<http://www.c-sharpcorner.com/UploadFile/cda5ba/object-oriented-programming-with-real-world-scenario/>

Object Oriented Programming With Real-World Scenario **Download Files:**

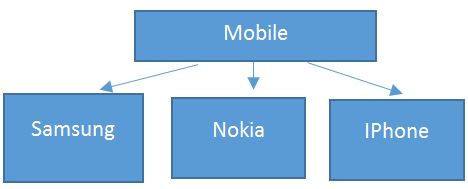
* [**MobileEx.zip**](javascript:__doPostBack('ctl00$RepeaterDownloadFiles$ctl00$LinkButtonDownloadFiles',''))

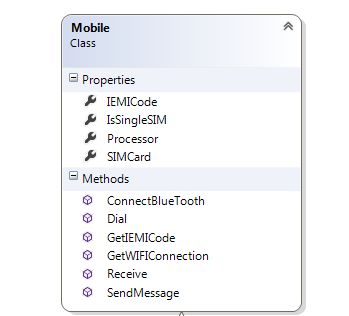
Normally every interviewer ask for a real world scenario explaining OOP and many of them fail to answer. This is the reason I am writing this article. This article is mainly intended for the audience who are knowing the Object Oriented Programming (OOP) concept theoretically but are unable to link with real world & programming world.

  
  
We write programs to solve our problem and get our work done.  
  
Object Oriented Programming is basically considered as design methodology for creating a non-rigid application. In OOPS every logic is written to get our work done, but this is done based on entity which we call it as Objects. OOP allow us to decompose our problem in to small unit of work which are accessed via Objects. We build function around this objects. There are mainly four pillars (features) of OOP.  
  
If all this features fulfill our programming, then we can say it as perfect Object Oriented Programming.

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Let’s consider an example for explaining each pillar, which at the end will make you understand & follow Object Oriented Programming. Before that, we need to know something  
  
When we take a mobile as an object, its basic functionality for which it was invented were Calling & Receiving a call & Messaging. But now a days thousands of new features & models were added & the count is still increasing.

  
  
  
  
In above diagram, each brand (Samsung, Nokia, IPhone) have their own list of features along with basic functionality of dialing, receiving a call & messaging.   
  
When we talk about OOP, as the word indicate it will talk about an object (a real world object)  
  
**Objects**  
Any real world entity which can have some characteristics or which can perform some work is called as Object. This object is also called as an instance i.e. - a copy of entity in programming language. If we consider the above example, a mobile manufacturing company, at a time manufactures lacs of pieces of each model which are actually an instance. This objects are differentiated from each other via some identity or its characteristics. This characteristics is given some unique name.  
  
Mobile mbl1 = new Mobile ();  
Mobile mbl2 = new Mobile ();  
  
**Class**  
A Class is a plan which describes the object. We call it as a blue print of how the object should be represented. Mainly a class would consist of a name, attributes & operations. Considering the above example, A Mobile can be a class which has some attributes like Profile Type, IMEI Number, Processor, and some more.) & operations like Dial, Receive & SendMessage.  
  
There are some OOPS principle that need to be looked in while creating any of the class. This principle is called as SOLID where each letter has some specification. I won’t be going into this points deeper. A single line of each explanation may clear you with some points.  
  
SRP (The Single Responsibility Principle)   
A class should have one, and only one responsibility  
  
OCP (The Open Closed Principle)   
You should be able to extend a classes behavior, without modifying it. (Inheritance)   
  
LSP (The Liskov Substitution Principle)   
Derived classes must be substitutable for their base classes. (Polymorphism)  
  
ISP (The Interface Segregation Principle)   
Make fine chopped interface instead of huge interface as client cannot be forced to implement an interface which they don’t use.   
  
DIP (The Dependency Inversion Principle)   
Depend on abstractions, not on concretions. (Abstraction)  
  
Now this class is represented as shown below



    public class Mobile

    {

        private string IEMICode { get; set; }

        public string SIMCard { get; set; }

        public string Processor { get; }

        public int InternalMemory { get; }

        public bool IsSingleSIM { get; set; }

        public void GetIEMICode()

        {

            Console.WriteLine("IEMI Code - IEDF34343435235");

        }

        public void Dial()

        {

            Console.WriteLine("Dial a number");

        }

        public void Receive()

        {

            Console.WriteLine("Receive a call");

        }

        public virtual void SendMessage()

        {

            Console.WriteLine("Message Sent");

        }

    }

**Abstraction**  
Abstraction says, only show relevant details and rest all hide it. This is most important pillar in OOPS as it is providing us the technique to hide irrelevant details from User. If we consider an example of any mobile like Nokia, Samsung, IPhone.   
  
**Some features of mobiles**

1. Dialing a number call some method internally which concatenate the numbers and displays it on screen but what is it doing we don’t know.
2. Clicking on green button actual send signals to calling person’s mobile but we are unaware of how it is doing.

This is called abstraction where creating method which is taking some parameter & returning some result after some logic execution without understating what is written within the method

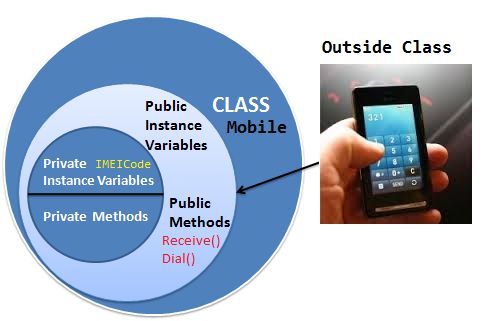
public void Dial()

{

      //Write the logic

      Console.WriteLine("Dial a number");  
}

**Encapsulation**

  
  
Encapsulation is defined as the process of enclosing one or more details from outside world through access right. It says how much access should be given to particular details. Both Abstraction & Encapsulation works hand in hand because Abstraction says what details to be made visible & Encapsulation provides the level of access right to that visible details. i.e. – It implements the desired level of abstraction.  
  
Talking about Bluetooth which we usually have it in our mobile. When we switch on the Bluetooth I am able to connect another mobile but not able to access the other mobile features like dialing a number, accessing inbox etc. This is because, Bluetooth feature is given some level of abstraction.   
  
Another point is when mobile A is connected with mobile B via Bluetooth whereas mobile B is already connected to mobile C then A is not allowed to connect C via B. This is because of accessibility restriction.   
This is handled by access specifier like public, private, protected, and internal

private string IMEICode = "76567556757656";

**Polymorphism**  
Polymorphism can be defined as the ability of doing the same operation but with different type of input.  
  
More precisely we say it as ‘many forms of single entity’. This play a vital role in the concept of OOPS.  
  
