

Chapter-2

The C# Language Basic

IDENTIFIERS AND KEYWORDS

Identifiers:

- ☞ Identifiers are the name given to entities such as variables, methods, classes, etc. They are tokens in a program which uniquely identify an element.

Rules for naming an identifier are as follows:

- An identifier cannot be a C# keyword.
- An identifier must begin with a letter, an underscore or @ symbol. The remaining part of identifier can contain letters, digits and underscore symbol.
- Whitespaces are not allowed. Neither it can have symbols other than letter, digits and underscore.
- Identifiers are case-sensitive. So, **getName**, **GetName** and **getname** represents 3 different identifiers.

- ☞ Here some example of valid and invalid identifiers are:

Identifiers	Remarks
Number	Valid
calculateMarks	Valid
hello\$	Invalid (Contains \$)
name1	Valid
@if	Valid (Keyword with prefix @)
If	Invalid (C# Keyword)
My name	Invalid (Contains whitespace)
_hello_hi	Valid

Keywords:

- ☞ Keywords are predefined sets of reserved words that have special meaning in a program. The meaning of keywords cannot be changed, neither can they be directly used as identifiers in a program.
- ☞ C# has a total of 79 keywords. All these keywords are in lowercase.
- ☞ Some important keywords are listed below:

abstract
as
base
bool
break
byte

case
catch
char
checked
class
const
continue
decimal
default
delegate
do
double
else
enum
event
explicit
extern
false
finally
fixed
float
for
foreach
goto
if
implicit
in
int
interface
internal
is
lock
long
namespace
new
null
object
operator
out
override
params

private
protected
public
readonly
ref
return
sbyte
sealed
short
sizeof
stackalloc
static
string
struct
switch
this
throw
true
try
typeof
uint
ulong
unchecked
unsafe
ushort
using
using
void
volatile
while

Contextual Keywords

- ☞ Besides regular keywords, C# has 25 contextual keywords. Contextual keywords have specific meaning in a limited program context and can be used as identifiers outside that context. They are not reserved words in C#.
- ☞ Some contextual keywords are as follows:

add
alias
ascending
async

await
descending
dynamic
from
get
global
group
into
join
let
orderby
partial
remove
select
set
var
when
yield

Variable

- ☞ A variable is a symbolic name given to a memory location. Variables are used to store data in a computer program.
- ☞ Variables in C# must be declared before they can be used. This means, the name and type of variable must be known before they can be assigned a value. This is why C# is called a ***statically-typed language***.

Rules for Naming Variables in C#

- ☞ The rules for naming a variable in C# are:
 - I. The variable name can contain letters (uppercase and lowercase), underscore(_) and digits only.
 - II. The variable name must start with either letter, underscore or @ symbol.
 - III. C# is case sensitive. It means **age** and **Age** refers to 2 different variables.
 - IV. A variable name must not be a C# keyword. For example, **if**, **for**, **using** cannot be a variable name.

Data Types in C#

- ☞ Variables in C# are broadly classified into two types: **Value types** and **Reference types**.

Value Types:

Boolean (bool): It is used to check true or false value. It's default value is false.

Signed Integral: These data types hold integer values (both positive and negative). Out of the total available bits, one bit is used for sign. Its default value is zero.

Short: It is 16 bit data type. Default value is zero.

Int: It is 32 bit data type. Default value is zero.

Long: It is 64 bit data type. Default value is 0L, where L represents the value is of long type.

Unsigned Integral: These data types only hold values equal to or greater than 0. We generally use these data types to store values when we are sure, we won't have negative values.

Some example of Unsigned Integrals are:

- ❖ **Byte:** Its size is 8 bit. It ranges 0 to 255. The default value is zero.
- ❖ **Ushort:** Its size is 16 bit. It ranges 0 to 65,535. Its default value is zero.
- ❖ **Unit:** Its size is 32 bits
- ❖ **Ulong:** Its size is 64 bits

Floating Point: These data types hold floating point values i.e. numbers containing decimal values. For example, 12.36, -92.17, etc.

- ❖ **Float:** This is Single-precision floating point type. Its size is 32 bits. It ranges from 1.5×10^{-45} to 3.4×10^{38} . The default value is 0.0F.
- ❖ **Double:** This is double precision floating point type. Its size is 64 bits. It ranges from 5.0×10^{-324} to 1.7×10^{308} . The default value is 0.0D.

Character: It represents 16 bit Unicode character. Its size is 16 bits. The default value is '\0'.

Decimal: Decimal type has more precision and a smaller range as compared to floating point types (double and float). So it is appropriate for monetary calculations. Its size is 128 bits. It ranges. The default value is 0.0M.

Literals

- ☞ Literals are fixed values that appear in the program. They do not require any computation. For example, 10, false, 'w' are literals that appear in a program directly without any computation.

Punctuators

- ☞ Punctuators are used in C# as special symbols to a group or divide the code. It includes] () { , ; * = #.
- For example: class Demo{ }

Comments in C#

- ☞ Generally there are two comments line available in C#: Single line comment and multi-line comments:

- ❖ **Single-line comment:** A single line comment begins with a double forward slash and continue until the end of the line.

For example:

int x = 5; //assign value 5 to x.

❖ Multi-line Comment:

☞ Multiline comments start with /* and end with */.

Escape Sequence Characters

☞ C# also supports escape sequence characters such as:

Character	Meaning
\'	Single quote
\"	Double quote
\\	Backslash
\n	Newline
\r	Carriage return
\t	Horizontal Tab
\a	Alert
\b	Backspace

Operators in C#

☞ Operator is a symbol which perform operation between two or more operand. Operands may be variable or constants.

☞ Some C# operators are as follows:

1. Arithmetic Operator
2. Relational Operator
3. Logical Operator
4. Bitwise Operator
5. Assignment Operator
6. Miscellaneous Operator

1. Arithmetic Operator

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

Operator	Description	Example
+	Adds two operands	A + B =30
-	Subtract second operand from the first	A - B = -10
*	Multiplies both operands	A * B=200
/	Divides numerator by de-numerator	B /A=2

%	Modulus operator and remainder of after an integer division	B%A=0
++	Increment operator increases integer value by one	A++=11
--	Decrement operator decreases integer value by one	A-- =9

2. Relational Operator

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

Operator	Description	Example
=	Equal to operator	(A = B) is not true
!=	Not equal to operator	(A! = B) is true
>	Greater than operator	(A > B) is not true
<	Less than operator	(A < B) is true
>=	Greater than and equal to operator	(A > = B) is not true
<=	Less than and equal to operator	(A < =B) is true

3. Logical Operator

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

Operator	Descriptions	Example
&&	Logical AND operator	(A && B) is false
 	Logical OR operator	(A B) is true
!	Logical NOT operator	!(A && B) is true

4. Bitwise Operator

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

P	Q	P & Q	P Q	P ^ Q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

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

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

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

Operator	Description	Example
&	Binary AND operator	(A & B) =12
 	Binary OR operator	(A B) =61
^	Binary XOR operator	(A ^ B) =49
~	Binary ones complement operator	(~A) = 61
<<	Binary left shift operator	(A << 2) = 240
>>	Binary right shift operator	(A >> 2) = 15

5. Assignment Operator

☞ There are following assignment operators supported by C#:

Operator	Descriptions	Example
=	Simple assignment operator	C = A + B assigns value of A+B into C
+=	ADD and assignment operator	C+= A is equivalent to C = C+A

-=	Subtract and assignment operator	C-= A is equivalent to C = C-A
=	Multiplication and assignment operator	C= A is equivalent to C = C*A
/=	Division and assignment operator	C/= A is equivalent to C = C/A
%=	Modulus division and assignment operator	C%= A is equivalent to C = C%A

6. Miscellaneous Operator

☞ There are few other important operators including **sizeof**, **typeof** and **? :** supported by C#.

