# **Basics**

* C# is a Programming language, **.**Net is a Framework for developing applications on Windows
* **.**Net consists of two important components

1. CLR( Common Language Runtime)
2. Class Library

* When you compile a C# code , the compiler converts it into an intermediate language code called IL Code, CLR is an application which converts the IL code into Machine code & the process is called Just In time Compilation
* WPF(Windows Presentation Foundation) Application/ Windows Form Application either of these are used to build Desktop applications for windows
* **.**Net application consists of Assembly(DLL dynamically linked libraries or Exe) which contains related Namespaces which in turn consists of related classes which are like classes in C++ which have data and methods
* C# is a **case sensitive** language , statements end with a **;** c# is a statically typed language
* A **namespace** allows you to organize your code in order to avoid class name collisions
* **Console** is a class in **System** namespace which consists of the **WriteLine** method which is a static method and can be used to print logs to console

Ex: **using System;**

**Console.WriteLine(**“HelloWorld!”**);**

Console.WriteLine(“{0},{1}”,x,y);

Convert.ToInt32(Console.ReadLine()); // To read a number from console and convert to int

* An EXE file represents a program that can be executed. A DLL is a file that includes code that can be re-used across different programs
* The commonly used data types in C# are

**bool**

**char**

**int**

**short**

**long**

**double**

**float**

**decimal**

**byte**

The equivalent of unsigned char of c/c++ in C# is **byte**

Equivalent of unsigned long of C/C++ is **uint** or **ulong** in c#.

* **checked** keyword is used to check for overflows

Ex:

checked

{

byte x = 255;

x =x+1; // this will throw an exception now

}

* Declaring variables in c# is done either by specifying the data type or with **var** keyword, if you use **var** keyword you do not need to specify the data type.
* Variables declared with var keyword cannot be null.
* To declare a constant in C# use the below syntax

Syntax : const <datatype> <name> = <value>;

const float PI = 3.14;

* C# supports Implicit typing , meaning you can use the **var** keyword and c# compiler would figure out what type you mean based on the assignment

Ex :

byte num =10;

int n =100;

var x = 20;

var y =”helloworld”;

float z = 20.5f;

* Implicit Type conversion has no data loss
* Explicit Type conversion or casting is done by placing the data type to be casted in a **()**

Ex :

int i =300;

byte b = **(**byte**)** i;

* For Type conversion of non compatible types like from string to int use Convert class or Parse method of primitive data type s like int, float, double every primitive data type has Parse method

Ex :

string s = “1”;

int i = Convert.ToInt32(s);

int j = int.Parse(s);

Convert has **ToByte()**, **ToInt16()**, **ToInt32()**, **ToInt64()**

* Its advisable to use **TryParse**(string in, out num) instead of Parse , as TryParse would return false in case the parsing failed.

Ex : string s =”128”

int num;

if ( int.TryParse(s, out num))

{

cw(“Parsing successful output is in{0}, num);

}

else {

}

* Operators in C# are

|  |  |  |  |
| --- | --- | --- | --- |
| Arithmetic | **+,-,\*, /,%, ++, --** |  |  |
| Comparison | **==**,**!=**, **>**, **>=**,**<**,**<=** | The result of a comparison operation is a boolean (True or False) |  |
| Assignment | **=,+=, -=, \*=, /=** |  |  |
| Logical | **&& , || ,!** |  |  |
| Bitwise | **& , |** |  |  |
| Unary |  | sign |  |
| Increment | **++** |  |  |

* **//** is single line comment
* **/\*** This is a Multi line

Comment

**\*/**

* Everything's an object in c#

Ex: you can do **int.MaxValue;**

* To declare a class in c# use the below syntax

Syntax:

<access specifier> **class** <classname>

{

<access modifier> <datatype> <varname>;

}

**public class** Person

{

public string FirstName;

public string SecondName;

public void Smefn()

{

Console.WriteLine(“Doing Smethng”);

}

}

Create objects using the below syntax

Person p1 = **new** Person();

We need to allocate memory using **new** keyword to each object, we don't need to deallocate the memory as C# ‘s Garbage collector will take care of it

* Structures in c# are defined as follows

Syntax :

<access specifier> **struct** <structname>

{

<access specifier> <datatype> <varname>;

}

* Arrays are declared as follows , the declared array needs to be allocated memory using **new** keyword

Syntax : <datatype>**[ ]** <arrayname> = **new** <datatype> **[** <sizeofarray> **];**

Ex : int[ ] arr = **new** int[3];

They can be initialized while creation using the below syntax

int[ ] a = **new** int[5] **{**1,2,3,4,5**}**;

If not initialized c# initialize its array elements with zero values of the data type of array

* Strings are put under double quotes **“**Hi**”** in C#
* Strings are concatenated using **+** operator in C#

Ex : var s = “Dhaval”;

string x = **string.Format(“{0} {1}”**, firstName, lastName**);**

Strings are Immutable in C#

Escape Character is **\**

Verbatim String is created by prefixing a string with **@**

Ex: string path = @”C:\projects\sme”;

* All primitive types are structures, Arrays and Strings are classes
* Structures are Value types , meaning memory is allocated on the stack and its lost as soon as the variable goes out of scope
* Classes are Reference types, meaning memory is allocated from the heap using new keyword , and no need to deallocate the memory. It will be taken care by Garbage Collector run by CLR
* If ( condition)

{

Sme statements;

}

else if (condition)

{

Sme other statements;

}

else

{

}

* **switch(**smething**)**

{

case x:

--

break;

case y:

---

break;

default:

Sme statements;

break;

}

* **for (** int i =0; i< 10; i++**)**

**{**

**}**

**foreach(**var num in numbers**)**

**{**

**}**

**while (** <condition>**)**

**{**

**}**

**do**

**{**

**}while(**<condition>**);**

* Random class is used to generate random numbers

Usage: var r = new **Random()**;

r.Next();

r.Next(stop);

r.Next(start, stop);

* **..** is the range operator

Ex : a[2..5]; // it includes index 2 to 4 .

