TOKENS

A token is defined as the smallest unit of a program. C# tokens are classified into keywords, identifiers, constants, operators and punctuators.

1. Keywords: - Keywords are some special word for a programming language. Keywords has some specific meaning for the compiler. It is also known as reserved words. Like void, int, char, float, else, if etc.
2. Identifiers:- Identifiers are the names given to the uniquely identified various programming elements like variables, arrays, methods, classes, objects, namespaces, interface and so on.
   The following are rules for naming identifiers:-
   i). An identifier must be unique in a program.
   ii). Alphabets, digits, underscore and dollar sign characters can be used in an identifier. iii). An identifier must not start with a digit.

LITERALS in C#

Literals are the value constants assigned to variables in a program in a C# program The following are different types of Literals in c#.

i). Numeric Literals are two type integer Literals and Real Integer
Integer Literals: e.g 0, 654321(decimal integers) and 0x2, 0x9f for (hexdicimal integers)
Real Literals: e.g 0.0083, -0.75, 435.36

ii). Boolean Literals: There are two Boolean literal values: True False. They are used as values of relational expressions.

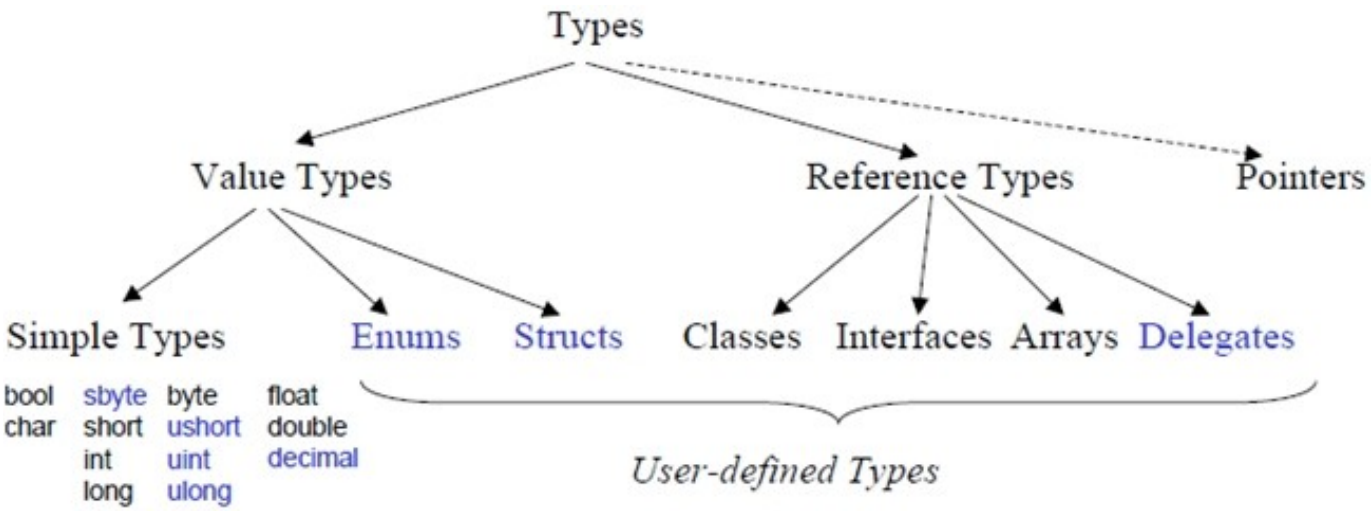
iii). Single Character Literals A single character literal contains a single character enclosed within a pair of single quote marks. Example of character in the above constant are: '5' 'x'

iv). String Literals A string literal in a sequence of character enclosed between double quotes. The characters may be alphabets, digits, special character and blank spaces. Example "Hello C#" "2001" "5+3" etc.

