ASP.NET USING C# PART-1

SYLLABUS

UNIT	TOPICS	PAGE NO
Unit-I	Review of .NET frameworks, Introduction to C#, Variables and expressions, flow controls, functions, debugging and error handling, OOPs with C#, Defining classes and class members.	1
Unit-II	Assembly, Components of Assembly, Private and Shared Assembly, Garbage Collector, JIT compiler. Namespaces Collections, Delegates and Events. Introduction to ASP.NET 4: Microsoft.NET framework, ASP.NET lifecycle. CSS: Need of CSS, Introduction to CSS, Working with CSS with visual developer.	40
Unit-III	ASP.NET server controls: Introduction, How to work with button controls, Textboxes, Labels, checkboxes and radio buttons, list controls and other web server controls, web.config and global.asax files. Programming ASP.NET web pages: Introduction, data types and variables, statements, organizing code, object oriented basics.	65
Unit-IV	Validation Control: Introduction, basic validation controls, validation techniques, using advanced validation controls. State Management: Using view state, using session state, using application state, using cookies and URL encoding. Master Pages: Creating master pages, content pages, nesting master pages, accessing master page controls from a content page. Navigation: Introduction to use the site navigation, using site navigation controls.	73
Unit-V	Databases: Introduction, using SQL data sources, GridView Control, DetailsView and FormView Controls, ListView and DataPager controls, Using object datasources. ASP.NET Security: Authentication, Authorization, Impersonation, ASP.NET provider model	88
Unit-VI	LINQ: Operators, implementations, LINQ to objects,XML,ADO.NET, Query Syntax. ASP.NET Ajax: Introducing AJAX, Working of AJAX, Using ASP.NET AJAX server controls. JQuery: Introduction to JQuery, JQuery UI Library, Working of JQuery	131

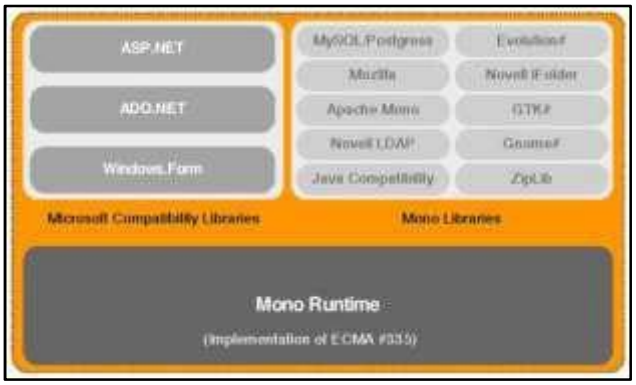
UNIT 1

Q: Reviews of .NET Framework?

Ans: is a revolutionary platform created by Microsoft for developing applications.

note that it doesn"t „develop applications on the Windows operating system."" Although the Microsoft release of the .NET Framework runs on the Windows operating system, it is possible to find alternative versions that will work on other systems.

One example of this is Mono, an open-source version of the .NET Framework (including a C# compiler) that runs on several operating systems, including various flavors of Linux and Mac OS you can use the Microsoft .NET Compact Framework



.NET Framework enables the creation of Windows applications, Web applications, Web services, and pretty much anything else you can think of.

The .NET Framework has been designed so that it can be used from any language, including C#, C++, Visual Basic, JScript, and even older languages such as COBOL.

Not only all of these languages have access to the .NET Framework, but they can also communicate with each other. It is perfectly possible for C# Developers to make use of code written by Visual Basic programmers, and vice versa.

Q: What is .NET Framework?

The .NET Framework consists primarily of a gigantic library of code that you use from your client languages (such as C#) using object-oriented programming (OOP) techniques.

This library is categorized into different modules — you use portions of it depending on the results you want to achieve.

one module contains the building blocks for Windows applications, another for Network programming, and another for Web development

.NET Framework library defines some basic types. A type is a representation of data, and specifying some of the most fundamental of these (such as „a 32-bit signed integer"") facilitates interoperability between languages using the .NET Framework. This is called the Common Type System (CTS).

As well as supplying this library, the .Net Framework also includes the .NET Common Language Runtime (CLR), which is responsible for maintaining the execution of all applications developed using the .NET library

In order for C# code to execute, it must be converted into a language that the target operating system understands, known as native code.

This conversion is called compiling code, an act that is performed by a compiler. Under the .NET Framework, this is a two-stage process.

Q: CIL and JIT

When you compile code that uses the .NET Framework library, you don"t immediately create operating-system-specific native code. Instead, you compile your code into Common Intermediate Language (CIL) code. This code isn"t specific to any operating system (OS) and isn"t specific to C#. Other .NET languages — Visual Basic .NET, for example — also compile to this language as a first stage. This compilation step is carried out by VS or VCE when you develop C# applications.

Obviously, more work is necessary to execute an application. That is the job of a just-in-time (JIT) compiler, which compiles CIL into native code that is specific to the OS and machine architecture being targeted. Only at this point can the OS execute the application.

In the past, it was often necessary to compile your code into several applications, each of which targeted a specific operating system and CPU architecture. Typically, this was a form of optimization (to get code to run faster on an AMD chipset, for example), but at times it was critical (for applications to work in both Win9x and WinNT/2000 environments, for example). This is now unnecessary, because JIT compilers (as their name suggests) use CIL code, which is independent of the machine, operating system, and CPU. Several JIT compilers exist, each targeting a different architecture, and the appropriate one is used to create the native code required.

The beauty of all this is that it requires a lot less work on your part — in fact, you can forget about system-dependent details and concentrate on the more interesting functionality of your code.

Q: Explain Assemblies?

When you compile an application, the CIL code created is stored in an assembly. Assemblies include both executable application files that you can run directly from Windows without the need for any other programs (these have a .exe file extension) and libraries (which have a .dll extension) for use by other applications.

In addition to containing CIL, assemblies also include meta information (that is, information about the information contained in the assembly, also known as metadata) and optional resources (additional data used by the CIL, such as sound files and pictures). The meta information enables assemblies to be fully self-descriptive. You need no other information to use an assembly, meaning you avoid situations such as failing to add required data to the system registry and so on, which was often a problem when developing with other platforms.

This means that deploying applications is often as simple as copying the files into a directory on a remote computer. Because no additional information is required on the target systems, you can just run an executable file from this directory and (assuming the .NET CLR is installed) you"re good to go.

Of course, you won"t necessarily want to include everything required to run an application in one place. You might write some code that performs tasks required by multiple applications. In situations like that, it is often useful to place the reusable code in a place accessible to all applications. In the .NET Framework, this is the global assembly cache (GAC). Placing code in the GAC is simple — you just place the assembly containing the code in the directory containing this cache.

Q: Managed Code

The role of the CLR doesn"t end after you have compiled your code to CIL and a JIT compiler has compiled that to native code. Code written using the .NET Framework is managed when it is executed (a stage usually referred to as runtime). This means that the CLR looks after your applications by managing memory, handling security, allowing cross-language debugging, and so on. By contrast, applications that do not run under the control of the CLR are said to be unmanaged, and certain languages such as C++ can be used to write such applications, which, for example, access low-level functions of the operating system. However, in C# you can write only code that runs in a managed environment. You will make use of the managed features of the CLR and allow .NET itself to handle any interaction with the operating system.

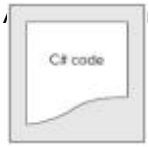
Q: Garbage Collection

One of the most important features of managed code is the concept of garbage collection. This is the .NET method of making sure that the memory used by an application is freed up completely when the application is no longer in use. Prior to .NET this was mostly the responsibility of programmers, and a few simple errors in code could result in large blocks of memory mysteriously disappearing as a result of being allocated to the wrong place in memory. That usually meant a progressive slowdown of your computer followed by a system crash.

N.ET garbage collection works by periodically inspecting the memory of your computer and removing anything from it that is no longer needed. There is no set time frame for this; it might happen thousands of times a second, once every few seconds, or whenever, but you can rest assured that it will happen.

Q: steps required to create a .NET application

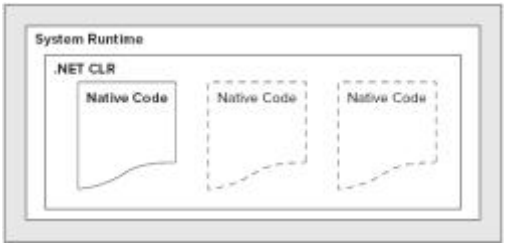
Ans:

1. A 

code is written using a .NET-compatible language
2. That code is compiled into CIL, which is stored in an assembly


3. When this code is executed it must first be compiled into native code using a JIT compiler


4. The native code is executed in the context of the managed CLR



Q: Linking

Note one additional point concerning this process. The C# code that compiles into CIL in step 2 needn"t be contained in a single file. It"s possible to split application code across multiple source code files, which are then compiled together into a single assembly. This extremely useful process is known as *linking*. It is required because it is far easier to work with several smaller files than one enormous one. You can separate out logically related code into an individual file so that it can be worked on independently and then practically forgotten about when completed. This also makes it easy to locate specific pieces of code when you need them and enables teams of developers to divide the programming burden into manageable chunks, whereby individuals can ""check out"" pieces of code to work on without risking damage to otherwise satisfactory sections or sections other people are working on.

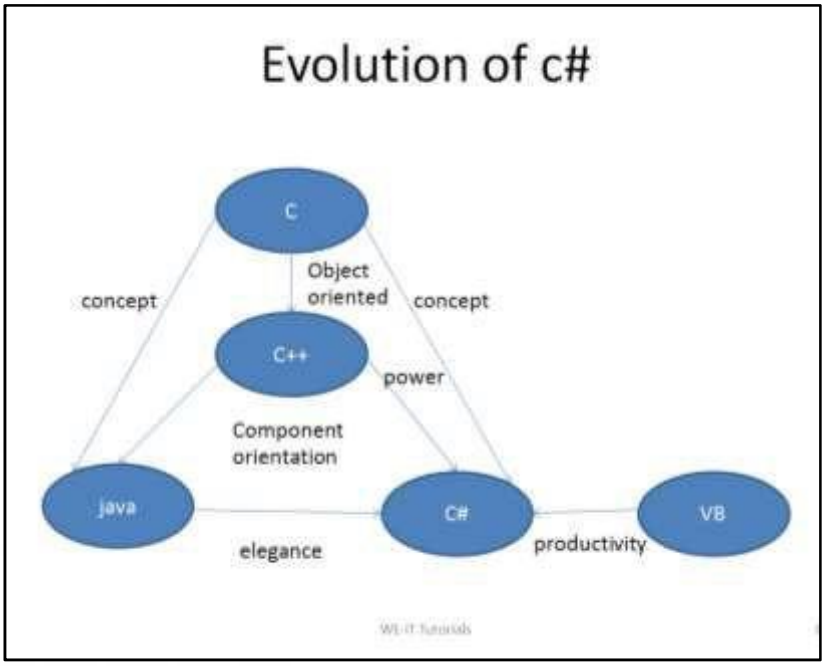
Q: Introduction to C#

Ans: C# is a relatively new language that was unveiled to the world when Microsoft announced the first version of its .NET Framework in July 2000.

