**Interface**

An interface looks like a class, but has no implementation. The only thing it contains are declarations/signatures of methods, properties, indexers & events. A class or struct that implements the interface must implement the members of the interface that are specified in the interface definition.

Interfaces can contain methods, properties, events, indexers, or any combination of those four member types. An interface can't contain constants, fields, operators, instance constructors, finalizers, or types. Interface members are automatically public, and they can't include any access modifiers. Members also can't be [static](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static).

To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.

When a class or struct implements an interface, the class or struct must provide an implementation for all of the members that the interface defines. The interface itself provides no functionality that a class or struct can inherit in the way that it can inherit base class functionality. However, if a base class implements an interface, any class that's derived from the base class inherits that implementation.

**Interfaces Summary**

An interface has the following properties:

* An interface is like an abstract base class. Any class or struct that implements the interface must implement all its members.
* An interface can't be instantiated directly Its members are implemented by any class or struct that implements the interface.
* Interfaces can contain events, indexers, methods, and properties.
* Interfaces contain no implementation of methods.
* A class or struct can implement multiple interfaces. A class can inherit a base class and implement one or more interfaces.

using System;

interface IValue

{

int Count { get; set; } // Property interface.

string Name { get; set; } // Property interface.

}

class Image : IValue // Implements interface.

{

int \_Count;

public int Count // Property implementation.

{

get { return \_Count; }

set { \_Count = value; }

}

string \_name;

public string Name // Property implementation.

{

get { return \_name; }

set { \_name = value; }

}

}

class Article : IValue // Implements interface.

{

int \_Count;

public int Count // Property implementation.

{

get { return \_Count; }

set { \_Count = value; }

}

string \_name;

public string Name // Property implementation.

{

get { return \_name; }

set { \_name = value; }

}

}

class Program

{

static void Main()

{

IValue value1 = new Image();

IValue value2 = new Article();

value1.Count = 1;

value2.Count = 2;

value1.Name = "Mona Lisa"; // Use setter on interface.

value2.Name = "Resignation"; // Set.

Console.WriteLine(value1.Count); // Use getter on interface.

Console.WriteLine(value2.Count); // Use getter on interface.

Console.WriteLine(value1.Name); // Use getter on interface.

Console.WriteLine(value2.Name); // Get.

Console.ReadKey();

}

}

**Output**

1

2

Mona Lisa

Resignation

using System;

interface IBankAccount

{

bool Deposit(decimal amount);

bool Withdraw(decimal amount);

decimal Balance { get; }

}

public class SavingAccount : IBankAccount

{

private decimal \_Balance;

public bool Deposit(decimal amount)

{

\_Balance += amount;

Console.WriteLine(String.Format("Successfully Diposit in Saving A/C: {0,6:C}", amount));

return true;

}

public bool Withdraw(decimal amount)

{

if (\_Balance < amount)

{

Console.WriteLine("Insufficient balance in Saving A/C!");

return false;

}

else

{

\_Balance -= amount;

Console.WriteLine(String.Format("Successfully withdraw in Saving A/C: {0,6:C}", amount));

return true;

}

}

public decimal Balance

{

get { return \_Balance; }

}

public override string ToString()

{

return String.Format("Saving Account Balance in Saving A/C = {0,6:C}", \_Balance);

}

}

public class CurrentAccount : IBankAccount

{

private decimal \_Balance;

public bool Deposit(decimal amount)

{

\_Balance += amount;

Console.WriteLine(String.Format("Successfully Diposit in Current A/C: {0,6:C}", amount));

return true;

}

public bool Withdraw(decimal amount)

{

if (\_Balance < amount)

{

Console.WriteLine("Insufficient balance in Current A/C!");

return false;

}

else

{

\_Balance -= amount;

Console.WriteLine(String.Format("Successfully withdraw in Current A/C: {0,6:C}", amount));

return true;

}

}

public decimal Balance

{

get { return \_Balance; }

}

public override string ToString()

{

return String.Format("Current Account Balance in Current A/C= {0,6:C}", \_Balance);

}

}

class Program

{

static void Main()

{

IBankAccount savingAccount = new SavingAccount();

IBankAccount currentAccount = new CurrentAccount();

savingAccount.Deposit(200);

savingAccount.Withdraw(100);

Console.WriteLine(savingAccount.ToString());

currentAccount.Deposit(500);

currentAccount.Withdraw(600);

currentAccount.Deposit(500);

currentAccount.Withdraw(600);

Console.WriteLine(currentAccount.ToString());

Console.ReadKey();

}

}

**Output**

Successfully Diposit in Saving A/C: $200.00

Successfully withdraw in Saving A/C: $100.00

Saving Account Balance in Saving A/C = $100.00

Successfully Diposit in Current A/C: $500.00

Insufficient balance in Current A/C!

