

**Classes**

Fundamental building block of OOPS

Serves as a blueprint for creating objects.

Class declaration:

public class MyClass

{

// Fields

private int myField;

// Methods

public void MyMethod()

{

// Method implementation

}

// Properties

public int MyProperty { get; set; }

}

Once class is created, we can create instances of the class. Objects represent real world entities and encapsulate the properties and methods defined in the object.

Creation of object:

MyClass myObject = new MyClass();

Fields : variables inside the class

Methods: behaviour of class.

Properties: provide a way to encapsulate the private fields of the class and have getter and setter method, to get and set the values of the associated field.

Using get and set:

public MyClass

{

private string id;

public string Id

{

get{

return this.id;

}

Set{

this.id=value;

}

}

}

We can use the above program in main method by :

MyClass m=new MyClass(); //creating object

m.Id=1; //setting id

Console.WriteLine(m.Id); //getting id

We can also write the above program as:

public MyClass

{

Private string id;

Public string Id{

get;

set;

}

}

**Access Modifiers**

Control the visibility, accessibility of classes, members(fields,methods,properties) and types within the program.

Determine which part of the project or the program can access certain elements.

There are five different types of access modifiers:

1. Private
2. Public
3. Protected
4. Internal
5. Protected Internal

Public : accessible from any other class or assembly.

Private : accessible only within the same class.

Protected : accessible within the same class and by derived classes (subclasses).

Internal : only accessible within the same assembly.

Protected Internal : accessible within the same assembly and also derived classes of the same assembly or any other assembly.

**Virtual keyword**

virtual keyword is used to declare a method, property in a base class that can be overridden in a derived class.

allows a method in the base class to be redefined in a derived class using the override keyword.

Virtual Method:

When a method is declared as virtual in a base class, it means that it can be overridden by a method with the same signature in a derived class. The derived class provides a specific implementation for that method.

public class MyBaseClass

{

public virtual void MyMethod()

{

Console.WriteLine("Base class implementation");

}

}

public class MyDerivedClass : MyBaseClass

{

public override void MyMethod()

{

Console.WriteLine("Derived class implementation");

}

}

Virtual Properties:

Similar to methods, properties can also be declared as virtual in a base class. This allows a derived class to provide its own implementation for the property.

public class MyBaseClass

{

public virtual int MyProperty { get; set; }

}

public class MyDerivedClass : MyBaseClass

{

public override int MyProperty

{

get { return base.MyProperty \* 2; }

set { base.MyProperty = value; }

}

}

**Generics**

Introduces in C# 2.0

Allow us to create classes and methods decoupled from data types.

They make your code type independent.

Extensively used in collections.

class MyClass

{

public static void Main(string[] args)

{

bool equal=Calculator.AreEqual(10,10);

if(equal){

Console.WriteLine(“Equal”);

}

else{

Console.WriteLine(“Not Equal”);

}

}

}

class Calculator

{

public static AreEqual(object a,object b){

return a==b;

}

}

If we use Object, unnecessary boxing and unboxing should be done(in the above example, when we convert the integer type to object, i.e., Value type converted to reference type in runtime), where the performance gets degraded.

In the above program, if we call the method like:

bool equal=Calculator.AreEqual(10,”AB");

then also, it will be compiled and this method became strongly typed.

So, to avoid these problems, we have to use **generics.**

class Calculator

{

public static AreEqual<T>(T a,T b){

return a.equals(b);

}

}

We call the method like:

bool equal=Calculator.AreEqual<int>(10,”AB");

We can also make the class generic

class Calculator<T>

{

public static AreEqual(T a,T b){

return a.equals(b);

}

}

bool equal=Calculator<int>.AreEqual(10,”AB");

While specifying the Type it need not be only T.