Since then its popularity has rocketed and it has arguably become the language of choice for both Windows and Web developers who use the .NET framework.

Part of the appeal of C# comes from its clear syntax, which derives from C/C++ but simplifies some things that have previously discouraged some programmers.

- Fully object oriented language like java and first component oriented language
- Designed to support key features of .NET framework
- Its is simple, effective, productive, type-safe
- Belongs to family of C/C++
- It is designed to build robust, reliable and durable components to handle real world applications

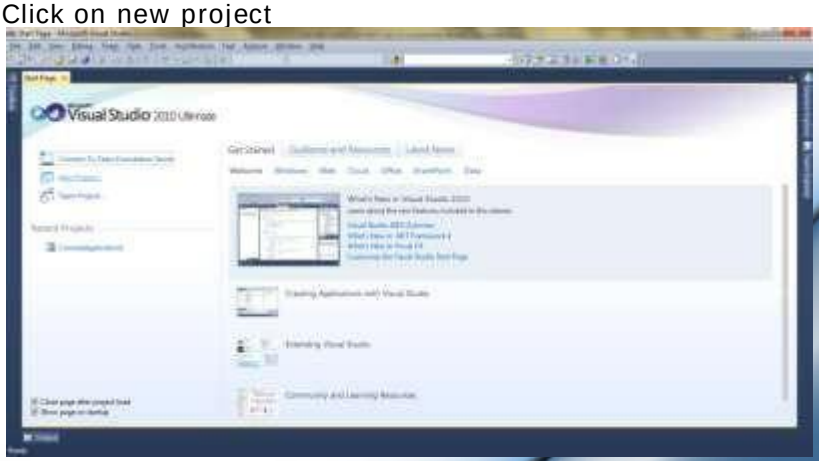


Q: starting with VISUAL STUDIO 2010

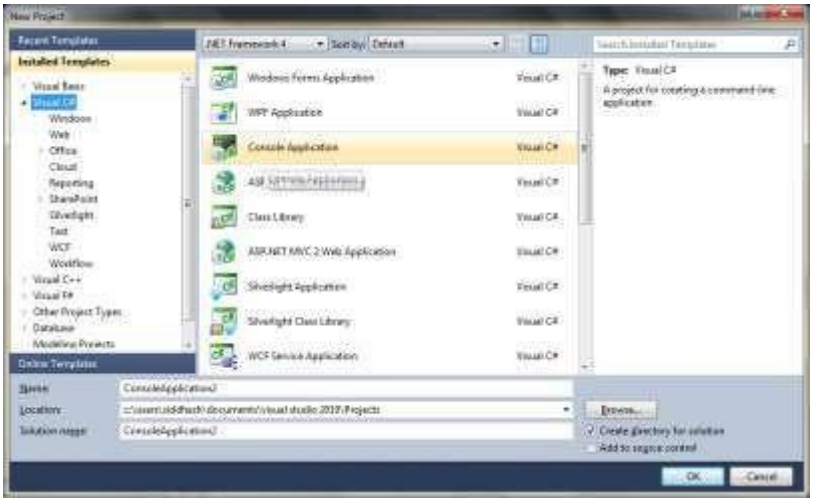
Ans :

1. Start visual studio 2010 IDE

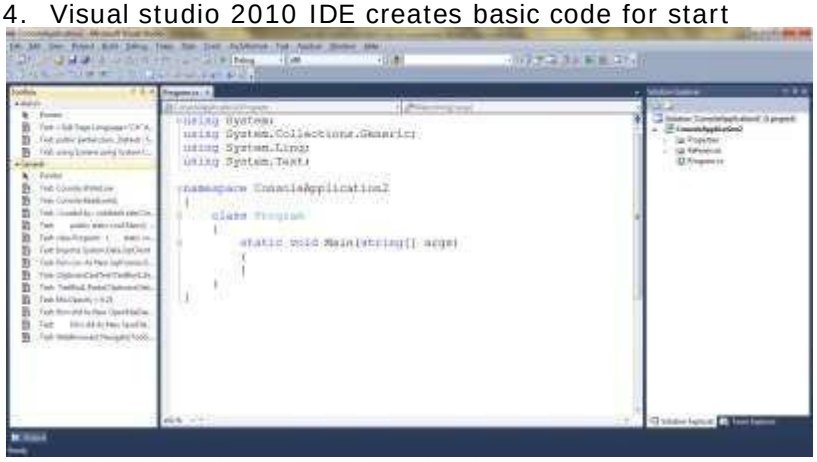
2. Click on new project



3. Click on visual c# language, click on console application



4. Visual studio 2010 IDE creates basic code for start



These are the namespaces imported using using keyword

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


namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
-----------------------------------------------------------
namespace ConsoleApplication2

      //works like a container which holds namespaces and classes inside it.
-------------------------------------------------
class Program

      //It is a class which can hold classes and methods, function, fields etc.

static void Main(string[] args)
        {
```

```
        }

      //This is a Method from which the execution starts
-------------------------------------------
```

Variables and Expressions

BASIC C# SYNTAX

The look and feel of C# code is similar to that of C++ and Java.

C# code is made up of a series of statements, each of which is terminated with a semicolon. Because whitespace is ignored, multiple statements can appear on one line, although for readability it is usual to add carriage returns after semicolons, to avoid multiple statements on one line.

C# is a block-structured language, meaning statements are part of a block of code. These blocks, which are delimited with curly brackets ({ and }), may contain any number of statements, or none at all.

COMMENTS IN C#

```
/* This is a comment */
```

Multiline comment

```
/* And so...

... is this! *
```

Single line comment

```
// this is comment line
```

code outlining functionality

You can do this with the #region and #endregion

```
#region Using directives
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
#endregion
```

C# syntax for declaring variables

```
<type> <name>;
String ename
```

number of different integer types can be used to store various ranges of numbers
u characters before some variable names are shorthand for *unsigned*

TYPE	ALIAS FOR	ALLOWED VALUES
sbyte	System.SByte	Integer between –128 and 127
byte	System.Byte	Integer between 0 and 255
short	System.Int16	Integer between –32768 and 32767
ushort	System.UInt16	Integer between 0 and 65535
int	System.Int32	Integer between –2147483648 and 2147483647
uint	System.UInt32	Integer between 0 and 4294967295
long	System.Int64	Integer between –9223372036854775808 and 9223372036854775807
ulong	System.UInt64	Integer between 0 and 18446744073709551615

floating-point values

TYPE	ALIAS FOR	MIN M	MAX M	MIN E	MAX E	APPROX. MIN VALUE	APPROX. MAX VALUE
float	System.Single	0	2^{24}	–149	104	1.5×10^{-45}	3.4×10^{38}
double	System.Double	0	2^{53}	–1075	970	5.0×10^{-324}	1.7×10^{308}
decimal	System.Decimal	0	2^{96}	–28	0	1.0×10^{-28}	7.9×10^{28}

three other simple types

TYPE	ALIAS FOR	ALLOWED VALUES
char	System.Char	Single Unicode character, stored as an Integer between 0 and 65535
bool	System.Boolean	Boolean value, true or false
string	System.String	A sequence of characters

Declaring and assigning values to variables

```
class Program
    {
        static void Main(string[] args)
        {
            string strEname; //declaring variable
            strEname = "We-It tutorials"; //initalizing value
            Console.WriteLine(strEname); //printing value of variable
        }
    }
```

Naming Conventions

two naming conventions are used in the .NET Framework namespaces: *PascalCase* and *camelCase*.

camelCase variable names:

age
firstName
timeOfDeath

These are PascalCase:

Age
LastName
WinterOfDiscontent

Literal Values

TYPE(S)	CATEGORY	SUFFIX	EXAMPLE/ALLOWED VALUES
`bool`	Boolean	None	`true` or `false`
`int`, `uint`, `long`, `ulong`	Integer	None	`100`
`uint`, `ulong`	Integer	`u` or `U`	`100U`
`long`, `ulong`	Integer	`l` or `L`	`100L`
`ulong`	Integer	`ul`, `uL`, `Ul`, `UL`, `lu`, `lU`, `Lu`, or `LU`	`100UL`
`float`	Real	`f` or `F`	`1.5F`
`double`	Real	None, `d`, or `D`	`1.5`
`decimal`	Real	`m` or `M`	`1.5M`
`char`	Character	None	`'a'`, or escape sequence
`string`	String	None	`"a ... a"`, may include escape sequences

String Literals

ESCAPE SEQUENCE	CHARACTER PRODUCED	UNICODE VALUE OF CHARACTER
`\'`	Single quotation mark	0x0027
`\"`	Double quotation mark	0x0022
`\\`	Backslash	0x005C
`\0`	Null	0x0000
`\a`	Alert (causes a beep)	0x0007
`\b`	Backspace	0x0008
`\f`	Form feed	0x000C
`\n`	New line	0x000A
`\r`	Carriage return	0x000D
`\t`	Horizontal tab	0x0009
`\v`	Vertical tab	0x000B

Eg :

If we want to print

