C#

C# is an object-oriented, typed safe and structured language that is compiled by .NET framework and was developed by Microsoft

Follows a Structured approach

Produces readable and efficient programmes

Once written can be compiled on different platform

Variable – a name given to storage location in memory

Naming Conventions – For variables 🡪 camelCase

For Constants 🡪 PascalCase

**Comments in C#**

1. Single line comments // hello, this is a single line comment
2. Multiline comments /\* Hello this is a multiline comment last line of comment\*/
3. XML comments /// Hello this is XML comment

**Code Compilation in C#**

1. Compilation of Source code in managed code
2. Clubbing the newly created code into assembly
3. Loading the CLR (Common Language Runtime)
4. Execution of assembly through CLR

The C# compiler compiles the C# source code into the module, which is finally converted into the assembly.

The assembly contains the Intermediate Language (IL) code along with the metadata information about the assembly.

The common language runtime (CLR) works with the assembly. It loads the assembly and converts it into the native code to execute the assembly.

Then this native code is executed by the Operating system and the output will shows according to your requirement.

Before C# in C/C++ when we execute the program, program code will be converted into native code (particular to that machine), compiled won’t run in another machine with different OS.

That JAVA introduced the concept called intermediate code.

Java program 🡪 Intermediate (Byte Code) 🡪 Machine Code (Native code)

Same way in C#

C# 🡪 Intermediate Language code 🡪 Machine code

So CLR is the one sits in the memory and converts IL into native code. This process is called Just-in-time compilation (JIT).

**Architecture:**

C# application is a collection of classes

Class - is a container contains data and methods

Namespace – container contain collection of related classes

Assembly(DLL/exe) – collection of namespaces

Application – Collection of assemblies depends on your partition of code

**Break and Continue**

break completely exits the loop. continue skips the statements after the continue statement and keeps looping.

**Type Safety**

If a variable is mentioned the type of int, then we can’t assign other data type

**Overflowing:**

In General, max value a byte can have is 255.

Byte number = 255;

Number = number + 1;

If we output number result will be 0

**Type Conversion:**

**Implicit:** No special syntax is required because the conversion always succeeds and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.

int num = 2147483647;

long bigNum = num;

**Explicit:** Explicit conversions require a cast expression. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class

double x = 1234.7;

int a = (int)x;

Console.WriteLine(a); // output: 1234

Conversion examples

double d = 765.12;

int i = d;

* Error cannot implicitly convert type double to int

double d = 765.12;

byte i = (byte)d;

* Output: 253 (max occupancy of byte)

double d = 765.12;

int i = (int)d;

* Output: 765 (loosy conversion loss of 0.12)

C# built-in Conversions:

int i = 12;

            double d = 765.12;

            float f = 56.123F;

            Console.WriteLine(Convert.ToString(f));

            Console.WriteLine(Convert.ToInt32(d));

            Console.WriteLine(Convert.ToUInt32(f));

            Console.WriteLine(Convert.ToDouble(i));

            Console.WriteLine("GeeksforGeeks");

Output:

56.123 765 56 12 GeeksforGeeks

Non compatible types: string into int or boolean

string to int 🡪 string a = “1234”; int I =(int) a 🡪 throw error

string to int 🡪 string a = “1234”; int I = Convert.ToInt32(a); 🡪 work fine

General: By default, C# compiler treats real number like 2.434, 24.44 as double. To initialize we must explicitly mention 2.434f, 24.44f Float f= 23.4F; Double d = 23.4;

Operators:

**Increment Decrement**

Int a = 1; int b = a++; 🡪 a=2 b=1

Int a = 1; int b = ++a; 🡪 a=2 b=2

**OOPS:**

Object Oriented Programming is a programming paradigm in which programming is based on objects rather than functions and procedures.

It is based on bottom up approach, where smaller components are resolved and then later reintegrate everything to form a solution

Reduces the redundancy

Class

A class is the core of any modern Object-Oriented Programming language such as C# and it’s a blueprint of an object that contains variables for storing data and functions to perform operations on the data.

* It will not occupy any memory space
* If not mentioned, then the default access specifier for a class type is internal. Default access for the members is private

Object

It is an instance of class, basic unit of OOP and represents real life entities

Structure

* C# also provides a light version of classes called structures that are useful when you need to create large array of objects and do not want to consume too much memory for that.
* A struct type is a value type that is typically used to encapsulate small groups of related variables, such as the coordinates of a rectangle or the characteristics of an item in an inventory
* A struct cannot contain parameter less constructor. It can only contain parameterized constructors or a static constructor.
* structures cannot inherit other structures or classes

**Inheritance**

A property in which a class allows to inherit properties from other classes

Multiple Inheritance 🡪 Class C : Class A, B

Hierarchical Inheritance 🡪 Class Vehicle

Class Car, Class Bus, Class Bike

**Super class**: It’s a base class acts as a parent to some other class

**Polymorphism:**

It refers to the ability to exists in multiple forms.

