Agenda: C# Language Syntax

- Why Programming Language
- Introduction to C# and its Evolution
- Primitive Data Types and Variable Declarations
- Value Types and Reference Types
- Implicit and Explicit Casting
- String and String Builder
- Boxing and Unboxing
- Type Inference
- Constant and Enumerated Data types
- Operators
- Control Statements
- Working with Arrays
- Working with Methods

Why Programming Language

An Application is a collection of instructions and data flowing through those instructions.

As a developer you are supposed to write/build an application, implies you have to instruct the computer to perform an action as per your needs.

Computer cannot understand our Native language like English, French, Chinese, Hindi, etc…

It will only understand Binary Language and as Humans we don't understand binary so easily…

So we need now an intermediate solution like Programming Language.

Introduction to C# and its Evolution

- C# (pronounced see sharp) is a programming language encompassing strong typing, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines.
- It can be considered as "C" family language. The name "C sharp" was inspired by musical notation where a sharp indicates high pitch note. This is similar to the language name of C++, where "++" indicates that a variable should be incremented by 1.
- C# is one of the programming languages designed for the Common Language Infrastructure.
- C# is intended to be a simple, **modern**, general-purpose, object-oriented programming language.
- Its development team is led by **Anders Hejlsberg.**

C# Version History

Version	Date	.NET Framework	Visual Studio
C# 1.0	January 2002	.NET Framework 1.0	Visual Studio 2002
C# 1.2	April 2003	.NET Framework 1.1	Visual Studio 2003

C# 2.0	November 2005	.NET Framework 2.0	Visual Studio 2005
C# 3.0	November 2007	.NET Framework 2.0 (Except LINQ) .NET Framework 3.0 (Except LINQ) .NET Framework 3.5	Visual Studio 2008 Visual Studio 2010
C# 4.0	April 2010	.NET Framework 4	Visual Studio 2010
C# 5.0	August 2012	.NET Framework 4.5	Visual Studio 2012 Visual Studio 2013
C# 6.0		.NET Framework 4.51	Visual Studio 2015
C# 7.X		.NET Framework 4.6.X	Visual Studio 2017

	C# 2.0	*C# 3.0*	*C# 4.0*	*C# 5.0*
Features added	• Generics • Partial types • Anonymous methods • Iterators • Nullable types • Private setters (properties) • Method group conversions (delegates)	• Implicitly typed local variables • Object and collection initializers • Auto-Implemented properties • Anonymous types • Extension methods • Query expressions • Lambda expressions • Expression trees • Partial Methods	• Dynamic binding • Named and optional arguments • Generic co- and contravariance • Embedded interop types ("NoPIA")	• Asynchronous methods • Caller info attributes

Primitive Data Types and Variable Declarations

Why Data types:

1. Based on the data type the **size** of variable and the **format** in memory is decided.
2. Based on the data type the **compiler** is going to **validate** the expressions making the language type safe.
3. Based on the data types of **operands** the **expression** will be evaluated to particular at **runtime**.

Primitive Datatypes

Integral Types

DataType	Size	. Net (CTS)	Comments
byte	1	System.Byte	0 - 255 It is Unsigned (0 to 2^8)
sbyte	1	System.SByte	-128 to 127 – Signed (-2^7 to 2^7 - 1)
short	2	System.Int16	Range for n – bits signed numbers: **-2^{n-1} to 2^{n-1} - 1** n-bits unsigned numbers = **0 to 2^n - 1**
ushort	2	System.UInt16	
int	4	System.Int32	
uint	4	System.Unt32	

long	8	System.Int64	
ulong	8	System.UInt64	

Floating Point Types

float	4	System.Single	Has up to 8 digits after decimal
double	8	System.Double	Has up to 16 digits after decimal

Fixed Point Type

decimal	16	System.Decimal	Has **fixed** 28 digits after decimal and its fixed

Note: Even though the decimal datatype as decimal point **it's not floating**.

Other Types

char	2	System.Char	Uses Unicode Charset
string *		Systring.String	Uses Unicode Charset
bool	1	System.Boolean	
object *		System.Object	Generic Datatype

All the above datatypes are ValueTypes except **String and **Object, which** are **Reference Types**.

Variable Declaration Syntax:

```
int a,b;
int a=10, b=20;
```

- A **local variable** declared must be **explicitly initialized** before used in an expression otherwise gives a compilation error.
- A variable declared in a block is local to the block in which it is declared.
- A variable declared in outer block cannot be re-declared in the inner block.

