Learning C#

- Starting with demo code

```
using System;

namespace Demo
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("This is a demo application!!");
    }
  }
}
```

> ***From here marks the start of the course to learn C Sharp from W3 Schools***

Introduction

- C Sharp is a object oriented programming language created by Microsoft and it is based on .NET Framework
- It has roots with other languages like C,C++ and JAVA

- Can be used for a lot of things

Getting Started

- Use an IDE like Visual Studio
- Start a new project of Console App
- Use F5 for debugging

C# Syntax

- We can look at a demo applicaiton like

```
using System;

namespace demo
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("This is a demo application");
    }
  }
}
```

- **Line 1:** `using System` means that we can use classes from the `System` namespace.
- **Line 2:** A blank line. C# ignores white space. However, multiple lines makes the code more readable.
- **Line 3:** `namespace` is used to organize your code, and it is a container for classes and other namespaces.
- **Line 4:** The curly braces `{}` marks the beginning and the end of a block of code.
- **Line 5:** `class` is a container for data and methods, which brings functionality to your program. Every line of code that runs in C# must be inside a class. In our example, we named the class Program.
- Don't worry if you don't understand how `using System`, `namespace` and `class` works. Just think of it as something that (almost) always appears in your program, and that you will learn more about them in a later chapter.
- **Line 7:** Another thing that always appear in a C# program, is the `Main` method. Any code inside its curly brackets `{}` will be executed. You don't have to understand the keywords before and after Main. You will get to know them bit by bit while reading this tutorial.
- **Line 9:** `Console` is a class of the `System` namespace, which has a `WriteLine()` method that is used to output/print text. In our example it will output "Hello World!".

- If you omit the `using System` line, you would have to write `System.Console.WriteLine()` to print/output text.
- **Note:** Every C# statement ends with a semicolon `;`.
- **Note:** C# is case-sensitive: "MyClass" and "myclass" has different meaning.
- **Note:** Unlike Java, the name of the C# file does not have to match the class name, but they often do (for better organization). When saving the file, save it using a proper name and add ".cs" to the end of the filename. To run the example above on your computer, make sure that C# is properly installed. The output should be: `Hello World!`

C# Output

- We can use `WriteLine` to output on the console
- We can add multiple `WriteLine`

```
Console.WriteLine("Hey!");
Console.WriteLine("How u doin?");
Console.WriteLine("This looks fun!");
```

- This will add text to new line
- We can also use the `Write` method which does not add to the new line

```
Console.Write("Hey!");
Console.Write("How u doin?");
Console.Write("This looks fun!");
```

- We can also use math expressions in it

```
Console.WriteLine(984 + 353);
Console.WriteLine(2000 - 663);
Console.WriteLine(10 / 5);
Console.WriteLine(100 * 5);
```

Comments

- We can add comments to our code to make it more readable and understandable
- Single line comments work with `//`
- Multiple line comment start with `/*` and end with `*/`

```
using System;

namespace demo
{
  class Program
  {
    /*
    This
    is
    a
    multi
    line
    comment
    */
    static void Main(string[] args)
    {
      // This is a line comment
      Console.WriteLine("This is a demo c sharp application!!");
    }
  }
}
```

C# Variables

- In C# we can use different types of variables
  - `int` is used to store integers
  - `double` is used to store decimal numbers
  - `bool` is used to store two states: true or false
  - `char` is used to store charaters like A or B. They are in single quotes.
  - `string` is used to store strings like Hello World. They are in double quotes.
- Syntax is

```
type variableName = value;
```

- We can use it like

```
string name = "Ace";
Console.WriteLine(name);

int number = 1337;
Console.WriteLine(number);

double dec = 13.37;
Console.WriteLine(dec);

char a = 'a';
Console.WriteLine(a);

bool t = true;
Console.WriteLine(t);
```

- We can also assign a variable first and give it value later

```
int myNum;
myNum = 15;
Console.WriteLine(myNum);
```

- We can also overwrite the assigned variables

```
string demo = "Demo 0";
demo = "Demo 1";
Console.WriteLine(demo); // This will print "Demo 1"
```

