Introducing C# Programing Language

C# is a high-level and object oriented programming language developed by Microsoft. It was first released with version 1.0 in 2002 along with .NET Framework 1.0. The founder of C# was Anders Hejlsberg. See the releases of both .Net and C# below:

Releases of .NET:

.NET Version	CLR Version	Release Date	Tool
1.0	1.0	2002	Visual Studio .NET
1.1	1.1	2003	Visual Studio 2003
2.0	2.0	2005	Visual Studio 2005
3.0	2.0	2006	No New Version
3.5	2.0	2007	Visual Studio 2008
4.0	4	2010	Visual Studio 2010
4.5	4	2012	Visual Studio 2012
4.5.1	4	2013	Visual Studio 2013
4.5.2	4	2014	NA
4.6	4	2015	Visual Studio 2015
4.6.1	4	2015	VS2015 Update 1

Releases of C#.

C# Version	Features	Release Date	Tool
1.0	First Release	2002	Visual Studio .NET
1.1	**#line** pragma and xml doc comments	2003	Visual Studio 2003
2.0	Anonymous methods, generics, nullable types, iterators/yield, **static** classes, co/contra variance for delegates	2005	Visual Studio 2005
3.0	Object and collection initializers, lambda expressions, extension methods, anonymous types, automatic properties, Language Integrated Query (LINQ), anonymous types, local **var** type inference, LINQ	2008	No New Version
4.0	**Dynamic**, named arguments, optional parameters, generic co/contra variance	2010	Visual Studio 2008
5.0	**Async** / **await**, caller information attributes	2012	Visual Studio 2010
6.0	Some new features like nameof, string interpolation, Index initializer.	2015	Visual Studio 2015

There is a plethora of programming language available these days but a user prefers a language that involves less coding and at the same time enables them to develop applications in an interactive and visually powerful environment such as Visual Studio 2012 IDE.

C# is a language that involves less coding due to which it is easy to learn and understand as compared to a low-level programming language. A low programming language also known as machine language is difficult to learn and interpret. In addition, it is designed for a specific computer to reflect its machine code. On the contrary, a high-level programming language is easy to read and close to spoken language like English. In C#, you can write source code using statements which are similar to English words and statements therefore people choose high-level programming language, such as C#, Java and C++ for programming purposes.

C# combines the power and efficiency of C++, the simple and clean object-oriented design of Java, and the simplification of Visual Basic. With a wide features, C# becomes the integral part of .NET Framework. The features of C# are flexible, powerful, easy to use, visually oriented, Internet friendly and secure.

C# is available in Visual Studio IDE through which you can use all the Framework Class Library and their members in Visual C# syntax.

C# does not use multiple inheritance and pointers, but provides garbage collection at runtime. Moreover C# includes the useful operations such as operator overloading, enumerations, properties, indexers, preprocessor directives, pointers, and function pointers with the help of delegates.

Explaining the relationship between C# and .NET Framework 4.5

C# is an object-oriented programming language developed by Microsoft. It helps application developers to build secure and robust applications that target the .NET Framework. C# is compatible with CLR and CLS due to which it gets the environment for developing and running the various applications developed by it. C# has been created to be used with .NET Framework to provide the business logic for various type of applications that can be developed by using .NET Framework.

Describing C# 5.0 new features

In C# 5.0, several new features have been added to increase the productivity of developers. Following is the list of new features and enhancements in C# 5.0:

- Async and await
- Caller information
- Lambda Expressions
- LINQ expressions
- Named arguments.
- Overload resolution

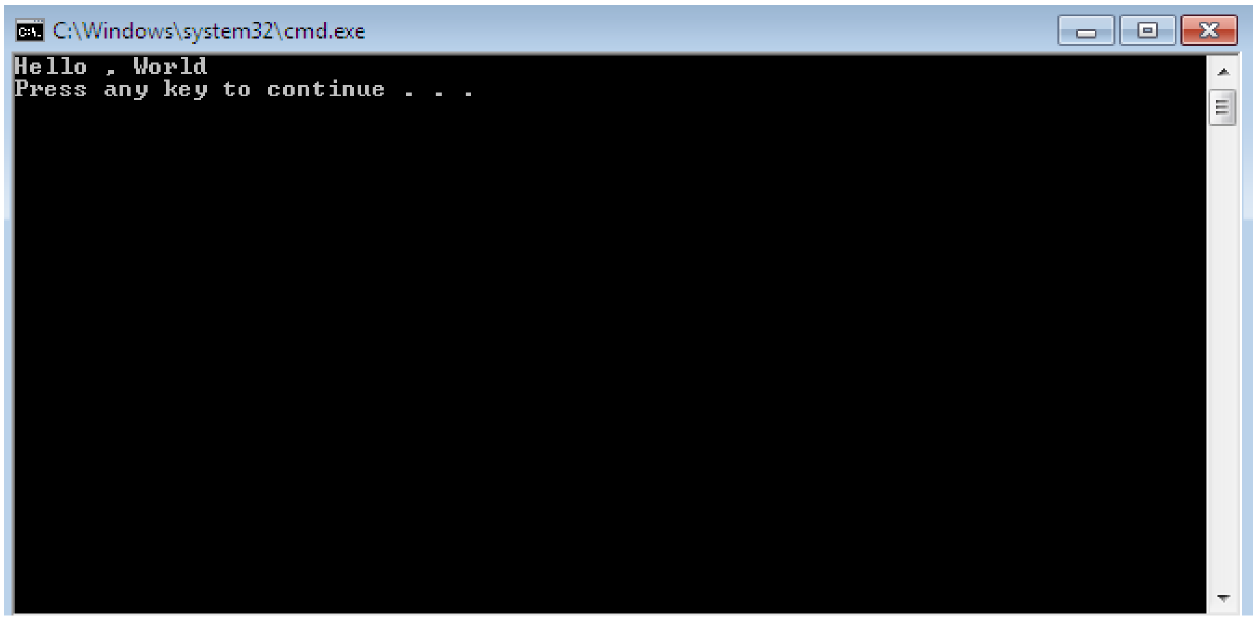
You will learn all the new features in coming modules.

Understanding the basic structure of C# Program

Here we will create the simple program of C# by which we will understand the basic structure. See the below code:

```
using System;
class Hello
{
    public static void Main()
    {
        Console.WriteLine("Hello , World");
    }
}
```

Above is the first program most people write when learning a new language. The example code contains all the essential elements of C# program. The above program will give the output as given below:



```
Hello , World
Press any key to continue . . .
```

We will understand all the elements one by one.

The Using Directive and the System Namespace:

As part of the Microsoft .NET Framework, C# is supplied with many utility classes that perform a range of useful operations. These classes are organized into namespaces. A namespace is a set of related classes. A namespace may also contain the other namespaces.

The .NET Framework is made up of many namespaces, the most important of which is called System. The System namespace contains the classes that most applications use for interacting with the operating system.

In the above code, we also have used System namespace in the below given way:

```
        using System;
```

using is the directive used to include the namespace in the program and System is the name of namespace.

You can refer to objects in namespaces by prefixing them explicitly with the identifier of the namespace. For example, the System namespace contains the Console class, which provides several methods including WriteLine. You can access the WriteLine method of the Console class as follows:

```
System.Console.WriteLine("Hello , World");
```

However, using a fully qualified name to refer to objects can be unwieldy and error prone. To ease this burden, you can specify a namespace by placing a using directive at the beginning of your application before the first class is defined. A using directive specifies a namespace that will be examined if a class is not explicitly defined in the application. You can put more than one using directive in the source file, but they must all be placed at the beginning of the file.

The Class "Hello"

In C#, an application is collection of one or more classes, data structures and other types. A class is defined as a set of data combined with methods (or functions) that can manipulate that data.

You can look into the code in the above program, you will see that there is a single class called Hello. This class is introduced by using the keyword class. Following the class name is an open brace '{'. Everything up to the corresponding closing brace '}' is part of the class.

The Main Method

Every application must start somewhere. When a C# application is run, execution starts at the method called Main. If you are used to programming in C, C++, or even Java, you are already familiar with the concept.

Although there can be many classes in a C# application, there can only be one entry point. It is possible to have multiple classes each with Main in the same application, but only one Main will be executed. You need to specify which one should be used when the application is compiled.

The signature of Main is important too. If you use Visual Studio, it will be created automatically as static void (You will learn more about static and void in coming modules).

The applications runs either until the end of Main is reached or until a return statement is executed by Main.

The Console Class

The console class provides a C# application with access to the standard input, standard output, and standard error streams.

> Standard input is normally associated with the keyboard- anything that the user types on the keyboard can be read from the standard input stream. Similarly, the standard output stream is usually directed to the screen, as is the standard error stream.
>
> **Note:** These streams and the **Console** class are only meaningful to console applications. These are applications that run in command window.

Write and WriteLine Methods

You can use the Console.Write and Console.WriteLine methods to display information on the console screen. These two methods are very similar, the main difference is that WriteLine appends a new line/carriage return to the end of the output, and Write does not.

Both methods are overloaded, you will learn more about method overloading in later modules but in simple terms it means you can use both functions in various ways. You can call them with variable numbers and types of parameters. For example, you can use the following codes to write “99” to the screen:

Console.WriiteLine(99) or Console.WriteLine(“99”).

