**C – Sharp**

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.-

C# Syntax

using System;

namespace HelloWorld

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Hello World!");

}

}

}

## C# Variables

* int - stores integers (whole numbers), without decimals, such as 123 or -123
* double - stores floating point numbers, with decimals, such as 19.99 or -19.99
* char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
* string - stores text, such as "Hello World". String values are surrounded by double quotes
* bool - stores values with two states: true or false
* int myNum = 5;
* double myDoubleNum = 5.99D;
* char myLetter = 'D';
* bool myBool = true;
* string myText = "Hello";

## C# Identifiers

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

C# Type 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
* int myInt = 9;
* double myDouble = myInt; // Automatic casting: int to double
* Console.WriteLine(myInt); // Outputs 9
* Console.WriteLine(myDouble); // Outputs 9

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

Console.WriteLine(myInt); // Outputs 9

Console.WriteLine(myDouble); // Outputs 9

## Type Conversion Methods

It is also possible to convert data types explicitly by using built-in methods, such as Convert.ToBoolean, Convert.ToDouble, Convert.ToString, Convert.ToInt32 (int) and Convert.ToInt64 (long):

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

10  
10  
5  
True

# **C# Operators**

## Arithmetic Operators

1. + =Addition
2. - = Substraction
3. \* = Multiplication
4. / = Division
5. % = Modulous - Returns the division remainder

using System;

namespace MyApplication

{

class Program

{

static void Main(string[] args)

{

int x = 5;

int y = 2;

Console.WriteLine(x % y);

}

}

}

**Output : 1**

* **Assignment Operator**

= , += ,-= ,\*=, /= , %= , &= , |=

* **Comparison Operator**
* == Equal to x == y
* ! = Not equal x != y
* > Greater than x > y
* < Less thanx < y
* >= Greater than or equal to x >= y
* <= Less than or equal tox <= y

**Logical Operator**

&& = Logical and Returns True if both statements are true x < 5 &&  x < 10

|| = Logical or Returns True if one of the statements is true x < 5 || x < 4

!= Logical not Reverse the result, returns False if the result is true !(x < 5 && x < 10)

## C# Strings

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

string greeting = "Hello";

## String Concatenation

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

string firstName = "John ";

string lastName = "Doe";

string name = firstName + lastName;

Console.WriteLine(name);

OUTPUT : - Jhon Doe

## String Interpolation

Another option of [string concatenation](https://www.w3schools.com/cs/cs_strings_concat.php), 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:

string firstName = "John";

string lastName = "Doe";

string name = $"My full name is: {firstName} {lastName}";

Console.WriteLine(name);

OUTPUT : - Jhon Doe

## Strings - Special Characters (escape sequence character)

\’ = ‘ - the single quote will reflect around the string

\” = “ - the Double quote will reflect around the string

\\ = \ - it will reflect a back slash

\n = for new line

\t = for tab

\b = for backspace

## Boolean Values

bool isCSharpFun = true;

bool isFishTasty = false;

Console.WriteLine(isCSharpFun); // Outputs True

Console.WriteLine(isFishTasty); // Outputs False

A Boolean expression returns a boolean value: True or False, by comparing values/variables.

int x = 10;

int y = 9;

Console.WriteLine(x > y); // returns True, because 10 is higher than 9

# **C# If ... Else**

* Use if to specify a block of code to be executed, if a specified condition is true
* Use else to specify a block of code to be executed, if the same condition is false
* Use else if to specify a new condition to test, if the first condition is false

# **C# Short Hand If...Else**

There is also a short-hand if else, which is known as the **ternary operator** because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements:

### **Syntax**

variable *= (*condition*) ?* expressionTrue *:*  expressionFalse*;*

int time = 20;

string result = (time < 18) ? "Good day." : "Good evening.";

Console.WriteLine(result);

Output: - Good Evening

## C# Switch Statements

* 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
* When C# reaches a break keyword, it breaks out of the switch block.
* The default keyword is optional and specifies some code to run if there is no case match:

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;

}

// Outputs "Looking forward to the Weekend.