**Delegates/Events**

Is a type-safe functional pointer

Points to a function and the function will be invoked (holds a reference to function)

using System;

public delegate void HellofunDelegate(string message);

class Program

{

public static void Main()

{

HellofunDelegate hfd=new HellofunDelegate(hello);

hfd(“Hello from Delegate”);

}

public static void hello(string strmessage){

Console.WriteLine(strmessage);

} }

A delegate to similar to class. You can create an instance of the delegate and have to pass the function name as the parameter to the delegate constructor an the function written inside it is the function to which the delegate points to.

Syntax : similar to method

<access modifier> delegate <return\_name> <delegate\_name>(arguments present in the function to which you want this delegate to point)

Creating a pointer to function :

Delegate\_name obj =new Delegate\_name(function\_name to which it points);

**Out Keyword**

A parameter to indicate the output process

The method is expected to assign a value before it returns.

The out parameter need not be initialised before being passed to the method, and the method must assign a value to it before exits.

public class Program

{

static void Main()

{

int result;

// Call a method with an out parameter

Add(3, 4, out result);

// 'result' now contains the output value assigned by the method

Console.WriteLine(result); // Output: 7

}

// Method with an 'out' parameter

static void Add(int a, int b, out int sum)

{

sum = a + b;

}

}

A method can have multiple out parameters.

static void MultiplyAndDivide(int a, int b, out int product, out int quotient)

{

product = a \* b;

quotient = a / b;

}

// Usage

int resultProduct, resultQuotient;

MultiplyAndDivide(6, 2, out resultProduct, out resultQuotient);

**Reference and Value Types**

1. Value Types : contains only an object with the value.

int a=50

1. Reference types : contains a reference to an object. Another variable may contain a reference to the same object.

Object obj=new Object();

obj.val=50;

Predefined data types: Size of these data types are predefined.

string X = "\"Hello \nHow are you\""; \ -> “ , \n->new line

There are three types:

1. Value Types:

Integral Types:

Signed : sbyte,short,int,long

Unsigned : byte, ushort,uint,ulong

* + - byte: 8-bit unsigned integer (0 to 2^8-1)
    - sbyte: 8-bit signed integer ( -2^7 to 2^7-1
    - short: 16-bit signed integer ( -2^15 to 2^15-1)
    - ushort: 16-bit unsigned integer (0 to 2^16-1)
    - int: 32-bit signed integer (-2^31 to 2^31-1)
    - uint: 32-bit unsigned integer (0 to 2^32-1)
    - long: 64-bit signed integer ( -2^63 to 2^63-1)
    - ulong: 64-bit unsigned integer (0 to 2^64-1)

Floating-Point Types:

* + - float: 32-bit single-precision floating-point ( a real number represented using 32 bits of memory, with 24 bits dedicated to the significant digits )
    - double: 64-bit double-precision floating-point ( a real numnber represented using 64 bits of memory, with 53 bits dediacted to significant digits allowing a wide range of precission. The larger the number of bits allocated for the significant digits, the higher the precision of the floating-point representation.)
    - decimal: 128-bit decimal floating-point (useful for financial calculations) (a real number represented using 128 bits of memory, with a base-10 representation for precise decimal calculations.)

Character Types:

* + - char: 16-bit Unicode character

Boolean Type:

* + - bool: Represents true or false values

1. Reference Types:
   * + Object Type:

object: The ultimate base class for all types in C#.

* + - String Type: Represents a sequence of characters.

1. User-Defined Types:
   * + Struct Type:

struct: Represents lightweight, stack-based structures.

* + - Enum Type:

enum: Represents a set of named integral constants.

**Operators**

symbols that represent computations or operations on variables and values.

fundamental building blocks for expressing logic and performing calculations in a program.