Successfully Diposit in Current A/C: $500.00

Successfully withdraw in Current A/C: $600.00

Current Account Balance in Current A/C= $400.00

**Abstract Class**

The [abstract](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract) keyword enables you to create classes and [class](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/class) members that are incomplete and must be implemented in a derived class.

abstract class ShapesClass

{

}

An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share.

For example, a class library may define an abstract class that is used as a parameter to many of its functions, and require programmers using that library to provide their own implementation of the class by creating a derived class.

abstract class ShapesClass

{

abstract public int Area();

}

Abstract methods have no implementation, so a semicolon instead of a normal method block follows the method definition. Derived classes of the abstract class must implement all abstract methods.

Example:

using System;

namespace AbstractApplication

{

abstract class ShapesClass

{

abstract public int Area();

}

class Square : ShapesClass

{

int side = 0;

public Square(int n)

{

side = n;

}

// Area method is required to avoid

// a compile-time error.

public override int Area()

{

return side \* side;

}

static void Main()

{

Square sq = new Square(12);

Console.WriteLine("Area of the square = {0}", sq.Area());

Console.ReadKey();

}

}

}

Output:

Area of the square = 144

**Constructor**

Constructor is a special method of a class, which will invoke automatically whenever instance or object of class is created. Constructors are responsible for object initialization and memory allocation of its class. If we create any class without constructor, the compiler will automatically create one default constructor for that class. There is always at least one constructor in every class.

Here you need to remember that a class can have any number of constructors, constructors don’t have any return type, not even void, and within a class we can create only one static constructor.

Generally, constructor name should be same as class name. If we want to create constructor in a class, we need to create a constructor method name same as class name.

**Some of the key points regarding the Constructor are:**

* A class can have any number of constructors.
* A constructor doesn't have any return type, not even void.
* A static constructor cannot be a parametrized constructor.
* Within a class you can create only one static constructor.

**Constructors can be divided into 5 types:**

* Default Constructor
* Parametrized Constructor
* Copy Constructor
* Static Constructor
* Private Constructor

Now let us see each constructor type with example as below

**Default Constructor**

A constructor without any parameters is called a default constructor; in other words, this type of constructor does not take parameters. The drawback of a default constructor is that every instance of the class will be initialized to the same values and it is not possible to initialize each instance of the class to different values. The default constructor initializes:

1. All numeric fields in the class to zero.
2. All string and object fields to null.

using System;

namespace DefaultConstructor

{

class Sample

{

int a, b;

public Sample() //default contructor

{

a = 100;

b = 175;

}

public static void Main()

{

Sample obj = new Sample(); //an object is created , constructor is called

Console.WriteLine(obj.a);

Console.WriteLine(obj.b);

Console.Read();

}

}

}

**Output**

100

175

**Parameterized Constructor**

A constructor with at least one parameter is called a parametrized constructor. The advantage of a parametrized constructor is that you can initialize each instance of the class to different values.

using System;

namespace Constructor

{

class Sample

{

public int a, b;

public Sample(int x, int y) // decalaring Paremetrized Constructor with ing x,y parameter

{

a = x;

b = y;

}

}

class MainClass

{

static void Main()

{

Sample obj = new Sample(100, 175); // Creating object of Parameterized Constructor and ing values

Console.WriteLine("Value of a=" + obj.a);

Console.WriteLine("Value of b=" + obj.b);

Console.Read();

}

}

}

**Output**

Value of a=100

Value of b=175

**Constructor Overloading**

In C #, we can overload constructor by creating another constructor with same method name and different parameters like as shown below

using System;

namespace ConsoleApplication

{

class Sample

{

public string param1, param2;

public Sample() // Default Constructor

{

param1 = "Hi";

param2 = "I am Default Constructor";

}

public Sample(string x, string y) // Declaring Parameterized constructor with Parameters

{

param1 = x;

param2 = y;

}

}

class Program

{

static void Main(string[] args)

{

Sample obj = new Sample(); // Default Constructor will Called

Sample obj1 = new Sample("Welcome", "Kamal"); // Parameterized Constructor will Called

Console.WriteLine(obj.param1 + ", " + obj.param2);

Console.WriteLine(obj1.param1 + ", " + obj1.param2);

Console.ReadLine();

}

}

}

**Output**

Hi, I am Default Constructor

Welcome, Kamal

**Copy Constructor**

A parameterized constructor that contains a parameter of same class type is called as copy constructor. Main purpose of copy constructor is to initialize new instance to the values of an existing instance.

using System;

namespace ConsoleApplication

{

class Sample

{

public string param1, param2;

public Sample(string x, string y)