Let’s say Samsung mobile have the 5MP camera available i.e. – it is having a functionality of CameraClick(). Now same mobile is having Panorama mode available in camera, so functionality would be same but with mode. This type is said to be Static polymorphism or Compile time polymorphism. See the example below:

    public class Samsumg : Mobile

    {

        public void GetWIFIConnection()

        {

            Console.WriteLine("WIFI connected");

        }

        //This is one mwthod which shows camera functionality

        public void CameraClick()

        {

            Console.WriteLine("Camera clicked");

        }

        //This is one overloaded method which shows camera functionality as well but with its camera's different mode(panaroma)

        public void CameraClick(string CameraMode)

        {

            Console.WriteLine("Camera clicked in " + CameraMode + " Mode");

        }  
    }

Compile time polymorphism the compiler knows which overloaded method it is going to call.  
  
Compiler checks the type and number of parameters passed to the method and decides which method to call and it will give an error if there are no methods that matches the method signature of the method that is called at compile time.  
  
Another point where in SendMessage was intended to send message to single person at a time but suppose Nokia had given provision for sending message to a group at once. i.e. - Overriding the functionality to send message to a group. This type is called Dynamic polymorphism or Runtime polymorphism.  
  
For overriding you need to set the method, which can be overridden to virtual & its new implementation should be decorated with override keyword.

public class Nokia : Mobile

{

        public void GetBlueToothConnection()

        {

            Console.WriteLine("Bluetooth connected");

        }

        //New implementation for this method which was available in Mobile Class

        //This is runtime polymorphism

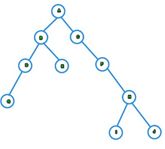
        public override void SendMessage()

        {

            Console.WriteLine("Message Sent to a group");

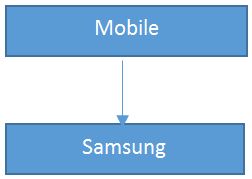
        }  
}

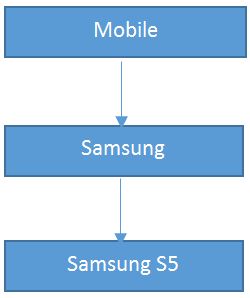
By runtime polymorphism, we can point to any derived class from the object of the base class at runtime that shows the ability of runtime binding.  
  
**Inheritance**

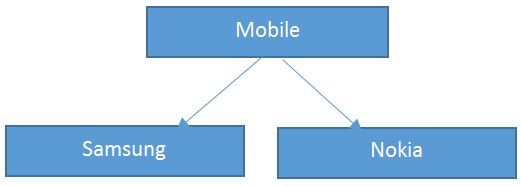
  
Ability to extend the functionality from base entity in new entity belonging to same group. This will help us to reuse the functionality which is defined before.   
  
Considering the example, the above figure 1.1 itself shows what is inheritance. Basic Mobile functionality is to Send Message, dial & receive call. So the brands of mobile is using this basic functionality by extending the mobile class functionality and adding their own new features to their respective brand.  
  
There are mainly 4 types of inheritance:

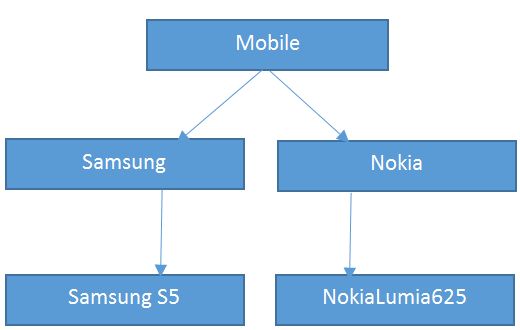
1. Single level inheritance
2. Multi-level inheritance
3. Hierarchical inheritance
4. Hybrid inheritance
5. Multiple inheritance

**Single level inheritance**  
In Single level inheritance, there is single base class & a single derived class i.e. - A base mobile features is extended by Samsung brand.

  
  
**Multilevel inheritance**In Multilevel inheritance, there is more than one single level of derivation. i.e. - After base features are extended by Samsung brand. Now Samsung brand has manufactured its new model with new added features or advanced OS like Android OS, v4.4.2 (kitkat). From generalization, getting into more specification.

  
 **Hierarchal inheritance**  
In this type of inheritance, multiple derived class would be extended from base class, it’s similar to single level inheritance but this time along with Samsung, Nokia is also taking part in inheritance.

  
  
**Hybrid inheritance**  
Single, Multilevel, & hierarchal inheritance all together construct a hybrid inheritance.



public class Mobile

{

       //Properties

       //Methods

}

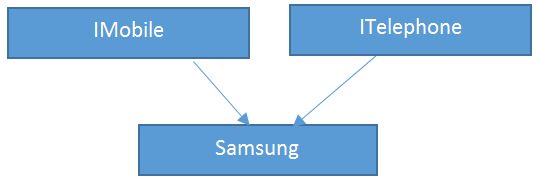
public class Samsumg : Mobile

{  
       //Properties  
       //Methods  
}

public class Nokia : Mobile

{  
    //Properties  
   //Methods  
}

**Interface**Multiple inheritance where derived class will extend from multiple base classes.  
  
Samsung will use the function of multiple Phone (Mobile & Telephone). This would create a confusion for complier to understand which function to be called when any event in mobile is triggered like Dial () where Dial is available in both the Phone i.e. - (Mobile & Telephone). To avoid this confusion C# came with the concept of interface which is different from multiple inheritance actually.  
  
If we take an interface it is similar to a class but without implementation & only declaration of properties, methods, delegates & events. Interface actually enforces the class to have a standard contract to provide all implementation to the interface members. Then what’s is the use of interface when they do not have any implementation? Answer is, they are helpful for having readymade contracts, only we need to implement functionality over this contract.   
  
I mean to say, Dial would remain Dial in case of Mobile or Telephone. It won’t be fair if we give different name when its task is to Call the person.  
  
Interface is defined with the keyword ‘interface’ .All properties & methods with in the interface should be implemented if it is been used. That’s the rule of interface.



interface IMobile

{

   Void Dial();

}

interface ITelephone

{

   void Dial();

}

public class Mobile : IMobile, ITelephone

{  
    public void Dial()  
    {  
            Console.WriteLine("Dial a number");  
    }     
}

**Conclusion**  
Following the above principle & keeping in mind the four pillars would lead you develop a good program & linking it with real world scenario will make you think more deeply. I hope you like this article. Don’t forget to share your comment whether it’s good or bad. Sharing is valuable no matter what.  
  
