

**Class Definition:**

A class is a an object blueprint or template for creating objects. It defines a set of attributes (properties) and methods (behaviors) that characterize of that class.

Ex: The car class encapsulates year,model,speed,color,made(properties) and with (methods) accelerate ,brake associated with cars, providing a structured and reusable way to model and interact with car objects in a software system.

Note:

Polymorphism allows objects of different classes to be treated as objects of a common base class.

Uses:

Modularity and Reusability

Abstraction

Organizing Code

Maintainability

**Access modifiers:**

Public: The public keyword allows its members to be visible from anywhere inside the project.

Private: The private members can only be accessed by the member within the same class.

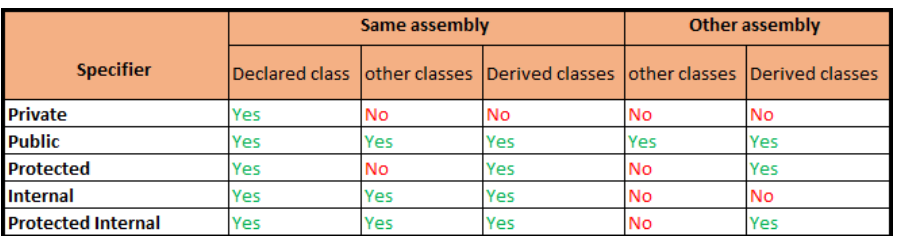
Protected: Protected accessibility allows the member to be accessed from within the class and

from another class that inherits this class.

Internal: Internal provides accessibility from within the project. Another similar internal

accessibility is protected internal. This allows the same as the internal and the only difference is

that a child class can inherit this class and reach its members even from another project



**Virtual keyword:**

The virtual keyword is used to declare a method or property in a base class that can be overridden by a method or property with the same signature in a derived class.

It enables runtime polymorphism by allowing objects of the derived class to be treated as objects of the base class.

When a method is marked as virtual, the decision about which method (base or derived) to call is made at runtime. This is known as late binding or runtime polymorphism. The virtual keyword is commonly used in the design of extensible frameworks where base classes provide default behavior, and derived classes can override or extend that behavior.

using System;

public class Animal

{

// Virtual method that can be overridden by derived classes

public virtual void MakeSound()

{

Console.WriteLine("Generic animal sound");

}

}

// Derived class 1

public class Dog : Animal

{

// Override the MakeSound method in the derived class

public override void MakeSound()

{

Console.WriteLine("Woof! Woof!");

}

}

// Derived class 2

public class Cat : Animal

{

// Override the MakeSound method in the derived class

public override void MakeSound()

{

Console.WriteLine("Meow!");

}

}

class Program

{

static void Main()

{

// Creating objects of the derived classes

Animal animal1 = new Dog();

Animal animal2 = new Cat();

// Calling the overridden method

animal1.MakeSound(); // Outputs: Woof! Woof!

animal2.MakeSound(); // Outputs: Meow!

}

}

**Generics**

It allows our code to be datatype independent. Generics is a programming concept in which algorithms and data structures are written in a way that allows them to work with any data type. This provides flexibility and type safety, allowing developers to create reusable and type-agnostic components.Mostly used in collections,algorithms,typesafety.

using System;

class Box<T>

{

private T item;

public void AddItem(T newItem)

{

item = newItem;

}

public T GetItem()

{

return item;

}

}

class Program

{

static void Main()

{

// Creating a Box for integers

Box<int> intBox = new Box<int>();

intBox.AddItem(42);

Console.WriteLine($"Item in the box: {intBox.GetItem()}");

// Creating a Box for strings

Box<string> stringBox = new Box<string>();

stringBox.AddItem("Hello, Generics!");

Console.WriteLine($"Item in the box: {stringBox.GetItem()}");

}

}

**Delegates:**

It is a type safe function pointer. A delegate in C# is a type that represents references to methods with a specific signature. It allows you to treat methods as first-class citizens, which means you can pass them as parameters, store them in variables, and invoke them dynamically.

