C# .Net

(OOPS Material)

PRESENTED BY ARUNENDRA TIWARI

C# .Net - Object Oriented Programming

Index:

1. Introduction
2. Programming elements (variables & methods)
3. Classes and Objects
4. Static methods
5. Static variables
6. Instance methods
7. Object creation
8. Constructor
9. Instance variables
10. This and parameterized constructor
11. Access specifiers
12. Encapsulation
13. Constructor chaining using this()
14. Inheritance
15. Virtual and protected keywords
16. Method overriding
17. Base keyword
18. Base()
19. Polymorphism
20. sealed modifier or sealed classes
21. abstract classes
22. interfaces
23. Multiple inheritance through interface

C# – Object Oriented Programming

Application:

- Programming Languages and Technologies are used to develop applications.
- Application is a collection of Programs.
- We need to design and understand a single program before developing an application.

Program Elements: Program is a set of instructions. Every Program consists,

1. Identity
2. Variables
3. Methods

1. Identity:

- Identity of a program is unique.
- Programs, Classes, Variables and Methods having identities
- Identities are used to access these members.

2. Variable:

- Variable is an identity given to memory location.

  or

- Named Memory Location
- Variables are used to store information of program(class/object)

Syntax	Examples

datatype identity = value;	String name = "amar";
	int age = 23;
	double salary = 35000.00;
	bool married = false;

3. Method:

- Method is a block of instructions with an identity
- Method performs operations on data(variables)
- Method takes input data, perform operations on data and returns results.

Syntax	Example

returntype identity(arguments)
{
body;
}
	int add(int a, int b)
	{
	int c = a+b;
	return c;
	}

Introduction to Object oriented programming:

C# .Net - Object Oriented Programming

- C# is Object Oriented Programming language.
- OOPs is the concept of defining objects and establish communication between them.
- The Main principles of Object-Oriented Programming are,
    1. Encapsulation
    2. Inheritance
    3. Abstraction
    4. Polymorphism

Note: We implement Object-Oriented functionality using Classes and Objects

Class: Class contains variables and methods**.** C# application is a collection of classes

Syntax	Example

class ClassName
{
Variables ;
&
Methods ;
}
	class Account{
	long num;
	double balance;
	void withdraw(){
	logic;
	}
	void deposit(){
	logic;
	}
	}

Object: Object is an instance of class. Instancevariables of class get memory inside the Object.

Syntax: ClassName reference = new ClassName();
Example: Account acc = new Account();

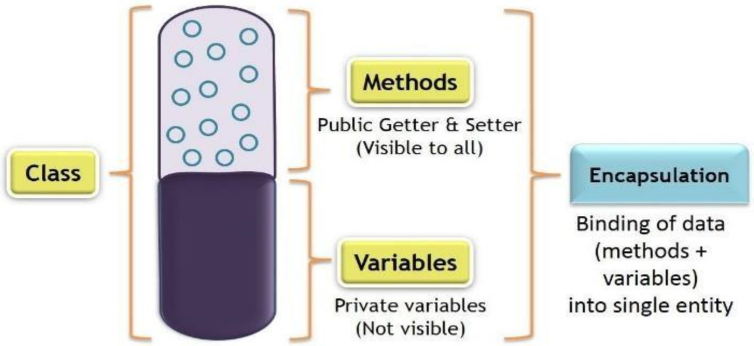
Note: Class is a Model from which we can define multiple objects of same type

class
Plan
Object
Instance
Object
Instance
Object
Instance

Encapsulation:

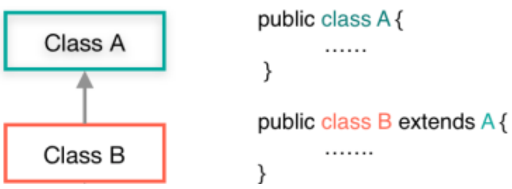
- The concept of protecting the data with in the class itself.
- **Implementation rules:** (POJO rules)
    - Class is Public (to make visible to other classes).
    - Variables are Private (other objects cannot access the data directly).
    - Methods are public (to send and receive the data).

C# .Net - Object Oriented Programming



Inheritance: Defining a new class by re-using the members of other class. We can implement inheritance using “extends” keyword.

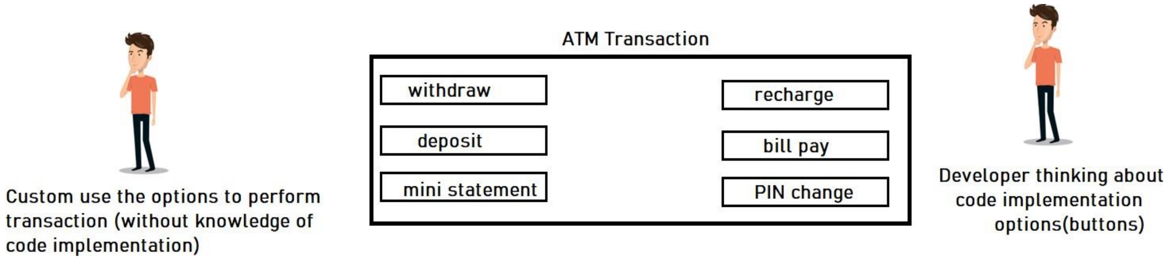
- **Terminology:**
  - **Parent/Super class:** The class from which members are re-used.
  - **Child/Sub class:** The class which is using the members



```
public class A {
        ......
 }
public class B extends A {
        .......
}
```

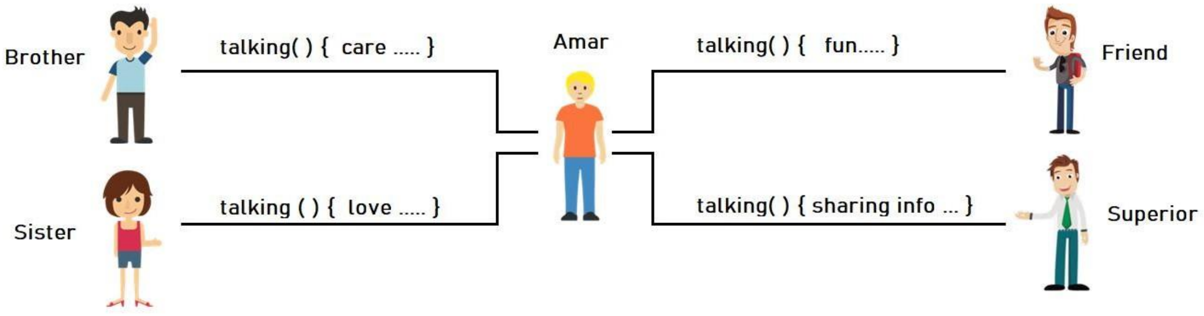
Abstraction:

- Abstraction is a concept of hiding implementations and shows functionality.
- Abstraction describes "What an object can do instead how it does it?".