Download the file for example  
  
**Reference**: [Introduction to Object Oriented Programming Concepts (OOP) and More](http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep)

<http://www.dotnetspider.com/forum/301020-OOP-Real-time-Scenario.aspx>

OOPs mainly consists of 4 principles:-  
  
1. Encapsulation  
2. Abstraction  
3. Polymorphism  
4. Inheritance  
  
so these principles are used in the real-time according to the requirements in the project applications.  
  
1. Encapsulation:- Accessing the public properties by using private variables is one of the good examples of Encapsulation in all the projects.  
  
e.g.

private string \_name;  
public String Name  
{  
get{return \_name}  
set{\_name= value;}  
}

Here you are not able to access the private variable \_name but its value can be accessible by the public property Name. So a kind of Encapsulation here.  
it means its binding the data and the logic but hiding the private data.  
  
2. Abstraction:- When there are the specific requirements and that can be flown to their child, then we can use abstraction. it means whatever is essential, we implement to its child classes, else will be inherited automatically.  
like you create an abstract class which contain abstract as well as concrete members. so we least bother about the concrete members but we only concentrate on the Abstract members becasue they must be implemented in the child classes.  
  
e.g.

public abstract class A  
{  
public abstract void Add(int a, int b);  
protected int x;  
public string Name{get; set;};  
}  
Class B:A  
{  
 public override void Add(int a, int b)  
 {  
 Console.WriteLine("Sum is:" + int. Parse(a + b).toString());  
 }  
}

Here we least bother about the variable x as well as the Name property.  
  
3. Polymorphism:- When we have the requirement of the similar functionality with more specifications then we go for Polymorphism concepts.  
e.g. we need the sum of 2 numbers, 3 number and 4 numbers. In this case, instead of creating 3 different methods, we will create 3 methods with same name but with different arguments:

public void Add(int a, int b)  
{  
 Console.WriteLine("Sum of 2 numbers" + int. Parse(a + b).toString());  
}  
  
public void Add(int a, int b, int c)  
{  
 Console.WriteLine("Sum of 3 numbers" + int. Parse(a + b +c).toString());  
}  
  
public void Add(int a, int b, int c, int d)  
{  
 Console.WriteLine("Sum of 4 numbers" + int. Parse(a + b + c + d).toString());  
}

Polymorphism:  
  
Example of Compile Time Polymorphism  
  
Method Overloading  
- Method with same name but with different arguments is called method overloading.  
- Method Overloading forms compile-time polymorphism.  
- Example of Method Overloading:  
class A1  
{  
void hello()  
{ Console.WriteLine("Hello"); }  
  
void hello(string s)  
{ Console.WriteLine("Hello {0}",s); }  
}  
  
Example of Run Time Polymorphism  
  
Method Overriding  
- Method overriding occurs when child class declares a method that has the same type arguments as a method declared by one of its superclass.  
- Method overriding forms Run-time polymorphism.  
- Note: By default functions are not virtual in C# and so you need to write "virtual" explicitly. While by default in Java each function are virtual.  
- Example of Method Overriding:  
Class parent  
{  
virtual void hello()  
{ Console.WriteLine("Hello from Parent"); }  
}  
  
Class child : parent  
{  
override void hello()  
{ Console.WriteLine("Hello from Child"); }  
}  
  
static void main()  
{  
parent objParent = new child();  
objParent.hello();  
}  
//Output  
Hello from Child.  
  
4. Inheritance:- Inheriting one class from other class is called as inheritance. it can be from base class to child class or from one base class to other based class/parent class.  
So when we have such situation where we have the similar kind of objects which has the similar attributes then we use this concept.  
  
I hope you will get the idea of real-time implementation with these.  
for more details and other OOPs concepts with the real-time scenarios, you can go through the below link which will take you for rest of the OOPs concepts implementation in the real-time scenarios:

http://pawantechit.blogspot.com/2011/09/oops-concepts-when-and-how-to-use.html

Encapsulation and abstraction:  
  
class House  
  
{  
  
// Data Members  
  
public string DrawingRoom;  
  
private string Kitchen;  
  
// Member Functions  
  
private bool EnterDrawingRoom()  
  
{  
  
DrawingRoom = "The Drawing Room";  
  
}  
  
public bool EnterKitchen()  
  
{  
  
Kitchen = "The Kitchen"; /\* Even though Kitchen is declared Private, EnterKitchen function  
  
can modify it because the member function itself has access to all other members. \*/  
  
}  
  
}  
  
class MainClass  
  
{  
  
public static void Main()  
  
{  
  
House TheBlackHouse = new House();  
  
TheBlackHouse.DrawingRoom = "Main Drawing Room"; /\* OK - Public Data Member is accessible  
  
from outside the class \*/  
  
TheBlackHouse.Kitchen = "Main Kitchen"; /\* Error - Private Data Member is not accessible  
  
from outside the class. \*/  
  
TheBlackHouse.EnterDrawingRoom(); /\* Error - Private Member Function is not accessible  
  
// from outside the class. \*/  
  
TheBlackHouse.EnterKitchen(); /\* OK - Public Data Member is accessible from outside  
  
the class. \*/  
  
}  
  
}  
  
  
  
Lets study the code. Inside the main function we declare an object TheBlackHouse of the House class. Thereafter, we set the value of the data member DrawingRoom for TheBlackHouse to "Main Drawing Room". Since, DrawingRoom is declared as public, the above step works without any error. On the next line, TheBlackHouse.Kitchen = \"Main Kitchen\"; we are trying to assign the value "Main Kitchen" to the Kitchen variable of TheBlackHouse object. However, Kitchen is declared as private. Hence, this step generates an error.  
  
In the next line, we are trying to call the private method EnterDrawingRoom for the object. This generates an error as well. Finally, we are calling the EnterKitchen function which is declared as public. Notice that inside the EnterKitchen function, we are assigning the value \"The Kitchen\" to the private variable Kitchen. This does not generate an error because the EnterKitchen method belongs to the same class and it has access to all the members inside the class, irrespective of the access specifier.

In this webinar we try to understand OOP with 10 real time scenario as shown below.   
1. The duplicate code scenario (Classes and objects)  
2. Complicated class scenario (Abstraction and Encapsulation)   
3. The cluttercode class scenario (Polymorphism)   
4. The confusing half class scenario (Abstract classes)   
5. Non-uniform vocabulary scenario (Interfaces)   
6. Decoupling scenario (Interface + Dynamic polymorphism)   
7. Confusing method scenario (Static polymorphism)   
8. Third party property scenario (Shadowing)   
9. Object life time issue (aggregation and composition)   
10. Small datatype scenario (structures and classes)   
  
The Target Audience: Level 100 - Level 300   
(Developers)  
  
This is a free Online Webcast, so you can visit from anywhere. Please visit the link to get complete instructions.  
  
