method - it can't take any parameters, but you are able to process the value before it's assigned to our returned.
- Properties are like data fields (variables), but have logic behind them.
- From the outside, they look like any other member variable.
    - But they act like a member function.
- Defined like a field, with “get” and “set” accessors code added.
- Properties are also used for encapsulation.

Example

- Consider fields that store names and IDs of employees.
- You can create properties for these fields to ensure accuracy and validity of values stored in them.

Properties:

- Properties allow to protect a field in the class by reading and writing to the field through a property declaration.
- Properties allow to access private fields, which would otherwise be inaccessible.

- Properties can validate values before allowing you to change them and also perform specified actions on those changes.
- Properties ensure security of private data.
- Properties support abstraction and encapsulation by exposing only necessary actions and hiding their implementation.

Types Of Properties In C#

- Read / write properties
- Read only properties
- Write only properties
- Auto implemented properties

The following syntax is used to declare a property in C#.

```
<access_modifier><return_type><PropertyName>
{
   //body of the property
}
```

where,

- **access_modifier**: Defines the scope of access for the property, which can be private, public, protected, or internal.
- **return_type**: Determines the type of data the property will return.
- **PropertyName**: Is the name of the property.

Get And Set Accessors

Property accessors allow you to read and assign a value to a field by implementing get and set accessors as follows:

The Get Accessor

- The get accessor is used to read a value and is executed when the property name is referred.
- It does not take any parameter and returns a value that is of the return type of the property.

The Set Accessor

- The set accessor is used to assign a value and is executed when the property is assigned a new value using the equal to (=) operator.
- This value is stored in the private field by an implicit parameter called value (keyword in C#) used in the set accessor.

Syntax

```
<access_modifier><return_type>PropertyName>
{
get
{
   // return value
}
set
{
   // assign value
}
}
```

The following code demonstrates the use of the get and set accessors.

```
using System;
class SalaryDetails
{
private string _empName;
public string EmployeeName
{
get
{
return _empName;
}
```

```
set
{
_empName = value;
}
}
static void Main (string [] args)
{
SalaryDetails objSal = new SalaryDetails();
objSal.EmployeeName = "Patrick Johnson";
Console.WriteLine("Employee Name: " + objSal.EmployeeName);
}
}
```

In Above Code,

- The class SalaryDetails creates a private variable _empName and declares a property called EmployeeName.
- The instance of the SalaryDetails class, objSal, invokes the property EmployeeName using the dot (.) operator to initialize the value of employee name. This invokes the set accessor, where the value keyword assigns the value to _empName.
- The code displays the employee name by invoking the property name.
- This invokes the get accessor, which returns the assigned employee name. This invokes the set accessor, where the value keyword assigns the value to _empName.

Output

Employee Name: Patrick Johnson

Read-Only Property

The read-only property allows you to retrieve the value of a private field. To create a read-only property, you should define the get accessor.

The following syntax creates a read-only property.

```
<access_modifier><return_type><PropertyName>{
get
{
// return value
}
}
```

The following code demonstrates how to create a read-only property.

```
using System;
class Books {
string _bookName;
long _bookID;
public Books(string name, int value){
_bookName = name;
_bookID = value;
}
public string BookName {
get{ return _bookName; }
}
public long BookID {
get { return _bookID; }
}
}
class BookStore {
static void Main(string[] args) {
Books objBook = new Books("Learn C# in 21 Days", 10015);
Console.WriteLine("Book Name: " + objBook.BookName);
Console.WriteLine("Book ID: " + objBook.BookID);
}
}
```

In Above Code,

- The Books class creates two read-only properties that returns the name and ID of the book.
- The class BookStore defines a Main() method that creates an instance of the class Books by passing the parameter values that refer to the name and ID of the book.
- The output displays the name and ID of the book by invoking the get accessor of the appropriate read-only properties.

Output

Book Name: Learn C# in 21 Days
Book ID: 10015

Write-Only Property

- The write-only property allows you to change the value of a private field.
- To create a write-only property, you should define the set accessor.

The following syntax creates a write-only property.

```
<access_modifer><return_type><PropertyName>
{
set
{
// assign value
}
}
```

The following code demonstrates how to create a write-only property.

```
using System;
class Department {
string _deptName;
int _deptID;
public string DeptName{
set { _deptName = value; }
}
public intDeptID {
set { _deptID = value; }
}
public void Display() {
Console.WriteLine("Department Name: " + _deptName);
Console.WriteLine("Department ID: " + _deptID);
}
}
class Company {
static void Main(string[] args) {
Department objDepartment = new Department();
objDepartment.DeptID = 201;
objDepartment.DeptName = "Sales";
objDepartment.Display();
```

```
}
}
```

In Above Code,

- The Department class consists of two write-only properties.
- The Main() method of the class Company instantiates the class Department and this instance invokes the set accessor of the appropriate write-only properties to assign the department name and its ID.
- The Display() method of the class Department displays the name and ID of the department.

Output

Department Name: Sales
Department ID: 201

Read-Write Property

- The read-write property allows you to set and retrieve the value of a private field. To create a read-write property, you should define the set and get accessors.

The following syntax creates a read-write property.

```
<access_modifer><return type><PropertyName>
{
     get
{
// return value
}
set
{
// assign value
}
}
```

The following code demonstrates how to create a read-write property.

```
using System;
class Product {
string _productName;
int _productID;
float _price;
public Product(string name, intval) {
_productName = name;
_productID = val;
}
public float Price {
get { return _price;}
set { if (value < 0) { _price = 0; }
else { _price = value; }
}
}
public void Display() {
Console.WriteLine("Product Name: " + _productName);
Console.WriteLine("Product ID: " + _productID);
Console.WriteLine("Price: " + _price + "$");
}
}
class Goods {
static void Main(string[] args) {
Product objProduct = new Product("Hard Disk", 101);
objProduct.Price = 345.25F;
objProduct.Display();
}
}
```

In Above Code,

- The class Product creates a read-write property Price that assigns and retrieves the price of the product based on the if statement.
- The Goods class defines the Main()method that creates an instance of the class Product by passing the values as parameters that are name and ID of the product.
- The Display() method of the class Product is invoked that displays the name, ID, and price of the product.

Output

Product Name: Hard Disk
Product ID: 101
Price: 345.25$

Properties can be further classified as static, abstract, and boolean properties.

Static Properties

- The static property is used to access and manipulate static fields of a class in a safe manner.
- The static property declared by using the static keyword.
- The static property accessed using the class name and thus, belongs to the class rather than just an instance of the class.
- The static property called by a programmer without creating an instance of the class.
- We cannot initialize instance fields within static property.
- The static property accessed using the class name and thus, belongs to the class rather than just an instance of the class.
- The static property is used by a programmer without creating an instance of the class.
- The static property is used to access and manipulate static fields of a class in a safe manner.

The following code demonstrates a class with a static property.

```
using System;
class University
{
private static string _department;
private static string _universityName;
public static string Department
{
get
{
return _department;
```

```
}
set
{
_department = value;
}
}
public static string UniversityName
{
get { return _universityName; }
set { _universityName = value; }
}
}
class Physics
{
static void Main(string[] args)
{
University.UniversityName = "University of Maryland";
University.Department = "Physics";
Console.WriteLine("University Name: " + University.UniversityName);
Console.WriteLine("Department name: " + University.Department);
}
}
```

In Above Code,

- The class University defines two static properties UniversityName and DepartmentName.
- The Main() method of the class Physics invokes the static properties UniversityName and DepartmentName of the class University by using the dot (.) operator.
- This initializes the static fields of the class by invoking the set accessor of the appropriate properties.
- The code displays the name of the university and the department by invoking the get accessor of the appropriate properties.

Output

University Name: University of Maryland
Department name: Physics

Abstract Properties

- The word Abstract means incomplete, which means no implementation in programming.
- This is a duty of child class to implement an abstract member of their parent class.
- Abstract property declared by using the abstract keyword.
- Abstract properties contain only the declaration of the property without the body of the get and set accessors (which do not contain any statements and can be implemented in the derived class).
- Abstract properties are only allowed in an abstract class.

Abstract Properties are used:

- when it is required to secure data within multiple fields of the derived class of the abstract class.
- to avoid redefining properties by reusing the existing properties.

The following code demonstrates a class that uses an abstract property.

```
using System;
public abstract class Figure {
public abstract float DimensionOne {
     set;
}
public abstract float DimensionTwo {
set; }
}
class Rectangle : Figure {
float _dimensionOne;

float _dimensionTwo;
public override float DimensionOne {
set {
if (value <= 0){
_dimensionOne = 0;
}
else {
_dimensionOne = value;
}
}
```

```
}
public override float DimensionTwo {
set {
if (value <= 0)
{
_dimensionTwo = 0;
}
else {
_dimensionTwo = value;
}
}
}
float Area() {
return _dimensionOne * _dimensionTwo;
}
static void Main(string[] args) {
Rectangle objRectangle = new Rectangle();
objRectangle.DimensionOne = 20;
objRectangle.DimensionTwo = 4.233F;
Console.WriteLine("Area of Rectangle: " + objRectangle.Area());
}
}
```

In Above Code,

- The abstract class Figure declares two write-only abstract properties, DimensionOne and DimensionTwo.
- The class Rectangle inherits the abstract class Figure and overrides the two abstract properties DimensionOne and DimensionTwo by setting appropriate dimension values for the rectangle.
- The Area() method calculates the area of the rectangle.
- The Main() method creates an instance of the derived class Rectangle.
- This instance invokes the properties, DimensionOne and DimensionTwo, which, in turn, invokes the set accessor of appropriate properties to assign appropriate dimension values.
- The code displays the area of the rectangle by invoking the Area() method of the Rectangle class.

Output

Area of Rectangle: 84.66

Boolean Properties

- A boolean property is declared by specifying the data type of the property as bool.
- Unlike other properties, the boolean property produces only true or false values.
- While working with boolean property, a programmer needs to be sure that the get accessor returns the boolean value.

Source Code Of Properties

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

namespace PROPERTIES
{
    class Student
    {
        public string firstName { get; private set; }
        public string lastName { get; private set; }

        public Student(string fname, string lname)
        {
            firstName = fname;
            lastName = lname;
        }

    }

    class Program
    {
        static void Main(string[] args)
        {
            Student s = new Student("Adil","Mehmood");

            Console.WriteLine(s.firstName);
            Console.WriteLine(s.lastName);

            //s.StdId = 23;
            //s.Name = "Adil";
```

```
            //s.FName = "Mehmood";
            //Console.WriteLine(s.StdId);
            //Console.WriteLine(s.Name);
            //Console.WriteLine(s.FName);
            Console.ReadLine();
        }
    }
}
```

Source Code Of Abstract Properties

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

namespace Abstract_Properties
{
    abstract class person
    {
        public abstract uint Id { get; set; }
        public abstract string Name { get; set; }
    }

    class student : person
    {
        uint StudentId;
        string StudentName;

        public override uint Id
        {
            set
            {
                if (value == 0)
                {
                    Console.WriteLine("Id cannot be Zero !!");
                }
                else
                {
                    this.StudentId = value;
                }

            }
            get
            {
                return this.StudentId;
            }

        }

        public override string Name
        {
            set
            {
```

```
                if(string.IsNullOrEmpty(value))
                {
                    Console.WriteLine("Name cannot be null or empty !!");
                }
                else
                {
                    this.StudentName = value;
                }

            }
            get
            {
                return this.StudentName;
            }

        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            student Anas = new student();
            Anas.Id = 22;
            Anas.Name = "Anas Qureshi";
            Console.WriteLine(Anas.Id);
            Console.WriteLine(Anas.Name);
            Console.ReadLine();
        }
    }
}
```

Source Code Of Static Properties

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

namespace STATIC_PROPERTY
{
    class university
    {
        private static string UniversityName;
        private static string DepartmentName;

        public static string _UniversityName
        {
            set
            {
                if (string.IsNullOrEmpty(value))
                {
                    Console.WriteLine("You cannot enter null or empty value in
University NAme !!");
```

```
                }
                else
                {
                    UniversityName = value;
                }
            }

            get
            {
                return UniversityName;
            }


        }

        public static string _DepartmentName {

            set
            {
                if(string.IsNullOrEmpty(value))
                {
                    Console.WriteLine("You are not allowed to insert null or
empty value in department Name !!");
                }
                else
                {
                    DepartmentName = value;
                }
            }
            get
            {
                return DepartmentName;
            }