Polymorphism:

- Polymorphism is the concept where object behaves differently in different situations.


Brother
talking( ) { care ..... }
Amar
talking( ) { fun..... }
Friend
Sister
talking ( ) { love ..... }
talking( ) { sharing info ... }
Superior

Variables in C#

Variables:

- Variable stores information of class(object).
- Variables classified into 4 types to store different types of data.

1. **Static Variables:** Store common information of all Objects. Access static variables using class-name.
2. **Instance Variables:** Store specific information of Object. Access instance variables using object-reference.
3. **Method Parameters:** Takes input in a Method. Access Method parameters directly and only inside the method.
4. **Local Variables:** Store processed information inside the Method. Access local variables directly and only inside the method.

```
class Employee
{
    static String company = "Anasol";       -> static variables : store common
    static String address = "Hyderabad";        info of all Employees
    int empId;
    String empName;                 -> instance variables : store specific
    double empSalary;                    information of Employee
    void totalSalary(double basic) -> Method parameter : takes method input
    {
        double hra = 0.2 * basic;
        double ta = 0.15 * basic ;
        double da = 0.25 * basic ;             -> Local variables : store processed
        double total = basic + hra + ta + da ; information inside the method
        System.out.println("Total Salary : " + total);
    }
}
```

Accessing different types of variables in C#:

Static Variables: Access using Class-Name.
Instance Variables: Access using Object-Name.
Local Variables: Direct access & only with in the method.
Method Parameters: Direct access & only with in the method.

Methods

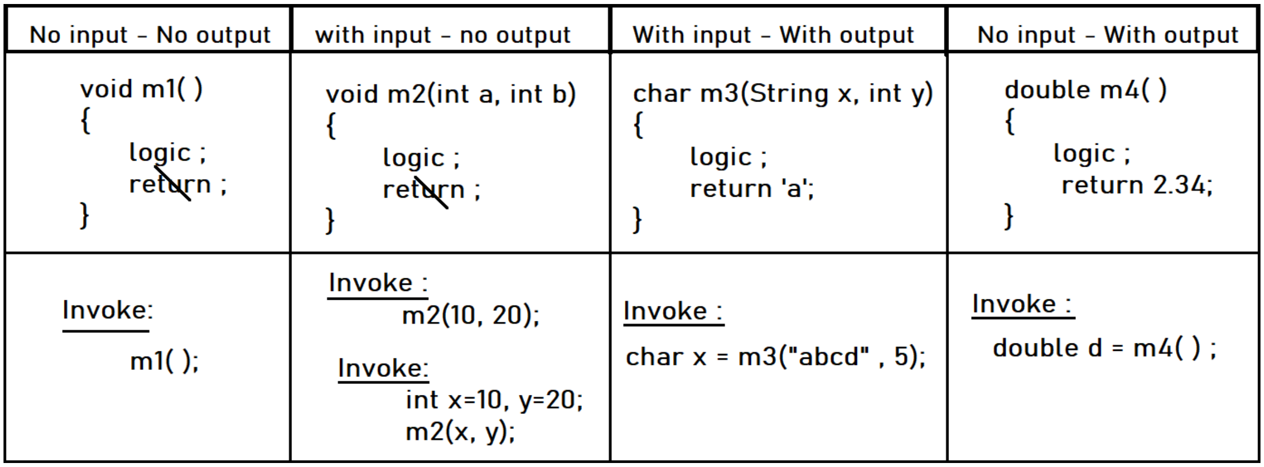
Method:

- A block of instructions having identity.
- Methods takes input(parameters), process input and return output.
- Methods are used to perform operations on data

Syntax	Example

returntype identity(arguments){
        statements;
}
	int add(int a, int b){
	        int c=a+b;
	        return c;
	}

Classification of Methods: Based on taking input and returning output, methods are classified into 4 types.



No input - No output	with input - no output	With input - With output	No input - With output

void m1( )
{
    logic ;
    ~~return~~ ;
}
	void m2(int a, int b)
	{
	    logic ;
	    ~~return~~ ;
	}
		char m3(String x, int y)
		{
		    logic ;
		    return 'a';
		}
			double m4( )
			{
			    logic ;
			    return 2.34;
			}

Invoke:
    m1( );
	Invoke :
	    m2(10, 20);
	Invoke:
	    int x=10, y=20;
	    m2(x, y);
		Invoke :
		char x = m3("abcd" , 5);
			Invoke :
			  double d = m4( ) ;

Method Definition: Method definition consists logic to perform the task. It is a block.
Method Call: Method Call is used to invoke the method logic. It is a single statement.