## Loops

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

int i = 0;

while (i < 5)

{

Console.WriteLine(i);

i++;

}

## 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.

int i = 0;  
do

{

Console.WriteLine(i);

i++;

}

while (i < 5)

If the Condition in become false in this case the code will run atleast one time .

## 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:

**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.

for (int i = 0; i < 5; i++)

{

Console.WriteLine(i);

}

Output : 0 to 4

# **C# Arrays**

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**:

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**:

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

Console.WriteLine(cars[0]);

// Outputs Volvo

## Change an Array Element

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

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

cars[0] = "Opel";

Console.WriteLine(cars[0]);

// Now outputs Opel instead of Volvo

## Array Length

To find out how many elements an array has, use the Length property:

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

Console.WriteLine(cars.Length);

// Outputs 4

## Other Ways to Create an Array

// Create an array of four elements, and add values later

string[] cars = new string[4];

// Create an array of four elements and add values right away

string[] cars = new string[4] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements without specifying the size

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

// Create an array of four elements, omitting the new keyword, and without specifying the size

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

Loop Through an Array

You can loop through the array elements with the for loop, and use the Length property to specify how many times the loop should run.

The following example outputs all elements in the **cars** array:

**Example**

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

for (int i = 0; i < cars.Length; i++)

{

Console.WriteLine(cars[i]);

}

## Sort an Array

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

using System;

namespace MyApplication

{

class Program

{

static void Main(string[] args)

{

// 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);

}

}

}

}

Output :

BMW

Ford

Mazda

Volvo

1

5

8

9

## System.Linq Namespace

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

using System;

using System.Linq;

namespace MyApplication

{

class Program

{

static void Main(string[] args)

{

int[] myNumbers = {5, 1, 8, 9};

Console.WriteLine(myNumbers.Max()); // largest value

Console.WriteLine(myNumbers.Min()); // smallest value

Console.WriteLine(myNumbers.Sum()); // sum of myNumbers

}

}

}

## 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:

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

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

## 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:

using System;

namespace MyApplication

{

class Program

{

static void Main(string[] args)

{

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

Console.WriteLine(numbers[0, 2]);

}

}

}

## Change Elements of a 2D Array

using System;

namespace MyApplication

{

class Program

{

static void Main(string[] args)

{

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

numbers[0, 0] = 5;

Console.WriteLine(numbers[0, 0]);

}

}

}

1 will changed by 5

## Loop Through a 2D Array

using System;

namespace MyApplication

{

class Program

{

static void Main(string[] args)

{

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

foreach (int i in numbers)

{

Console.WriteLine(i);

}

}

}

}

1  
4  
2  
3  
6  
8

You can also use a [for loop](https://www.w3schools.com/cs/cs_for_loop.php). For multidimensional arrays, you need one loop for each of the array's dimensions.

Also note that we have to use GetLength() instead of Length to specify how many times the loop should run:

using System;

namespace MyApplication

{

class Program

{

static void Main(string[] args)

{

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]);

}

}

}

}

}

Explanation:

The outer for loop iterates over the rows of the numbers array.

numbers.GetLength(0) returns the number of rows in the array(which is 2).

i is the row index, starting from 0 and increasing until it is less than the number of rows.

The inner for loop iterates over the columns of the current row.

numbers.GetLength(1) returns the number of columns in the array (which is 3).

j is the column index, starting from 0 and increasing until it is less than the number of columns.

Inside the inner loop, Console.WriteLine(numbers[i, j]) prints the element at the current row i and column j to the console.

# **C# Methods**

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

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

Methods are used to perform certain actions, and they are also known as **functions**.

Why use methods? To reuse code: define the code once, and use it many times.

Class program

{

Static void MyMethod()

{

// Code to be Execute

}

}

MyMethod() is the name of the method

Static = it means the method belongs to the program class not an object of the program class.