Operator	Description	Example
Sizeof()	Returns the size of data type	Sizeof(int), returns 4
typeof()	Returns type of class	typeof(StreamReader);
&	Returns address of variable	&a; returns actual address of variable a.
? :	Conditional expression	If condition is true? then value x: otherwise value y

Structure of C# program

1. Importing namespace
2. Declaring class
3. Main method

Write a C# program to print the statement "Hello World"?

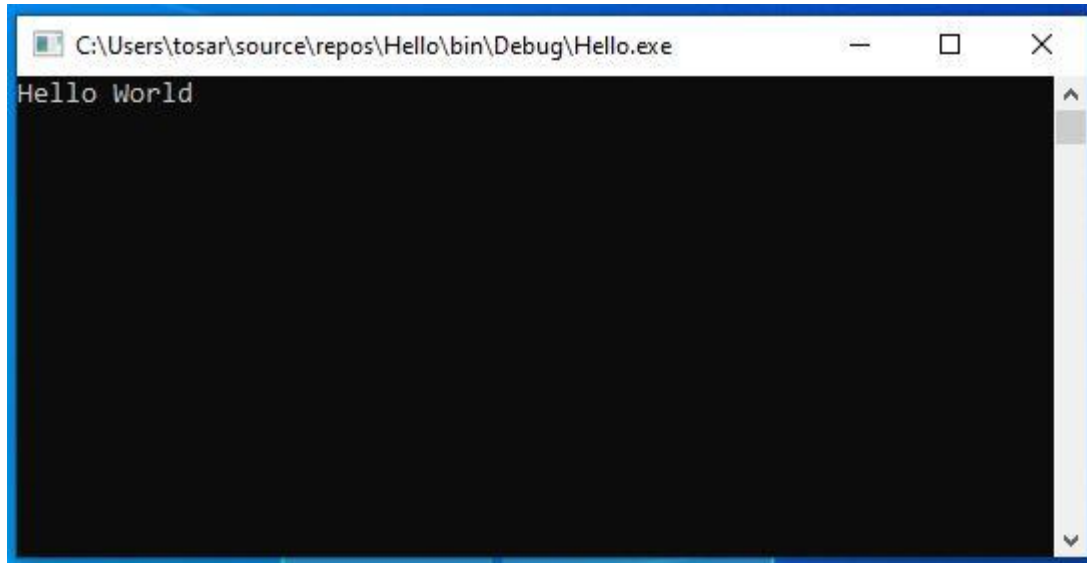
Code:

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

```
namespace Hello
{
    internal class Program
    {
        static void Main(string[] args)
        {
```

```
        Console.WriteLine("Hello World");  
        Console.ReadLine();  
    }  
}  
}
```

Out Put:



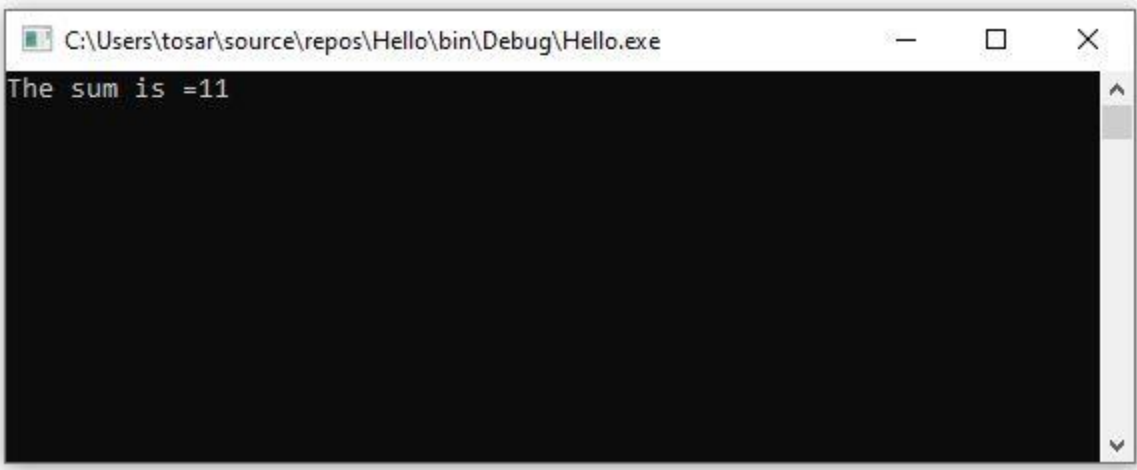
Write a C# program to find the sum of two numbers?

Code:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace Hello  
{  
    internal class Program  
    {  
        static void Main(string[] args)  
        {  
            int a = 5, b = 6, sum;  
            sum = a + b;  
            Console.WriteLine("The sum is =" + sum);  
            Console.ReadLine();  
        }  
    }  
}
```

}

Out Put:

A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Users\tosar\source\repos\Hello\bin\Debug\Hello.exe". The window has standard Windows window controls (minimize, maximize, close) on the right. The main area of the window is black with white text. The first line of text is "The sum is =11". There is a small cursor at the end of the line.

Statement and expression

- ☞ The actions that a program takes are expressed in statements. Common actions include declaring variables, assigning values, calling methods, looping through collections, and branching to one or another block of code, depending on a given condition.
- ☞ The order in which statements are executed in a program is called the flow of control or flow of execution. The flow of control may vary every time that a program is run, depending on how the program reacts to input that it receives at run time.
- ☞ A statement can consist of a single line of code that ends in a semicolon, or a series of single-line statements in a block. A statement block is enclosed in {} brackets and can contain nested blocks. The following code shows two examples of single-line statements, and a multi-line statement block:

Types of Statements

Declaration statements:

- ☞ A declaration statement introduces a new variable or constant. A variable declaration can optionally assign a value to the variable. In a constant declaration, the assignment is required.

e.g int a;
 a=5;

Local Variable:

- ☞ The scope of a local variable or local constant extends throughout the current block. You cannot declare another local variable with the same name in the current block or in any nested blocks.
For example:

```

Static void main()
{
    int x;
    {
        int y;
        int x;           // Error: x already defined
    }
    {
        int y;           // ok : y not in scope
    }
    Console.Write(y);    //Error: y is out of scope
}

```

Expression statements:

- ☞ Expression statements that calculate a value must store the value in a variable.
- ☞ Changing state essentially means changing a variable. The possible expression statements are:

- ❖ Assignment expression (including increment and decrement expression)
- ❖ Method call expressions (both void and non-void)
- ❖ Object instantiation expressions

e.g. int c;

c= a*a; //Declare variables with declaration statements

System.Text.StringBuilder sb; //Expression statements

X=5+6 // Assignment expression

x++; // Increment expression

new StringBuilding(); // Object instantiation expression

Control Statement

- ☞ Control statement is used to regulate the flow of program.
- ☞ C# program control statement can be put into the following categories:
 - I. Selection Control Statement
 - II. Iteration or Loop Control Statement
 - III. Jump Control Statement

Selection Control Statement

- ☞ Selection control statement is used to make decision in a program. It works on the basis of two binary values i.e. Zero and One, where zero means 'No' and One means 'Yes'.
- ☞ The different types of control statements used in C# are:
if Statement:
- ☞ When the given condition is true it print the statement otherwise it skips the statement.

Syntax:

if(condition)

Statement;

Example:

Write a C# program to check equality of two numbers?

Using System;

Class Program

```
{  
    Static void Main(String Args[])  
{  
    int a, b;  
    Console.WriteLine("Enter the value of a=");  
    a=Conver.ToInt32(Console.ReadLine());  
    Console.WriteLine("Enter the value of b=");  
    b=Conver.ToInt32(Console.ReadLine());  
    if(a==b)  
        Console.WriteLine(" Both are equal");  
    Console.Readkey();  
}  
}
```

```
}
```

Out Put:

if-else Statement

- ☞ When the given condition is true if part will be execute otherwise else part will be execute.

Syntax:

```
if(condition)
{
    Statements-1;
}
else
{
    Statement-2;
}
```

Example program to demonstrate if statement using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Project1

```
{
```

```
    internal class Program
```

```
    {
```

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

```
        {
```

```
            int a = 5, b = 10;
```

```
            if(a==b)
```

```
            {
```

```
                Console.WriteLine("Both are equal");
```

```
            }
```

```
            else
```

```
            {
```

```
                Console.WriteLine("Both are unequal");
```

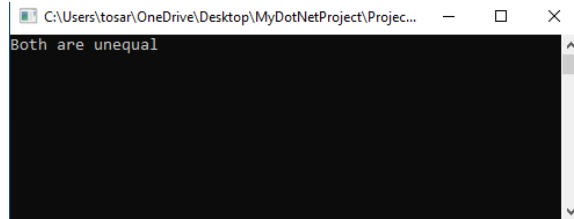
```
            }
```

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

```
    }
```

```
}  
}
```

Out Put:



Nested if-else Statement

☞ A if-else within another if-else is known as nested if-else statement.

Syntax:

```
if(condition-1)  
{  
    if(condition-2)  
    {  
        Statement-1;  
    }  
    else  
    {  
        Statement-2;  
    }  
}  
else  
{  
    if(condition-3)  
    {  
        Statement-3;  
    }  
    else  
    {  
        Statement-4;  
    }  
}
```

Example Program to demonstrate the nested if-else statement.

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

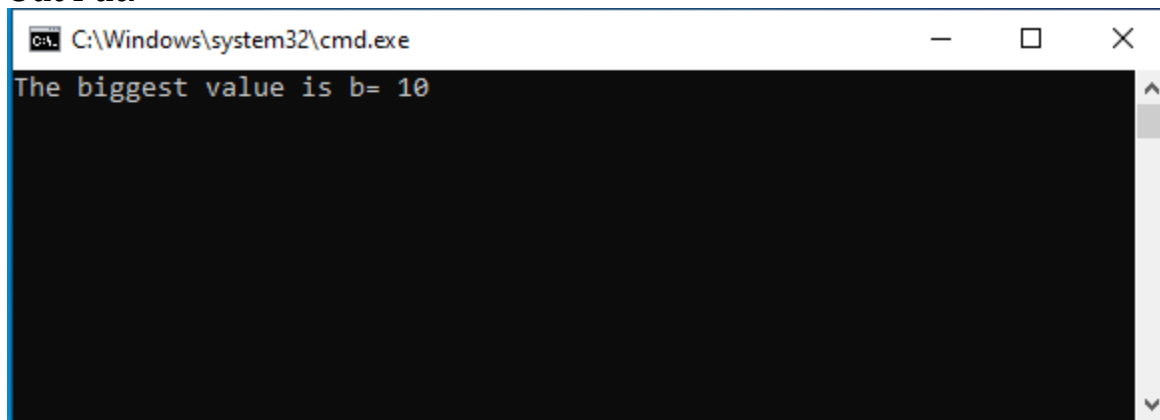
```
namespace Project2  
{
```

```

internal class Program
{
    static void Main(string[] args)
    {
        int a=5, b=10 , c=4;
        if(a>b)
        {
            if(a>c)
            {
                Console.WriteLine("The biggest value is a= {0}", a);
            }
            else
            {
                Console.WriteLine("The biggest value is c= {0}", c);
            }
        }
        else
        {
            if(b>c)
            {
                Console.WriteLine("The biggest value is b= {0}", b);
            }
            else
            {
                Console.WriteLine("The biggest value is c= {0}", c);
            }
        }
        Console.ReadLine();
    }
}

```

Out Put:



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The output displayed is "The biggest value is b= 10". The window has a standard Windows title bar with minimize, maximize, and close buttons.

The if-else-if Ladder (Compound if-else) Statement

- ☞ Compound if-else statement print only one statement at a time when more than one condition is given.
- ☞ A common programming construct that is based upon a sequence of nested **ifs** is the if-else-if ladder.

Syntax:

```
if(condition-1)
{
    Statement-1;
}
else if(condition-2)
{
    Statement-2;
}
else if(condition-3)
{
    Statement-3;
}
.
.
.
.
else
{
    Statement;
}
```

Example program to demonstrate the compound if- else statement in C#

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

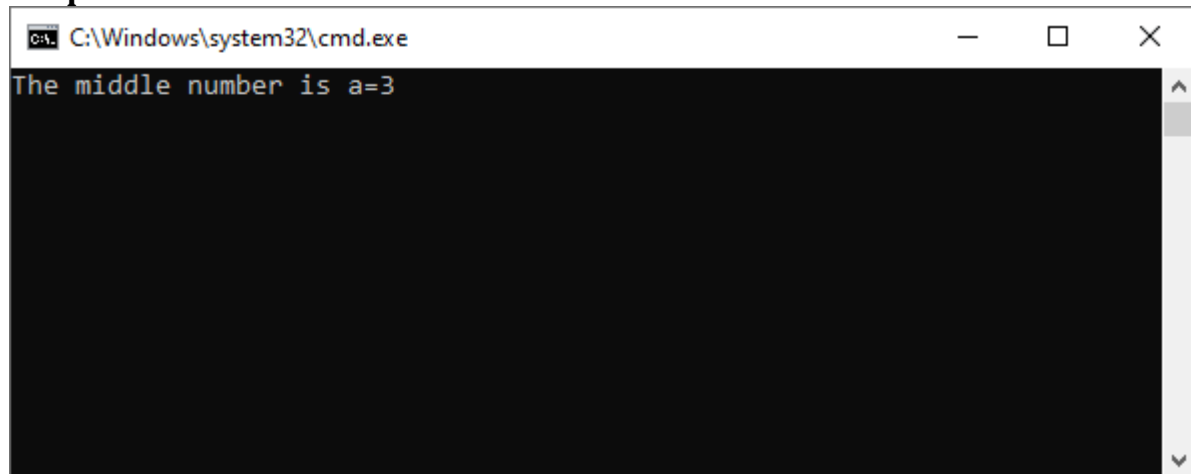
namespace Project2
{
    internal class Program
    {
        static void Main(string[] args)
        {
            int a = 3, b = 2, c = 5;
            if ((a > b && a < c) || (a < b && a > c))
            {
                Console.WriteLine("The middle number is a={0}", a);
            }
            else if ((b > c && b < a) || (b < c && b > a))
            {
                Console.WriteLine("The middle number is b={0}", b);
            }
            else if((c>a&&c<b)|| (c<a&& c>b))
            {
            }
        }
    }
}
```

```

        Console.WriteLine("The middle number is c={0}", c);
    }
    else
    {
        Console.WriteLine("Case doesnot match");
    }
    Console.ReadLine();
}
}
}

```

Output:



Switch Case

- ☞ Case control statement is alternative to compound if-else statement but only difference is that compound if-else statement print the statement by checking its condition whereas case control statement print the statement by matching its label.

Syntax:

```

switch(expression)
{
    case label-1:
        Statement-1;
        break;
    case label-2:
        Statement-2;
        break;
    case label-3:
        Statement-3;
        break;
    .
    .
}

```

```
.  
.   
.   
case label-n:  
    Statement-n;  
    break;
```

```
default:  
    statement;  
    break;
```

```
}
```

Example program to demonstrate switch case statement:

Write a C# program to print name of week?

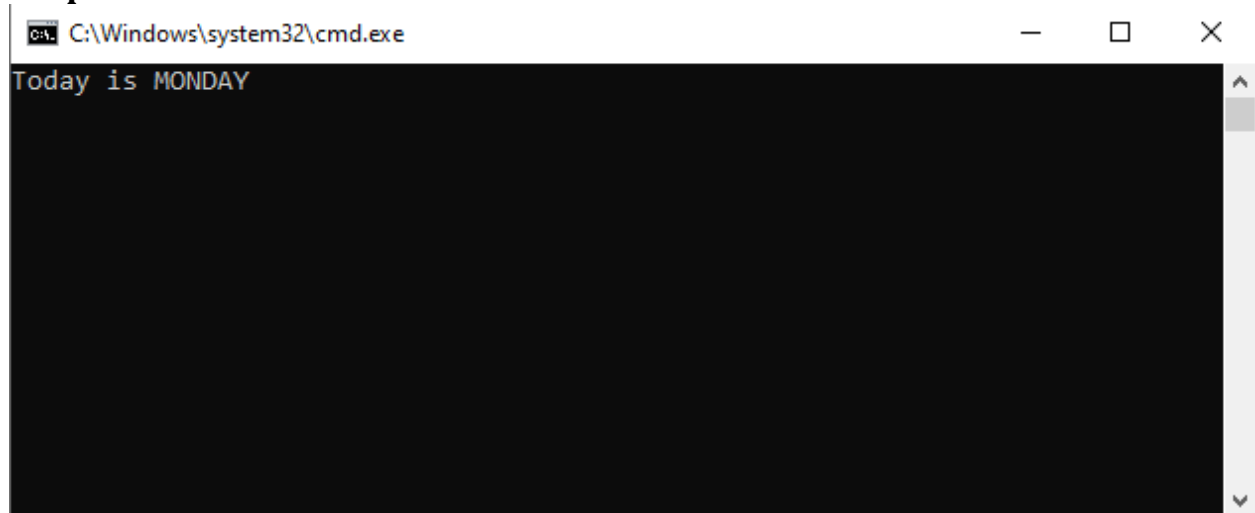
```
using System;  
namespace Project2  
{  
    internal class Program  
    {  
        static void Main(string[] args)  
        {  
            int day = 2;  
            switch(day)  
            {  
                case 1:  
                {  
                    Console.WriteLine("Today is SUNDAY");  
                    break;  
                }  
                case 2:  
                {  
                    Console.WriteLine("Today is MONDAY");  
                    break;  
                }  
                case 3:  
                {  
                    Console.WriteLine("Today is TUESDAY");  
                    break;  
                }  
                case 4:  
                {  
                    Console.WriteLine("Today is WEDNESDAY");  
                    break;  
                }  
                case 5:  
                {  
                    Console.WriteLine("Today is THURSDAY");  
                    break;  
                }  
                case 6:  
                {
```

```

        Console.WriteLine("Today is FRIDAY");
        break;
    }
    case 7:
    {
        Console.WriteLine("Today is SATURDAY");
        break;
    }
    default:
    {
        Console.WriteLine("Case doesnot match");
        break;
    }
}
Console.ReadLine();
}
}
}

```

Output:



Iteration Statement

- ☞ When one statement requires number of times then it is said to be a loop or iteration.
- ☞ A loop repeatedly executes the same set of instructions until a termination is met.
- ☞ A loop statement allows us to execute a statement or group of statements multiple times.

Loop control variable:

- ☞ A variable which controls the flow of loop is known as loop control variable.
- ☞ Basically, there are three loop control variable
 - ❖ Initialization
 - ❖ Condition
 - ❖ Updation (Increment/Decrement)

☞ The different types of iteration control statements are:

❖ **while-loop**

☞ while loop print the statement after checking its condition.

☞ A while loop statement in C# programming language repeatedly executes a target statement as long as a given condition is true.