Static Method: Defining a method using static keyword. We can access static methods using class-name.

```
static void display()
{
        logic;
}
```

Instance Method: Defining a method without static keyword. We can access instance methods using object-reference.

```
void display()
{
        logic;
}
```

No arguments and No return values method:

```
public class Program{
        public static void Main(){
                Program.fun();
        }
        static void fun(){
                Console.WriteLine("fun");
        }
}
```

With arguments and No return values:

```
public class Program{
        public static void Main(){
                Program.isEven(13);
        }
        static void isEven(int n){
                if(n%2==0)
                        Console.WriteLine("Even");
                else
                        Console.WriteLine("Odd");
        }
}
```

With arguments with return values:

```
public class Program{
        public static void Main(){
                int sum = Program.add(10,20);
                Console.WriteLine("Sum is = " + sum);
        }
        static int add(int a, int b){
                return a+b;
        }
}
```

No arguments and with return values:

```
public class Program{
        public static void Main(){
                double PI = Program.getPI();
                Console.WriteLine("PI value is = " + PI);
        }
        static double getPI()
        {
                double pi = 3.142;
                return pi;
        }
}
```

Static Variables

Static Variable:

- Defining a variable inside the class and outside to methods.
- Static variable must define with static keyword.
- Static Variable access using Class-Name.

```
class Bank
{
    static String bankName = "AXIS";
}
```

Note: We always process the data (perform operations on variables) using methods.

Getter and Setter Methods:

- **set()** method is used to set the value to variable.
- **get()** method is used to get the value of variable.
- **Static variables :** process using static set() and get() methods
- **Instance variables :** process using instance set() and get() methods

```
public class Program
{
        static int a;
        static void setA(int a){
                Program.a = a;
        }
        static int getA(){
                return Program.a;
        }
        public static void Main()
        {
                Program.setA(10);
                Console.WriteLine(Program.getA());
        }
}
```

Static variables automatically initialized with default values based on datatypes:

Datatype	Default Value
int	0
double	0
char	Blank
bool	false
String	Null (blank)

```
public class Program
{
        static int a;
        static double b;
        static char c;
        static bool d;
        static string e;
        static void values(){
                Console.WriteLine("Default values : ");
                Console.WriteLine("int : " + Program.a);
                Console.WriteLine("double : " + Program.b);
                Console.WriteLine("char : " + Program.c);
                Console.WriteLine("bool : " + Program.d);
                Console.WriteLine("String : " + Program.e);
        }
        public static void Main()
        {
                Program.values();
        }
}
```

We can also display using default(datatype):

```
public class Program
{
        public static void Main()
        {
                Console.WriteLine("Default values : ");
                Console.WriteLine("int : " + default(int));
                Console.WriteLine("double : " + default(double));
                Console.WriteLine("char : " + default(char));
                Console.WriteLine("bool : " + default(bool));
                Console.WriteLine("String : " + default(string));
        }
}
```

It is recommended to define get() and set() method to each variable in the class:

```
public class Program
{
        public static void Main()
        {
                First.setA(10);
                First.setB(20);
                First.setC(30);
                Console.WriteLine("A val : " + First.getA());
                Console.WriteLine("B val : " + First.getB());
                Console.WriteLine("C val : " + First.getC());
        }
}
class First{
        static int a, b, c;
        public static void setA(int a){
                First.a = a;
        }
        public static void setB(int b){
                First.b = b;
        }
        public static void setC(int c){
                First.c = c;
        }
        public static int getA(){
                return First.a;
        }
        public static int getB(){
                return First.b;
        }
        public static int getC(){
                return First.c;
        }
}
```

Instance Members in C#

Instance Members:

- Instance members also called non-static members.
- Instance members related to Object.
- We invoke instance members using Object-address

Object Creation of a Class in C#:
Syntax:

```
        ClassName ref = new ClassName();
```

Example:

```
        Employee emp = new Employee();
```

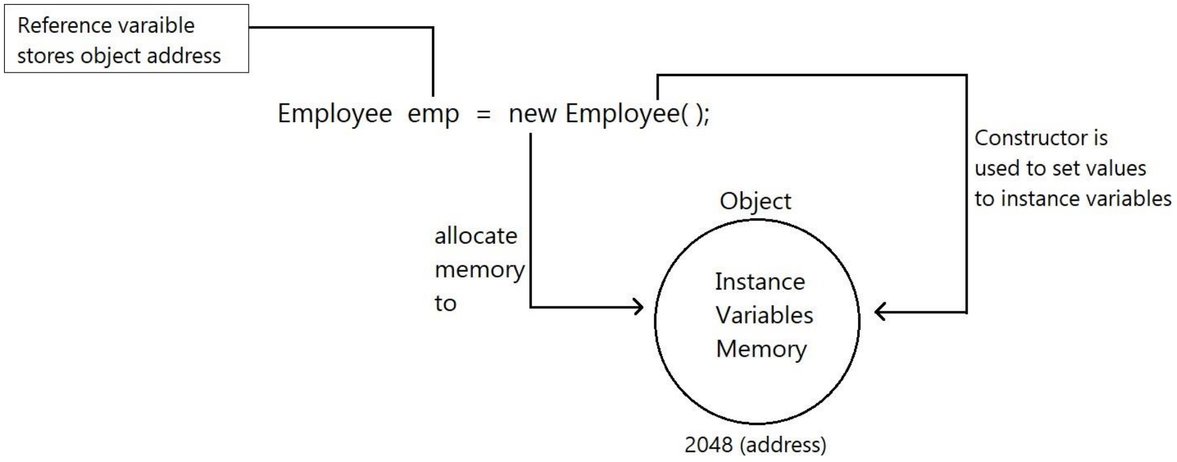
Constructor:

- Defining a method with Class-Name.
- Constructor Not allowed return-type.

```
class Employee{
        public Employee(){
                Console.WriteLine("Object created");
        }
}
```

We must invoke the constructor in Object creation process:

```
public class Program{
        public static void Main(){
                Employee emp = new Employee();
        }
}
class Employee{
        public Employee(){
                Console.WriteLine("Object created");
        }
}
```



Instance Method:

- Defining a method without static keyword.
- We must invoke the method using object-reference.