Void = means that the method doesn’t have any return value

## Call a Method

To call (execute) a method, write the method's name followed by two parentheses **()** and a semicolon**;**

In the following example, MyMethod() is used to print a text (the action), when it is called:

using System;

namespace MyApplication

{

class Program

{

static void MyMethod()

{

Console.WriteLine("I just got executed!");

}

static void Main(string[] args)

{

MyMethod();

}

}

}

Output :- I just got executed!

A method can be called multiple times:

## 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:

static void MyMethod(string fname)

{

Console.WriteLine(fname + " Refsnes");

}

static void Main(string[] args)

{

MyMethod("Liam");

MyMethod("Jenny");

MyMethod("Anja");

}

Output :=

// Liam Refsnes

// Jenny Refsnes

// Anja Refsnes

When a **parameter** is passed to the method, it is called an **argument**. So, from the example above: fname is a **parameter**, while Liam, Jenny and Anja are **arguments**.

## Multiple Parameters

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

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);

}

// Liam is 5

// Jenny is 8

// Anja is 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"):

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

## Return Values

In the [previous page](https://www.w3schools.com/cs/cs_method_parameters.php), 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:

static int MyMethod(int x)

{

return 5 + x;

}

static void Main(string[] args)

{

Console.WriteLine(MyMethod(3));

}

// Outputs 8 (5 + 3)

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

static int MyMethod(int x, int y)

{

return x + y;

}

static void Main(string[] args)

{

Console.WriteLine(MyMethod(5, 3));

}

// 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)

## Named Arguments

It is also possible to send arguments with the key: value syntax.

That way, the order of the arguments does not matter:

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

## Method Overloading

int MyMethod(int x)

float MyMethod(float x)

double MyMethod(double x, double y)

example

using System;

namespace MyApplication

{

class Program

{

static int PlusMethod(int x, int y)

{

return x + y;

}

static double PlusMethod(double x, double y)

{

return x + y;

}

static void Main(string[] args)

{

int myNum1 = PlusMethod(8, 5);

double myNum2 = PlusMethod(4.3, 6.26);

Console.WriteLine("Int: " + myNum1);

Console.WriteLine("Double: " + myNum2);

}

}

}

Int: 13  
Double: 10.559999999999999

**OOP Concept in C#**

## Classes and Objects

A Class is like an object constructor, or a "blueprint" for creating objects.

class Car

{

string color = "red";

}

When a variable is declared directly in a class, it is often referred to as a **field** (or attribute).

An object is created from a class. We have already created the class named Car, so now we can use this to create objects.

To create an object of Car, specify the class name, followed by the object name, and use the keyword new

Create an object called "myObj" and use it to print the value of color:

class Car

{

string color = "red";

static void Main(string[] args)

{

Car **myObj** = new Car();

Console.WriteLine(myObj.color);

}

}

Note that we use the dot syntax (.) to access variables/fields inside a class (myObj.color). You will learn more about fields in the next chapter.

 Car myObj = new Car(); creates an instance of the Car class.

a class is a blueprint for objects. It defines what attributes and behaviours the objects created from it will have.

An instance (or object) is a specific realization of a class. Each instance can have its own state (values for the attributes defined by the class).

//filename: Car.cs

//filename: Car.cs

using System;

​

namespace MyApplication

{

 class Car

{

   string model;

   string color;

   int year;

​

   static void Main(string[] args)

  {

     Car Ford = new Car();

     Ford.model = "Mustang";

     Ford.color = "red";

     Ford.year = 1969;

​

     Car Opel = new Car();

     Opel.model = "Astra";

     Opel.color = "white";

     Opel.year = 2005;

​

     Console.WriteLine(Ford.model);

     Console.WriteLine(Opel.model);