Static Poly: Or Static binding occurs at compile time. Eg: method overloading

Dynamic Poly: Or runtime binding, which is resolved during run time. Eg: method over riding

Method Overloading: Same function name but with different parameters

Method Overriding: Child class redefines the method which present in the base class, it also has same name and signature

**Encapsulation:**

Process of binding together the data and function to work as single unit. Access specifiers are used to implement encapsulation.

Useful for data hiding

Access Modifiers:

Public – can be accessed by any other code in the same assembly or another assembly that references it

Private – can be accessed only with in the class

Protected – can be accessed within same class and inherited class

Internal - can be accessed by any code in the same assembly, but not from another assembly

Protected internal – same assembly and derived assembly

Properties:

GET SET

**Data Abstraction:**

It allows displaying only the important information and hiding all other implementation details. It is achieved using abstract class and abstract methods.

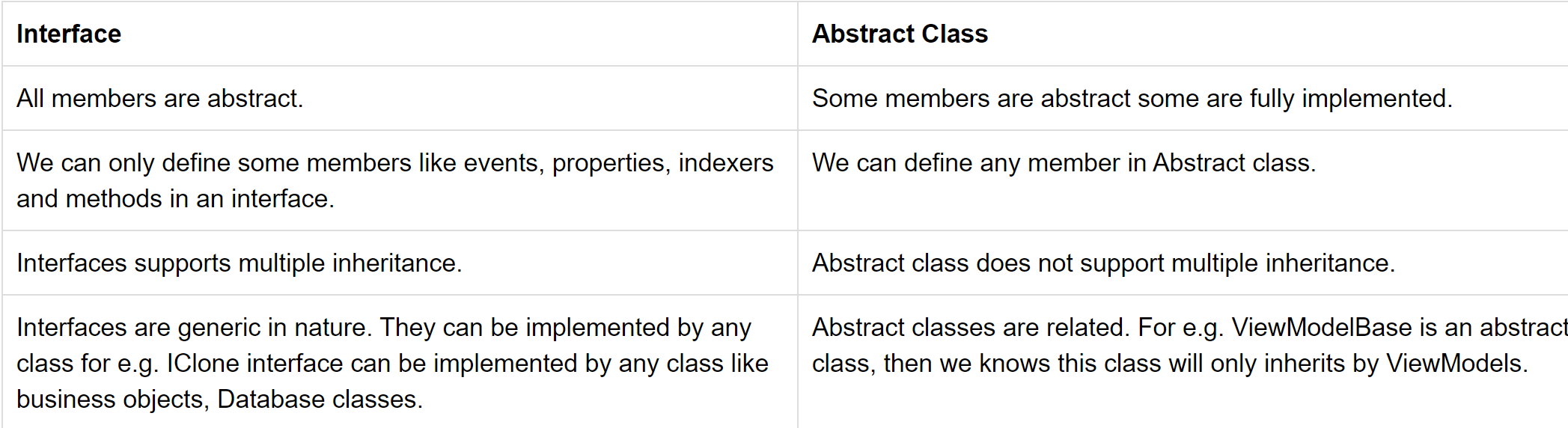
Abstract class:

Abstract methods:

Interface: is basically a class with no implementation.

Static method:

Difference abstract class and Interface



Interface does not have constructor There can be a constructor for Abstract class

**When to use Abstract class and interface**

If you want to increase reusability in inheritance then abstract classes are good. If you want implement or force some methods across classes must be for uniformity you can use a interface. So to increase reusability via inheritance use abstract class as it is nothing but a base class and to force methods use interfaces.

**Constructor**: Special type of method that has the same name as class name and is used to initialize objects of class

**Destructor**: Automatically invoked when an object is destroyed. It closes the file & database connection and recovers the heap space.

**Boxing**: Converting a value type to a reference type

int value = 10;

object boxedvalue = value1;

**Unboxing**:

int UnBoxing = int(boxedvalue);

**Why Should main be static?**

A static method can be called without instantiating an object. You need an entry point into your program. Static means that you can call the function without having to instantiate an object/instance of a class. Therefore main() needs to be static in order to allow it to be the entry to your program.

**Is it possible to have more than one main () method in a C# program?**

You can have more than one main method, you can specify which to use as the entry point for the application by your compiler.

**Array**

Is a collection of homogeneous elements

Initiaization 🡪 var arr = new int [3];

Generally after the initialization of array all spaces will be initialized to 0 🡪 arr[1] = 0same for arr[2] = 0

If array type is bool then elements are initialized as false.

