https://github.com/mdsamiulhasan

----------C# CheatSheet----------

----------Md Samiul Hasan----------

----------Mob: 01538380598----------

==========

C# (C-Sharp) is a programming language developed by Microsoft that runs on the .NET Framework.

C# is used to develop web apps, desktop apps, mobile apps, games and much more.

ALL keywords and short description (78 keywords)

- **abstract - Used to define an incomplete class or method that must be implemented in a derived class.**
- **as - Performs safe type conversion and returns null if the conversion fails.**
- **base - Refers to the base class of the current instance and can be used to access base class members.**
- **bool - Represents a Boolean data type with values true or false.**
- **break - Exits the current loop or switch statem**
- **byte - Represents an 8-bit unsigned integer.**
- **case - Defines a branch in a switch statement.**
- **catch - Handles exceptions thrown in a try block.**
- **char - Represents a single 16-bit Unicode character.**

- **checked - Enables overflow checking for arithmetic operations and conversions.**
- **class - Declares a class, which is a blueprint for objects.**
- **const - Declares a constant field or local variable that cannot be modified.**
- **continue - Skips the current iteration of a loop and moves to the next iteration.**
- **decimal - Represents a 128-bit decimal value for financial and monetary calculations.**
- **default - Specifies the default case in a switch statement or a default value for a generic type parameter.**
- **delegate - Declares a type that can reference methods with a specific signature.**
- **do - Executes a loop body at least once and then repeats based on a condition.**
- **double - Represents a 64-bit double-precision floating point type.**
- **else - Defines the alternative path of execution in an if statement.**
- **enum - Declares an enumeration, a distinct type with named constant values.**
- **event - Declares an event, which is a way for a class to notify other classes when something happens.**
- **explicit - Defines a user-defined cast that must be invoked explicitly.**

- **extern** – Declare a method that is implemented externally, often in unmanaged code.
- **false** - Represents the Boolean value false.
- **finally** - Defines a block of code that will always execute, regardless of whether an exception was thrown.
- **fixed** - Pins a variable in memory, preventing the garbage collector from relocating it.
- **float** - Represents a 32-bit single-precision floating point type.
- **for** - Defines a loop with an initializer, condition, and iterator.
- **foreach** - Iterates over elements in a collection or array.
- **goto** - Transfers control to a labeled statement within the same method.
- **if** - Evaluates a condition and executes a block of code if the condition is true.
- **implicit** - Defines a use re -defined cast that can be invoked implicitly.
- **in** - Specifies that a parameter is passed by reference but cannot be modified by the called method.
- **int** - Represents a 32-bit signed integer.
- **interface** - Declares a contract that classes or structs can implement.
- **internal** - Limits access to the current assembly.
- **is** - Checks if an object is compatible with a given type.

- **lock - Ensures that one thread does not enter a critical section of code while another thread is in it.**
- **long - Represents a 64-bit signed integer.**
- **namespace - Declares a scope that contains a set of related objects like classes and interfaces.**
- **new - Creates a new instance of a type or hides a member inherited from a base class.**
- **null - Represents a null reference, indicating that an object does not point to any data.**
- **object - The root type in the C# type hierarchy, from which all types derive.**
- **operator - Defines a custom implementation for a built-in operator or a custom operator.**
- **out - Specifies that a parameter is passed by reference and must be assigned before the method returns.**
- **override - Provides a new implementation for a virtual or abstract method in a base class.**
- **params - Allows a method to accept a variable number of arguments as an array.**
- **private - Restricts access to the containing class.**
- **protected - Restricts access to the containing class and derived classes.**
- **public - Allows access from any other code.**
- **readonly - Specifies that a field can only be assigned during initialization or in a constructor.**

- **ref** - Passes an argument by reference, allowing the method to modify the argument's value.
- **return** - Exits a method and optionally returns a value.
- **sbyte** - Represents an 8-bit signed integer.
- **sealed** - Prevents a class from being inherited or a method from being overridden.
- **short** - Represents a 16-bit signed integer.
- **sizeof** - Returns the size in bytes of an unmanaged type.
- **stackalloc** - Allocates a block of memory on the stack.
- **static** - Declares a member that belongs to the type itself rather than to any specific object.
- **string** - Represents a sequence of Unicode characters.
- **struct** - Declares a value type that can encapsulate data and related functionality.
- **switch** - Selects a case to execute based on the value of a variable.
- **this** - Refers to the current instance of the class or struct.
- **throw** - Signals the occurrence of an exception.
- **true** - Represents the Boolean value true.
- **try** - Defines a block of code to be tested for exceptions.
- **typeof** - Obtains the System.Type object for a type.
- **uint** - Represents a 32-bit unsigned integer.