- If we want to store the variables constantly with the same value we can use `const`

```
const string testing = "This is a test";
string testing = "Oh no"; // IDE gives error
```

- We can also append variables to `WriteLine`

```
string demoName = "Ace";
Console.WriteLine("Hey there" + demoName);
```

- We can append multiple variables together as well

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

- We can also use it with other variables to solve problems

```
int a = 10;
int b = 5;
Console.WriteLine(a + b);
```

- We can assign multiple variable same values

```
int x, y, z;
x = y = z = 50
Console.WriteLine ( x + y + z );
```

Identifiers

- The identifiers for variables should be unique and easy to understand
- Like `minutesPerHour` instead of `m`

- Some rules:
  - Names can contain letters, digits and the underscore character (_)
  - Names must begin with a letter
  - Names should start with a lowercase letter and it cannot contain whitespace
  - Names are case sensitive ("myVar" and "myvar" are different variables)
  - Reserved words (like C# keywords, such as `int` or `double`) cannot be used as names

---

Data Types

- There are specific data types we can use

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 quotes

Numbers

- The number types are divided into two groups
  - Integer: This contains `int` and `long`
  - Floating: This contains `double` and `float`

Integer Types

- The `int` type is used the most
- It can store numbers from -2147483648 to 2147483647
- We can use `long` for values more than what `int` can store
- It can store numbers from -9223372036854775808 to 9223372036854775807
- The value should end with L

```
long myNum = 15000000000L;
Console.WriteLine(myNum);
```

Floating Point Types

- The `double` type is used the most
- We can use `flat` to store more values than `double`
- The value should end with F or D respectively
- We can also use scientific notations like e to indicate a power of 10

```
float f1 = 35e3F;
double d1 = 12E4D;
Console.WriteLine(f1); // 35000
Console.WriteLine(d1); // 120000
```

Boolean

- Boolean data type outputs either True or False

```
bool isCSharpFun = true;
bool isFishTasty = false;
Console.WriteLine(isCSharpFun);   // Outputs True
Console.WriteLine(isFishTasty);   // Outputs False
```

- They are mostly used for conditional testing

Characters

- A character data type is used to store a single character
- Like `A` or `B` etc
- They are enclosed in single quotes

```
char grade = A;
Console.WriteLine(grade);
```

Strings

- We can use string data type write strings
- They are enclosed in double quotes

```
string name = "Ace";
Console.WriteLine(name);
```

Type Casting

- When you assign a value of one data type to another data type
- There are two types of casting we can do
  - Implicit Casting: (Automatically) Converting small to larger type size
    - `char` -> `int` -> `long` -> `float` -> `double`
  - Explicit Casting: (Manually) Converting large to smaller type size
    - `double` -> `float` -> `long` -> `int` -> `char`

Implicit Casting

- It is used to automatically pass smaller size type to larger size type

```
int myNum = 50;
double myDouble = myNum;

Console.WriteLine(myNum);
Console.WriteLine(myDouble);
```

Explicit Casting

- It is used to manually pass large sized types to small size

```
double myDouble = 13.37;
int myNum = (int) myDouble;

Console.WriteLine(myNum);
Console.WriteLine(myDouble);
```

Other Type Conversions

- We can use other type conversions as well
- `Convert.ToBoolean`
- `Convert.ToDouble`
- `Convert.ToString`
- `Convert.ToInt32` (`int`)
- `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
```

User Input

- We can use `Console.ReadLine()` for user input

```
Console.WriteLine("Enter username: ");
string username = Console.ReadLine();
Console.WriteLine("Welcome " + username);
```

- We can change input types also

```
Console.WriteLine("Whats your age?: ");
string age = Console.ReadLine();
int ageNum = Convert.ToInt32(age);
Console.WriteLine("You are " + ageNum + " years old.");
```

Operators

- We can use operators to perform further operations on the input

```
int sum = 1000 + 30;
int sum2 = sum + 300;
int sum3 = sum2 + 7;
Console.WriteLine(sum3); // output = 1337
```

- We can use these operators

Name		**Description**	**Example**
+	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

- We can use these assignment operators

Operator	**Example**	**Same As**
=	x = 5	x = 5

+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Comparison Operators

- We can use these comparison operators

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators

- We can use these logical operators

Operator	Name	Description	Example
&&	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)

Math

- We can use the Math class in C# which has many methods
- We can use `Math.Max` to find the greater number from two