Syntax:

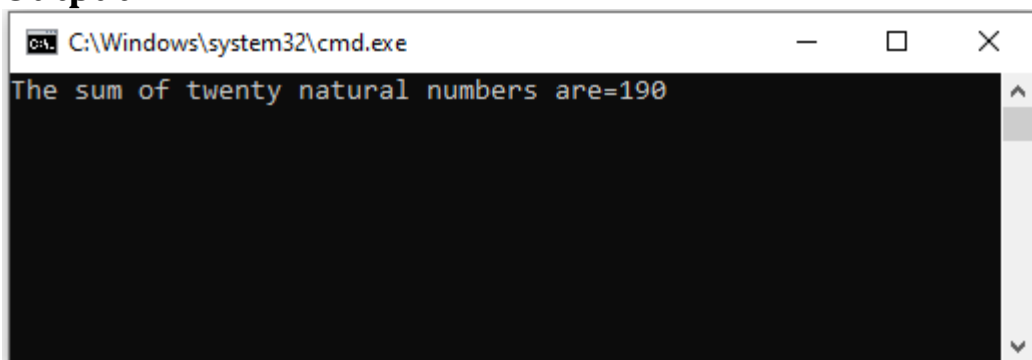
```
while(condition)
{
    Statements;
}
```

Example program to demonstrate while loop

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

namespace Project2
{
    internal class Program
    {
        static void Main(string[] args)
        {
            int i = 1, sum = 0;
            while(i<20)
            {
                sum = sum + i;
                i++;
            }
            Console.WriteLine("The sum of twenty natural numbers are={0}", sum);
            Console.ReadLine();
        }
    }
}
```

Output:

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\system32\cmd.exe'. The command prompt has a black background with white text. The first line of output is 'The sum of twenty natural numbers are=190'. There is a vertical scrollbar on the right side of the window.

```
C:\Windows\system32\cmd.exe
The sum of twenty natural numbers are=190
```

❖ do-while Loop

☞ do-while loop prints the statement at least one before checking its condition.

Syntax:

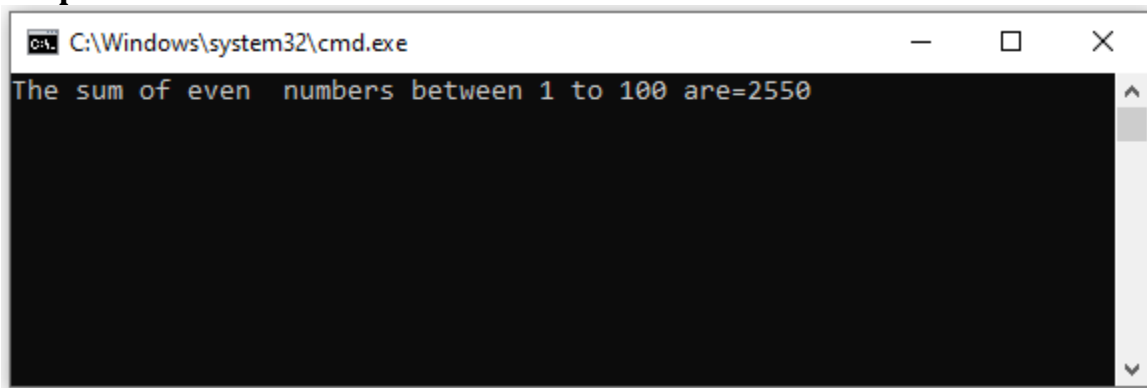
```
do
{
    Statements;
}while(condition);
```

Example program to demonstrate do-while loop

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

namespace Project2
{
    internal class Program
    {
        static void Main(string[] args)
        {
            int i = 2, sum = 0;
            do
            {
                sum = sum + i;
                i = i + 2;
            } while (i <= 100);
            Console.WriteLine("The sum of even numbers between 1 to 100 are={0}",
sum);
            Console.ReadLine();
        }
    }
}
```

Output:

A screenshot of a Windows Command Prompt window. The title bar shows 'C:\Windows\system32\cmd.exe'. The command prompt displays the output of the program: 'The sum of even numbers between 1 to 100 are=2550'. The text is in a monospaced font, with the output line in green. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

❖ for Loop

- ☞ A for loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.
- ☞ A for loop is useful when you know how many times a task is to be repeated.

Syntax:

```
for(exp-1(initializer); exp-2(condition);exp-3 (update))
{
    Statements;
}
```

Example:

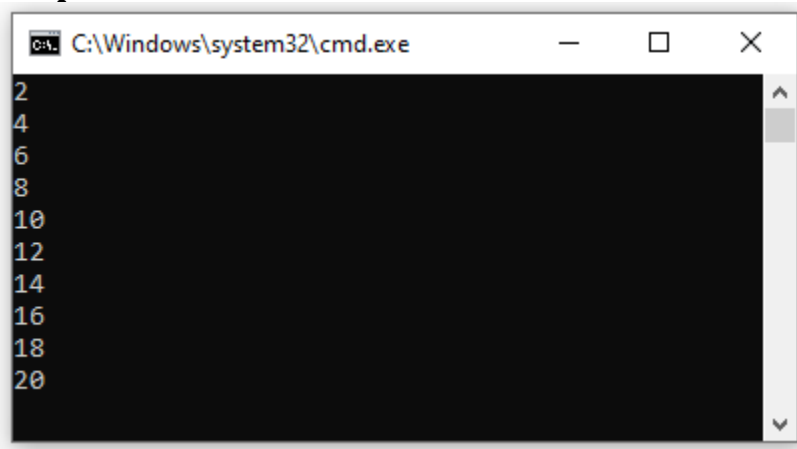
Write a C# program to print the multiplication table?

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

namespace Project2
{
    internal class Program
    {
        static void Main(string[] args)
        {
            int n = 2,r;
            Console.WriteLine("The multiplication table is:");
            for(int i=1; i<=10; i++)
            {
                r = n * i;
                Console.WriteLine(r);
            }

            Console.Read();
        }
    }
}
```

Output:



```
C:\Windows\system32\cmd.exe
2
4
6
8
10
12
14
16
18
20
```

foreach Loop

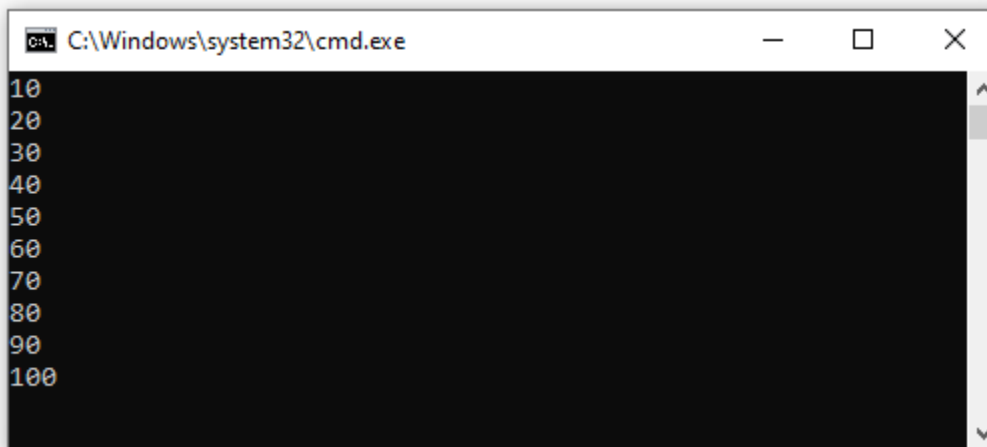
- ☞ The for each statement iterates over each element in an enumerable object. Most of the types in C# and the .Net Framework that represent a set or list of elements are enumerable.

For example:

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

namespace Project2
{
    internal class Program
    {
        static void Main(string[] args)
        {
            int[] a = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };
            foreach (int val in a)
            {
                Console.WriteLine(val);
            }
            Console.ReadLine();
        }
    }
}
```