  }

}

}

Mustang  
Astra

# **C# 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:

The code without the constructor

using System;

namespace MyApplication

{

// Create a Person class

class Person

{

public string name; // Create a field for the name

public int age; // Create a field for the age

static void Main(string[] args)

{

// Create an object of the Person class

Person person1 = new Person();

// Set the values of the fields

person1.name = "Alice";

person1.age = 30;

// Print the values of the fields

Console.WriteLine("Name: " + person1.name);

Console.WriteLine("Age: " + person1.age);

}

}

}

You can have as many parameters as you want:

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

# **C# Properties (Get and Set)**

## Properties and 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:

class Person

{

private string name; // field

public string Name // property

{

get { return name; } // get method

set { name = value; } // set method

}

}

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.

We can provide control access by using the get & set method

Encapsulation Using Set & Get method properties

using System;

namespace MyApplication

{

class Person

{

private string name; // field

public string Name // property

{

get { return name; }

set { name = value; }

}

}

}

using System;

namespace MyApplication

{

class Program

{

static void Main(string[] args)

{

Person myObj = new Person();

myObj.Name = "Liam";

Console.WriteLine(myObj.Name);

}

}

}

**Another Example of Encapsulation**

It involves bundling the data (fields) and methods (functions) that operate on the data into a single unit, typically a class, and restricting direct access to some of the object's components, which is often accomplished by making the fields private and providing public methods to access and modify those fields.

using System;

namespace MyApplication

{

// Create a Person class

class Person

{

// Private fields

private string name;

private int age;

// Public method to set the name

public void SetName(string personName)

{

name = personName;

}

// Public method to get the name

public string GetName()

{

return name;

}

// Public method to set the age

public void SetAge(int personAge)

{

// You can add validation logic here

if (personAge > 0)

{

age = personAge;

}

else

{

Console.WriteLine("Please enter a valid age.");

}

}

// Public method to get the age

public int GetAge()

{

return age;

}

static void Main(string[] args)

{

// Create an object of the Person class

Person person1 = new Person();

// Set the values using public methods

person1.SetName("Alice");

person1.SetAge(30);

// Get and print the values using public methods

Console.WriteLine("Name: " + person1.GetName());

Console.WriteLine("Age: " + person1.GetAge());

}

}

}

**Example 2:**

{

class Account

{

Int accountBalnace = 1000;

Public setBalance(int amount)

{

accountBalance = amount;

}

}

Class program

{

Static void Mian(string[] args)

{

Account myaccount = new Account();

Myaccount.SetBalance(1000);

}

}

}

## Inheritance (Derived and Base Class)

* **Derived Class** (child) - the class that inherits from another class
* **Base Class** (parent) - the class being inherited from

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);

}

## Polymorphism and Overriding Methods

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

class Animal // Base class (parent)

{

public void animalSound()

{

Console.WriteLine("The animal makes a sound");

}

}

class Pig : Animal // Derived class (child)

{

public void animalSound()

{

Console.WriteLine("The pig says: wee wee");

}

}

class Dog : Animal // Derived class (child)

{

public void animalSound()

{

Console.WriteLine("The dog says: bow wow");

}

}

class Program

{

static void Main(string[] args)

{

Animal myAnimal = new Animal(); // Create a Animal object

Animal myPig = new Pig(); // Create a Pig object

Animal myDog = new Dog(); // Create a Dog object

myAnimal.animalSound();

myPig.animalSound();

myDog.animalSound();

}

}

The output will be:

The animal makes a sound  
The animal makes a sound  
The animal makes a sound

The output from the example above was probably not what you expected. That is because the base class method overrides the derived class method, when they share the same name.

