[C#](https://www.c-sharpcorner.com/technologies/csharp-programming)

Understanding Polymorphism in C#

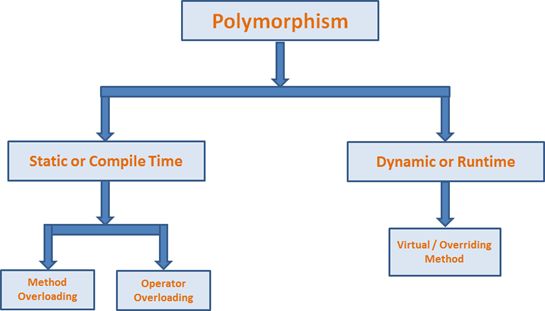
Introduction

Polymorphism is a Greek word meaning "one name many forms". In other words, one object has many forms or has one name with multiple functionalities. "Poly" means many and "morph" means forms. Polymorphism provides the ability to class multiple implementations with the same name. It is one principle concept in Object Oriented Programming after encapsulation and inheritance.

Types of Polymorphism

There are basically the following two types of polymorphism in C#:

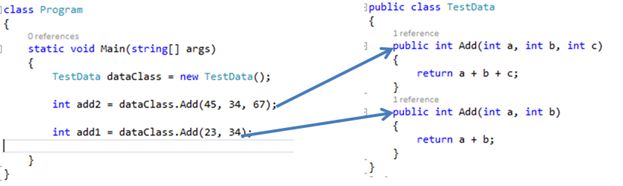
* Static / Compile Time Polymorphism.
* Dynamic / Runtime Polymorphism.



Static or Compile Time Polymorphism

It is also known as Early Binding. Method overloading is an example of Static Polymorphism. In Overloading, the method / function has the same name but different signatures. It is also known as Compile Time Polymorphism because the decision of which method is to be called is made at compile time. Overloading is the concept in which method names are the same with a different set of parameters.  
  
Here the compiler checks the number of parameters passed and the type of parameter and make the decision of which method to call and it throw an error if no matching method is found.  
  
In the following example the class has two methods with the same name "Add" but with different input parameters (the first method has three parameters and the second method has two parameters).

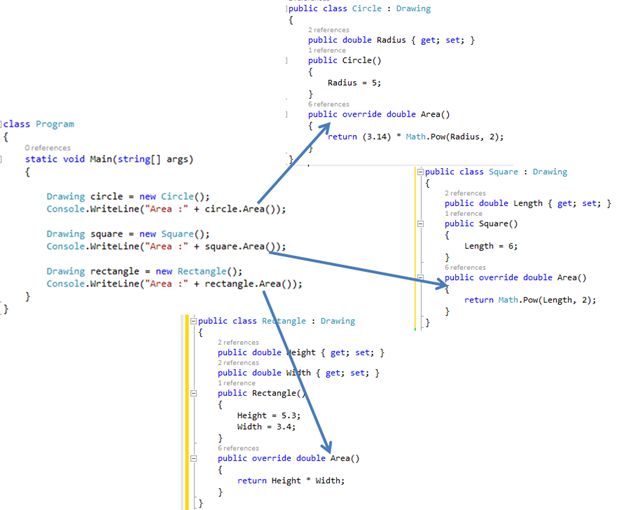
1. **public** **class** TestData
2. {
3. **public** **int** Add(**int** a, **int** b, **int** c)
4. {
5. **return** a + b + c;
6. }
7. **public** **int** Add(**int** a, **int** b)
8. {
9. **return** a + b;
10. }
11. }
12. **class** Program
13. {
14. **static** **void** Main(**string**[] args)
15. {
16. TestData dataClass = **new** TestData();
17. **int** add2 = dataClass.Add(45, 34, 67);
18. **int** add1 = dataClass.Add(23, 34);
19. }
20. }



Dynamic / Runtime Polymorphism

Dynamic / runtime polymorphism is also known as late binding. Here, the method name and the method signature (number of parameters and parameter type must be the same and may have a different implementation). Method overriding is an example of dynamic polymorphism.  
  
Method overriding can be done using inheritance. With method overriding it is possible for the base class and derived class to have the same method name and same something. The compiler would not be aware of the method available for overriding the functionality, so the compiler does not throw an error at compile time. The compiler will decide which method to call at runtime and if no method is found then it throws an error.