        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            university._UniversityName = "Mehran University Jamshoro";
            university._DepartmentName = "Software Engineering";
            Console.WriteLine("University Name is:
{0}",university._UniversityName);
            Console.WriteLine("University Department Name is: {0}",
university._DepartmentName);
            Console.ReadLine();
        }
    }
}
```

STATIC PROPERTY IN C#

- The static property is used to access and manipulate static fields of a class in a safe manner.
- The static property declared by using the **static** keyword.
- The static property accessed using the class name and thus, belongs to the class rather than just an instance of the class.
- The static property called by a programmer without creating an instance of the class.
- We cannot initialize instance fields within static property.

STATIC CONSTRUCTOR IN C#

→ A static constructor is used to initialize static variables of the class and to perform a particular action only once.

→ Static constructor is called only once, no matter how many objects you create.

→ Static constructor is called before instance (default or parameterized) constructor.

→ A static constructor does not take any parameters and does not use any access modifiers.

KEY POINTS OF STATIC CONSTRUCTOR

→ Only one static constructor can be created in the class.

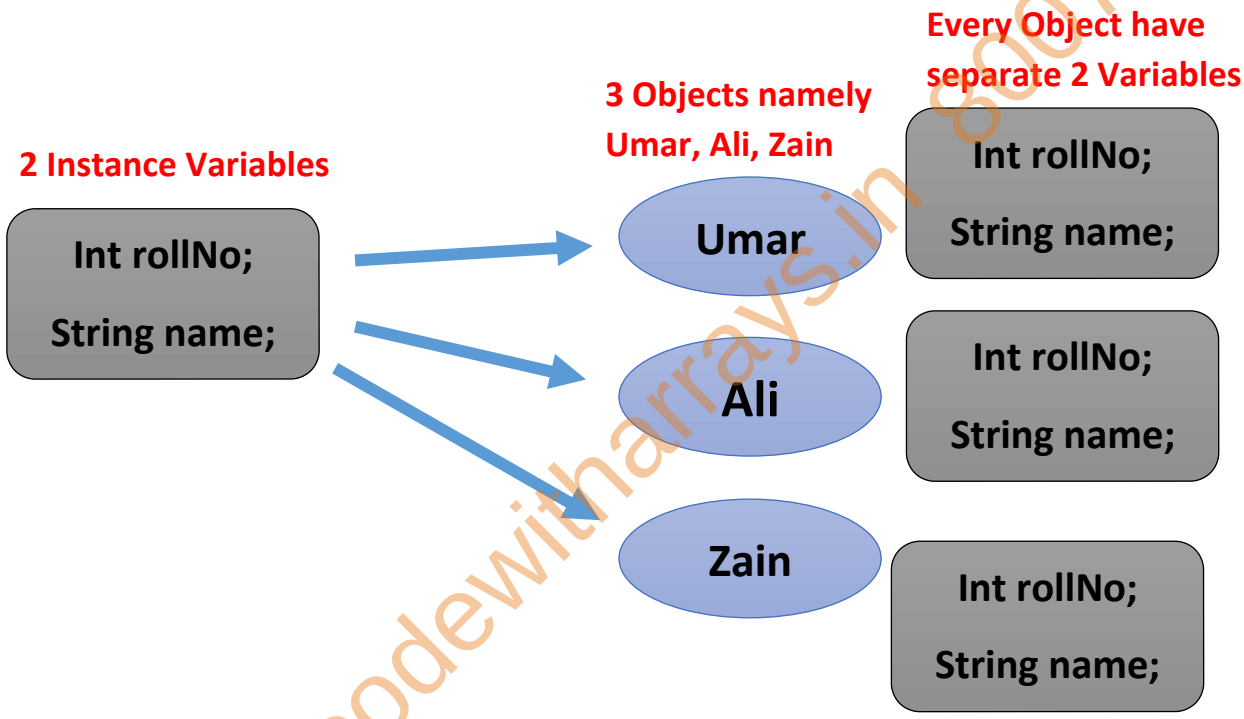
→ It is called automatically before the first instance of the class created.

→ We cannot call static constructor directly. CLR (COMMON LANGUAGE RUNTIME)

STATIC AND INSTANCE MEMBERS OF CLASS IN C#

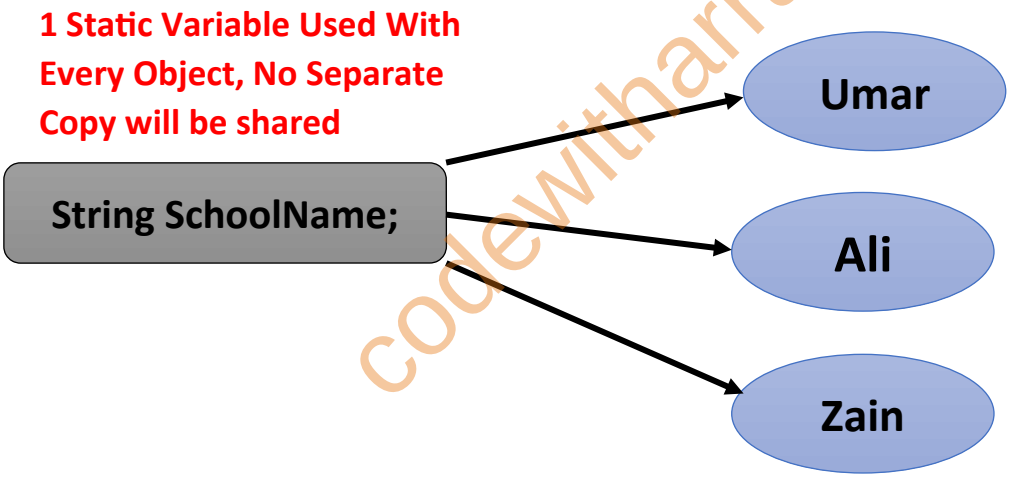
Instance Member:

→ Instance member have a separate copy for each and every object of the class.
→ Instance member belongs to the objects of the class.
→ When no static keyword is present the class member is called non-static or instance member.
→ Instance or non-static members are invoked using objects of the class.
→ "this" keyword is used with instance members not with static members.
→ Instance means "Example", so object have different names in programming like instance and example.



Static Member:

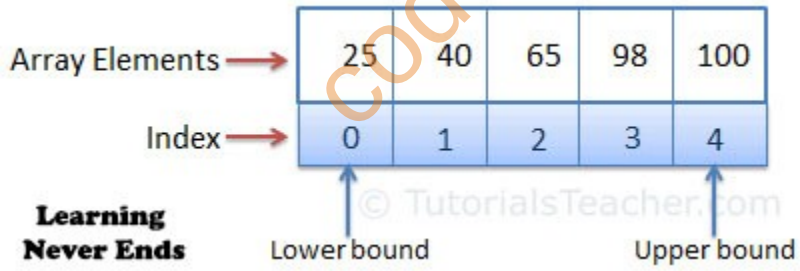
→ Static member belongs to the class.
→ We can define class members as static using the static keyword.
→ When we declare a member of a class as static, it means no matter how many objects of the class are created, there is only one copy of the static member.
→ Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it.
→ Static variables can be initialized outside the member function or class definition.
→ You can also initialize static variables inside the class definition.
→ You can also declare a member function as static.
→ Such functions can access only static variables.
→ Static member are invoked using class name.



Note: class members can be fields, methods, properties, events, indexers, constructors.

Arrays In C#

- An array is a list of items or collection of elements which is of same or similar data type (homogenous elements).
- An array is a collection of elements of a single data type stored in adjacent memory locations.
- An array is a collection of related values placed in contiguous memory locations and these values are referenced using a common array name.
- An array simplifies the task of maintaining these values
- An array always stores values of a single data type.
- Each value is referred to as an **element**.
- These elements are accessed using **subscripts** or **index numbers** that determine the position of the element in the array list.
- C# supports **zero-based index** values in an array.
- This means that the first array element has an index number zero while the last element has an index number n-1, where n stands for the total number of elements in the array.
- This arrangement of storing values helps in efficient storage of data, easy sorting of data, and easy tracking of the data length.



Declaring Arrays

Arrays are reference type variables whose creation involves two steps:

Declaration

- An array declaration specifies the **type of data** that it can hold and an **identifier**.
- This identifier is basically an **array name** and is used with a **subscript / index** to retrieve or set the data value at that location.

Memory allocation:

- Declaring an array does not allocate memory to the array.

Following is the syntax for declaring an array:

```
data_type[] arrayName;
```

In Above syntax:

- **data_type:** Specifies the data type of the array elements (for example, int and char).
- **arrayName**: Specifies the name of the array.

Initializing Arrays

An array can be:

- Created using the new keyword and then initialized.
- Initialized at the time of declaration itself, in which case the new keyword is not used.

Creating and initializing an array with the new keyword involves specifying the size of an array.

The number of elements stored in an array depends upon the specified size.

The new keyword allocates memory to the array and values can then be assigned to the array.

If the elements are not explicitly assigned, default values are stored in the array.

The following table lists the default values for some of the widely used data types:

DATA TYPES	DEFAULT VALUES
int	0
float	0.0
double	0.0
char	'\0'
string	Null

The following syntax is used to create an array:

```
type[] arrayName = new type[size-value];
```

In Above Syntax,

- size-value: Specifies the number of elements in the array. You can specify a variable of type int that stores the size of the array instead of directly specifying a value.

The following code creates an integer array which can have a maximum of five elements in it:

```
public int[] number = new int[5];

number[0] = 11;
number[1] = 22;
number[2] = 33;
number[3] = 44;
number[4] = 55;
```

The following syntax is used to create and initialize an array without using the new keyword:

```
type[] arrayIdentifier = {val1, val2, val3, ..., valN};
```

In Above Syntax,

- **val1**: It is the value of the first element.
- **valN**: It is the value of the nth element.

Example:

```
public string[] studNames = new string{“Allan”, “Wilson”, “James”, “Arnold”};
```

Using The Foreach Loop For Arrays

The foreach loop:

- In C# is an extension of the for loop.

- Is used to perform specific actions on large data collections and can even be used on arrays.
- Reads every element in the specified array.
- Allows you to execute a block of code for each element in the array.
- Is particularly useful for reference types, such as strings.

Source Code - Example 1

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;


namespace ArraysDemo

{

    class Program

    {

        static void Main(string[] args)

        {


            //string[] myArray = { "Adil", "Zubia", "Sameed", "Ammad" };

            //Console.WriteLine(myArray.Length);

            //for (int i = 0; i < myArray.Length; i++)

            //{

            //    Console.WriteLine(myArray[i]);

            //}
```

```
            //foreach (string name in myArray)
            //{
            //    Console.WriteLine(name);
            //}

            //Console.WriteLine("Foreach Loop Ends..");

            //int[] myArray = new int[] { 10,20,30,40 };

            //int[] myArray = new int[4];

            //myArray[0] = 10;
            //myArray[1] = 20;
            //myArray[2] = 30;
            //myArray[3] = 40;

            //Console.WriteLine(myArray[0]);
            //Console.WriteLine(myArray[1]);
            //Console.WriteLine(myArray[2]);
            //Console.WriteLine(myArray[3]);
            Console.ReadLine();
        }
    }
}
```

Source Code - Example 2

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


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

            Console.WriteLine("How many names you want to store ??");
            int size = int.Parse(Console.ReadLine()); // 5


            string[] names = new string[size];


            for (int i = 0; i < size; i++)
            {
                Console.WriteLine("Enter Data: ");
                string name = Console.ReadLine();
                names[i] = name;
```

```
            }


            Console.WriteLine("----------------");


            foreach (string name in names)

            {

                Console.WriteLine(name);

            }


            //string[] names = new string[] {"Adil","Ali","Anas","Amir" };


            //string[] names = {"Adil", "Zubia", "Anum", "Farah","Abdul
Rehman","Sameed","Ammad" };


            //foreach(string i in names)

            //{

            //    Console.WriteLine(i);

            //}


            //Console.WriteLine("Foreach loop terminates..");

            //for(int i = 0; i < names.Length; i++)

            //{

            //    Console.WriteLine(names[i]);