- **ulong - Represents a 64-bit unsigned integer.**
- **unchecked - Disables overflow checking for arithmetic operations and conversions.**
- **unsafe - Allows code that uses pointers and direct memory manipulation.**
- **ushort - Represents a 16-bit unsigned integer.**
- **using - Imports types from a namespace or ensures the disposal of an object.**
- **virtual - Allows a method to be overridden in a derived class.**
- **void - Specifies that a method does not return a valu**
- **volatile - Indicates that a field may be modified by multiple threads, preventing optimizations that assume otherwise.**
- **while - Executes a block of code as long as a specified condition is true.**
- **yield - Pauses the execution of a method and returns a value to the caller in an iterator**

C# IDE

1.visula studio 2022

DESCRIPTION:

```
using System;


namespace HelloWorld
{
```

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("Hello World!");
  }
}
```

using System******

Using Directive: This line imports the System namespace, which contains fundamental classes, including Console, that are necessary for basic input and output operations.

namespace HelloWorld

Namespace Declaration: Defines a namespace called HelloWorld. Namespaces are used to organize code and prevent naming conflicts.

csharp

class Program

Class Declaration: Defines a class named Program. A class is a blueprint for objects and contains methods and properties. In C#, every program must have at least one class with a Main method.

static void Main(string[] args)

Main Method: This is the entry point of the program. When the program runs, the execution starts from this method. The Main method is static because it is called by the runtime without creating an instance of the class. The string[] args parameter allows the program to accept command-line arguments.

****

Console.WriteLine("Hello, World!");

Console.WriteLine: This line prints the string "Hello, World!" to the console. Console is a class in the System namespace, and WriteLine is a method of the Console class that outputs text followed by a newline

-variable-------

- **type:5**

```
int myNum = 5;
double myDoubleNum = 5.99D;
```

```
char myLetter = 'D';
bool myBool = true;
string myText = "Hello";
```

❖ Variable Constants**********

```
const int myNum = 15;
myNum = 20; // error
```

❖ Display Variables

The WriteLine() method is often used to display variable values to the console window.

Example

```
string name = "John";
Console.WriteLine("Hello " + name);
```

❖Data Types*

Data Type	Size	Description
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

float 4 bytes = Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits

double 8 bytes = Stores fractional numbers. Sufficient for storing 15 decimal digits

bool 1 bit = Stores true or false values

char 2 bytes Stores a single character/letter, surrounded by single quotes

string 2 bytes per character == Stores a sequence of characters, surrounded by double quote

Type casting

TYPE CASTING IS WHEN YOU ASSIGN A VALUE OF ONE DATA TYPE TO ANOTHER TYPE.

IN C#, THERE ARE TWO TYPES OF CASTING:

IMPLICIT CASTING (AUTOMATICALLY) - CONVERTING A SMALLER TYPE TO A LARGER TYPE SIZE

CHAR -> INT -> LONG -> FLOAT -> DOUBLE

EXPLICIT CASTING (MANUALLY) - CONVERTING A LARGER TYPE TO A SMALLER SIZE TYPE

DOUBLE -> FLOAT -> LONG -> INT -> CHAR

----------EXAMPLE

```
INT MYINT = 9;
DOUBLE MYDOUBLE = MYINT;      // AUTOMATIC CASTING: INT TO DOUBLE

CONSOLE.WRITELINE(MYINT);     // OUTPUTS 9
```

```
Console.WriteLine(myDouble);  // Outputs 9
```

explicit to implicit

```
---------double myDouble = 9.78;
int myInt = (int) myDouble;   // Manual casting: double to int

Console.WriteLine(myDouble);  // Outputs 9.78
Console.WriteLine(myInt);     // Outputs 9
```

Type Conversion Methods

```
int myInt = 10;
double myDouble = 5.25;
bool myBool = true;

Console.WriteLine(Convert.ToString(myInt));    // convert int to string
Console.WriteLine(Convert.ToDouble(myInt));    // convert int to double
Console.WriteLine(Convert.ToInt32(myDouble));  // convert double to int
Console.WriteLine(Convert.ToString(myBool));   // convert bool to string
```

User Input and Numbers

The Console.ReadLine() method returns a string. Therefore, you cannot get information from another data type,

such as int. The following program will cause an error:

```
//Get User Input
Console.WriteLine("Enter User Name: ");
string username = Console.ReadLine();
Console.WriteLine("Your Name is: "+ username);

Console.WriteLine("Enter Your age: ");
int ageNumber = Console.ReadLine();
Console.WriteLine("Your age is: " + ageNumber);
```

Operators

Operators are used to perform operations on variables and values.

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations:

Operator	NameDescription	Example	Try it
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y

/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	x++
--	Decrement	Decreases the value of a variable by 1	x--

Assignment Operators

Assignment operators are used to assign values to variables.

we use the assignment operator (=) to assign the value 10 to a variable called x:

Example

```
int x = 10;
```

----------C# Math----------

The C# Math class has many methods that allows you to perform mathematical tasks on numbers.

Math.Max(x,y)

The Math.Max(x,y) method can be used to find the highest value of x and y:

ExampleGet your own C# Server

```
Math.Max(5, 10);
```

Math.Min(x,y)

The Math.Min(x,y) method can be used to find the lowest value of of x and y:

Example

```
Math.Min(5, 10);
```

Math.Sqrt(x)

The Math.Sqrt(x) method returns the square root of x:

Example

```
Math.Sqrt(64);
```

C# Strings=====

Strings are used for storing text.

A string variable contains a collection of characters surrounded by double quotes:

Create a variable of type string and assign it a value:

```
string greeting = "Hello";
```

A string variable can contain many words, if you want:

- **Example**

```
string greeting2 = "Nice to meet you!";
```

❖ String Length

A string in C# is actually an object, which contain properties and methods that can perform certain operations on strings. For example, the length of a string can be found with the Length property:

❖ Example

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
Console.WriteLine("The length of the txt string is: " + txt.Length);
```

❖ Other Methods

There are many string methods available, for example ToUpper() and ToLower(), which returns a copy of the string converted to uppercase or lowercase:

Example

```
string txt = "Hello World";
Console.WriteLine(txt.ToUpper());   // Outputs "HELLO WORLD"
Console.WriteLine(txt.ToLower());   // Outputs "hello world"
```

****
*

C# String Concatenation

The + operator can be used between strings to combine them. This is called concatenation:

ExampleGet your own C# Server

```
string firstName = "John ";
string lastName = "Doe";
string name = firstName + lastName;
Console.WriteLine(name);
```

You can also use the string.Concat() method to concatenate two strings:

Example

```
string firstName = "John ";
string lastName = "Doe";
string name = string.Concat(firstName, lastName);
Console.WriteLine(name);
```

❖ Adding Numbers and Strings

WARNING!