Understanding Text Formatting

You can use more powerful forms of Write and WriteLine that take a format string and additional parameters. The format string specifies how the data can be displayed in different-different formats, for example you need to write the sentence like

“The Sum of 100 and 130 is 230”

You can write above sentence either by simple concept of concatenation like below:

Console.WriteLine(“The sum of “ + 100 + “ and “ + 130 + “ is “ + (100+130));

Above syntax is workable but quite complicated and prone to error, so you can replace the above syntax by the below code:

Console.WriteLine(“The sum of {0} and {1} is {2} “,100,130,100+130);

Here the place of {0} is taken by 100 similarly {1} will be replaced by 130 and {2} will be replaced by the sum of 100 and 130.

Read and ReadLine Methods

You can obtain user input from the keyboard by using the Console.Read and Console.ReadLine methods.

Read method reads the next character from the keyboard. It returns the int value -1 if there is no more input available. Otherwise it returns an int representing the character read. You can write below code to read a single character:

```
int ch = Console.Read();
Console.WriteLine(ch);
```

Here code will generate the output in the form of ascii value of the given character for e.g. if user gives the character 'a' then it will generate the output 97 which is an ascii value of 'a'.

ReadLine reads all character up to end of the input line(the carriage return character). The input is returned as a string of characters. You can sue the following code to read a line of text from the keyboard and display it to the screen:

```
String input = Console.ReadLine();
Console.WriteLine(input);
```

Understanding Comments:

It is important to provide adequate documentation for all of your program. Programmer needs to write the explanation of code while writing the program to make the code more understandable. That means whatever you comment in the program is invisible to compiler and hence compiler does not compile and skip it.

It is a good practice to write the comments for describing the code because it helps the developer who was not involved in creating the original application or program to follow the comments and understand how applications works.

Always use thorough and meaningful comments. Good comments add information that cannot be expressed easily by the code statement alone- they explain "why" rather than the "what".

C# provides basically two mechanism for adding comments to application code: Single-line comments and multi-line comments.

You can add single-line comment by using the forward slash characters (//). When you run your application, everything following these two characters until the end of the line is ignored by compiler.

You can also use block comments that span multiple lines. A block comment starts with the /* character pair and continues until a matching */ character pair is reached. You cannot nest block comments. See the below example which includes both single-line and multi-line comments.

```
using System; // System is the namespace
class Hello // declared a class with name "Hello"
{
    public static void Main()
    {
        /*here we will accept the string value from user
         *and then will display on the console screen , we will use Console class
         *and WriteLine method for it ...
       */
        Console.WriteLine("Give he string ");
        string value = Console.ReadLine();
        Console.WriteLine("Value : " + value);

    }
}
```

Exploring Tokens and Keywords

Tokens are generally any unit that is not whitespace or a comment. Tokens includes keywords, identifiers, literals and operators.

Keywords are those words or identifiers, which are reserved to be used for a specific task. You cannot use a keyword to define the name of a class, variable or method. Keywords are the reserved words whose meanings are predefined to the C# compiler. They are sued in all programming languages because keywords are an essential part of a language definition. However, there are some keywords that have special meaning in the context of code but such keywords are not reserved words. These keywords are known as contextual keywords. Below is the list of reserved and contextual keywords.

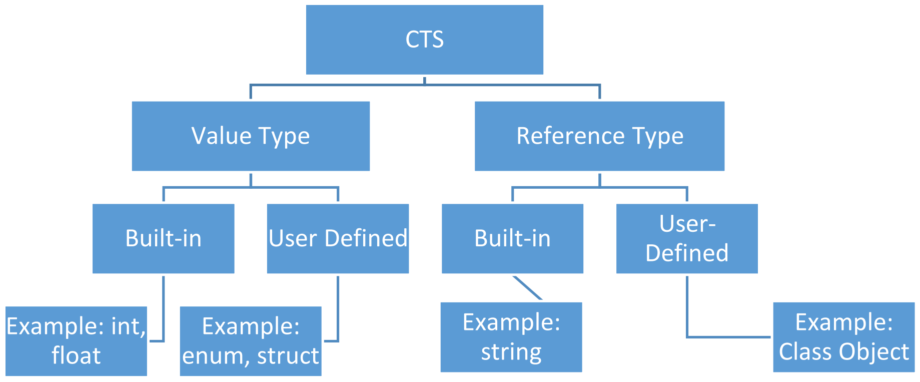
Reserved and Contextual Keywords in C# 5.0			
Reserved Keyboard			
abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	in(generic modifier)	int	interface
internal	is	lock	long
namespace	new	null	object
operator	out	out(generic modifier)	override
params	private	protected	public
readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc
static	string	struct	switch
this	throw	true	try
typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual
volatile	void	while	
Contextual Keywords			
Add	alias	ascending	async
await	descending	dynamic	from
group	get	global	into
join	let	orderby	partial(type)
partial(method)	remove	select	set
value	var	where(generic type)	where(query clause)

Note: You can use keywords as identifier in your program if they include @ as a prefix. For instance @try is a valid identifier but you cannot use try as an identifier because it is a keyword.

Explaining Data Types in C#

Every variable has a data type that determines what values can be stored in the variable. C# is a type-safe language, meaning that the C# compiler guarantees that values stored in variables are always of the appropriate type.

The **Common Language Runtime (CLR)** includes a **Common Type System (CTS)** that defines a set of built-in data types that you can use to define your variables. CTS is an integral part of the common language runtime. The compiler, tools, and the runtime itself share CTS. It is the model that defines the rules that the runtime follows when declaring, using, and managing types. CTS establishes a framework that enables cross-language integration, type safety and high performance code execution. CTP provides basically two types of data-types, see the below image:



Value Types

Value types have fixed length and are stored on the stack of memory. A stack is used to store certain information in a computer. It is basically a data structure, which follows the last-in-first-out principle. A stack is used to store value type variables in C#. When a value of variable is assigned to another variable, the value is copied from one variable to another. That means that two identical copies of the values are available in the memory.

Value-type variables directly contain their data. Each value-type variable has its own copy of the data, so it is not possible for one variable to affect another variable. Value types includes built-in and user-defined data types. The difference between built-in and user-defined types in C# is minimal because user-defined types can be used in the same way as built-in ones. The only real difference between built-in data types-defines and user-defined data types is that you can write literal values for the built-in types. All values types directly contain data, and they cannot be null.

Built-in value types are also referred to as basic data types or simple types. Simple types are identified by means of reserved keywords. These reserved keywords are aliases for predefined struct types. Every built-in value type represent the Structure available in the .NET Library.

A simple type and the struct type it aliases are completely indistinguishable. In your code, you can use the reserved keywords or you can use the struct type. The following below is the example:

Int // Reserved Keyword

System.Int32 //struct type

See the table below:

Reserved Keywords	Alias for struct type	Size in Bits	Default Values
sbyte	System.SByte	8 bits	0
byte	System.Byte	8 bits	0
short	System.Int16	16 bits	0
ushort	System.UInt16	16 bits	0
int	System.Int32	32 bits	0
uint	System.UInt32	32 bits	0
long	System.Int64	64 bits	0L
ulong	System.UInt64	64 bits	0
char	System.Char	16 bits	'\0'
float	System.Single	32 bits	0.0F
double	System.Double	64 bits	0.0D
bool	System.Boolean	1 bit	false
decimal	System.Decimal	128 bits	0.0M

We can also create our own user-defined value types like struct or enum in coming modules.

Note: All of the base data types are defined in the System namespace. All types are ultimately derived from System.Object. Value types are derived from System.ValueType.

Reference Types

Reference types have varied lengths and are stored on the heap of memory. When a value is assigned to a reference type variable, only the reference is copied and the actual memory remains the same in the memory location. Reference type variables contain references to their data. The data for reference-type variable is stored in an object. It is possible for two references type variables to reference the same object, so it is possible for operations on one reference variable to affect the object referenced by another reference variable.

Reference type further can be divided into two types:

A. **Pre-defined Types**

   Pre-defined reference types can also be divided into three types:

   - **Dynamic Type**: Performs the type checking of the dynamic type variable at run time instead of compile time.
   - **Object type**: Refers to the ultimate base type of all other pre-defined and user-defined data types in C#. You can use the Object type to convert a value type on stack into an Object type to be placed on the heap.
   - **String type:** Refers to the creation and manipulation of strings in C#.