No arguments and No return values method:

```
public class Program
{
        public static void Main(){
                Program obj = new Program();
                obj.fun();
        }
```

```
    void fun(){
        Console.WriteLine("fun");
    }
}
```

With arguments and No return values method:

```
public class Program
{
    public static void Main(){
        Program obj = new Program();
        obj.isEven(12);
    }
    void isEven(int n){
        if(n%2==0)
            Console.WriteLine("even");
        else
            Console.WriteLine("odd");
    }
}
```

With arguments and with return values method:

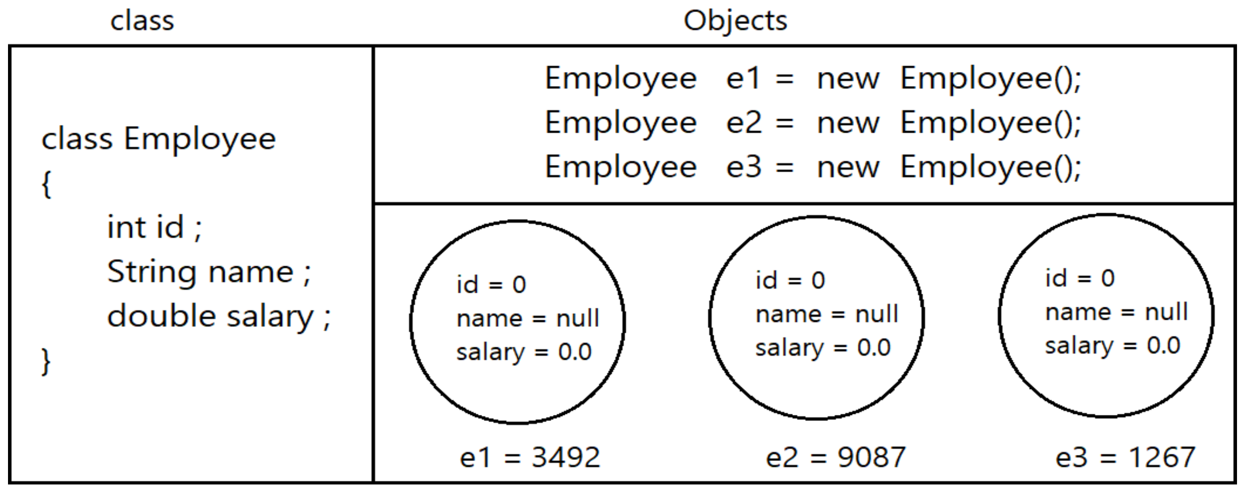
```
public class Program{
    public static void Main(){
        Program obj = new Program();
        int sum = obj.add(10, 20);
        Console.WriteLine("Sum is = " + sum);
    }
    int add(int a, int b){
        int c = a+b;
        return c;
    }
}
```

No arguments and with return values:

```
public class Program
{
    public static void Main(){
        Program obj = new Program();
        double PI = obj.getPI();
        Console.WriteLine("PI value is = " + PI);
    }
    double getPI(){
        double PI = 3.142;
        return PI;
    }
}
```

Instance Variables:

- Defining a variable inside the class and outside to methods.
- Instance variables get memory inside every object and initializes with default values.



```
class Employee
{
      int id ;
      String name ;
      double salary ;
}
```

```
Employee   e1 =  new  Employee();
Employee   e2 =  new  Employee();
Employee   e3 =  new  Employee();
```

Memory allocation to objects

this:

- It is a keyword and pre-defined instance variable in C#.
- It is also called **Default Object Reference Variable**.
- “this-variable” holds object address.
    - this = object_address;
- It is used to access object inside the instance methods and constructor.

Parameterized constructor:

- Constructor with parameters is called Parametrized constructor.
- We invoke the constructor in every object creation process.
- Parameterized constructor is used to set initial values to instance variables in Object creation process.

Note: We invoke the constructor with parameter values in object creation as follows

```
public class Program
{
        int a;
        Program(int a){
                this.a = a;
        }
        public static void Main(){
                Program p = new Program(10); //pass parameter while invoking constructor
        }
}
```

Program to create two Employee objects with initial values:

```
public class Employee
{
    int id;
    String name;
    Employee(int id, String name){
        this.id = id;
        this.name = name;
    }
    void details(){
        Console.WriteLine(this.id + ", " + this.name);
    }
    public static void Main() {
        Employee e1 = new Employee(101, "Amar");
        Employee e2 = new Employee(102, "Annie");
        e1.details();
        e2.details();
    }
}
```

Accessing variables using set() and get() methods:

```
using System;
class Employee{
    private int id;
    private string name;
    public void setId(int id){
        this.id=id;
    }
    public void setName(string name){
        this.name = name;
    }
    public int getId(){
        return this.id;
    }
    public string getName(){
        return this.name;
    }
}

public class Access{
    public static void Main(){
        Employee e = new Employee();
        e.setId(101);
```

```
                    e.setName("Amar");
                    Console.WriteLine(e.getId());
                    Console.WriteLine(e.getName());
          }
}
```

Access Modifiers

Access Modifiers:

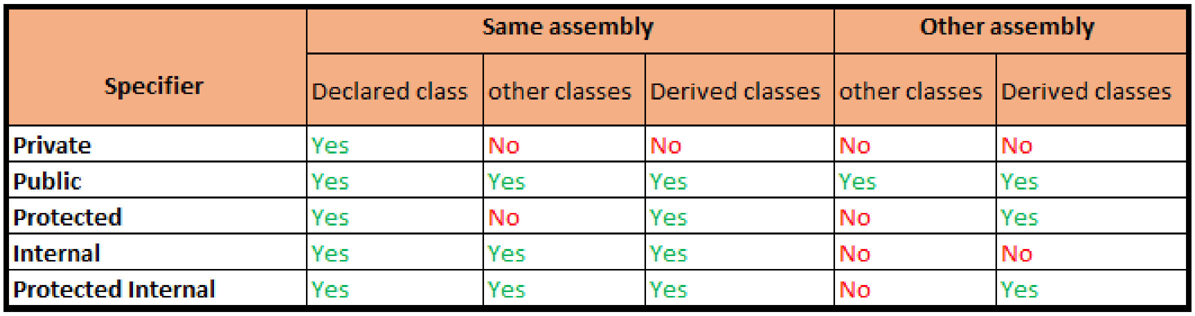
- Access modifiers are used to set permissions to access the Class and its members (variables, methods & constructors).
- C# supports 4 access modifiers
    - private
    - protected
    - public
    - internal
    - protected internal

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.