Real-world example:

Suppose you have a scenario where you want to calculate the total price of items in a shopping cart. You can define a delegate to represent a method that calculates the price of an individual item, and then use that delegate to calculate the total price.

using System;

// Delegate definition

public delegate void SimpleDelegate(string message);

// SimpleClass that contains methods compatible with the delegate

public class SimpleClass

{

// Method to display a message

public void DisplayMessage(string message)

{

Console.WriteLine($"DisplayMessage: {message}");

}

// Method to show an alert

public void ShowAlert(string message)

{

Console.WriteLine($"ShowAlert: {message}");

}

}

class Program

{

static void Main()

{

SimpleClass simpleObject = new SimpleClass();

// Instantiate the delegate with the DisplayMessage method

SimpleDelegate displayDelegate = new SimpleDelegate(simpleObject.DisplayMessage);

// Instantiate the delegate with the ShowAlert method

SimpleDelegate showAlertDelegate = new SimpleDelegate(simpleObject.ShowAlert);

// Create an instance of SimpleClass

// Invoke the delegate, calling the DisplayMessage method

displayDelegate("Hello, Delegates!");

// Invoke the delegate, calling the ShowAlert method

showAlertDelegate("Important Alert!");

// You can also combine delegates using the "+" operator

SimpleDelegate combinedDelegate = displayDelegate + showAlertDelegate;

// Invoke the combined delegate, calling both methods

combinedDelegate("Combined Invocation");

// Remove a delegate from the combined delegate

combinedDelegate -= displayDelegate;

// Invoke the modified combined delegate

combinedDelegate("After Removing DisplayMessage");

}

}

Multicasting in C# allows a delegate to reference and invoke multiple methods. It involves combining two or more delegates using the + or - operators. Here's an example of how multicast delegates work:

Anonymous delegates are useful when you need to provide a short, one-time implementation for a delegate without creating a separate method. They are particularly common when working with events and asynchronous programming in C#.

**Events**

In C#, events are a powerful mechanism for implementing the observer pattern, enabling communication between components in a loosely coupled manner. Events are often used in scenarios where one part of the application (the publisher) needs to notify other parts (subscribers) when something of interest happens.

Events provide a mechanism for decoupling components, allowing them to communicate without having explicit knowledge of each other. This enhances code modularity, maintainability, and flexibility in various scenarios.

**out keyword:**

out keyword is used in method parameters to indicate that the parameter is being passed by reference and is used for output purposes. It allows a method to return multiple values through its parameters. When a parameter is marked with out, it doesn't need to be initialized before being passed to the method, and the method is responsible for assigning a value to it. out keyword to cover various aspects, including passing parameters by reference, initialization inside a method, and returning multiple values.

using System;

public class OutKeywordExample

{

// Method to calculate sum and product and return the results using 'out'

public static void CalculateSumAndProduct(int a, int b, out int sum, out int product)

{

// Initialization of 'sum' and 'product' is required before returning

sum = a + b;

product = a \* b;

}

}

class Program

{

static void Main()

{

int num1 = 5, num2 = 7;

// Declare variables to receive values from the method using 'out'

int resultSum, resultProduct;

// Call the method with 'out' parameters

OutKeywordExample.CalculateSumAndProduct(num1, num2, out resultSum, out resultProduct);

// Display the results

Console.WriteLine($"Sum: {resultSum}, Product: {resultProduct}");

}

}

**Parameters:**

parameters play a crucial role in defining and passing information to methods. They are used to receive input values, provide data for method execution, and facilitate communication between different parts of a program.

They are value parameters, Reference parameters ,Output parameters, Prams parameters ,Default parameters, named arguments.

int a = 5, b = 10;

ExampleMethod(a); // Passing by value

ExampleMethod(ref a); // Passing by reference (ref)

ExampleMethod(out b); // Passing by reference (out)

ExampleMethodWithDefault(5); // Using a named argument

**Params parameter:**

Params is a keyword which is used to specify a parameter that takes variable number of arguments. It is useful when we don't know the number of arguments prior. Only one params keyword is allowed and no additional parameter is permitted after params keyword in a function declaration.

using System;

namespace AccessSpecifiers

{

class Program

{

// User defined function

public void Show(params object[] items) // Params Paramater

{

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

{

Console.WriteLine(items[i]);

}

}

// Main function, execution entry point of the program

static void Main(string[] args)

{

Program program = new Program(); // Creating Object

program.Show("Ramakrishnan Ayyer","Ramesh",101, 20.50,"Peter", 'A'); // Passing arguments of variable length

}

}

}

**Interface:**

Interfaces provide a way to define a contract for a class without specifying the implementation details. Interface in C# is a blueprint of a class. It is like abstract class because all the methods which are declared inside the interface are abstract methods. It cannot have method body and cannot be instantiated.