B. **User-defined Types**

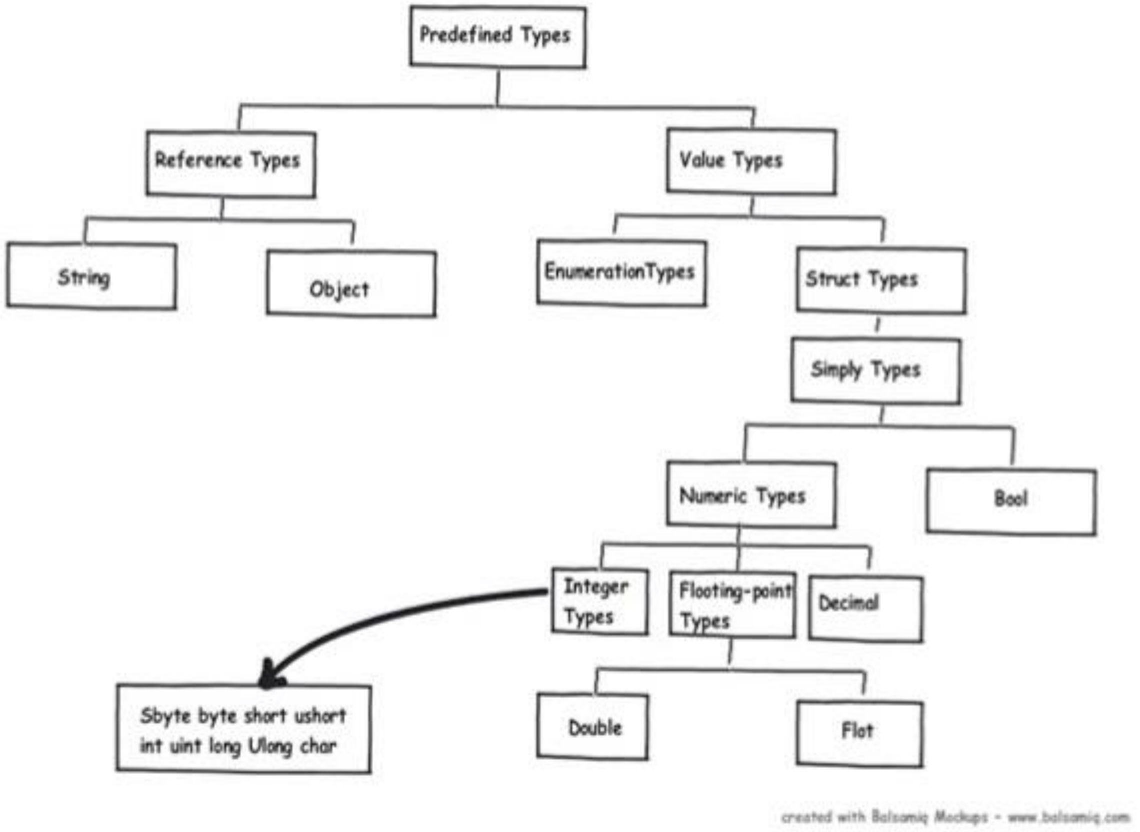
   User-defined reference types refer to those types, which are defined using the pre-defined types. Following are the user-defined reference types:

   - **Class:** Specifies that C# is purely an object-oriented language, so the base of all the C# program is class. Using classes, you can group logically related data items and functions.
   - **Interfaces:** Refers to the collection of members, such as methods, delegates, events, and properties which are implemented by classes. Interfaces cannot be instantiated but you can use them to offer a set of functionalities that is common to several other classes. The concept of interface was introduced to support multiple inheritance because in C#, a class does not support multiple inheritance.
   - **Delegates:** Refers to the objects that store references of the methods signatures match with the signatures of the delegates. Delegates are

the reference types that encapsulates a method with a specific signature.

- **Array:** Refers to an ordered arrangement of data elements. It is a data structure that contains number of variables of the same data type at contiguous locations of the memory.

See the graphical representation for value types and reference types:



Working with Variables

Every Program requires the data on which it can perform operations. Take an example of calculating the addition of two numbers by the program. Hence for calculating two numbers we require to store two numbers somewhere in memory so that their values can be fetched for further processing. So a variable is an identifier that denotes a storage location in memory. Using the name of variable in a program you can refer to the information stored at a particular location. A value of variable can be changed any time if required. Every variable has a type that determines the values to be stored in it. You can give any name to a variable but it should be meaningful because it makes the code more readable.

Rules for naming variable:

- Start each variable name with a letter or underscore character but not begin with a digit.
- After the first character you can use letter, digits, or the underscore character.
- You cannot use any reserved words or keywords in the variable name like break, foreach and class.
- You can't use white spaces in the variable name for example "emp name" is not a valid variable name due to white space coming in between, white space can be replaced with underscore(_) to differentiate one word to another word like "emp_name" is valid.

You need to follow the above rules while making the name of variable which are mandatory, further below are some Recommendations that you can use while naming variable to make your variable name more effective.

Recommendations

- Avoid using all uppercase letters.
- Avoid starting with an underscore.
- Avoid using abbreviations.
- Use PascalCasing naming in multiple-word names like variable name "EmployeeName" is written in PascalCasing because both words e.g. Employee and Name are starting with uppercase , hence when you are required to write multiple words in single variable name then it is good practice to start every word with uppercase.
- Variable name should always resemble with the type of data it is holding, for example you are going to store salary then can make variable name "Salary" instead of "a" or sal" etc.

Below are some examples of valid and invalid variable name:

Variable Name	Status
EmpName	Valid Name
Total_Salary	Valid Name
Emp_Yearly_Cost	Valid Name
_EmpMobile	Valid but not recommended
Total#amount	Not Valid due to # coming.
double	Not Valid, it is an keyword
8Employee	Not Valid, starting with digit.
Product Cost	Not Valid , due to white space coming
Emp_Mobile9	Valid Name

Declaring Variables

You can declare variable to store the various types of values but for which you need to first analyze the value type of your data. After analyzing your value type, you can assign the similar data types available in C# with the variable name at the time of declaration. Variable declaration consist of two things first the data type and second the variable name. By assigning data type you are confirming about what type of value the variable is going to hold and by assigning variable name you are giving the name or identifier to the location where value is going to be stored in the memory. We define our user-friendly name which resembles English like words to the variable for our benefit while doing programming. Because every time while programming, you will access the variable with its name which is an identifier of the location where value is stored in the stack memory.

A variable declaration perform the following tasks:

- Tells the compiler the name of the variable.
- Tells about the type of data a variable holds.
- Decides the scope of a variable.

Moreover a variable must be declared and must be assigned before using it. To declare a variable, you must specify a data type to it according to the value assigned following by the name of the variable. Below is the syntax for declaration of single variable:

datatype variablename;

E.G. int Emp_ID;

    string Emp_Name;

Above we have declared a variable with name Emp_ID which will hold employee id that is numeric so data type assigned is int. Similarly for variable with name Emp_Name.

If you require to declare more than one variable of same data type than instead of declaring many times, you can declare all in single line for E.G. see the below code:

```
Int Emp_ID,Age,Total_Employee;
```

Here we have declared three variables in the same line separated by comma because they have common data type.

Assigning Values to variables:

After the declaration of a variable, you need to initialize its value. Initialization of a value to a variable means assigning a value to a variable. While initializing a variable, ensure that a variable is assigned a value before being used in a program. It is not mandatory to initialize a variable at the time of declaration. See below the syntax;

```
Int Age; //declaration of variable

Age = 23; // Assigning value to variable.
```

You can also combine the declaration and assigning of value like below:

```
Int Age = 23;
```

Now you have to decide whether you want to declare only or you want to declare with assigning value, it depends on the situations that will come while doing programming.

Similarly. You can declare and assign many variables in the single line in case they all are of same data types like below:

```
Int Age=23, Emp_ID=123, Total_Employee=44;
```

Working with Constants

Constants are immutable values which are known at compile time and do not change for the life of the program. Constants are declared with the const modifier. Only the C# built-in types (excluding System.Object) may be declared as const. User-defined types, including classes, structs, and arrays, cannot be const.

Use the readonly modifier to create a class, struct, or array that is initialized one time at runtime (for example in a constructor) and thereafter cannot be changed.

C# does not support const methods, properties, or events.

The enum type enables you to define named constants for integral built-in types (for example int, uint, long, and so on).Constants must be initialized as they are declared.

```
using System;
class calendar
{
    public const int month = 12;
}
```

In this example, the constant months is always 12, and it cannot be changed even by the class itself. In fact, when the compiler encounters a constant identifier in C# source code (for example, months), it substitutes the literal value directly into the intermediate language (IL) code that it produces. Because there is no variable address associated with a constant at run time, const fields cannot be passed by reference and cannot appear as an l-value in an expression.