Initializations:

Var numbers = new int [5];

Var numbers = new int [5]{1, 2, 3, 4, 5};

Multi-dimensional:

**Rectangular** 🡪 same no of columns

Var numbers = new int [3, 5];

int[, ] intarray = new int[4, 2];

Var numbers = new int [3, 5]

{

{1,2,3,4,5},

{6,7,8,9,0},

{3,7,8,2,5}

};

**3D Array** 🡪 int[ , , ] intarray1 = new int[4, 2, 3];

        int[, ,] arr3d3 = new int[2, 2, 3]{

{ { 1, 2, 3}, {4, 5, 6} },

{ { 7, 8, 9}, {10, 11, 12} }

};

arr3d2[0, 0, 0]; // returns 1

arr3d2[0, 0, 1]; // returns 2

arr3d2[0, 1, 0]; // returns 3

arr3d2[0, 1, 1]; // returns 4

arr3d2[1, 0, 0]; // returns 5

arr3d2[1, 0, 1]; // returns 6

arr3d2[1, 1, 0]; // returns 7

arr3d2[1, 1, 1]; // returns 8

3d – [0 -1st set, 1 – 2nd set in 1st set ,0 – first element in 2ndsets1st]

To access value 11 🡪 arr3d3[1, 1, 1]

To access value 3 🡪 arr3d3[0, 0, 2]

**Jagged** 🡪 Array of Arrays (Different no of columns)

Var numbers = new int [3][];

numbers[0] = new int [4];

numbers[1] = new int [5];

numbers[2] = new int [3];



Array Operations:

Var numbers = new[] {1, 2, 3, 4, 5, 6, 7};

//Length()

Length of an array 🡪 numbers.Length;

//Index()

Index of number 4 🡪 Array.IndexOf(numbers, 4);

//Clear()

Clearing (setting elements to 0/false) 🡪 Array.Clear(numbers, startIndex, endIndex)

//Copy()

Copying elements to new array 🡪 int[] newarray = new int[3];

Array.Copy(numbers, newarray, 3); (upto index 3)

//Sort()

Array.Sort(numbers); 🡪 by default ascending

//Reverse()

Array.Reverse(numbers);

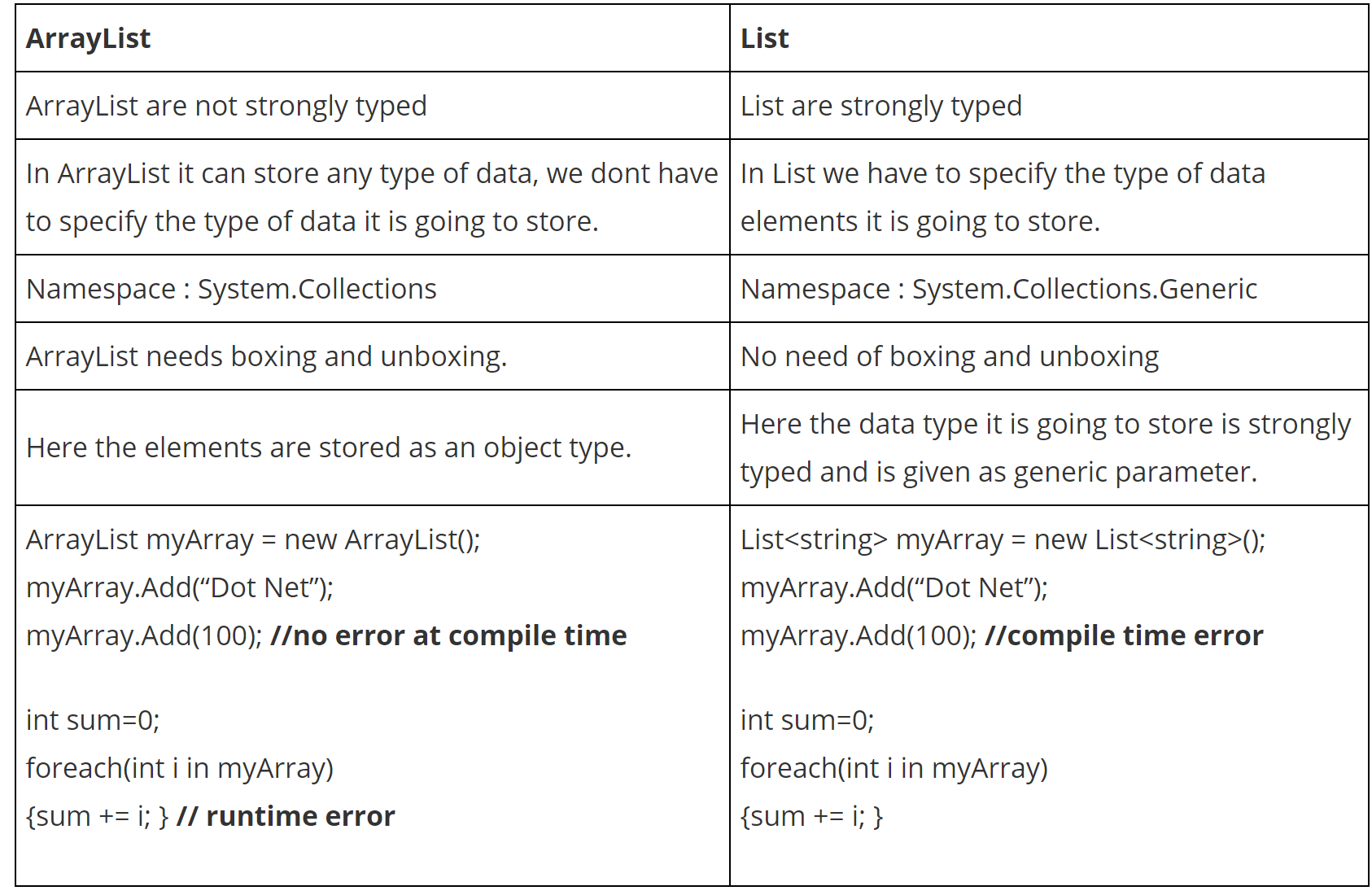
For and foreach:

* a for loop is a construct that says "perform this operation n. times". For is more efficient.
* a foreach loop is a construct that says "perform this operation against each value/object in this IEnumerable"
* disadvantage of enumerators is that they require calls to Current and MoveNext for every element in the sequence. Conversely, the for-loop only has to call get Item for every element in the list. That’s one method call less than the foreach-loop

**Lists:**

Array is affixed size whereas lists are dynamic. Hence list is a collection of heterogeneous elements.

Array memory allocated is static and continuous. List memory allocated is dynamic and random



ArrayList is a old concept in .Net

Lists initialization:

Var numbers = new List<int>();

Var numbers = new List<int>() {1,2,3,4,5};

Lists methods:

Add 🡪 adding an object to list

Numbers.Add(10);

AddRange 🡪 adding several objects at one go

Numbers.AddRange(new int[3] {6,7,8});

IndexOf 🡪 finding the index of object from the beginning

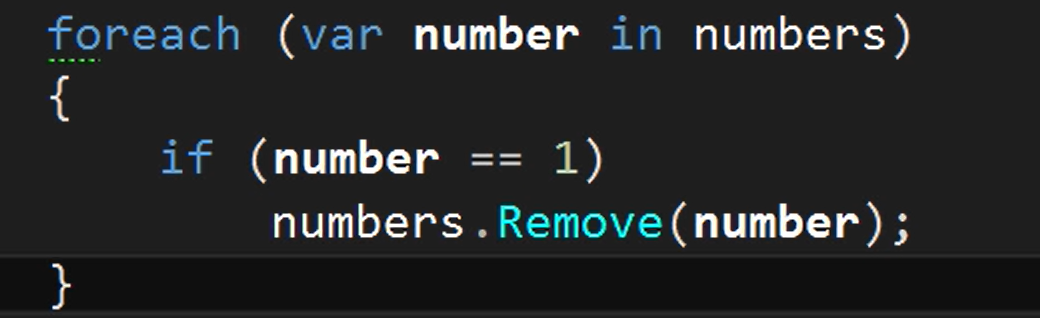
Numbers.IndexOf(10);

LastIndexOf 🡪 finding the index from the last

Count 🡪 Total number of objects in the list

Numbers.Count;

Remove🡪 to remove certain object from the list



Removing object using foreach is not allowed in c# to modify Collections have to do manually or by using for loop

Clear 🡪 To clear objects in List

Numbers.clear();

**Interface:**

Interfaces only have methods declared. So all the methods are abstract in nature.

Interfaces are used along with classes to define what is known as a contract. A contract is an agreement on what the class will provide to an application. An interface declares the properties and methods. It is up to the class to define exactly what the method will do.

They're great for putting together plug-n-play like architectures where components can be interchanged at will. Since all interchangeable components implement the same interface, they can be used without any extra programming

* It is used to achieve loose coupling.
* It is used to achieve total abstraction.
* To achieve component-based programming
* To achieve multiple inheritance and abstraction.
* Interfaces add a plug and play like architecture into applications.

**Generics:**

Generics

**Delegates:**

Delegate is an object

**Collections:**

Collection

LINQ:

Langusge Integrated Query

var query = context.GetQueryable<SearchResultItem>().Where(i => i.Name.Contains(“search”)).Filter(i => i.Language == “en”); //Sitecore

Asynchronous programming

WebAPI

* What are the company's goals for the upcoming year?
* If I were hired for this role, what would you want me to achieve in my first two months?
* Are there any qualifications that you think I'm missing?