C# uses the + operator for both addition and concatenation.

Remember: Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

❖ Example

```
int x = 10;
int y = 20;
```

```
int z = x + y;  // z will be 30 (an integer/number)
```

If you add two strings, the result will be a string concatenation:

❖ Example

```
string x = "10";
string y = "20";
string z = x + y;  // z will be 1020 (a string)
```

String Interpolation

Another option of string concatenation, is string interpolation,

which substitutes values of variables into placeholders in a string.

Note that you do not have to worry about spaces, like with concatenation:

❖ Example

```
string firstName = "John";
string lastName = "Doe";
string name = $"My full name is: {firstName} {lastName}";
Console.WriteLine(name);
```

C# Access Strings

You can access the characters in a string by referring to its index number inside square brackets [].

This example prints the first character in myString:

ExampleGet your own C# Server

```
string myString = "Hello";
Console.WriteLine(myString[0]);  // Outputs "H"
```

Note: String indexes start with 0: [0] is the first character. [1] is the second character, etc.

❖ Example

```
string myString = "Hello";
Console.WriteLine(myString[1]);  // Outputs "e"
```

You can also find the index position of a specific character in a string, by using the IndexOf() method:

❖ Example

```
string myString = "Hello";
Console.WriteLine(myString.IndexOf("e"));  // Outputs "1"
```

Another useful method is Substring(), which extracts the characters from a string,

starting from the specified character position/index, and returns a new string.

This method is often used together with IndexOf() to get the specific character position:

❖ Example

```
// Full name
string name = "John Doe";

// Location of the letter D
int charPos = name.IndexOf("D");

// Get last name
string lastName = name.Substring(charPos);

// Print the result
Console.WriteLine(lastName);
```

C# Special Characters

Strings - Special Characters

Because strings must be written within quotes, C# will misunderstand this string, and generate an error:

```
string txt = "We are the so-called "Vikings" from the north.";
```

**********Switch

❖ C# Switch

C# Switch Statements

Use the switch statement to select one of many code blocks to be executed.

❖ Syntax

```
switch(expression)
{
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
    break;
}
```

This is how it works:

The switch expression is evaluated once

The value of the expression is compared with the values of each case

If there is a match, the associated block of code is executed

The break and default keywords will be described later in this chapter

The example below uses the weekday number to calculate the weekday name:

❖ **Example**

```
int day = 4;
switch (day)
{
  case 1:
    Console.WriteLine("Monday");
    break;
  case 2:
    Console.WriteLine("Tuesday");
    break;
  case 3:
    Console.WriteLine("Wednesday");
    break;
  case 4:
    Console.WriteLine("Thursday");
    break;
  case 5:
    Console.WriteLine("Friday");
    break;
  case 6:
    Console.WriteLine("Saturday");
    break;
  case 7:
    Console.WriteLine("Sunday");
```

```
  break;
}
// Outputs "Thursday" (day 4)
```

❖ The break Keyword

When C# reaches a break keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

The default keyword is optional and specifies some code to run if there is no case match:

❖ Example

```
int day = 4;
switch (day)
{
  case 6:
    Console.WriteLine("Today is Saturday.");
    break;
  case 7:
```

```
    Console.WriteLine("Today is Sunday.");
    break;
  default:
    Console.WriteLine("Looking forward to the Weekend.");
    break;
}
```

**********Do while Loop

❖ C# While Loop

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

❖ C# While Loop

The while loop loops through a block of code as long as a specified condition is True:

```
while (condition)
{
  // code block to be executed
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

❖ **Example**

```
int i = 0;
while (i < 5)
{
  Console.WriteLine(i);
  i++;
}
```

Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!

The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

❖ **Syntax**

```
do
{
  // code block to be executed
}
while (condition);
```

The example below uses a do/while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

❖ Example

```
int i = 0;
do
{
  Console.WriteLine(i);
  i++;
}
while (i < 5);
```

**********FOR LOOP

C# For Loop

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

❖ Syntax

```
for (statement 1; statement 2; statement 3)
{
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

❖ Example

```
for (int i = 0; i < 5; i++)
{
  Console.WriteLine(i);
}
```

❖ Example explained

Statement 1 sets a variable before the loop starts (int i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

❖ Another Example

This example will only print even values between 0 and 10:

❖ Example

```
for (int i = 0; i <= 10; i = i + 2)
{
  Console.WriteLine(i);
}
```

****

C# Arrays

❖ Create an Array

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with square brackets:

❖ string[] cars;

We have now declared a variable that holds an array of strings.