Multiple constants of the same type can be declared at the same time, for example:

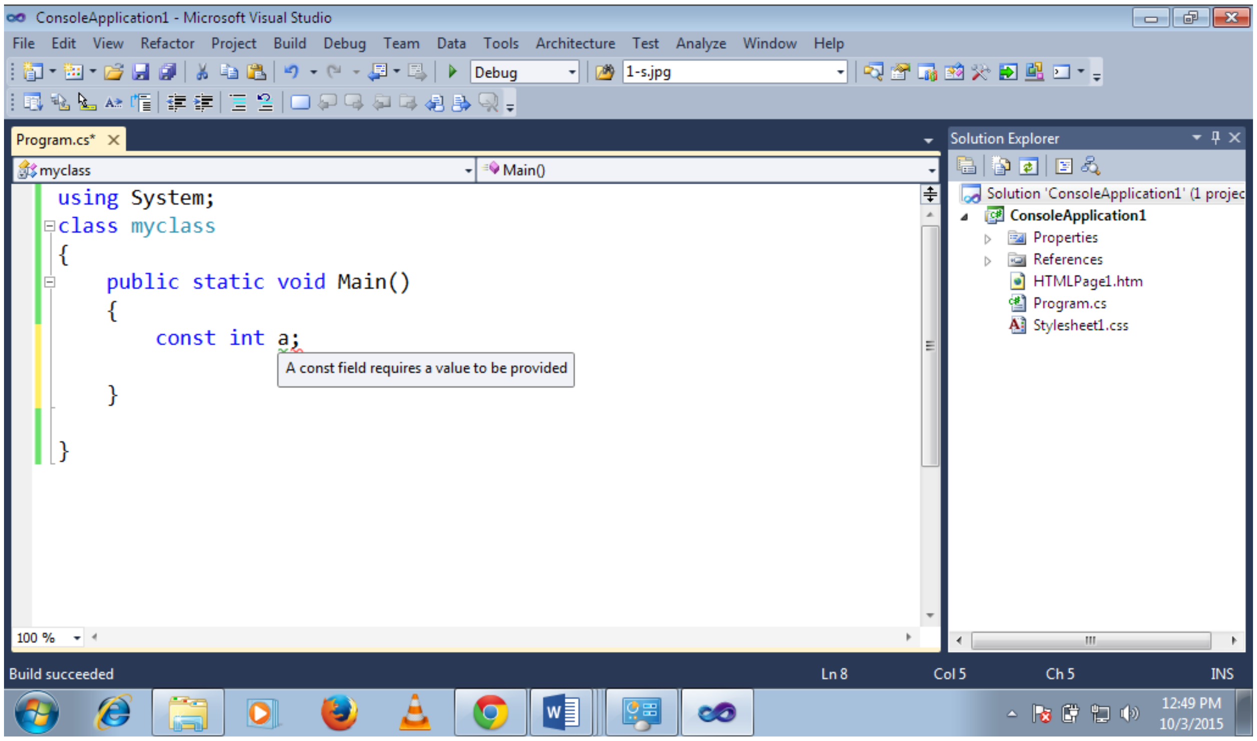
```
using System;
class calendar
{
    const int months = 12, weeks = 52, days = 365;
}
```

Constants can be marked as public, private, protected, internal, or protected internal. These access modifiers define how users of the class can access the constant. We will come to know about all access specifiers in the coming module.

Constants are accessed as if they were static fields because the value of the constant is the same for all instances of the type. You do not use the static keyword to declare them. Expressions that are not in the class that defines the constant must use the class name, a period, and the name of the constant to access the constant. For example, you can access the month constant directly in the main without instance of class. We will cover static in the coming module.

Case Studies on Constants:

Case 1: It is compulsion to initialize the constant at the time of declaration. Like below Example:



```
using System;
class myclass
{
    public static void Main()
    {
        const int a;

    }

}
```

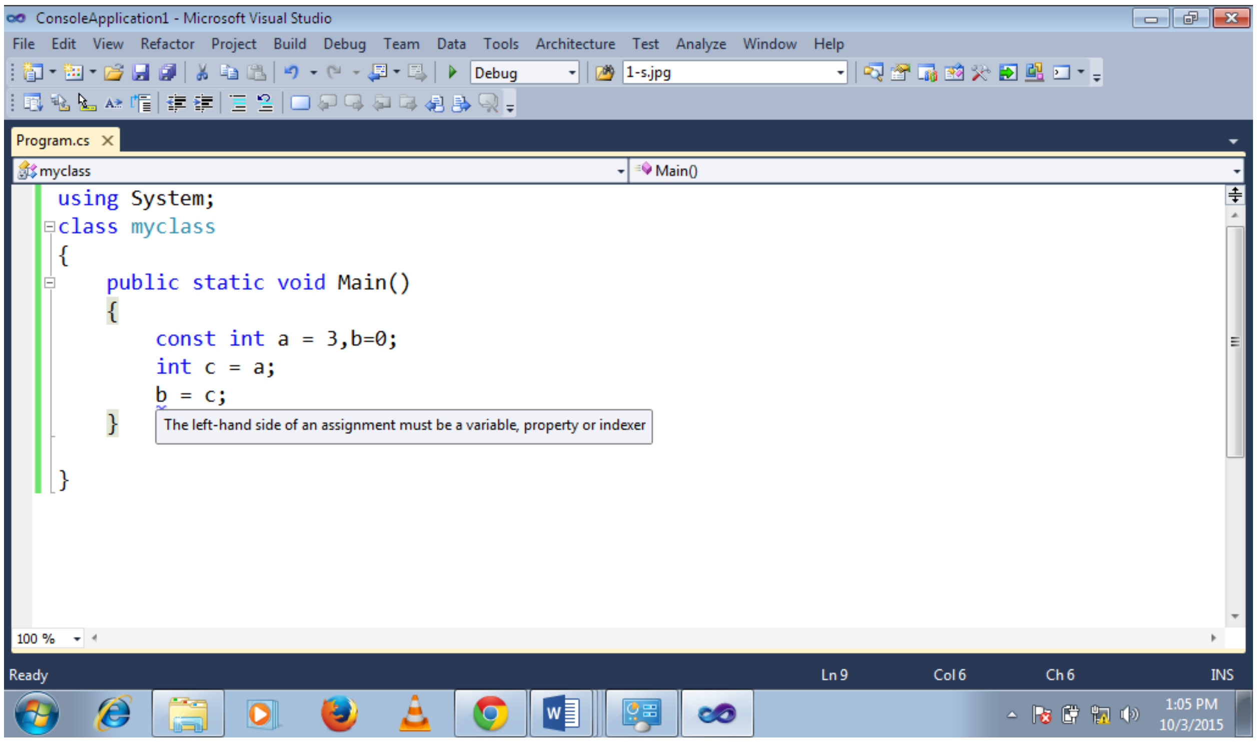
A const field requires a value to be provided

We have declared a constant but not initialized with any value , so it is giving the error, as coming in tool tip that " A const field requires a value to be provided". But in case of variable this error will not come because it is not compulsion to initialize it at the time of declaring

So the correct code for above is below:

```
 public static void Main()
{
     const int a = 3;}
```

Case 2: A variable can hold the value of constant but its reverse is not true. Take an example below:


ConsoleApplication1 - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Team Data Tools Architecture Test Analyze Window Help
Debug
1-s.jpg
Program.cs
myclass
Main()
100 %
Ready
Ln 9
Col 6
Ch 6
INS
1:05 PM
10/3/2015

```
using System;
class myclass
{
    public static void Main()
    {
        const int a = 3,b=0;
        int c = a;
        b = c;
    }
}
```

The left-hand side of an assignment must be a variable, property or indexer

In the above example , we have hold the value of a constant name “a” to variable “c” but it’s reverse is false, therefore it is giving error when we tried to hold the value of variable name “c” in constant “b”.

Case 3: You have to take care of while Performing calculations with constants like below:

```
public static void Main()
    {
        int a = 3, b = 4;
        const int c = a + b;
        Console.WriteLine(c);
    }
```

We should be very careful while performing calculations with constant fields because its result can be stored into constant or variables but its reverse it not true, hence you cannot store the sum of two variables into the variable. So the above code will give error because we have stored the sum of two variables into the constant which is not allowed.

Point to remember for Constants:

1. Const is nothing but "constant", a variable of which the value is constant but at compile time.
2. Constant fields aren't variables and may not be modified. Hence, Constants are immutable values which are known at compile time and do not change for the life of the program.
3. And it's mandatory to assign a value to it.
4. Constants can be of any in-built value types like numbers, Boolean values, float as well as reference types like string which can also be null. User-defined types, including classes, structs, and arrays, cannot be const.
5. You cannot make nullable types as constants.
6. A constant expression is an expression that can be fully evaluated at compile time. Therefore, the only possible values for constants of reference types are string and a null reference.
7. The constant declaration can declare multiple constants, such as: public const double x = 1.0, y = 2.0, z = 3.0;
8. The static modifier is not allowed in a constant declaration because it is by-default static in nature. Constants are accessed as if they were static fields because the value of the constant is the same for all instances of the type. You do not use the static keyword to declare them. Expressions that are not in the class that defines the constant must use the class name, a period, and the name of the constant to access the constant. For example:
9. Constants can also be declared and initialized as local , means inside the methods .
10. C# does not support const methods, properties, or events.
11. The enum type enables you to define named constants for integral built-in types (for example int, uint, long, and so on).
12. Take an example below :

```
class Calendar1
{
    public const int months = 12;
}
```

In this example, the constant months is always 12, and it cannot be changed even by the class itself. In fact, when the compiler encounters a constant identifier in C# source code (for example, months), it substitutes the literal value directly into the intermediate language (IL) code that it produces. Because there is no variable address associated with a constant at run time, const fields cannot be passed by reference and cannot appear as an l-value in an expression.

13. Constants can be marked as public, private, protected, internal, or protected internal. These access modifiers define how users of the class can access the constant. For more information.
14. Constants cannot be declared into parameters.