BASIC DATA TYPES

Data type is used to define the type of data and size of data. It tells the program which type of data enter into the program. Data types in C# are classified into two types:

i). Value Types (stack allocated)
ii). Reference Types (heap allocated)



Implicit data types (simple types) are defined in the language core by the language vendor, while explicit data types (user defined types) are types that are made by using or composing implicit data types.
Implicit data types in .Net compliant languages are mapped to types in the Common Type System (CTS) and CLS (Common Language Specification). Hence, each implicit data type in C# has its corresponding .Net type.
The following are the implicit data types in C#:

C# type	.Net type	Size in bytes	Description
Integral Types			
byte	Byte	1	May contain integers from 0-255
sbyte	SByte	1	Signed byte from -128 to 127
short	Int16	2	Ranges from -32,768 to 32,767
ushort	UInt16	2	Unsigned, ranges from 0 to 65,535
int (default)	Int32	4	Ranges from -2,147,483,648 to 2,147,483,647
uint	UInt32	4	Unsigned, ranges from 0 to 4,294,967,295
long	Int64	8	Ranges from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	UInt64	8	Unsigned, ranges from 0 to 18,446,744,073,709,551,615
Floating Point Types			
float	Single	4	Ranges from $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$ with 7 digits precision. Requires the suffix 'f' or 'F'
double (default)	Double	8	Ranges from $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ with 15-16 digits precision
Other Types			
bool	Boolean	1	Contains either true or false
char	Char	2	Contains any single Unicode character enclosed in single quotation mark such as 'c'
decimal	Decimal	12	Ranges from 1.0×10^{-28} to 7.9×10^{28} with 28-29 digits precision. Requires the suffix 'm' or 'M'

Implicit data types are represented in language using keywords. It is worth noting that string is also an implicit data type in C#. Lastly, implicit data types are value types and thus stored on the stack, while referenced types are stored using the heap. The garbage collector searches for non-referenced data in heap during the execution of program and returns that space to Operating System.

Note: The difference is that a value type stores its value directly, while a reference type stores a reference to the value.

System.Object

C# provides a "unified type system", whereby the object class is the ultimate base type for both reference and value types. All types - including value types - can be treated like objects. It is possible to call Object methods on any value, even values of "primitive" types such as `int`.

The object type is based on System.Object class in the .NET Framework and every method defined in the Object class is available in all objects in the system, including:

- Equals - Supports comparisons between objects.
- Finalize - Performs cleanup operations before an object is automatically reclaimed.
- GetHashCode - Generates a number corresponding to the value of the object to support the use of a hash table.
- ToString - Manufactures a human-readable text string that describes an instance of the class.

Example

```
using System;
class Test {
     static void Main() {
          Console.WriteLine(3.ToString());
```

```
            Console.ReadKey();
        }
    }
```

Calls the Object-defined `ToString` method on a constant value of type `int`.

Variables

A variable is the name given to a memory location holding a particular type of data. So, each variable is associated with a data type and a value. In C#, variables are declared as follows:

```
<data type> <variable>
```

You can initialize the variable as you declare it and can also declare/initialize multiple variables of the same type in a single statement, e.g.,

```
bool isReady = true;
float percentage = 87.88, average = 43.9;
char digit = '7';
```

In C# you must declare variables before using them. Also, there is the concept of "Definite Assignment" which requires local variables (variables defined in a method) to be initialized before being used as shown below

```
static void Main()
   {
     int age;
     //age = 18;
     Console.WriteLine(age);    // error
   }
```

Generally, C# does not assign default values to local variables. Also C# is a type safe language, i.e., values of particular data type can only be stored in their respective (compatible) data type. You can't store integer values in Boolean data types like we used to do in C/C++. C# is also a strongly typed language. Thus, all operations on variables are performed with consideration of what the variable's "Type" is. This is enforced at compile time.

Variable Default Values

A variable is either explicitly assigned a value or automatically assigned a default value.
The following are categories of variables are automatically initialized to their default values.