To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

To create an array of integers, you could write:

```
int[] myNum = {10, 20, 30, 40};
```

Access the Elements of an Array

You access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

❖ Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
Console.WriteLine(cars[0]);
// Outputs Volvo
```

❖ ADVERTISEMENT

Change an Array Element

To change the value of a specific element, refer to the index number:

Example

```
cars[0] = "Opel";
```

Example

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
Console.WriteLine(cars[0]);
// Now outputs Opel instead of Volvo
```

==========

❖ C# Sort Arrays

❖ Sort an Array

There are many array methods available, for example Sort(), which sorts an array alphabetically or in an ascending order:

ExampleGet your own C# Server

```
// Sort a string
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
Array.Sort(cars);
```

```
foreach (string i in cars)
{
  Console.WriteLine(i);
}

// Sort an int
int[] myNumbers = {5, 1, 8, 9};
Array.Sort(myNumbers);
foreach (int i in myNumbers)
{
  Console.WriteLine(i);
}
```

Other useful array methods, such as Min, Max, and Sum, can be found in the System.Linq namespace:

❖ **Example**

```
using System;
using System.Linq;

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

```
  {
    int[] myNumbers = {5, 1, 8, 9};
    Console.WriteLine(myNumbers.Max());  // returns the largest value
    Console.WriteLine(myNumbers.Min());  // returns the smallest value
    Console.WriteLine(myNumbers.Sum());  // returns the sum of elements
  }
 }
}
```

❖C# Multidimensional Arrays

In the previous chapter, you learned about arrays, which is also known as single dimension arrays. These are great, and something you will use a lot while programming in C#. However, if you want to store data as a tabular form, like a table with rows and columns, you need to get familiar with multidimensional arrays.

A multidimensional array is basically an array of arrays.

Arrays can have any number of dimensions. The most common are two-dimensional arrays (2D).

Two-Dimensional Arrays

To create a 2D array, add each array within its own set of curly braces, and insert a comma (,) inside the square brackets:

❖ Example

```
int[,] numbers = { {1, 4, 2}, {3, 6, 8} };
```

Good to know: The single comma [,] specifies that the array is two-dimensional. A three-dimensional array would have two commas: int[,,].

numbers is now an array with two arrays as its elements. The first array element contains three elements: 1, 4 and 2, while the second array element contains 3, 6 and 8. To visualize it, think of the array as a table with rows and columns:

❖ Access Elements of a 2D Array

To access an element of a two-dimensional array, you must specify two indexes: one for the array, and one for the element inside that array. Or better yet, with the table visualization in mind; one for the row and one for the column (see example below).

This statement accesses the value of the element in the first row (0) and third column (2) of the numbers array:

❖ Example

```
int[,] numbers = { {1, 4, 2}, {3, 6, 8} };
Console.WriteLine(numbers[0, 2]);  // Outputs 2
```

==========

❖ Access Elements of a 2D Array

To access an element of a two-dimensional array, you must specify two indexes: one for the array, and one for the element inside that array. Or better yet, with the table visualization in mind; one for the row and one for the column (see example below).

This statement accesses the value of the element in the first row (0) and third column (2) of the numbers array:

❖ Example

```
int[,] numbers = { {1, 4, 2}, {3, 6, 8} };
Console.WriteLine(numbers[0, 2]);  // Outputs 2
```

Remember that: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

❖ Change Elements of a 2D Array

You can also change the value of an element.

The following example will change the value of the element in the first row (0) and first column (0):

❖ Example

```
int[,] numbers = { {1, 4, 2}, {3, 6, 8} };
numbers[0, 0] = 5;  // Change value to 5
Console.WriteLine(numbers[0, 0]); // Outputs 5 instead of 1
```

❖ Loop Through a 2D Array

You can easily loop through the elements of a two-dimensional array with a foreach loop:

❖ Example

```
int[,] numbers = { {1, 4, 2}, {3, 6, 8} };


foreach (int i in numbers)
{
  Console.WriteLine(i);
}
```

You can also use a for loop. For multidimensional arrays, you need one loop for each of the array's dimensions.

❖ Example

```
int[,] numbers = { {1, 4, 2}, {3, 6, 8} };


for (int i = 0; i < numbers.GetLength(0); i++)
{
  for (int j = 0; j < numbers.GetLength(1); j++)
   Console.WriteLine(numbers[i, j]);
}
}
```

❖-C# Method

A method is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

❖ example:

```
class Program
{
  static void MyMethod()
  {
    // code to be executed
  }
}
```

MyMethod() is the name of the method

static means that the method belongs to the Program class and not an object of the Program class. You will learn more about objects and how to access methods through objects later in this tutorial.

void means that this method does not have a return value. You will learn more about return values later in this chapter

❖ Example

Inside Main(), call the myMethod() method:

```
static void MyMethod()
{
  Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
  MyMethod();
}

// Outputs "I just got executed!"
```

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

They are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a string called fname as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

❖ Example

```
static void MyMethod(string fname)
{
  Console.WriteLine(fname + " Refsnes");
}

static void Main(string[] args)
{
  MyMethod("Liam");
  MyMethod("Jenny");
  MyMethod("Anja");
}
```

==========

❖ Multiple Parameters

You can have as many parameters as you like, just separate them with commas:

❖ Example

```
static void MyMethod(string fname, int age)
{
  Console.WriteLine(fname + " is " + age);
}
```

```
static void Main(string[] args)
{
  MyMethod("Liam", 5);
  MyMethod("Jenny", 8);
  MyMethod("Anja", 31);
}
```

==========

❖ Default Parameter Value

You can also use a default parameter value, by using the equals sign (=).

If we call the method without an argument, it uses the default value ("Norway"):

ExampleGet your own C# Server