```
"sid"s place"
```

Solution use escape character

```
"sid\"s\ place"
```

EXPRESSIONS

By combining operators with variables and literal values (together referred to as *operands* when used with operators), you can create *expressions*, which are the basic building blocks of computation

Unary— Act on single operands
Binary—Act on two operands
Ternary—Act on three operands

Mathematical Operators

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
+	Binary	`var1 = var2 + var3;`	var1 is assigned the value that is the sum of var2 and var3.
-	Binary	`var1 = var2 - var3;`	var1 is assigned the value that is the value of var3 subtracted from the value of var2.
*	Binary	`var1 = var2 * var3;`	var1 is assigned the value that is the product of var2 and var3.
/	Binary	`var1 = var2 / var3;`	var1 is assigned the value that is the result of dividing var2 by var3.
%	Binary	`var1 = var2 % var3;`	var1 is assigned the value that is the remainder when var2 is divided by var3.
+	Unary	`var1 = +var2;`	var1 is assigned the value of var2.
-	Unary	`var1 = -var2;`	var1 is assigned the value of var2 multiplied by -1.
++	Unary	`var1 = ++var2;`	var1 is assigned the value of var2 + 1. var2 is incremented by 1.
--	Unary	`var1 = --var2;`	var1 is assigned the value of var2 - 1. var2 is decremented by 1.
++	Unary	`var1 = var2++;`	var1 is assigned the value of var2. var2 is incremented by 1.
--	Unary	`var1 = var2--;`	var1 is assigned the value of var2. var2 is decremented by 1.

Assignment Operators

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
=	Binary	`var1 = var2;`	`var1` is assigned the value of `var2`.
+=	Binary	`var1 += var2;`	`var1` is assigned the value that is the sum of `var1` and `var2`.
-=	Binary	`var1 -= var2;`	`var1` is assigned the value that is the value of `var2` subtracted from the value of `var1`.
*=	Binary	`var1 *= var2;`	`var1` is assigned the value that is the product of `var1` and `var2`.
/=	Binary	`var1 /= var2;`	`var1` is assigned the value that is the result of dividing `var1` by `var2`.
%=	Binary	`var1 %= var2;`	`var1` is assigned the value that is the remainder when `var1` is divided by `var2`.

Operator Precedence

PRECEDENCE	OPERATORS
Highest	++, -- (used as prefixes); +, - (unary) *, /, % +, - =, *=, /=, %=, +=, -=
Lowest	++, -- (used as suffixes)

Namespaces

namespaces. These are the .NET way of providing containers for application code, such that code and its contents may be uniquely identified. Namespaces are also used as a means of categorizing items in the .NET Framework.

```
namespace outer
{
    namespace inner
    {
        namespace moreinner
        {
        }
    }
}
```

Using namespaces

```
using outer.inner.moreinner;
```

Flow Control

Boolean logic and how to use it
How to branch code
How to loop code

BOOLEAN LOGIC

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
==	Binary	var1 = var2 == var3;	var1 is assigned the value true if var2 is equal to var3, or false otherwise.
!=	Binary	var1 = var2 != var3;	var1 is assigned the value true if var2 is not equal to var3, or false otherwise.
<	Binary	var1 = var2 < var3;	var1 is assigned the value true if var2 is less than var3, or false otherwise.
>	Binary	var1 = var2 > var3;	var1 is assigned the value true if var2 is greater than var3, or false otherwise.
<=	Binary	var1 = var2 <= var3;	var1 is assigned the value true if var2 is less than or equal to var3, or false otherwise.
>=	Binary	var1 = var2 >= var3;	var1 is assigned the value true if var2 is greater than or equal to var3, or false otherwise.
!	Unary	var1 = !var2;	var1 is assigned the value true if var2 is false, or false if var2 is true. (Logical NOT)
&	Binary	var1 = var2 & var3;	var1 is assigned the value true if var2 and var3 are both true, or false otherwise. (Logical AND)
|	Binary	var1 = var2 | var3;	var1 is assigned the value true if either var2 or var3 (or both) is true, or false otherwise. (Logical OR)
^	Binary	var1 = var2 ^ var3;	var1 is assigned the value true if either var2 or var3, but not both, is true, or false otherwise. (Logical XOR or exclusive OR)

***conditional Boolean* operators**

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
&&	Binary	var1 = var2 && var3;	var1 is assigned the value true if var2 and var3 are both true, or false otherwise. (Logical AND)
||	Binary	var1 = var2 || var3;	var1 is assigned the value true if either var2 or var3 (or both) is true, or false otherwise. (Logical OR)

THE GOTO STATEMENT

C# enables you to label lines of code and then jump straight to them using the goto statement

```
goto <labelName>;
```

```
class Program
 {
    static void Main(string[] args)
    {
       string strEname; //declaring variable
       goto jumpon;
       strEname ="sid\"s place"; //initalizing value
       Console.WriteLine(strEname); //printing value of variable
       jumpon:
       Console.ReadKey();
    }
 }
```

Goto jumpon; will directly jump on label jumpon: escaping between code.

The Ternary Operator

```
<test> ? <resultIfTrue>: <resultIfFalse>
string a = (1 == 1) ? "its true" : "its false";

output:
      value of a “its true”
```

The if Statement

```
if (<test>)
<code executed if <test> is true>;

if (1 == 1)
{
  Console.WriteLine(“condition true”);
}
```

The If else

```
if (<test>)
<code executed if <test> is true>;
else
<code executed if <test> is false>;

if (1 == 1)
{
    Console.WriteLine("condition true");// condition true
}
else
{
    Console.WriteLine("condition false");
}
```

Checking More Conditions Using if Statements

```
if (1 == 2)
  {
          Console.WriteLine("condition true");
  }
  else
  {
          Console.WriteLine("condition false");
```

```
        if (2 == 2)
        {
        Console.WriteLine("inside next condition check");
        }
    }
```

The switch Statement

```
switch (<testVar>)
{
case <comparisonVal1>:
<code to execute if <testVar> == <comparisonVal1> >
break;
case <comparisonVal2>:
<code to execute if <testVar> == <comparisonVal2> >
break;
default:
<code to execute if <testVar> != comparisonVals>
break;
}
```

```
int a = 2;
switch (a)
{
    case 1:
      Console.WriteLine("one");
      break;
    case 2:
      Console.WriteLine("two");// executes
      break;
    default:
      Console.WriteLine("default");
      break;
}
```

```
Fall through in switch case
```

```
int a = 2;
switch (a)
{
    case 1:
      Console.WriteLine("one");
      break;
    case 2:
      Console.WriteLine("two");// executes
      Goto default:
    default:
      Console.WriteLine("default");//executes
      break;
}
```

LOOPING

Looping refers to the repeated execution of statements. This technique comes in very handy because it means that you can repeat operations as many times as you want (thousands, even millions, of times) without having to write the same code each time.

do Loops

The structure of a do loop is as follows, where <Test> evaluates to a Boolean value:

```
do
{
<code to be looped>
} while (<Test>);
```

Do while loop is also called as exit condition checking loop, means it will check the condition while exiting loop.

Eg:

```
int i = 0;//initialization
do
{
System.Console.Write(" " + i.ToString());
i++; //increment
} while (i != 10); //condition
```

while Loops

The Boolean test in a while loop takes place at the start of the loop cycle, not at the end "entry level checking"

```
while (<Test>)
{
<code to be looped>
}
```

Eg:

```
int i = 1;  //initialization
while (i <= 10) // condition
{
Console.WriteLine(i);
i++; // increment
}
```

for Loops

This type of loop executes a set number of times and maintains its own counter.

```
for (<initialization>; <condition>; <operation>)
{
<code to loop>
}
```

Eg:

```
for (int i = 0; i <= 10; i++)
{
Console.Write(" " + i.ToString());
}
```

Interrupting Loops

break—Causes the loop to end immediately

continue—Causes the current loop cycle to end immediately (execution continues with the next loop cycle)
goto—Allows jumping out of a loop to a labeled position
return— Jumps out of the loop and its containing function

About Variables

- How to perform implicit and explicit conversions between types
- How to create and use enum types
- How to create and use struct types
- How to create and use arrays
- How to manipulate string values

Enumerations: Variable types that have a user-defined discrete set of possible values that can be used in a human-readable way. (user defined integer datatype)

Structs: Composite variable types made up of a user-defined set of other variable types. (userdefined datatype)

Arrays: Types that hold multiple variables of one type, allowing index access to the individual value.

Declaring and initialization of variables

```
string s1, s2, s3; // declaration
int i1, i2, i3; // declaration

s1 = "hello"; // initialization
s2 = "world"; // initialization

i1 = 2; // initialization
i3 = 3; // initialization
```

Implicit conversion: Conversion from type A to type B is possible in all circumstances, and the rules for performing the conversion are simple enough for you to trust in the compiler.

Explicit conversion: Conversion from type A to type B is possible only in certain circumstances or where the rules for conversion are complicated enough to merit additional processing of some kind.

Implicit conversion chart

TYPE	CAN SAFELY BE CONVERTED TO
byte	short, ushort, int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

Explicit conversion

```
private void Form1_Load(object sender, EventArgs e)
{
    int a = "11";
}
```
Cannot implicitly convert type 'string' to 'int'

Converting string to integer

```
private void Form1_Load(object sender, EventArgs e)
{
   int a = Convert.ToInt16("11");
}
```

Convert class for explicit conversions

COMMAND	RESULT
Convert.ToBoolean(val)	val converted to bool
Convert.ToByte(val)	val converted to byte
Convert.ToChar(val)	val converted to char
Convert.ToDecimal(val)	val converted to decimal
Convert.ToDouble(val)	val converted to double
Convert.ToInt16(val)	val converted to short
Convert.ToInt32(val)	val converted to int
Convert.ToInt64(val)	val converted to long
Convert.ToSByte(val)	val converted to sbyte
Convert.ToSingle(val)	val converted to float
Convert.ToString(val)	val converted to string
Convert.ToUInt16(val)	val converted to ushort
Convert.ToUInt32(val)	val converted to uint
Convert.ToUInt64(val)	val converted to ulong

Overflow checking

Two keywords exist for setting what is called the overflow checking context for an expression: checked and unchecked. You use these in the following way:

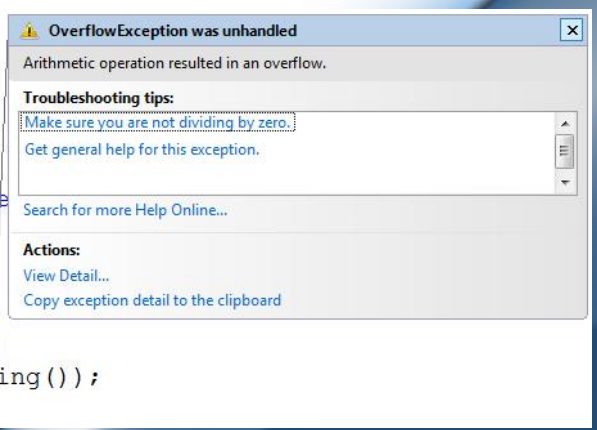
checked(<expression>)

unchecked(<expression>)

```
public Form1()
{
    InitializeComponent();
}

private void Form1_Load(obje
{
    int a = 99999;
    int b = 99999;

    int c = checked(a * b);
    MessageBox.Show(c.ToString());
}
```



if you replace checked with unchecked in this code, you get the result shown earlier, and no error occurs. That is identical to the default behavior