            //}



            // 2 types of arrays
```

```
// single dimensional array
// multi dimensional array


//string[] names = new string[3];
//names[0] = "Abdul Rehman";
//names[1] = "Zubia";
//names[2] = "Saad";


//Console.WriteLine(names[0]);
//Console.WriteLine(names[1]);
//Console.WriteLine(names[2]);


// 4 types of loops
// for loop
// while loop
// do while loop
// foreach loop - arrays




//int[] nums = new int[3];
//nums[0] = 11;
//nums[1] = 22;
//nums[2] = 33;
//nums[3] = 44;
```

```
            //Console.WriteLine(nums[2]);


            //comment - ctrl + k, ctrl + c

            //uncomment - ctrl + k, ctrl + u

            Console.ReadLine();

        }

    }

}
```

CLASSES AND OBJECTS IN C#

→ C# programs are composed of classes that represent the entities of the program which also include code to instantiate the classes as objects.

→ When the program runs, objects are created for the classes and they may interact with each other to provide the functionalities of the program.

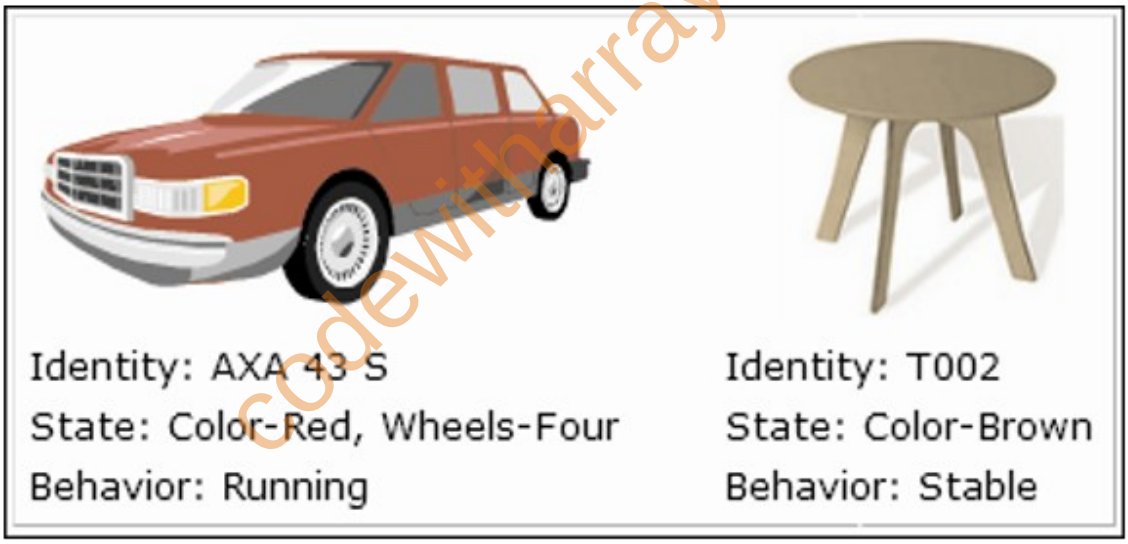
→ An object is a tangible entity such as a car, a table, or a briefcase. Every object has some characteristics and is capable of performing certain actions.

→ The concept of objects in the real world can also be extended to the programming world. An object in a programming language has a unique identity, state, and behavior.

→ The state of the object refers to its characteristics or attributes whereas the behavior of the object comprises its actions.

→ An object has various features that can describe it which could be the company name, model, price, mileage, and so on.

- The following figure shows an example of objects:



- An object stores its identity and state in fields (also called variables) and exposes its behavior through methods.

→ The concept of classes in the real world can be extended to the programming world, similar to the concept of objects.

→ In object-oriented programming languages like C#, a class is a template or blueprint which defines the state and behavior of all objects belonging to that class.

→ A class comprises fields, properties, methods, and so on, collectively called data members of the class. In C#, the class declaration starts with the **class** keyword followed by the **name of the class**.

ABSTRACT CLASS AND ABSTRACT METHODS IN C#

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain *abstract methods*, i.e., methods without body (public void get();)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- Abstract class is used only as a base class.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.
- Like any other class, an abstract class can contain fields that describe the characteristics and methods that describe the actions that a class can perform.

ABSTRACT METHODS

→ An abstract method contains a method signature, but no method body.

→ Instead of curly braces, an abstract method will have a semi colon (;) at the end.

→ Abstract keyword is used to declare the method as abstract.

→ You have to place the abstract keyword before the method name in the method declaration.

→ If a class has any abstract method, whether declared or inherited, the entire class must be declared as abstract.

→ If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.

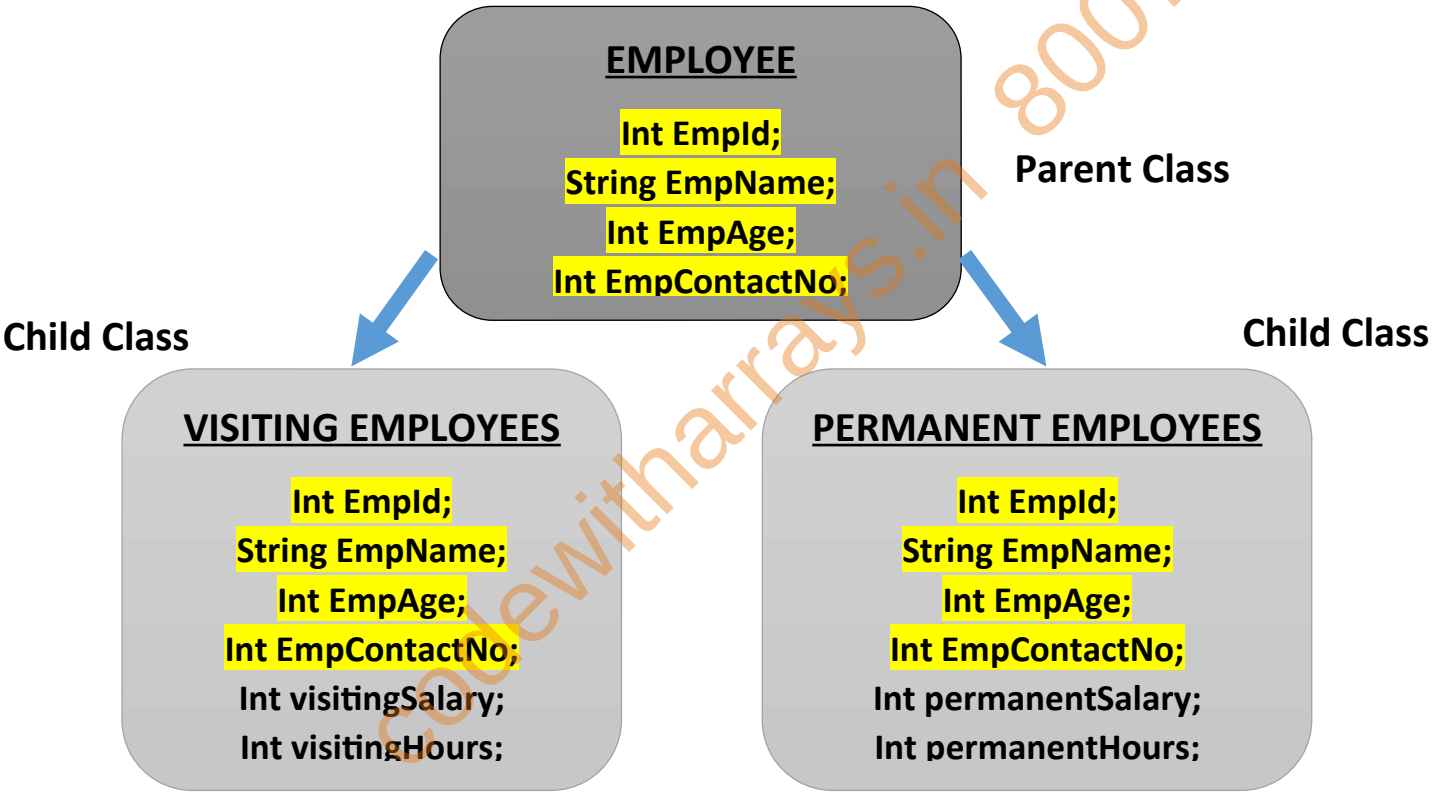
Inheritance In C# Programming

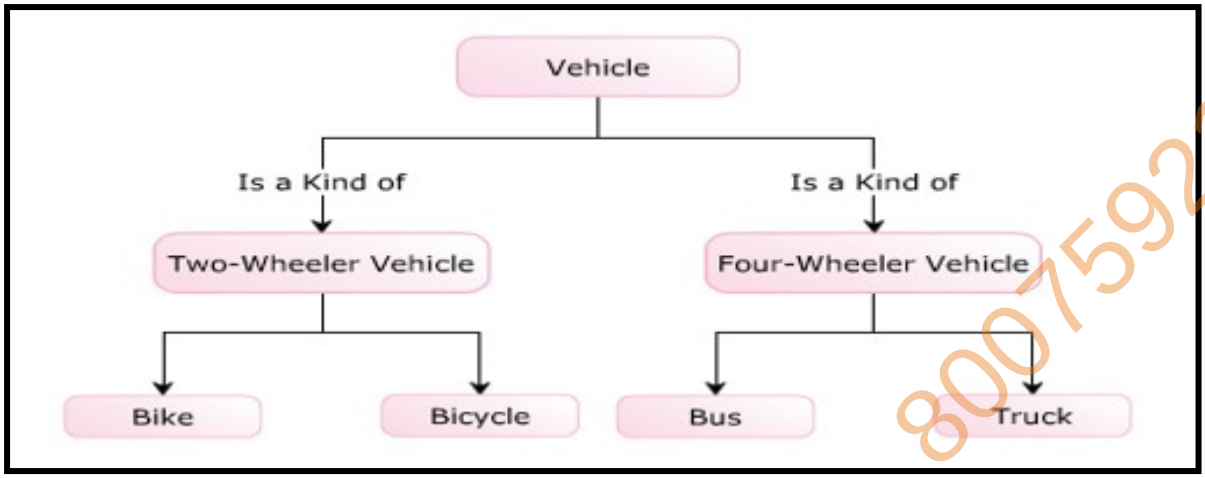
→ The similarity in physical features of a child to that of its parents is due to the child having inherited these features from its parents.

→ Similarly, in C#, inheritance allows you to create a class by deriving the common attributes and methods of an existing class.

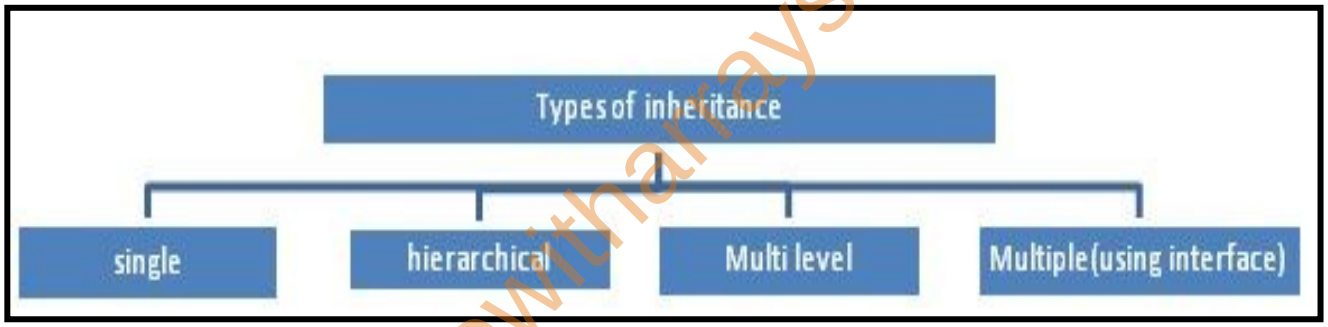
→ Inheritance provides reusability by allowing us to extend an existing class.

→ The reason behind OOP programming is to promote the reusability of code and to reduce complexity in code and it is possible by using inheritance.





→ The following are the types of inheritance in C#.



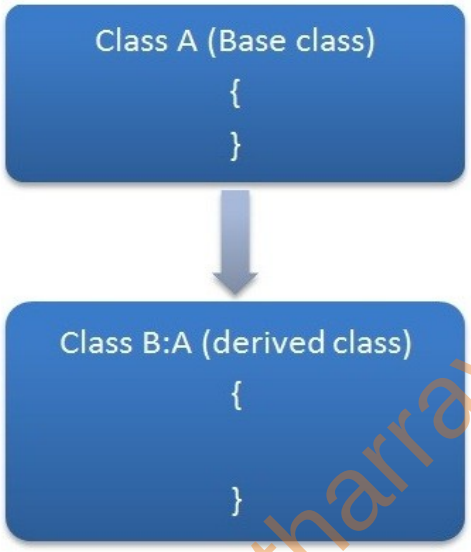
The inheritance concept is based on a base class and derived class. Let us see the definition of a base and derived class.

→ **BASE CLASS** - is the class from which features are to be inherited into another class.

→ **DERIVED CLASS** - it is the class in which the base class features are inherited.

Single Inheritance

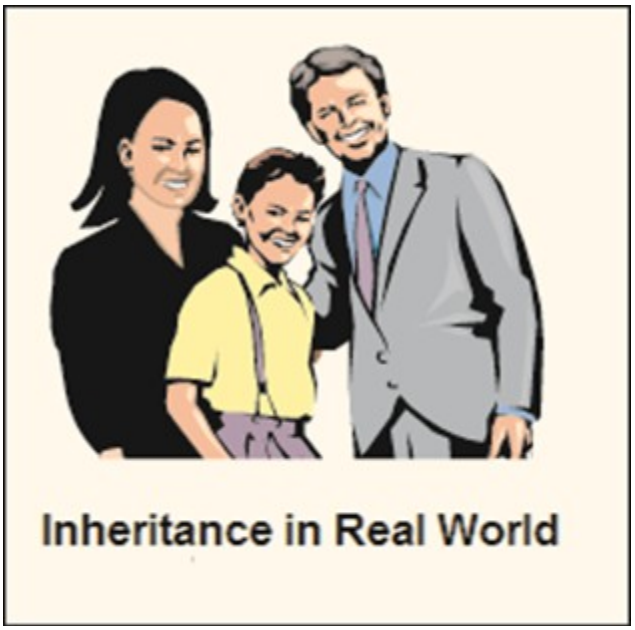