Classification of Memory in an application:

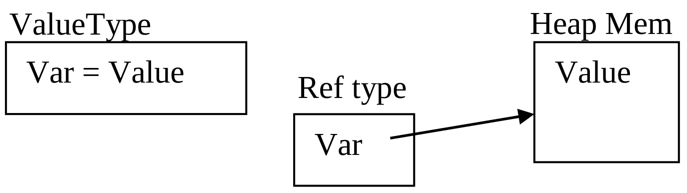
1. **Global Memory**: Used by all global variables. These variables are allocated memory when the application begins and will remain in memory throughout the life of the application.
2. **Stack Memory**: Used by local variables of a method. When a method is invoked a stack of memory is allocated to and the stack is cleared when the method returns.
3. **Heap Memory:** All dynamic memory requirements of an application are fulfilled from the heap memory. After allocating some memory from heap to a variable, once its job is completed the memory must be returned back to heap so that the same can be reused for another variable.

Category of Data Types in .NET

1. Value Types
2. Reference Types

- The value type of variable has value where as the value of a reference type is reference to value (object) on heap.

- The value (object) of a reference type is always allocated heap memory.
- Value types are allocated memory based on the scope of the variable. If it's Local Variable or Parameter its allocated memory on stack and if it's a member of an object its allocated memory on Heap.

Value Types directly hold the value. Ex: All Basic Types, Structures & Enum **Reference Types** hold the reference to the value on HEAP. Ex: String, Object, Class, Arrays, Delegates	 ValueType Var = Value Ref type Var Heap Mem Value

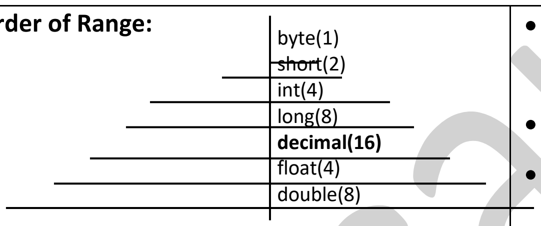
Types of Casting

Casting is converting data from one form to another form.

If RHS expression and LHS variable are not of same datatype then casting is required.

Implicit Casting: If every possible value of RHS expression is valid for a variable on LHS variable.

Explicit Casting: If an RHS expression is assigned to LHS and if there is a possibility of data loss then explicit casting is needed.

Increasing order of Range:
 byte(1)
~~short~~(2)
int(4)
long(8)
decimal(16)
float(4)
double(8)
	• Casting is done based on **range** and **not** based on **size** of the data type.
	• byte can be implicitly assigned to any data type.
	• short can be implicitly assigned to all data types except byte and so on

```
using System;
class Program
{
  public static void Main()
  {
    int n = 256;
    byte b;
    byte b1, b2, b3;
    b1 = b2 = 10;
    b1 = b2 + b3; //Compilation Error
//if either byte, short or char variables are used in an expression they are automatically raised to the rank of int.
    b1 = (byte) (b2 + b3);
```

Handling Overflow Checks

```
    unchecked //Overflow checks are not done…
    {
      b = (byte) n;
    Console.WriteLine(b);
}
checked //Overflow checks are done…
{
    b = (byte) n;
    Console.WriteLine(b);
}
```

To Enable / Disable Overflow checks:

Project → Properties → Compile → Scroll and Click on Advanced Compile Options → Check Integer Overflow Checks

You cannot use the **checked** and **unchecked** keywords to control floating point (non-integer) arithmetic. **The checked and unchecked keywords control only integer arithmetic**. Floating point arithmetic never throws **OverflowException**.

```
decimal dec = 0.6M;
```

//long to float

```
    long lng = 10L;
    f = lng; //because range of long is smaller than float
    lng = (long) f;
```

//float and decimal requires casting

```
    dec = 10; //Integer to decimal
    //f = dec; //Invalid
    //dec = f; //Invalid
    f = (float) dec;
    dec = (decimal) f;
//"decimal" should be explicitly casted to every other type if required.
```

//int to char.

```
    n = c; //Implicit Casting
    /*c = n; invalid */
    c = (char) n;  //Explicit Casting
```

Character: 'A' --- 'Z'   'a' --- 'z'   '0' --- '9'