1. Arithmetic operators

* + (addition)
* - (subtraction)
* \* (multiplication)
* / (division)
* % (modulus, remainder after division)
* ++ (increment)
* -- (decrement)

1. Relational Operators

* == (equal to)
* != (not equal to)
* < (less than)
* >(greater than)
* <= (less than or equal to)
* >= (greater than or equal to)

1. Logical Operators

* && (logical AND)
* || (logical OR)
* ! (logical NOT)

1. Assignment Operators

* = (assignment)
* += (addition assignment)
* -= (subtraction assignment)
* \*= (multiplication assignment)
* /= (division assignment)
* %= (modulus assignment)

1. Bitwise Operators

* & (bitwise AND)
* | (bitwise OR)
* ^ (bitwise XOR)
* ~ (bitwise NOT)
* << (left shift)
* >> (right shift)

1. Ternery Operator

* ? : (conditional operator)

**Assemblies**

Fundamental unit of deployment

Pre-compiled chunk of dotnet code, which can run by CLR.

What is an assembly:

An assembly is a group of forms and assets that have been designed to interact with one another and provide a specific set of capabilities when used together.

The information required by the CLR, with which it needs to be conscious is present in the assembly.

In dotnet assemblies are the basic building blocks for:

* + Implementation
  + Version Control
  + Initialisation
  + Safety privileages

Assembly cache

When .net framework is installed a new directory is added to the hard drive during the installation process. We store all the assembled files in this assembly cache.

Cache is divided into:

• Private : restricted files for applications.

• Global: All assemblies in the global cache must have different namespaces. The folder name have the assembly distinct identity.

Assembly has four parts:

• Assembly name

• Version number

• Cultural Information

• Public key token

There are two types of Assemblies :

1. Private Assemblies

Used for local application development.

Each application has its own private class of assemblies.

It is easy to deploy in the application as it kept local to the application and as a result there are no versioning conflicts with other applications.

If multiple applications wants to use the same assembly then we need to develop the assembly for each application separately as a result we duplicate the code and disk space usage is increased.

1. Shared Assemblies

Used by multiple applications.

All these shared assemblies are stored in GAC ( Global Assembly Cache)

GAC : a common location for shared assemblies on a computer.

Reduces code reusability and avoids code duplication.

But, if not deployed properly with appropriate steps there is a chance of version conflicts.

There are two types of Shared assemblies:

1. Strong-named Assemblies

Shared assemblies that are associated with a unique key pair providing unique identifier for the assembly.

Prevents versioning conflicts

Allows to add into the Gac

This is useful in scenarios where multiple versions of the same assembly coexist.

1. Satelite Assemblies

Type of shared assembly that contains localized resources (e.g., strings, images) for specific cultures or languages. Used to separate language-specific resources from the main application, enabling support for multiple languages.

Allows applications to be easily localized without modifying the main executable. GAC:

Location : C:\Windows\Microsoft.NET\assembly

To ways to install assembly to GAC:

* Simply drag and drop
* GacUtil.exe (GAC Utility Tool)

Adding an assembly to GAC:

* Create an assembly
* Create a key pair

Strong named assembly can be saved into GAC.

* In visual studio developer cmd :
  + sn.exe -k C:\MyStrongKeys.sk
* Specify the below line in the assembly properties/assemblyinfo.cs ( at the end)
  + [assembly: AssemblyKeyFile("C:\\MyStrongKeys.sk")]
* Go to the assembly created .dll file

cd C:\Users\nikhita\_palla\Documents\Github\Dotnet\_Practice\C\_Sharp Basics Coding\Calculator\_Assembly\Calculator\_Assembly\bin\Debug

* Adding to GAC

gacutil.exe -i Calculator\_Assembly.dll

* Check in C:\Windows\Microsoft.NET\assembly\GAC\_MSIL if Calculator\_Assembly is added or not.
* To uninstall the GAC added assembly

gacutil.exe -u Calculator\_Assembly.dll

* Using The Gac assembly in program

Calculator\_Assembly.Class1 calculator = new Calculator\_Assembly.Class1();

Assemblies can be of various forms:

1. Exe

Executable.

Runs in its own space.

we can run these .exe files by just opening it, as they have their own address spaces.

Can be directly run by the operating system.

Can include both application code and the resources needed for its execution.

Have a specific entry point like Main method in c#.