```
static void MyMethod(string country = "Norway")
{
  Console.WriteLine(country);
}

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

```
  MyMethod("Sweden");
  MyMethod("India");
  MyMethod();
  MyMethod("USA");
}

// Sweden
// India
// Norway
// USA
```

EXAMPLE:

```
using System;


namespace DefaultParameter
{
    class SamiulHasan
    {
        static void MyMethod(string name, int age,string country)
        {
            Console.WriteLine(name + " is " + age + " years old." + "He Lives in " + country +".");
        }
```

```
        static void Main(string[] args)
        {
            MyMethod("Samiul", 24, "Bangladesh");
             MyMethod("Hasan", 24, "india");
              MyMethod("ASIF", 24, "Bangladesh");
        }




    }
}
```

❖ ==========example

```
/*
namespace Program
{
    class Samiul
    {
        static void MyMethod(string name, int age)
        {
            Console.WriteLine(name + " is " + age + " years old");
        }
        static void Main(string[] args)
        {
```

```
        MyMethod("Samiul", 24);
        MyMethod("Hasan", 23);
        MyMethod("Asif", 22);
      }
   }
}

*/
```

==========

❖ Return Values

In the previous page, we used the void keyword in all examples, which indicates that the method should not return a value.

If you want the method to return a value, you can use a primitive data type (such as int or double) instead of void, and use the return keyword inside the method:

ExampleGet your own C# Server

```
static int MyMethod(int x)
{
  return 5 + x;
}

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

```
  Console.WriteLine(MyMethod(3));
}
```

This example returns the sum of a method's two parameters:

Example

```
static int MyMethod(int x, int y)
{
  return x + y;
}

static void Main(string[] args)
{
  Console.WriteLine(MyMethod(5, 3));
}

// Outputs 8 (5 + 3)
```

==========

❖ Example

```
static int MyMethod(int x, int y)
{
  return x + y;
}
```

```
static void Main(string[] args)
{
  int z = MyMethod(5, 3);
  Console.WriteLine(z);
}

// Outputs 8 (5 + 3)
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):

❖ **Example**

```
static int MyMethod(int x, int y)
{
  return x + y;
}

static void Main(string[] args)
{
  int z = MyMethod(5, 3);
  Console.WriteLine(z);
}

// Outputs 8 (5 + 3)
```

----------another example

```
namespace Program
{
    class Samiul
    {
        static double MyMethod(int x, int y)
        {
            return x + y;
        }
        static void Main(string[] args)
        {
            double z = MyMethod(5,5);
            Console.WriteLine("The output is: " + z );
        }
    }
}
```

C# Method Parameters

Information can be passed to methods as parameter. Parameters act as variables inside the method.

They are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma

```
static void MyMethod(string child1, string child2, string child3)
{
  Console.WriteLine("The youngest child is: " + child3);
}

static void Main(string[] args)
{
  MyMethod(child3: "John", child1: "Liam", child2: "Liam");
}

// The youngest child is: John
```

```
using System;
namespace Program
{

   class Samiul
   {
      static void MyMethod(string sam1,string sam2,string sam3)
      {
         Console.WriteLine("The younger Public is: "+ sam3);
```

```
    }
    static void Main(string[] args)
    {
        MyMethod(sam1: "Samiul",sam2: "Hasan", sam3: "Roxy");
    }
  }
}
```

❖ Method Overloading

With method overloading, multiple methods can have the same name with different parameters:

Example

```
static int PlusMethodInt(int x, int y)
{
  return x + y;
}

static double PlusMethodDouble(double x, double y)
{
  return x + y;
}

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

```
 int myNum1 = PlusMethodInt(8, 5);
 double myNum2 = PlusMethodDouble(4.3, 6.26);
 Console.WriteLine("Int: " + myNum1);
 Console.WriteLine("Double: " + myNum2);
}
```

==========

```
namespace Program
{
   class samiul
   {
      static int MyMethod(int x,int y)
      {
         return x + y;
      }
      static double MyMethod(double x, double y)
      {
         return x + y;

      }
      static void Main(string[] args)
      {
         int z1 = MyMethod(10,30);
```

```
        double z2 = MyMethod(10.32, 32.21);
        Console.WriteLine("int is: " + z1);
        Console.WriteLine("Double is: " + z2);
    }
  }
}
```

❖ C# Classes and Objects

You learned from the previous chapter that C# is an object-oriented programming language.

Everything in C# is associated with classes and objects, along with its attributes and methods.

For example: in real life, a car is an object. The car has attributes, such as weight and color,

and methods, such as drive and brake.

❖ example:

```
namespace Program
{
  class Samiul
  {
    string color = "red";
```

```
        static void Main(string[] args)
        {
            Samiul myObj = new Samiul();
            Console.WriteLine("The best color is: " + myObj.color);
        }
    }
}
```

❖ example of Multiple Objects:

```
namespace Program
{
    class samiul
    {
        string color = "Red";
        static void Main(string[] args)
        {
            samiul MyObj = new samiul();
            samiul MyObject = new samiul();
            Console.WriteLine("The best color is: " + MyObject.color);
            Console.WriteLine("The best choice is: " + MyObj.color);