<http://kolkatageeks.com/WebCast/JoinWebCast.aspx>

<http://www.c-sharpcorner.com/UploadFile/tusharkantagarwal/objectorientedcsharp11162005070743AM/objectorientedcsharp.aspx>

# Object Oriented Concepts in C#

In this article we will discuss key concepts of object orientation with their practical implementation in C#. We will discuss here basics of **OOPS** including Interfaces, Access Modifiers, inheritance, polymorphism etc. This is my second article on csharp-corner.com.

My first article was "Memory Management in .NET",  
you can find this article at [http://www.c-sharpcorner.com/  
UploadFile/tkagarwal/  
MemoryManagementInNet11232005064832AM/  
MemoryManagementInNet.aspx](http://www.c-sharpcorner.com/UploadFile/tkagarwal/MemoryManagementInNet11232005064832AM/MemoryManagementInNet.aspx)

Key Concepts of Object Orientation

* Abstraction
* Encapsulation
* Polymorphism
* Inheritance.

**Abstraction** is the ability to generalize an object as a data type that has a specific set of characteristics and is able to perform a set of actions.

Object-oriented languages provide abstraction via classes. Classes define the properties and methods of an object type.

Examples:

* You can create an abstraction of a dog with characteristics, such as color, height, and weight, and actions such as run and bite. The characteristics are called properties, and the actions are called methods.
* A Recordset object is an abstract representation of a set of data.

Classes are blueprints for Object.  
Objects are instance of classes.

C# Example of Class:  
  
public class Draw  
{  
// Class code.  
}

**Object References**When we work with an object we are using a reference to that object. On the other hand, when we are working with simple data types such as Integer, we are working with the actual value rather than a reference.

When we create a new object using the New keyword, we store a reference to that object in a variable. For instance:

Draw MyDraw = new Draw;

This code creates a new instance of Draw. We gain access to this new object via the MyDraw variable. This variable holds a reference to the object.

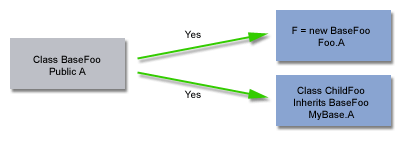
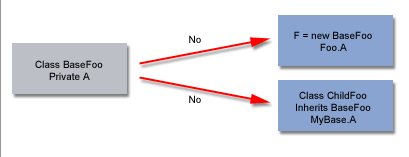
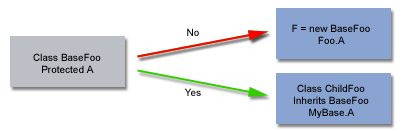
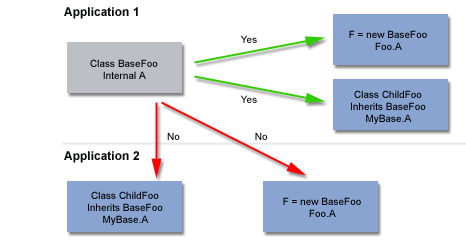
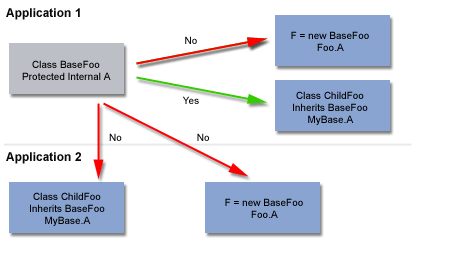
Now we have a second variable, which also has a reference to that same object. We can use either variable interchangeably, since they both reference the exact same object. The thing we need to remember is that the variable we have is not the object itself but, rather, is just a reference or pointer to the object itself.

**Early binding** means that our code directly interacts with the object, by directly calling its methods. Since the compiler knows the object's data type ahead of time, it can directly compile code to invoke the methods on the object. Early binding also allows the IDE to use IntelliSense to aid our development efforts; it allows the compiler to ensure that we are referencing methods that do exist and that we are providing the proper parameter values.

**Late binding** means that our code interacts with an object dynamically at run-time. This provides a great deal of flexibility since our code literally doesn't care what type of object it is interacting with as long as the object supports the methods we want to call. Because the type of the object isn't known by the IDE or compiler, neither IntelliSense nor compile-time syntax checking is possible but we get unprecedented flexibility in exchange.

If we enable strict type checking by using Option Strict On at the top of our code modules, then the IDE and compiler will enforce early binding behavior. By default, Option Strict is turned off and so we have easy access to the use of late binding within our code.

**Access Modifiers**Access Modifiers are keywords used to specify the declared accessibility of a member of a type.

**Public** is visible to everyone. A public member can be accessed using an instance of a class, by a class's internal code, and by any descendants of a class.  
  
  
  
**Private** is hidden and usable only by the class itself. No code using a class instance can access a private member directly and neither can a descendant class.   
  
  
  
**Protected** members are similar to private ones in that they are accessible only by the containing class. However, protected members also may be used by a descendant class. So members that are likely to be needed by a descendant class should be marked protected.  
  
   
  
**Internal/Friend**is public to the entire application but private to any outside applications. Internal is useful when you want to allow a class to be used by other applications but reserve special functionality for the application that contains the class. Internal is used by C# and Friend by VB .NET.  
  
   
  
**Protected Internal**may be accessed only by a descendant class that's contained in the same application as its base class. You use protected internal in situations where you want to deny access to parts of a class functionality to any descendant classes found in other applications.  
  


**Composition of an OBJECT**

We use an interface to get access to an object's data and behavior. The object's data and behaviors are contained within the object, so a client application can treat the object like a black box accessible only through its interface. This is a key object-oriented concept called Encapsulation. The idea is that any programs that make use of this object won't have direct access to the behaviors or data-but rather those programs must make use of our object's interface.

There are three main parts of Object:

1. Interface  
2. Implementation or Behavior  
3. Member or Instance variables

**Interface**

The interface is defined as a set of methods (Sub and Function routines), properties (Property routines), events, and fields (variables or attributes) that are declared Public in scope.

**Implementation or Behavior**

The code inside of a method is called the implementation. Sometimes it is also called behavior since it is this code that actually makes the object do useful work.   
Client applications can use our object even if we change the implementation-as long as we don't change the interface. As long as our method name and its parameter list and return data type remain unchanged, we can change the implementation all we want.

So Method Signature depends on:

* Method name
* Data types of parameters
* Either Parameter is passed ByVal or ByRef.
* Return type of method.

It is important to keep in mind that encapsulation is a syntactic tool-it allows our code to continue to run without change. However, it is not semantic-meaning that, just because our code continues to run, doesn't mean it continues to do what we actually wanted it to do.