It is used to achieve multiple inheritance which can't be achieved by class. It is used to achieve fully abstraction because it cannot have method body. This allows for better separation of concerns and promotes code flexibility.

**Example:**

Database Connectors

Imagine a system that needs to support multiple database types (SQL, NoSQL). By defining a common interface for database connectors, you can create plugins for each specific database, ensuring a consistent way to interact with different data stores.

Strategy pattern:

Example: Sorting Algorithms

When designing a sorting algorithm, you might define an interface for sorting strategies. Different algorithms (quicksort, bubblesort) can then implement this interface.

**Expressions**

An expression is a piece of code that produces a value. It can be a simple value, a combination of values, or a call to a method that returns a value.

// Arithmetic expression

int sum = 5 + 3;

// Method call expression

string message = GetMessage();

// Conditional expression (ternary operator)

bool isEven = (sum % 2 == 0) ? true : false;

// Object creation expression

Person person = new Person("John", 25);

**Statements:**

A statement is a complete line of code that performs a specific action. Statements are typically composed of expressions and control flow constructs. statements are executed sequentially. Statements are crucial for directing the flow of the program.

// Variable declaration statement

int x = 10;

// Assignment statement

x = x + 5;

// Conditional statement (if)

if (x > 15)

{

Console.WriteLine("x is greater than 15");

}

else

{

Console.WriteLine("x is not greater than 15");

}

// Loop statement (for)

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

{

Console.WriteLine(i);

}

// Method call statement

PrintMessage("Hello, World!");

The main similarity between expressions and statements is that they’re both executed in computer programs.

A black background with a black square

Description automatically generated with medium confidence **Differences**

**Type casting:**

Implicit Casting:

Implicit casting is performed by the compiler automatically when there is no risk of losing data. It's also known as widening or upcasting.

Explicit Casting:

Explicit casting is required when there is a risk of losing data. It's performed using the cast operator and is known as narrowing or downcasting.

Type Conversion Methods:

C# provides methods for explicit conversion between types. These methods are available on numeric types and can be used for more complex conversions.

double doubleValue = 10.5;

int intValue = Convert.ToInt32(doubleValue);

Boxing and Unboxing:

Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type. Unboxing is the reverse process.

is and as Operators:

The is operator checks if an object is compatible with a given type, and the as operator performs a safe cast.

object someObject = "Hello";

if (someObject is string)

{

string stringValue = (string)someObject;

}

// Using 'as' for safe casting

object anotherObject = "World";

string anotherStringValue = anotherObject as string;

if (anotherStringValue != null)

{

// The cast was successful

}

User-Defined Type Conversions:

You can define implicit or explicit user-defined conversions by overloading conversion operators in your custom classes.

**Using Keyword:**

Resource Management

The using statement is used to ensure that the resources are properly disposed of, even if an exception occurs. It is commonly used with types that implement the IDisposable interface.

Namespace Alias

The using directive is used to create an alias for a namespace or to import types from a namespace, making it easier to use those types in your code.

Multiple using Directives:

You can use multiple using directives in a file to import types from different namespaces.

**Arrays:**

An array is a collection of elements of the same type stored in contiguous memory locations. Arrays provide a way to group related data items under a single name.

Advantages:

Code Optimization (less code)

Random Access

Easy to traverse data

Easy to manipulate data

Easy to sort data etc.

Disadvantages

Fixed size Code Optimization (less code)

**Single Dimension array:**

int[] arr = new int[5];//creating array

int arr[] = new int[5];//compile time error

There are 3 ways to initialize array at the time of declaration.

int[] arr = new int[5]{ 10, 20, 30, 40, 50 };

We can omit the size of array.

int[] arr = new int[]{ 10, 20, 30, 40, 50 };

We can omit the new operator also.

int[] arr = { 10, 20, 30, 40, 50 };

To reuse the array logic, we can create function. To pass array to function in C#, we need to provide only array name.

using System;

public class ArrayExample

{

static void printArray(int[] arr)

{

Console.WriteLine("Printing array elements:");

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

{

Console.WriteLine(arr[i]);

}

}

public static void Main(string[] args)

{

int[] arr1 = { 25, 10, 20, 15, 40, 50 };

int[] arr2 = { 12, 23, 44, 11, 54 };

printArray(arr1);//passing array to function

printArray(arr2);

}

}

**Multidimensional array:**

The multidimensional array is also known as rectangular arrays in C#. It can be two dimensional or three dimensional. The data is stored in tabular form (row \* column) which is also known as matrix.