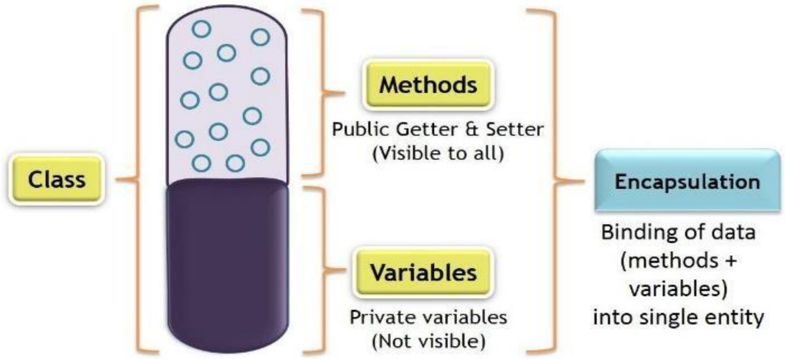
Internal: Internal provides accessibility from within the project. Another similar internal accessibility is protected internal. This allows the same as the internal and the only difference is that a child class can inherit this class and reach its members even from another project.



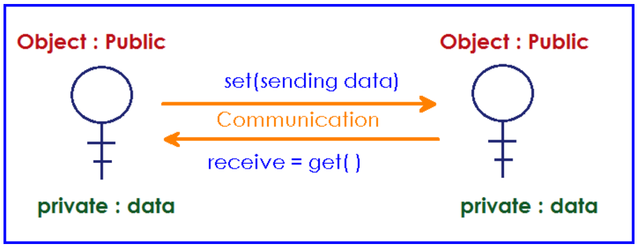
Specifier	Same assembly			Other assembly	
	Declared class	other classes	Derived classes	other classes	Derived classes
Private	Yes	No	No	No	No
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	No	Yes	No	Yes
Internal	Yes	Yes	Yes	No	No
Protected Internal	Yes	Yes	Yes	No	Yes

Encapsulation:

- The concept of protecting the data with in the class itself.
- We implement Encapsulation through POJO rules
- POCO – Plain Old CLR Object



- **Implementation rules:** (POJO rules)
  - Class is Public (to make the object visible in communication).
  - Variables are Private (other objects cannot access the data directly).
  - Methods are public (to send and receive the data).



Defining get() and set() methods to Balance variable in Account class:

Set() : takes input value and set to instance variable.

Get() : returns the value of instance variable.

```
public class Bank{
        private double balance;
        public void setBalance(double balance){
                this.balance = balance;
        }
        public double getBalance(){
                return this.balance;
        }
}
```

Employee.cs: (POCO class)

```
public class Employee {
    private int num;
    private string name;
    private double salary;
    public void setNum(int num){
        this.num = num;
    }
public int getNum(){
        return this.num;
    }
    public void setName(string name){
        this.name = name;
    }
    public string getName(){
        return this.name;
    }
    public void setSalary(double salary){
        this.salary = salary;
    }
    public double getSalary(){
        return this.salary;
    }
}
```

AccessEmployee.cs:

```
public class AccessEmployee{
    public static void Main() {
        Employee e = new Employee();
        e.setNum(101);
        e.setName("Amar");
        e.setSalary(35000);
        Console.WriteLine("Emp Num : "+e.getNum());
        Console.WriteLine("Emp Name : "+    e.getName());
        Console.WriteLine("Emp Salary : "+e.getSalary());
    }
}
```

Constructor Chaining:

- Invoking constructor from another constructor is called Chaining.
- this() method is used to invoke constructor.

```
public class Test{
        public Test() : this(10)
        {
                Console.WriteLine("Zero args");
        }
        public Test(int x) : this(10, 20)
        {
                Console.WriteLine("Args");
        }
        public Test(int x, int y)
        {
                Console.WriteLine("Two args");
        }
}
public class Access
        public static void Main() {
                Test obj = new Test();
        }
}
```

Inheritance

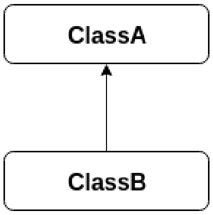
Inheritance:

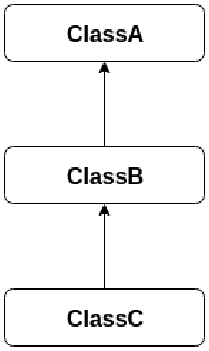
- Defining a new class by re-using the members of other class.
- We can implement inheritance using “extends” keyword.
- **Terminology:**
    - **Parent/Super class:** The class from which members are re-used.
    - **Child/Sub class:** The class which is using the members

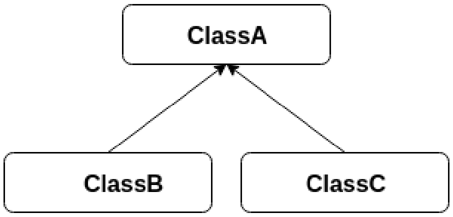
Types of Inheritance:

1. Single Inheritance
2. Multi-Level Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance

Note: We can achieve above relations through classes

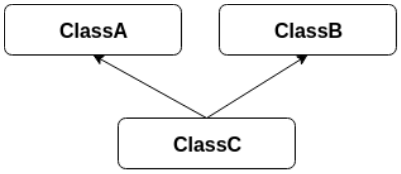
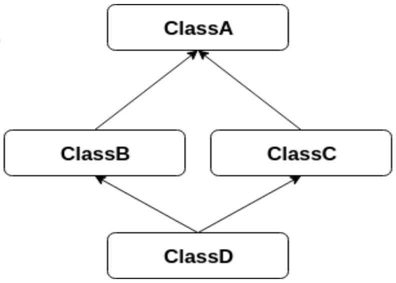

Single Inheritance
	class A{
	.....
	}
	class B : A{
	.....
	}


Multilevel Inheritance
	class A{
	....
	}
	class B : A{
	....
	}
	class C : B{
	....
	}