        }
    }
}
```

==========

❖ C# Class Members

==Fields and methods inside classes are often referred to as "Class Members":

❖ Fields

In the previous chapter, you learned that variables inside a class are called fields,

and that you can access them by creating an object of the class, and by using the dot syntax (.).

==========

==========

```
namespace Program
{
   class Samiul
   {
      string color;
      int age;
      string entertaiment;
      static void Main(string[] args)
      {
         Samiul MyObj = new Samiul();
         MyObj.color = "Red";
         MyObj.color = "Green";
```

```
        Samiul Myobj = new Samiul();
        Myobj.age = 24;

        Samiul MyHobby = new Samiul();
        MyHobby.entertaiment = "Watching Movie";

        Console.WriteLine("The output is: " + MyObj.color);
        Console.WriteLine("The output is: " + Myobj.age);
        Console.WriteLine("The output is: " + MyHobby.entertaiment);
    }


  }
}
```

==========Constructor

Constructors

A constructor is a special method that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields:

ExampleGet your own C# Server

Create a constructor:

```
// Create a Car class
class Car
{
  public string model;  // Create a field

  // Create a class constructor for the Car class
  public Car()
  {
    model = "Mustang"; // Set the initial value for model
  }

  static void Main(string[] args)
  {
    Car Ford = new Car();  // Create an object of the Car Class (this will call the constructor)
    Console.WriteLine(Ford.model);  // Print the value of model
  }
}

// Outputs "Mustang"
```

Note that the constructor name must match the class name, and it cannot have a return type (like void or int).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, C# creates one for you. However, then you are not able to set initial values for fields.

Constructors save time! Take a look at the last example on this page to really understand why.

Constructor Parameters

Constructors can also take parameters, which is used to initialize fields.

The following example adds a string modelName parameter to the constructor. Inside the constructor we set model to modelName (model=modelName). When we call the constructor, we pass a parameter to the constructor ("Mustang"), which will set the value of model to "Mustang":

❖ Example

```
class Car
{
 public string model;

 // Create a class constructor with a parameter
 public Car(string modelName)
 {
  model = modelName;
```

```
  }

  static void Main(string[] args)
  {
    Car Ford = new Car("Mustang");
    Console.WriteLine(Ford.model);
  }
}

// Outputs "Mustang"
```

You can have as many parameters as you want:

❖ **Example**

```
class Car
{
  public string model;
  public string color;
  public int year;

  // Create a class constructor with multiple parameters
  public Car(string modelName, string modelColor, int modelYear)
  {
    model = modelName;
```

```
    color = modelColor;
    year = modelYear;
  }

  static void Main(string[] args)
  {
    Car Ford = new Car("Mustang", "Red", 1969);
    Console.WriteLine(Ford.color + " " + Ford.year + " " + Ford.model);
  }
}


// Outputs Red 1969 Mustang
```

❖ Access Modifiers

By now, you are quite familiar with the public keyword that appears in many of our examples:

❖ C# has the following access modifiers:

Modifier	Description
public	The code is accessible for all classes
private	The code is only accessible within the same class

protected The code is accessible within the same class, or in a class that is inherited from that class. You will learn more about inheritance in a later chapter

internal The code is only accessible within its own assembly, but not from another assembly. You will learn more about this in a later chapter

❖ Private Modifier

If you declare a field with a private access modifier, it can only be accessed within the same class:

❖ Example

```
class Car
{
 private string model = "Mustang";

 static void Main(string[] args)
 {
  Car myObj = new Car();
  Console.WriteLine(myObj.model);
 }
}
```

The output will be:

❖ Mustang

If you try to access it outside the class, an error will occur:

❖ **Example**

```
class Car
{
  private string model = "Mustang";
}

class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
```

The output will be:

```
'Car.model' is inaccessible due to its protection level
The field 'Car.model' is assigned but its value is never used
```

❖ **Public Modifier**

If you declare a field with a public access modifier, it is accessible for all classes:

❖ Example

```
class Car
{
  public string model = "Mustang";
}

class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
```

The output will be:

```
Mustang
```

❖ Why Access Modifiers?

To control the visibility of class members (the security level of each individual class and class member).

To achieve "Encapsulation" - which is the process of making sure that "sensitive" data is hidden from users. This is done by declaring fields as private. You will learn more about this in the next chapter.

Note: By default, all members of a class are private if you don't specify an access modifier:

❖ Example

```
class Car
{
  string model;  // private
  string year;   // private
}
```


❖ Properties and Encapsulation

Before we start to explain properties, you should have a basic understanding of "Encapsulation".

The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

declare fields/variables as private

provide public get and set methods, through properties, to access and update the value of a private field

❖ Properties

You learned from the previous chapter that private variables can only be accessed within the same class (an outside class has no access to it). However, sometimes we need to access them - and it can be done with properties.

A property is like a combination of a variable and a method, and it has two methods: a get and a set method:

❖ Example

```
class Person
{
 private string name; // field

 public string Name   // property
 {
   get { return name; }   // get method
   set { name = value; }  // set method
 }
}
```

❖ Example explained

The Name property is associated with the name field. It is a good practice to use the same name for both the property and the private field, but with an uppercase first letter.

The get method returns the value of the variable name.

The set method assigns a value to the name variable. The value keyword represents the value we assign to the property.

If you don't fully understand it, take a look at the example below.

Now we can use the Name property to access and update the private field of the Person class:

❖ **Example**