**Member or Instance Variables**

The third key part of an object is its data, or state. Every instance of a class is absolutely identical in terms of its interface and its implementation-the only thing that can vary at all is the data contained within that particular object.

Member variables are those declared so that they are available to all code within our class. Typically member variables are Private in scope-available only to the code in our class itself. They are also sometimes referred to as instance variables or as attributes. The .NET Framework also refers to them as fields.   
We shouldn't confuse instance variables with properties. A Property is a type of method that is geared around retrieving and setting values, while an instance variable is a variable within the class that may hold the value exposed by a Property.

Interface looks like a class, but has no implementation.

The only thing it contains is definitions of events, indexers, methods and/or properties. The reason interfaces only provide definitions is because they are inherited by classes and structs, which must provide an implementation for each interface member defined. So, what are interfaces good for if they don't implement functionality? They're great for putting together plug-n-play like architectures where components can be interchanged at will. Since all interchangeable components implement the same interface, they can be used without any extra programming. The interface forces each component to expose specific public members that will be used in a certain way.

Because interfaces must be defined by inheriting classes and structs, they define a contract. For instance, if class foo inherits from the IDisposable interface, it is making a statement that it guarantees it has the Dispose() method, which is the only member of the IDisposable interface. Any code that wishes to use class foo may check to see if class foo inherits IDisposable. When the answer is true, then the code knows that it can call foo.Dispose().

Defining an Interface: MyInterface.c  
  
interface IMyInterface  
{  
void MethodToImplement();  
}  
  
Above listing shows defines an interface named IMyInterface. A common naming convention is to prefix all interface names with a capital "I", but this is not mandatory. This interface has a single method named MethodToImplement(). This could have been any type of method declaration with different parameters and return types. Notice that this method does not have an implementation (instructions between curly braces- {}), but instead ends with a semi-colon, ";". This is because the interface only specifies the signature of methods that an inheriting class or struct must implement.

All the methods of Interface are public by default and no access modifiers (like private, public) are allowed with any method of Interface.

Using an Interface: InterfaceImplementer.cs  
  
class InterfaceImplementer : IMyInterface  
{  
public void MethodToImplement()  
{  
Console.WriteLine("MethodToImplement() called.");  
}  
}  
  
The InterfaceImplementer class in above listing implements the IMyInterface interface. Indicating that a class inherits an interface is the same as inheriting a class. In this case, the following syntax is used:  
  
class InterfaceImplementer : IMyInterface

Note that this class inherits the IMyInterface interface; it must implement its all members. While implementing interface methods all those needs to be declared public only. It does this by implementing the MethodToImplement() method. Notice that this method implementation has the exact same signature, parameters and method name, as defined in the IMyInterface interface. Any difference will cause a compiler error. Interfaces may also inherit other interfaces. Following listing shows how inherited interfaces are implemented.

Interface Inheritance: InterfaceInheritance.cs  
  
using System;  
interface IParentInterface  
{  
void ParentInterfaceMethod();  
}  
interface IMyInterface : IParentInterface  
{  
void MethodToImplement();  
}  
class InterfaceImplementer : IMyInterface  
{  
public void MethodToImplement()  
{  
Console.WriteLine("MethodToImplement() called.");  
}  
public void ParentInterfaceMethod()  
{  
Console.WriteLine("ParentInterfaceMethod() called.");  
}  
}  
  
The code in above listing contains two interfaces: IMyInterface and the interface it inherits, IParentInterface. When one interface inherits another, any implementing class or struct must implement every interface member in the entire inheritance chain. Since the InterfaceImplementer class in above listing inherits from IMyInterface, it also inherits IParentInterface. Therefore, the InterfaceImplementer class must implement the MethodToImplement() method specified in the IMyInterface interface and the ParentInterfaceMethod() method specified in the IParentInterface interface.

In summary, you can implement an interface and use it in a class. Interfaces may also be inherited by other interface. Any class or struct that inherits an interface must also implement all members in the entire interface inheritance chain.

**Inheritance** is the idea that one class, called a subclass, can be based on another class, called a base class. Inheritance provides a mechanism for creating hierarchies of objects.

Inheritance is the ability to apply another class's interface and code to your own class.

Normal base classes may be instantiated themselves, or inherited. Derived classes can inherit base class members marked with protected or greater access. The derived class is specialized to provide more functionality, in addition to what its base class provides. Inheriting base class members in derived class is not mandatory.  
  
**Access Keywords**  
base -> Access the members of the base class.  
this -> Refer to the current object for which a method is called.

The base keyword is used to access members of the base class from within a derived class:   
Call a method on the base class that has been overridden by another method. Specify which base-class constructor should be called when creating instances of the derived class. A base class access is permitted only in a constructor, an instance method, or an instance property accessor.

In following example, both the base class, Person, and the derived class, Employee, have a method named Getinfo. By using the base keyword, it is possible to call the Getinfo method on the base class, from within the derived class.

// Accessing base class members  
  
using System;  
public class Person  
{  
protected string ssn = "444-55-6666";  
protected string name = "John L. Malgraine";  
public virtual void GetInfo()  
{  
Console.WriteLine("Name: {0}", name);  
Console.WriteLine("SSN: {0}", ssn);  
}  
}  
class Employee: Person  
{  
public string id = "ABC567EFG";  
public override void GetInfo()  
{  
// Calling the base class GetInfo method:  
base.GetInfo();  
Console.WriteLine("Employee ID: {0}", id);  
}  
}  
class TestClass   
{  
public static void Main()  
{  
Employee E = new Employee();  
E.GetInfo();  
}  
}  
  
Output  
Name: John L. Malgraine  
SSN: 444-55-6666  
Employee ID: ABC567EFG

Base class constructors can be called from derived classes. To call a base class constructor, use the base() constructor reference. This is desirable when it's necessary to initialize a base class appropriately.

Here's an example that shows the derived class constructor with an address parameter:  
  
abstract public class Contact  
{  
private string address;  
public Contact(string b\_address)  
{  
this.address = b\_address;  
}  
}  
public class Customer : Contact  
{  
public Customer(string c\_address) : base(C\_address)  
{  
}  
}  
  
In this code, the Customer class does not have an address, so it passes the parameter to its base class constructor by adding a colon and the base keyword with the parameter to its declaration. This calls the Contact constructor with the address parameter, where the address field in Contact is initialized.