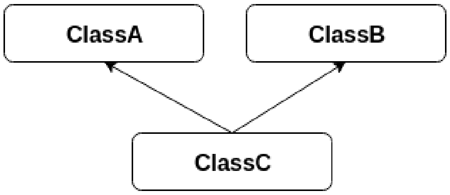

Hierarchical Inheritance
	class A{
	....
	}
	class B : A{
	....
	}
	class C : A{
	....
	}

The two other inheritance types are:

1. Multiple Inheritance
2. Hybrid Inheritance

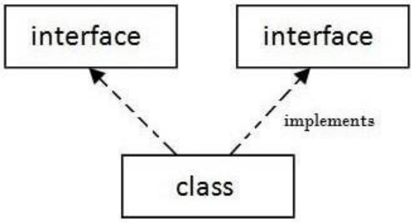
 **Multiple Inheritance**	

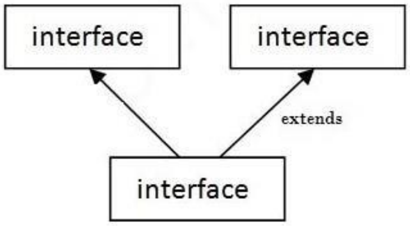
We cannot achieve multiple inheritance through Classes:


Multiple Inheritance
	class A{
	....
	}
	class B{
	....
	}
	class C : A, B{
	....
	}

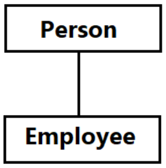
We can achieve multiple inheritance in C# through interfaces:

- A class can implements more than one interface
- An interface extends more than one interface is called Multiple Inheritance

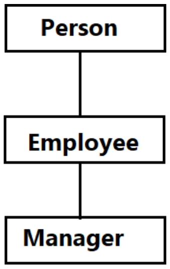
	Interface A{
	}
	Interface B{
	}
	Class C : A, B{
	}


Multiple Inheritance
	Interface A{
	}
	Interface B{
	}
	Interface C : A, B{
	}

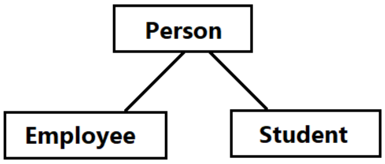
Note: We always instantiate (create object) of Child in Inheritance



Single



Multi-Level



Hierarchical

Single Inheritance: By instantiating child class, we can access both Parent & Child functionality.

```
using System;
class Employee {
        public void doWork(){
                Console.WriteLine("Employee do work");
        }
}
class Manager : Employee {
        public void monitorWork(){
                Console.WriteLine("Manage do work as well as monitor others work");
        }
}
public class Company {
        public static void Main() {
                Manager m = new Manager();
                m.doWork();
                m.monitorWork();
        }
}
```

Accessing Protected functionality of Parent from Child:

```
using System;
class Employee {
        protected void doWork(){
                Console.WriteLine("Employee do work");
        }
}
class Manager : Employee {
        public void monitorWork() {
                doWork();
                Console.WriteLine("Manage do work as well as monitor others work");
        }
}
public class Company {
        public static void Main() {
                Manager m = new Manager();
                m.monitorWork();
        }
}
```

In Object creation, Parent object creates first to inherit properties into Child.
We can check this creation process by defining constructors in Parent and Child.

```
using System;
class Parent{
        public Parent(){
                Console.WriteLine("Parent instantiated");
        }
}
class Child : Parent{
        public Child(){
                Console.WriteLine("Child instantiated");
        }
}
public class Inherit {
        public static void Main(){
                new Child();
        }
}
```

this	this()
A reference variable used to invoke instance members.	It is used to invoke the constructor of same class.
It must be used inside instance method or instance block or constructor.	It must be used inside the constructor.

base	base()
A reference variable used to invoke instance members of Parent class from Child class.	It is used to invoke the constructor of same class.
It must be used inside instance method or instance block or constructor of Child class.	It must be used inside Child class constructor.

Method overriding:

- If derived class defines same method as defined in its base class, it is known as method overriding in C#.
- It is used to achieve runtime polymorphism.
- It enables you to provide specific implementation of the method which is already provided by its base class.
- To perform method overriding in C#, you need to use virtual keyword with base class method and override keyword with derived class method

```
using System;
class Parent{
    public virtual void fun(){
        Console.WriteLine("Parent functionality");
    }
}
class Child : Parent{
    public override void fun(){
        Console.WriteLine("Child functionality");
    }
}
public class Inherit {
    public static void Main(){
        Parent p = new Child();
        p.fun();
    }
}
```

Accessing Parent class overridden method using “base”:

```
using System;
class Parent{
    public virtual void fun(){
        Console.WriteLine("Parent functionality");
    }
}
class Child : Parent{
    public override void fun(){
        Console.WriteLine("Child functionality");
    }
    public void access(){
        this.fun();
        base.fun();
    }
}
public class Inherit {
    public static void Main(){
        Child c = new Child();
        c.access();
    }
}
```

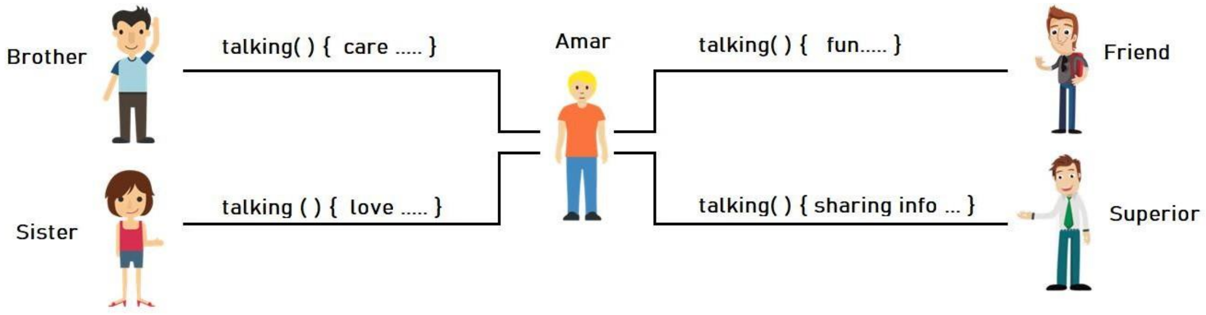
base():