These are loaded when the application is launched, this is called static linking.

The exe files are standalone and encapsulate the entire application, so change in exe donot affect other applications.

To access a Program.exe file for Program.dll

csc Program.cs

To access the IL :

ildasm Program.exe

1. Dll

Dynamic link library

Doesn’t run in its own space needs and hoster to consume it.

Not meant to be directly executed. They are loaded at runtime and used in other applications.

The code if wanted to reuse, can be written using DLL and run in the .exe address space.

Promotes code reusability and modularity.

Donot have specific entry point.

DLL are loaded dynamically when needed, so these are called dynamic link libraries.

Dynamic linking allows sharing of the same copy of DLL in multiple applications.

As a DLL is used by multiple applications, change in one DLL can affect the other applications that are using it.

To access a Program.dll file for a Program.cs:

csc /target:library /out:Program.dll Program.cs

To access the Program.dll : ildasm Program.dll

**Parameter Types**

Parameters are the variables that are used in method or constructor to receive values from the caller.

There are different types of parameters:

1. Value parameters:

Receive a copy of the actual argument value

Changes made to the parameter inside the method donot affect the original argument.

void ModifyValue(int x)

{

x = x \* 2;

}

// Usage

int value = 5;

ModifyValue(value);

// 'value' remains 5, not affected by the method

1. Reference parameter

Receive a reference to the actual argument.

Changes made to the reference parameter in the method affects the original argument.

void ModifyReference(ref int x)

{

x = x \* 2;

}

// Usage

int value = 5;

ModifyReference(ref value);

// 'value' is now 10, modified by the method

1. Output parameters

Method is required to assign values to these output parameters before returning.

Used for returning multiple values from the method.

void GetValues(out int a, out int b)

{

a = 10;

b = 20;

}

// Usage

int value1, value2;

GetValues(out value1, out value2);

// 'value1' is 10, 'value2' is 20

1. Params parameters

Allow a method to accept a variable number of arguments to the method.

void Sum(params int[] numbers)

{

int total = 0;

foreach (int num in numbers)

{

total += num;

}

Console.WriteLine(total);

}

// Usage

Sum(1, 2, 3); // Output: 6

Sum(1, 2, 3, 4, 5); // Output: 15

1. Optional parameters

Have default values specified in the method.

If the user does not specify any value, then the default values will be considered.

void PrintNumber(int x = 0)

{

Console.WriteLine(x);

}

// Usage

PrintNumber(); // Output: 0

PrintNumber(5); // Output: 5

1. Array Parameters

Allows a method to accept an array

void ProcessNumbers(int[] numbers)

{

foreach (int num in numbers)

{

Console.WriteLine(num);

}

}

// Usage

int[] values = { 1, 2, 3, 4, 5 };

ProcessNumbers(values);

**Interfaces**

Defines contract for class.

Contains only the abstract methods.

The implementation of these abstract methods must be done by the class implementating this class.

Provides a way to implement abstraction, multiple inheritance and polymorphism.

Allow for more flexibility and modular code.

Properties in interfaces can only have signature but no body.

can have methods, properties, events, and indexers as its members. But interfaces will contain only the declaration of the members.

// Definition of an interface

public interface IShape

{

// Abstract method

double CalculateArea();

// Property

string Name { get; set; }

}

// Implementation of the interface in a class

public class Circle : IShape

{

// Properties from the interface

public string Name { get; set; }

public double Radius { get; set; }

// Constructor

public Circle(double radius)

{

Radius = radius;

Name = "Circle";

}

// Implementation of the abstract method from the interface

public double CalculateArea()

{

return Math.PI \* Math.Pow(Radius, 2);

}

}

**Casting**

Process of converting a one data type to another.

Types of casting:

1. Implicit casting

When conversion is done automatically by compiler.

No loss of data.

Also called as widening as target type has wide range than the source type.

Example:

int intValue = 10;

long longValue = intValue; // Implicit casting from int to long

1. Explicit casting

Converting a type with wide range to a narrow range.