```
int bigger = Math.Max(5 ,10);
Console.WriteLine(bigger);
```

- We can use `Math.Min` to find the smaller number from two

```
int smaller = Math.Min(5 ,10);
Console.WriteLine(smaller);
```

- We can use `Math.Sqrt` to find square root of the number

```
double square = Math.Sqrt(64);
Console.WriteLine(square);
```

- We can use `Math.Abs` to find the absolute positive value of a number

```
int positive = Math.Abs(-1337);
Console.WriteLine(positive);
```

- We can use `Math.Round` to round off numbers

```
int rounded = Math.Round(10.79);
Console.WriteLine(rounded);
```

Strings

- Strings are used to store text
- We can get string length by using `Length`

```
string a = "ABCDEFGHIJKLMNOPQRESTUVWXYZ";
Console.WriteLine("The string length is " + a.Length);
```

- We can use `ToUpper` and `ToLower` to convert cases

```
string a = "Hey there!";
Console.WriteLine(a.ToUpper());
Console.WriteLine(a.ToLower());
```

- We can concatenate strings as well

```
string firstname = "Crypt0";
string lastname = "Ace";
string fullname = firstname + lastname;
Console.WriteLine("Welcome " + fullname);

// Can also use this to concatenate strings
Console.WriteLine(string.Concat(firstname + lastname));
```

- Strings get concatenated and numbers get added

```
string num1 = "10";
string num2 = "20";
string concat = num1 + num2;
Console.WriteLine(concat); // 1020

in num3 = 10;
in num4 = 20;
int added = num3 + num4;
Console.WriteLine(added); // 30
```

- We can also use string Interpolation

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

- We can access strings as well to print out individual characters

```
string text = "Hey! Im Ace. Nice to meet you!";
Console.WriteLine(text[0]); // H
Console.WriteLine(text[13]); // N
```

- We can also find index of characters

```
string text = "Hey! Im Ace. Nice to meet you!";
Console.WriteLine(myString.IndexOf("t"));
```

- We can use `Substring()` to print the string
- It will find the character position first and then we can use substring to print text from that character position

```
int charpos = text.IndexOf("N");
string good = text.Substring(charpos);
Console.WriteLine(good);
```

- We can use special characters too

```
string test = "This is a test \" of\" special chatacters"
string txt = "The character \\ is called backslash.";
```

- We can use these special characters

Escape character	**Result**	\'
\’	'	Single quote
\"	"	Double quote
\\	\	Backslash

- Other userdul characters are

Code	Result
\n	New Line
\t	Tab
b	Backspace

Booleans

- C# can give out boolean results for True and False

```
bool isCSharpFun = true;
bool isFishTasty = false;
Console.WriteLine(isCSharpFun);   // Outputs True
Console.WriteLine(isFishTasty);   // Outputs False
```

- We can also use boolean expressions

```
int x = 20;
int y = 15;
Console.WriteLine(x > y); // True
Console.WriteLine(y > x); // False
Console.WriteLine( 10 > 5); // True
Console.WriteLine(x == 20); // True
```

If Else Conditions

- We can use these conditional statements in C#
    - 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
    - Use `switch` to specify many alternative blocks of code to be executed
- We can use the `if` statement

```
if (20 > 18)
{
  Console.WriteLine("20 is greater than 18");
}
```

- We can also test variables

```
int x = 20;
int y = 18;
if (x > y)
{
  Console.WriteLine("x is greater than y");
}
```

- We can use `else` statement to print out different output for false statements

```
int time = 20;
if (time < 18)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
// Outputs "Good evening."
```

- A short way to do it can be

```
int time = 20;
string result = (time < 18) ? "Good day." : "Good evening.";
Console.WriteLine(result);
```

- We can also add `else if`

```
int time = 22;
if (time < 10)
{
  Console.WriteLine("Good morning.");
}
else if (time < 20)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
// Outputs "Good evening."
```

Switch

- We can use `switch` to have multiple blocks of code execute

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

- It works as
  - 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
- For 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)
```