- In inheritance, we always create Object to Child class.
- In Child object creation process, we initialize instance variables of by invoking Parent constructor from Child constructor using base().

```
using System;
class Parent
{
        public int a, b;
        public Parent(int a, int b)
        {
                this.a = a;
                this.b = b;
        }
}
class Child : Parent
{
        public int c, d;
        public Child(int a, int b, int c, int d) : base(a, b)
        {
                this.c = c;
                this.d = d;
        }
        public void details()
        {
                Console.WriteLine("Parent a : " + base.a);
                Console.WriteLine("Parent b : " + base.b);
                Console.WriteLine("Child c : " + this.c);
                Console.WriteLine("Child d : " + this.d);
        }
}
public class Inherit
{
        public static void Main()
        {
                Child c = new Child(10, 20, 30, 40);
                c.details();
        }
}
```

Polymorphism:

- Polymorphism is the concept where object behaves differently in different situations.


Brother
talking( ) { care ..... }
Amar
talking( ) { fun..... }
Friend
Sister
talking ( ) { love ..... }
talking( ) { sharing info ... }
Superior

Polymorphism is of two types:

1. Compile time polymorphism
2. Runtime polymorphism

Compile time polymorphism:

- It is method overloading technique.
- Defining multiple methods with same name and different signature(parameters).
- Parameters can be either different length or different type.
- Overloading belongs to single class(object).

```
using System;
class Calculator {
        public void add(int x, int y) {
                int sum = x+y;
                Console.WriteLine("Sum of 2 numbers is : " + sum);
        }
        public void add(int x, int y, int z)
        {
                int sum = x+y+z;
                Console.WriteLine("Sum of 3 numbers is : " + sum);
        }
}
public class Overload{
        public static void Main()
        {
                Calculator calc = new Calculator();
                calc.add(10, 20);
                calc.add(10, 20, 30);
        }
}
```

WriteLine() method is pre-defined and overloaded. Hence it can print any type of data:

```
using System;
public class Overload
{
	public static void Main()
	{
		Console.WriteLine(10);
		Console.WriteLine(23.45);
		Console.WriteLine("C#");
	}
}
```

Runtime polymorphism:

- Runtime Polymorphism is a **Method overriding technique**.
- Defining a method in the Child class with the **same name and same signature** of its Parent class.
- We can implement Method overriding only in Parent-Child (Is-A) relation.

Child object shows the functionality(behavior) of Parent and Child is called Polymorphism

```
using System;
class Parent{
	public virtual void behave(){
		Console.WriteLine("Parent behavior");
	}
}
class Child : Parent{
	public override void behave(){
		Console.WriteLine("Child behavior");
	}
	public void behavior(){
		base.behave();
		this.behave();
	}
}
public class Overrriding{
	public static void Main(){
		Child child = new Child();
		child.behavior();
	}
}
```

Object Up-casting:

- We can store the address of Child class into Parent type reference variable.

```
Parent addr = new Child();  // upcast

              or

Child  obj  = new Child();
Parent addr = obj ; // upcast
```

Using parent address reference variable, we can access the functionality of Child class.

```
using System;
class Parent {
        public virtual void fun(){
                Console.WriteLine("Parent's functionality");
        }
}
class Child : Parent{
        public override void fun(){
                Console.WriteLine("Updated in Child");
        }
}
public class Upcast {
        public static void Main() {
                Parent addr = new Child();
                addr.fun();
        }
}
```

Why it is calling Child functionality in the above application?

- Hostel address = new Student();
  - address.post(); -> The Post reaches student
- Owner address = new Tenant();
  - address.post(); -> The Pose reaches tenant

Down casting: The concept of collecting Child object address back to Child type reference variable from Parent type.

```
Parent  p = new Parent( );
Child c = (Child)p ; ⟶ It is not down casting

Parent  p = new Child( );
Child c = (Child)p ; ⟶ It is down casting
```

Sealed Classes

sealed:

- sealed is a keyword/modifier.
- A member become constant if we define it is sealed hence cannot be modified.
- We can apply sealed modifier to Class or Method or Variable.

If Class is sealed, cannot be inherited:

```
sealed class A{
}
class B : A{
}
```

If Method is final, cannot be overridden:

```
using System;
class X  {
        protected virtual void F1(){
        }
        protected virtual void F2(){
        }
}
class Y : X  {
        sealed protected override void F1(){
        }
        protected override void F2() {
        }
}
class Z : Y {
        protected override void F() {  // Error:
        }
}
```

If the variable is final become constant and cannot be modified:

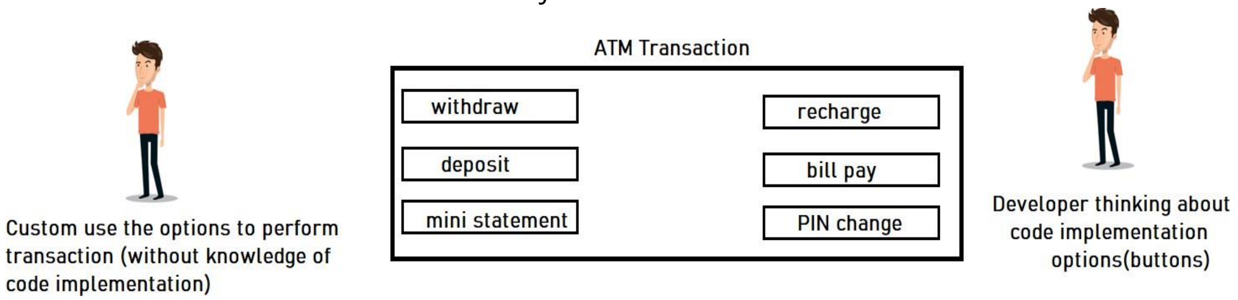
```
using System;
public class Test{
    static double PI = 3.14;
    public static void Main() {
        Console.WriteLine("PI val = " + PI);
        PI = 3.142;
        Console.WriteLine("PI val = " + PI);
    }
}
```

```
using System;
public class Test{
    sealed static double PI = 3.14;
    public static void Main(){
        Console.WriteLine("PI val = " + PI);
        PI = 3.142; // Error:
        Console.WriteLine("PI val = " + PI);
    }
}
```

Abstraction

Abstraction:

- Abstraction is a concept of hiding implementations and shows required functionality.
- Abstraction describes "What an object can do instead how it does it?".


