C# Core

- C# language core, Console applications
- Variables, data types, operators, expressions
- Selection structures – if-else, switch
- Iteration structures (loops) – while, do-while, for, foreach, recursion
- Data structures – arrays, collections, lists, dictionaries, structs
- Methods, Functions
- Error handling
- Algorithms, flowcharts and decision tables

	C# Language Development environment, IDE Vusual Studio	
	Data Types	
	Sequence	
	Selection	
	Iteration	
	Methods	
	Arrays, Lists, Dictionaries	
	Project 1 Guess a Number	
	Project 2 Grade Book	
	Project 3 Hangman	

Data Types

bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
string			
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	(-7.9 x 10^{28} to 7.9 x 10^{28}) / $10^{0}$to 28	0.0M
double	64-bit double-precision floating point type	(+/-)5.0 x 10^{-324} to (+/-)1.7 x 10^{308}	0.0D
float	32-bit single-precision floating point type	-3.4 x 10^{38} to + 3.4 x 10^{38}	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0L

When dealing with floating point values, you should use a *float* or a *double* data type when precision is less important than performance.
On the other hand, if you want the maximum amount of precision and you are willing to accept a ***lower level of performance***, you should go with the decimal data type - especially when dealing with financial numbers.

```
Console.WriteLine("Size of int: {0}", sizeof(int));

         Console.ReadLine();
```

Which of these variables will compile?

```
            double num = 2;
            string ohSnap = "%$^&$ ";
            int num2 = 10.9;
            byte smallNum = -42;
            char word = 'word';
            long bigNum = 12345678.9;
            float x = 3.5F;
            decimal deciNum = 4.2m;
```

Floating Point Numbers

Floating-point numbers are numbers that have fractional parts (usually expressed with a decimal point). You might wonder why there's isn't just a single data type for dealing with numbers (fractions or no fractions), but that's **because it's a lot faster for the computer to deal with whole numbers** than with numbers containing fractions.

char

The System.Char data type is used to hold a single, unicode character. C# has an alias for it, called char

```
string helloWorld = "Hello, world!";
foreach(char c in helloWorld)
{
    Console.WriteLine(c);
}
```

strings

A *string* is a piece of text. It's usually consists of 2 characters or more, since if it's just one, you should consider using a *char* instead. However, strings can be empty as well or even null, since it's a reference type http://csharp.net-tutorials.com/data-types/strings/
In C#, strings are immutable, which basically means that once they are created, they can't be changed to a different type.

```
            string name = "John Doe";
            int age = 42;
           // string userString = String.Format("{0} is {1} years old and lives in {2}", name, age, "New York");
            Debug.WriteLine(name);
            Console.WriteLine("test");
            Console.WriteLine("What do you mean by \"Hello, world\" ??");

            string userString = String.Format("{0} is {1} years old and lives in {2}", name, age, "New York");
            Console.WriteLine(userString);
            Console.WriteLine(@"In a verbatim string \ everything is literal: \n & \t");
            int myInteger;
            string myString;
            myInteger = 17;
            myString = "\"myInteger\" is";
            Console.WriteLine("{0} {1}.", myString, myInteger);
            Console.ReadKey();
```

Date Time

```
DateTime dt = new DateTime(2042, 12, 24, 18, 42, 0);

            Console.WriteLine("Short date pattern (d): " + dt.ToString("d"));
            Console.WriteLine("Long date pattern (D): " + dt.ToString("D"));
            Console.WriteLine("Full date/time pattern (F): " + dt.ToString("F"));
            Console.WriteLine("Year/month pattern (y): " + dt.ToString("y"));
```

Nullable Types

NULL literally means nothing - a variable that doesn't yet hold a value. you always have to make sure that a variable has a value before you try to access this value. If not, you will likely be met with a NullReferenceException.

Implicit and Explicit Types

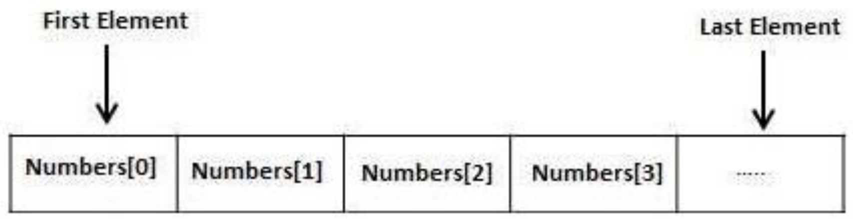
```
            int age = 42; // Explicitly typed variable

            var name = "John Doe"; // Implicitly typed variable

            Console.WriteLine(age);
            Console.WriteLine(name);
```