- `break` breaks out of code execution
- We can add a `default` as well for default code to run when no case matches

- Another example can be

```
Console.WriteLine("What time is it?");
string user_time = Console.ReadLine();
int time_for_user = Convert.ToInt32(user_time);

switch (time_for_user)
  {
  case 9:
    Console.WriteLine("Good Morning!");
    break;
  case 12:
    Console.WriteLine("Good Evening!");
    break;
  case 6:
    Console.WriteLine("Good Night!");
    break;
  default:
    Console.WriteLine("Have a good day!");
    break;
```

Loops

While Loop

- We can use while loop to run code until the condition is met

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

- For example

```
int i = 0;
while (i < 5)
{
    Console.WriteLine("Not yet " + i);
    i++;
}
```

Do/While Loop

- We can use a Do/While Loop for code to run atleast once before checking if the statement is true

```
int a = 5;
do
{
  Console.WriteLine(a);
  a++;
}
while(a < 5); // output = 5
```

For Loop

- When you know how many times you want to loop through a code you can use for loop

```
for (statement 1; statement 2; statement 3)
{
  // code to execute
}
```

- Here
    - **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 example

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

For/Each Loop

- We can execute code from arrays as well

```
string[] days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};
foreach (string d in days)
{
  Console.WriteLine(d);
}
```

Break/Continue

- We can also use break to jump out of a loop

```
for (int e = 0; e < 10; e++);
{
  if (e == 5)
  {
    break;
  }
}
```

- We can use break or continue in a while loop as well

```
int i = 0;
while (i < 10)
{
  Console.WriteLine(i);
  i++
  for (i == 5)
  {
    break;
  }
}
```

Arrays

- Arrays are used to store multiple values in a single variable
- It is declared with a square bracket

```
string[] days;
```

- We can add values to it

```
string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
Console.WriteLine(days);
Console.WriteLine(days[6]); // Output = Sunday
```

- We can change the value of specific array object as well

```
string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
days[0] = "test";
Console.WriteLine(days[0]); // Output = test
```

- We can get the array length as well

```
string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
Console.WriteLine(days.Length); // Output = 7
```

- We can loop through arrays as well

```
string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
for (int i = 0; i < days.Length; i++)
{
  Console.WriteLine(days[i]);
}
```

- We can use foreach loop as well

```
string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
foreach (string x in days)
{
  Console.WriteLine(x);
}
```

- We can sort th e list alphabetically or ascending to decending with `sort`

```
string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
Array.Sort(days);
foreach (string y in days)
{
  Console.WriteLine(y);
}
```

- Other methods of `Min` `Max` etc can be done using `System.Linq` namespace

```
int[] myNum = {1, 2, 3, 4, 5};
Console.WriteLine(myNum.Max());  // returns the largest value
Console.WriteLine(myNum.Min());  // returns the smallest value
Console.WriteLine(myNum.Sum());  // returns the sum of elements
```

- There are other ways to create arrays as well

```
string[] cars;
cars = new string[] {"Audi", "BMW", "Cross", "Tesla"}

// Other methods
// 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"};
```

Methods

- Methods are blocks of code that can be reused
- We can pass data into methods and actions which a method does are known as functions
- Some methods are predefined such as `Main()`
- For example

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

- Here
  - `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
- We can call a method using

```
static void MyMethod()
{
  Console.WriteLine("Hey there!");
}

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

Parameters and Arguments

- We can use parameters to store ainformation and pass them to methods
- We can add as many parameters we want inside the method

```
static void Main(string[] args)
{
  Demo("Hello ");
}
static void Demo(string greetings)
{
  Console.WriteLine(greetings + "Ahmed!");
}
```

- In this code snippet, greetings is a parameter and hello is the argument
- We can use default values and call them as well

```
static void Country(string = "United Kingdom")
{
  Console.WriteLine("This is from: ");
}

string void Main(string[] args)
{
  Country("USA");
  Country("Italy");
  Country(); // Output = This is from: United Kingdom
  Country("Canada");
}
```

- We can add multiple values in parameters