Points to Remember for ReadOnly Fields:

1. The readonly keyword is a modifier that you can use on fields. When a field declaration includes a readonly modifier, assignments to the fields introduced by the declaration can only occur as part of the declaration or in a constructor in the same class.
2. The readonly keyword is different from the const keyword. A const field can only be initialized at the declaration of the field. A readonly field can be initialized either at the declaration or in a constructor. Therefore, readonly fields can have different values depending on the constructor used. Also, while a const field is a compile-time constant, the readonly field can be used for runtime constants . See the example below :

```
public class ReadOnlyTest
{
    class SampleClass
    {
        public int x;
        // Initialize a readonly field
        public readonly int y = 25;
        public readonly int z;

        public SampleClass()
        {
            // Initialize a readonly instance field
            z = 24;
        }

        public SampleClass(int p1, int p2, int p3)
        {
            x = p1;
            y = p2;
            z = p3;
        }
    }
    static void Main()
    {
        SampleClass p1 = new SampleClass(11, 21, 32);   // OK
        Console.WriteLine("p1: x={0}, y={1}, z={2}", p1.x, p1.y, p1.z);
        SampleClass p2 = new SampleClass();
        p2.x = 55;   // OK
        Console.WriteLine("p2: x={0}, y={1}, z={2}", p2.x, p2.y, p2.z);
    }
}
```

/*

```
        Output:

            p1: x=11, y=21, z=32

            p2: x=55, y=25, z=24

    */
```

3. You can also make the readonly variable as static which can either be initialized at the time of declaration or in the static constructors.

4. Readonly fields can be initialized only while declaration or in the constructor.

5. Once you initialize a readonly field, you cannot reassign it.

6. You can use static modifier for readonly fields

7. Readonly modifier can be used with reference types

8. Readonly modifier can be used only for instance or static fields, you cannot use readonly keyword for variables in the methods.

Difference between Constants and ReadOnly Fields:

1. const fields has to be initialized while declaration only, while readonly fields can be initialized at declaration or in the constructor.

2. const variables can declared in methods ,while readonly fields cannot be declared in methods.

3. const fields cannot be used with static modifier, while readonly fields can be used with static modifier.

4. A const field is a compile-time constant, the readonly field can be used for run time constants.

5. Constant variables have to be accessed using "Classname.VariableName", Read only variables have to be accessed using the "InstanceName.VariableName", Static Read only variables have to be accessed using the "Classname.VariableName".

6. See the example:

```
class Test
{
    public string Name;
}

class myclass
{

    public static readonly Test test = new Test();
    // public const Test test1 = new Test(); //Error:  A const field of a reference type
other than string can only be initialized with null.

    static void Main(string[] args)
    {
        test.Name = "Program-1";

        //test = new Test(); // Error:A static readonly field cannot be assigned to
(except in a static constructor or a variable initializer)
        Console.WriteLine(test.Name);

    }
}
```

In the above example we have create the instance of class as static readonly . We are initializing the value of name by the instance of class. But as we are initializing the test with the new instance memory, than it is giving error. Hence it is static readonly so it can only be initialized at the time of declaration or in static member. Similarly while making reference type constant, than also it is giving error as shown below.

Now below is same program but with structure, see the example above:

```
struct Test
{
    public string Name;
}
```

```
class myclass
{

    public static readonly Test test = new Test();
    static myclass()
    {
        test.Name = "Program -1"; //OK
    }
    static void Main(string[] args)
    {

        test.Name = "Program -2"; // Error:A static readonly field cannot be assigned to
(except in a static constructor or a variable initializer) .
        // above value can be intitalized into static constructor

        Console.WriteLine(test.Name);

    }
}
```

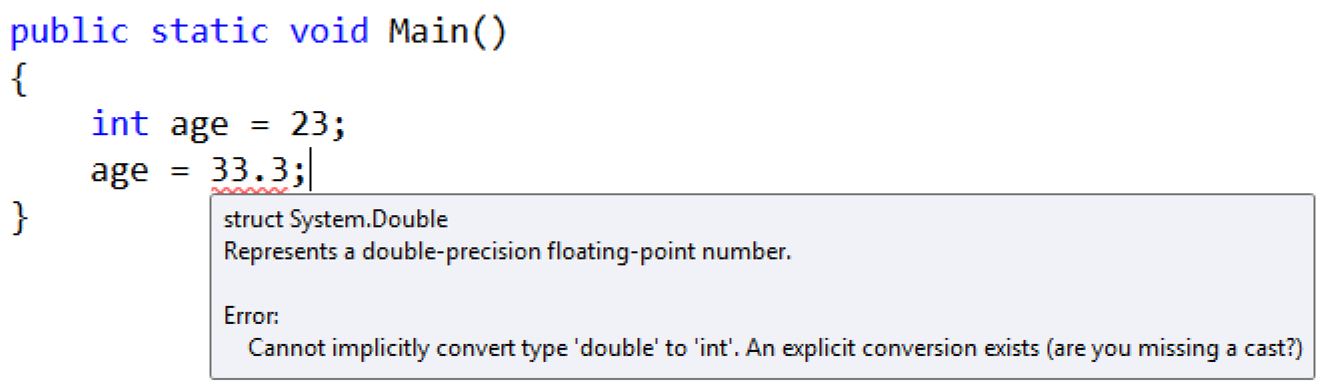
In the above , we have initialized a struct at the time of declaration . Because struct is value type, so it will give error on changing the value inside it's member variables after member variables are initialized in constructors. Irrespective of class which is reference type, which was giving error at the time of creating the new instance, struct is giving error while changing the value of it's member variable in Main function.

7. We can not make methods or properties as readonly.

Understanding Conversions and Casting

C# is a type-safe language and the type-safety property of any programming language ensures that a variable can only be accessed by the type associated with that variable. The pre-requisite of a type safe programming language is that you can perform only those operations on a type, which are valid for it. For example you want to add two numbers than it is sure that you need integer type values for performing operations not string or char. There are certain cases where we need to change one data type two another data type for operation purpose, in the case conversion is used.

C# is a statically typed language, therefore if you declare a variable as an int type, then you cannot assign double value to it. See the below example:



```
public static void Main()
{
    int age = 23;
    age = 33.3;
}
```

```
struct System.Double
Represents a double-precision floating-point number.

Error:
   Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)
```

It is generating the compile time error which is "Cannot implicitly convert type 'double' to 'int'."

This means you can use type conversions to convert the data of one data type into another data type. Type conversions can be of two types, as given below:

- **Implicit Conversion:** No special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.
- **Explicit Conversion (Casting):** Explicit conversions require a cast operator. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class.
- **User-defined Conversion:** User-defined conversions are performed by special methods that you can define to enable explicit and implicit conversions between custom types that do not have a base class–derived class relationship.
- **Conversions with helper classes:** To convert between non-compatible types, such as integers and System.DateTime objects, or hexadecimal strings and byte arrays, you can use the System.BitConverter class, the System.Convert class, and the Parse methods of the built-in numeric types such as Int32.Parse.

Implicit Conversion:

For built-in numeric types, an implicit conversion can be made when the value to be stored can fit into the variable without being truncated or rounded off. The built in numeric types can be assigned to each other if a narrow type is being assigned to wider type. This is possible because the compiler know that the only problem in such operations is that a little more memory will be needed to hold this type and no data will be truncated or lost. So the following conversion will not need any explicit cast. For example, a variable of type long (8 byte integer) can store any value that an int (4 bytes on a 32-bit computer) can store. In the following example, the compiler implicitly converts the value on the right to a type long before assigning it to bigNum.

```
             // Implicit conversion. num long can
            // hold any value an int can hold, and more!
            int num = 2147483647;
            long bigNum = num;
```

Secondly, If we try to assign a value from derived class to a base class it will work, that is because a derived class always is-a base class. Also, from a memory perspective a base class variable pointing to a derived class object can safely access the base class part of the object in memory without any problem. SO following code will work without needing any explicit casts.

```
using System;
class Base
{

}

class Derived : Base
{

}

class Program
{
    static void Main(string[] args)
    {
        Derived d = new Derived();
        Base b = d;
    }
}
```

Other than these two possible scenarios, all the conversions will create compile time errors. Still, If we need to perform the conversions, we will have to use explicit casting/conversion.

The following table shows the predefined implicit numeric conversions. Implicit conversions might occur in many situations, including method invoking and assignment statements.

From	To
sbyte	**short** , **int**, **long**, **float**, **double**, or **decimal**
byte	**short** , **ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double**, or **decimal**
short	**int** , **long**, **float**, **double**, or **decimal**
ushort	**int** , **uint**, **long**, **ulong**, **float**, **double**, or **decimal**
int	**long** , **float**, **double**, or **decimal**
uint	**long** , **ulong**, **float**, **double**, or **decimal**
long	**float** , **double**, or **decimal**
char	**ushort** , **int**, **uint**, **long**, **ulong**, **float**, **double**, or **decimal**
float	**double**
ulong	**float** , **double**, or **decimal**

Explicit Conversion:

If we find ourselves in need of conversion that is either narrowing conversion or conversion between unrelated types then we will have to use explicit conversions. Using explicit conversions we are actually letting the compiler know that we know there is possible information loss but still we need to make this conversion. So if we need to convert a long type to an integer type we need to cast it explicitly.