{

param1 = x;

param2 = y;

}

public Sample(Sample obj) // Copy Constructor

{

param1 = obj.param1;

param2 = obj.param2;

}

}

class Program

{

static void Main(string[] args)

{

Sample obj = new Sample("Welcome", "Kamal"); // Create instance to class Sample

Sample obj1 = new Sample(obj); // Here obj details will copied to obj1

Console.WriteLine(obj1.param1 + ", " + obj1.param2);

Console.ReadLine();

}

}

}

**Output**

Welcome, Kamal

**Static Constructor**

When a constructor is created as static, it will be invoked only once for all of instances of the class and it is invoked during the creation of the first instance of the class or the first reference to a static member in the class. A static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.

Some key points of a static constructor is:

* A static constructor does not take access modifiers or have parameters.
* A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.
* A static constructor cannot be called directly.
* The user has no control on when the static constructor is executed in the program.
* A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.

using System;

namespace ConsoleApplication

{

class Sample

{

public string param1, param2;

static Sample()

{

Console.WriteLine("Static Constructor");

}

public Sample()

{

param1 = "Sample";

param2 = "Instance Constructor";

}

}

class Program

{

static void Main(string[] args)

{

// Here Both Static and instance constructors are invoked for first instance

Sample obj = new Sample();

Console.WriteLine(obj.param1 + " " + obj.param2);

// Here only instance constructor will be invoked

Sample obj1 = new Sample();

Console.WriteLine(obj1.param1 + " " + obj1.param2);

Console.ReadLine();

}

}

}

**Output**

Static Constructor

Sample Instance Constructor

Sample Instance Constructor

**Private Constructor**

When a constructor is created with a private specifier, it is not possible for other classes to derive from this class, neither is it possible to create an instance of this class. They are usually used in classes that contain static members only. Some key points of a private constructor are:

1. One use of a private constructor is when we have only static members.
2. If we want to create object of class even if we have private constructors, then we need to have public constructor along with private.
3. Once we provide a constructor that is either private or public or any, the compiler will not allow us to add public constructor without parameters to the class.

using System;

namespace ConsoleApplication

{

public class Sample

{

public string param1, param2;

public Sample(string a, string b)

{

param1 = a;

param2 = b;

}

private Sample() // Private Constructor Declaration

{

Console.WriteLine("Private Constructor with no prameters");

}

}

class Program

{

static void Main(string[] args)

{

// Here we don't have chance to create instace for private constructor

Sample obj = new Sample("Welcome", "to Fresh Code Hub");

Console.WriteLine(obj.param1 + " " + obj.param2);

Console.ReadLine();

}

}

}

**Output**

Welcome to Fresh Code Hub

In above method we can create object of class with parameters will work fine. If create object of class without parameters, it will not allow us create.

// it will works fine

Sample obj = new Sample(("Welcome", "to Fresh Code Hub");

// it will not work because of inaccessability

Sample obj=new Sample();

**Generics**

Generics are the most powerful feature of C# 2.0. Generics allow you to define type-safe data structures, without committing to actual data types. This results in a significant performance boost and higher quality code, because you get to reuse data processing algorithms without duplicating type-specific code.

Generics refers to the technique of writing the code for a class without specifying the data type(s) that the class works on.

You specify the data type when you declare an instance of a generic class. This allows a generic class to be specialized for many different data types while only having to write the class once.

A great example are the many collection classes in .NET. Each collection class has its own implementation of how the collection is created and managed. But they use generics to allow their class to work with collections of any type.

### **Type Safety**

Generics are mostly used for collections but the framework class library also has non-generic collection classes like ArrayList, Hashtable, SortedList, Stack, and Queue. Therefore, first, we see an example of a non-generic collection class. In other words, ArrayList where we add integer values to the array list and perform addition operation on the list values.

using System;

using System.Collections;

namespace TypeSafety

{

class ArrayListOperation

{

public int Addition()

{

ArrayList list = new ArrayList();

list.Add(1);

list.Add(2);

list.Add(3);

int result = 0;

foreach (int value in list)

{

result += value; //Addition of value

}

return result;

}

}

class Program

{

static void Main(string[] args)

{

ArrayListOperation obj = new ArrayListOperation();

Console.WriteLine(obj.Addition());

Console.ReadKey();

}

}

}

Output

6

As you saw in the previous example there is an array list of integer values and get the result as expected but now add one more value to the ArrayList that the data type is float and perform the same Addition operation.

using System;

using System.Collections;

namespace TypeSafety

{

class ArrayListOperation

{

public int Addition()

{

ArrayList list = new ArrayList();

list.Add(1);

list.Add(2);

list.Add(3);

list.Add(4.5);

int result = 0;

foreach (int value in list)

{

result += value; //Addition of value

}

return result;

}

}

class Program

{

static void Main(string[] args)

{

ArrayListOperation obj = new ArrayListOperation();

Console.WriteLine(obj.Addition());

Console.ReadKey();

}

}

}

In the code above, all three values are easily added to the array list because the ArrayList class Add() method values are an object type, but when retrieving the values using each statement, each value will be assigned an int data type variable. However, this array list has a combination of both integer and float type values and float values won't cast to an int implicitly. That is why the code will give the exception "Specified cast is not valid." That means that an ArrayList is not type safe. This also means that an ArrayList can be assigned a value of any type.