```
private void Form1_Load(object sender, EventArgs e)
{
    int a = 99999;
    int b = 99999;

    int c = unchecked(a * b);
    MessageBox.Show(c.ToString());
}
```

Enumerations

```
enum <typeName> : <underlyingType>
{
<value1> = <actualVal1>,
<value2>,
<value3> = <value1>,
<value4>,
...
<valueN> = <actualValN>
}
```

```
enum internationalcode
{
    india = 91,
    uk = 44
}

private void button1_Click(object sender, EventArgs e)
{
    int code = (int)internationalcode.india; // casting enum to integer
    MessageBox.Show(code.ToString());
}
```

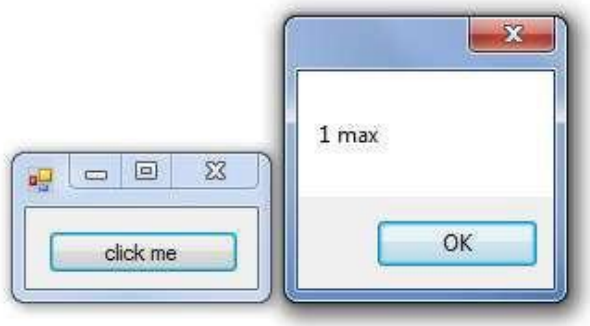
Structs

```
struct employee
      {
        public int empno;
        public string empname;
        public string empsname;
      }
private void button1_Click(object sender, EventArgs e)
      {
         string s; // this is how we declare
                   // variable of predefined datatype

         employee e; // this is how we declare
                     // variable of userdefined datatype

         e.empno = 1;  // initalization
         e.empname = "max"; // initalization
         e.empsname = "payne"; // initalization

         MessageBox.Show(e.empno + " " + e.empname); // retrival
      }
```



Output:

Arrays

All the types you"ve seen so far have one thing in common: Each of them stores a single value (or a single set of values in the case of structs). Sometimes, in situations where you want to store a lot of data, this isn"t very convenient. You may want to store several values of the same type at the same time, without having to use a different variable for each value.

The alternative is to use an *array*. Arrays are indexed lists of variables stored in a single array type variable.

Declaring Arrays

```
<baseType>[ ] <name>;
```

This type declaration is also called as dynamic size array.

```
int[ ] myArray; //
```

use the new keyword to explicitly initialize the array, and a constant value to define the size.

```
int[ ] myIntArray = new int[5]; // the depth of an array is to strore 5 values
```

Declaring and initializing array

```
int[ ] myArray = { 1, 2, 3, 4, 5 };

int[ ] myArray = new int[5] { 1, 2, 3, 4, 5 };

myArray[0] = 1;
myArray[1] = 2;
myArray[2] = 3;
myArray[3] = 4;
myArray[4] = 5;
```

eg:

```
int[] myarray; // declaration

public void test()
{

int[] myarray = new int[2];
```

/* You can either specify the complete contents of the array in a literal form or specify the size of the array and use the new keyword to initialize all array elements. */

```
myarray[0] = 1; // initialization
myarray[1] = 2;  // initialization

for (int i = 0; i < myarray.Length; i++)
{
MessageBox.Show(myarray[i].ToString()); // retrival
}
}
```

Op: 1 2 in messagebox

Working with foreach Loops

Syntax:

foreach (<*baseType*> <*name*> in <*array*>)
{
// can use <*name*> for each element
}

Eg:

```
for (int i = 0; i < myarray.Length; i++)
{
MessageBox.Show(myarray[i].ToString()); // retrival
}
```

Same thing can be achived by foreach loop

```
foreach (int x in myarray)
{
        MessageBox.Show(x.ToString());
}
```

Multidimensional Arrays

A multidimensional array is simply one that uses multiple indices to access its elements. You might specify a position using two coordinates, x and y. You want to use these two coordinates as indices.

A two-dimensional array such as this is declared as follows:

<baseType>[,] <name>;

Arrays of more dimensions simply require more commas:

<baseType>[, , ,] <name>;

Eg:

```
int[,] myarray =  new int[2,2]; // declaration

  public void test()
  {
      myarray[0, 0] = 1; // initalization
      myarray[0, 1] = 2;
      myarray[1, 0] = 3;
      myarray[1, 1] = 4; // initalization

      foreach (int x in myarray)
      {
          MessageBox.Show(x.ToString());
      }
```

```
        }
Op: 1 2 3 4 in message box
```

Arrays of Arrays

Multidimensional arrays, as discussed in the last section, are said to be *rectangular* because each „„row‟‟ is the same size. Using the last example, you can have a y coordinate of 0 to 3 for any of the possible x coordinates.

It is also possible to have ***jagged*** arrays, whereby „„rows‟‟ may be different sizes.

Eg:

```
public void test()
      {

        int[][] myarr = new int[2][]; // declaring arrays
        myarr[0] = new int[2]; // initialize the array that contains other arrays
        myarr[1] = new int[3]; // initialize the array that contains other arrays

         /*     0  1  2
          * 0   3  4
          * 1   5  6  6
          *
          */
        myarr[0][0] = 3; // initialize value
        myarr[0][1] = 4; // initialize value
        myarr[1][0] = 5; // initialize value
        myarr[1][1] = 6; // initialize value
        myarr[1][2] = 7; // initialize value

         foreach (int[] x in myarr)
         {
            foreach (int a in x)
            {
               MessageBox.Show(a.ToString());
            }
         }
      }
```

Another way of writing jagged array:

```
int[ ][ ] jaggedIntArray = { new int[ ] { 1, 2, 3 }, new int[ ] { 1 }, new int[ ] { 1, 2 }
};
```

Collection

SYSTEM.COLLECTIONS

An ArrayList is a class that holds values like an array, but elements can be added or removed at will (dynamic array).

They offer greater functionality over an array, but they also have a larger overhead.

ArrayLists are not type safe, meaning that each element of the ArrayList can be of a different type.

Eg:

```
using System.Collections;
ArrayList myArrayList = new ArrayList(); //variable length
```

```
ArrayList myArrayList = new ArrayList(5) //fixed length
myArrayList.Add("Hello");
myArrayList.Add("World");
myArrayList.Add(10);
```

Arraylist properties and methods

properties	description
Capacity	Contains the allocated length of the arraylist
Count	Contains the number of items currently in the arraylist

methods	description
Add()	Adds an item to the arraylist and returns the newly added index a.add(“abc”) ;
Clear()	Clears all items from the arraylist a.Clear();
Contains()	Returns True if the given object is in the current arraylist a.Contains(“a”);
CopyTo()	Copies all or part of the current arraylist to another arraylist that is passed in as an argument. Eg a.copyto(onedimentionarray);
Insert()	Inserts an item into the arraylist a.Insert(index,object);
Remove()	Removes an item from the arraylist a.Remove(“a”);
RemoveAt()	Removes an item from the arraylist by index a.RemoveAt(index);
RemoveRange()	Removes a range of items from the arraylist a.RemoveRange(index,range) ;
Sort()	Sorts the items in the arraylist a.sort();

STRING MANIPULATION

Your use of strings so far has consisted of writing strings to the console, reading strings from the console, and concatenating strings using the + operator.

Eg:

```
string str1, str2;
            string[] strArray = new string[3];
            strArray[0] = "abc";
            strArray[1] = "def";
            strArray[2] = "jkl";
            str1 = "we-it tutorials";
            str2 = "hello world";
//public int IndexOf(char value);
MessageBox.Show(str1.IndexOf('-').ToString()); // op: 2

//public int IndexOf(string value);
MessageBox.Show(str1.IndexOf("ria").ToString()); // op: 10

//public int IndexOf(char value, int startIndex);
MessageBox.Show(str1.IndexOf('i',6).ToString()); // op: 11
```

```
//public int IndexOf(string value, int startIndex);
MessageBox.Show(str1.IndexOf("ria",4).ToString()); // op: 10

//public string Insert(int startIndex, string value);
MessageBox.Show(str1.Insert(15, " thane")); // op: we-it tutorials thane

//public static string Join(string separator, string[] value);
MessageBox.Show(string.Join("*", strArray)); // op: abc*def*jkl

//public int LastIndexOf(string value);
MessageBox.Show(str1.LastIndexOf('t').ToString());// op: 8

//public string Remove(int startIndex);
MessageBox.Show(str2.Remove(4)); // op : hell

//public string Replace(string oldValue, string newValue);
MessageBox.Show(str2.Replace("world", "india")); // op: hello india

//public string[] Split(params char[] separator);
string[] xyz = str2.Split(' ');
foreach (string temp in xyz) { MessageBox.Show(temp); } // op: hello and world

//public string Substring(int startIndex);
MessageBox.Show(str2.Substring(6)); // op: world

//public string Substring(int startIndex, int length);
MessageBox.Show(str2.Substring(6,3)); // op: wor

//public string ToLower();
MessageBox.Show(str1.ToLower()); // op: we-it tutorials

//public string ToUpper();
MessageBox.Show(str1.ToUpper()); // op: WE-IT TUTORIALS


//public string Trim();
MessageBox.Show(str1.Trim()); // we-it tutorials with out whitespace on both
side
```

Functions

DEFINING AND USING FUNCTIONS

```
static void Main(string[] args)
{
}
```

This is how we write Main method in c# also called entry method from which the program starts execution

Syntax:

```
<Access specifier> <return type> <identifier> (parameters)
{
        Return <datatype>;
}
```

Eg:

```
Public static int add()
{
```

```
Return 2+2;
}
```

Public : access specifier
Static : no class object needed to access function
Int : return type
Add : name of the method

Parameters

Syntax:

```
static <returnType> <FunctionName>(<paramType> <paramName>, ...)
{
...
return <returnValue>;
}
```

Eg:

```
Public int add(int a, int b)
{
Return a + b;
}
```

Calling the function

```
Int sum = add(10,20);
```

Passing Array as parameter

```
Public int add(int[] x)
{
      Return x.length;
}
```

Int[] x : is a array parameter

Parameter Arrays

A parameter declared with a params modifier is a parameter array.

Eg:

```
class Test
{
   static void F(params int[] args) {
      Console.Write("Array contains {0} elements:", args.Length);
      foreach (int i in args)
         Console.Write(" {0}", i);
      Console.WriteLine();
   }
   static void Main() {
      int[] arr = {1, 2, 3};
      F(arr);
      F(10, 20, 30, 40);
      F();
   }
}
```

Op:
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:

Parameters by ref and by val

Ref : shares same memory location of variable
Val : copies value from one variable to another (default)

Eg:

```
class Program
   {
      static void Main(string[] args)
      {
         int x = 10;
         Program.swap(ref x);
         Console.Write(x);  // op : 20
         Console.ReadLine();
      }

      public static void swap(ref int a)
      {
         a = 20;
      }
   }
```

Out Parameter

Out : used to take output back from parameter even we use void method.

Eg:

```
class Program
   {
      static void Main(string[] args)
      {
         int x = 10;
         int ret;
         Program.test(x,out ret);
         Console.Write(ret);  // op : 20
         Console.ReadLine();
      }

      public static void test(int a, out int b)
      {
         b = 20;
      }
   }
```

Command line arguments

Eg:

```
class Program
   {
      static void Main(string[] args)
      {
         foreach (string x in args)
         {
```

```
            Console.WriteLine(x);
        }
        Console.ReadLine();
    }
}
```

Op:

```
program.exe hello world
Hello
World
```

Overloading function

It allows the programmer to make one or several parameters optional, by giving them a default value.