```
static void more(string firstName, int age)
{
  Console.WriteLine("This is " + firstName + ". He is " + age + " years old.");
}

static void Main(string[] args)
{
  more("Ace", 22);
  more("Kendal". 27);
}
```

Return Values

- We can use a return value to return some data instead of void

```
static int ans(int x)
{
  return(1300 + x);
}

static void Main(string[] args)
{
  Console.WriteLine(ans(37));
}
```

- We can also use named arguments to call

```
static void usernames(string name1, string name2, string name3)
{
  Console.WriteLine("This is " + name2);
}

static void Main(string[] args)
{
  names(name1: "John", name2: "Bob", name3: "Charles");
}
```

Method Overloading

- With this we can use multiple methods with same name but different values

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

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

static void Main(string[] args)
{
  int num1 = PlusNumber(10, 10);
  double num2 = PlusNumber(10.5, 3.6);
  Console.WriteLine(num1);
  Console.WriteLine(num2);
}
```

OOP (Object Oriented Programing)

- Classes and objects are two main points of OOP
- For example
  - Cars is a class and BMW, Audi are objects in the class
- All objects inherit the methods and variables from a class

Class

- 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.
- A Class is like an object constructor, or a "blueprint" for creating objects.
- We can create classes with `Class`

```
Class car
{
  string color = "red";
}
```

Objects

- We can create objects and multiple objects in the class

```
class demo
    {
        string test = "Testing";
        static void Main(string[] args)
        {
            demo newDemo = new demo();
            demo newDemo2 = new demo();
            Console.WriteLine(newDemo.test);
            Console.WriteLine(newDemo2.test);
        }
    }
```

- We can create a class and access in other classes as well
- This is done for better code management

```
//Program1.cs
class car
{
  public string name = "Audi";
}

// Program2.cs
class car
{
  car newObj = new car();
  Console.WriteLine(newObj.name);
}
```

Class Members

- Class members are whats inside the class. They are fields and methods

```
class New
{
  string test = "Test"; // Field
  int num = 50; // Field

  public void Method() // Method
  {
    New newObj = new New();
    Console.WriteLine("Hey!");
  }
}
```

- We can also access the fields in the method

```
class Car
    {
        string color = "Red";
        int speed = 200;
        static void Main(string[] args)
        {
            Car car = new Car();
            Console.WriteLine("This " + car.color + " colored car is going " + car.speed + " miles per hour.");
        }
    }
```

- We can also create empty fields and fill them when creating the object

```
class Car
{
  string color;
  int speed;

  static void Main(string[] args)
  {
    Car car = new Car();
    car.color = "Red";
    car.speed = 50;
    Console.WriteLine("This " + car.color + " colored car is going " + car.speed + " miles per hour.");
  }
}
```

- We can call methods in classess as well
- We create a public method which can be accesed by objects and static method can be accessed without creating object

```
class Car
    {
        string color = "Red";
        int speed = 200;

        public void Ticket()
        {
            Car car = new Car();
            Console.WriteLine("This " + car.color + " colored car is going " + car.speed + " miles per hour.");
        }

        static void Main(string[] args)
        {
            Car car = new Car();
            car.Ticket();
        }
    }
```

- We can also use multiple classes for better organized code like `Program1.cs` and `Program2.cs`

```
// Class.cs
class DemoClass
{
  public string name;
  public int age;

  public void intro()
  {
    Console.WriteLine("Hey there!");
  }
}

// Program.cs
class Program
{
  DemoClass demo = new DemoClass()
  demo.name = "Ace";
  demo.age = 22;
  static void Main()
  {
    Console.WriteLine("Hey! My name is " + demo.name + ". I'm " + demo.age + " years old.");
  }
}
```

Class Constructors

- We can use class constructors to initialize objects and call them later

```
class Car
{
  public string color;

  public Car()
  {
    string color = "Red";
  }
  static void Main(string[] args)
  {
    Car newObj = new Car();
    Console.WriteLine(newObj.color);
  }
}
```