Data loss occurs.

Explicitly to be done by programmer.

Also called narrowing

Example:

double doubleValue = 123.45;

int intValue = (int)doubleValue; // Explicit casting from double to int

**Strings**

Sequence of characters

Fundamental data types

Instances of System.String class.

Immutable ( once created, cant be changed.)

1. Declaration

string myString = "Hello, C#!";

1. Concatenation

string firstName = "John";

string lastName = "Doe";

string fullName = firstName + " " + lastName; // "John Doe"

(or)

string greeting = "Hello, ";

greeting += "world!"; // "Hello, world!"

1. String interpolation

Embed expression directly in strings

string name = "Alice";

int age = 25;

string message = $"Hello, {name}! You are {age} years old.";

1. String length

string text = "Programming";

int length = text.Length; // 11

1. Accessing Characters

string word = "CSharp";

char firstCharacter = word[0]; // 'C'

1. String comparison

== operator compares content and not references

string str1 = "hello";

string str2 = "HELLO";

bool areEqual = str1.Equals(str2, StringComparison.OrdinalIgnoreCase);

// true

1. Formatting

int number = 42;

string formatted = string.Format("The answer is {0}", number);

// "The answer is 42"

1. String literals

Helps in including special characters without escape sequence

string path = @"C:\Program Files\MyApp";

1. String Builder

StringBuilder builder = new StringBuilder();

builder.Append("Hello, ");

builder.Append("world!");

string result = builder.ToString(); // "Hello, world!"

String Methods:

1. Clone

returns another copy of same data

string s1 = "Hello ";

string s2 = (String)s1.Clone();

1. Compare

to compare first string with second string lexicographically.

Returns an integer value

s1==s2 returns 0

s1>s2 returns 1

s1<s2 returns -1

1. Concat

string s1 = "Hello ";

string s2 = "C#";

Console.WriteLine(string.Concat(s1,s2));

1. Contains

return a value indicating whether the specified substring occurs within this string or not.

If the specified substring is found in this string, it returns true otherwise false.

string s1 = "Hello ";

string s2 = "He";

string s3 = "Hi";

Console.WriteLine(s1.Contains(s2));

Console.WriteLine(s1.Contains(s3));

**Statements and Expressions**

Statement:

Complete unit of execution that performs an action.

Types:

1. Expression statement

Expression followed by semicolon.

Perform any action, call method, or also include any valid expression.

int x = 5; // Declaration and assignment statement

Console.WriteLine("Hello, World!"); // Method call statement

x++; // Increment statement

1. Declaration statement

Declare a variable or constant.

Allocates memory for variable and associates a name with the memory location.

End with semicolon.

int x; // Declaration statement

double pi = 3.14; // Declaration and assignment statement

const int MaxValue = 100; // Constant declaration statement

1. Selection statements

executes a block of code if a condition is true; otherwise, it executes another block.

if (x > 0)

{

// Code to execute if the condition is true

}

else

{

// Code to execute if the condition is false

}

1. Loop statements

Repeatedly executes a block of code if a particular condition is true.

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

{

// Code to repeat in the loop

}

1. Break statements

Used for altering the normal flow of control.

break; // Breaks out of a loop

continue; // Skips the rest of the loop and goes to the next iteration

return value; // Exits a method and optionally returns a value

Expression:

a sequence of one or more operands and operators that can be evaluated to produce a single value.

fundamental building block in programming languages and play a crucial role in performing computations, making decisions, and defining data.