Output:



```
C:\Windows\system32\cmd.exe
10
20
30
40
50
60
70
80
90
100
```

Nested for Loop

- ☞ A for loop within another for loop is known as nested for Loop. For example:

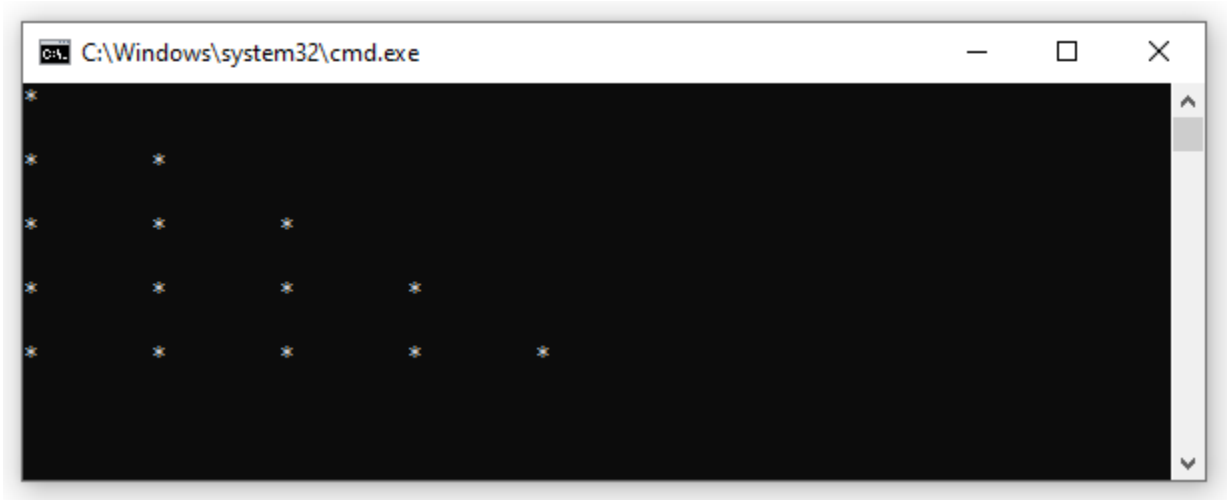

```

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

namespace Project1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            int i, j;
            for(i=1; i<=5; i++)
            {
                for(j=1; j<=i; j++)
                {
                    Console.Write("*\t");
                }
                Console.WriteLine("\n");
            }
            Console.ReadKey();
        }
    }
}

```

Output:



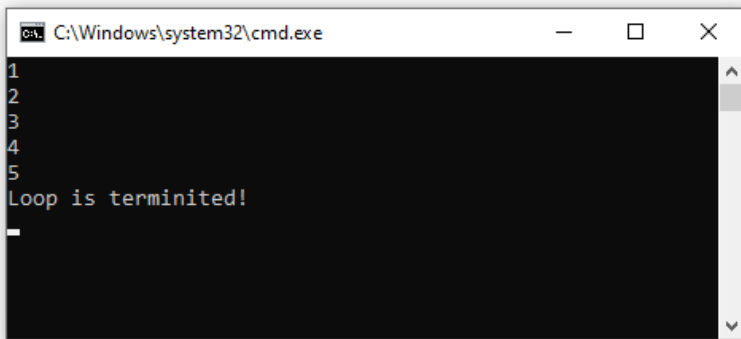
Jump Statement

- ☞ In C# jump statements are **break, continue, goto, return and throw**.
- ☞ These statements transfer control to another part of your program.

❖ Using **break**

- ☞ It is used to terminate the program.
- ☞ It can be also used to exit a loop.

Example program to demonstrate a loop.



```
1
2
3
4
5
Loop is terminated!
```

❖ Using **continue**

- ☞ The continue statement breaks one iteration if a specified condition occurs and continues with the next iteration in the loop.
- ☞ In a while loop and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop.
- ☞ In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression.
- ☞ For all three loop, any intermediate code is bypassed.

Example:

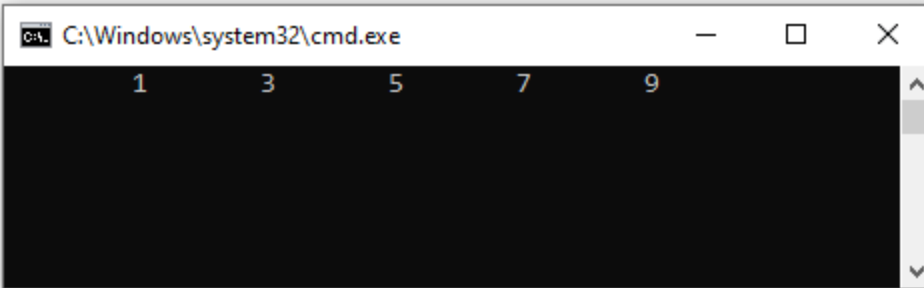
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Project1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            int i;
            for(i=1;i<=10;i++)
            {
                if (i % 2==0)
                    continue;
                Console.Write("\t" + i);

            }

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

Output:



❖ Using **return**

- ☞ The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

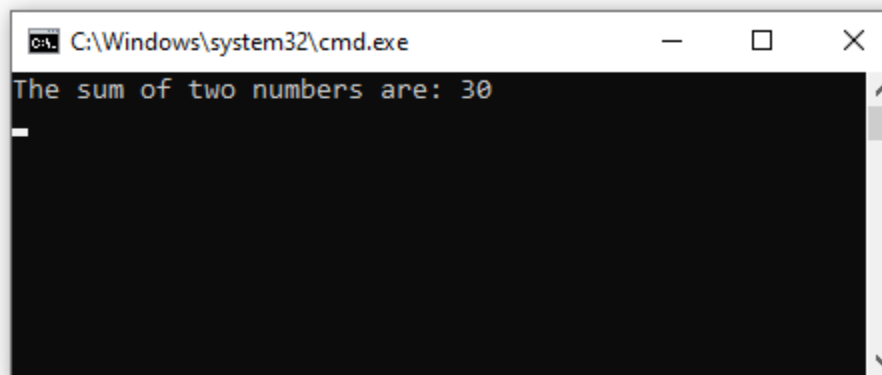
Example:

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

namespace Project1
{
    class Program
    {
        static int Addition()
        {
            int a = 10, b = 20, sum;
            sum = a + b;
            return sum;
        }
        static void Main(string[] args)
        {
            int result;
            result = Addition();
            Console.WriteLine("The sum of two numbers are: {0}", result);

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

Output:



```
C:\Windows\system32\cmd.exe
The sum of two numbers are: 30
```

❖ Using **goto**

- ☞ The go to statement transfers execution to another label within a statement block.
- ☞ The form is as follows:
goto statement-label;
OR
goto case case-constant;