UNICODE: 65 --- 90   97 ---122   48 --- 57

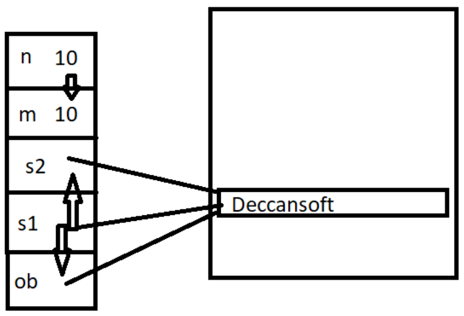
//bool – int - anydatatype explicit or implicit casting is not allowed either direction

Object Data Type - Boxing & Unboxing

```
int n;
n = 10;
int m = n;

string s1 = "Deccansoft";
string s2 = s1;

object ob;
ob = s1;
```

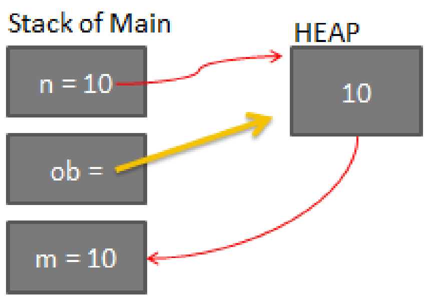


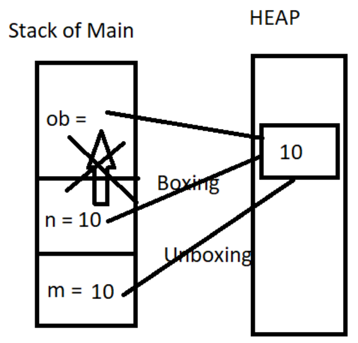
Many times until runtime we don't know the kind of data our variable will have…In this kind of situations we have to use "object" data type.

Object is reference data type which can store any kind of value including value types and reference types.

- **Boxing** is the term used to describe the transformation from **Value Type to object type (specifically when variable if type Object)** only. The runtime creates a temporary reference-type box for the object on the heap.
- **UnBoxing** is the term used to describe the transformation from **object to value type**. We use the term cast here, as this has to be done explicitly.

```
static void Main()
{
   int n = 10;
   object ob = n; //boxing
   int m = (int)ob; //unboxing
}
```





```
static void Main(string[] args)
{
    int n;
    int m;
    object ob;
    n = 10;
    ob = n;
    //...
    m = (int)ob;
    //...
}
```

1. Boxing / Unboxing should be used only in situations where until runtime we don't know the type of data we are going to deal with.
2. When a value is boxed to an object type, the object type variable **cannot** be used in any **mathematical operations**.
3. When the value of object type variable cannot be assigned to variable on LHS, an Exception of type **InvalidCastException** is thrown.
4. Excessive usage of Object datatype makes the language "**Loosely Typed**" and also because of frequent casting requirement while Boxing and Unboxing performance is also **degraded**.

var - Type Inference (3.5 feature)

- Based on the data type of RHS expression in declaration compiler will automatically inter the type of the variable.
- Once inferred the type cannot be changed.

```
var n = 10; //n is int
var s = “Demo” //s is string
```

var is static typed - the compiler and runtime know the type - they just save you some typing...

The following are 100% identical:

```
    var s = "abc";
    Console.WriteLine(s.Length);
```

and

```
    string s = "abc";
    Console.WriteLine(s.Length);
```

All that happened was that the compiler figured out that s must be a string (from the initializer). In both cases, it knows (in the IL) that s.Length means the (instance) string.Length property.

dynamic Data Type – **Late Binding (4.0 Feature)**

- The dynamic type enables the operations in which it occurs to bypass compile-time type checking. Instead, these operations are resolved at run time.
- The dynamic type simplifies access to COM APIs such as the Office Automation APIs, and also to dynamic APIs such as HTML Document Object Model (DOM).
- Type dynamic behaves like type object in most circumstances. However, operations that contain expressions of type dynamic are not resolved or type checked by the compiler. The compiler packages together information about the operation, and that information is later used to evaluate the operation **at run time**.
- As part of the process, variables of type dynamic are compiled into variables of type object. Therefore, type dynamic exists only at compile time, not at run time.

```
class ExampleClass
{
  static dynamic field;
  dynamic prop { get; set; }
  public dynamic exampleMethod(dynamic d)
  {
    dynamic local = "Local variable";
    int two = 2;
    if (d is int)
      return local;
    else
      return two;
  }
}
class Program
{
  static void Main(string[] args)
```

```
    {
        ExampleClass ec = new ExampleClass();
        Console.WriteLine(ec.exampleMethod(10));
        Console.WriteLine(ec.exampleMethod("value"));

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

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

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

Understanding Strings

//string and other types