It is the type of inheritance in which there is one base class and one derived class.

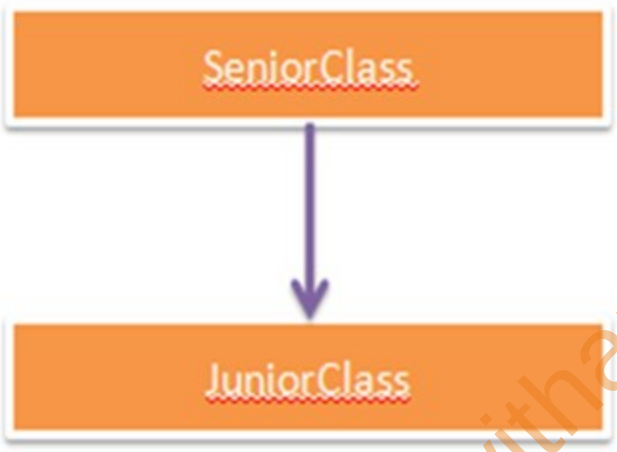


Definition of Inheritance

- The similarity in physical features of a child to that of its parents is due to the child having inherited these features from its parents.
- Similarly, in C#, inheritance allows you to create a class by deriving the common attributes and methods of an existing class.
- The class from which the new class is created is known as the base class and the created class is known as the derived class.

- The process of creating a new class by extending some features of an existing class is known as inheritance.

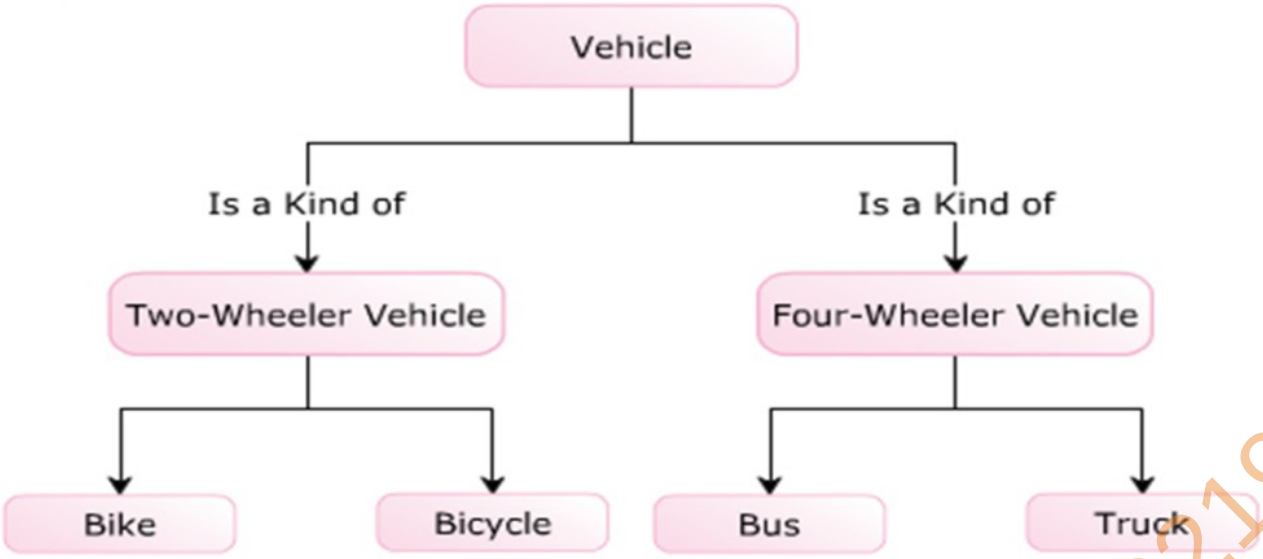




Example

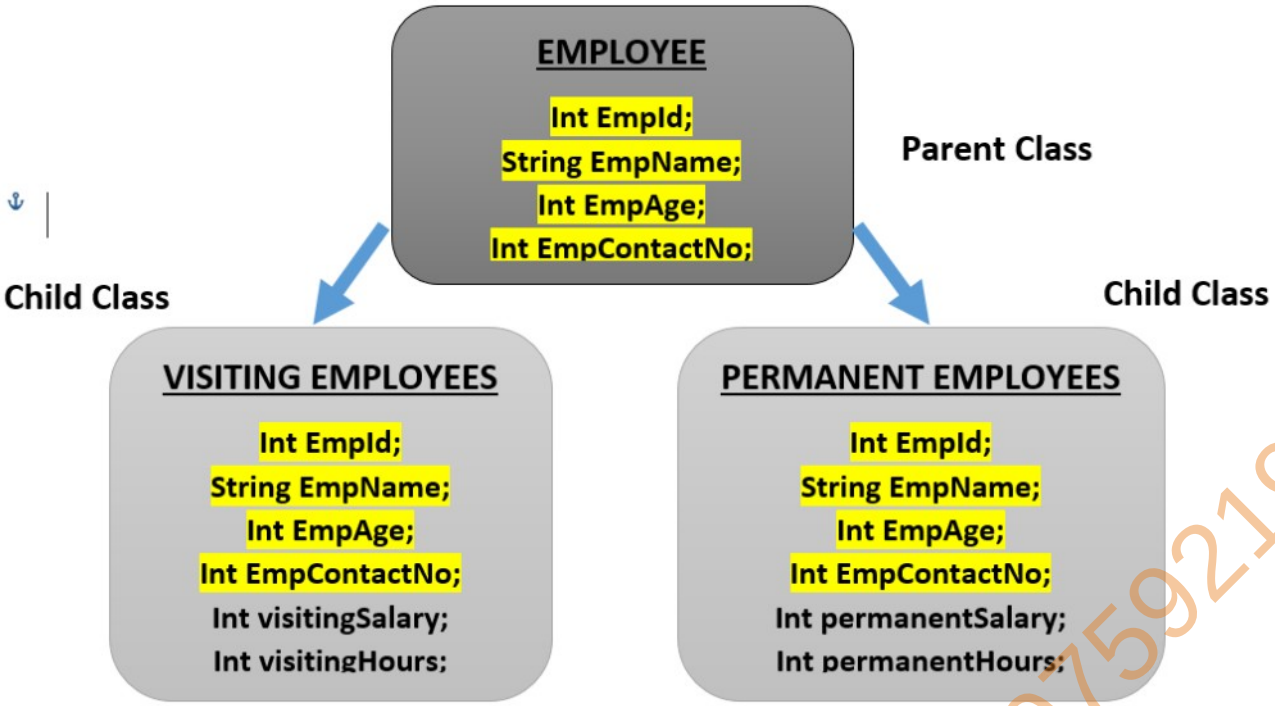
- Consider a class called Vehicle that consists of a variable called color and a method called Speed().
- These data members of the Vehicle class can be inherited by the TwoWheelerVehicle and FourWheelerVehicle classes.

The following figure illustrates an example of inheritance:



Inheritance In C#

- The similarity in physical features of a child to that of its parents is due to the child having inherited these features from its parents.
- Similarly, in C#, inheritance allows you to create a class by deriving the common attributes and methods of an existing class.
- Inheritance provides reusability by allowing us to extend an existing class.
- The reason behind OOP programming is to promote the reusability of code and to reduce complexity in code and it is possible by using inheritance.



Purpose Of Inheritance

- The purpose of inheritance is to reuse common methods and attributes among classes without recreating them.
- Reusability of a code enables you to use the same code in different applications with little or no changes.

Example

- Consider a class named Animal which defines attributes and behavior for animals.
- If a new class named Cat has to be created, it can be done based on Animal because a cat is also an animal.
- Thus, you can reuse the code from the previously-defined class.

Apart from reusability, inheritance is widely used for:

1. Generalization
2. Specialization
3. Extension

Generalization

- Inheritance allows you to implement generalization by creating base classes. For example, consider the class Vehicle, which is the base class for its derived classes Truck and Bike.

- The class Vehicle consists of general attributes and methods that are implemented more specifically in the respective derived classes.

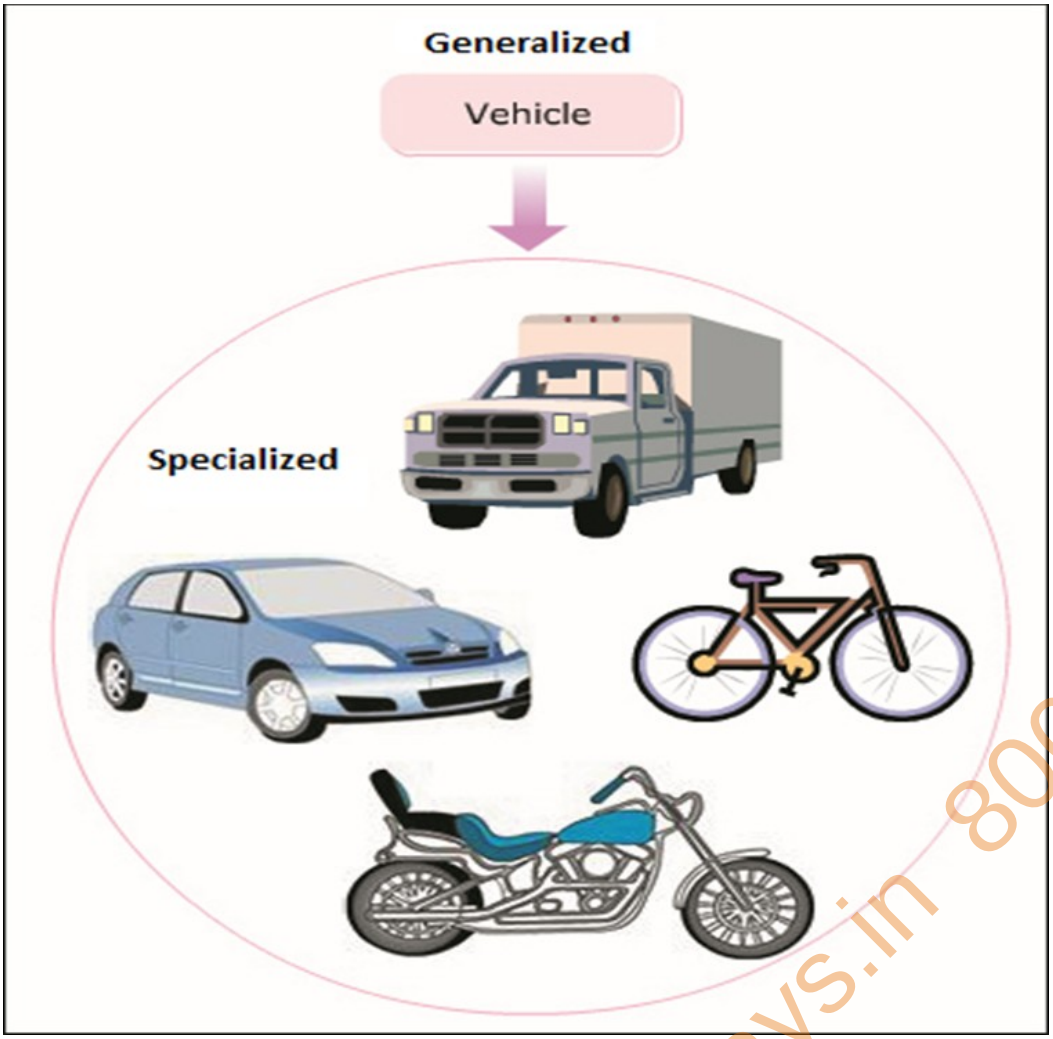
Specialization

- Inheritance allows you to implement specialization by creating derived classes.
- For example, the derived classes such as Bike, Bicycle, Bus, and Truck are specialized by implementing only specific methods from its generalized base class Vehicle.

Extension

- Inheritance allows you to extend the functionalities of a derived class by creating more methods and attributes that are not present in the base class. It allows you to provide additional features to the existing derived class without modifying the existing code.

The following figure displays a real-world example demonstrating the purpose of inheritance:



The inheritance concept is based on a base class and derived class. Let us see the definition of a base and derived class.

- **BASE CLASS -** is the class from which features are to be inherited into another class.
- **DERIVED CLASS -** it is the class in which the base class features are inherited.

Constructor In Inheritance

→ A constructor is a method with the same name as the class name and is invoked automatically when a new instance of a class is created.

→ Constructors of both classes must be executed when the object of child class is created.

→ Derived Class's constructor invokes constructor of Base class.

→ Explicit call to the super class constructor from sub class's can be made using **base()**.

→ If u don't write **base()** explicitly then C# compiler implicitly write the **base()**.

→ If base Class have parameterized constructor then you can add parameters in base().

Base Keyword

The base keyword allows you to do the following:

- Access the variables and methods of the base class from the derived class.
- Re-declare the methods and variables defined in the base class.
- Invoke the derived class data members.
- Access the base class members using the base keyword.

The following syntax shows the use of the base keyword:

```
class <ClassName>
{
<accessmodifier><returntype><BaseMethod>{}
}
class <ClassName1>:<ClassName>
{
base.<BaseMethod>;
}
```

where,

- <**ClassName**>: Is the name of the base class.
- <**accessmodifier**>: Specifies the scope of the class or method.> : Specifies the scope of the class or method.
- <**returntype**>: Specifies the type of data the method will return.
- <**BaseMethod**>: Is the base class method.
- <**ClassName1**>: Is the name of the derived class.
- **base**: Is a keyword used to access the base class members.

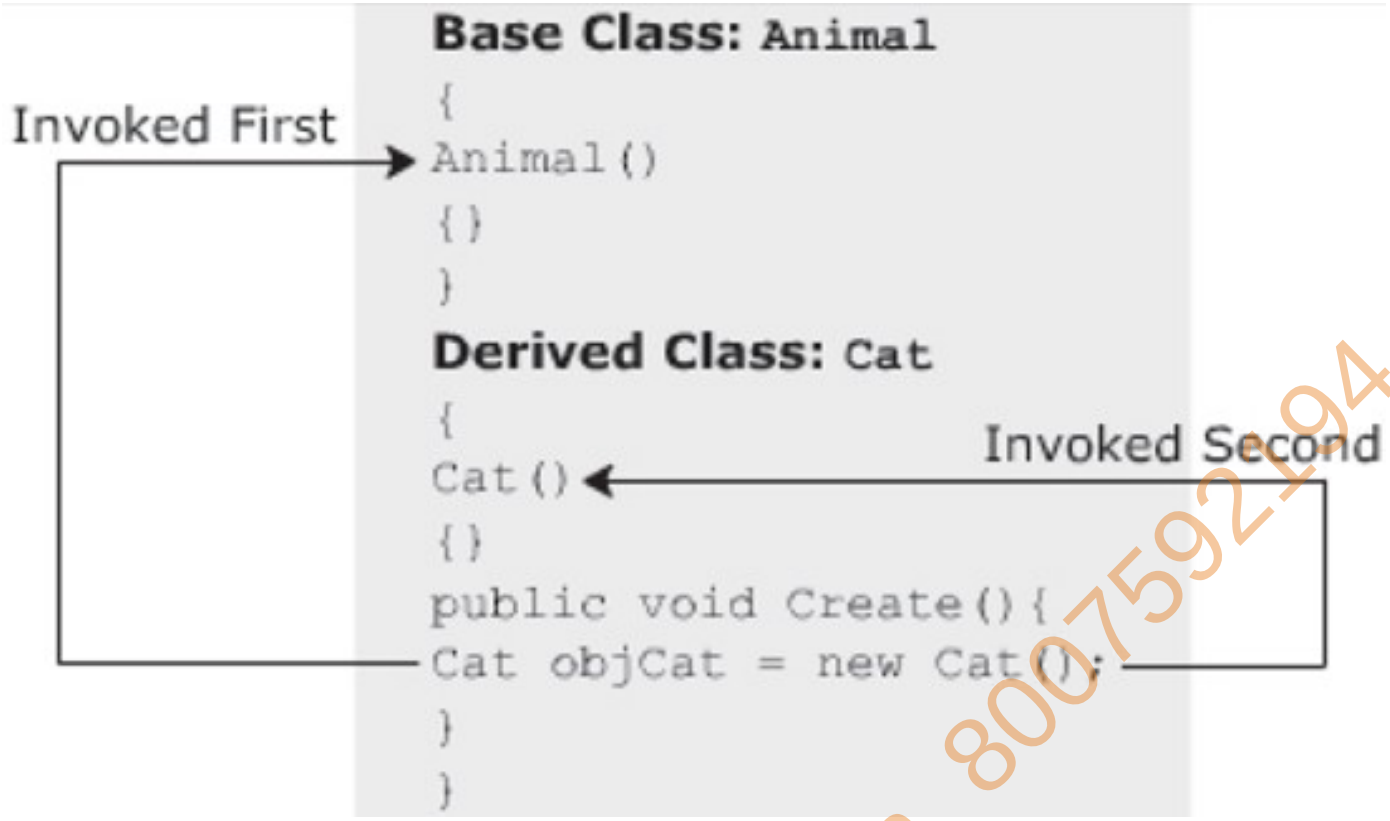
The following figure displays an example of using the base keyword:

```
class Animal
{
    public void Eat(){}
}
class Dog : Animal
{
    public void Eat(){}
    public static Main(string args[])
    {
        Dog objDog = new Dog();
        objDog.Eat;
        base.Eat();
    }
}
```


Constructor In Inheritance

- In C# you can invoke the base class constructor by either instantiating the derived class or the base class.
- In C# you can invoke the constructor of the base class followed by the constructor of the derived class.
- In C# you can invoke the base class constructor by using the base keyword in the derived class constructor declaration.
- In C# you can pass parameters to the constructor.
- However, C# cannot inherit constructors similar to how you inherit methods.

The following figure displays an example of constructor inheritance:



The following code explicitly invokes the base class constructor using the base keyword:

```
class Animal
{
public Animal()
{
Console.WriteLine("Animal constructor without parameters");
}
public Animal(Stringname)
{
Console.WriteLine("Animal constructor with a string parameter");
}
}
class Canine:Animal
{
//base()takes a string value called"Lion"
public Canine():base("Lion")
{
Console.WriteLine("DerivedCanine");
}
}
class Details
{
static voidMain(String[]args)
{
Canine objCanine=new Canine();
```

```
}
}
```

In Above code,

- The class **Animal** consists of two constructors, one without a parameter and the other with a string parameter.
- The class **Canine** is inherited from the class **Animal**.
- The derived class **Canine** consists of a constructor that invokes the constructor of the base class Animal by using the base keyword.
- If the base keyword does not take a string in the parenthesis, the constructor of the class **Animal** that does not contain parameters is invoked.
- In the class **Details**, when the derived class constructor is invoked, it will in turn invoke the parameterized constructor of the base class.

Output

Animal constructor with a string parameter
Derived Canine

Invoking Parameterized Base Class Constructors

- The derived class constructor can explicitly invoke the base class constructor by using the base keyword.
- If a base class constructor has a parameter, the base keyword is followed by the value of the type specified in the constructor declaration.
- If there are no parameters, the base keyword is followed by a pair of parentheses.

The following code demonstrates how parameterized constructors are invoked in a multi-level hierarchy:

```
using System;
class Metals
{
```

```
string_metalType;
public Metals(stringtype)
{
_metalType=type;
Console.WriteLine("Metal:\t\t"+_metalType);
}
}
class SteelCompany : Metals
{
string_grade;
public SteelCompany(stringgrade):base("Steel")
{
_grade=grade;
Console.WriteLine("Grade:\t\t"+_grade);
}
}
class Automobiles:SteelCompany
{
string_part;
public Automobiles(stringpart):base("CastIron")
{
_part=part;
Console.WriteLine("Part:\t\t"+_part);
}
static voidMain(string[]args)
{
Automobiles objAutomobiles=new Automobiles("Chassies");
}
}
```

In Above code,

- The **Automobiles** class inherits the **SteelCompany** class.
- The **SteelCompany** class inherits the **Metals** class.
- In the Main()method, when an instance of the **Automobiles** class is created, it invokes the constructor of the **Metals** class, followed by the constructor of the **SteelCompany** class.
- Finally, the constructor of the **Automobiles** class is invoked.

Output

Metal: Steel
Grade: CastIron
Part: Chassies

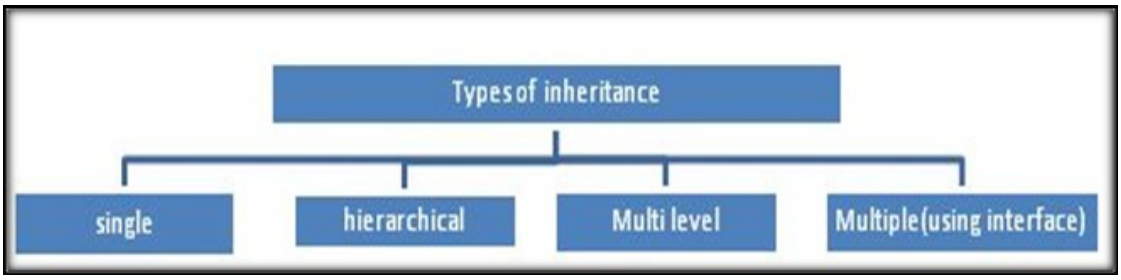
Source Code Of Constructor In Inheritance

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

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

namespace Constructor_In_Inheritance
{

    class BaseClass
    {
        public BaseClass(string message)
        {
            Console.WriteLine(message);
        }
        public BaseClass(int i)
        {
            Console.WriteLine(i);
        }

    }

    class DerivedClass : BaseClass
    {
        public DerivedClass() : base(25)
        {
            Console.WriteLine("this is a constructor of Derived class !!");

        }

    }

    class Program
    {
        static void Main(string[] args)
        {
            DerivedClass dc = new DerivedClass();
            Console.ReadLine();
        }
    }
}
```

types of inheritance in C#

The following are the types of inheritance in C#



There are 4 types of Inheritance in C#

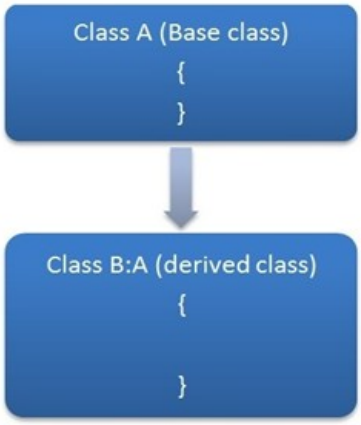
1. Single
2. Multi-level
3. Hierarchical
4. Multiple

The inheritance concept is based on a base class and derived class. Let us see the definition of a base and derived class.