Arrays

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type stored at contiguous memory locations.



https://www.tutorialspoint.com/csharp/csharp_arrays.htm

```
           string[] names = new string[2];
```

```
            names[0] = "John Doe";
            names[1] = "Jane Doe";

            foreach (string s in names)
            {
                Console.WriteLine(s);
            }

            Console.ReadLine();
```

```
            int[] numbers = { 4, 3, 8, 0, 5 };

            Array.Sort(numbers);

            foreach (int i in numbers)
                Console.WriteLine(i);

            Console.ReadLine();
```

Lists

C# has a range of classes for dealing with lists. They implement the IList interface and the most popular implementation is the generic list, often referred to as List<T>. The T specifies **the type of objects contained in the list**, which has the added benefit that the compiler will check and make sure that you only add objects of the correct type to the list - in other words, the List<T> is type-safe.
List is similar to array but List is simpler and easier to work with. For **instance, you don't have to create a List with a specific size** - instead, you can just create it and .NET will automatically expand it to fit the amount of items as you add them.

```
  class Program
        {
            static void Main(string[] args)
            {
                List<User> listOfUsers = new List<User>()
            {
                new User() { Name = "John Doe", Age = 42 },
                new User() { Name = "Jane Doe", Age = 34 },
                new User() { Name = "Joe Doe", Age = 8 },
            };

                for (int i = 0; i < listOfUsers.Count; i++)
                {
                    Console.WriteLine(listOfUsers[i].Name + " is " + listOfUsers[i].Age + " years old");
                }
                Console.ReadKey();
            }
        }

        class User
        {
            public string Name { get; set; }
            public int Age { get; set; }
        }
```

Define a simple class for holding information about a User - just a name and an age. Back to the top part of the example, where I have changed our list to use this User class instead of simple strings. I use a collection initializer to populate the list with users - notice how the syntax is the same as before, just a bit more complex because we're dealing with a more complex object than a string.
Once we have the list ready, I use a *for* loop to run through it I access the user in question through the indexer of the list, using the square bracket syntax (e.g. listOfUsers[i]). Once I have the user, I output name and age.

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

        List<Money> myMoney = new List<Money>()
        {
        new Money { amount = 10, type = "US" },
        new Money { amount = 20, type = "US" },
        };


        foreach (var money in myMoney)
        {
            Console.WriteLine("Amount is {0} and type is {1}", money.amount, money.type);
        }


    }//end main

    class Money
    {
        public int amount { get; set; }
        public string type { get; set; }
    }

}//end program
```

Dictionaries

Dictionaries in C# all implement the IDictionary interface. There are several Dictionary types, but the most commonly used is the generic Dictionary, often referred to as Dictionary<TKey, TValue> - it holds a type-specific key and a corresponding type-specific value. This is basically what separates a Dictionary from a List –

The items of a list comes in a specific order and can be accessed by a numerical index, where items in a Dictionary is stored with a unique key, which can then be used to retrieve the item again.

```
        Dictionary<string, int> users = new Dictionary<string, int>();
            users.Add("John Doe", 42);
            users.Add("Jane Doe", 38);
            users.Add("Joe Doe", 12);
            users.Add("Jenna Doe", 12);

            Console.WriteLine("John Doe is " + users["John Doe"] + " years old");
```

```
 Dictionary<string, int> users = new Dictionary<string, int>()
{
    { "John Doe", 42 },
    { "Jane Doe", 38 },
    { "Joe Doe", 12 },
    { "Jenna Doe", 12 }
};

  foreach (KeyValuePair<string, int> user in users)
    {
         Console.WriteLine(user.Key + " is " + user.Value + " years old");
      }
```

Struct (Structure)

In C#, a structure is a value type data type. It helps you to make a single variable hold related data of various data types. The **struct** keyword is used for creating a structure. Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

eNum

An enumeration is a set of named integer constants. An enumerated type is declared using the **enum** keyword. C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or cannot pass inheritance.

```
    public class testenum
    {
         enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

        static void Main(string[] args)
        {
            int WeekdayStart = (int)Days.Mon;
            int WeekdayEnd = (int)Days.Fri;

            Console.WriteLine("Monday: {0}", WeekdayStart);
            Console.WriteLine("Friday: {0}", WeekdayEnd);
            Console.ReadKey();
        }

    }
```

Reference Types

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.
In other words, they refer to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Example of **built-in** reference types are: **object**, **dynamic,** and **string**.