```
class Program
    {
        static void Main(string[] args)
        {
            Program pp = new Program();
            Console.WriteLine(pp.add(10, 20).ToString);
            Console.WriteLine(pp.add(10, 20, 30).ToString);
            Console.ReadLine();
        }

        public int add(int a, int b)
        {
            return a + b;
        }
        public int add(int a, int b, int c)
        {
            return a + b + c;
        }

    }
```

Debugging and Error Handling

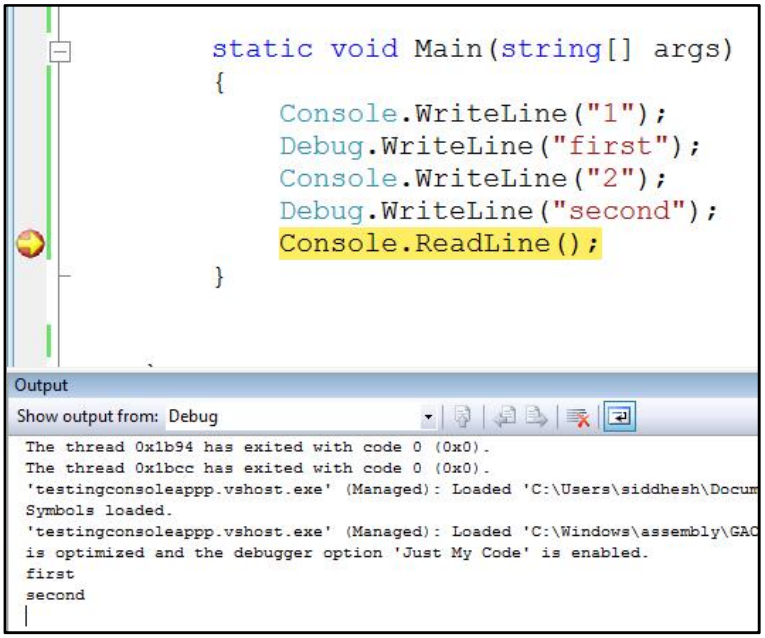
Outputting Debugging Information

Writing text to the Output window at runtime is easy.

There are two commands you can use to do this:

Debug.WriteLine()
Trace.WriteLine()

To use the commands we use System.Diagnostics namespace



Tracepoints

An alternative to writing information to the Output window is to use *tracepoints*.

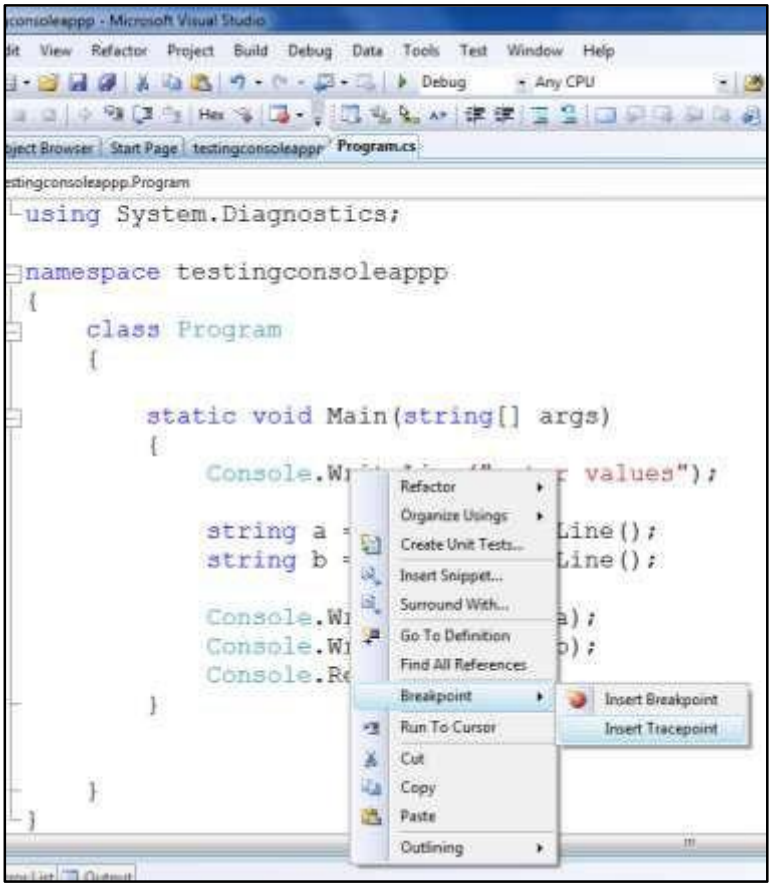
1.Position the cursor at the line where you want the tracepoint to be inserted. The tracepoint will be processed *before* this line of code is executed.

2. Right-click the line of code and select Breakpoint ➪ Insert Tracepoint.

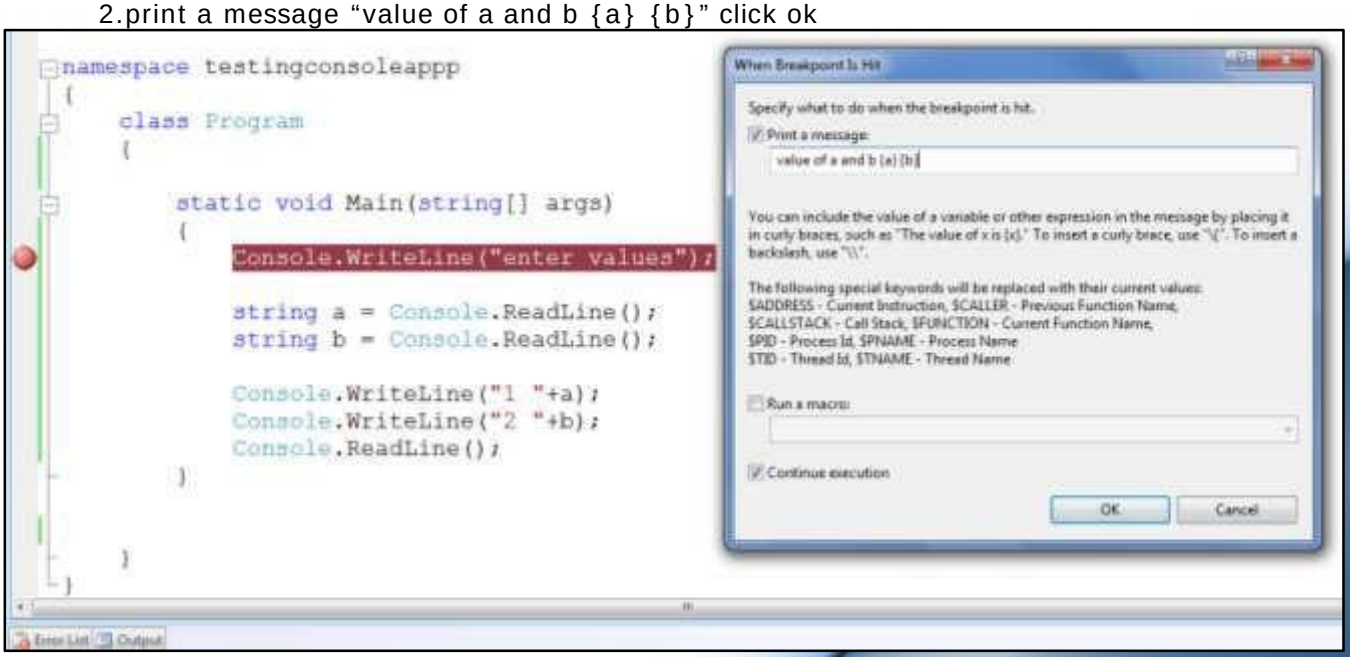
3.Type the string to be output in the Print a Message text box in the When Breakpoint Is Hit dialog that appears. If you want to output variable values, enclose the variable name in curly braces.

4.Click OK. A red diamond appears to the left of the line of code containing a tracepoint, and the line of code itself is shown with red highlighting.

1.Right click on code→breakpoint→insert tracepoint



2.print a message “value of a and b {a} {b}” click ok



3.while debugging the output of trace is shown output window

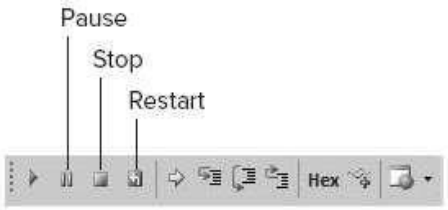
```
namespace testingconsoleappp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("enter values");

            string a = Console.ReadLine();
            string b = Console.ReadLine();
```

```
Output
Show output from: Debug
The thread 0x1a84 has exited with code 0 (0x0).
The thread 0x1b5c has exited with code 0 (0x0).
The thread 0x191c has exited with code 0 (0x0).
The thread 0x14bc has exited with code 0 (0x0).
The thread 0x1a54 has exited with code 0 (0x0).
'testingconsoleappp.vshost.exe' (Managed): Loaded 'C:\Users\siddhesh\Documents\Visual
Symbols loaded.
value of a and b null null
```

Entering Break Mode

The simplest way to enter break mode is to click the Pause button in the IDE while an application is running.



- Pause the application and enter break mode.
- Stop the application completely (this doesn"t enter break mode, it just quits).
- Restart the application

Breakpoints

A *breakpoint* is a marker in your source code that triggers automatic entry into break mode.

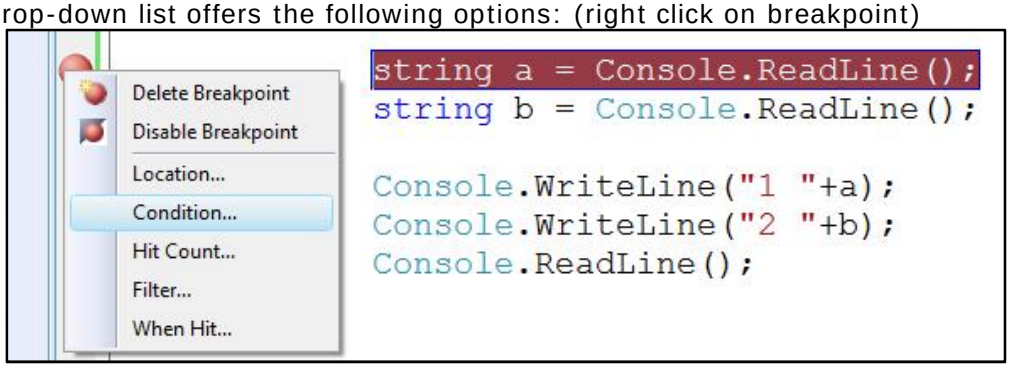
These features are available only in debug builds. If you compile a release build, all breakpoints are ignored. There are several ways to add breakpoints. To add simple breakpoints that break when a line is reached, just left-click on the far left of the line of code, right-click on the line, and select Breakpoint ➪ Insert Breakpoint; select Debug ➪ Toggle Breakpoint from the menu; or press F9.

A breakpoint appears as a red circle next to the line of code