However, if a conversion cannot be made without a risk of losing information, the compiler requires that you perform an explicit conversion, which is called a cast. A cast is a way of explicitly informing the compiler that you intend to make the conversion and that you are aware that data loss might occur. To perform a cast, specify the type that you are casting to in parentheses in front of the value or variable to be converted. The following program casts a double to an int. The program will not compile without the cast.

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

See the table below which shows the possible explicit conversions:

From	To
sbyte	**byte** , **ushort**, **uint**, **ulong**, or **char**
byte	**Sbyte** or **char**
short	**sbyte** , **byte**, **ushort**, **uint**, **ulong**, or **char**
ushort	**sbyte** , **byte**, **short**, or **char**
int	**sbyte** , **byte**, **short**, **ushort**, **uint**, **ulong**, or **char**
uint	**sbyte** , **byte**, **short**, **ushort**, **int**, or **char**
long	**sbyte** , **byte**, **short**, **ushort**, **int**, **uint**, **ulong**, or **char**
ulong	**sbyte** , **byte**, **short**, **ushort**, **int**, **uint**, **long**, or **char**
char	**sbyte** , **byte**, or **short**
float	**sbyte** , **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, or **decimal**
double	**sbyte** , **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, **float**, or **decimal**

decimal	**sbyte** , **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, **float**, or **double**

For reference types, an explicit cast is required if you need to convert from a base type to a derived type:

```
using System;
class Animal { }
class Giraffe:Animal { }
class Program
{
    public static void Main()
    {
        // Create a new derived type.
        Giraffe g = new Giraffe();

        // Implicit conversion to base type is safe.
        Animal a = g;

        // Explicit conversion is required to cast back
        // to derived type. Note: This will compile but will
        // throw an exception at run time if the right-side
        // object is not in fact a Giraffe.
        Giraffe g2 = (Giraffe)a;
    }
}
```

A cast operation between reference types does not change the run-time type of the underlying object; it only changes the type of the value that is being used as a reference to that object.

Explicit casting actually tells the compiler that we know about possible information loss/mismatch but still we need to perform this cast. This is ok for inbuilt numeric types but in case of reference types, there is a possibility that the types are not at all compatible i.e. casting from one type to another is not at all possible. For example casting a string "abc" to Integer is not possible.

Such casting expressions will compile successfully but they will fail at run-time. What C# compiler does it that it checks whether these two types are cast compatible or not and if not it raises an exception InvalidCastException.

User-defined Conversion

C# also provides the flexibility for defining conversions on classes and structs so that they can be converted to and from other. The conversion operators simply contains the logic of how the conversion should happen. We can define these conversion operators to be implicit or explicit. If we define them implicit the conversion will happen without needing as explicit cast. If we define it as explicit the casting will be required. You can read more about it in the extra module file available with this file.

Conversions with helper classes

There are a lot of helper classes and helper functions available in C# to perform the frequently needed conversions. It is always a good idea to refer to the documentation to achieve the desired conversions before putting in the code for conversions.

We can use methods of Convert class or Parse method of available structures.

Convert Class:

Convert class is static class which provides various static functions for conversions. The static methods of the Convert class are used to support conversion to and from the base data types in the .NET Framework. The supported base types are Boolean, Char, SByte, Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Decimal, DateTime and String.

Conversion from Base to Base Types:

A conversion method exists to convert every base type to every other base type. However, the actual call to a particular conversion method can produce one of five outcomes, depending on the value of the base type at run time and the target base type.

These five outcomes are:

1. **No Conversion**: This occurs when an attempt is made to convert from a type to itself (for example, by calling Convert.ToInt32(Int32) with an argument of type Int32). In this case, the method simply returns an instance of the original type.

2. **An InvalidCastException**: This occurs when a particular conversion is not supported. An InvalidCastException is thrown for the following conversions:
    - Conversions from Char to Boolean, Single, Double, Decimal, or DateTime.
    - Conversions from Boolean, Single, Double, Decimal, or DateTime to Char.
    - Conversions from DateTime to any other type except String.
    - Conversions from any other type, except String, to DateTime.
3. **A FormatException:** This occurs when the attempt to convert a string value to any other base type fails because the string is not in the proper format. The exception is thrown for the following conversions:
    - A string to be converted to a Boolean value does not equal Boolean.TrueString or Boolean.FalseString.
    - A string to be converted to a Char value consists of multiple characters.
    - A string to be converted to any numeric type is not recognized as a valid number.
    - A string to be converted to a DateTime is not recognized as a valid date and time value.

4. **A successful conversion:** For conversions between two different base types not listed in the previous outcomes, all widening conversions as well as all narrowing conversions that do not result in a loss of data will succeed and the method will return a value of the targeted base type.
5. **An OverflowException**: This occurs when a narrowing conversion results in a loss of data. For example, trying to convert a Int32 instance whose value is 10000 to a Byte type throws an OverflowException because 10000 is outside the range of the Byte data type. An exception will not be thrown if the conversion of a numeric type results in a loss of precision (that is, the loss of some least significant digits). However, an exception will be thrown if the result is larger than can be represented by the particular conversion method's return value type. For example, when a Double is converted to a Single, a loss of precision might occur but no exception is thrown. However, if the magnitude of the Double is too large to be represented by a Single, an overflow exception is thrown.

Conversions from Custom Objects to Base Types

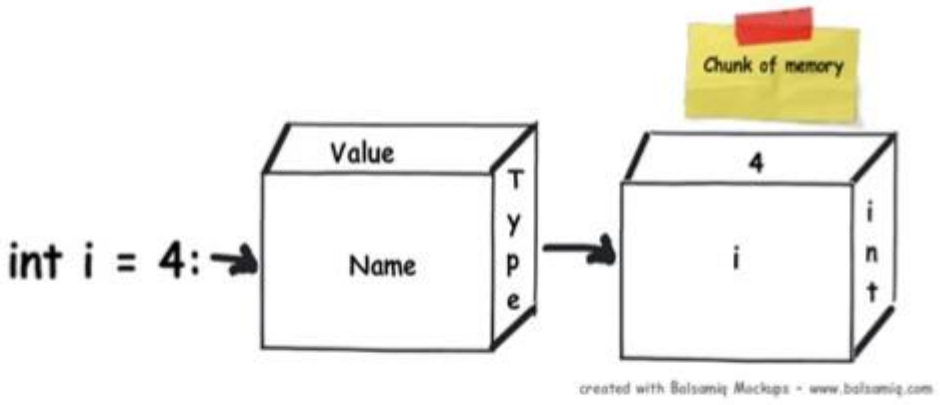
In addition to supporting conversions between the base types, the Convert method supports conversion of any custom type to any base type. To do this, the custom type must implement the IConvertible interface, which defines methods for converting the implementing type to each of the base types.

Conversions that are not supported by a particular type should throw an InvalidCastException.

When the ChangeType method is passed a custom type as its first parameter, or when the Convert.ToType method (such as Convert.ToInt32(Object) or Convert.ToDouble(Object, IFormatProvider) is called and passed an instance of a custom type as its first parameter, the Convert method, in turn, calls the custom type's IConvertible implementation to perform the conversion. For more information, see Type Conversion in the .NET Framework.

Understanding the memory allocations in Value Types and Reference Types

When you declare a variable in a .NET application, it allocates some chunk of memory in the RAM. This memory has three things: the name of the variable, the data type of the variable, and the value of the variable. That was a simple explanation of what happens in the memory, but depending on the data type, your variable is allocated that type of memory. There are two types of memory allocation: stack memory and heap memory. In the coming sections, we will try to understand these two types of memory in more detail.



See the example below:

```
public void Method1()
  {
      // Line 1
      int i = 4;
      // Line 2
      int y = 2;
      //Line 3
      class1 cls1 = new class1();
  }
```

When the above lines get executed, than below will happen internally:

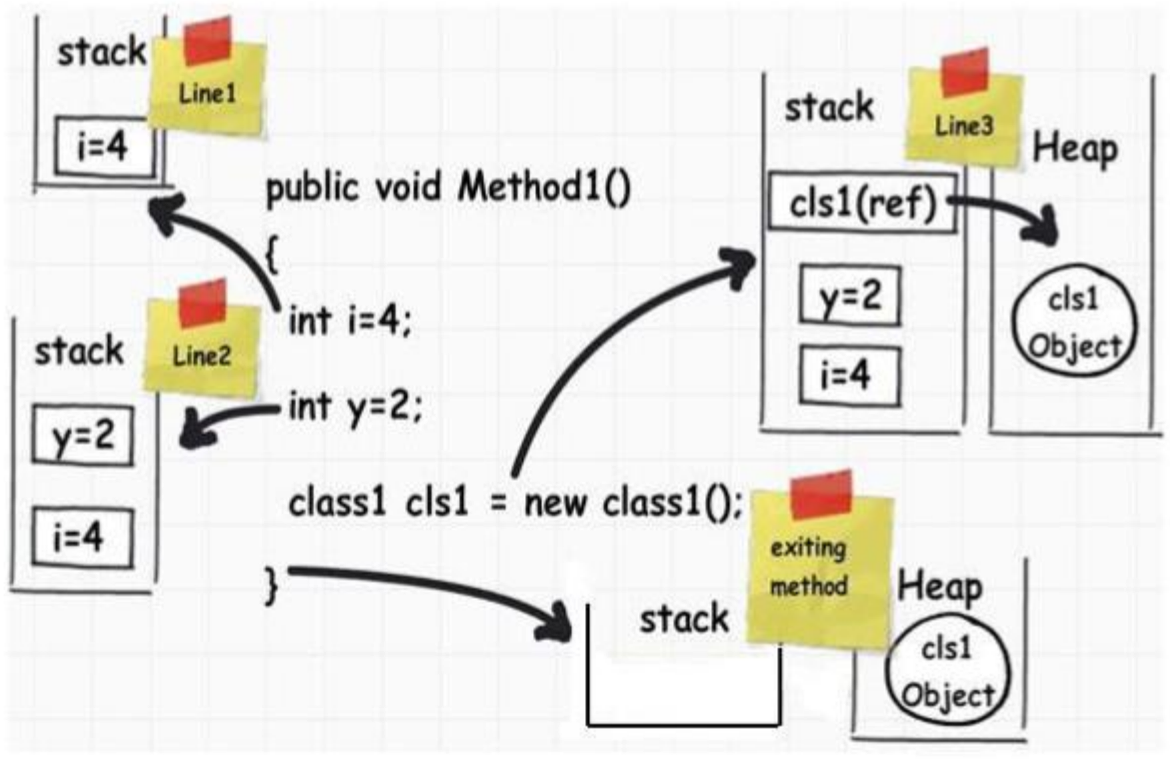
It’s a three line code, let’s understand line by line how things execute internally.