- Static variables
- Instance variables
- Array elements

Type	Default Value
All integer types	0
char type	'\x000'
float type	0.0f
double type	0.0d
decimal type	0.0m
bool type	False
enum type	0
All reference type	Null

Constants

Constants are variables whose values, once defined, and cannot be changed by the program. The value of a

constant must be computable at compile time i.e. you cannot initialize a constant with a value taken from a variable since by default, const members are static. Constants are declared by the const keyword as shown below

```
const double PI = 3.142;
```

It is conventional to use capital letters when naming constant variables.

Naming Conventions for variables and methods

Microsoft suggests using Lower Camel Notation (first letter in lowercase) for variables and Upper Camel Notation (first letter in uppercase) for methods. Each word after the first word in the name of both variables and methods should start with a capital letter. For example, `totalSalary` variable names and `GetTotal()` for method names. Although it is not mandatory to follow this convention, it is highly recommended that you strictly follow the convention.

Note: An identifier must not clash with a keyword. As a special case, the @ prefix can be used to avoid such a conflict.

Scope of Variables

In a program, a variable can be accessed within a particular block or can be accessed in the whole program. This is known as scope of variable. The following are various types of variables in C#.

i). Static variable: - The static variables are those variables which retain their value through the execution of the program.

ii). Instance variables: - In C#, one of the variables of a class that may have a different value for each object of that class. The instance variables hold the state of an object.

iii). Local variables: - These variables declared within a particular block of statements for instance for, while, if etc or method.

Boxing and Unboxing

An int value can be converted to object and back again to int. This is called boxing and unboxing. When a variable of a value type needs to be converted to a reference type, an object box is allocated to hold the value, and the value is copied into the box.

Unboxing is just the opposite. When an object box is cast back to its original value type, the value is copied out of the box and into the appropriate storage location. When unboxing converting from a reference type to a value type the cast is needed. This is because in the case of unboxing, an object could be cast to any type. Therefore, the cast is necessary so that the compiler can verify that the cast is valid per the specified variable type.

Example

```
class Test2 {
     static void Main() {
          int i = 123;
          object o = i; // boxing
          int j = (int) o; // unboxing
     }
}
```

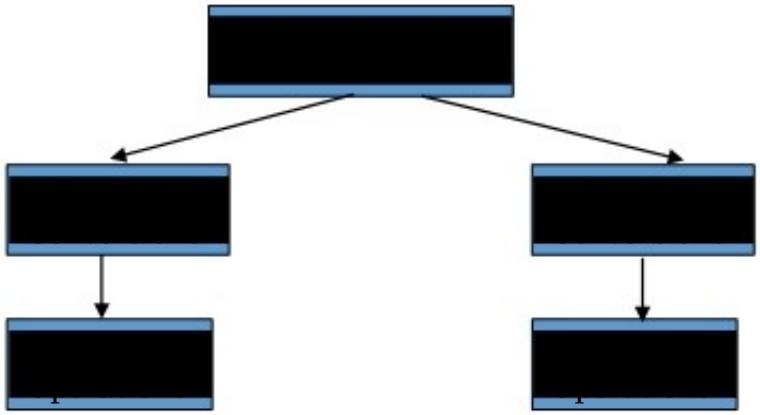
This type system unification provides value types with the benefits of object-ness, without introducing unnecessary overhead i.e. for programs that don't need int values to act like object, int values are simply 32 bit values while those that need int's to behave like objects, this functionality is available on-demand.

Type Conversion

Type conversion is converting one type of data to another type. It is also known as Type Casting. In C#, type casting has two forms:

- Implicit type conversion - These conversions are performed by C# in a type-safe manner. For example, conversions from smaller to larger integral types and conversions from derived classes to base classes.

Explicit type conversion - These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.