```
    s = "100";
    n = (int) s; //Invalid – String cannot be casted
    n = int.Parse(s); //if fails throws, FormatException
    bool bln = int.TryParse(s, out n); //If Unsuccessful n will reset to 0
    //n = (int) s; //Direct casting of string to int is not allowed.
```

//int to string

```
    s = n.ToString();
    //s = (string) n //Casting of int to string is not allowed
```

Program to Print the Char equivalent of the Ascii value read from the keyboard

```
class Program
{
    static void Main(string[] args)
    {
        string str;
        str = Console.ReadLine();
        int n = 100;
        if (int.TryParse(str, out n))
            Console.WriteLine((char)n);
    else
      Console.WriteLine("Invalid number");
  }
}
```

Try: Write a Program to Read two Integers from Keyboard and Print the sum of these numbers

String interpolation

It's a feature which can eradicate the use of String.Format method which at times becomes little confusing.

```
class Person
{
  public string Name;
  public int Age;
}
class Program
{
  static void Main()
  {
    Person p = new Person() { Name = "Abcd", Age = 25 };
    var s = String.Format("{0} is {1} year(s) old", p.Name, p.Age);
    Console.WriteLine(s);
    var s1 = $"{p.Name} is {p.Age} year(s) old";
    Console.WriteLine(s1);
  }
}
```

About String and StringBuilder

Strings in .net are called as **Immutable** (not modifiable) objects. They are **Reference Types** and hence are allocated memory on heap.

```
string str = "Microsoft";
str.Replace("c", "K"); //Incorrect Usage – str doesn't change.
Console.WriteLine(str); //Output = Microsoft
str = str.Replace("c", "K"); //Correct Usage
Console.WriteLine(str);// Output = "MiKrosoft"
```

StringBuilder:

Because of the above behavour it is not recommended that the string datatype is used for those variables which need frequent modifications in their value.

Instead we should use the class called as “**System.Text.StringBuilder**” for those operations which need very frequent modification to the string. It allocates memory to the string in blocks and the capacity is automatically managed.

```
class Program
{
  static void Main(string[] args)
  {
    string str;
    System.Text.StringBuilder sb = new System.Text.StringBuilder();
    Console.WriteLine(sb.Capacity);  //output=4
    for (int i = 0; i < 100; i++)
      sb.Append("A");
    Console.WriteLine(sb.Capacity); //output=128
    Console.WriteLine(sb.Length); //output=100
    str = sb.ToString();
  }
}
```

Comparison of Time taken by string and StringBuilder

```
using System;
class Program
{
  static void Main()
  {
    System.Diagnostics.Stopwatch sw = System.Diagnostics.Stopwatch.StartNew();
    string s = "";
    for (int i = 0; i < 10000; i++)
    {
      s += "A";
    }
    sw.Stop();
    Console.WriteLine(sw.ElapsedTicks);

    sw = System.Diagnostics.Stopwatch.StartNew();
    System.Text.StringBuilder sb = new System.Text.StringBuilder(10000);
    for (int i = 0; i < 10000; i++)
    {
      sb.Append("A");
    }
    s = sb.ToString();
    sw.Stop();
    Console.WriteLine(sw.ElapsedTicks);
  }
}
```

Constant and Enumerated Data Types

Constants

1. Constants are either Literal constants or Named constants (they are not variables)
2. **Constants increase readablity of code**.
3. When the code is compiled all the occurances of contant in code is replaced with the value of that constant.
4. If a constant value has to be changed during development, it can be changed at one place and that will be reflected everywhere.

Literal Constants

L – long, M – decimal, F – float, D – double

Example: 10L is Long, "A" is String, 'A' is Char, 10.01 or 10.01D is double

Declaration Syntax:

const double **PI** = 3.14;

Enumerated Data Types

1. Enums can be subtype of integral types only i.e. byte, short, int, long and sbyte, ushort, uint and ULong
2. Enum is a collection of constants and can be used for **more redability in code**.
3. If an invalid integer is casted to enum variable, it doesn't throw an exception.
4. Default value of the first Enum member is "0".
5. Enum is a datatype by itself and can be declared outside the Class or Module.