1. Line 1: When this line is executed, the compiler allocates a small amount of memory in the stack. The stack is responsible for keeping track of the running memory needed in your application.
2. Line 2: Now the execution moves to the next step. As the name says stack, it stacks this memory allocation on top of the first memory allocation. You can think about stack as a series of compartments or boxes put on top of each other. Memory allocation and de-allocation is done using LIFO (Last In First Out) logic. In other words memory is allocated and de- allocated at only one end of the memory, i.e., top of the stack.
3. Line 3: In line 3, we have created an object. When this line is executed it creates a pointer on the stack and the actual object is stored in a different type of memory location called ‘Heap’. ‘Heap’ does not track running memory, it’s just a pile of objects which can be reached at any moment of time. Heap is used for dynamic memory allocation.

One more important point to note here is reference pointers are allocated on stack. The statement, Class1 cls1; does not allocate memory for an instance of Class1, it only allocates a stack variable cls1 (and sets it to null). The time it hits the new keyword, it allocates on "heap". When it passes the end control, it clears all the memory variables which are assigned on stack. In other words all variables which are related to int data type are de-allocated in ‘LIFO’ fashion from the stack.

The big catch – It did not de-allocate the heap memory. This memory will be later de-allocated by the garbage collector.

See the graphical representation of above code:



Need of storing the reference type in heap and value type on stack:

If you look closely, primitive data types are not complex, they hold single values like 'int i = 0'. Object data types are complex, they reference other objects or other primitive data types. In other words, they hold reference to other multiple values and each one of them must be stored in memory. Object types need dynamic memory while primitive ones needs static type memory. If the requirement is of dynamic memory, it's allocated on the heap or else it goes on a stack.

Difference between Value type and Reference Type:

Value types are types which hold both data and memory on the same location. A reference type has a pointer which points to the memory location. See more differences:

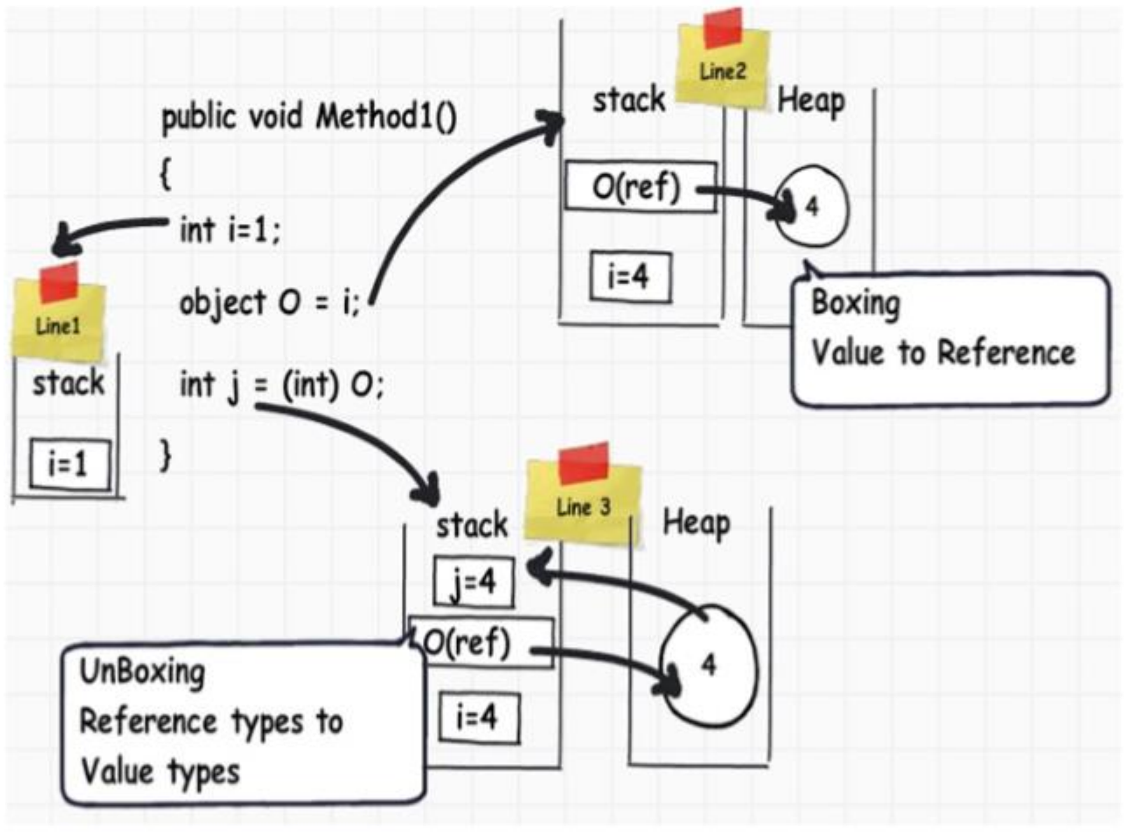
Value Type	**Reference Type**
They are stored on stack	They are stored on heap
Contains actual value	Contains reference to a value
Cannot contain null values. However this can be achieved by nullable types	Can contain null values.
Value type is popped on its own from stack when they go out of scope.	Required garbage collector to free memory.
Memory is allocated at compile time	Memory is allocated at run time

Boxing and Unboxing:

Boxing is the process of converting a value type to the reference type e.g. object or to any interface type implemented by this value type. When the CLR boxes a value type, it wraps the value inside a System.Object and stores it on the managed heap. Unboxing extracts the value type from the object. Boxing is implicit; unboxing is explicit. The concept of boxing and unboxing underlies the C# unified view of the type system in which a value of any type can be treated as an object.

Boxing and unboxing is an essential concept in .NET's type system. With Boxing and unboxing one can link between value-types and reference-types by allowing any value of a value-type to be converted to and from type object. Boxing and unboxing enables a unified view of the type system wherein a value of any type can ultimately be treated as an object.

Consider the below code snippet. When we move a value type to reference type, data is moved from the stack to the heap. When we move a reference type to a value type, the data is moved from the heap to the stack. This movement of data from the heap to stack and vice-versa creates a performance hit. When the data moves from value types to reference types, it is termed 'Boxing' and the reverse is termed 'UnBoxing'. See the graphical representation of boxing and unboxing below:


```
public void Method1()
{
  int i=1;
  object O = i;
  int j = (int) O;
}
```

In order to see how the performance is impacted, we ran the below two functions 10,000 times. One function has boxing and the other function is simple. We used a stop watch object to monitor the time taken. The boxing function was executed in 3542 ms while without boxing, the code was executed in 2477 ms. In other words try to avoid boxing and unboxing. In a project where you need boxing and unboxing, use it when it's absolutely necessary.

Working with operators and operator precedence

In C#, we can use the values stored inside the variables and constants in the form of an expression. An expression is nothing but a set of language elements, arranged together to perform a specific task or computation. To write expressions, C# 5.0 provides a complete set of language elements, such as variables, constants, operators, properties, and literals. A typical examples of an expression is shown in the following code snippet:

Int total = 2+3;

In the Preceding code snippet, expression comprises of an integer variable, total an assignment operator =, integer values 2 and 3 and the addition operator +. The evaluation of the expression yields the result 5, which is then assigned to the integer variable total.

Operators are nothing but special symbols that are used to specify some computation or other operations such as arithmetic and logical operations on the operands. Therefore, operations specify the operations to be performed in the expression. The operators in C# 5.0 falls under several categories, such as primary, unary, multiplicative, additive, shift, relational and type testing, equality, logical and many more. C# has rich set of built-in operators and provides the following type of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

Arithmetic Operators

Following table shows all the arithmetic operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B = 30
-	Subtracts second operand from the first	A - B = -10
*	Multiplies both operands	A * B = 200
/	Divides numerator by de-numerator	B / A = 2
%	Modulus Operator and remainder of after an integer division	B % A = 0

Unary Operators

There are various unary operators available out of which we are covering increment and decrement operators.   Consider the Value of A is 10 and B is 10.