- The classs constructor and class must have same name
- We can add many parameters to the constructors

```
class Intro
{
  string fName;
  string lName;
  int age;

  public greet(string first, string last, int ageNum)
  {
    fName = first;
    lName = last;
    age = ageNum;
  }

  static void Main(string[] args)
  {
    Intro newObj = new Intro("Crypt0", "Ace", "22");
    Console.WriteLine("Hey! My name is " + newObj.first + newObj.last + ". I'm" + newObj.ageNum + " years old.");
  }
}
```

- Constructors save a lot of time
- The difference between with constructors and without can be seen as

```
// Without constructors
class Program
{
  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);
  }
}

// With constructors
class Program
{
  static void Main(string[] args)
  {
    Car Ford = new Car("Mustang", "Red", 1969);
    Car Opel = new Car("Astra", "White", 2005);

    Console.WriteLine(Ford.model);
    Console.WriteLine(Opel.model);
  }
}
```

Access Modifiers

- We have seen the public access modifier
- There are other access modifiers that we use also

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

- For example

```
class Demo
{
  private string test = "Test";

  static void Main(string[] args)
  {
    Demo newObj = new Demo();
    Console.WriteLine(newObj.test);
  }
}
```

- We can access it within class
- If we try to access it outside of this class we get an error
- If no access modifier is mentioned it is always private

Properties

- Encapsulation is a way to to hide sensitive data from public
- This can be done with
  - declare fields/variables as `private`
  - provide `public` `get` and `set` methods, through **properties**, to access and update the value of a `private` field
- Properties can be used to access private variables
- They have two methods `get` and `set`

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

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

- And now we can access this class anywhere

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

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

class Program
{
  static void Main(string[] args)
  Intro newObj = new Intro();
  newObj.Name = "Ace";
  Console.WriteLine(newObj.Name);
}
```

- We can als ouse a shorter way

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

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

class Program
{
  static void Main(string[] args)
  Intro newObj = new Intro();
  newObj.Name = "Ace";
  Console.WriteLine(newObj.Name);
}
```

Inheritance

- We can inherit fields and methods from one class to another
- There are two types of inheritance
    - **Derived Class** (child) **-** the class that inherits from another class
    - **Base Class** (parent) **-** the class being inherited from
- Class inheritance is done with the use of `:`

```
class Access
{
  public string name = "Ace.";
  public void greet()
  {
    Console.WriteLine("Heyyaa!");
  }
}

class Intro : Access
{
  public string start = "Its nice to meet you";
}

class Program
{
  static void Main(string[] args)
  {
    Intro newObj = new Intro();
    newObj.greet();
    Console.WriteLine(newObj.start);
  }
}
```

- Inheritance can be used for code reuseablitiy

Polymorphism

- This situation occurs when we have many classes related to each other by inheritence

```
class AnimalSounds
    {
        public void Sounds()
        {
            Console.WriteLine("This is what these animals sound like!");
        }
    }

    class Dog : AnimalSounds
    {
        public void Sounds()
        {
            Console.WriteLine("Woof Woof");
        }
    }

    class Cat : AnimalSounds
    {
        public void Sounds()
        {
            Console.WriteLine("Meow Meow");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            AnimalSounds sounds = new AnimalSounds();
            Dog dog = new Dog();
            Cat cat = new Cat();
            sounds.Sounds();
            dog.Sounds();
            cat.Sounds();
        }
    }
```

- We can also use `virtual` and `override` to override the base class when the base class and derive class both use the same name

```
class AnimalSounds // base class
    {
        public virtual void Sounds()
        {
            Console.WriteLine("This is what these animals sound like!");
        }
    }

    class Dog : AnimalSounds // derived class
    {
        public override void Sounds()
        {
            Console.WriteLine("Woof Woof");
        }
    }

    class Cat : AnimalSounds // derived class
    {
        public override void Sounds()
        {
            Console.WriteLine("Meow Meow");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            AnimalSounds sounds = new AnimalSounds();
            AnimalSounds dog = new Dog();
            AnimalSounds cat = new Cat();
            sounds.Sounds();
            dog.Sounds();
            cat.Sounds();
        }
    }
```

Abstract Classes and Methods

- Data abstraction is hiding certain details from the user
- The `abstract` keyword can be 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).
- Absract class can have both in it

```
abstract class Demo
    {
        public abstract void Method();
        public void Test()
        {
            Console.WriteLine("This is a demo.");
        }
    }
```