1. **public** **class** Drawing
2. {
3. **public** **virtual** **double** Area()
4. {
5. **return** 0;
6. }
7. }
9. **public** **class** Circle : Drawing
10. {
11. **public** **double** Radius { **get**; **set**; }
12. **public** Circle()
13. {
14. Radius = 5;
15. }
16. **public** **override** **double** Area()
17. {
18. **return** (3.14) \* Math.Pow(Radius, 2);
19. }
20. }
22. **public** **class** Square : Drawing
23. {
24. **public** **double** Length { **get**; **set**; }
25. **public** Square()
26. {
27. Length = 6;
28. }
29. **public** **override** **double** Area()
30. {
31. **return** Math.Pow(Length, 2);
32. }
33. }
35. **public** **class** Rectangle : Drawing
36. {
37. **public** **double** Height { **get**; **set**; }
38. **public** **double** Width { **get**; **set**; }
39. **public** Rectangle()
40. {
41. Height = 5.3;
42. Width = 3.4;
43. }
44. **public** **override** **double** Area()
45. {
46. **return** Height \* Width;
47. }
48. }
50. **class** Program
51. {
52. **static** **void** Main(**string**[] args)
53. {
55. Drawing circle = **new** Circle();
56. Console.WriteLine("Area :" + circle.Area());
58. Drawing square = **new** Square();
59. Console.WriteLine("Area :" + square.Area());
61. Drawing rectangle = **new** Rectangle();
62. Console.WriteLine("Area :" + rectangle.Area());
63. }
64. }

  
  
The compiler requires an Area() method and it compiles successfully but the right version of the Area() method is not being determined at compile time but determined at runtime. Finally the overriding methods must have the same name and signature (number of parameters and type), as the virtual or abstract method defined in the base class method and that it is overriding in the derived class.  
  
A method or function of the base class is available to the child (derived) class without the use of the "overriding" keyword. The compiler hides the function or method of the base class. This concept is known as shadowing or method hiding. You may find the difference between overriding and shadowing [here](http://www.c-sharpcorner.com/UploadFile/ff2f08/overriding-vs-shadowing-in-C-Sharp/).

Preventing Derived class from overriding virtual members

Virtual members remain “virtual” indefinitely. In other words, virtual members remain “virtual” regardless of how many classes have been between virtual members and the class that originally declared it. For example, if class X has the virtual method "A" and the class Y is derived from X and the class Z is derived from Y, class Z inherits the virtual method "A" and override it.

1. **public** **class** X
2. {
3. **public** **virtual** **void** A()
4. {
5. }
6. }
7. **public** **class** Y : X
8. {
9. **public** **override** **void** A()
10. {
11. }
12. }

A derived class is able to stop virtual inheritance by declaring an override member as "sealed".

1. **public** **class** Y : X
2. {
3. **public** **sealed** **override** **void** A()
4. {
5. }
6. }

Accessing Base class virtual member

Using the "base" keyword, the derived class is able to access the method.

1. **public** **class** X
2. {
3. **public** **virtual** **void** A()
4. {
5. }
6. }
7. **public** **class** Y : X
8. {
9. **public** **override** **void** A()
10. {
11. **base**.A();
12. }
13. }

Summary

* The meaning of Polymorphism is one name having multiple forms.
* The following are the two types of Polymorphism:  
  + Static or compile-time polymorphism (for example, method overloading and operator overloading).
  + Dynamic or runtime polymorphism (for example, overriding).
* Method Overriding differs from shadowing.
* Using the "new" keyword, we can hide the base class member.
* We can prevent a derived class from overriding virtual members.
* We can access a base class virtual member from the derived class.

**Understand Concept Of Abstract Base Class In C#**

**Abstraction**  
  
The word abstract means a concept or an idea not associated with any specific instance.  
  
In programming, we apply the same meaning of abstraction by making classes not associated with any specific instance.  
  
Abstraction is done when we need to only inherit from a certain class, but do not need to instantiate objects of that class. In such a case the base class can be regarded as "Incomplete". Such classes are known as an "Abstract Base Class".  
  
**Abstract Base Class**  
  
There are some important points about Abstract Base Class :

1. An Abstract Base class cannot be instantiated; it means the object of that class cannot be created.
2. Class having the abstract keyword with some of its methods (not all) is known as an Abstract Base Class.
3. Class having the Abstract keyword with all of its methods is known as pure Abstract Base Class.
4. The method of the abstract class that has no implementation is known as "operation". It can be defined as an abstract void method ();
5. An abstract class holds the methods but the actual implementation of those methods is made in derived class.