* Number separat is **\_**

Ex : int d = 123456; // it is equal to int d = 123\_456;

float f = 1\_234.52;

var x = 0x AB\_CD\_EF; // is equal to var x = 0xabcdef;

var b = 0b1101\_1110\_0010\_0000;

* Null-coalescing operator **??**

Ex : var x = y ?? 0; // if y is not null it takes y in to x or else it sets x to zero.

Ex : string str = null;

str ??= “default value”;

cw(str); // default value

# **Keywords**

c# keywords are always lower case

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [**abstract**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/abstract) | [**as**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/as) | [**base**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/base) | [**bool**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/bool) | [**break**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/break) |
| [**byte**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/byte) | [**case**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/case) | [**catch**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/catch) | [**char**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/char) | [**checked**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/checked) |
| [**class**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/class) | [**const**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/const) | [**continue**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/continue) | [**decimal**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/decimal) | [**default**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/default) |
| [**delegate**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/delegate) | [**do**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/do) | [**double**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/double) | [**else**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/else) | [**enum**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/enum) |
| [**event**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/event) | [**explicit**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/explicit) | [**extern**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/extern) | [**false**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/false) | [**finally**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/finally) |
| [**fixed**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/fixed) | [**float**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/float) | [**for**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/for) | [**foreach**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/foreach) | [**goto**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/goto) |
| [**if**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/if) | [**implicit**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/implicit) | [**in**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/in) | [**int**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/int) | [**interface**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/interface) |
| [**internal**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/internal) | [**is**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/is) | [**lock**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/lock) | [**long**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/long) | [**namespace**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/namespace) |
| [**new**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/new) | [**null**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/null) | [**object**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/object) | [**operator**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/operator) | [**out**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/out) |
| [**override**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/override) | [**params**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/params) | [**private**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/private) | [**protected**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/protected) | [**public**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/public) |
| [**readonly**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/readonly) | [**ref**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/ref) | [**return**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/return) | [**sbyte**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/sbyte) | [**sealed**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/sealed) |
| [**short**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/short) | [**sizeof**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/sizeof) | [**stackalloc**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/stackalloc) | [**static**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/static) | [**string**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/string) |
| [**struct**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/struct) | [**switch**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/switch) | [**this**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/this) | [**throw**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/throw) | [**true**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/true) |
| [**try**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/try) | [**typeof**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/typeof) | [**uint**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/uint) | [**ulong**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/ulong) | [**unchecked**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/unchecked) |
| [**unsafe**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/unsafe) | [**ushort**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/ushort) | [**using**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/using) | [**var**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/var) | [**virtual**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/virtual) |
| [**void**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/void) | [**volatile**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/volatile) | [**while**](https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords/while) |  |  |

# **Arrays and Lists**

* Arrays are declared as follows , the declared array needs to be allocated memory using **new** keyword

int[ ] arr = **new** int[3];

They can be initialized while creation using the below syntax

int[ ] a = **new** int[5] **{**1,2,3,4,5**}**;

If not initialized c# initialize its array elements with zero values of the data type of array

If you do a[**^**1] it would fetch the last element of the array , its not 0 indexed from the last , only from the first

* Two types of Multi Dimensional Arrays are available in c#

1. Rectangular Multi Dimensional Array - Each Row has exact same number of columns

Ex : var m = new int [3,5]; // Declaring two dimensional rectangular array

var arr =new int[3,6,4]; // Declaring three dimensional rectangular array

1. Jagged Multidimensional Array

* No need to have same number of columns in each row , It is viewed as array of arrays , where each row is considered as a single dimensional array

Ex :

var arr = new int [3] [ ];

arr[0] = new int[4];

arr[1] = new int[5];

arr[2] = new int[3];

* All Arrays are objects of Array class
* Some properties of Array class are

1. **Length** - Returns the number of elements in the array

Ex : var num = new int[] {1,2,3,4,5};

**Console.WriteLine(**“Lenght is “ + num**.Length);**

b) **IndexOf** - Returns the index of the element specified in the parameter

Ex: var ix = **Array.IndexOf(**num, 3**);**

**Console.WriteLine(**“Index of 3 is” + ix);

c) **Array.Clear()**

d)**Array.Copy()**

e) **Array.Sort()**

f) **Array.Reverse()**

* Lists (Dynamic size Arrays) are created using the below syntax , To use Lists we need to include System.Collections.Generic namespace

Ex:

**using System.Collections.Generic**;

var n = new **List<**int**>();**

var l = new **List<**int**>() {**1,2,3,4**};**

var lt = new List<string>();