```
class Person
{
 private string name; // field
 public string Name   // property
 {
  get { return name; }
  set { name = value; }
 }
}

class Program
{
 static void Main(string[] args)
 {
  Person myObj = new Person();
```

```
    myObj.Name = "Liam";
    Console.WriteLine(myObj.Name);
  }
}
```

- The output will be:

```
Liam
```

Automatic Properties (Short Hand)

C# also provides a way to use short-hand / automatic properties, where you do not have to define the field for the property, and you only have to write get; and set; inside the property.

The following example will produce the same result as the example above. The only difference is that there is less code:

- **Example**
- **Using automatic properties:**

```
class Person
{
```

```
  public string Name  // property
  { get; set; }
}

class Program
{
  static void Main(string[] args)
  {
    Person myObj = new Person();
    myObj.Name = "Liam";
    Console.WriteLine(myObj.Name);
  }
}
```

The output will be:

Liam

❖ Why Encapsulation?

Better control of class members (reduce the possibility of yourself (or others) to mess up the code)

Fields can be made read-only (if you only use the get method), or write-only (if you only use the set method)

Flexible: the programmer can change one part of the code without affecting other parts

Increased security of data

Inheritance (Derived and Base Class)

In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

Derived Class (child) - the class that inherits from another class

Base Class (parent) - the class being inherited from

To inherit from a class, use the : symbol.

In the example below, the Car class (child) inherits the fields and methods from the Vehicle class (parent):

- **Example**

```
class Vehicle  // base class (parent)
{
 public string brand = "Ford";  // Vehicle field
 public void honk()            // Vehicle method
 {
   Console.WriteLine("Tuut, tuut!");
 }
}
```

```
class Car : Vehicle  // derived class (child)
{
  public string modelName = "Mustang";  // Car field
}

class Program
{
  static void Main(string[] args)
  {
    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (From the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand field (from the Vehicle class) and the value of the modelName from the Car class
    Console.WriteLine(myCar.brand + " " + myCar.modelName);
  }
}
```

- **Why And When To Use "Inheritance"?**

- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

Tip: Also take a look at the next chapter, Polymorphism, which uses inherited methods to perform different tasks.

The sealed Keyword

If you don't want other classes to inherit from a class, use the sealed keyword:

If you try to access a sealed class, C# will generate an error:

```
sealed class Vehicle
{
  ...
}
class Car : Vehicle
{
  ...
}
```


❖ Polymorphisom

Polymorphism and Overriding Methods

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; Inheritance lets us inherit fields and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

```
namespace property
{
   class Program
   {
      public string name
      {
         get; set;
      }



      static void Main(string[] args)
      {
         Program obj = new Program();
         obj.name = "Samiul Hasan";
         Console.WriteLine(obj.name);
      }
   }
}
```

- ----------Moreover example

```
namespace Sonarbangla
{
```

```
    class baseclass
    {
        public string FirstName = "Samiul";
        public void baseC()
        {
            Console.WriteLine("Learn Inheritence");
        }
    }
}
```

❖ ----------More over example

```
namespace Sonarbangla
{
    class driveclass : baseclass
    {
        public string LastName = "Hasan";
    }
}
```

❖ ----------More over example

```
namespace Sonarbangla
{
   class Myclass
    {
        public static void Main(string[] args)
        {
```

```
            driveclass omg = new driveclass();
            omg.baseC();
            Console.WriteLine(omg.FirstName + " " + omg.LastName);
        }
    }
}

*/
//create base class
/*
```

❖ ----------More over example

```
namespace polymorphisom
{
    class Animal
    {
        public void soundofAnimal() //make constractor
        {
            Console.WriteLine("Make a sound");

        }
    }
    //derive class
    class dog : Animal
    {
```

```
    public void soundofAnimal()
    {
        Console.WriteLine("Make a kutta sound");
    }
}

class cat : Animal
{
    public void soundofAnimal()
    {
        Console.WriteLine("Make  a cat Sound");
    }

}

class tiger : Animal
{
    public void soundofAnimal()
    {
        Console.WriteLine("Make a tiger sound");
    }
}
//MAIN METHOD

class Program {
```

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

        {
            Animal MyAnimal = new Animal();
            Animal MyCat = new cat();
            Animal MyTiger = new tiger();
            Animal MyDog = new dog();
            //call the function
            MyDog.soundofAnimal();
            MyCat.soundofAnimal();
            MyTiger.soundofAnimal();
            MyAnimal.soundofAnimal();
        }
    }
}

*/
```

❖ ----------More over example

```
namespace poly
{
    class Animal
    {
        public void Sound() //base class
        {
```

```
    Console.WriteLine("Make a Sound");


}
//derive class making
class cat: Animal
{
    public void Sound()
    {
        Console.WriteLine("Make a ca sound");
    }
}
//derive class make
class dog: Animal
{
    public void Sound()
    {
        Console.WriteLine("Make a dogsound");
    }
}
//make a drive class
class cow: Animal
{
    public void Sound()
    {
        Console.WriteLine("Make a humba humba sound");
```

```
        }
    }
    //Main class
    class program
    {
      static void Main(string[] args)
       {
          Animal MyAnimal = new Animal();
          Animal Mycat = new cat();
          Animal Mydog = new dog();
          Animal Mycow = new cow();
          //display the output
          MyAnimal.Sound();
          Mycat.Sound();
          Mydog.Sound();
          Mycow.Sound();

       }
    }
  }
}
```

❖ Abstract Classes and Methods

Data abstraction is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either abstract classes or interfaces (which you will learn more about in the next chapter).