- **BASE CLASS -** is the class from which features are to be inherited into another class.
- **DERIVED CLASS -** it is the class in which the base class features are inherited.

Single Inheritance In C#

It is the type of inheritance in which there is one base class and one derived class.



Single Inheritance In C#

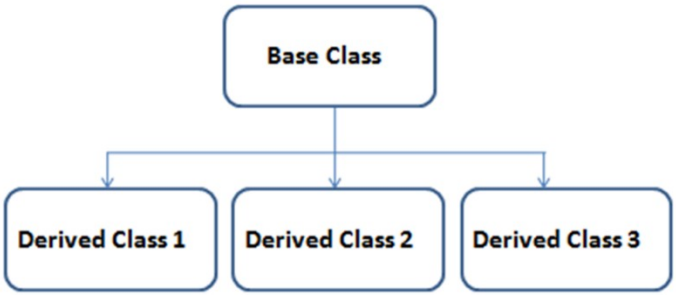
Source Code Of Single Inheritance

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

namespace INHERITANCE_CSHARP
{

    class PermanentEmployees : Employees
    {

        public int permanentSalary;
        public int permanentHours;
    }
    class Employees
    {
        public int EmpId;
        public string EmpName;
        public int EmpAge;
        public int EmpContactNo;

        public void show()
        {
            Console.WriteLine("This is a method of base class !!");
        }

    }


    class Program
    {
        static void Main(string[] args)
        {
            PermanentEmployees Asad = new PermanentEmployees();
            Asad.EmpId = 12;
```

```
            Asad.show();


            Console.WriteLine(Asad.EmpId);

            Console.ReadLine();
        }

    }
}
```

Hierarchical Inheritance

- This is the type of inheritance in which there are multiple classes derived from one base class.
- This type of inheritance is used when there is a requirement of one class feature that is needed in multiple classes.



Multilevel Inheritance

When one class is derived from another derived class then this type of inheritance is called multilevel inheritance.

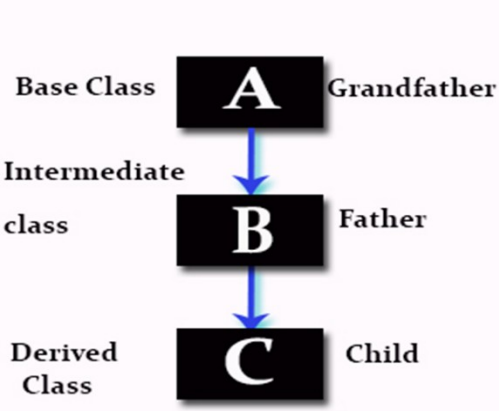


Fig: Multilevel Inheritance

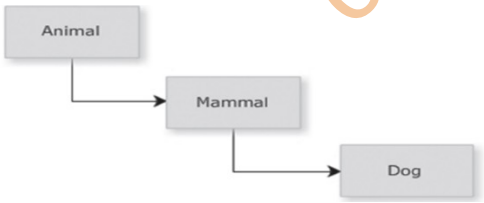
Multi-level Inheritance

Inheritance allows the programmer to build hierarchies that can contain multiple levels of inheritance.

Example

- Consider three classes Mammal, Animal, and Dog. The class Mammal is inherited from the base class Animal, which inherits all the attributes of the Animal class.
- The class Dog is inherited from the class Mammal and inherits all the attributes of both the Animal and Mammal classes.

The following figure depicts multi-level hierarchy of related classes:



The following code demonstrates multiple levels of inheritance:

```
using System;
class Animal
{
public void Eat()
{
Console.WriteLine("Every animal eats something.");
}
}
class Mammal : Animal
{
public void Feature()
{
Console.WriteLine("Mammals give birth to young ones.");
}
}

class Dog : Mammal
{
        public void Noise()
        {
           Console.WriteLine("Dog Barks.");
        }
static void Main(string[] args)
        {
        Dog objDog = new Dog();
        objDog.Eat();
        objDog.Feature();
        objDog.Noise();
        }
}
```

In Above code, the Main() method of the class Dog invokes the methods of the class Animal, Mammal, and Dog.

Output

Every animal eats something.
Mammals give birth to young ones.
Dog Barks.

Polymorphism In C#

→ Polymorphism is one of the four pillars of Object Oriented Programming.

→ Polymorphism in C# is a concept by which we can perform a single action by different ways.

→ Polymorphism is derived from 2 Greek words: POLY and MORPHS.

→ The word "poly" means **many** and "morphs" means **forms**.

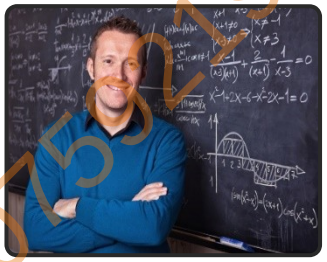
→ So polymorphism means many forms.

Real World Example

FATHER

SON

HUSBAND



TEACHER

There Are Two Types Of Polymorphism

1. Static Polymorphism (Compile Time Polymorphism)
2. Dynamic Polymorphism (Run Time Polymorphism)

Static Polymorphism (Compile Time Polymorphism) In C#

→ The mechanism of linking a function with an object during compile time is called static polymorphism or early binding.

→ It is also called static binding.

C# provides two techniques to implement static polymorphism. They are −

- Method Or Function Overloading
- Operator Overloading

Method Or Function Overloading

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

Dynamic Or Runtime Polymorphism In C#

→Run time polymorphism is achieved by method overriding.

→Method overriding allows us to have virtual and abstract methods in the base using derived classes with the same name and the same parameter.

C# Method Overriding

→If derived class defines same method as defined in its base class, it is known as method overriding in C#.

→It is used to achieve runtime polymorphism.

→It enables you to provide specific implementation of the method in child class which is already provided by its base class.

→To perform method overriding in C#, you need to use **virtual** keyword with base class method and **override** keyword with derived class method.

→A method declared using the virtual keyword is referred to as a virtual method.

→In the derived class, you need to declare the inherited virtual method using the override keyword.

→In the derived class, you need to declare the inherited virtual method using the override keyword which is mandatory for any virtual method that is inherited in the derived class.

→The override keyword overrides the base class method in the derived class.

Difference Between Method Hiding And Method Overriding In C#

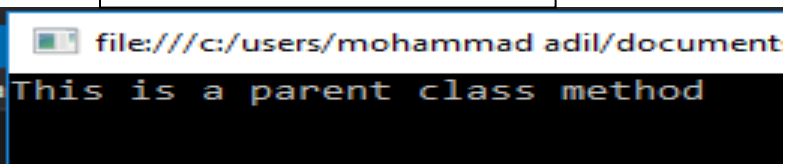
METHOD HIDING

```
class parent
{   public void show()
    {
        Console.WriteLine("This is a parent class method");
    }
}
class child : parent
{   public new void show()
    {
        Console.WriteLine("This is a Child class method");
    }
}
class Program
{   static void Main(string[] args)
    {
        parent p = new child();
        p.show();
        Console.ReadLine();
    }
}
```

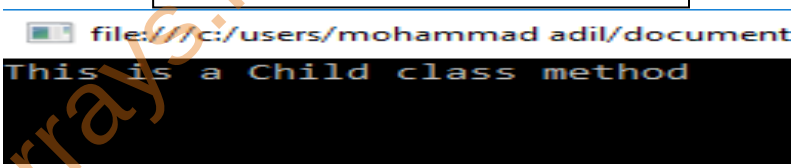
METHOD OVERRIDING

```
class parent
{   public virtual void show()
    {
        Console.WriteLine("This is a parent class method");
    }
}
class child : parent
{   public override void show()
    {
        Console.WriteLine("This is a Child class method");
    }
}
class Program
{   static void Main(string[] args)
    {
        parent p = new child();
        p.show();
        Console.ReadLine();
    }
}
```

METHOD HIDING OUTPUT



METHOD OVERRIDING OUTPUT



In **method hiding**, a base class reference variable pointing to a child class object, will invoke the hidden method of the base class.

In **method overriding**, a base class reference variable pointing to a child class object, will invoke the overridden method of the child class.

Constructor Overloading

- Declaring more than one constructor in a class is called constructor overloading.
- The process of overloading constructors is similar to overloading methods where every constructor has a signature similar to that of a method.
- Multiple constructors in a class can be declared wherein each constructor will have different signatures.
- Constructor overloading is used when different objects of the class might want to use different initialized values.
- Overloaded constructors reduce the task of assigning different values to member variables each time when needed by different objects of the class.

The following code demonstrates the use of constructor overloading:

```
using System;
public class Rectangle
{
double _length;
double _breadth;
public Rectangle()
{
}
_length = 13.5;
_breadth = 20.5;
}
public Rectangle(double len, double wide)
{
_length = len;
_breadth = wide;
}

public double Area()
{
return _length * _breadth;
}
static void Main(string[] args)
{
Rectangle objRect1 = new Rectangle();
Console.WriteLine("Area of rectangle = " + objRect1.Area());
Rectangle objRect2 = new Rectangle(2.5, 6.9);
```

```
Console.WriteLine("Area of rectangle = " + objRect2.Area());
}
}
```

In Above Code,

- Two constructors are created having the same name, Rectangle.
- However, the signatures of these constructors are different. Hence, while calling the method Area()from the Main() method, the parameters passed to the calling method are identified.
- Then, the corresponding constructor is used to initialize the variables _length and _breadth. Finally, the multiplication operation is performed on these variables and the area values are displayed as the output.

Output

Area of rectangle1 = 276.75
Area of rectangle2 = 17.25

Source Code of Constructor Overloading

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

namespace ConstructorOverloading
{
    class Program
    {
        public Program()
        {
            Console.WriteLine("this is a first constructor !!");
        }
        public Program(int a, int b)
        {
            Console.WriteLine("this is a Second constructor !! {0}", (a+b));
        }
        public Program(int a, int b, int c)
```

```
        {
            Console.WriteLine("this is a third constructor !! {0}", (a + b
+c));
        }
        public Program(string a, string b, string c)
        {
            Console.WriteLine("this is a fourth constructor !! {0}", (a + b +
c));
        }



        static void Main(string[] args)
        {
            // CONSTRUCTOR OVERLOADING
            Program p = new Program("A","B","C");
            Console.ReadLine();
        }
    }
}
```

ABSTRACT CLASS IN C#

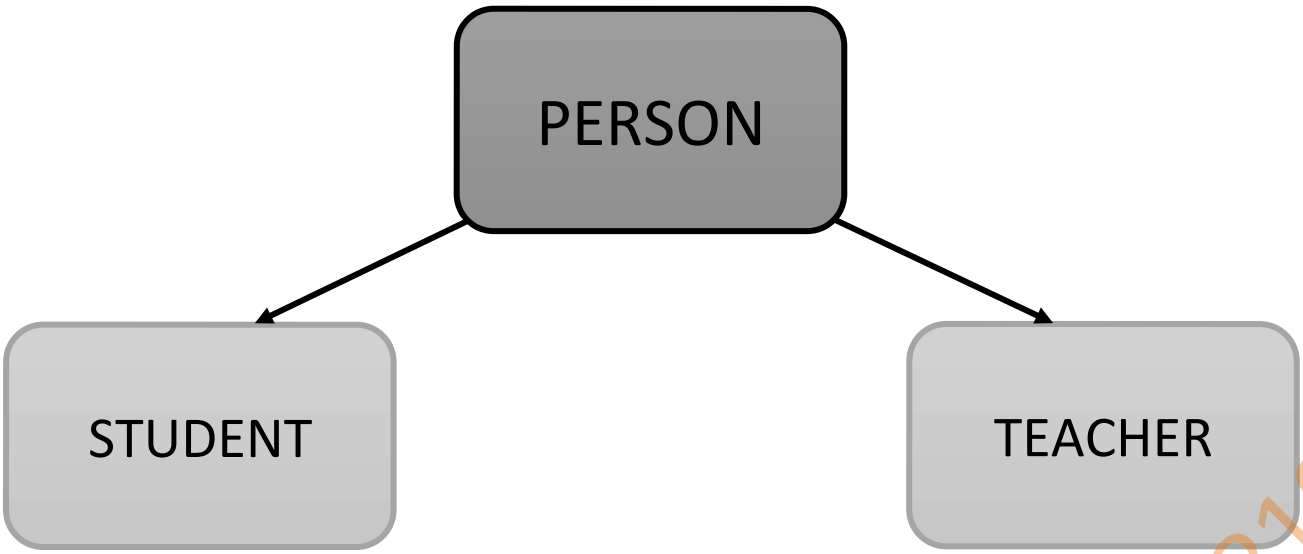
The word abstract means a concept or an idea not associated with any specific instance. In programming we apply the same meaning of abstraction by making classes not associated with any specific instance.