Some Useful Methods for List are

n.**Add**(<element>);

n.**AddRange**(<array or another list>);

n.**IndexOf**(<list element>); // Searches from the beginning of the list

n.**LastIndexOf**(<list element>); // Searches from the end of the list

n.**Count**; // returns the number of elements in the list

n.**Remove**(<element>); // Removes the first occurence of element

n.**Clear**(); // Clears the whole list and makes it empty

In C# its not allowed to modify collections inside a **foreach** loop

# **DateTime & TimeSpan Structures**

* DateTime is a structure having properties to know date, time, like in hours, minutes etc

var d = new **DateTime**(2019,1,19);

var nw = DateTime.now;

var tdy = DateTime.today;

cw(“Hour =” + nw.Hour);

cw(“Min = “ + nw.Minute);

DateTime & TimeSpan objects are immutable that is once we set them we cannot change them , we can use Add methods to add days, min, hour s etc

There are some functions to show the date and time in strings they are

nw.**ToLongDateString**();

nw.**ToShortDateString**();

nw.**ToLongTimeString**();

nw.**ToShortTimeString**();

nw.**ToString**(yy-MM-dd HH:mm“);

* TimeSpan objects denote the length of time or duration

There are three ways to create timespan object

1. var t = new TimeSpan(1,2,3);
2. var t1 = TimeSpan.FromHours(1);
3. Subtracting two DateTime Objects

Ex : var strt = DateTime.Now

var end = DateTime.Now.AddMinutes(10);

var duration = end-start;

* Properties of TimeSpan structure are

t.Minutes // returns 2

t.TotalMinutes // returns62.05

t.Add(TimeSpan.FromMinutes(8)) // returns another TimeSpan Object adding 8 minutes to current TimeSpan object

t.Subtract(TimeSpan.FromMinutes(3))

t.ToString()

# **Strings**

* Some of the important string functions in C# are

All string functions do not modify the existing string

To create a string from an array arr is

var s = new **string(**arr**)**;

a) **Trim()**,**TrimStart()**,**TrimEnd()** : TrimStart() removes whitespace only at the start , TrimEnd() removes white space only at the end

o/p Returns a new string

Ex : var s = “ Hi Hello “;

cw(“Time output is ‘{0}’” ,s**.Trim()**);

s**.ToUpper();**

s**.ToLower();**

s.IndexOf(‘ ‘)

s.**Substring**(<startindex>) // From start index till the end of string

s**.Substring**(<startindex>, <length>) // from start index to start index + length

s**.Split(**‘ ‘**)** // Returns an array of words

s**.Length**;

String.**IsNullOrWhiteSpace**(“ “)

String.**IsNullOrEmpty**(“ “)

String.Join(“ “ , <list of words to be joined>) // Joins a list of words by the separator specified in first argument

* **IndexOf**(“x”);

This string function returns the index of specified character or start of the string in any string , If the character or string is not found in the main string it returns -1 .

Helps in searching for a character with in a string

* String Formatter :

Ex : var city = “Melbourne”;

var temp = 18.1f;

var currentDt = DateTime.Now;

**string.Format**(“Welcome to {0} , the time is {1:t}, temp is {2:0.00}”, city , currentDt, temp);

* **StringBuilder** is a class defined in **System.Text** namespace and Represents a Mutable string, StringBuilder is not for searching
* The idea behind StringBuilder is that you instantiate it perhaps with some seed text, and then you build the string up over the course of several lines of code after that
* using StringBuilder feels easier and less error prone than string concatenation and it generally makes your code easier to read and maintain

Ex: var sb = new **StringBuilder()**;

sb.Append(“Hi”);

sb.Append(“ How are you? “);

sb.ToString(); // returns Hi How are you?

sb.AppendLine(“I am fine”);

* Some important functions of StringBuilder class are

**Append()**

**AppendLine()**

**Insert()**

**Remove()**

**Replace()**

**Clear()**

Ex:

using System.Text;

var b = new StringBuilder();

* string concatenation (+)
* string formatting ( {0) {1} ..)
* string interpolation ($)
* verbatim strings (@)

# **Yield**

* There are two use cases of yield key word

1. yield return; returns to the caller but preserves the iterator method location and executes from that location if the method is called again.

1. yield break; -

The yield break is different from break statement because break statement terminates the closest loop in normal method and yield break terminates the iterator method and transfers the control of the program to the caller.

yield break works just like return statement that returns nothing.

<https://www.programiz.com/csharp-programming/yield-keyword#:~:text=The%20yield%20keyword%20is%20used,yield%20break%20%2D%20terminates%20the%20iteration>

# **Data Structures**

## **Tuples:**

* Each value in a tuple is accessed by Itemn syntax

(double, double,double) GetStockValues(string cmp) {

if (cmp == “ABCD”) {

return (21.0,10.0,22.0);

}

}

Ex : var result = GetStockValues(“ABCD”);

Or you can simply do i.e deconstruction of a tuple

(x1,x2,x3) = GetStockValues(“ABCD”);

# **Classes**

* Classes has data and member functions
* Classes can have two types of class members

1. Instance : accessible from an object
2. Static: accessible from the class

Declaring static members is done by specifying **static** keyword after the access specifier

Ex : public static int x = 0;

* A constructor is a method of a class which is called when an instance of the class is created, to initialize the object

a) Constructors have no return type and named exactly as the class name

b) Constructors can be overloaded

c) In visual studio to write constructor code use **ctor** and then tab

d) There is default or parameter less constructor which is provided by the compiler when there is no user provided constructor

e) **this** keyword can be used to pass control from one constructor to another

Ex :

public Customer()