However, C# provides an option to override the base class method, by adding the virtual keyword to the method inside the base class, and by using the override keyword for each derived class methods:

class Animal // Base class (parent)

{

public **virtual** void animalSound()

{

Console.WriteLine("The animal makes a sound");

}

}

class Pig : Animal // Derived class (child)

{

public **override** void animalSound()

{

Console.WriteLine("The pig says: wee wee");

}

}

class Dog : Animal // Derived class (child)

{

public **override** void animalSound()

{

Console.WriteLine("The dog says: bow wow");

}

}

class Program

{

static void Main(string[] args)

{

Animal myAnimal = new Animal(); // Create a Animal object

Animal myPig = new Pig(); // Create a Pig object

Animal myDog = new Dog(); // Create a Dog object

myAnimal.animalSound();

myPig.animalSound();

myDog.animalSound();

}

}

The output will be:

The animal makes a sound  
The pig says: wee wee  
The dog says: bow wow

[Try it Yourself »](https://www.w3schools.com/cs/trycs.php?filename=demo_polymorphism2)

#### **Why And When To Use "Inheritance" and "Polymorphism"?**

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

# **C# Abstraction**

## 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**](https://www.w3schools.com/cs/cs_interface.php) (which you will learn more about in the next chapter).

* **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).
* An abstract class can have both abstract and regular methods:

abstract class Animal

{

public abstract void animalSound();

public void sleep()

{

Console.WriteLine("Zzz");

}

}

using System;

namespace MyApplication

{

// Abstract class

abstract class Animal

{

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep()

{

Console.WriteLine("Zzz");

}

}

// Derived class (inherit from Animal)

class Pig : Animal

{

public override void animalSound()

{

// The body of animalSound() is provided here

Console.WriteLine("The pig says: wee wee");

}

}

class Program

{

static void Main(string[] args)

{

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}

}

}

The pig says: wee wee  
Zzz

# **C# Interface**

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

It is considered good practice to start with the letter "I" at the beginning of an interface, as it makes it easier for yourself and others to remember that it is an interface and not a class.

By default, members of an interface are abstract and public.

**Note:** Interfaces can contain properties and methods, but not fields.

using System;

namespace MyApplication

{

// Interface

interface IAnimal

{

void animalSound(); // interface method (does not have a body)

}

// Pig "implements" the IAnimal interface

class Pig : IAnimal

{

public void animalSound()

{

// The body of animalSound() is provided here

Console.WriteLine("The pig says: wee wee");

}

}

class Program

{

static void Main(string[] args)

{

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

}

}

}

The pig says: wee wee

* 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)

#### **Why And When To Use Interfaces?**

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) C# does not support "multiple inheritance" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

# **C# Multiple Interfaces**

using System;

​

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();

  }

}

}

Some text..  
Some other text...

# **C# Enum**

An enum is a special "class" that represents a group of **constants** (unchangeable/read-only variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the enum items with a comma:

using System;

​

namespace MyApplication

Medium

{

 class Program

{

   enum Level

  {

     Low,

     Medium,

     High

  }

   static void Main(string[] args)

  {

     Level myVar = Level.Medium;

     Console.WriteLine(myVar);

  }

}

}

Medium

## Enum Values

By default, the first item of an enum has the value 0. The second has the value 1, and so on.

To get the integer value from an item, you must [explicitly convert](https://www.w3schools.com/cs/cs_type_casting.php) the item to an int:

Result Size: 753 x 575

using System;

​

namespace MyApplication

{

 class Program

{

   enum Months

  {

     January,    // 0

     February,   // 1

     March,      // 2

     April,      // 3

     May,        // 4

     June,       // 5

     July        // 6

  }

   static void Main(string[] args)

  {

     int myNum = (int) Months.April;

     Console.WriteLine(myNum);

  }

}

}

3

## 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

{

// *Block of code to try*

}

catch (Exception e)

{

// *Block of code to handle errors*

}

## The throw keyword

The throw statement allows you to create a custom error.

The throw statement is used together with an **exception class**. There are many exception classes available in C#: ArithmeticException, FileNotFoundException, IndexOutOfRangeException, TimeOutException, etc:

using System;

​

namespace MyApplication

{

 class Program

{

   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(20);

  }

}

}

Access granted - You are old enough!