Generics allow you to realize type safety at compile time. They allow you to create a data structure without committing to a specific data type. When the data structure is used, however, the compiler ensures that the types used with it are consistent for type safety. Generics provide type safety, but without any loss of performance or code bloat. The System.Collections.Generics namespace contains the generics collections. Now let's see an example with a generic collection List.

using System;

using System.Collections;

using System.Collections.Generic;

namespace TypeSafety

{

class ListOperation

{

public int Addition()

{

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

list.Add(5);

list.Add(9);

int result = 0;

foreach (int value in list)

{

result += value;

}

return result;

}

}

class Program

{

static void Main(string[] args)

{

ListOperation obj = new ListOperation();

Console.WriteLine(obj.Addition());

Console.ReadKey();

}

}

}

**Performance**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Diagnostics;

namespace Performance

{

class Program

{

static void Main(string[] args)

{

NonGenericPerformance();

GenericPerformance();

Console.ReadKey();

}

static void NonGenericPerformance()

{

long operationTime = 0;

ArrayList arraylist = new ArrayList();

Stopwatch sw = new Stopwatch();

sw.Start();

for (int i = 1; i <= 100000; i++)

{

arraylist.Add(i);

}

operationTime = sw.ElapsedMilliseconds;

Console.WriteLine("Array List {0} values add time is {1} milliseconds", arraylist.Count, operationTime);

sw.Restart();

foreach (int i in arraylist)

{

int value = i;

}

operationTime = sw.ElapsedMilliseconds;

Console.WriteLine("Array List {0} values retrieve time is {1} milliseconds", arraylist.Count, operationTime);

}

static void GenericPerformance()

{

long operationTime = 0;

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

Stopwatch sw = new Stopwatch();

sw.Start();

for (int i = 1; i <= 100000; i++)

{

list.Add(i);

}

operationTime = sw.ElapsedMilliseconds;

Console.WriteLine("List {0} values add time is {1} milliseconds", list.Count, operationTime);

sw.Restart();

foreach (int i in list)

{

int value = i;

}

operationTime = sw.ElapsedMilliseconds;

Console.WriteLine("List {0} values retrieve time is {1} milliseconds", list.Count, operationTime);

}

}

}

**Output**

Array List 100000 values add time is 1 milliseconds

Array List 100000 values retrieve time is 1 milliseconds

List 100000 values add time is 0 milliseconds

List 100000 values retrieve time is zero milliseconds

**How to reuse code-using generics?**

Create Table tbl\_Generic

(

Id int identity(1,1) primary key,

Title nvarchar(50) not null,

Type int not null

)

Insert Into tbl\_Generic (Title,Type)

values ('Jaipur',1),('Jhunjhunu',1),('Cricket',2),('Football',2),('Male',3),('Female',3)

<p>Places : <asp:DropDownList ID="ddlNativePlace" runat="server"></asp:DropDownList></p>

<p>Hobbies : <asp:CheckBoxList ID="chkHobbies" runat="server"></asp:CheckBoxList></p>

<p>Gender : <asp:RadioButtonList ID="rbGender" runat="server"></asp:RadioButtonList></p>

using System;

using System.Web.UI;

using System.Web.UI.WebControls;

using System.Configuration;

using System.Data.SqlClient;

using System.Data;

namespace GenericUIList

{

public partial class \_Default : System.Web.UI.Page

{

protected void Page\_Load(object sender, EventArgs e)

{

if (!Page.IsPostBack)

{

PopulateUIList<DropDownList>(ddlNativePlace, Types.NativePlace);

PopulateUIList<CheckBoxList>(chkHobbies, Types.Hobby);

PopulateUIList<RadioButtonList>(rbGender, Types.Gender);

}

}

public enum Types

{

NativePlace = 1,

Hobby = 2,

Gender = 3

}

public void PopulateUIList<T>(T list, Types type) where T : ListControl

{

string connectionString = ConfigurationManager.ConnectionStrings["genericConn"].ConnectionString;

using (SqlConnection con = new SqlConnection(connectionString))