```
static void Main(string[] args)
        {
            int number = 20;
            AddFive(ref number);
            Console.WriteLine(number);
            Console.ReadKey();
        }

        static void AddFive(ref int number)
        {
            number = number + 5;
        }
```

```
static void Main(string[] args)
        {
            int number = 20;
            AddFive(number);
            Console.WriteLine(number);
            Console.ReadKey();
        }

        static void AddFive(int number)
        {
            number = number + 5;
        }
```

Value Types

```
int
```

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, are 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.

```
double d = 5673.74;
         int i;

         // cast double to int.
         i = (int)d;
         Console.WriteLine(i);
         Console.ReadKey();
```

Sr.No.	Methods & Description
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.

```
int i = 75;
         float f = 53.005f;
         double d = 2345.7652;
         bool b = true;

         Console.WriteLine(i.ToString());
         Console.WriteLine(f.ToString());
```

```
        Console.WriteLine(d.ToString());
        Console.WriteLine(b.ToString());
        Console.ReadKey();
```

Operators

Arithmetic Operators

Operator	Description	Example
+	Adds two operands	A + B = 30
-	Subtracts second operand from the first	A - B = -10
*	Multiplies both operands	A * B = 200
/	Divides numerator by de-numerator	B / A = 2
%	Modulus Operator and remainder of after an integer division	B % A = 0
++	Increment operator increases integer value by one	A++ = 11

--	Decrement operator decreases integer value by one	A-- = 9

Relational Operators

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

Logical Operators

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

Variables, Operators, Simple Expressions

1 Create a console application displaying “Hello World”

```
  using System;
namespace SimplestProgram
    {
        class Program
        {
            static void Main(string[] args)
            {
                Console.WriteLine(“Hello, world!”);
            }
        }
    }
```

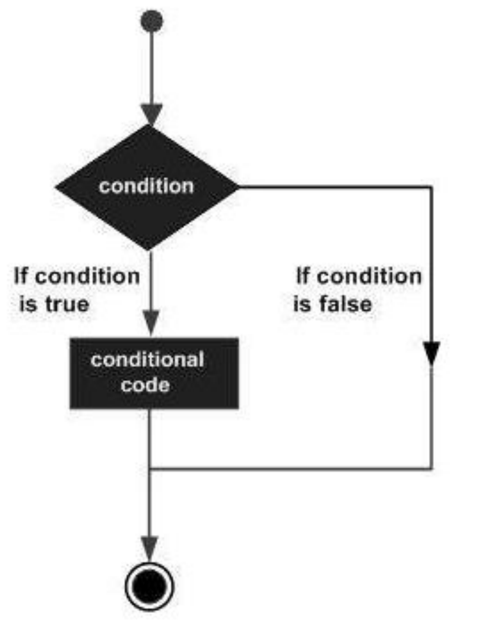
2	Collect 2 number values from the user and display the sum of those values on the console

	`Console.WriteLine("Enter first number");`
	`//When reading the program treats as a string so need to convert to int`
	`int Number1 = Convert.ToInt32(Console.ReadLine());`
	`Console.WriteLine("Enter second number");`
	`int Number2 = Convert.ToInt32(Console.ReadLine());`
	
	`//Add together`
	`int Number3 = Number2 + Number1;`
	`Console.WriteLine("{0} + {1} = {2}",Number1,Number2,Number3);`

4	Collect a single number value from the user and display on the console the number one less and the number one more than the number entered
5	Collect 2 numbers from the user and display the result of adding them together, subtracting the first number from the second, multiplying the two numbers and dividing the first number by the second.
6	Collect 4 numbers from the user and display the average of the 4 numbers
7	Write a program that converts Celsius into kelvin
8	Write a program that converts pounds to kilograms and displays the kilograms to 2 decimal places
9	Write a program to display the largest and lowest of 3 numbers entered by the user

Selection structures

Selection structures requires the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.



Common Selection Structures

if statement
An **if statement** consists of a boolean expression followed by one or more statements.

if...else statement
An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.

switch statement
A **switch** statement allows a variable to be tested for equality against a list of values.