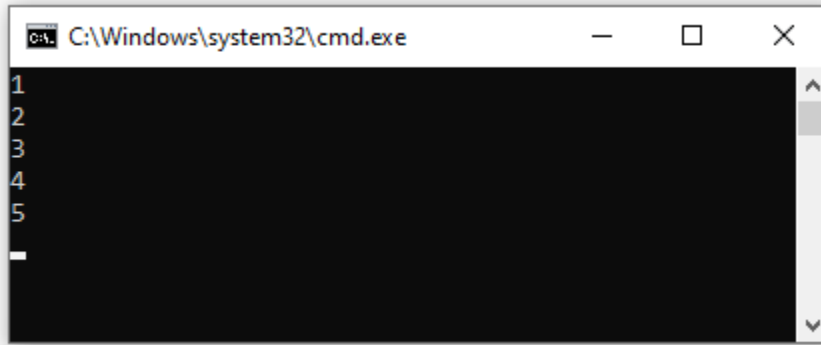
Example:

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

namespace Project1
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 1;
            StartLoop:
            if(i<=5)
            {
                Console.WriteLine(i + " ");
                i++;
                goto StartLoop;
            }

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

Output:



❖ The **throw** statement

- ☞ The throw statement throws an exception to indicate an error has occurred.

Example:

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

namespace Project1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            int age = 18;
            if (age < 18)
                throw new ArithmeticException("Not Eligible to Vote");
            Console.ReadKey();
        }
    }
}
```

ARRAYS

- ☞ An array is consecutive group of memory location that all have same name and same type.
- ☞ An array is denoted with square brackets after the element type.

Array declaration:

```
int[ ] a=new int[5];
char[] c=new char[10];
```

Square bracket is used to index an array, accessing a particular element by position:

```
vowels[0] = 'a';  
vowels[1] = 'e';  
vowels[2] = 'i';  
vowels[3] = 'o';  
vowels[4] = 'u';
```

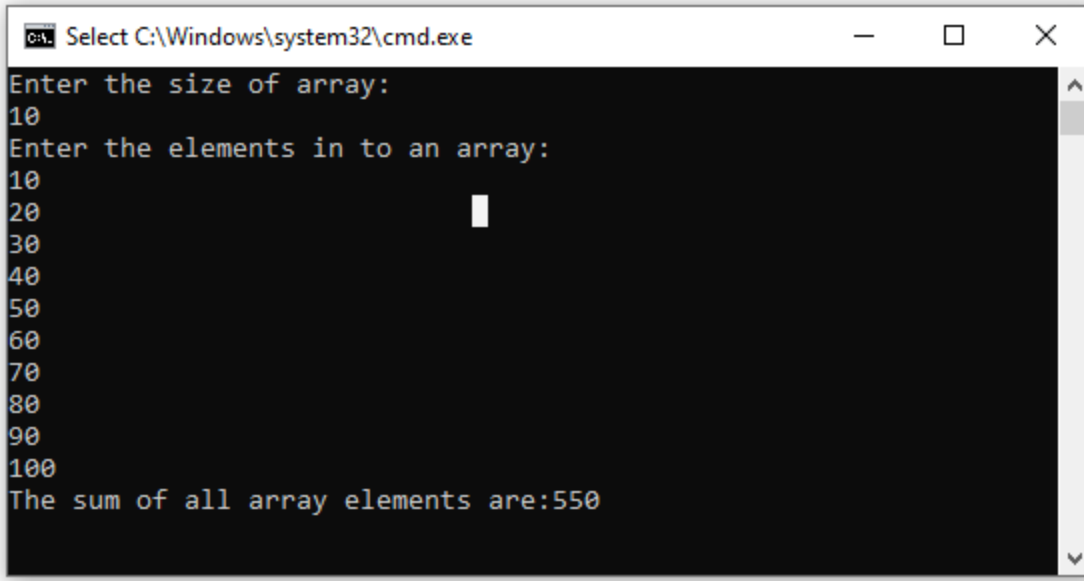
```
Console.WriteLine(vowels[1]);
```

It will print 'e'.

Example:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace Project1  
{  
    internal class Program  
    {  
  
        static void Main(string[] args)  
        {  
            int[] a = new int[10];  
            int n, i, sum = 0;  
            Console.WriteLine("Enter the size of array:");  
            n = Convert.ToInt32(Console.ReadLine());  
            Console.WriteLine("Enter the elements in to an array:");  
            for (i = 0; i < n; i++)  
                a[i] = Convert.ToInt32(Console.ReadLine());  
            for (i = 0; i < n; i++)  
                sum = sum + a[i];  
            Console.WriteLine("The sum of all array elements are:" + sum);  
  
            Console.ReadKey();  
        }  
    }  
}
```

Output:



```
C:\Windows\system32\cmd.exe
Enter the size of array:
10
Enter the elements in to an array:
10
20
30
40
50
60
70
80
90
100
The sum of all array elements are:550
```

Multi-dimensional Array:

- ☞ When we want to arrange more than one rows or column then such arrangement is not done by one dimensional array so we need multidimensional array.
- ☞ Multidimensional arrays come into two varieties:

- ❖ Rectangular array
- ❖ Jagged array

Rectangular array

- ☞ Rectangular arrays are declared using commas to separate each dimension.
- ☞ The following declares a rectangular two dimensional array, where the dimensions are 3 by 3.

Syntax:

```
int[ , ] matrix = new int[3 , 3];
```

A rectangle array can be initialized as follows:

```
int[ , ]matrix = new int[ , ]
{
    {0, 1, 2}
    {3, 4, 5}
    {6, 7, 8}
```

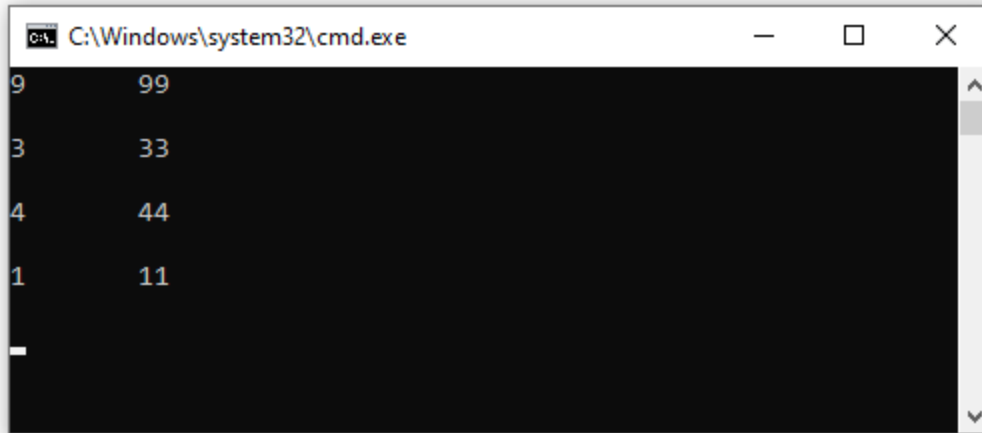
```
};
```

Example program to demonstrate rectangular arrays

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

namespace Project1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            int[,] vals = new int[4, 2]
            {
                {9, 99 },
                {3, 33 },
                {4, 44 },
                {1, 11 }
            };
            for(int i=0; i<4;i++)
            {
                for(int j=0;j<2;j++)
                {
                    Console.Write(vals[i, j]);
                    Console.Write("\t");
                }
                Console.WriteLine("\n");
            }
            /*
            foreach (var val in vals)
            {
                Console.Write(val);
            }
            */
            Console.ReadKey();
        }
    }
}
```

Output:



```
C:\Windows\system32\cmd.exe
9      99
3      33
4      44
1      11
_
```

Jagged arrays

- ☞ Jagged arrays are declared using successive square brackets to represent each dimension. Here is an example of declaring a jagged two-dimensional array, where the outermost dimension is 3:

```
int[ ] matrix = new int[3][ ];
```

- ☞ The inner dimensions aren't specified in the declaration because, unlike a rectangular array, each inner array can be an arbitrary length. Each inner array is implicitly initialized to null rather than an empty array.
- ☞ A jagged array can be initialized as follows:

```
int[ ] [ ] matrix = new int[ ] [ ]
{
    new int[ ] {0, 1, 2 },
    new int [ ] {3, 4, 5 },
    new int [ ] {6, 7, 8, 9}
};
```

Example program to demonstrate jagged array

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

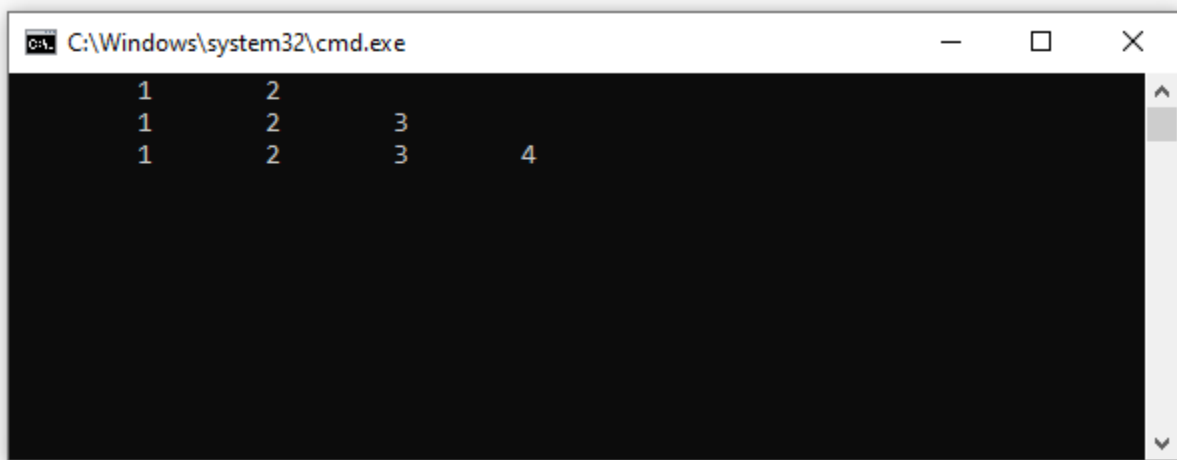
namespace Project1
{
    internal class Program
    {
        static void Main(string[] args)
        {
```

```

int[][] jagged = new int[][]
{
    new int[] {1,2},
    new int[] {1, 2, 3},
    new int[] {1, 2, 3, 4}
};
foreach (int[] array in jagged)
{
    foreach (int e in array)
    {
        Console.Write("\t" + e);

    }
    Console.WriteLine();
}
Console.ReadKey();
}
}

```



Namespaces

- ☞ Namespaces are used in C# to organize and provide a level of separation of codes.
- ☞ They can be considered as a container which consists of other namespaces, classes etc.
- ☞ The concept of namespace is similar in C#. It help us to organize different members by putting related members in the same namespace.
- ☞ Namespace also solves the problem of naming conflict. Two or more classes when put into different namespaces can have same name.

Defining a namespace in C#

- ☞ We can define a namespace in C# using the **namespace** keyword as:

Syntax:

namespace Namespace-Name

```
{  
    //body of namespace.  
}
```

Example:

```
using System;  
namespace MyNamespace  
{  
    class MyClass  
    {  
        public void MyMethod()  
        {  
            System.Console.WriteLine("Creating MyNamespace");  
        }  
    }  
}
```

Accessing members of namespace in C#

The using Directive

☞ The using directive imports a namespace, allowing you to refer to types without their fully qualified names.

☞ The following imports the previous example's **Outer.Middle.Inner** namespace:

```
using Outer.Middle.Inner;  
class Test  
{  
    static void Main()  
    {  
        class1 c; //Don't need fully qualified name
```

```
    }  
}
```

Using **static**

- ☞ From C# 6, you can import not just a namespace, but a specific type, with the `using static` directive.
- ☞ All static members of that type can then be used without being qualified with the type name.
- ☞ In the following example, we call the `Console` class's static `WriteLine` method:

```
using static System.Console;  
class Test  
{  
    static void Main()  
    {  
        WriteLine("Hello world");  
    }  
}
```

Rules Within a Namespace

Name scoping

- ☞ Names declared in outer namespace can be used unqualified within inner name-spaces. In this example, `Class1` does not need qualification within `Inner`:

```
namespace Outer  
{  
    class Class1  
    {  
  
    }  
    namespace Inner  
    {  
        class Class : class1  
        {  
  
        }  
    }  
}
```

Name hiding

- ☞ If the same type name appears in both an inner and an outer namespace, the inner name wins.
- ☞ To refer to the type in the outer namespace, you must qualify its name. For example:

```
namespace Outer
{
    class Foo
    {
    }
    class Test
    {
        Foo f1;
        Outer.Foo f2;
    }
}
```

Repeated namespaces

- ☞ You can repeat a namespace declaration, as long as the type names within the namespace don't conflict:

```
namespace Outer.Middle.Inner
{
    Class class1
    {
    }
}
namespace Outer.Middle.Inner
{
    class Class2
    {
    }
}
```

We can even break the example into two source files such that we could compile each class into a different assembly.

Source file 1:

```
namespace Outer.Middle.Inner
```

```

    {
        class Class1
        {
        }
    }
Source file 2:
    namespace Outer.Middle.Inner
    {
        class C lass2
        {
        }

    }

```

Nested using directive

- ☞ You can nest a using directive within a namespace. This allows you to scope the using directive within a namespace declaration. In the following example, Class1 is visible in one scope, but not in another.

```

namespace N1
{
    class Class1
    {
    }
}
namespace N2

{
    using N1;
    class Class2: class1
    {
    }
}
namespace N2

```

```
{  
    using N1;  
    class Class3: Class1  
    {  
    }  
}
```