- To access teh abstract class it must be inhertited from another class

```
abstract class Demo
    {
        public abstract void Method();
        public void Test()
        {
            Console.WriteLine("This is a demo.");
        }
    }

    class Access : Demo
    {
        public override void Method()
        {
            Console.WriteLine("Accessing abstract class from another class");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Access newObj = new Access();
            newObj.Test();
            newObj.Method();
        }
    }
```

Interfaces

- Another way to use abstraction is using interfaces
- It is an abstract class which has only methods and properties with no body

```
interface Animal
{
  void animalSound(); // interface method (does not have a body)
  void run(); // interface method (does not have a body)
}
```

- We can use them in

```
interface IDemo
    {
        void Test();
        void Method();
    }

    class Demo : IDemo
    {
        public void Test()
        {
            Console.WriteLine("This is a test");
        }

        public void Method()
        {
            Console.WriteLine("This is a demo");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Demo newObj = new Demo();
            newObj.Test();
            newObj.Method();
        }
    }
```

- 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)
- We can also use multiple interfaces

```
interface ITest
    {
        void Test();
    }

interface IDemo
    {
        void Method();
    }

    class Demo : ITest, IDemo
    {
        public void Test()
        {
            Console.WriteLine("This is a test");
        }

        public void Method()
        {
            Console.WriteLine("This is a demo");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Demo newObj = new Demo();
            newObj.Test();
            newObj.Method();
        }
    }
```

Enums

- Enums are a list of unchangeable constants
- We can create and access them like this

```
enum level
{
  low,
  medium,
  high
}

// Access them
level myVar = level.medium;
Console.WriteLine(myVar);
```

- Enums are used for values that are not going to change like days or months
- They are often used with the switch statements

```
class Program
    {
        enum Days
        {
            Monday,
            Tuesday,
            Wednesday,
            Thursday,
            Friday,
            Saturday,
            Sunday
        }

        static void Main()
        {
            Days dayNum = Days.Friday;

            switch (dayNum)
            {
                case Days.Monday:
                    Console.WriteLine("Today is Monday");
                    break;
                case Days.Tuesday:
                    Console.WriteLine("Today is Tuesday");
                    break;
                case Days.Wednesday:
                    Console.WriteLine("Today is Wednesday");
                    break;
                case Days.Thursday:
                    Console.WriteLine("Today is Thursday");
                    break;
                case Days.Friday:
                    Console.WriteLine("Today is Friday");
                    break;
                case Days.Saturday:
                    Console.WriteLine("Today is Saturday");
                    break;
                case Days.Sunday:
                    Console.WriteLine("Today is Sunday");
                    break;
                }
            }
        }
```

Working with Files

- We can work with files using the `System.IO` namespace
- The common methods we can use are

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.

To write to files and read them

- We can use

```
// To write
string text = "Hello World!";
File.WriteAllText("test.txt", text);

// To read
string whatWeWrote = File.ReadAllText("test.txt");
Console.WriteLine(whatWeWrote);
```

Exceptions

- Errors can occur in the code or user input
- We can use the `try` and `catch` statment to test the code and give out error
  - 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.

```
try
            {
                int[] myNum = { 1, 5, 10 };
                Console.WriteLine(myNum[5]);
            }
            catch (Exception e)
            {
                Console.WriteLine("ERROR!");
                throw;
            }
```

- We can add finally to execute the code even after the try and catch

```
try
            {
                int[] myNum = { 1, 5, 10 };
                Console.WriteLine(myNum[5]);
            }
            catch (Exception e)
            {
                Console.WriteLine("ERROR!");
                throw;
            }
            finally
            {
                Console.WriteLine("END");
            }
```

- We can use throw statements to give a custom error

```
static void checkAge(int age)
        {
            if (age > 18)
            {
                Console.WriteLine("Access Granted - You are old enough.");
            }
            else
            {
                throw new ArithmeticException("Access Denied - Not old enough.");
            }
        }
        static void Main(string[] args)
        {
            checkAge(15);
            checkAge(20);
        }
```