One more example which shows how base-class constructor is called when creating instances of a derived class:  
  
using System;  
public class MyBase  
{  
int num;  
public MyBase()   
{  
Console.WriteLine("In MyBase()");  
}  
public MyBase(int i)  
{  
num = i;  
Console.WriteLine("in MyBase(int i)");  
}  
public int GetNum()  
{  
return num;  
}  
}  
public class MyDerived : MyBase  
{  
static int i = 32;  
// This constructor will call MyBase.MyBase()  
public MyDerived(int ii) : base()  
{  
}  
// This constructor will call MyBase.MyBase(int i)  
public MyDerived() : base(i)  
{  
}  
public static void Main()  
{  
MyDerived md = new MyDerived(); // calls public MyDerived() : base(i) and  
// passes i=32 in base class  
MyDerived md1 = new MyDerived(1); // call public MyDerived() : base(i)  
}  
}  
  
Output   
in MyBase(int i)  
in MyBase()

The following example will not compile. It illustrates the effects of not including a default constructor in a class definition:  
  
abstract public class Contact  
{  
private string address;  
public Contact(string address)  
{  
this.address = address;  
}  
}  
public class Customer : Contact  
{  
public Customer(string address)  
{  
}  
}  
  
In this example, the Customer constructor does not call the base class constructor. This is obviously a bug, since the address field will never be initialized.

When a class has no explicit constructor, the system assigns a default constructor. The default constructor automatically calls a default or parameterless base constructor. Here's an example of automatic default constructor generation that would occur for the preceding example:

public Customer() : Contact()  
{  
}  
  
When a class does not declare any constructors, the code in this example is automatically generated. The default base class constructor is called implicitly when no derived class constructors are defined. Once a derived class constructor is defined, whether or not it has parameters, a default constructor will not be automatically defined, as the preceding code showed.

**Calling Base Class Members**

Derived classes can access the members of their base class if those members have protected or greater access. Simply use the member name in the appropriate context, just as if that member were a part of the derived class itself. Here's an example:

abstract public class Contact  
{  
private string address;  
private string city;  
private string state;  
private string zip;  
public string FullAddress()  
{  
string fullAddress = address + '\n' + city + ',' + state + ' ' + zip;  
return fullAddress;  
}  
}  
public class Customer : Contact  
{  
public string GenerateReport()  
{  
string fullAddress = FullAddress();  
// do some other stuff...  
return fullAddress;  
}  
}  
  
In above example, the GenerateReport() method of the Customer class calls the FullAddress() method in its base class, Contact. All classes have full access to their own members without qualification. Qualification refers to using a class name with the dot operator to access a class member-MyObject.SomeMethod(), for instance. This shows that a derived class can access its base class members in the same manner as its own.

More Tips regarding Inheritance:

* A static member cannot be marked as override, virtual, or abstract. So following is an error:  
  public static virtual void GetSSN()
* You can't call static methods of base class from derived class using base keyword.  
  In above example if you declare a static method as follows:

public class Person  
{  
protected string ssn = "444-55-6666";  
protected string name = "John L. Malgraine";  
public static void GetInfo()  
{  
// Implementation  
}  
}

now you can't call this method using base.GetInfo() from derived class instead you have to call Person.GetInfo() from derived class.

Inside Static members we can access only static fields, methods etc.  
Following example will give error, because we can't access name in GetInfo() because name is not static.

public class Person  
{  
protected string ssn = "444-55-6666";  
protected string name = "John L. Malgraine";  
public static void GetInfo()  
{  
Console.WriteLine("Name: {0}", name);  
Console.WriteLine("SSN: {0}", ssn);  
}  
}  
  
Virtual or abstract members cannot be private.

* If you are not overriding a virtual method of base class in derived class, you can't use base class method by using base keyword in derived class. Also when you will create an instance of derived class, it will call derived class method and you will only be able to access base class method when you will create instance of base class.
* You can't decrease access level of a method in derived class when you are overriding a base class method in derived class, vice versa is possible.  
  Means you can make protected method of base class to public in derived class.

The "this" keyword refers to:

* the current instance for which a method is called. Static member functions do not have a this pointer. The this keyword can be used to access members from within constructors, instance methods, and instance accessors.   
  The following are common uses of this:

To qualify members hidden by similar names, for example:

public Employee(string name, string alias)  
{  
this.name = name;  
this.alias = alias;  
}  
  
In above example, this.name refers to private variable name in the class. If we write name = name, then this will refer to argument name of the constructor Employee and not to private variable name in the class. In this case private variable name will never be initialized.

* To pass an object as a parameter to other methods, for example:   
  CalcTax(this);

To declare indexers, for example: 

public int this [int param]  
{  
get  
{  
return array[param];  
}  
set  
{  
array[param] = value;  
}  
}  
  
It is an error to refer to this in a static method, static property accessor, or variable initializer of a field declaration.

In this example, this is used to qualify the Employee class members, name and alias, which are hidden by similar names. It is also used to pass an object to the method CalcTax, which belongs to another class.

// keywords\_this.cs  
// this example  
using System;  
public class Employee  
{  
public string name;  
public string alias;  
public decimal salary = 3000.00m;  
// Constructor:  
public Employee(string name, string alias)  
{  
// Use this to qualify the fields, name and alias:  
this.name = name;  
this.alias = alias;  
}  
// Printing method:  
public void printEmployee()  
{  
Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);  
// Passing the object to the CalcTax method by using this:  
Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));  
}  
}  
public class Tax  
{  
public static decimal CalcTax(Employee E)  
{  
return (0.08m\*(E.salary));  
}  
}  
public class MainClass  
{  
public static void Main()  
{  
// Create objects:  
Employee E1 = new Employee ("John M. Trainer", "jtrainer");  
// Display results:  
E1.printEmployee();  
}  
}  
  
Output   
Name: John M. Trainer  
Alias: jtrainer  
Taxes: $240.00

**Abstract Classes**

Abstract classes are a special type of base classes. In addition to normal class members, they have abstract class members. These Abstract class members are methods and properties that are declared without an implementation. All classes derived directly from abstract classes must implement all of these abstract methods and properties.