{

Orders = new List<Order>();

}

public Customer(int id):this()

{

this.Id = id;

}

f) **static** constructors

* **Object Initializer list** is a syntax for quickly initializing an object without the need to call one of its constructor ,to avoid creating multiple constructors

Syntax :

var p = new Person

**{**

FirstName =”Dhaval”;

SecondName= “Parikh”;

Age =32;

**};**

* When your method has varying number of arguments **params** modifier makes it easier for the consumer of that method to call
* **ref** modifier can be used to make a pass by value parameter to pass by reference , we need to specify the ref keyword both at the caller and in the definition of the method
* **out** modifier is something similar to ref , but here out will make the parameter by value and we can return it too
* **readonly** modifier makes the fields of class to be initialized only once either directly at the time of declaration or in one of the constructors
* There are five access modifiers in C#

**public**

**private**

**Protected** // accessible only from the class & its derived classes

**internal**// often used with the class not their members, accessible only in the same class library or assembly , **its the default access specifier**

**protected internal** // accessible from the same assembly or any of the derived classes

* Object Oriented Programming has three concepts

1. Encapsulation/Information Hiding
2. Inheritance
3. Polymorphism

## Properties

Property is a class member that encapsulates a getter/setter for accessing a field, It is useful to create a getter/setter with less code, property is defined as **public**

private DateTime \_birthdate; // private fields should follow camel case after \_ its general coding practice

public DateTime BirthDate

{

**get** **{** return \_birthdate;**}**

**set{** \_birthdate = **value**;**}**

}

value is this variable which just sort of exists in the language and works with in that context of getters and setters when you are creating your properties

It is possible to not have a set method for normal properties .

* Auto Implemented Property

Ex : **public DateTime BirthDate { get;set;}**

In case of Auto Implemented property if the property is to be set only once then place the private accessor before set and it should be initialized in the constructor as in example below .

Keyboard shortcut **prop** and tab can be used to create auto implemented properties

For properties which have private set; as set accessor , property cannot be set outside class ,it can be set only in constructor.

Erx: public DateTime BirthDate {get; private set;}

Ex:

**public class** Person

**{**

**public DateTime** BirthDate **{ get; private set;}**

**public void** Person(DateTime b)

**{**

BirthDate = b;

**}**

**}**

* Indexers are a way to access elements in a class that represents a list of values
* Indexer is nothing but a property it is defined as shown below, instead of property identifier use the **this** key word

Ex: public class HttpCookie

{

public string **this[**string key**]**

{

get {return \_dictionary[key];}

set {\_dictionary[key] = value;}

}

}

var ck = new HttpCookie();

**ck[“name”]** = “Dhaval”;

* Static methods are the methods which you can call without instantiating the class
* Static class can't contain non-static members because by definition it can't be instantiated so there's no possibility to use these members. However, static members in non-static class can be used without having class instance - a bit different scenario, i.e. for utility methods or factory methods.
* In general there are two types of relationships between classes

1. Inheritance
2. Composition

# **Inhertiance**

* Inheritance **is a** kind of relationship between two classes that allows one to inherit code from the other
* Benefits of Inheritance are

1. Code reuse
2. Polymorphic behaviour

* Syntax to inherit from a base class is as follows

Ex : public class BaseEx

{

}

public class DerivedEx **:** BaseEx

{

}

* **object** class is the parent class of all the classes in C#
* In c# a class cannot have multiple base classes
* Base class constructors are always executed first
* Base class constructors are not inherited
* The private fields in base class cannot be initialized in the dervied class’s constructor

For ex :

public class Vehicle

{

private string \_registrationnum;

}

public class Car : Vehicle

{

public Car(string rnum)

{

\_registrationnum = rnum; // error \_registrationnum is not accessible here

}

}

Solution is to use the **base** keyword

public Car(string rnum)**:base**(rnum)

{

// Initialise fields specific to the car class

}

# **Upcasting & Downcasting**

* Conversion from a derived class to a base class (Upcasting)
* Conversion from a base class to a derived class (Downcasting)
* In upcasting no explicit cast is required , its implicit , An object reference can be implicitly converted to a base class reference
* In downcasting explicit casting is required using () syntax

Ex: public class Shape

{

}

public class Circle : Shape

{

}

Circle circle = new Circle();

Shape shape = circle; // up casting

Circle anothercircle = **(**Circle**)**shape; // downcasting

* Casting can throw InvalidCastException
* **as** keyword is use to ensure the casting is reliable the return of the expression using as keyword is null if the casting is not a correct one

Ex:

private void Button\_Click(object sender, .)