1. Arithmetic expression

int result = 3 + 5 \* (2 - 1); // Arithmetic expression

1. Logical expression

bool isTrue = (x > 0) && (y < 10); // Logical expression

1. Relational Expression

relational operators like ==, !=, <, >, <=, >=.

bool isEqual = (a == b); // Relational expression

1. Ternery Expression

int max = (a > b) ? a : b; // Ternary expression

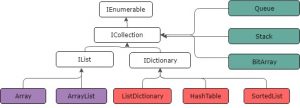
1. Method call expression

SayHello("John");

1. Object creation expression

MyClass obj = new MyClass(); // Object creation expression

**Collections**



Non-generic Generic

ArrayList -------------> List

HashTable -------------> Dictionary

SortedList -------------> SortedList

Stack -------------> Stack

Queue -------------> Queue

Arrays are declared with a specific size of array. So, whenever we want to change the size of array, then we can copy the array into new increased array. Or we can also use the Arry.resize(copying old values to new one with increased size). Array is fixed size.

In array, we cant add a new value in middle of array and deleting or removing values from the middle of array.

So, to overcome these problems in an array, we have the collections also called as dynamic arrays.

In collections, we have auto resizing, and it is possible to insert or delete values from the middle of the array.

Non-generic collections : System.Collection classes ( not strongly typed)

* ArrayList
* HashTable
* Queue
* Stack
* LinkedList

Generic Collections : System.Collection.Generic classes ( strongly typed)

* Dictionary<Tkey,TValue>
* List<T>
* Queue<T>
* Stack<T>
* ArrayList<T>

ArrayList

Methods:

1. Add ( object value)

ArrayList list = new ArrayList();

list.Add(1);

list.Add("Hello");

1. AddRange(ICollection c)

Adds range of elements at the end

ArrayList list = new ArrayList();

list.AddRange(new int[] { 1, 2, 3 });

1. Clear()

Removes all elements from list

ArrayList list = new ArrayList();

list.Add(1);

list.Add(2);

list.Clear(); // Clears all elements

1. Contains(Object value)

Checks if a value is present in list and return true or false

ArrayList list = new ArrayList();

list.Add(1);

bool containsOne = list.Contains(1); // true

1. IndexOf(object value) and LastIndexOf(object value):

Returns the index of the first (or last) occurrence of a specific object in the ArrayList.

ArrayList list = new ArrayList() { 1, 2, 3, 1, 4 };

int firstIndex = list.IndexOf(1); // 0

int lastIndex = list.LastIndexOf(1); // 3

1. Remove(object obj) and RemoveAt(int index):

Removes the first occurrence of a specific object (or an element at a specific index) from the ArrayList.

ArrayList list = new ArrayList() { 1, 2, 3 };

list.Remove(2); // Removes the element 2

list.RemoveAt(0); // Removes the element at index 0 (1)

1. RemoveRange(int index, int count):

Removes a range of elements from the ArrayList.

ArrayList list = new ArrayList() { 1, 2, 3, 4, 5 };

list.RemoveRange(1, 3); // Removes elements from index 1 to 3

1. ToArray()

Copies the elements to a new array

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

int[] array = numbers.ToArray();

Properties:

1. Count

Gets the total number of elements in collection

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

int count = numbers.Count; // 3

1. Capacity

Gets or sets the total capacity of the collection without resizing.

List<int> numbers = new List<int>();

numbers.Capacity = 10;

Linked list

Represents doubly linked list

1. AddFirst(T value):

Adds a new node with the specified value at the beginning of the LinkedList<T>.

LinkedList<int> numbers = new LinkedList<int>();

numbers.AddFirst(1);

1. AddLast(T value)

Adds a new node with specified value at the end of the linked list

1. AddBefore(LinkedListNode<T> node, T value)

Adds the specified value before the specified node.

1. AddAfter(LinkedListNode<T> node, T value)

Adds the specified value after the specified node.

1. Remove(T value):

Removes the first occurrence of the specified value from linked list

1. RemoveFirst():

Removes the first node of the linked list

1. RemoveLast()

Removes the last node of the linked list.

1. Find(T value):

Finds the first node that contains the specified value.

1. Contains(T value):

Determines whether the linked list contains the specified value.

Returns true or false

Properties:

1. First:

Returns the first node of the linked list

1. Last:

Returns the last node of the linked list.

1. Count:

Identifies the count of the linked list.