{

if (con.State == ConnectionState.Closed)

{

con.Open();

}

string cmdText = "Select Id,Title from Master Where Type = @type";

using (SqlCommand cmd = new SqlCommand(cmdText, con))

{

cmd.Parameters.Add(new SqlParameter("@type", (int)type));

DataTable dt = new DataTable();

IDataReader dr = cmd.ExecuteReader();

dt.Load(dr);

list.DataSource = dt;

list.DataTextField = "Title";

list.DataValueField = "Id";

list.SelectedIndex = 0;

list.DataBind();

}

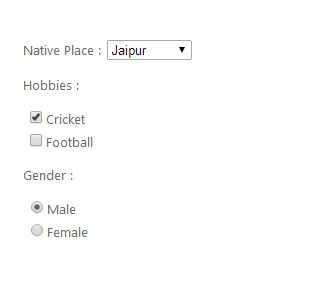
}

}

}

}

**Output**



**Delegates**

A delegate in C# is similar to a function pointer in C or C++. Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code which can call the referenced method, without having to know at compile time which method will be invoked. Unlike function pointers in C or C++, delegates are object-oriented, type-safe, and secure.

Delegates have the following properties:

1. Delegates are similar to C++ function pointers, but are type safe.
2. Delegates allow methods to be passed as parameters.
3. Delegates can be used to define callback methods.
4. Delegates can be chained together; for example, multiple methods can be called on a single event.
5. Methods don't need to match the delegate signature exactly.
6. Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code that can call the referenced method, without having to know at compile time which method will be invoked.

**What is the use of Delegates?**

Suppose if you have multiple methods with same signature (return type & number of parameters) and want to call all the methods with single object then we can go for delegates.

Example:

using System;

delegate void MyDelegate(string s);

class MyClass

{

public static void Hello(string s)

{

Console.WriteLine(" Hello, {0}!", s);

}

public static void Goodbye(string s)

{

Console.WriteLine(" Goodbye, {0}!", s);

}

public static void Main()

{

MyDelegate a, b, c, d;

// Create the delegate object a that references

// the method Hello:

a = new MyDelegate(Hello);

// Create the delegate object b that references

// the method Goodbye:

b = new MyDelegate(Goodbye);

// The two delegates, a and b, are composed to form c:

c = a + b;

// Remove a from the composed delegate, leaving d,

// which calls only the method Goodbye:

d = c - a;

Console.WriteLine("Invoking delegate a:");

a("A");

Console.WriteLine("Invoking delegate b:");

b("B");

Console.WriteLine("Invoking delegate c:");

c("C");

Console.WriteLine("Invoking delegate d:");

d("D");

Console.ReadKey();

}

}

**Output**

Invoking delegate a:

Hello, A!

Invoking delegate b:

Goodbye, B!

Invoking delegate c:

Hello, C!

Goodbye, C!

Invoking delegate d:

Goodbye, D!

Example:

using System;

public delegate void MultiDelegate(int a, int b);

public class Sampleclass

{

public static void Add(int x, int y)

{

Console.WriteLine("Addition Value: " + (x + y));

}

public static void Sub(int x, int y)

{

Console.WriteLine("Subtraction Value: " + (x - y));

}

public static void Mul(int x, int y)

{

Console.WriteLine("Multiply Value: " + (x \* y));

}

}

class Program

{

static void Main(string[] args)

{

Sampleclass sc = new Sampleclass();

MultiDelegate del = Sampleclass.Add;

del += Sampleclass.Sub;

del += Sampleclass.Mul;

del(10, 5);

Console.ReadLine();

}

}

**Output**

Addition Value : 15

Subtraction Value : 5

Multiply Value : 50

**Introduction to Object Oriented Programming Concepts (OOP)**

It is a methodology to write the program where we specify the code in form of classes and objects.

The main advantages and goals of OOP are to make complex software faster to develop and easier to maintain. OOP enables the easy reuse of code by applying simple and widely accepted rules (principles).

**Class**

A class is a group of related methods and variables.

public class student

{

}

**Object**

An object is an instance of a class through which we access the methods of that class. “New” keyword is used to create an object. A class that creates an object in memory will contain the information about the methods, variables and behavior of that class.

student objstudent = new student();

According to the above sample, we can say that Student object, named objstudent, has created out of the student class.

**Methods**

A method is an action that an object can perform.

class SampleClass

{

public int sampleMethod(string sampleParam)

{

// Insert code here

}

}

**Finalizers**

Finalizers are used to destruct instances of classes. In the .NET Framework, the garbage collector automatically manages the allocation and release of memory for the managed objects in your application. However, you may still need finalizers to clean up any unmanaged resources that your application creates. There can be only one finalizers for a class.