To create multidimensional array, we need to use comma inside the square brackets. For example:

int[,] arr=new int[3,3];//declaration of 2D array

int[,,] arr=new int[3,3,3];//declaration of 3D array

int[,] arr = new int[3,3]= { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

or

int[,] arr = new int[,]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

or

int[,] arr = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

**Jagged arrays:**

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

Initialization of Jagged array

arr[0] = new int[4];

arr[1] = new int[6];

public class JaggedArrayTest

{

public static void Main()

{

int[][] arr = new int[3][]{

new int[] { 11, 21, 56, 78 },

new int[] { 2, 5, 6, 7, 98, 5 },

new int[] { 2, 5 }

};

// Traverse array elements

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

{

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

{

System.Console.Write(arr[i][j]+" ");

}

System.Console.WriteLine();

}

}

}

**Strings**

Sequence of characters

Fundamental data types

Instances of System.String class.

Immutable ( once created, cant be changed.)

Declaration

string myString = "Hello, C#!";

Concatenation

string firstName = "John";

string lastName = "Doe";

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

(or)

string greeting = "Hello, ";

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

String interpolation

Embed expression directly in strings

string name = "Alice";

int age = 25;

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

String length

string text = "Programming";

int length = text.Length; // 11

Accessing Characters

string word = "CSharp";

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

String comparison

== operator compares content and not references

string str1 = "hello";

string str2 = "HELLO";

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

// true

Formatting

int number = 42;

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

// "The answer is 42"

String literals

Helps in including special characters without escape sequence

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

String Builder

StringBuilder builder = new StringBuilder();

builder.Append("Hello, ");

builder.Append("world!");

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

**String Methods:**

**Clone**

returns another copy of same data

string s1 = "Hello ";

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

**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

**Concat**

string s1 = "Hello ";

string s2 = "C#";

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

**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));

**Collections:**

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 collection**s : 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:

Add ( object value)

ArrayList list = new ArrayList();

list.Add(1);

list.Add("Hello");

AddRange(ICollection c)

Adds range of elements at the end

ArrayList list = new ArrayList();

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

Clear()

Removes all elements from list

ArrayList list = new ArrayList();

list.Add(1);

list.Add(2);

list.Clear(); // Clears all elements

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

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

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)

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

ToArray()

Copies the elements to a new array

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

int[] array = numbers.ToArray();

Properties:

Count

Gets the total number of elements in collection

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

int count = numbers.Count; // 3

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

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);

AddLast(T value)

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

AddBefore(LinkedListNode<T> node, T value)

Adds the specified value before the specified node.

AddAfter(LinkedListNode<T> node, T value)

Adds the specified value after the specified node.

Remove(T value):

Removes the first occurrence of the specified value from linked list

RemoveFirst():

Removes the first node of the linked list

RemoveLast()

Removes the last node of the linked list.

Find(T value):

Finds the first node that contains the specified value.

Contains(T value):

Determines whether the linked list contains the specified value.

Returns true or false

Properties:

First:

Returns the first node of the linked list

Last:

Returns the last node of the linked list.

Count:

Identifies the count of the linked list.

**Stack<t>**

**LIFO principle**

Methods:

Push( T item)

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

Pop()

Pops the element at the top of stack.

Peek()

Returns the element at the top of stack.

Clear()

Clears the stack

Contains(t item)

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

Properties:

Count

Stack.Count

Gets the number of elements in the stack.

**Queue<t>**

**FIFO principle**

Methods:

Enqueue(T item)

Adds the element at the end of the queue

Dequeue()

Deleted the element from the start of the queue

Peek()

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

Clear()

Deletes the elements from the queue.

Contains(T item)

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

Properties:

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 Hashtable\_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 means it allows the program to perform multiple tasks concurrently without waiting for each one to complete before moving on to the next.

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");

}

}