Explicit numeric conversions require a cast and runtime circumstances determine whether the conversion succeeds or information is lost during the conversion. An example of this:

```
int x = 123456; // 4bytes
short s = (short)x; //convert to 2 bytes
```

C# Type Conversion Methods

C# provides the following built-in type conversion methods as described:

Sr. No.	Methods
1	ToBoolean: - Converts a type to a Boolean value, where possible.
2	ToByte - Converts a type to a byte.
3	ToChar - Converts a type to a single Unicode character, where possible.
4	ToDateTime - Converts a type (integer or string type) to date-time structures.
5	ToDecimal - Converts a floating point or integer type to a decimal type.
6	ToDouble - Converts a type to a double type.
7	ToInt16 - Converts a type to a 16-bit integer.
8	ToInt32 - Converts a type to a 32-bit integer.
9	ToInt64 - Converts a type to a 64-bit integer.
10	ToSbyte - Converts a type to a signed byte type.
11	ToSingle - Converts a type to a small floating point number.
12	ToString - Converts a type to a string.
13	ToType - Converts a type to a specified type.
14	ToUInt16 - Converts a type to an unsigned int type.
15	ToUInt32 - Converts a type to an unsigned long type.
16	ToUInt64 - Converts a type to an unsigned big integer.

E.g. The function Convert.ToInt32() converts the data entered by the user to int data

Pointers

A pointer is a variable that holds the memory address of another type. In C#, pointers can only be declared to hold the memory addresses of value types.

Pointers are declared implicitly, using the dereferencer symbol *. The operator & returns the memory address of the variable it prefixes.

Example:

What is the value of i?

```
int i = 5;
```

```
int  *p;
p  =  &i;
*p = 10;
```

The use of pointers is restricted to code which is marked as unsafe (memory access).
A pointer can be declared in relation to an array, as in the following:

```
int[] a = {4, 5};
int *b = a;
```

Enumerations

An enumeration is a special kind of value type limited to a restricted and unchangeable set of numerical values. When we define an enumeration we provide literals which are then used as constants for their corresponding values. Generally, enumerations provides ways for attaching names to numbers.
The following code shows an example of such a definition:

```
public enum DAYS { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
Sunday};
```

The numerical values are set up according to the following two rules:
- For the first literal: if it is unassigned, set its value to 0.
- For any other literal: if it is unassigned, then set its value to one greater than the value of the preceding literal.

Notice that by default, the storage type for each item in an enumeration maps to `System.Int32`

System.Enum base class

.NET enumerations are implicitly derived from System.Enum and they are value types
The following are selected static members of System.Enum

- Format
- GetNames
- GetValues
- IsDefined
- Parse

Example

```
public class EnumTest
{
    public enum DAYS
    {
        Monday=1, Tuesday, Wednesday, Thursday, Friday,
        Saturday, Sunday
    };
    public static void Main() {
    Array dayArray =
     Enum.GetValues(typeof(EnumTest.DAYS));
     foreach (DAYS day in dayArray)
          Console.WriteLine("Number "+day.ToString("d")+ "
          of EnumTest.DAYS is " +day);
     Console.ReadKey();
    }
}
```

Console I/O

Console I/O is provided by the System.Console class, which gives you access to the standard input (Console.In), standard output (Console.Out) and standard error (Console.Error) streams.

i). Console Input

Console.In has two methods for obtaining input. Read() returns a single character as an int, or -1 if no more characters are available. ReadLine() returns a string containing the next line of input, or null if no more lines are available. Notice that Console.In.ReadLine() and Console.ReadLine() are equivalent.

ii). Console Output

Console.Out has two methods for writing output. Write() outputs one or more values without a newline character. WriteLine() does the same but appends a newline.
Write() and WriteLine() have numerous overloads, so that you can easily output many different types of data. Notice that Console.Out.WriteLine() and Console.WriteLine() are equivalent.

Command Line Arguments

Command Line Arguments allows a program to behave in a specific manner depending on the input provided at the time of execution. Command line arguments stored in the string array and specific with the Main() method.

Example

```
using System;
namespace MyProgram
{
     class CommandLine
     {
          static void Main(string [] args)
          {
               Console.WriteLine("Hello, {0} {1} !", args[0], args[1]);
               Console.WriteLine("Welcome to the CSharp Programming!"); }
     }
}
```

Note: .Net Framework SDK installation places the Visual C# .NET compiler (csc) in the directory such as C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727 directory for version 2.0.