Abstract classes can never be instantiated. This would be illogical, because of the members without implementations.So what good is a class that can't be instantiated? Lots! Abstract classes sit toward the top of a class hierarchy. They establish structure and meaning to code. They make frameworks easier to build. This is possible because abstract classes have information and behavior common to all derived classes in a framework. Take a look at the following example:

abstract public class Contact // Abstract Class Contact.  
{  
protected string name;  
public Contact()  
{  
// statements...  
}  
public abstract void generateReport();  
abstract public string Name  
{  
get;  
set;  
}  
}  
  
Contact, is an abstract class. Contact has two abstract members, and it has an abstract method named generateReport(). This method is declared with the abstract modifier in front of the method declaration. It has no implementation (no braces) and is terminated with a semicolon. The Name property is also declared abstract. The accessors of properties are terminated with semicolons.

public class Customer : Contact // Customer Inherits Abstract Class Contact.  
{  
string gender;  
decimal income;  
int numberOfVisits;  
public Customer()  
{  
// statements  
}  
public override void generateReport()  
{  
// unique report  
}  
public override string Name  
{  
get  
{  
numberOfVisits++;  
return name;  
}  
set  
{  
name = value;  
numberOfVisits = 0;  
}  
}  
}  
public class SiteOwner : Contact  
{  
int siteHits;  
string mySite;  
public SiteOwner()  
{  
// statements  
}  
public override void generateReport()  
{  
// unique report  
}  
public override string Name  
{  
get  
{  
siteHits++;  
return name;  
}  
set  
{  
name = value;  
siteHits = 0;  
}  
}  
}  
  
The abstract base class Contact has two derived classes, Customer and SiteOwner. Both of these derived classes implement the abstract members of the Contact class. The generateReport() method in each derived class has an override modifier in its declaration. Likewise, the Name declaration contains an override modifier in both Customer and SiteOwner.

C# requires explicit declaration of intent when overriding methods. This feature promotes safe code by avoiding the accidental overriding of base class methods, which is what actually does happen in other languages. Leaving out the override modifier generates an error. Similarly, adding a new modifier also generates an error. Abstract methods must be overridden and cannot be hidden, which the new modifier or the lack of a modifier would be trying to do.

The most famous of all abstract classes is the Object class. It may be referred to as object or Object, but it's still the same class. Object is the base class for all other classes in C#. It's also the default base class when a base class is not specified. The following class declarations produce the same exact results:  
  
abstract public class Contact : Object  
{  
// class members  
}  
abstract public class Contact  
{  
// class members  
}  
  
Object is implicitly included as a base class if it is not already declared. Besides providing the abstract glue to hold together the C# class framework, object includes built-in functionality, some of which is useful for derived classes to implement.

**Difference between Interface and Abstract Class**

* Interfaces are closely related to abstract classes that have all members abstract.
* For an abstract class, at least one method of the class must be an abstract method that means it may have concrete methods.
* For an interface, all the methods must be abstract
* Class that implements an interface much provide concrete implementation of all the methods definition in an interface or else must be declare an abstract class
* In C#, multiple inheritance is possible only through implementation of multiple interfaces. Abstract class can only be derived once.
* An interface defines a contract and can only contains four entities viz methods, properties, events and indexes. An interface thus cannot contain constants, fields, operators, constructors, destructors, static constructors, or types.
* Also an interface cannot contain static members of any kind. The modifiers abstract, public, protected, internal, private, virtual, override is disallowed, as they make no sense in this context.
* Class members that implement the interface members must be publicly accessible.

**Overriding Summery:**A derived class may override a virtual method of the base class with the keyword override. The following restrictions must be followed.

* Keyword override is used in the definition of child class method that is going to override the base class's virtual method.
* The return type must be the same as the virtual method have in base class.
* The name of the method should also be same.
* The parameter-list must also be same in order, number and type of parameters.
* The accessibility of the overriding method should not be more restricted than that of the accessibility defined with virtual method of the base class. This accessibility either be the same or less restricted.
* The virtual methods can be sealed in the child or derived classes to prevent further modifications in the implementation of the virtual method in the derived classes, by declaring them as sealed methods.

**Hiding Base Class Members**

Sometimes derived class members have the same name as a corresponding base class member. In this case, the derived member is said to be "hiding" the base class member.

When hiding occurs, the derived member is masking the functionality of the base class member. Users of the derived class won't be able to see the hidden member; they'll see only the derived class member. The following code shows how hiding a base class member works.

abstract public class Contact  
{  
private string address;  
private string city;  
private string state;  
private string zip;  
public string FullAddress()  
{  
string fullAddress =address + '\n' +city + ',' + state + ' ' + zip;  
return fullAddress;  
}  
}  
public class SiteOwner : Contact  
{  
public string FullAddress()  
{  
string fullAddress;  
// create an address...  
return fullAddress;  
}  
}  
  
In this example, both SiteOwner and its base class, Contact, have a method named FullAddress(). The FullAddress() method in the SiteOwner class hides the FullAddress() method in the Contact class. This means that when an instance of a SiteOwner class is invoked with a call to the FullAddress() method, it is the SiteOwner class FullAddress() method that is called, not the FullAddress() method of the Contact class.

Although a base class member may be hidden, the derived class can still access it. It does this through the base identifier. Sometimes this is desirable. It is often useful to take advantage of the base class functionality and then add to it with the derived class code. The next example shows how to refer to a base class method from the derived class.

abstract public class Contact  
{  
private string address;  
private string city;  
private string state;  
private string zip;  
public string FullAddress()  
{  
string fullAddress =address + '\n' +city + ',' + state + ' ' + zip;  
return fullAddress;  
}  
}  
public class SiteOwner : Contact  
{  
public string FullAddress()  
{  
string fullAddress = base.FullAddress();  
// do some other stuff...  
return fullAddress;  
}  
}  
  
In this particular example, the FullAddress() method of the Contact class is called from within the FullAddress() method of the SiteOwner class. This is accomplished with a base class reference. This provides another way to reuse code and add on to it with customized behavior.

**Versioning**

Versioning, in the context of inheritance, is a C# mechanism that allows modification of classes (creating new versions) without accidentally changing the meaning of the code. Hiding a base class member with the methods previously described generates a warning message from the compiler. This is because of the C# versioning policy. It's designed to eliminate a class of problems associated with modifications to base classes.

Here's the scenario: A developer creates a class that inherits from a third-party library. For the purposes of this discussion, we assume that the Contact class represents the third-party library. Here's the example:

public class Contact  
{  
// does not include FullAddress() method  
}  
public class SiteOwner : Contact  
{  
public string FullAddress()  
{  
string fullAddress = mySite.ToString();  
return fullAddress;  
}  
}  
  
In this example, the FullAddress() method does not exist in the base class. There is no problem yet. Later on, the creators of the third-party library update their code. Part of this update includes a new member in a base class with the exact same name as the derived class:

public class Contact  
{  
private string address;  
private string city;  
private string state;  
private string zip;  
public string FullAddress()  
{  
string fullAddress =address + '\n' +city + ',' + state + ' ' + zip;  
return fullAddress;  
}  
}  
public class SiteOwner : Contact  
{  
public string FullAddress()  
{  
string fullAddress = mySite.ToString();  
return fullAddress;  
}  
}  
  
In this code, the base class method FullAddress() contains different functionality than the derived class method. In other languages, this scenario would break the code because of implicit polymorphism. However, this does not break any code in C# because when the FullAddress() method is called on SiteOwner, it is still the SiteOwner class method that gets called.