```
enum WeekDay : int //WeekDay is subtype of int
{
  Sun=1, Mon, Tues, Wed=5, Thus, Fri, Sat
}
```

In Main:

```
n = 2; //n is int
WeekDay wd = WeekDay.Sat;
wd = (WeekDay) n; //Explicit
//n = wd; //Invalid
n = (int) wd; //Explicit
```

Operators

Arithmetic	+, -, * , / , %
Logical Operators	&& , ||, ^, !
Ternary Operator	?:
String concatenation	+
Increment, decrement	++ , --
Bitwise	<< , >>, & , | , ~ (complement/negation) (Only Integral Types)

Relational	= = , != , < , > , <= , >=
Assignment	= , += , -= , *= , /= , %= , &= , | = , ^= , <<= , >>=
Type information	is , sizeof , typeof, as
Indirection and Address	* , -> , [] , &

- Dividing a Integral Type with zero (Integral Division Operator) throws **DivideByZeroException**
- Floating Point Division by zero **is not** a runtime exception but the value would be either **PositiveInfinity** (if numerator is greater than zero) or **NegativeInfinity** (if numerator is less than zero) or **NaN** (if numerator is also zero)

Control Statements

if, switch, goto, while, for, foreach, try…catch…finally

if-statement:

```
  if (BooleanExpression)
  {
      statement;
  }
  else if (Boolean-Expression)
  {
     statement;
  }
  else
  {
     statement;
  }
```

Write a program to print if the command line argument provided is an odd number or even number

```
class Program
{
  public static void Main(string []args)
  {
    int n;
    if (args.Length == 0)
      Console.WriteLine("Please provide a number");
    else if (!int.TryParse(args[0], out n))
      Console.WriteLine("Not a number");
    else if (int.Parse(args[0]) % 2 == 0)
      Console.WriteLine("Even number");
    else
      Console.WriteLine("Odd number");
  }
}
```

Switch Statement

```
  int expr;
  switch  (expr) //expr can only integral type / char / string
  {
       case 0: //value of case must be a constant.
          statements;
          break; // or goto default;  can be used.
      case 1:
      case 2:
          statements;
          break; //break must be present after every case with statements
      default:
          statements;
          break; //break must be present after default also.
  }
```

Program: Write a program to print grade of the person based on the marks scored.

goto statement:

```
    int n = 0;
 l1:
    Console.WriteLine("One");
    n++;
    if (n < 5)
       goto l1;
    Console.WriteLine("Two");
 exit:
    Console.WriteLine("Exit");
```

while…..loop

```
while (BooleanExpression)
{
      Statements;
}
do
{
       Statements
}
while (BooleanExpression)
```

Program: Print the table of any number read from the keyboard.

```
public static void Main()
{
    int n;
    Console.Write("Table of: ");
    n = int.Parse(Console.ReadLine());
    string s = "";
    for (int i = 1; i <= 10; i++)
        s += n + "*" + i + "=" + n * i + "\n";
    Console.WriteLine(s);
}
```

for and foreach statements

```
for ( initializer; condition; iterator )
{
        statements;
}
class Program
{
    static void Main(string[] args)
    {
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                if (i == j)
                    break;
                Console.WriteLine(i + " " + j);
            }
        }
    }
}
```

What is the o/p of the above program???

```
for (int i = 0; i < 10; i++)
{
        if (i == 7) break;
        if (i == 3)  continue;
        Console.WriteLine(i);
}
```

What is the o/p of the above program???

Program: To show the Pyramid of Numbers

```
class ProgramForPyramid
{
   static void Main(string[] args)
   {
      int k=0;
      for (int i = 0; i < 4; i++)
      {
         for (int j = 0; j <= i; j++)
            Console.Write(k++ + "\t");
         Console.WriteLine();
      }
   }
}
```

```
foreach (DataType  identifier  in <Array or Collection>)
{
     embedded-statements;
}
```

The following for loop prints all the command line arguments.