{

var button = sender **as** **Button**; // Button is a class derived from object class

If (button != null)

{

}

}

# **Boxing & Unboxing**

* The process of converting a value type instance to an object reference is called Boxing

Ex: int x =10;

object o = x;

Or

object o1 = 15;

* Unboxing is opposite of boxing

Ex: object obj =10;

int n = (int)obj;

# **Composition**

* A kind of relationship between two classes that allows one to contain the other
* Has-a relationship
* Benefits of Composition are

1. Code- reuse
2. Flexibility
3. Means for designing loosely coupled applications

* Just create a variable of the class which we want to have a relationship in the class which we want to use it

Ex:

public class Logger

{

public void Log(string msg)

{

cw(“Logging in Logger {0}”, msg);

}

}

public class Installer

{

private Logger logger;

public Installer(Logger l)

{

Logger =l;

}

public void inst()

{

\_logger.Log(“We are installing smething”);

}

}

void main()

{

var loger = new Logger();

var ins = new Installer(loger);

inst.inst();

}

# **Composition vs Inheritance**

* Problems with Inheritance are

1. Large Hierarchies are fragile, if we change something in the most parent class all the derived classes are affected , this is due to the tightly coupled design

* Any Inheritance relationship can be translated to Composition

# **Polymorphism**

* Modifying the implementation of an Inherited method is called method overriding
* **virtual** keyword is used in the method of parent class which can be overridden
* **override** keyword is used in the derived class for the methods which are being overridden from the parent class

Ex:

public class Shape

{

public virtual void Draw()

{

}

}

public class Circle: Shape

{

public override void Draw()

{

}

}

* Abstract modifier indicates that a class or member is missing implementation, its implementation is to provided by its Derived classes
* The class that has abstract mehtod should be defined as abstract as well

Ex : public **abstract** class Shape

{

public **abstract** void Draw()**;**

}

public class Circle : Shape

{

public override void Draw()

{

}

}

* Rules for using Abstract classes are

1. Abstract member cannot include implementation ,meaning the method will not have any body, its purely a declaration ending with ;
2. If a member is declared as abstract the containing class needs to be declared as abstract too
3. Derived class must implement all abstract members in the base abstract class
4. Abstract classes cannot be instantiated

* Sealed modifier prevents derivation of classes or overriding of methods, sealed modifier can be applied to a class or a class member
* If applied to a class **sealed** modifier prevents derivation of classes from that class
* If applied to a method **sealed** modifier prevents overriding of that method
* Sealed classes are slightly faster because of some runtime optimizations

# **Interfaces**

* Interface is a language construct that is similar to a class ( in terms of syntax) but is fundamentally different, Interfaces are used to build loosely coupled applications

Syntax :

**public interface** ITaxCalc

{

int calculate();

}

In .net all interfaces start with a letter **I**

* Unlike classes Interfaces do not have an implementation, also interface members do not have access modifiers like public, private, protected, its just a declaration
* A class that implements an interface is as shown below

Ex:

**public interface** IShippingCalculator

{

float calculateshipping(Order ord);

}

**public class** ShippingCalculator **:** IShippingCalculator

{

public float calculateshipping(Order ord)

{

// some code to calculate shipping cost based on order total price

}

}

* Interfaces are used for unit testing and also for Extensibility
* A class can implement multiple interfaces
* Interfaces and Inheritance are completely different , note: in C# a class cannot have multiple base classes.
* Interfaces are not used for building multiple inheritance
* Interfaces can be used to implement polymorphic behaviour by creating a IList of classes implementing the interface and on runtime the functions in the interface are called depending on the class at run time

# **Generics**

* With Generics we create a class once and re use it multiple times, The syntax to create a Generic is as shown below, We usually don't need to create a generic instead we use available Generics in System.Collections.Generics class
* Generics have a parameter and the parameter are specified with angle brackets **<T>**, we usually call the parameter T as in short for Template or Type

Ex :

public class GenericList**<T>**

{

public void Add(**T** value)

{

}

}

var nums = new GenericList<int>();

var books = new GenericList<Book>();

books.Add(new Book());

* We achieve Code reusability with no performance penalty with Generics as there is no casting or boxing
* Generics can have more than one parameter

public class GenericDict<Tkey, Tval>

{

public void Add(Tkey k, Tval val)

{

}

}

## **Constraints**

* Sometimes we may need to apply constraints on the type of the Template to be used , it is done using **where T: IComparable ,new()**

or **where T: Product** // another class or any of its child classes

**where T: struct** // T has to be a value type

**where T: class** // constraint to a class,it has to be a class type

**where T: new()** // an object that has a default constructor

Ex: public class Utilities<T> where T: Icomparable // T is Icomparable (constraint to an I

{

}

* Generics can be applied to both class and class members , Also generic methods can exist inside non generic class

Ex : public class utl

{

public T Max<T>(T a, Tb) where T: IComparable

{

return (a.CompareTo(b) > 0?a:b);

}

}

* **System.Nullable** is a structure

Ex: public class Nullable<T> where T:struct

{

private object \_value;

public void Nullable()

{

}

public void Nullable(T val)

{

\_value = val;

}

public bool HasValue

{

get { return \_value !=null;}

}

public T GetValueOrDefault()

{

if(HasValue)

return (T)\_value;

else

return **default**(T); // keyword which returns default value of a type

}

}

var num = Nullable<int>(); // HasValue = false, returns default of int when we call GetvalueOrDefault fn

var num1 =Nullable<int>(10);

# **Delegates**

* An object that knows how to call a method (or group of methods)
* A reference to a function
* Delegates are useful for designing extensible and flexible applications
* Delegates are declared as shown below

**public delegate void** PhotoFilterHandler(Photo photo);

PhotoFilterHandler fh = filters.ApplyBackGround();

* There are two framework provided Generic Delegates

They are

a)**System.Action** it has 16 overloads, it doesn't return anything (void)

b)**System.Func** it has 18 overloads and returns a output

* Use a Delegate when

1. Eventing design pattern is used
2. The caller does not need to access other properties or methods on the object implementing the method