Multiple Main Methods

C# allows you to define more than one class with Main() method. The Main() method indicates the entry point for execution of a program. As a result, where you define more than one Main() method in a C# program then there are more than one entry points for execution of a program. You need to specify the class name with the Main () method, which you want to execute first. It is mostly used to place test code in your classes.
The /main compiler option is used to specify a class in which to look for a Main method

Format:
/main:< className >

E.g.
csc MultipleMain.cs /main:ClassOne or csc MultipleMain.cs /main:ClassTwo

Example

```
using System;
namespace MyProgram
{
    class ClassOne
    {
        public static void Main(string [] args)
        {
            Console.WriteLine("Testing class one ");
        }
```

```
    }
     class ClassTwo
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Testing class two");
        }
    }
}
```

PROGRAM CONTROL

C# borrows most of its statements directly from C and C++, though there are some noteworthy additions and modifications. Control structures are categorized into sequence, decision and iteration

- Decision Construct statements
  - i). if / else statement
  - ii). switch statement
- Iteration Construct statements
  - i). for loop
  - ii). foreach loop
  - iii). while loop
  - iv). do/while loop

Decision Construct statements

i). if statement

An if statement selects a statement for execution based on the value of a boolean expression, and may optionally include an else clause that executes if the boolean expression is false.

Example

C# program that uses an if statement to write out two different messages depending on whether command-line arguments were provided or not.

```
using System;
class Test {
      static  void  Main(string[]  args)
      { if (args.Length == 0)
            Console.WriteLine("No arguments were provided");
      else
            Console.WriteLine("Arguments were provided");
      }
}
```

ii). switch statement

A switch statement executes the statements that are associated with the value of a given expression, or default of statements if no match exists.

The switch expression must be an integer type (including char) or a string. The case labels have to be constants. Unlike Java or C++, you cannot fall through from case to case if you omit the break and there's code in the case. You will get a compiler error instead, and have to use a goto to jump to the next case. If you have adjacent case labels, (i.e. No code in the case) then you can fall through.

Example:

```
using System;
class SwitchSelect
{
    static void Main()
    {
        string myInput;
```

```
        int myInt;
        begin:
        Console.Write("Please enter a number between 1 and 3: ");
        myInput = Console.ReadLine();
        myInt = Int32.Parse(myInput);
        // switch with integer type
        switch (myInt)
        {
            case 1:
                Console.WriteLine("Your number is {0}.", myInt);
                break;
            case 2:
                Console.WriteLine("Your number is {0}.", myInt);
                break;
            case 3:
                Console.WriteLine("Your number is {0}.", myInt);
                break;
            default:
                Console.WriteLine("Your number {0} is not between 1 and 3.",
                myInt);
                break;
        }
        decide:
        Console.Write("Type \"continue\" to go on or \"quit\" to stop: ");
        myInput = Console.ReadLine();
        // switch with string type
        switch (myInput)
        {
            case "continue":
                goto begin;
            case "quit":
                Console.WriteLine("Bye.");
                break;
            default:
                Console.WriteLine("Your input {0} is incorrect.", myInput);
                goto decide;
        }
    }
}
```

Iteration Construct Statements

i). **while statement:-** A while statement conditionally executes a statement zero or more times - as long as a Boolean test is true.

ii). **do..while statement:-** A do statement conditionally executes a statement one or more times.

Example

A C# program that reads from the console until the user types “Exit”.

```
using System;
class Test {
     static void Main() {
          string s;
          do {
               Console.WriteLine("Enter your name or Exit to quit"); s
               = Console.ReadLine();
          }while (s != "Exit");
     }
```

```
}
```

iii). for statement

A for statement evaluates a sequence of initialization expressions and then, while a condition is true, repeatedly executes a statement and evaluates a sequence of iteration expressions.

Example

A C# program that uses a for statement to generate and display integer values 1 through 10.

```
using System;
class Test {
     static void Main() {
          for (int i = 0; i < 10; i++)
          Console.WriteLine(i);
          Console.ReadKey();
      }
}
```

iv). foreach statement

A foreach statement lets you iterate over the elements in arrays and collections.