The abstract keyword is used for classes and methods:

Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

Abstract method: can only be used in an abstract class, and it does not have a body.

The body is provided by the derived class (inherited from).

```
// abstract method
namespace application
{
   //abstract
   abstract  class samiul {
      public abstract void hasan();
      public void wakeup()
      {
         Console.WriteLine("Please,Wake up");
      }
```

```
    }

    //derived method
    class roxy : samiul
    {
        public override void hasan()
        {
            Console.WriteLine("Hi,Iam samiul Hasan");
        }
        public static void Main(string[] args)
        {
            roxy rox = new roxy();
            rox.hasan();
            rox.wakeup();
            Console.WriteLine("");
        }


    }


}
```

❖ Interfaces

Another way to achieve abstraction in C#, is with interfaces.

An interface is a completely "abstract class", which can only contain abstract methods and properties (with empty bodies):

❖ Notes on Interfaces:

Like abstract classes, interfaces cannot be used to create objects (in the example above, it is not possible to create an "IAnimal" object in the Program class)

Interface methods do not have a body - the body is provided by the "implement" class

On implementation of an interface, you must override all of its methods

Interfaces can contain properties and methods, but not fields/variables

Interface members are by default abstract and public

An interface cannot contain a constructor (as it cannot be used to create objects)

❖ ----------More over example

```
namespace sonarbangla
{
   interface animal
   {
      void animalsound();
   }

   class dog : animal
   {
```

```
        public void animalsound()
        {
            Console.WriteLine("Hukka huwa ,hukka huwa");
        }

    }
    class Program
    {
        public static void Main(string[] args)
        {
            dog obj = new dog();

            obj.animalsound();


        }

    }
}
```

❖ Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

```
namespace MyApplication
```

```
{
 interface IFirstInterface
 {
   void myMethod(); // interface method
 }

 interface ISecondInterface
 {
   void myOtherMethod(); // interface method
 }

 // Implement multiple interfaces
 class DemoClass : IFirstInterface, ISecondInterface
 {
   public void myMethod()
   {
     Console.WriteLine("Some text..");
   }
   public void myOtherMethod()
   {
     Console.WriteLine("Some other text...");
   }
 }

 class Program
```

```
  {
    static void Main(string[] args)
    {
      DemoClass myObj = new DemoClass();
      myObj.myMethod();
      myObj.myOtherMethod();
    }
  }
}
```

❖ **example level:**

```
namespace eval
{
    class Program
    {
        enum Level{
            low,
            medium,
            high,
        }
        static void Main(string[] args)
        {
            Level mylevel = Level.high;
            Console.WriteLine("The output is: " + mylevel);
```

```
        }
    }
}
*/
```

❖ ----------More over example

```
namespace sami
{
    class program
    {
        enum Level
        {
            small,
            big,

        }
        static void Main(string[] args)
        {
            Level Mylev = Level.big;
            Console.WriteLine(Mylev);
        }
    }
}
```

❖ Working With Files

The File class from the System.IO namespace, allows us to work with files:

ExampleGet your own C# Server

```
using System.IO;  // include the System.IO namespace


File.SomeFileMethod();  // use the file class with methods
```

The File class has many useful methods for creating and getting information about files. For example:

Method	Description
AppendText()	Appends text at the end of an existing file
Copy()	Copies a file
Create()	Creates or overwrites a file
Delete()	Deletes a file
Exists()	Tests whether the file exists
ReadAllText()	Reads the contents of a file
Replace()	Replaces the contents of a file with the contents of another file
WriteAllText()	Creates a new file and writes the contents to it. If the file already exists, it will be overwritten.

For a full list of File methods, go to Microsoft .Net File Class Reference.

In the following example, we use the WriteAllText() method to create a file named "filename.txt" and write some content to it. Then we use the ReadAllText() method to read the contents of the file:

❖ Example

```
using System.IO;  // include the System.IO namespace

string writeText = "Hello World!";  // Create a text string

File.WriteAllText("filename.txt", writeText);  // Create a file and write the content of writeText to it

string readText = File.ReadAllText("filename.txt");  // Read the contents of the file

Console.WriteLine(readText);  // Output the content
```

❖ C# Exceptions

When executing C# code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C# will normally stop and generate an error message. The technical term for this is: C# will throw an exception (throw an error).

❖ C# try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

```
//try and catch
/*
namespace Hasina
{
   class khaleda
   {
     public static void Main(string[] args)
       {
          try
          {
             int[] myNumbers = { 1, 2, 3 };
             Console.WriteLine(myNumbers[3]);
          }
          catch(Exception e)
          {
             Console.WriteLine(e.Message);
          }
```

```
        }
    }
}
```

❖ ----------More over example

```
namespace error
{
    class errorhandle
    {
        public static void Main(string[] args)
        {
            try
            {
                int[] math = { 1, 2, 3, 4, 5 };
                Console.WriteLine(math[7]);
            }catch(Exception e)
            {
                Console.WriteLine("Something went wrong");
            }
        }
    }
}

static void checkAge(int age)
{
```

```
  if (age < 18)
  {
    throw new ArithmeticException("Access denied - You must be at least 18 years old.");
  }
  else
  {
    Console.WriteLine("Access granted - You are old enough!");
  }
}

static void Main(string[] args)
{
  checkAge(15);
}
```