++	Increment operator increases integer value by one	A++ = 11
--	Decrement operator decreases integer value by one	A-- = 9

Relational Operators

Following table shows all the relational operators supported by C#. Assume variable **A** holds 10 and variable **B** holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C#. Assume variable **A** holds Boolean value true and variable **B** holds Boolean value false, then:

Show Examples

Operator	Description	Example

&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit by bit operation. The truth tables for &, |, and ^ are as follows:

p	**q**	**p & q**	**p | q**	**p ^ q**
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; then in the binary format they are as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The Bitwise operators supported by C# are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = 61, which is 1100 0011 in 2's complement due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the	A >> 2 = 15, which is 0000 1111

	number of bits specified by the right operand.	

Assignment Operators

There are following assignment operators supported by C#:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B assigns value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent

		to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Operator Precedence in C#

Operator precedence determines the grouping of terms in an expression. This affects evaluation of an expression. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition and subtraction operator.

When an expression contains multiple operators, the precedence of the operators controls the order in which the individual operators are evaluated. For example, the expression x+y*z is evaluated as x + (y*z) because the multiplicative operator has higher precedence than additive operator.

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so the first evaluation takes place for 3*2 and then 7 is added into it.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators are evaluated first.

Category	**Operator**	**Associativity**
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right

Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Associativity: You can see the word associativity coming in the above table, what does it mean? When an operand occurs between two operators with the same precedence, the associativity of the operators controls the order in which the operations are performed. For example in addition operation like, x + y + z is evaluated as (x + y) + z. This is particularly due to left- associativity. On the other side for assignment operator, like x=y=z is evaluated as x = (y = z). This is particularly due to right-associativity.

Note:

A. Except for the assignment operators, all binary operators are left- associative, meaning that operations are performed from left to right.
B. The assignment operators and the conditional operator (?:) are right- associative, meaning that operations are performed from right to left.

You can control precedence and associativity by using parentheses. For example, x + y * z first multiplies y by z and then adds the result to x, but (x + y) * z first adds x and y and then multiplies the result by z.

Escape Sequences

To display special characters, such as the new line character or the backspace character, you need to include appropriate escape sequences in your code. An escape sequence is a combination of characters consisting of a backslash (\) followed by a letter or a combination of digits. The following table lists some of the escape sequences provided in C#.

Escape Sequence	***Sequence Description***
\’	*Single quotation mark*
\ "	*Double quotation mark*
\\	*Backslash*
\0	*NULL*

\a	*Alert*
\b	*Backspace*
\n	*New line*
\r	*Carriage return*
\t	*Horizontal tab*
\v	*Vertical tab*

Understanding implicitly typed local variables (var keyword)

Beginning in Visual C# 3.0, variables that are declared at method scope can have an implicit type **var**. An implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type. The following two declarations of i are functionally equivalent:

```
var i = 10; // implicitly typed
        int i = 10; //explicitly typed
```

Local variables can be given an inferred "type" of var instead of an explicit type. The var keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialization statement. The inferred type may be a built-in type, an anonymous type, a user-defined type, or a type defined in the .NET Framework class library.

Just take an example below:

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };
```

It is important to understand that the **var** keyword does not mean "variant" and does not indicate that the variable is loosely typed, or late-bound. It just means that the compiler determines and assigns the most appropriate type.

The **var** keyword may be used in the following contexts:

On local variables (variables declared at method scope) as shown in the previous example.

- In a for initialization statement. E.g. for(var x = 1; x < 10; x++)

- In a foreach initialization statement. E.g. foreach(var item in list){...}
- In a using statement. E.g. using (var file = new StreamReader("C:\\myfile.txt")) {...}

The following restrictions apply to implicitly-typed variable declarations:

- **var** can only be used when a local variable is declared and initialized in the same statement; the variable cannot be initialized to null, or to a method group or an anonymous function.
- **var** cannot be used on fields at class scope.
- Variables declared by using **var** cannot be used in the initialization expression. In other words, this expression is legal: int i = (i = 20); but this expression produces a compile-time error: var i = (i = 20);
- Multiple implicitly-typed variables cannot be initialized in the same statement.
- If a type named var is in scope, then the **var** keyword will resolve to that type name and will not be treated as part of an implicitly typed local variable declaration.

Understanding dynamic keyword:

Visual C# 2010 introduces a new type, **dynamic**. The type is a static type, but an object of type **dynamic** bypasses static type checking. In most cases, it functions like it has type **object**. At compile time, an element that is typed as **dynamic** is assumed to support any operation.

The dynamic language runtime (DLR) is a new API in .NET Framework 4. It provides the infrastructure that supports the **dynamic** type in C#, and also the implementation of dynamic programming languages such as IronPython and IronRuby.

The **dynamic** type enables the operations in which it occurs to bypass compile-time type checking. Instead, these operations are resolved at run time.

The following example contrasts a variable of type **dynamic** to a variable of type **object**. To verify the type of each variable at compile time, place the mouse pointer over dyn or obj in the **WriteLine** statements. IntelliSense shows **dynamic** for dyn and **object** for obj.

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
```

```
            object obj = 1;

            // Rest the mouse pointer over dyn and obj to see their
            // types at compile time.
            System.Console.WriteLine(dyn.GetType());
            System.Console.WriteLine(obj.GetType());
        }
    }
```

The WriteLine statements display the run-time types of dyn and obj. At that point, both have the same type, integer. The following output is produced:

System.Int32

System.Int32

To see the difference between dyn and obj at compile time, add the following two lines between the declarations and the **WriteLine** statements in the previous example.

dyn = dyn + 3;

obj = obj + 3;

A compiler error is reported for the attempted addition of an integer and an object in expression obj + 3. However, no error is reported for dyn + 3. The expression that contains dyn is not checked at compile time because the type of dyn is **dynamic**.

Similarly take an example below , where we have used same dynamic type for storing different kind of values :

```
dynamic a = 5;
                Console.WriteLine(a); // output : 5
            a = "Hello";
  Console.WriteLine(a); // output : Hello
```

Points for dynamic:

1. It can be used in declarations, as the type of a property, field, indexer, parameter, return value, local variable, or type constraint. The following class definition uses **dynamic** in several different declarations. E.g

```
class ExampleClass
{
    // A dynamic field.
```

```
        static dynamic field;

        // A dynamic property.
        dynamic prop { get; set; }

        // A dynamic return type and a dynamic parameter type.
        public dynamic exampleMethod(dynamic d)
        {
            // A dynamic local variable.
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
```

2. Dynamic does not give any compile time error , if any error exists but will give run time error.Take an example below :

```
using System;

namespace DynamicExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.exampleMethod(10));
            Console.WriteLine(ec.exampleMethod("value"));

            // The following line causes a compiler error because exampleMethod
            // takes only one argument.
            //Console.WriteLine(ec.exampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.exampleMethod(10));

            // Because dynamic_ec is dynamic, the following call to exampleMethod
            // with two arguments does not produce an error at compile time.
            // However, itdoes cause a run-time error.
            //Console.WriteLine(dynamic_ec.exampleMethod(10, 4));
        }
    }
    class ExampleClass
    {
        static dynamic field;
        dynamic prop { get; set; }

        public dynamic exampleMethod(dynamic d)
        {
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}
// Results:
// Local variable
// 2
// Local variable
```

Difference between dynamic and var:

var	dynamic
Introduced in **C# 3.0(.NET 3.5 & VS 2008)**	Introduced in **C# 4.0 (.Net 4.0 & VS 2010)**
Statically typed – This means the type of variable declared is decided by the compiler at compile time.	**Dynamically typed** - This means the type of variable declared is decided by the compiler at runtime time.
Need to initialize at the time of declaration. e.g., `var str=”I am a string”;` Looking at the value assigned to the variable `str`, the compiler will treat the variable `str` as string.	**No need** to initialize at the time of declaration. e.g., `dynamic str;` `str=”I am a string”;` //Works fine and compiles `str=2;` //Works fine and compiles
Errors are caught at compile time.	**Errors are caught at runtime**

var	**dynamic**
Since the compiler knows about the type and the methods and properties of the type at the compile time itself	Since the compiler comes to about the type and the methods and properties of the type at the run time.
Visual Studio shows intellisense since the type of variable assigned is known to compiler.	**Intellisense is not available** since the type and its related methods and properties can be known at run time only
e.g., `var obj1;` will **throw a compile error** since the variable is not initialized. The compiler needs that this variable should be initialized so that it can infer a type from the value.	e.g., `dynamic obj1;` **will compile;**
e.g. `var obj1=1;` will compile `var obj1=” I am a string”;` **will throw error** since the compiler has already decided that the type of obj1 is System.Int32 when the value 1 was assigned to it. Now assigning a string value to it violates the type safety.	e.g. `dynamic obj1=1;` will compile and run `dynamic obj1=” I am a string”;` **will compile and run** since the compiler creates the type for obj1 as System.Int32 and then recreates the type as string when the value “I am a string” was assigned to it. This code will work fine.