Stack<t>

LIFO principle

Methods:

1. Push( T item)

Pushes the specified element at the end(top of stack)

1. Pop()

Pops the element at the top of stack.

1. Peek()

Returns the element at the top of stack.

1. Clear()

Clears the stack

1. Contains(t item)

Determines a specified element is present in the stack or not, Returns true or false.

Properties:

1. Count

Stack.Count

Gets the number of elements in the stack.

Queue<t>

FIFO principle

Methods:

1. Enqueue(T item)

Adds the element at the end of the queue

1. Dequeue()

Deleted the element from the start of the queue

1. Peek()

Returns the element at the beginning of the queue without deleting it.

1. Clear()

Deletes the elements from the queue.

1. Contains(T item)

Determines if the specified item is present in the queue, returns a Boolean value.

Properties:

1. Count

Gets the number of elements in the queue.

HashTable

A Hashtable is a collection of key/value pairs that are arranged based on the hash code of the key.

// C# program to illustrate a hashtable

using System;

using System.Collections;

class Hahtable\_Example{

// Main method

static public void Main()

{

// Create a hashtable

// Using Hashtable class

Hashtable my\_hashtable = new Hashtable();

// Adding key/value pair in the hashtable

// Using Add() method

my\_hashtable.Add("A1", "Welcome");

my\_hashtable.Add("A2", "to");

my\_hashtable.Add("A3", "GeeksforGeeks");

foreach(DictionaryEntry element in my\_hashtable)

{

Console.WriteLine("Key:- {0} and Value:- {1} ",

element.Key, element.Value);

}}}

Dictionary<Tkey,TValue>

// C# program to illustrate Dictionary

using System;

using System.Collections.Generic;

class Dictionary\_example{

// Main Method

static public void Main()

{

// Creating a dictionary

// using Dictionary<TKey, TValue> class

Dictionary<string, string> My\_dict =

new Dictionary<string, string>();

// Adding key/value pairs in the Dictionary

// Using Add() method

My\_dict.Add("a.01", "C");

My\_dict.Add("a.02", "C++");

My\_dict.Add("a.03", "C#");

foreach(KeyValuePair<string, string> element in My\_dict)

{

Console.WriteLine("Key:- {0} and Value:- {1}",

element.Key, element.Value);

}

}

}

**Boxing/Unboxing**

Boxing

converting a value type to the type object or to any interface type implemented by the value type.

int intValue = 42;

object boxedValue = intValue; // Boxing

Unboxing

Converting a object reference back to value type.

Reverse of boxing

int unboxedValue = (int)boxedValue; // Unboxing

The boxing and unboxing mainly used in collections.

These can be avoided if using generics.

**Async**

Allows you to write asynchronous code more easily and efficiently.

Asynchronous programming is used in tasks which take much time to complete like the I/0 operations, network requests, etc.

The async keyword is used to declare a method as asynchronous.

It indicates that the method contains an asynchronous operation, and it can be paused and resumed.

public async Task MyAsyncMethod()

{

// Asynchronous code here

}

An asynchronous method typically returns a Task or Task<T>.

Task : ongoing operation

T : represents an operation that produces a result of type <T>

The await keyword is applied to an asynchronous operation (typically a method returning Task or Task<T>).

It allows the method to await the completion of the operation without blocking the execution of the entire application.

Example:

using System;

using System.Threading.Tasks;

class Program

{

static async Task Main()

{

Console.WriteLine("Start");

await MyAsyncMethod();

Console.WriteLine("End");

}

static async Task MyAsyncMethod()

{

Console.WriteLine("Async Method Start");

await Task.Delay(2000); // Simulate an asynchronous operation

Console.WriteLine("Async Method End");

}

}

**Using Keyword**

1. Namespace Import

allows you to use types from a particular namespace without fully qualifying their names.