# **Lambda Expressions**

* An Anonymous method having no Access modifier like public/private , having no name, and no return statement is called a Lambda expression
* Lambda expressions are used for convenience and to shorten the code
* Syntax to write lambda expressions is as follows

args **=>** expression

If you dont need arguments

**() =>** expression

x **=>** expression // One argument

(x,y,z) **=>** .. // Multiple arguments

number **=>** number \* number;

* We can assign a lambda expression to a delegate like this

Func<int,int> square = number **=>** number \* number**;**

cw(square(6));

* Lambda expression has access to all the arguments and also to all the variables in the function in which this lambda expression is defined

# **Events & Delegates**

* Events are a mechanism for communication between objects used for loosely coupled applications. There are Event publisher who publish the event and Event Subscriber who subscribe to the event
* Delegates are Agreement/Contract between publisher and subscriber
* Delegates determine the signature of the event handler method in subscriber
* Three steps for writing the code of Events & Delegates are

1. Define a delegate

Ex: **public delegate** void VideoEncodedEventHandler(object source, Eventargs args);

1. Define an Event based on that delegate

Ex: **public event** VideoEncodedEventHandler VideoEncoded;

1. Raise the Event

Ex:

protected virtual void OnVideoEncoded()

{

if(VideoEncoded !=null)

VideoEncoded(this, EventArgs.Empty);

}

* .net framework provides **EventHandler** delegates for easing the purpose of using Events

With EventHandler delegates you can create a delegate easily

1. **public event EventHandle**r<VideoEventArgs> VideoEncoded;

2) **public event EventHandler** VideoEncoding;

* Events provide notifications and send data using EventArgs
* Delegates act as the glue/pipeline between events & event handlers
* Event Handlers receive and process EventArgs data
* Steps :

1. Creating a delegate

Syntax: **public delegate** <returntype> <delname>(<parameters>);

1)Delegates can stand on their own , you dont have to use delegates with events

2)To instantiate a delegate & to move data from one point to another , we need an handler , for somewhere that data to go

3)Delegates’ signature as far as the parameters and handler methods’ parameters must match up

4) To create a delegate instance just new up a delegate

Ex :

public delegate exdelhandler(int x, int y);

exdelhandler ex1 = new exdelhandler(nf); // delegate initialized to nf

void nf(int z, int w): // A handler

4) you can make a delegate to invoke more than one function handler

Ex : exdelhander ex2 = new exdelhandler(nf2);

ex1 += ex2;

Adds ex2 to the invocation list of ex1

Delegate

Method

Target

GetInvocation List

MultiCastDelegate

Custom Delegate

1. Defining an event based on delegate( marrying a delegate with an event)

Syntax : **public event** <delegatename> <eventname>;

Events on its own doesnt do anything, it should be associated with a delegate

Events are friendly wrappers around delegates

1. Raising an event : Events are raised by calling the event like a method

Syntax: <eventname>**(**<params>**);**

Another option is to access the event’s delegate and invoke it directly

Ex: WorkPerformedHandler del1 = WorkPerformed as WorkPerformedHandler;

if (del1 !=null)

del1(8,WorkType.Golf);

Event can be cast as a delegate

Usually .net developers ue the On<eventname> () function or Raise() function to raise events , where in they check if the event is not null and then raise/call the event

# **EventArgs & custom EventArgs**

* In .net we have a standard way to raise events & pass data , i.e by passing a sender and an event Args
* The **EventArgs** class is used in the signature of many delegates & event handlers , when custom data needs to be passed the EventArgs class can be extended , by using custom Event Args it keeps the signature of event handlers and delegates really clean , plus a lot of data can be encapsulated in the EventArgs
* Signature to define a custom event Args class is

public class WorkPeformedEventArgs **: EventArgs**

{

}

* Framework provided delegate **EventHandler<T>** , where T represents the EventArgs , now we don't need to write a delegate

Ex : public event EventHandler<WorkPerformedEventArgs> WorkPerformed;

# **Handling Events**

* The delegate signature must be mimicked by a handler method
* **+=** operator is used to attach an event to an event handler
* Defining and attaching event handler

Ex : var worker = new Worker(); // Worker is a class which has event s declared

worker.WorkPerformed += new EventHandler<WorkPerformedEventArgs>(Worker\_WorkPerformed);

// Worker\_WorkPerformed is a stand alone function /Event handler

When event happens then this stand alone function will be called

# **Delegate Inference**

* When we compile c# can figure out what the delegate is just by looking at the event signature & then generate an automatic delegate for us as it builds the code

So instead of

worker.WorkPerformed += new EventHandler<WorkPerformedEventArgs>(Worker\_WorkPerformed);

worker.WorkPerformed **+=** Worker\_WorkPerformed;

* We can also use **-=** to remove/detach an event handler from an event

# **Anonymous Methods**

* Event Handlers are generally standalone functions , but there maybe times when as an event fires you want to hook the action of what to do with the event
* Anonymous methods allow event handler code to be hooked directly to an event
* Anonymous methods are defined using the delegate keyword
* Anonymous methods have no name
* This Anonymous method cannot be attached to other events

Ex:

worker.workPerformed += **delegate** (object sender, WorkPeformedEventArgs e)