**Access Modifiers and Access Levels**

All classes and class members can specify what access level they provide to other classes by using access modifiers. Access Specifiers defines the scope of a class member.

The following access modifiers are available:

**public** :- The type or member can be accessed by any other code in the same assembly or another assembly that references it.

**private** :- The type or member can only be accessed by code in the same class.

**protected** :- The type or member can only be accessed by code in the same class or in a derived class.

**internal** :- The type or member can be accessed by any code in the same assembly, but not from another assembly.

**protected internal** :- The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.

Example:

using System;

namespace Protected\_Internal

{

class access

{

// String Variable declared as protected internal

protected internal string name;

public void print()

{

Console.WriteLine("\nMy name is " + name);

}

}

class Program

{

static void Main(string[] args)

{

access ac = new access();

Console.Write("Enter your name:\t");

// Accepting value in protected internal variable

ac.name = Console.ReadLine();

ac.print();

Console.ReadLine();

}

}

}

**Encapsulation**

Encapsulation is a process of binding data members (variables, properties) and member functions (methods) into a single unit.

**Encapsulation Summary**

* Through encapsulation, a class can hide the internal details of how an object does something. Encapsulation solves the problem at the implementation level.
* A class or structure can specify how accessible each of its members (variables, properties, and methods) is to code outside of the class or structure. Encapsulation simplifies the interaction between objects. An object can use another object without knowing all its data or how its data is maintained. For example, a Client object might have name, address, company, and department properties. If a Bank object wants to use a Client object, it can request the name and address for the bank without needing to know the company and department details of the Client object.
* With the help of encapsulation, a class can change the internal implementation without hurting the overall functionality of the system.
* Encapsulation protects abstraction.

Ways to achieve encapsulation with code example

1. By using the get and set methods (Accessors and Mutators)

using System;

namespace Encaptulation

{

public class Account

{

private string accoutName;

// get methods

public string GetAccount()

{

return accoutName;

}

// Set method

public void SetAccount(string name)

{

accoutName = name;

}

}

public class ExecuteAccount

{

static void Main()

{

string name = "SAVING\_ACCOUNT";

Account account = new Account();

account.SetAccount(name);

name = string.Empty;

name = account.GetAccount();

Console.WriteLine("Account Type: {0}", name);

Console.ReadKey();

}

}

}

In the above example we use the get and set methods (GetAccount and SetAccount) to return account and set account name. We use the private variable accountName and as it is not accessible directly, to use this variable, we use the get and set methods.

1. By using properties (read only properties, write only properties)

using System;

namespace Encaptulation

{

// Encapsulation using properties

public class Account

{

private string accoutname = "SAVING\_ACCOUNT";

// property which has get and set

public string AccoutName

{

get

{

return accoutname;

}

set

{

accoutname = value;

}

}

private string address = "India";

// readonly property

public string Address

{

get

{

return address;

}

}

private string phone = "1234567890";

// writeonly property

public string Phone

{

set

{

phone = value;

}

}

}

public class ExecuteAccount

{

static void Main()

{

// Encapsulation using properties

string name = string.Empty;

Account account = new Account();

// call get part

name = account.AccoutName;

// change the value

name = "CURRENT\_ACCOUNT";

// call set part

account.AccoutName = name;

string address = string.Empty;

// call readonly property

address = account.Address;

// now address has value "India"

string phone = string.Empty;

// call writeonly property

account.Phone = phone;

// now account.Phone has value "1234567890"

}

}

}

We can use a property (which has a get and set part), or we can use a read only property (which has only a get part) or we can also use a write only property (which has only a set part). But in all cases we can achieve encapsulation.

**Abstraction**

Abstraction is a process of hiding the implementation details and displaying the essential features.

Abstraction lets you focus on what the object does instead of how it does it.

Abstraction provides you a generalized view of your classes or objects by providing relevant information.  
   
Abstraction is the process of hiding the working style of an object, and showing the information of an object in an understandable manner.

*Encapsulation:* -- **Information hiding**.

*Abstraction:* -- **Implementation hiding**.

**Polymorphism**

Polymorphism means many forms (ability to take more than one form). In Polymorphism poly means “multiple” and morph means “forms” so polymorphism means many forms.

In polymorphism, we will declare methods with same name and different parameters in same class or methods with same name and same parameters in different classes. Polymorphism has ability to provide different implementation of methods that are implemented with same name.

In Polymorphism, we have two different types those are

- Compile Time Polymorphism (Called as Early Binding or Overloading or static binding)

- Run Time Polymorphism (Called as Late Binding or Overriding or dynamic binding)

**Compile Time Polymorphism**

Compile time polymorphism means we will declare methods with same name but different signatures because of this we will perform different tasks with same method name. This compile time polymorphism also called as early binding or method overloading.