```
Ternary Statement: Exp1 ? Exp2 : Exp3;
```

If Statement

```
  /* local variable definition */
            int a = 10;

            /* check the boolean condition using if statement */
            if (a < 20)
            {
                /* if condition is true then print the following */
                Console.WriteLine("a is less than 20");
            }
            Console.WriteLine("value of a is : {0}", a);
            Console.ReadLine();
        }
```

If Else

```
  /* local variable definition */
            int a = 100;

            /* check the boolean condition */
            if (a < 20)
            {
                /* if condition is true then print the following */
                Console.WriteLine("a is less than 20");
            }
            else
            {
                /* if condition is false then print the following */
                Console.WriteLine("a is not less than 20");
            }
            Console.WriteLine("value of a is : {0}", a);
            Console.ReadLine();
        }
```

Nested If

```
//* local variable definition */
            int a = 100;
            int b = 200;

            /* check the boolean condition */
            if (a == 100)
            {

                /* if condition is true then check the following */
                if (b == 200)
                {
                    /* if condition is true then print the following */
                    Console.WriteLine("Value of a is 100 and b is 200");
                }
            }
            Console.WriteLine("Exact value of a is : {0}", a);
            Console.WriteLine("Exact value of b is : {0}", b);
            Console.ReadLine();
```

Switch

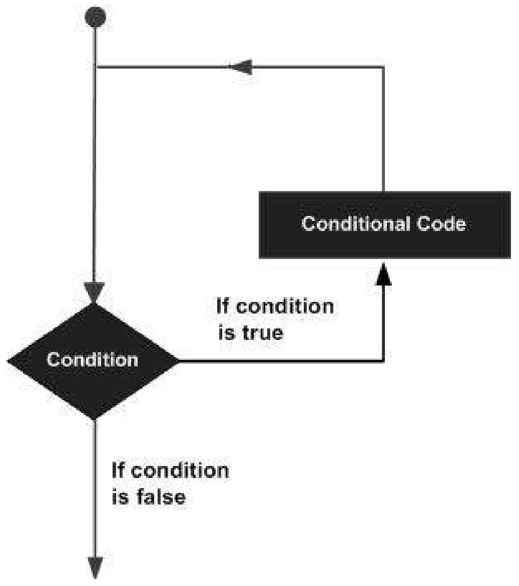
```
/* local variable definition */
            char grade = 'B';

            switch (grade)
            {
                case 'A':
                    Console.WriteLine("Your grade {0} is a Pass", grade);
                    break;
                case 'B':
                    Console.WriteLine("Your grade {0} is a Merit", grade);
```

	```
	                break;
	            case 'C':
	                Console.WriteLine("Your grade {0} is a Distinction", grade);
	                break;
	            default:
	                Console.WriteLine("Your grade {0} is an invalid grade", grade);
	                break;
	        }

	        Console.ReadLine();
	```
1	Collect 2 number values from a user and display the larger of those numbers on the console
2	Collect a number from the user and display in the console if the number entered is negative, less than 10 or greater than 10
3	Collect 2 numbers from the user and display on the console if one of them or both of them are greater than 10
4	Collect 2 number values from a user and display in text on the console whether the numbers are the same or different
5	Use a switch statement that displays the **current day of the week** and **current time** onto the console

Iterations



Iterations are used when you need to execute a block of code several number of times.

Sr.No.	Loop Type & Description
1	**while loop** It repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body.
2	**for loop** It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	**do...while loop** It is similar to a while statement, except that it tests the condition at the end of the loop body
4	**nested loops** You can use one or more loop inside any another while, for or do..while loop.

For loop

```
            int i;
            string text = "";
            for (i = 0; i < 10; i++)
            {
                text = "Line " + Convert.ToString(i);
                Console.WriteLine("{0}", text);
            }
            Console.WriteLine("Last text output in loop: {0}", text);
            Console.ReadKey();
        }
```

Do While

```
            /* local variable definition */
            int a = 10;
            /* do loop execution */
            do
            {
                Console.WriteLine("value of a: {0}", a);
                a = a + 1;
            }
            while (a < 20);
            Console.ReadLine();
        }
```

While loop

```
int a = 10;
            /* while loop execution */
            while (a < 20)
            {
                Console.WriteLine("value of a: {0}", a);
                a++;
            }
            Console.ReadLine();
```

For Each Loop