```
foreach (string s in args)
   Console.WriteLine(s);
```

Technique1 : To Start building your Programming Skills

1. Write the program - Copy from handout...
2. Execute the program...
3. TRY - NOT NECESSARY to understand

For Second Program

Right click on Project --> Add Class --> Name=Program2

inside Class Program2 --> svm. tab tab

```
using System;
class Program2
{
    static void Main(string[] args)
    {


    }
}
```

To Run: Solution Explorer --> Right click on project --> Properties--> Application --> Startup Object = Program2

For Third Program

Repeat the same steps as in Program2.

.

Working with Arrays

1. Arrays are **reference types** and thus are allocated memory on heap.
2. Every element of an array is automatically initialized to a default value based on its datatype.
3. They are always **dynamic** because we can **SET** the size of the arrays at runtime.
4. Size of array can never be changed at runtime.
5. Trying to access an element of array with invalid index throws **IndexOutofRangeException** Exception.
6. All arrays irrespective of their type are by default inherited from **System.Array** class.

Single-Dimensional Arrays

```
int [] myArray = new int [5];
string []myStringArray = new string[5];
```

When you initialize an array upon declaration, it is possible to use the following shortcuts:

```
int[] myArray = {1, 3, 5, 7, 9};
string[] weekDays = {"Sun", "Sat", "Mon", "Tue"};
```

It is possible to declare an array variable without initialization, but you must use the new operator when you assign an array to this variable.

For example:

```
int[] myArray;
myArray = new int[] {1, 3, 5, 7, 9};   // OK
myArray = {1, 3, 5, 7, 9};   // Error
weekdays = new string[] {“Sunday”, “Monday”, “Tuesday”};
```

Multi-Dimensional Arrays

```
int[,] myArray = new int[4,2];
```

Also, the following declaration creates an array of three dimensions, 4, 2, and 3:

```
int[,,] myArray = new int [4,2,3];
```

You can initialize the array upon declaration as shown in the following example:

```
int[,] myArray = new  int[,] {{1,2}, {3,4}, {5,6}, {7,8}};
```

You can also initialize the array without specifying the rank:

```
int[,] myArray = {{1,2}, {3,4}, {5,6}, {7,8}};
```

If you choose to declare an array variable without initialization, you must use the new operator to assign an array to the variable. For example:

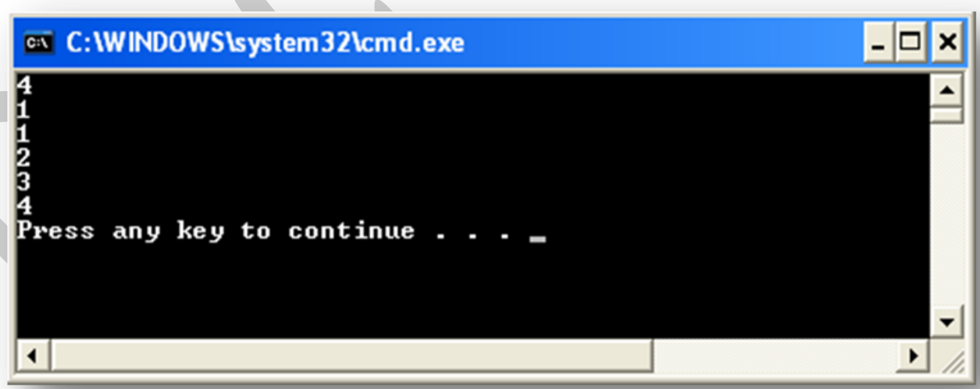
```
int[,] myArray;
myArray = new int[,] {{1,2}, {3,4}, {5,6}, {7,8}};   // OK
myArray = {{1,2}, {3,4}, {5,6}, {7,8}};   // Error
```

Program to use arrays:

```
using System;
class Program
{
   static void Main(string[] args)
   {
      int[] ar = new int[] { 1, 2, 3, 4 };
      Console.WriteLine(ar.Length);
      Console.WriteLine (ar.Rank); //Prints Number of Dimensions in array.
      foreach (int n in ar)
         Console.WriteLine(n);
   }
}
```

Code: 3.9 **C#**

Here is the output of executing above code:



```
4
1
1
2
3
4
Press any key to continue . . .
```

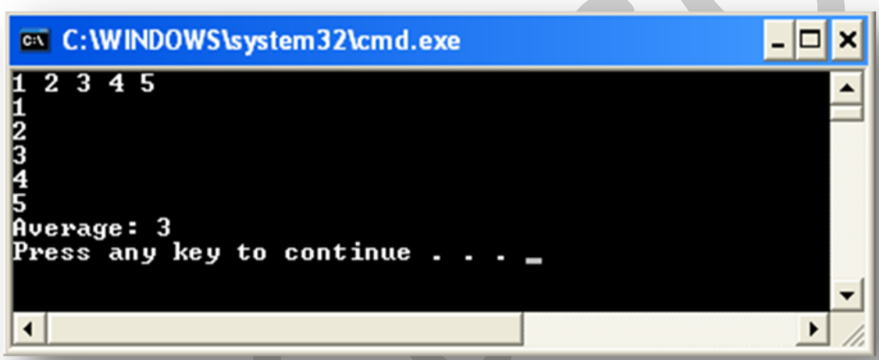
Program: To read a list of numbers separated by space and print the Average of all those numbers.

```
using System;
class ProgramForMaxOfAnyNumbers
{
   static void Main(string[] args)
```

```
    {
        string str = Console.ReadLine();
        string[] ar = str.Split(' ');
        int sum = 0;
        for (int i = 0; i < ar.Length; i++)
        {
            sum += int.Parse(ar[i]);
            Console.WriteLine(ar[i]);
        }
        Console.WriteLine("Average: " + 1.0 * sum / ar.Length);
    }
}
```

Code: 3.10 **C#**

Here is the output of executing above code:



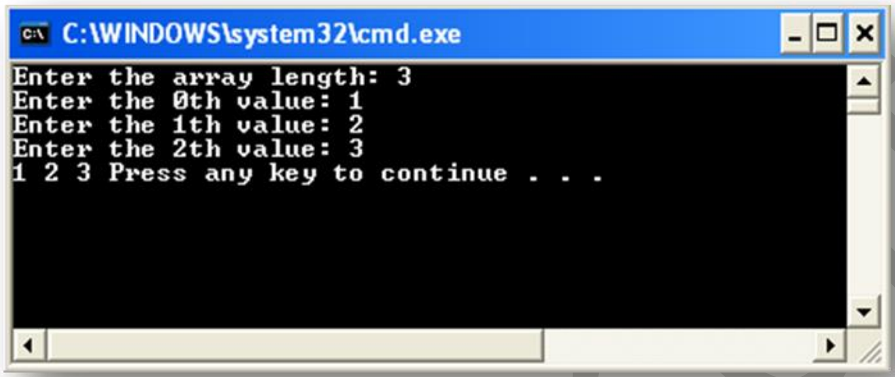
```
1 2 3 4 5
1
2
3
4
5
Average: 3
Press any key to continue . . . _
```

Program: To read length and data for an array from keyboard print the same.

```
using System;
class ProgramForMaxOfAnyNumbers
{
    static void Main()
    {
        Console.Write("Enter the array length: ");
        int n;
        n = int.Parse(Console.ReadLine());
        int[] ar = new int[n];
        for (int i = 0; i < n; i++)
        {
            Console.Write("Enter the " + i + "th value: ");
            ar[i] = int.Parse(Console.ReadLine());
        }
        for (int i = 0; i < ar.Length; i++)
        {
            Console.Write(ar[i] + " ");
        }
    }
}
```

Code: 3.11 | C#

Here is the output of executing above code:



```
Enter the array length: 3
Enter the 0th value: 1
Enter the 1th value: 2
Enter the 2th value: 3
1 2 3 Press any key to continue . . .
```

Working with Methods

1. Method Overloading
2. Optional Parameters (4.0 feature)
3. Named Arguments (4.0 feature)
4. params Parameter
5. Passing argument by value, ref and out

Method Overloading:

1 Methods are overloaded when they have same name and different parameters.

2 Parameters must be different either in their **data type** or in their **count**.

3 Method cannot be overloaded based on Return Type or Parameter names.

4 Call to the Overloaded method is resolved at compile time and is done based on data type and count of arguments passed to the method.

5 While resolving the called method, the compiler searches for a direct match of arguments and parameters. Only if a direct match is not available it would then use nearest match for resolving the call.

Optional parameters

1. Every Optional Parameter must have default value.
2. Once a parameter is declared as optional all subsequent parameters in the list also must be declared as optional
3. While calling the method with optional parameters, we may or may not provide the value/argument for optional parameters. If it’s not provided the default value assigned to the parameter will be used.

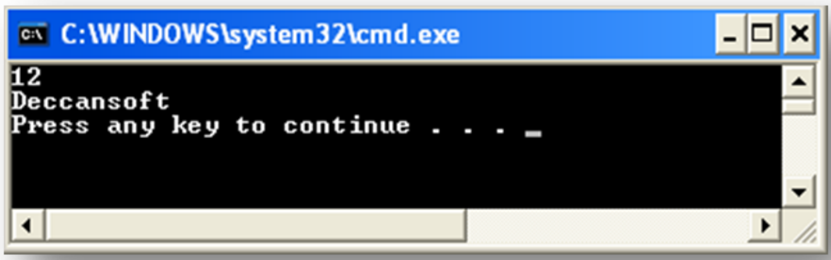
Named Arguments

While calling a method we can pass arguments based on parameter names and we don’t have to then pass them in sequence.

Method overloading