The abstraction is done when we need to only inherit from a certain class, but not need to instantiate objects of that class.

In C#, abstraction is achieved using Abstract classes and interfaces.

Abstract Class:

→ C# Abstract classes are used to declare common characteristics of subclasses.
→ A class which contains the abstract keyword in its declaration is known as abstract class.
→ It can only be used as a BASE class for other classes that extend the abstract class.
→ Abstract classes may or may not contain abstract methods, i.e., methods without body (public void get();)
→ Like any other class, an abstract class can contain fields that describe the characteristics and methods that describe the actions that a class can perform.
→ But, if a class has at least one abstract method, then the class must be declared abstract.
→ If a class is declared abstract, it cannot be instantiated.
→ To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
→ If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.
→ An abstract class can implement code with non-Abstract methods.
→ An Abstract class can have modifiers for methods, properties etc.
→ An Abstract class can have constants and fields.
→ An abstract class can implement a property.
→ An abstract class can have constructors or destructors.



ABSTRACT PROPERTIES IN C#

→ The word Abstract means incomplete, which means no implementation in programming.

→ This is a duty of child class to implement an abstract member of their parent class.

→ Declared by using the **abstract** keyword.

→ Contain only the declaration of the property without the body of the get and set accessors (which do not contain any statements and can be implemented in the derived class).

→ Are only allowed in an abstract class.

INTERFACES IN C#

An interface is a contract between itself and any class that implements it. This contract states that any class that implements the interface will implement the interface's properties, methods and/or events. An interface contains no implementation, only the signatures of the functionality the interface provides. An interface can contain signatures of methods, properties, indexers & events.

The interface defines the **'what'** part of the syntactical contract and the deriving classes define the **'how'** part of the syntactical contract.

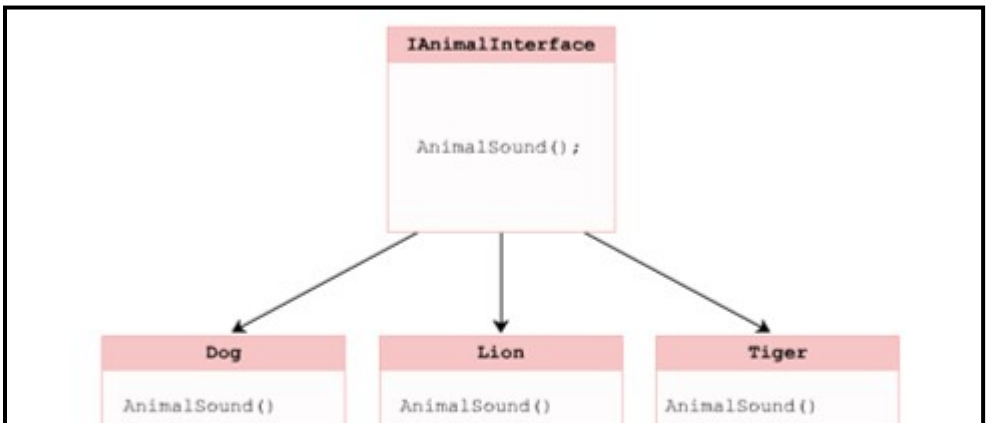
Interfaces define properties, methods, and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members.

One of the main reason to introduce interfaces is that it can be used in multiple inheritance.

- An interface contains only abstract members, just like classes interface also contains properties, methods, delegates or events, but only declarations, no implementations.
- An interface cannot be instantiated but can only be inherited by classes or other interfaces.
- Interface cannot have fields.
- An interface is declared using the keyword **interface**.
- In C#, by default, all members declared in an interface have **public** as the access modifier. They don't allow explicit access modifiers.

IMPLEMENTING AN INTERFACE

→ An interface is implemented by a class in a way similar to inheriting a class.
→ When implementing an interface in a class, implement all the abstract methods declared in the interface. If all the methods are not implemented, the class cannot be compiled.
→ The methods implemented in the class should be declared with the same name and signature as defined in the interface.



INTERFACE INHERITANCE IN C#

→ Interface can inherit from other interfaces.

→ A class that inherits this interface must provide implementation for all interface members in the entire interface inheritance chain.

→ Interface reference variable can have the reference of their child class.

Sealed Methods In C#

→ When the derived class overrides a base class method, variable, property or event, then the new method, variable, property, or event can be declared as sealed.

→ Sealing the new method prevents the method from further overriding.

→ An overridden method can be sealed by preceding the override keyword with the sealed keyword.

Steps To Remember For Sealed Methods:

→ Sealed method is always an override method of child class.

→ We cannot again override the sealed method.

→ Sealed method is only available with Method Overriding.

→ Sealed keyword is not available with the method hiding.

→ Sealed is used together with override method.

→ We cannot make normal methods as sealed.

SEALED CLASS IN C# PROGRAMMING

A sealed class is a class that prevents inheritance.

The features of a sealed class are as follows:

→ A sealed class can be declared by preceding the class keyword with the **sealed** keyword.

→ The sealed keyword prevents a class from being inherited by any other class.

→ The sealed class cannot be a base class as it cannot be inherited by any other class. If a class tries to derive a sealed class, the C# compiler generates an error.

Purpose of Sealed Classes

→ Consider a class named **SystemInformation** that consists of critical methods that affect the working of the operating system.

→ You might not want any third party to inherit the class **SystemInformation** and override its methods, thus, causing security and copyright issues.

→ Here, you can declare the **SystemInformation** class as sealed to prevent any change in its variables and methods.

Sealed Class & Sealed Methods In C#

Sealed Class

A sealed class is a class that **prevents inheritance**.

The features of a sealed class are as follows:

- A sealed class can be declared by preceding the class keyword with the sealed keyword.
- The sealed keyword prevents a class from being inherited by any other class.
- The sealed class cannot be a base class as it cannot be inherited by any other class.
- If a class tries to derive a sealed class, the C# compiler generates an error.

Purpose of Sealed Classes

- Consider a class named SystemInformation that consists of critical methods that affect the working of the operating system.
- You might not want any third party to inherit the class SystemInformation and override its methods, thus, causing security and copyright issues.
- Here, you can declare the SystemInformation class as sealed to prevent any change in its variables and methods.

The following syntax is used to declare a sealed class:

```
sealed class<ClassName>
{
//body of the class
```

```
}
```

where,

- **sealed**: Is a keyword used to prevent a class from being inherited.
- **ClassName**: Is the name of the class that needs to be sealed.

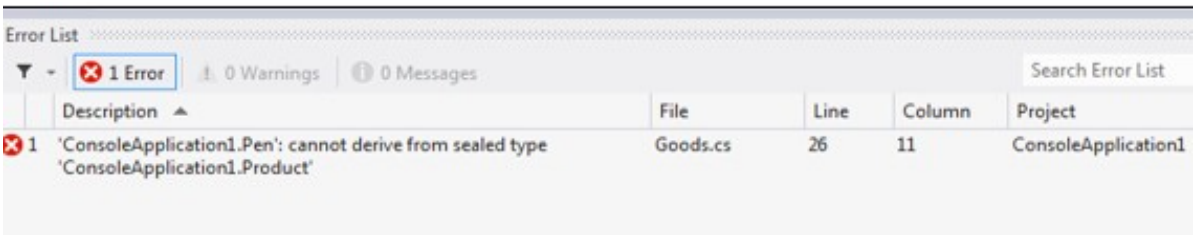
The following code demonstrates the use of a sealed class in C# which will generate a compiler error:

```
sealed class Product
{
public int Quantity;
public int Cost;
}
class Goods
{
static void Main(string [] args)
{
Product objProduct = new Product();
objProduct.Quantity = 50;
objProduct.Cost = 75;
Console.WriteLine("Quantity of the Product: " + objProduct. Quantity);
Console.WriteLine("Cost of the Product: " + objProduct.Cost);
}
}
class Pen : Product
{
}
```

In Above Code,

- The class Product is declared as sealed and it consists of two variables.
- The class Goods contains the code to create an instance of Product and uses the dot (.) operator to invoke variables declared in Product.

The class Pen tries to inherit the sealed class Product, the C# compiler generates an error, as shown in the following figure:



Guidelines

- Sealed classes are restricted classes that cannot be inherited where the list depicts the conditions in which a class can be marked as sealed:
    - If overriding the methods of a class might result in unexpected functioning of the class.
    - When you want to prevent any third party from modifying your class.

Source Code Of Sealed Class

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

namespace SEALED_CLASS
{
    sealed class ParentClass
    {
        public void Show1()
        {
            Console.WriteLine("This is the method of parent class !!");
        }
    }

    class ChildClass : ParentClass
    {
        public void Show2()
        {
            Console.WriteLine("This is the method of child class !!");
```

```
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ChildClass obj = new ChildClass();

        }
    }
}
```

Sealed Methods In C#

https://www.youtube.com/watch?v=m816QtdzDJI&list=PLX07l0qxoHFLZftsVKyj3k9kfMca2uaPR&index=66

- When the derived class overrides a base class method, variable, property or event, then the new method, variable, property, or event can be declared as sealed.
- Sealing the new method prevents the method from further overriding.
- An overridden method can be sealed by preceding the override keyword with the sealed keyword.

Steps To Remember For Sealed Methods:

- Sealed method is always an override method of child class.
- We cannot again override the sealed method.
- Sealed method is only available with Method Overriding.
- Sealed keyword is not available with the method hiding.
- Sealed is used together with override method.
- We cannot make normal methods as sealed.

The following syntax is used to declare an overridden method as sealed:

```
sealed override <return_type> <MethodName>() { }
```

where,

- **return_type**: Specifies the data type of value returned by the method.
- **MethodName**: Specifies the name of the overridden method.

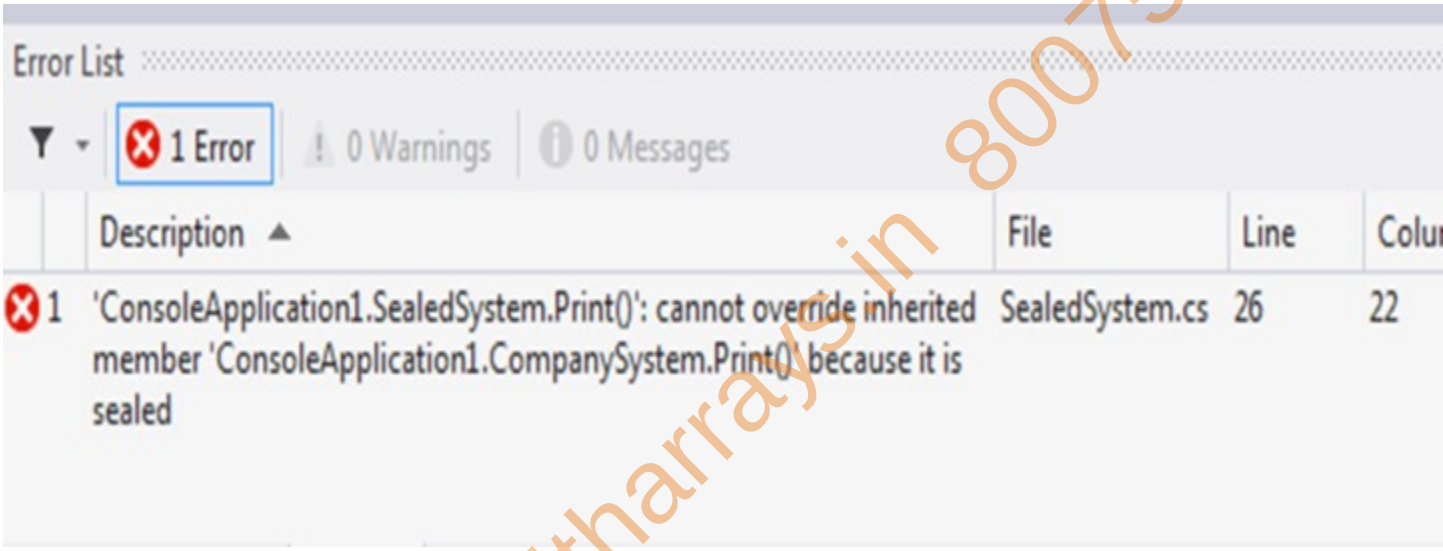
The following code declares an overridden method Print() as sealed:

```
using System;
class ITSystem
{
public virtual void Print()
{
Console.WriteLine ("The system should be handled carefully");
   }
}
class CompanySystem : ITSystem
{
public override sealed void Print()
{
  Console.WriteLine ("The system information is
  confidential");
  Console.WriteLine ("This information should not be
  overridden");
   }
}
class SealedSystem : CompanySystem
{
public override void Print()
{
 Console.WriteLine ("This statement won't get
 executed");
}
static void Main (string [] args)
{
 SealedSystem objSealed = new SealedSystem();
 objSealed.Print ();
}
}
```


In Above Code,

- The class ITSystem consists of a virtual function Print().

- The class CompanySystem is inherited from the class ITSystem.
- It overrides the base class method Print().
- The overridden method Print() is sealed by using the sealed keyword, which prevents further overriding of that method.
- The class SealedSystem is inherited from the class CompanySystem.
- When the class SealedSystem overrides the sealed method Print(), the C# compiler generates an error as shown in the following figure:



Source Code Of Sealed Method

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

namespace SEALED_METHOD
{
    class A
    {
        public void Print()
        {
            Console.WriteLine("This is a method of Class A !!");
        }
    }
```

```
    class B : A
    {
        public new void Print()
        {
            Console.WriteLine("This is a method of Class B !!");
        }
    }
    //class C : B
    //{
    //    public override void Print()
    //    {
    //        Console.WriteLine("This is a method of Class C !!");
    //    }
    //}

    class Program
    {
        static void Main(string[] args)
        {
            A obj = new B();
            obj.Print();

            //C obj = new C();
            //obj.Print();
            Console.ReadLine();

        }
    }
}
```

Delegates In C# Programming

→ **Delegate meaning from google:** a person sent or authorized to represent others, in particular an elected representative sent to a conference.
→ Delegate is a type which holds a method's reference in an object.
→ It is also called function pointer.
→ Delegate is of reference type.
→ Delegate signature should be as same as the method signature referencing by a delegate.
→ Delegate can point to a parameterized method or non-paramterized method.
→ Delagate has no implementation means no body with { }.
→ We can use **invoke()** method with delegates.
→ Delegates are used to encapsulate methods.
→ In the .net framework, a delegate points to one or more methods. Once you instantiate the delegate, the corresponding methods invoke.
→ Delegates are objects that contain references to methods that need to be invoked instead of containing the actual method names
→ Using delegates, you can call any method, which is identified only at run-time.
→ A delegate is like having a general method name that points to various methods at different times and invokes the required method at run-time.
→ In c#, invoking a delegate will execute the referenced method at run-time.
→ To associate a delegate with a particular method, the method must have the same return type and parameter type as that of the delegate.

Method Hiding In C# Programming

→ Method hiding occurs in inheritance relationship when base class and derived class both have a method with same name and same signature.

→ When we create the object of derived class it will hide the base class method and will call its own method and this is called method hiding or **name hiding** in C# inheritance.

→ We use “new” keyword in derived function name to show that implementation of the function in derived class is intentional and derived class no longer want to use base class method.

NOTE: If we do not use “new” keyword then compiler will raise only warning, but, program will work fine.

Different Ways To Call A Hidden Base Class Member From Derived Class

1. Use **base** keyword
2. Cast child type to parent type and invoke the hidden member.
3. ParentClass PC = new ChildClass();
   PC.hiddenMethod();

- Parent class can have the reference of its child class.
- When we create the object of child class, parent class object is also created.

Indexer In C# Programming

Indexers allow our object to be used just like an array, or we can say we can index the object using [ ] brackets just like arrays.

We can say indexers are special type of properties which adds logic that how can array or list store the values.

Syntax of indexer resembles to properties.

We can use all access modifiers with indexers and also have return types.

Indexers are the regular members of a class.

Indexer is created with the help of **this** keyword.

In C# introduce new concept is Indexer. This is very useful for some situation. Let as discuss something about Indexer.

→ Indexer Concept is object act as an array.
→ Indexer an object to be indexed in the same way as an array.
→ Indexer modifier can be private, public, protected or internal.
→ The return type can be any valid C# types.
→ Indexers in C# must have at least one parameter. Else the compiler will generate a compilation error.

Collections In C# Programming

ARRAY	COLLECTION
Cannot be resized at run-time.	Can be resized at run-time.
The individual elements are of the same data type.	The individual elements can be of different data types.
Do not contain any methods for operations on elements.	Contain methods for operations on elements.
We can never insert a value into a middle of an array.	We can insert a value into a middle of a collection.
We can never delete a value from a middle of an array.	We can delete a value from a middle of a collection.

→ A Collection is a set of related data that may not necessarily belong to the same data type that can be set or modified dynamically at run-time.

→ In other words, collection is a dynamic array.

- Its Length can increase on runtime. AUTO RESIZING

→ Normal Array's length is fixed, it means we cannot change the length after declaring an array.

→ Array.Resize()

→ Resize method of an array destroys old array and create a new array with new length.

→ We can never insert a value into a middle of an array, because if we want to do this then array length should be increased but we cannot increase the length of an array after declaring the length of an array.

→ We can never delete a value from a middle of an array.

→ Accessing collections is similar to accessing arrays, where elements are accessed by their index numbers. However, there are differences between arrays and collections in C#.

→ Collections were introduced in C# 1.0

→ We have two kinds of Collections.

- Non-Generic collections
  - Stack, ArrayList, Hashtable, SortedList etc.
  - **System.Collections** namespace have non-generic collections.
- Generic collections.
  - List<T>, LinkedList<T>, Queue<T>,SortedList<T,V>.
  - **System.Collections.Generic** namespace have generic collections.

Generic vs. non-generic collections

Generic collection (new)	Non-generic collection (old)
List<T> and LinkedList<T>	ArrayList
Dictionary<TKey, TValue> and SortedDictionary<TKey, TValue>	HashTable
Queue<T>	Queue
Stack<T>	Stack
SortedList<TKey, TValue>	
HashSet<T> and SortedSet<T>	
	Array []

→ Collections have a mechanism called **auto-resizing.**

→ **Capacity** property which tells number of items that can be stored under any collection.

→ INSERT METHOD.

→ REMOVE AND REMOVEAT METHOD.

Constants & Literals In C# Programming

A constant has a fixed value that remains unchanged throughout the program while a literal provides a mean of expressing specific values in a program.

Example

- Consider a code that calculates the area of the circle.
- To calculate the area of the circle, the value of pi and radius must be provided in the formula.
- The value of pi is a constant value.
- This value will remain unchanged irrespective of the value of the radius provided.
- Similarly, constants in C# are fixed values assigned to identifiers that are not modified throughout the execution of the code.
- They are defined when you want to preserve values to reuse them later or to prevent any modification to the values.

Constants In C#

- A constant has a fixed value that remains unchanged throughout the program.
- In C#, you can declare constants for all data types.
- You have to initialize a constant at the time of its declaration.
- Constants are declared for **value types** rather than for **reference types**.
- To declare an identifier as a constant, the **"const"** keyword is used in the identifier declaration. The compiler can

identify constants at the time of compilation, because of the **“const”** keyword.

The following syntax is used to initialize a constant:

```
const<data type><identifier name> = <value>;
```

where,

- **const**: Keyword denoting that the identifier is declared as constant.
- **data type:** Data type of constant.
- **identifier name:** Name of the identifier that will hold the constant.
- **value**: Fixed value that remains unchanged throughout the execution of the code.

The following code declares a constant named _pi and a variable named radius to calculate the area of the circle:

```
const float _pi = 3.14F;
float radius = 5;
float area = _pi * radius * radius;
Console.WriteLine(“Area of the circle is ” + area);
```

In Above Code,

- A constant called _pi is assigned the value 3.14, which is a fixed value. The variable, radius, stores the radius of the circle. The code calculates the area of the circle and displays it as the output.

Literals In C# Programming

- A literal is a static value assigned to variables and constants.

- Numeric literals might suffix with a letter of the alphabet to indicate the data type of the literal.
- This letter can be either in upper or lowercase.

Example

For example, in the following declaration,

```
string bookName = "Csharp"
```

Csharp is a literal assigned to the variable bookName of type string.

Types Of Literals

- Boolean Literal
- Integer Literal
- Real Literal
- Character Literal
- String Literal
- Null Literal

Boolean Literal

- Boolean Literal: Boolean literals have two values, true or false. For example,
  - bool val = true;
- where,
  - true: Is a Boolean literal assigned to the variable val.

Integer Literal

- Integer Literal: An integer literal can be assigned to int, uint, long, or ulongdata types. Suffixes for integer literals

include U, L, UL, or LU. U denotes uint or ulong, L denotes long. UL and LU denote ulong. For example,
    - long val = 53L;
- Where,
    - 53L: Is an integer literal assigned to the variable val.

Real Literal

- Real Literal: A real literal is assigned to float, double (default), and decimal data types. This is indicated by the suffix letter appearing after the assigned value. A real literal can be suffixed by F, D, or M. F denotes float, D denotes double, and M denotes decimal. For example,
    - float val = 1.66F;
- Where,
    - 1.66F: Is a real literal assigned to the variable val.

Character Literal

- Character Literal: A character literal is assigned to a char data type. A character literal is always enclosed in single quotes. For example,
    - char val = 'A';
- Where,
    - A: Is a character literal assigned to the variable val.

String Literal

- String Literal: There are two types of string literals in C#, regular and verbatim. A regular string literal is a standard string. A verbatim string literal is similar to a regular string literal but is prefixed by the '@' character. A string literal is always enclosed in double quotes. For example,

    - string mailDomain = “@gmail.com”;
- Where,
    - @gmail.com: Is a verbatim string literal.

Null Literal

- Null Literal: The null literal has only one value, null. For example,
    - string email = null;
- Where,
    - null: Specifies that e-mail does not refer to any objects (reference).

GENERICS IN C# PROGRAMMING

→ Generics introduced in C# 2.0.
→ Generics allow you to write a class or method that can work with any data type.
→ Generics allow you to define a class with placeholders for the type of its fields, methods, parameters, etc. Generics replace these placeholders with some specific type at compile time. It helps you to maximize code reuse, type safety, and performance.
→ You can create your own generic interfaces, classes, methods, events, and delegates.
→ You may get information on the types used in a generic data type at run-time.
→ A generic class or method can be defined using angle brackets <>

Generics can be applied to the following:

→ Interface
→ Abstract class
→ Class
→ Method
→ Static method
→ Property
→ Event
→ Delegates
→ Operator

Advantages of Generic:

→ Increases the reusability of the code.
→ Generic has a performance advantage because it removes the possibilities of boxing and unboxing.

GENERIC METHODS

Generic methods process values whose data types are known only when accessing the variables that store these values.

A generic method is declared with the generic type parameter list enclosed within angular brackets.

Defining methods with type parameters allow you to call the method with a different type every time.

You can declare a generic method within generic or non-generic class declarations.

Generic methods can be declared with the following keywords:

- → **Virtual**
- → **Override**
- → **Abstract**

GENERIC CLASS IN C# PROGRAMMING

→ Generic classes define functionalities that can be used for any data type and are declared with a class declaration followed by a type parameter enclosed within angular brackets.

→ Generics introduced in C# 2.0. Generics allow you to define a class with placeholders for the type of its fields, methods, parameters, etc. Generics replace these placeholders with some specific type at compile time.

https://www.youtube.com/@codewitharrays

https://www.instagram.com/codewitharrays/

https://t.me/codewitharrays Group Link: https://t.me/cceesept2023

+91 8007592194 +91 9284926333

codewitharrays@gmail.com

https://codewitharrays.in/project