Example

A C# program that uses a foreach statement to iterate over the elements of an array.

```
using System;
class ForEachTest {
     static void Main() {
          int[] numbers = new int[] {1,2,3,4,5};
          foreach (int i in numbers)
               Console.WriteLine("Value is {0}", i);
          Console.ReadKey();
     }
}
```

Jump Statements

C# permits a jump from one statement to the end or beginning of a loop as well as jump out of a loop. The following are various types of jump statements:

i). The break statement

The break statement cannot only be used to exit from a case in a switch statement but also from for, foreach, while, or do...while loops. The Control will switch to the statement immediately after the end of the loop. If the statement occurs in a nested loop, control will switch to the end of the innermost loop and if the break occurs outside of a switch statement or a loop, a compile-time error will occur.

Example

```
using System;
class BreakTest
{
    static void Main()
    {
        for (int i=1; i<=10; i++)
        {
            if (i > 5)
                break;
            Console.WriteLine("In the loop, value of i is {0}.", i);
        }
```

```
        Console.ReadKey();
    }
}
```

ii). The continue statement

The continue statement is similar to break, and must also be used within a for, foreach, while, or do...while loop. However, it exits only from the current iteration of the loop, meaning execution will restart at the beginning of the next iteration of the loop, rather than outside the loop altogether.

Example

```
using System;
class ContinueTest
{
    static void Main()
    {
        for (int i=1; i<=10; i++)
        {
            if (i==5)
                continue;
            Console.WriteLine("In the loop, value of i is {0}.", i);
        }
        Console.ReadKey();
    }
}
```

iii).The return statement

The return statement is used to exit a method of a class, returning control to the caller of the method. If the method has a return type, return must return a value of this type; otherwise if the method returns void, then you should use return without an expression.

iv). The goto Statement

The goto statement is used to make a jump to a particular labeled part of the program code.

```
goto Label1;
Console.WriteLine("This  won't  be  executed");
Label1:
Console.WriteLine("Continuing execution from here");
```

Restrictions on goto

- You can't jump into a block of code such as a for loop,

You can't jump out of a class, and

- we can't exit a finally block after try...catch blocks

METHODS in C#

Methods are extremely useful because they allow us to separate program logic into different units.
Syntax:

```
modifiers  type  methodname(formal-parameter-list)
{
  method---body
}
```

Example:

A method that accepts two integers and then it computes and returns the product of the numbers

```
int Product(int x,int y)
{
     int m=x*y;
     return(m);
}
```

Method Modifiers

The following are some of the methods modifies

Modifier	Description
Public	The method can be access from anywhere, including outside the class.
protected	The method can be access from within the class to which it belongs, or a type derived from that class.
internal	The method can be accessed from within the same program (assembly).
private	The method can only be accessed inside the class to which it belongs.
Static	The method does not operate on a specific instance of the class
Extern	The method is implemented externally, in a different language.

Types of Methods

There are two types of methods

i). Static Methods

ii). Non-static Methods (instance methods)

Static methods belongs to the whole class, while non-static methods belong to each instance created from the class.

Example

The following is a program that uses a static method to compute and return the square of a floating point number

```
using System;
class StaticMethod
{
     static void Main()
     {
          double y = Square (5F); //Method Call
          Console.WriteLine(y);
          Console.ReadKey();
     }
     static double Square ( float x )
     {
          return ( x * x );
     }
}
```

Method Parameters

For managing the process of passing values & getting back the results, C# employs four kinds of parameters.

i). Value Parameters

ii). Reference Parameters

iii). Output Parameters

iv). Parameter Arrays

i). Pass By Value

By default, method parameters are passed by value. When a method is invoked, the value of actual parameters are assigned to the corresponding formal parameters. Any changes to formal parameters does not affect the actual parameters.

ii). Pass By Reference

We can force the value parameters to be passed by reference using the ref keyword. This mechanism does not create a new storage location. It generally represents the same storage location as the actual parameter. When a formal parameter is declared as ref, the corresponding actual argument in the method invocation must be declared as ref. It is generally used, when we want to change the values of variables in the calling method.