```
string helloWorld = "Hello, world!";
            foreach (char c in helloWorld)
            {
                //Console.WriteLine(c);
                //a char is a numerical value, where each character has a specific number in the Unicode "alphabet".
                Console.WriteLine(c + ": " + (int)c);
            }
```

1	Create a For Loop that displays 1 to 50 onto the console
2	Write a console program that displays the odd numbers from 1 to 99
3	Write a while loop programme that displays the numbers 1 through to 10 onto the console
4	Create a do while loop that displays the numbers from 1 to 10 onto the console
5	Write a C# program that gets an integer number from the user and sums the digits of the integer

Methods

Passing in Parameters, Returning values

Methods can return a value to the caller. If the return type, the type listed before the method name, is not `void`, the method can return the value by using the `return`keyword. A statement with the `return` keyword followed by a value that matches the return type will return that value to the method caller.

Get 2 numbers from the user and create a method to add them together

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

            Console.Write("\n\nFunction to calculate the sum of two numbers :\n");
            Console.Write("--------------------------------------------------\n");
            Console.Write("Enter a number: ");
            int n1 = Convert.ToInt32(Console.ReadLine());
            Console.Write("Enter another number: ");
            int n2 = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("\nThe sum of two numbers is : {0} \n", Sum(n1, n2));

        }

        public static int Sum(int num1, int num2)
        {
            int total;
            total = num1 + num2;
            return total;
        }
```

1	Create a console program that the user enters a number, the square is calculated and the result displayed on the screen

Methods Passing in By Reference By Value

We create an integer, assign the number 20 to it, and then we use the AddFive() method, which should add 5 to the number. But does it? No.

The value we assign to number inside the function, is never carried out of the function, because we have **passed a copy of the number value** instead of a reference to it.

This is simply how C# works, and in a lot of cases, it's the preferred result. However, in this case, we actually wish to modify the number inside our function.

Enter the ref keyword:

```
static void Main(string[] args)
        {
            int number = 20;
            AddFive(ref number);
            Console.WriteLine(number);
            Console.ReadKey();
        }

        static void AddFive(ref int number)
        {
            number = number + 5;
        }
```

```
static void Main(string[] args)
        {
            int number = 20;
            AddFive(number);
            Console.WriteLine(number);
            Console.ReadKey();
        }

        static void AddFive(int number)
        {
            number = number + 5;
        }
```

Static

A static class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. In other words, you cannot use the new keyword to create a variable of the class type. Because there is no instance variable, you access the members of a static class by using the class name itself.

Create a simple calculator program to do simple arithmetic on 2 numbers entered by the user. The user enters 2 numbers and selects the arithmetic operation (+ - * or divide) The result is displayed on the console

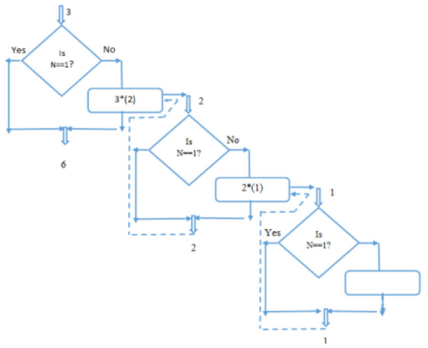
Write a program remove specified a character from a non-empty string using index of a character.

Recursion

They're just products, indicated by an exclamation mark. For instance, "four **factorial**" is written as "4!" and means 1×2×3×4 = 24. In general, n! ("enn **factorial**") means the product of all the whole numbers from 1 to n.

4 Factorial is 24 (1x2x3x4)

3 factorial is 1x2x3=6



```
//prompt the user, get the input and convert it to an integer
            Console.WriteLine("Please enter a number to find out its factorial: ");
            //an unsigned int is needed here so that the arithmetic in the method
            //works - C# doesn't like having signed and unsigned data types in the same
            //arithmetic statement
            uint number = (Convert.ToUInt32(Console.ReadLine()));
            //an unsigned long is the largest integer data type in C# and allows
            //us to get factorials up to 65!
            ulong result = Factorial(number);
            Console.WriteLine("The factorial of {0} is {1:N0}.", number, result);
            //keep the console window open until another key is pressed
            Console.ReadKey();
        }
        //method for finding the factorial
        public static ulong Factorial(uint number)
        {
            //base condition, which stops the recursion looping endlessly
            if (number == 0)
            {
                return 1;
            }
            else
            {
                return number * Factorial(number - 1);
            }

        }
```

Arrays

Array size must be declared at the start and cannot be altered

Display the sum of the numbers in an array

```
  int[] ids = new int[] { 6, 7, 8, 10 };
            int sum = 0;
            int i = 0;
            do
            {
                sum += ids[i];
                i++;
            } while (i < 4);


   System.Console.WriteLine(sum);
```