ATM Transaction
withdraw
deposit
mini statement
recharge
bill pay
PIN change
Custom use the options to perform transaction (without knowledge of code implementation)
Developer thinking about code implementation options(buttons)

Abstract Class: Define a class with abstract keyword. Abstract class consists concrete methods and abstract methods.

Concrete Method: A Method with body
Abstract Method: A Method without body

Class	Abstract Class
Class allows only Concrete methods	Abstract Class contains Concrete and Abstract methods

class A{
public void m1(){
logic;
}
public void m2(){
logic;
}
}
	abstract class A
	{
	public void m1(){
	logic;
	}
	public abstract void m2();
	}

Note: We cannot instantiate (create object) to abstract class because it has undefined methods.

```
abstract class A{
        public abstract void m1();
        public void m2(){
        }
}
public class Access{
        public static void Main(){
                A obj = new A();  // Error :
        }
}
```

Extending Abstract class:

- Every abstract class need extension(child).
- Child override the abstract methods of Abstract class.
- Through Child object, we can access the functionality of Parent (Abstract).

```
using System;
public abstract class Parent{
    public abstract void m1();
    public void m2(){
        Console.WriteLine("Parent concrete method");
    }
}
public class Child : Parent{
    public override void m1(){
        Console.WriteLine("Parent abstract method");
    }
}
public class Access{
    public static void Main(){
        Child obj = new Child();
        obj.m1();
        obj.m2();
    }
}
```

Initializing abstract class instance variables:

- Abstract class can have instance variables.
- Using base(), we initialize Abstract class instance variables in Child object creation process.

```
using System;
abstract class Parent{
    public int a, b;
    public Parent(int a, int b){
         this.a = a;
        this.b = b;
    }
    public abstract void details();
}
```

```
class Child : Parent {
        public int c, d;
        public Child(int a, int b, int c, int d) : base(a, b) {
                this.c = c;
                this.d = d;
        }
        public override void details() {
                Console.WriteLine("Parent a : " + base.a);
                Console.WriteLine("Parent b : " + base.b);
                Console.WriteLine("Child c : " + this.c);
                Console.WriteLine("Child d : " + this.d);
        }
}
public class Abstraction {
        public static void Main() {
                Child c = new Child(10, 20, 30, 40);
                c.details();
        }
}
```

Interfaces

Interface:

- Interface allow to define only abstract methods.
- Interface methods are ‘public abstract’ by default.

```
interface Sample {
        void m1();
        void m2();
}
```

Implementing an interface:

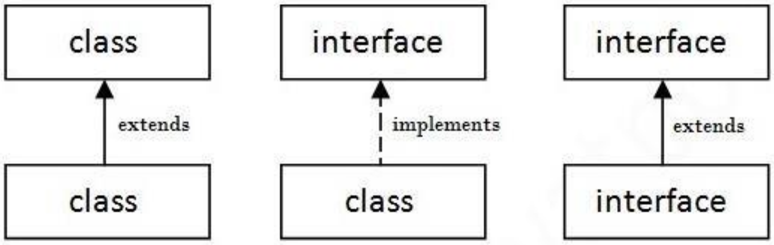
- Any class can implement the interface by overriding the methods of interface.
- We override the methods of interface using public modifier.

```
using System;
interface First {
        void m1();
        void m2();
}
```

```
class Second : First
{
        public void m1(){
                Console.WriteLine("m1....");
        }
        public void m2(){
                Console.WriteLine("m2....");
        }
}
public class Implement
{
        public static void Main(){
                First obj = new Second();
                obj.m1();
                obj.m2();
        }
}
```

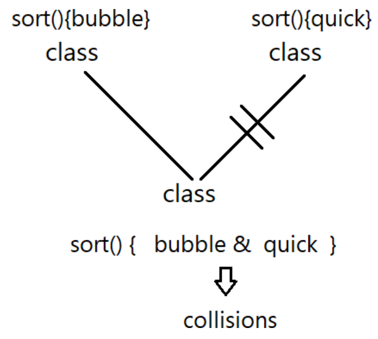
Upcasting: object reference of implemented class storing into Interface type variable

Relations:

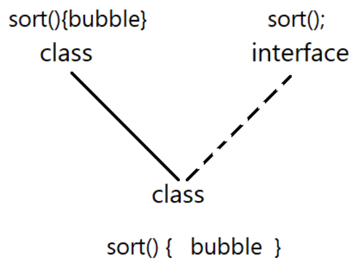


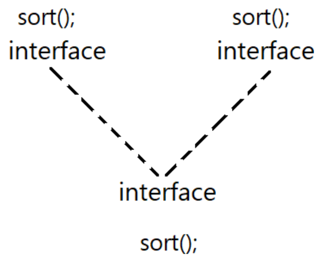
Multiple Inheritance in C#:

- A class can extends only class
- A class can extends class and implements any number of interfaces
- An interface can extend any number of interfaces called **'Multiple Inheritance'**



```
class A{
}
class B{
}
class C : A, B{
}
```

 sort(){bubble}
class
sort();
interface
class
sort() {   bubble }
	class A{
	}
	interface B{
	}
	class C : A, B{
	}

 sort();
interface
sort();
interface
interface
sort();
	interface A{
	}
	interface B{
	}
	class C : A, B{
	}

```
using System;
interface A{
        void m1();
}
interface B{
        void m2();
}
class C : A, B{
        public void m1(){
                Console.WriteLine("m1...");
        }
        public void m2(){
                Console.WriteLine("m2...");
        }
}
public class Multiple
{
        public static void Main()
        {
                C obj = new C();
                obj.m1();
                obj.m2();
        }
}
```