This is particularly helpful to avoid naming conflicts and to make the code more readable.

using System;

class Program

{

static void Main()

{

// You can now use Console without fully qualifying it

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

}

}

1. Resource Management

The using statement is used for resource management, especially with objects that implement the IDisposable interface. It ensures that the Dispose method of the object is called when the block is exited, even if an exception is thrown.

using (StreamWriter writer = new StreamWriter("example.txt"))

{

writer.WriteLine("Hello, using statement!");

} // Dispose is automatically called when exiting the block

In this example, the StreamWriter is disposed of properly when the using block is exited. The using statement is syntactic sugar for a try-finally block that ensures proper resource cleanup.

1. Alias Directive

ProjectA and ProjectB are aliases for different namespaces, allowing you to use types from both namespaces without conflicts.

using ProjectA = CompanyA.ProjectA;

using ProjectB = CompanyB.ProjectB;

class Program

{

static void Main()

{

ProjectA.SomeClass projectAClass = new ProjectA.SomeClass();

ProjectB.SomeClass projectBClass = new ProjectB.SomeClass();

}

}

1. Static Directive

to import the static members of a type, making them available without specifying the type name.

using static System.Console;

class Program

{

static void Main()

{

WriteLine("Hello, using static!");

}

}

**Arrays**

An array is a collection of elements of the same type that are stored in the same memory location.

Object of base type System.Array

Types of arrays:

1. Single – dimensional array

Is like row of elements.

int[] arr=new int[5];

Or

int[] arr=new int[5]{10,15,20,13,9};

Or

int[] arr={10,15,20,13,9};

Traversing the array:

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

Console.WriteLine(arr[i]);}

1. Multi dimensional array:

2d array

Have rows and columns

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

Or

int[,] arr=new int[3,3]{{1,2,3},{1,2,3},{1,2,3}};

Traversing the array:

for(int i=0;i<3;i++){

for(int j=0;j<3;j++){

Console.WriteLine(arr[i,j]+” “);

}

Console.WriteLine();

}

1. Jagged array

Array of arrays

int[][] arr=new int[2][];

arr[0]=new int[6]{42,61,37,41,59,63};

arr[1]=new int[4]{11,21,56,78};

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

for(j=0;j<arr[i].Length;j++){

Console.WriteLine(arr[i,j]+” ”);

}

Console.WriteLine();

}

Advantages:

* Used to represent numerous data objects of similar type in similar name
* Data structures like linked lists, trees, stacks and queues are implemented using arrays.
* 2d arrays are implemented using matrices.
* Arrays are strongly typed, so performance is increased as no boxing and unboxing is performed.

Disadvantages:

* Array size is fixed, so it is important to declare the size of array.
* If we allocate more memory than the required then extra memory is wasted.
* An element can never be inserted in the middle of the array.

**DOTNET CLI**

developing, building, running, and publishing .NET applications

Creating project

dotnet new console -n MyConsoleApp

Restore the dependencies from previous file

dotnet restore

Build the project and dependencies

dotnet build

Compiles and runs the application

dotnet run

Publishes application for deployment

dotnet publish -c Release

Adds a reference

dotnet add reference MyLibrary.csproj

Adds a package

dotnet add package MyPackage

Removes a reference

dotnet remove reference MyLibrary.csproj

Removes a package

dotnet remove package MyPackage

Clean the output of previous build

dotnet clean

Display info about Dotnet CLI

dotnet help

Display info about installed .Net SDK and runtime

dotnet –info

Display the installed version of .net sdk

dotnet –version

**Nuget**

A visual studio extension which helps to search, locate the library, download them, reference them in our project and also make changes in the App config and Web config files.

It provides a centralized repository of libraries, tools, and other assets that developers can use in their projects.

NuGet simplifies the process of managing and consuming third-party libraries or components in a .NET application.

The NuGet CLI (nuget.exe) allows developers to interact with NuGet from the command line.

Common CLI commands include nuget restore, nuget install, nuget update, etc.

NuGet is tightly integrated with Visual Studio, making it easy for developers to manage packages directly from the IDE.