Sum the values in an array

```
        static void Main(string[] args)
        {
            int sum = SumVals(1, 5, 2, 9, 8);
            Console.WriteLine("Summed Values = {0}", sum);
            Console.ReadKey();
        }

        static int SumVals(params int[] vals)
        {
            int sum = 0;
            foreach (int val in vals)
            {
                sum += val;
            }
            return sum;
        }
```

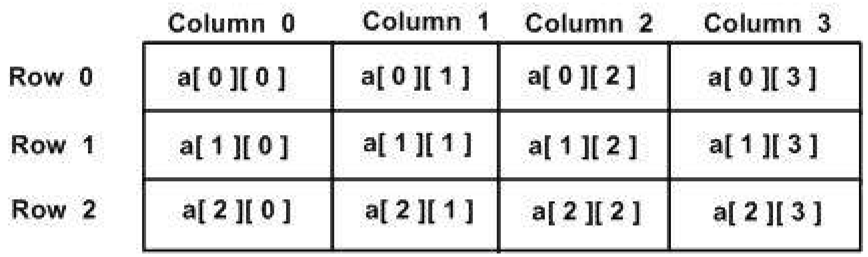
1	Create a program that stores the names of the 7 Dwarfs in an array and then displays the names onto the console

2	Create a program where the user inputs 5 numbers finds and displays the average of those 5 numbers
3	Create an algorithm that processes the following set of numbers and displays the largest one in the console. The numbers are 11, 8, 32, 61, 2, 53, 19, 3, 34, 51,

Two Dimensional Arrays

The simplest form of the multidimensional array is the 2-dimensional array. A 2-dimensional array is a list of one-dimensional arrays.

A 2-dimensional array can be thought of as a table, which has x number of rows and y number of columns. Following is a 2-dimensional array, which contains 3 rows and 4 columns –



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

```
/* an array with 5 rows and 2 columns*/
            int[,] a = new int[5, 2] { { 0, 0 }, { 1, 2 }, { 2, 4 }, { 3, 6 }, { 4, 8 } };
            int i, j;

            /* output each array element's value */
            for (i = 0; i < 5; i++)
            {

                for (j = 0; j < 2; j++)
                {
                    Console.WriteLine("a[{0},{1}] = {2}", i, j, a[i, j]);
                }
            }
            Console.ReadKey();
```

Lists

Lists are similar to arrays except you can extend the size of a list after declaring it

```
//Declare a list of integers
List<int> list = new List<int>();
            list.Add(2);
            list.Add(3);
            list.Add(7);

            // Loop through List with foreach.
            foreach (int prime in list)
            {
                System.Console.WriteLine(prime);
            }
```

```
List<int> list = new List<int>(new int[] { 2, 3, 7 });
            // Loop with for and use string interpolation to print values.
            for (int i = 0; i < list.Count; i++)
            {
                Console.WriteLine($"{i} = {list[i]}");
            }
```

```
List<int> primes = new List<int>(new int[] { 19, 23, 29 });

            int index = primes.IndexOf(23); // Exists.
            Console.WriteLine(index);

            index = primes.IndexOf(10); // Does not exist.
            Console.WriteLine(index);
```

Books Application

Brief

Create a program that manages a list of books. The program enables the user to add, delete, list and sort the books

Book Design

Dictionaries

Create a program that contains a dictionary of coding languages that you can look up by key or description, count the list or exit

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

Projects

Guess the Number

Build a small console application where the user has to guess a secret number between 1 and 100

Must Have	Number to guess is between 1 and 100 User is given feedback on each attempt whether guess is too high or low If user guesses secret number correctly, three pieces of information are displayed (1) the correct number (2)the number of guesses (3)a message “Well done – Correct guess!”
Could Have	User is allowed 4 attempts only and then game is over
Wishlist	Validate input Store a winner lowest number of guesses Reset Game on completion

Simple Calculator

Create a small console programme that enables users to enter 2 numbers and perform a simple arithmetic calculation + - * / and show the answer on the screen.

Must Have	Can perform the 4 basic calculations on 2 numbers Answer is displayed on the screen
Could Have	Can have additional mathematical calculations eg power of Can reset and clear the screen Screen styled to look like a calculator

Mini Project **Interest only Calculator** Create a small programme that enables users to calculate the interest they would pay on borrowing some money for a period of time at a fixed interest rate.	
Must Have	User inputs, amount to borrow, interest rate, length of time of loan Display the monthly interest payment on the loan, Total interest paid on the loan for the length of time of loan.