{

//do something

}

# **Extension Methods**

* Allows us to add methods to an existing class without changing its source code or creating a new class that inherits from it
* To create an Extension method you need to create a static class

public static class StringExtensions

{

public static string Shorten(this String s, int nbr\_of\_wrds)

{

}

}

# **Dynamics**

* C# started as a statically typed language / static language , but in .net4 they added dynamic capability to c# , to improve interoperability with COM and Dynamic languages (IronPython)
* In .net4 there is a component called DLR( Dynamic language Runtime) which sits on top of CLR which gives dynamic language capabilities to C#

Ex: **dynamic** d = 10;

d = “hi how are you”;

// Earlier d was of type int, and in the next line it changes its type dynamically to string , this is possible with dynamic types

int i =15;

**dynamic** j =i; // j will be of type int

* When converting from dynamic to static types if the runtime type of dynamic object is implicitly convertible to target type we don't need to cast it

Ex: dynamic d =15;

long l =d;

# **LINQ**

* LINQ stands for Language Integrated Query, it gives the capability to query objects in c# natively
* You can Query

1. Objects in Memory , eg collections (LINQ to objects)
2. Databases (LINQ to Entities)
3. XML(LINQ to XML)
4. ADO.NET Data sets(LINQ to datasets)

* We need to include Linq using the following statement

**using system.Linq;**

* LINQ Query operators

var cheaperbooks = from b in books

where b.Price <10

orderby b.Title

select b;

* LINQ Extension methods

1. **Where**
2. **OrderBy**
3. **Select**
4. Single// throws exception if condition not met
5. SingleOrDefault // returns null if condition is not satisfied
6. First // throws exceptionif condition not met
7. FirstOrDefault
8. Last
9. LastOrDefault
10. Skip
11. Take
12. Count
13. Max
14. Sum
15. Min

# **Nullable Value Types**

* Value types cannot be null
* But sometimes we may require a value to be nullable like storing customers birthdate in database, where some customer doesnt want to share his/her birthday

Ex : DateTime d = null; // not allowed in c# as its value type

Solution is to use

**Nullable<**DateTime**>** d = null;

Short form of above statement is

DateTime**?** D = null;

* Commonly used members of Nullable type are

**GetValueOrDefault()** Ex: d.GetValueOrDefault() // returns value if its have or else returns the default value of that type

**HasValue** Ex: d.HasValue // returns true if it has a value or false if its null

**Value** Ex: d.Value // Throws an exception if the value is null

* You cannot assign a nullable type to another type

Ex :

DateTime d2 =d; // error as d is of Nullable type

Solution is to use GetValueOrDefault()

DateTime d2 = d.GetValueOrDefault()

* A value type can be easily converted to a nullable type without any type casting

Ex: DateTime d3? = d2;

* Null Coalescing operator **??**

DateTime? d =null;

Ex: DateTime d2 = d ?? DateTime.Today; //similar to ternary operator if d is not null assign d2 to d or assign DateTime.Today

# **Exceptions**

* try { } -> any block of code which can cause an exception or crash
* catch (Exception) {} -> catch a particular type of exception of just Exception which is generic for all type of Exceptions
* finally { } -> finally block will be run every time if there was an error or no error in case of try and catch . finally block is placed below try and catch
* K
* K
* K
* K
* K
* K
* K
* The using statement ensures that **Dispose()** is called even if an exception occurs when you are creating objects and calling methods, properties and so on. Dispose() is a method that is present in the **IDisposable** interface that helps to implement custom Garbage Collection
* You can do the same as what the using statement is doing by using try-catch block and calling the **Dispose()** inside the **finally** block explicitly. But the using statement does the calls automatically to make the code cleaner and more elegant.
* Within the **using** block, the object is read-only and cannot be modified or reassigned

Ex:

using(Employee emp = new Employee())

{

emp.getEmp();

}

# **Async/Await**

* Synchronous programming model is one where program is executed line by line,one at a time and when a function is called program has to wait until the function returns
* In Asynchronous programming when a function is called program execution continues to the next line without waiting for the function to complete
* Asynchronous programming model improves responsiveness of the application
* Its best to use Asynchronous model of programming whenever we need to access the web, working with files & data bases, working with images etc
* Traditionally Asynchronous programming model was achieved through Multi Threading and call backs but since .NET 4.5 there is the concept of task based using Async/Await keywords
* Return type of a Async method is a Task Object , A task is an object that encapsulates the state of an Asynchronous operation, it comes in two forms one is non generic form **Task** and the other is the Generic form **Task<> ,** The task returned from an asynchronous method is a reference to the operation , its result or potential errors, compiler takes care of returning reference of ongoing operation for us
* The only time you are allowed to async void is for event handler , using async void as return type is only appropriate for event handlers
* Without using Async method the below code will not be responsive for the wpf window on button click

Ex : **public void DownloadHtml(string url)**

**{**

var webClient = new WebClient();

var html = webClient.DownloadString(url);

using( var streamWriter = new StreamWriter(@”c:\projects\result.html”)

{

streamWriter.Write(html);

}

**}**

* Async method Declaration

Ex : **public async Task DownloadHtmlAsync(string url)**

**{**

var webClient = new WebClient();

var html = await webClient.DownloadStringTaskAsync(url);

using( var streamWriter = new StreamWriter(@”c:\projects\result.html”)

{

streamWriter.Write(html);