Method Overloading or compile time polymorphism means same method names with different signatures (different parameters)

Example:

using System;

namespace PolymorphismApplication

{

class Printdata

{

void print(int i)

{

Console.WriteLine("Printing int: {0}", i);

}

void print(double f)

{

Console.WriteLine("Printing float: {0}", f);

}

void print(string s)

{

Console.WriteLine("Printing string: {0}", s);

}

static void Main(string[] args)

{

Printdata p = new Printdata();

// Call print to print integer

p.print(5);

// Call print to print float

p.print(500.263);

// Call print to print string

p.print("Hello C++");

Console.ReadKey();

}

}

}

Output:

Printing int: 5

Printing float: 500.263

Printing string: Hello C++

In above class we have three methods with same name but having different input parameters this is called method overloading or compile time polymorphism or early binding.

**Run time polymorphism**

Run time polymorphism or method overriding means same method names with same signatures. Run time polymorphism also called as late binding or method overriding or dynamic polymorphism.

In this run time polymorphism or method overriding we can override a method in base class by creating similar function in derived class this can be achieved by using inheritance principle and using “virtual & override” keywords.

In base class if we declare methods with virtual keyword, then only we can override those methods in derived class using override keyword

Example:

using System;

namespace PolymorphismApplication

{

//Base Class

public class Bclass

{

public virtual void Sample()

{

Console.WriteLine("Base Class");

}

}

// Derived Class

public class DClass : Bclass

{

public override void Sample()

{

Console.WriteLine("Derived Class");

}

}

// Using base and derived class

class Program

{

static void Main(string[] args)

{

// calling the overriden method

DClass objDc = new DClass();

objDc.Sample();

// calling the base class method

Bclass objBc = new DClass();

objBc.Sample();

Console.ReadKey();

}

}

}

Output:

Derived Class

Derived Class

**Inheritance**

Inheritance is a process of deriving the new class from already existing class.

C# is a complete object oriented programming language. Inheritance is one of the primary concepts of object-oriented programming. It allows you to reuse existing code. Through effective use of inheritance, you can save lot of time in your programming and reduce errors, which in turn will increase the quality of work and productivity.

Consider a base class Shape and its derived class Rectangle:

Example:

using System;

namespace InheritanceApplication

{

class Shape

{

protected int width;

protected int height;

public void setWidth(int w)

{

width = w;

}

public void setHeight(int h)

{

height = h;

}

}

// Derived class

class Rectangle : Shape

{

public int getArea()

{

return (width \* height);

}

}

class RectangleTester

{

static void Main(string[] args)

{

Rectangle Rect = new Rectangle();

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

Console.WriteLine("Total area: {0}", Rect.getArea());

Console.ReadKey();

}

}

}

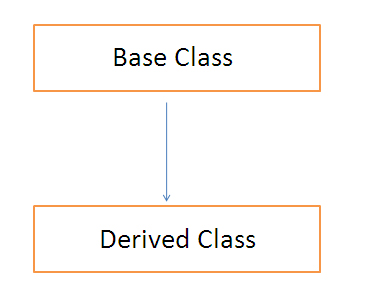
Output:

Total area: 35

**Types of inheritance in C#**

**Single Inheritance**

When a single base is been, implemented to single derived class is called as Single Inheritance. Means we have only one parent class and one child class.



using System;

namespace singleinheritance

{

class Company

{

public void CompanyName()

{

Console.WriteLine("Name of the Company");

}

public void CompanyAddress()

{

Console.WriteLine("Address of the Company");

}

}

class Employee : Company

{

public void NameofEmployee()

{

Console.WriteLine("Name of the Employee");

}

public void Salary()

{

Console.WriteLine("Salary of the Employee");

}

}

class output

{

static void Main(string[] args)

{

Employee emp = new Employee();

emp.CompanyName();

emp.CompanyAddress();

emp.NameofEmployee();

emp.Salary();

Console.ReadKey();

}

}

}

Output:

Name of the Company

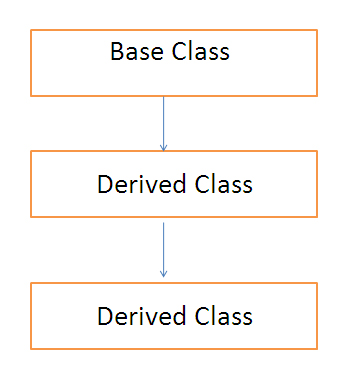
Address of the Company

Name of the Employee

Salary of the Employee

**Multilevel Inheritance**

When a derived class is created from another derived class or let me put it in a way that a class is created by using another derived class and this type of implementation is called as multilevel Inheritance.