This scenario generates a warning message. One way to eliminate the warning message is to place a new modifier in front of the derived class method name, as the following example shows:

using System;  
public class WebSite  
{  
public string SiteName;  
public string URL;  
public string Description;  
public WebSite()  
{  
}  
public WebSite( string strSiteName, string strURL, string strDescription )  
{  
SiteName = strSiteName;  
URL = strURL;  
Description = strDescription;  
}  
public override string ToString()  
{  
return SiteName + ", " +URL + ", " +Description;  
}  
}  
public class Contact  
{  
public string address;  
public string city;  
public string state;  
public string zip;  
public string FullAddress()  
{  
string fullAddress =address + '\n' +city + ',' + state + ' ' + zip;  
return fullAddress;  
}  
}  
public class SiteOwner : Contact  
{  
int siteHits;  
string name;  
WebSite mySite;  
public SiteOwner()  
{  
mySite = new WebSite();  
siteHits = 0;  
}  
public SiteOwner(string aName, WebSite aSite)  
{  
mySite = new WebSite(aSite.SiteName,aSite.URL,aSite.Description);  
Name = aName;  
}  
new public string FullAddress()  
{  
string fullAddress = mySite.ToString();  
return fullAddress;  
}  
public string Name  
{  
get  
{  
siteHits++;  
return name;  
}  
set  
{  
name = value;  
siteHits = 0;  
}  
}  
}  
public class Test  
{  
public static void Main()  
{  
WebSite mySite = new WebSite("Le Financier","http://www.LeFinancier.com","Fancy Financial Site");  
SiteOwner anOwner = new SiteOwner("John Doe", mySite);  
string address;  
anOwner.address = "123 Lane Lane";  
anOwner.city = "Some Town";  
anOwner.state = "HI";  
anOwner.zip = "45678";  
address = anOwner.FullAddress(); // Different Results  
Console.WriteLine("Address: \n{0}\n", address);  
}  
}  
  
Here's the output:  
Address:  
Le Financier, [http://www.LeFinancier.com](http://www.lefinancier.com/), Fancy Financial Site

This has the effect of explicitly letting the compiler know the developer's intent. Placing the new modifier in front of the derived class member states that the developers know there is a base class method with the same name, and they definitely want to hide that member. This prevents breakage of existing code that depends on the implementation of the derived class member. With C#, the method in the derived class is called when an object of the derived class type is used. Likewise, the method in the base class is called when an object of the Base class type is called. Another problem this presents is that the base class may present some desirable new features that wouldn't be available through the derived class.

To use these new features requires one of a few different workarounds. One option would be to rename the derived class member, which would allow programs to use a base class method through a derived class member. The drawback to this option would be if there were other classes relying upon the implementation of the derived class member with the same name. This scenario will break code and, for this reason, is considered extremely bad form.

Another option is to define a new method in the derived class that called the base class method. This allows users of the derived class to have the new functionality of the base class, yet retain their existing functionality with the derived class. While this would work, there are maintainability concerns for the derived class.

**Sealed Classes**

Sealed classes are classes that can't be derived from. To prevent other classes from inheriting from a class, make it a sealed class. There are a couple good reasons to create sealed classes, including optimization and security.

Sealing a class avoids the system overhead associated with virtual methods. This allows the compiler to perform certain optimizations that are otherwise unavailable with normal classes.

Another good reason to seal a class is for security. Inheritance, by its very nature, dictates a certain amount of protected access to the internals of a potential base class. Sealing a class does away with the possibility of corruption by derived classes. A good example of a sealed class is the String class. The following example shows how to create a sealed class:

public sealed class CustomerStats  
{  
string gender;  
decimal income;  
int numberOfVisits;  
public CustomerStats()  
{  
}  
}  
public class CustomerInfo : CustomerStats // error  
{  
}  
  
This example generates a compiler error. Since the CustomerStats class is sealed, it can't be inherited by the CustomerInfo class.The CustomerStats class was meant to be used as an encapsulated object in another class. This is shown by the declaration of a CustomerStats object in the Customer class.

public class Customer  
{  
CustomerStats myStats; // okay  
}  
  
**Polymorphism**

Polymorphism is reflected in the ability to write one routine that can operate on objects from more than one class-treating different objects from different classes in exactly the same way. For instance, if both Customer and Vendor objects have a Name property, and we can write a routine that calls the Name property regardless of whether we're using a Customer or Vendor object, then we have polymorphism.

A vehicle is a good example of polymorphism. A vehicle interface would only have those properties and methods that all vehicles have, a few of which might include paint color, number of doors, accelerator, and ignition. These properties and methods would apply to all types of vehicles including cars, trucks, and semi-trucks.

Polymorphism will not implement code behind the vehicle's properties and methods. Instead, polymorphism is the implementation of an interface. If the car, truck, and semitruck all implement the same vehicle interface, then the client code for all three classes can be exactly the same.

C# gives us polymorphism through inheritance. C# provides a keyword virtual that is used in the definition of a method to support polymorphism.

Child class are now free to provide their own implementation of this virtual method, that is called overriding. The following points are important regarding virtual keyword:-   
  
If the method is not virtual, the compiler simply uses the reference type to invoke the appropriate method.

If the method is virtual, the compiler will generate code to checkup the reference type at runtime it is actually denoting to, then the appropriate method is called from the class of the reference type.

When a virtual method is called, runtime check (late method binding) is made to identify the object and appropriate method is invoked, all this is done at runtime.

In case of non-virtual methods, this information is available at compile time, so no runtime check to identify the object is made, so slightly efficient in the way non-virtual methods are called. But the behavior of virtual method is useful in many ways; the functionality they provide is fair enough to bear this slight loss of performance.

**Implementing Polymorphism**The key factor here is the ability to dynamically invoke methods in a class based on their type. Essentially, a program would have a group of objects, examine the type of each one, and execute the appropriate method. Here's an example:

using System;  
public class WebSite  
{  
public string SiteName;  
public string URL;  
public string Description;  
public WebSite()  
{  
}  
public WebSite( string strSiteName, string strURL, string strDescription )  
{  
SiteName = strSiteName;  
URL = strURL;  
Description = strDescription;  
}  
public override string ToString()  
{  
return SiteName + ", " +URL + ", " +Description;  
}  
}  
  
When we inherit above class, we have two choices to invoke constructor of the class. So this is an example of design time polymorphism. Here at design time we have to decide which method we need to invoke while inheriting the class.

Polymorphism is the capability of a program to carry out dynamic operations by implementing methods of multiple derived classes through a common base class reference. Another definition of polymorphism is the ability to treat different objects the same way. This means that the runtime type of an object determines its behavior rather than the compile-time type of its reference.