```
static void Main(string[] args)
{
    Console.WriteLine("enter values");

    string a = Console.ReadLine();
At Program.cs, line 16 character 13 ('testingconsoleappp.Program.Main(string[] args)', line 5)
    Console.WriteLine("1 "+a);
    Console.WriteLine("2 "+b);
    Console.ReadLine();
}
```

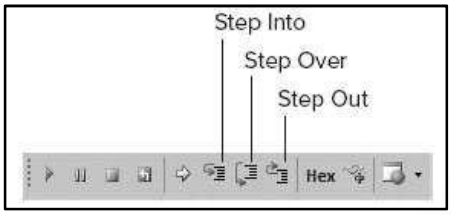
A drop-down list offers the following options: (right click on breakpoint)



- Break always
- Break when the hit count is equal to
- Break when the hit count is a multiple of
- Break when the hit count is greater than or equal to

Stepping Through Code

The yellow arrow on breakpoint shows you what point execution has reached when break mode is entered. At this point, you can have execution proceed on a line-by-line basis.



- **Step Into:** Execute and move to the next statement to execute.
- **Step Over:** Similar to Step Into, but won‟t enter nested blocks of code, including functions.
- **Step Out:** Run to the end of the code block and resume break mode at the statement that follows.

ERROR HANDLING

Error handling is the term for all techniques of this nature, and this section looks at exceptions and how you can deal with them. An exception is an error generated either in your code or in a function called by your code that occurs at runtime.

try . . . catch . . . finally

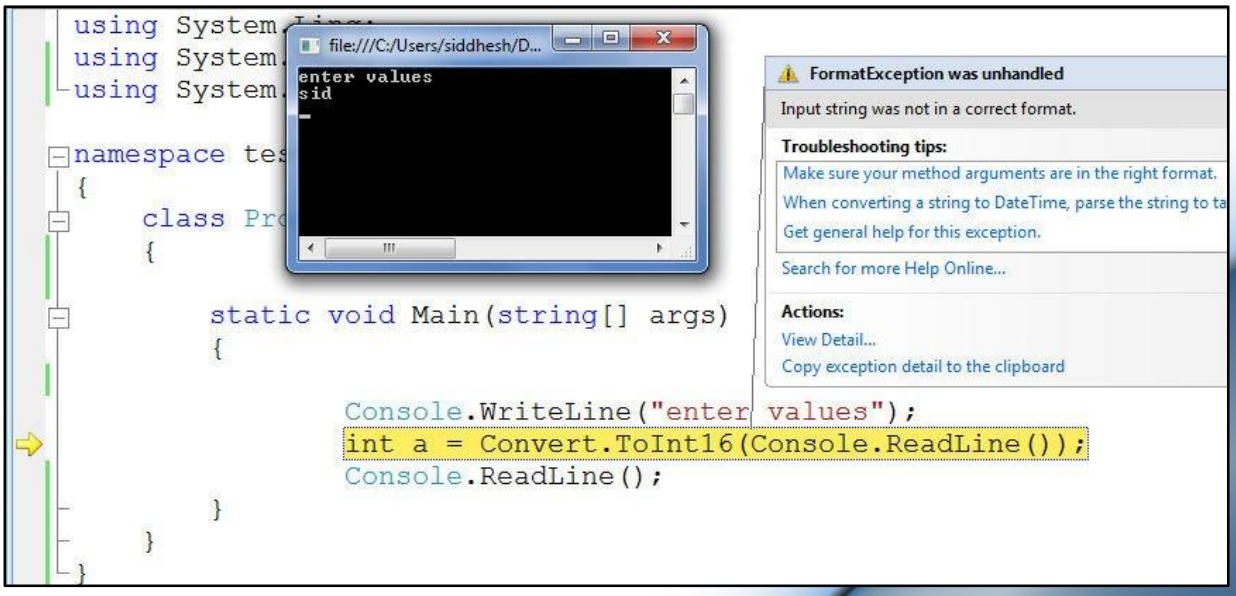
The C# language includes syntax for ***structured exception handling*** (SEH). Three keywords mark code as being able to handle exceptions, along with instructions specifying what to do when an exception occurs: ***try***, ***catch***, and ***finally***. Each of these has an associated code block and must be used in consecutive lines of code. The basic structure is as follows:

Syntax:

```
try
{
...
}
catch (<exceptionType> e)
{
...
}
finally
{
...
}
```

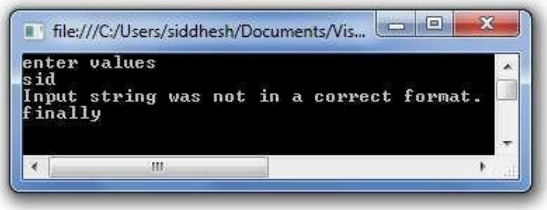
It is also possible, however, to have a try block and a finally block with no catch block, or a try block with multiple catch blocks. If one or more catch blocks exist, then the finally block is optional.

Without try catch program terminated unexceptionally



With try catch program dint stopped working

```
static void Main(string[] args)
{
    try
    {
        Console.WriteLine("enter values");
        int a = Convert.ToInt16(Console.ReadLine());
        Console.ReadLine();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        Console.WriteLine("finally");
        Console.ReadLine();
    }
}
```



Nested try

Try block inside try block is called nested try

The usage of try catch blocks

try— Contains code that might throw exceptions („„throw‟‟ is the C# way of saying „„generate‟‟ or „„cause‟‟ when talking about exceptions)

catch—Contains code to execute when exceptions are thrown. catch blocks may be set to respond only to specific exception types (such as System.IndexOutOfRangeException) using *<exceptionType>*, hence the ability to provide multiple catch blocks. It is also possible to omit this parameter entirely, to get a general catch block that responds to all exceptions.

finally— Contains code that is always executed, either after the try block if no exception occurs, after a catch block if an exception is handled, or just before an unhandled exception moves „„up the call stack.‟‟ This phrase means that SEH allows you to nest try...catch...finally blocks inside each other, either directly or because of a call to a function within a try block.

Object-Oriented Programming

WHAT IS OBJECT-ORIENTED PROGRAMMING?

Object-oriented programming is a relatively new approach to creating computer applications that seeks to address many of the problems with traditional programming techniques. The type of programming you have seen so far is known as *functional* (or *procedural*) programming, often resulting

in so-called monolithic applications, meaning all functionality is contained in a few modules of code (often just one). With OOP techniques, you often use many more modules of code, each offering specific functionality, and each module may be isolated or even completely independent of the others. This

modular method of programming gives you much more versatility and provides more opportunity for code reuse.

What Is an Object?

An object is a building block of an OOP application. This building block encapsulates part of the application, which may be a process, a chunk of data, or a more abstract entity.

Eg:

```
class Program
{
#region fields
   public string empname;
   private string empid;
#endregion

#region properties
//properties are used to access private fields
   public string accessempid
   {get{return empid;} set { empid = value; }}
#endregion

#region constructor
//constructor is used to set values to private fields
//while initialization the object of the classs
   public Program() // default constructor
   {

   }

   public Program(string a) // parametarized constructor
   {
      empid = a;
   }

#endregion

#region destructor
   ~Program()
   {
      // destructor code
   }
#endregion

#region methods and functions

   static void Main(string[] args)
   {
      Program p = new Program(); // creating object of program class
      Program p1 = new Program("testing"); // passing value to constructor
      p.accessempid = "12"; // accessing property
   }

#endregion
}
```

The Life Cycle of an Object

Construction: When an object is first instantiated it needs to be initialized. This initialization is known as *construction* and is carried out by a constructor function, often referred to simply as a *constructor* for convenience.

```
Program p = new Program(); // creating object of program class
```

Destruction: When an object is destroyed, there are often some clean-up tasks to perform, such as freeing memory. This is the job of a destructor function, also known as a *destructor*.

```
Program p1 = new Program("testing"); // passing value to constructor
```

Static and Instance Class Members

Static properties and fields enable you to access data that is independent of any object instances, and static methods enable you to execute commands related to the class type but not specific to object instances. When using static members, in fact, you d**on"t** even need to instantiate an object.

Static Constructors

A class can have a single static constructor, which must have no access modifiers and cannot have any parameters. A static constructor can never be called directly; instead, it is executed when one of the following occurs:

→An instance of the class containing the static constructor is created.
→A static member of the class containing the static constructor is accessed.

In both cases, the static constructor is called first, before the class is instantiated or static members accessed.

Static Classes

Often, you will want to use classes that contain only static members and cannot be used to instantiate objects (such as Console). A shorthand way to do this, rather than make the constructors of the class private, is to use a *static class*. A static class can contain only static members and can"t have instance constructors, since by implication it can never be instantiated.

OOP TECHNIQUES

Interfaces

An interface is a collection of public instance (that is, nonstatic) methods and properties that are grouped together to encapsulate specific functionality. After an interface has been defined, you can implement it in a class. This means that the class will then support all of the properties and members specified by the interface.