}

**}**

* You can use **await** keyword only in **async** method , If we dont have await keyword then we dont need async keyword either
* The **await** keyword is a way for us to indicate that we want to get the result out of the asynchronous operation only once the data is available without blocking the current thread
* The await keyword will pause execution of the method until a result is available without blocking the calling thread
* Using await will validate the success of the asynchronous operation as well as after successful execution gives you the potential result
* The await keyword introduces a continuation which is code executed after the asynchronous operation ( The code beneath the await keyword is only executed once the asynchronous operation you are awaiting is completed) The continuation is executed on the original context EX: UI thread
* Marking a method as async indicates that this method will have capability of executing asynchronous operations that we are then awaiting inside the same context ( Marking a method async introduces the capability of using the await keyword to leverage the asynchronous principles)
* Never use async void unless its an event handler or delegate
* Never block an Asynchronous operation by calling Result property or Wait() method

# **Files & Directories**

* **System.IO** namespace consists of **File**, **FileInfo**, **Directory**, **DirectoryInfo**, **Path** classes to help working with Files and Directories
* File(static methods), FileInfo(Instance methods) classes provides methods for creating, copying, deleting, moving and opening of files they provide following methods

**Create()**

**Copy()**

**Delete()**

**Exists()**

**GetAttributes()**

**Move()**

**ReadAllText()**

* Directory and DirectoryInfo classes are similar to File and FileInfo Classes , they provide following methods

**CreateDirectory()**

**Delete()**

**Exists()**

**GetCurrentDirectory()**

**GetFiles()**

**Move()**

**GetLogicalDrives()**

* Path class contains methods to work with a string that contains a file or directory path information , some useful methods of Path class are

**GetDirectoryName()**

**GetFileName()**

**GetExtension()**

**GetTempPath()**

# **P/Invoke**

* K
* p/Invoke code gen utility called "P/Invoke Interop Assistant”
* K
* K
* K

# **Unit Testing**

* For Unit testing in Visual Studio create a new project, select Test project from visual c# menu while creating a project
* The classes with **[TestClass]** attribute and methods with **[TestMethod]** attribute would be run by Microsoft test runner
* K
* K
* K
* k
* k

# **Debugging in Visual Studio**

* Ctrl+shift+B to compile , ctrl+F5 to run the application
* Ctrl+F5 to run in normal mode,
* **F5** to run application in debug mode,
* **F9** to set **breakpoint** ,
* **F10** to **step over** ,
* **F11** **Step Into** ,
* **shift +F11** to **step out** ,
* **shift+F5** to come out of debug mode,
* **F5** to continue to next breakpoint
* Debug->window->callstack to look at the call stack
* Debug->window->watch->watch1 to see the watch window
* Debug->window->Autos
* Debug->window->Locals similar to Autos window but doesn't show all kind of variations , just shows the locals

# **Important Links & References**

<https://docs.microsoft.com/en-us/dotnet/csharp/how-to/compare-strings>

<https://en.wikibooks.org/wiki/C_Sharp_Programming/Keywords>

<https://www.c-sharpcorner.com/UploadFile/efa3cf/different-types-of-method-parameters-in-C-Sharp/>

<https://www.geeksforgeeks.org/c-sharp-method-parameters/>

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members>

<https://www.infoworld.com/article/3006630/how-to-work-with-attributes-in-c.html>

<https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/attributes>

<http://dotnetpattern.com/csharp-concurrentqueue>

<https://www.geeksforgeeks.org/c-sharp-tuple/>

<https://jonskeet.uk/csharp/events.html>

<https://www.geeksforgeeks.org/types-of-threads-in-c-sharp/>

<http://dotnetpattern.com/threading-autoresetevent>

<https://www.dotnetperls.com/string-join>

<https://docs.microsoft.com/en-us/cpp/windows/walkthrough-creating-windows-desktop-applications-cpp?view=vs-2019>

<https://stackoverflow.com/questions/12734174/what-is-the-c-sharp-equivalent-to-strstr-in-c>

<https://docs.microsoft.com/en-us/dotnet/framework/interop/marshaling-classes-structures-and-unions#structures-sample>

<https://www.tangiblesoftwaresolutions.com/articles/csharp_equivalent_to_cplus_types.html>

<https://jonskeet.uk/csharp/memory.html>

<https://www.oreilly.com/library/view/c-in-a/0596001819/re525.html>

<https://limbioliong.wordpress.com/2011/06/16/returning-strings-from-a-c-api/>

<https://limbioliong.wordpress.com/2011/11/01/using-the-stringbuilder-in-unmanaged-api-calls/>

<https://stackoverflow.com/questions/370079/pinvoke-for-c-function-that-returns-char/370519#370519>

<https://tutorials.eu/50-interview-questions-for-your-csharp-interview/>

<https://tutorials.eu/how-to-check-the-palindrome-number-in-csharp-programming/>

<https://tutorials.eu/armstrong-number-and-how-to-check-it-with-csharp/>

<https://tutorials.eu/csharp-fibonacci-sequence-series/>

<https://tutorials.eu/what-is-a-prime-number-and-how-to-generate-prime-numbers-using-c-programming/>

## Nullable Value Types

<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-value-types>

## yield

<https://www.programiz.com/csharp-programming/yield-keyword#:~:text=The%20yield%20keyword%20is%20used,yield%20break%20%2D%20terminates%20the%20iteration>