Example

```
using System;
class  PassByRef
{
     static void Swap  ( ref int x, ref int y )
     {
          int  temp  =  x;
          x = y;
          y = temp;
     }
     static void Main( )
     {
          int m = 100;
          int n = 200;
          Console.WriteLine("Before Swapping;");
          Console.WriteLine("m = " + m);
          Console.WriteLine("n = " + n);
          Swap (ref m , ref n );
          Console.WriteLine("After Swapping;");
          Console.WriteLine("m = " + m);
          Console.WriteLine("n = " + n);
          Console.ReadKey();
     }
}
```

iii). The Output Parameters

It is used to pass results back to the calling method and is declared with the out keyword.
When a formal parameter is declared as out, the corresponding actual argument in the method invocation must also be declared as out.
The variable is passed by reference, so any changes that the method makes to the variable will persist when control returns from the called method. Generally out parameters allows the caller to obtain multiple return values from a single method invocation. If an out parameter isn't assigned a value within the body of the function, the method won't compile.

Example

```
using System;
class  Output
{
     static void Square  ( int x, out int y )
     {
          y = x * x;
     }
```

```
        static void Main( )
        {
             int m;  //need not be initialized
             Square (10, out m);
             Console.WriteLine ("m = " + m);
             Console.ReadKey();
        }
    }
```

iv). Variable Argument Lists

We can define methods that can handle variable number of arguments using parameter arrays. Parameter arrays are declared using the keyword params and should be a one-dimensional arrays.

Example

```
using System;
class  Params
{
     static void Parray (params int [] arr)
     {
          Console.Write("array  elements  are:");
          foreach ( int i in arr)
               Console.Write(" " + i);
          Console.WriteLine( );
     }
     static void Main( )
     {
          int [] x = {11, 22, 33, 44};
          Parray (x);            //call 1
          Parray ( )   ;         //call 2
          Parray (100, 200);    //call 3
          Console.ReadKey();
     }
}
```

Note: params cannot be used for ref and out parameters

Method Overloading

C# allows you to declare more than one method with the same name. Overloaded methods must differ in the number and/or type of arguments they take. The return type does not play any part in the overload resolution, since it's always possible to call a method without using the return value.

Example

```
using System;
class Overloading
{
     static void Main()
     {
          Console.WriteLine(add(2,3));
          Console.WriteLine(add(2.6F,3.1F));
          Console.WriteLine(add(312L,22L,21));
     }
     static int add(int a,int b)
     {
          return(a+b);
     }
```

```
     static float add(float a,float b)
     {
          return(a+b);
     }
     static long add(long a,long b,int c)
     {
          return(a+b+c);
     }
}
```

Math Class

The Math class present in the System Namespace contains

- Static Members: E and PI
- Mathematical Methods

The following are some of the methods found in the Math class

Method	Description
Sin()	Sine of an angle in radians
Cos()	Cosine of an angle in radians
Asin()	Inverse of Sine
Sinh()	Hyperbolic sine
Sqrt()	Square Root
Pow()	Number raised to a given power
Exp()	Exponential
Log()	Natural logarithm
Abs()	Absolute value
Min()	Lower of two numbers
Max()	Higher of two numbers

Example

Write a C# program that inputs the radius of a sphere and then computes and displays the volume and surface are respectively

```
using System;
class  Sphere
{
    static void Main()
    {
        int radius;
        double volume, surfaceArea;
        Console.WriteLine("Enter the radius of a sphere");
        radius = int.Parse(Console.ReadLine());
        volume = 4.0/3.0*Math.PI * Math.Pow(radius, 3);
        surfaceArea = 4 * Math.PI * Math.Pow(radius, 2);
        Console.WriteLine("Volume: {0} \t Suface area: {1}",
          volume, surfaceArea);
    }
}
```