```
using System;
class Program
{
    public static void Main()
    {
        int res = Add(10, 2);
        Console.WriteLine(res.ToString());
        string str;
        str = Add("Deccan", "soft");
        Console.WriteLine(str);
        res = Add(10, 2, 4); //Named Arguments
        res = Add(10, 2, d:4);
        res = Add(b:2, a:10, d:4);
    }
    static int Add(int a, int b)
    {
        return Add(a, b, 0);
    }
    static int Add(int a, int b, int c=0,int d=0)
    {
        return a + b + c + d;
    }
    static string Add(string s1, string s2)
    {
        return s1 + s2;
    }
}
```

Here is the output of executing above code:



```
12
Deccansoft
Press any key to continue . . .
```

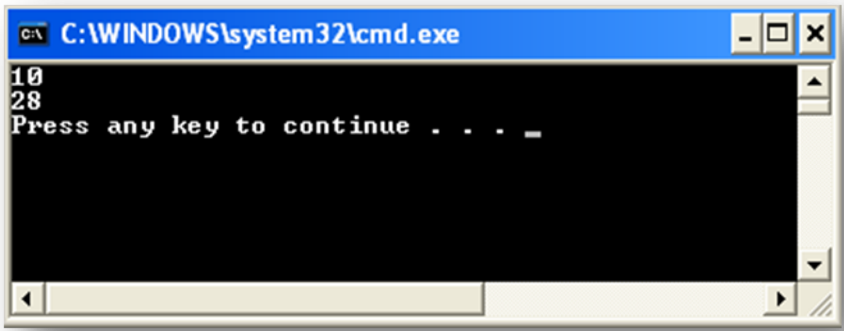
params Parameters:

1. Only Parameters which are of type array (of any data type) can be declared as **params**
2. If parameter is declared as **params** either a reference to the array can be passed as argument, 0 or more individual values can be passed to it.
3. Only one parameter of a method can be declared as params parameter.
4. It must be last parameter in the list of parameters for a given method.
5. If we have other parameters in the list they must be before the params parameter.

```
using System;
class Program
{
   public static void Main()
   {
      int res = Add(10, 2);
      int[] mar = { 1, 2, 3, 4 };
      res = Add(mar);
      Console.WriteLine(res.ToString());
      res = Add();
      res = Add(1, 2, 3, 4, 5);
      res = Add(1, 2, 3, 4, 5, 6);
      res = Add(1, 2, 3, 4, 5, 6, 7);
      Console.WriteLine(res.ToString());
   }

   static int Add(params int[] ar)
   {
      int sum = 0;
      foreach (int n in ar)
         sum += n;
      return sum;
   }
}
```

Here is the output of executing above code:



```
10
28
Press any key to continue . . .
```

Passing parameters by value, out or reference

```
using System;
class Program
{
   public static void Main()
   {
      int n1, n2, n3;
      n1 = n3 = 10;
      Foo(n1, out n2, ref n3);
      Console.WriteLine(n1 + " " + n2 + " " + n3);
   }
   static void Foo(int a, out int b, ref int c)
   {
      a++;
      b = 20;
      c++;
   }
}
```

The argument “n2” is **passed by reference** to “b”, i.e. both “b” and “n2” reference to same memory and hence change made to “b” is also reflected in “n2”.

Out parameter must be initialized in the method and are generally used in situations where we want to return more than one value from the method.

Return Type Rule

If a method has return type anything other than “void”, all code paths in it **must** return a value.

The example below compilation error because if a==0 nothing is mentioned as return value:

```
static int Foo(int a)
{
        if (a != 0)

              return a;

}
```