```
public interface int1 //interface
{
void display();// abstract method
}

public interface int2 //interface
{
void display();// abstract method
}


public class testing : int1, int2
// interface supports multiple inheritance
{
void int1.display()
{
    Console.WriteLine("interface1 method");
}
```

```
void int2.display()
{
    Console.WriteLine("interface2 method");
}
}
```

Inheritance

Inheritance is one of the most important features of OOP. Any class may *inherit* from another, which means that it will have all the members of the class from which it inherits. In OOP terminology, the class being inherited from (*derived* from) is the *parent* class (also known as the *base* class).

When using inheritance from a base class, the question of member accessibility becomes an important one. Private members of the base class are not accessible from a derived class, but public members are. However, public members are accessible to both the derived class and external code.

To get around this, there is a third type of accessibility, *protected*, in which only derived classes have access to a member.

```
public class parent
    {
        public void display()
        {
            Console.WriteLine("display method");
        }
    }
    public class child : parent
    {
        // no methods or functions
    }

class Program
{
    static void Main(string[] args)
{
child c = new child(); // creating object of child class
c.display();// even though  child class doesnt have methods display() method is
accessible because if inheritance
}
}
```

virtual, override , new

```
public class parent
{
   public virtual void display() // virtual method
   {
      Console.WriteLine("display method");
   }
}
public class child : parent
{
   public override void display() // overridden method
   {
      Console.WriteLine("new display");
   }
```

```
}

class Program
{
   static void Main(string[] args)
   {
      child c = new child(); // creating object of child class
      c.display();    // it will call display() of child class
   }
}
```

Virtual method in base class can be overridden in derived class using override keyword. when virtual keyword is not written in base class method we can not use override. Override can be written only when virtual keyword is written.

This can be done by a new keyword

```
public class parent
{
   public void display()
   {
      Console.WriteLine("display method");
   }
}
public class child : parent
{
   public new void display() // suppress base method
   {
      Console.WriteLine("new display");
   }
}

class Program
{
   static void Main(string[] args)
   {
      child c = new child(); // creating object of child class
      c.display();    // it will call display() of child class
   }
}
```

Sealed class and sealed method

```
public sealed class parent
```

sealed class cannot be further inherited

```
public sealed void display()
```

sealed method cannot be overridden

Abstract class and abstract method

```
public abstract  class parent // abstract class
{
   public abstract void display();// abstract method
}
```

Abstract methods are like interface which has only declaration no body or execution procedures

Abstract classes has abstract methods, it can not be used directly that"s why it is inhearited and method are overrided.

Polymorphism

C# gives us polymorphism through inheritance. Inheritance-based polymorphism allows us to define methods in a base class and override them with derived class implementations. Thus if you have a base class object that might be holding one of several derived class objects, polymorphism when properly used allows you to call a method that will work differently according to the type of derived class the object belongs to.

Operator Overloading

There are times when it is logical to use operators with objects instantiated from your own classes. This is possible because classes can contain instructions regarding how operators should be treated.

Use of same operator but performs different on different class

Eg:

1 + 2
Output is 3

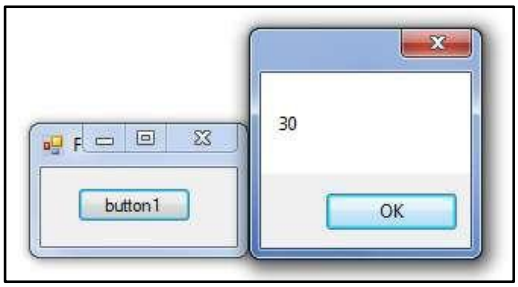
"a" + "b"
Output is "ab"

Even though operator is same it works different on string class and integer class

Code:

```
private void button1_Click(object sender, EventArgs e)
{
overload o1 = new overload();
overload o2 = new overload();
overload o3 = new overload();
o1.x = 5;
o2.x = 6;
o3 = o1 + o2; // calling operator overloading method
MessageBox.Show(o3.x.ToString());
}
}
public class overload
{
public int x;
public static overload operator +(overload a, overload b)
{
a.x = a.x * b.x;
return a;
}
}
```

Output:



UNIT 2

Assemblies

Every software has executable files (.exe). apart from the executable file, there are some Dynamic Link Libraries (DLL) & Library (LIB) files, which conyain the complicated code of some commonly used functions. These files come along with software. Any software package includes the executable file along with some DLLs & LIB files, which are necessary to run the application. In terms of .NET runtime, the process of packaging is called assembling. An assembly contains MSIL, metadata, & other files required to execute .NET program successfully.

In .NET Framework, assemblies play an important role. An assembly is an fundamental unit of deployment. Deployment is the process wherein an application installed on a machine. Assemblies can be created with the help of some development tools like Visual Studio or with the help of tools provided in .NET framework SDK. Assemblies can be make in form of .dll or .exe files using Visual Studio. When source code is compiled, the EXE/DLL, generated by default, is actually an assembly.

Assemblies Overview

Assemblies are a fundamental part of programming with the .NET Framework. An assembly performs the following functions:

- It contains code that the common language runtime executes. Microsoft intermediate language (MSIL) code in a portable executable (PE) file will not be executed if it does not have an associated assembly manifest. Note that each assembly can have only one entry point (that is, DllMain, WinMain, or Main).
- It forms a security boundary. An assembly is the unit at which permissions are requested and granted
- It forms a type boundary. Every type's identity includes the name of the assembly in which it resides. A type called MyType loaded in the scope of one assembly is not the same as a type called MyType loaded in the scope of another assembly.
- It forms a reference scope boundary. The assembly's manifest contains assembly metadata that is used for resolving types and satisfying resource requests. It specifies the types and resources that are exposed outside the assembly. The manifest also enumerates other assemblies on which it depends.
- It forms a version boundary. The assembly is the smallest versionable unit in the common language runtime; all types and resources in the same assembly are versioned as a unit. The assembly's manifest describes the version dependencies you specify for any dependent assemblies.
- It forms a deployment unit. When an application starts, only the assemblies that the application initially calls must be present. Other assemblies, such as localization resources or assemblies containing utility classes, can be retrieved on demand. This allows applications to be kept simple and thin when first downloaded

- It is the unit at which side-by-side execution is supported.

Assemblies can be static or dynamic. Static assemblies can include .NET Framework types (interfaces and classes), as well as resources for the assembly (bitmaps, JPEG files, resource files, and so on). Static assemblies are stored on disk in portable executable (PE) files. You can also use the .NET Framework to create dynamic assemblies, which are run directly from memory and are not saved to disk before execution. You can save dynamic assemblies to disk after they have executed.

There are several ways to create assemblies. You can use development tools, such as Visual Studio 2005, that you have used in the past to create .dll or .exe files. You can use tools provided in the Windows Software Development Kit (SDK) to create assemblies with modules created in other development environments. You can also use common language runtime APIs, such as Reflection.Emit, to create dynamic assemblies.

Assembly Benefits

- Assemblies are designed to simplify application deployment and to solve versioning problems that can occur with component-based applications.
- End users and developers are familiar with versioning and deployment issues that arise from today's component-based systems. Some end users have experienced the frustration of installing a new application on their computer, only to find that an existing application has suddenly stopped working. Many developers have spent countless hours trying to keep all necessary registry entries consistent in order to activate a COM class.
- Many deployment problems have been solved by the use of assemblies in the .NET Framework. Because they are self-describing components that have no dependencies on registry entries, assemblies enable zero-impact application installation. They also simplify uninstalling and replicating applications.

Versioning Problems

Currently two versioning problems occur with Win32 applications:

- Versioning rules cannot be expressed between pieces of an application and enforced by the operating system. The current approach relies on backward compatibility, which is often difficult to guarantee. Interface definitions must be static, once published, and a single piece of code must maintain backward compatibility with previous versions. Furthermore, code is typically designed so that only a single version of it can be present and executing on a computer at any given time.
- There is no way to maintain consistency between sets of components that are built together and the set that is present at run time.

These two versioning problems combine to create DLL conflicts, where installing one application can inadvertently break an existing application because a certain software component or DLL was installed that was not fully backward compatible with a previous version. Once this situation occurs, there is no support in the system for diagnosing and fixing the problem.

An End to DLL Conflicts

Microsoft® Windows® 2000 began to fully address these problems. It provides two features that partially fix DLL conflicts:

- Windows 2000 enables you to create client applications where the dependent .dll files are located in the same directory as the application's .exe file. Windows 2000 can be configured to check for a component in the directory where the .exe file is located before checking the fully qualified path or searching the normal path. This enables components to be independent of components installed and used by other applications.
- Windows 2000 locks files that are shipped with the operating system in the System32 directory so they cannot be inadvertently replaced when applications are installed.

The common language runtime uses assemblies to continue this evolution toward a complete solution to DLL conflicts.

The Assembly Solution

To solve versioning problems, as well as the remaining problems that lead to DLL conflicts, the runtime uses assemblies to do the following:

- Enable developers to specify version rules between different software components.
- Provide the infrastructure to enforce versioning rules.
- Provide the infrastructure to allow multiple versions of a component to be run simultaneously (called side-by-side execution).

Assembly Contents(Components Of Assembly)

In general, a static assembly can consist of four elements:

- The assembly manifest, which contains assembly metadata.
- Type metadata.
- Microsoft intermediate language (MSIL) code that implements the types.
- A set of resources.

Only the assembly manifest is required, but either types or resources are needed to give the assembly any meaningful functionality.

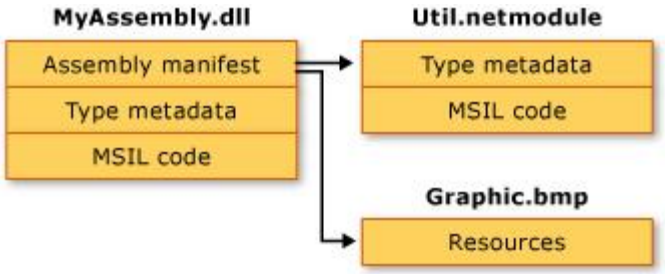
There are several ways to group these elements in an assembly. You can group all elements in a single physical file, which is shown in the following illustration.

Single-file assembly

Alternatively, the elements of an assembly can be contained in several files. These files can be modules of compiled code (.netmodule), resources (such as .bmp or .jpg files), or other files required by the application. Create a multifile assembly when you want to combine modules written in different languages and to optimize downloading an application by putting seldom used types in a module that is downloaded only when needed.

In the following illustration, the developer of a hypothetical application has chosen to separate some utility code into a different module and to keep a large resource file (in this case a .bmp image) in its original file. The .NET Framework downloads a file only when it is referenced; keeping infrequently referenced code in a separate file from the application optimizes code download.

Multifile assembly



The files that make up a multifile assembly are not physically linked by the file system. Rather, they are linked through the assembly manifest and the common language runtime manages them as a unit.

In this illustration, all three files belong to an assembly, as described in the assembly manifest contained in MyAssembly.dll. To the file system, they are three separate files. Note that the file Util.netmodule was compiled as a module because it contains no assembly information. When the assembly was created, the assembly manifest was added to MyAssembly.dll, indicating its relationship with Util.netmodule and Graphic.bmp.

As you currently design your source code, you make explicit decisions about how to partition the functionality of your application into one or more files. When designing .NET Framework code, you will make similar decisions about how to partition the functionality into one or more assemblies.

Private Assembly

- When you deploy an assembly which can be use by single application, than this assembly is called a private assembly.
- Private assemblies can be used by only one application they are deployed with.
- Private assemblies are deployed in the directory where the main application is installed.

Shared Assembly

- When you deploy an assembly which can be used by several application, than this assembly is called shared assembly.
- Shared assemblies are stored in a special folder called Global Assembly Cache (GAC), which is accessible by all applications.
- Shared assemblies must have a strong name. A strong name consists of an assembly name, a version number, a culture, a public key and an optional digital signature.
- GAC is capable of maintaining multiple copies of an assembly with the same name but different versions.

Q:Metadata Overview

Metadata is binary information describing program that is stored either in a common language runtime portable executable (PE) file or in memory. When code is complied into a PE file, metadata is inserted into one portion of the file, while code is converted to Microsoft intermediate language (MSIL) and inserted into another portion of the file. Every type and member defined and referenced in a module or assembly is described within metadata. When code is executed, the runtime loads metadata into memory and references it to discover information about code's classes, members, inheritance, and so on.

Metadata describes every type and member defined in code in a language-neutral manner. Metadata stores the following information:

- Description of the assembly.
  - Identity (name, version, culture, public key).
  - The types that are exported.
  - Other assemblies that this assembly depends on.
  - Security permissions needed to run.
- Description of types.
  - Name, visibility, base class, and interfaces implemented.
  - Members (methods, fields, properties, events, nested types).
- Attributes.
  - Additional descriptive elements that modify types and members.

Benefits of Metadata

Metadata is the key to a simpler programming model, eliminating the need for Interface Definition Language (IDL) files, header files, or any external method of component reference. Metadata allows .NET languages to describe themselves automatically in a language-neutral manner, unseen by both the developer and the user. Additionally, metadata is extensible through the use of attributes. Metadata provides the following major benefits:

- **Self-describing files.**

Common language runtime modules and assemblies are self-describing. A module's metadata contains everything needed to interact with another module. Metadata automatically provides the functionality of IDL in COM, allowing to use one file for both definition and implementation. Runtime modules and assemblies do not even require registration with the operating system. As a result, the descriptions used by the runtime always reflect the actual code in compiled file, which increases application reliability.

- **Language interoperability and easier component-based design.**

  Metadata provides all the information required about compiled code for programmer to inherit a class from a PE file written in a different language. Programmer can create an instance of any class written in any managed language (any language that targets the common language runtime) without worrying about explicit marshaling or using custom interoperability code.

- **Attributes.**

  The .NET Framework allows programmer to declare specific kinds of metadata, called attributes, in compiled file. Attributes can be found throughout the .NET Framework and are used to control in more detail how program behaves at run time. Additionally, programmer can emit his/her own custom metadata into .NET Framework files through user-defined custom attributes.

Microsoft intermediate language(MSIL)

Definition:

- It is a set of CPU independent instructions that are generated by the language compiler when the project is compiled.
- MSIL code is not executable but further processed by CLR/other runtime environments before it becomes executable.
- MSIL is contained in the assembly of the .NET application.

Features:

MSIL instructions map to the code that is written in .NET Language and are used for

1. loading,
2. storing,
3. initializing,
4. and calling methods on objects,
5. as well as for arithmetic and logical operations,
6. control flow,
7. direct memory access,
8. exception handling,
9. and other operations.

CLS(Common language Specification) provides the infrastructure for MSIL.

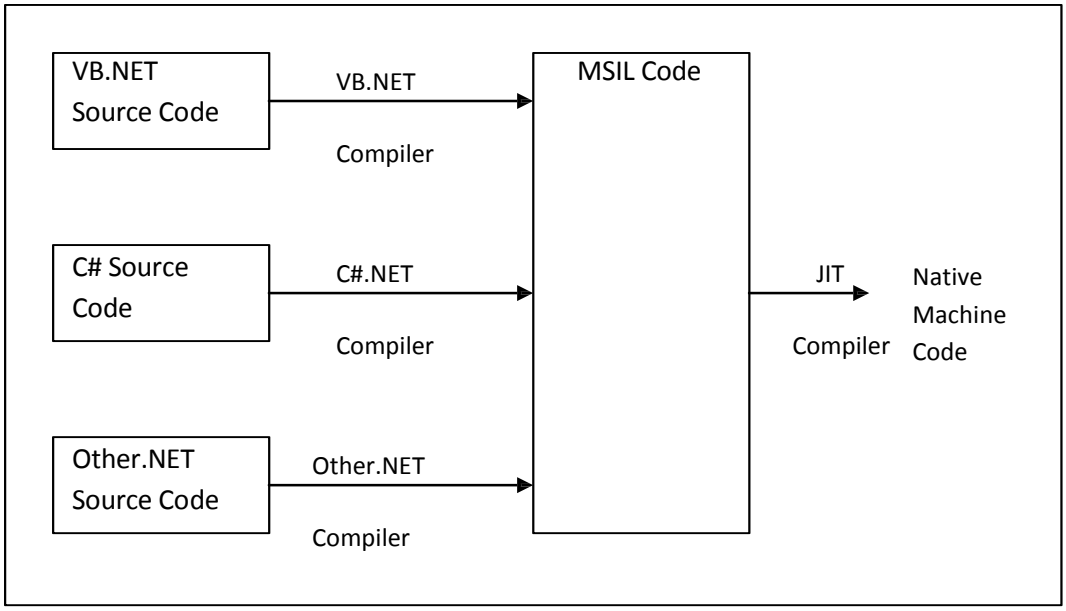
Benefits:

MSIL provides language interoperability as the code in any .NET language is compiled into MSIL.

Same performance for all the .NET Languages:

- Support for different runtime environments
- CLR can understand MSIL.
- Non .NET environments also support MSIL.

The JIT Compiler in CLR converts the MSIL code into native machine code which is then executed by the OS



Garbage collector(GC)

The Microsoft .NET Framework provides an automated mechanism for reclaiming an object that is no longer in use. This process is usually referred as Garbage Collection.

The Microsoft .NET Framework CLR reserves space for each object instantiated in the system.

Since memory is not infinite, CLR needs to get rid of those objects that are no longer in use so that the space can be used for other objects.

1. The very first step in Garbage Collection is identifying those objects that can be wiped out.
2. To accomplish this step, CLR maintains the list of references for an object.
3. If an object has no more references, i.e. there is no way that the object could be referred to by the application, CLR considers that object as garbage.
4. During Garbage Collection, CLR reclaims memory for all garbage objects.

benefits of garbage collector

- Enables you to develop your application without having to free memory.
- Allocates objects on the managed heap efficiently.
- Reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future allocations. Managed objects automatically get clean content to start with, so their constructors do not have to initialize every data field.

- Provides memory safety by making sure that an object cannot use the content of another object.

JIT Compilers

JIT compilers play a major role in the .NET platform because all .NET PE files contain IL and metadata, not native code. The JIT compilers convert IL to native code so that it can execute on the target operating system. For each method that has been successfully verified for type safety, a JIT compiler in the CLR will compile the method and convert it into native code.

One advantage of a JIT compiler is that it can dynamically compile code that is optimized for the target machine. If you take the same .NET PE file from a one-CPU machine to a two-CPU machine, the JIT compiler on the two-CPU machine knows about the second CPU and may be able to spit out the native code that takes advantage of the second CPU. Another obvious advantage is that you can take the same .NET PE file and run it on a totally different platform, whether it be Windows, Unix, or whatever, as long as that platform has a CLR.

For optimization reasons, JIT compilation occurs only the first time a method is invoked. Recall that the class loader adds a stub to each method during class loading. At the first method invocation, the VES reads the information in this stub, which tells it that the code for the method has not been JIT-compiled. At this indication, the JIT compiler compiles the method and injects the address of the native method into this stub. During subsequent invocations to the same method, no JIT compilation is needed because each time the VES goes to read information in the stub, it sees the address of the native method. Because the JIT compiler only performs its magic the first time a method is invoked, the methods you don't need at runtime will never be JIT-compiled.

The compiled, native code lies in memory until the process shuts down and until the garbage collector clears off all references and memory associated with the process. This means that the next time you execute the process or component, the JIT compiler will again perform its magic.

If you want to avoid the cost of JIT compilation at runtime, you can use a special tool called ngen.exe, which compiles your IL during installation and setup time. Using ngen, you can JIT-compile the code once and cache it on the machine so that you can avoid JIT compilation at runtime (this process is referred to as pre-JITting). In the event that the PE file has been updated, you must PreJIT the PE file again. Otherwise, the CLR can detect the update and dynamically command the appropriate JIT compiler to compile the assembly.

Q:: What are the different types of collections in .NET?

ANS: At its simplest, an object holds a single value. At its most complex, it holds references to many other objects. The .NET Framework provides collections—these include List and Dictionary. They are often useful.

List

First, the List type provides an efficient and dynamically-allocated array. It does not provide fast lookup in the general case—the Dictionary is better for this. List is excellent when used in loops.

example::

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

```
class Program
{
   static void Main()
   {
        // Use the List type.
        List<string> list = new List<string>();
        list.Add("cat");
        list.Add("dog");

        foreach (string element in list)
        {
           Console.WriteLine(element);
        }
   }
}
```

output::

```
cat dog
```

Dictionary

The Dictionary type in the base class library is an important one. It is an implementation of a hashtable, which is an extremely efficient way to store keys for lookup. The Dictionary in .NET is well-designed.

We try to reference items in a table directly by doing arithmetic operations to transform keys into table addresses.

code:

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

class Program
{
   static void Main()
   {
        // Use the dictionary.
        Dictionary<string, int> dict = new Dictionary<string, int>();
        dict.Add("cat", 1);
        dict.Add("dog", 4);

        Console.WriteLine(dict["cat"]);
        Console.WriteLine(dict["dog"]);
   }
}
```

output::

```
1
4
```

Thank You