using System;

namespace multilevelinheritance

{

class HeadOffice

{

public void HeadOfficeAddress()

{

Console.WriteLine("Head Office Address");

}

}

class BranchOffice : HeadOffice

{

public void BranchOfficeAddress()

{

Console.WriteLine("Branch Office Address");

}

}

class Employee : BranchOffice

{

public void NameofEmployee()

{

Console.WriteLine("Name of the Employee");

}

public void Salary()

{

Console.WriteLine("Salary of the Employee");

}

}

class output

{

static void Main(string[] args)

{

Employee emp = new Employee();

emp.HeadOfficeAddress();

emp.BranchOfficeAddress();

emp.NameofEmployee();

emp.Salary();

Console.ReadKey();

}

}

}

Output:

Head Office Address

Branch Office Address

Name of the Employee

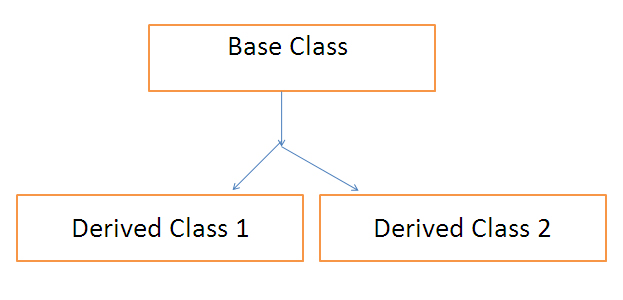
Salary of the Employee

**Multiple Inheritance**

Due to the complexity of a code multiple inheritance is not been supported in C# or in DOT.NET but DOT.NET or C# supports multiple interfaces.

**Hierarchical Inheritance**

When more than one derived classes are implemented from a same parent class or base class then that type of implementation is known as hierarchical inheritance.



using System;

namespace multilevelinheritance

{

class HeadOffice

{

public void HeadOfficeAddress()

{

Console.WriteLine("Head Office Address");

}

}

class BranchOffice1 : HeadOffice

{

public void BranchOfficeAddress()

{

Console.WriteLine("Branch Office Address");

}

}

class BranchOffice2 : HeadOffice

{

public void BranchOfficeAddress()

{

Console.WriteLine("Branch Office Address");

}

}

class output

{

static void Main(string[] args)

{

BranchOffice1 emp1 = new BranchOffice1();

emp1.HeadOfficeAddress();

emp1.BranchOfficeAddress();

BranchOffice2 emp2 = new BranchOffice2();

emp2.HeadOfficeAddress();

emp2.BranchOfficeAddress();

Console.ReadKey();

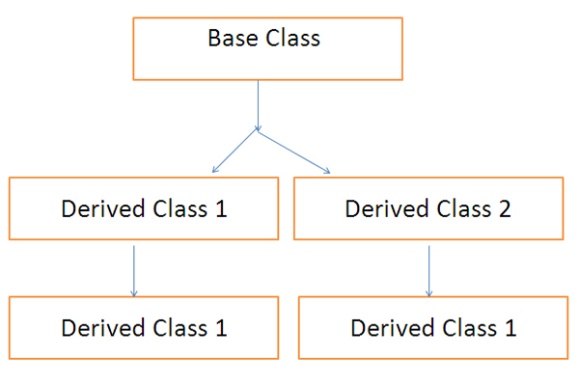
}

}

}

**Hybrid Inheritance**

This is a special type of inheritance and can be achieved from any combination of single, hierarchical and multi-level inheritance known as hybrid inheritance.



using System;

namespace multilevelinheritance

{

//This part of code is related to hierarchical inheritance

class HeadOffice

{

public void HeadOfficeAddress()

{

Console.WriteLine("Head Office Address");

}

}

class BranchOffice1 : HeadOffice

{

public void BranchOfficeAddress()

{

Console.WriteLine("Branch Office Address");

}

}

class BranchOffice2 : HeadOffice

{

public void BranchOfficeAddress()

{

Console.WriteLine("Branch Office Address");

}

}

////This part of code is related to combination of hierarchical inheritance and multi level inheritance

class Employee : BranchOffice2

{

public void NameofEmployee()

{

Console.WriteLine("Name of the Employee");

}

public void Salary()

{

Console.WriteLine("Salary of the Employee");

}

}

class output

{

static void Main(string[] args)

{

Employee emp1 = new Employee();

emp1.HeadOfficeAddress();

emp1.BranchOfficeAddress();

emp1.NameofEmployee();

emp1.Salary();

BranchOffice1 emp2 = new BranchOffice1();

emp2.HeadOfficeAddress();

emp2.BranchOfficeAddress();

Console.ReadKey();

}

}

}