Let's have a look at this code!

1. abstractclassanimal {
2. publicabstractvoid eat();
3. publicvoid sound() {
4. Console.WriteLine("dog can sound");
5. }
6. }

This is the Abstract Base Class, if I make both of its methods abstract then this class would become a pure Abstract Base Class.  
  
Now, we derive a class of 'dog' from the class animal.

1. abstractclassanimal {
2. publicabstractvoid eat();
3. publicvoid sound() {
4. Console.WriteLine("dog can sound");
5. }
6. }
7. classdog: animal {
8. publicoverridevoid eat() {
9. Console.WriteLine("dog can eat");
10. }
11. }

Here you can see we have 2 methods in the Abstract Base Class, the method eat() has no implementation; that is why it is being declared as 'abstract' while the method sound() has its own body so it is not declared as 'abstract'.  
  
In the derived class, we have the same name method but this method has its body.  
  
We are doing abstraction here so that we can access the method of the derived class without any trouble.  
  
Let's have a look!

1. classprogram {
2. abstractclassanimal {
3. publicabstractvoid eat();
4. publicvoid sound() {
5. Console.WriteLine("dog can sound");
6. }
7. }
8. classdog: animal {
9. publicoverridevoid eat() {
10. Console.WriteLine("dog can eat");
11. }
12. }
13. staticvoid Main(string[] args) {
14. dog mydog = newdog();
15. animal thePet = mydog;
16. thePet.eat();
17. mydog.sound();
18. }
19. }

Finally we created an Object 'mydog' of class dog, but we didn't instantiate any object of Abstract Base Class 'animal'.  
  
According to "Ivor Horton" (a programmer of Java) an object can not be instantiated, but we can declare a variable of the Abstract Class type. If this statement is true then it could be possible:  
  
animal thePet;  
  
This is an object which is declared as thePet and its data type is the abstract base class 'animal'.  
  
We can use this Object to store Objects of the subclass.  
  
In the above code, we declare an Object 'thePet', of the type animal (the Abstract Base Class) and simply copied the object of another object (only the reference is copied as they belong to reference type). Now we can use object 'thePet' just as object 'mydog'.   
  
The output of this code would be as,

* dog can eat
* dog can sound

**Conclusion**  
  
l conclude here by saying that Abstraction is not a difficult job to do, but you need to be confident while performing abstraction. Every new topic covers all the previous topics. In abstraction, polymorphism is covered and performed. The method overriding could be done by putting the keyword 'new' before that overridden method. Everything is possible in Programming and there are multiple ways to do a single job. Abstraction is one of the smart ways to do this kind of tasks.

# Encapsulation In C#

**INTRODUCTION**

The object oriented programming will give the impression very unnatural to a programmer with a lot of procedural programming experience. In Object Oriented programming Encapsulation is the first pace. Encapsulation is the procedure of covering up of data and functions into a single unit (called class). An encapsulated object is often called an abstract data type. In this article let us see about it in a detailed manner.

**NEED FOR ENCAPSULATION**

The need of encapsulation is to protect or prevent the code (data) from accidental corruption due to the silly little errors that we are all prone to make. In Object oriented programming data is treated as a critical element in the program development and data is packed closely to the functions that operate on it and protects it from accidental modification from outside functions.

Encapsulation provides a way to protect data from accidental corruption. Rather than defining the data in the form of public, we can declare those fields as private. The Private data are manipulated indirectly by two ways. Let us see some example programs in C# to demonstrate Encapsulation by those two methods. The first method is using a pair of conventional accessor and mutator methods. Another one method is using a named property. Whatever be the method our aim is to use the data with out any damage or change.

**ENCAPSULATION USING ACCESSORS AND MUTATORS**

Let us see an example of Department class. To manipulate the data in that class (String departname) we define an accessor (get method) and mutator (set method).

1. using system;
2. **public** **class** Department {
3. **private** string departname;.......
4. // Accessor.
5. **public** string GetDepartname() {
6. **return** departname;
7. }
8. // Mutator.
9. **public** **void** SetDepartname(string a) {
10. departname = a;
11. }
12. }

Like the above way we can protect the private data from the outside world. Here we use two separate methods to assign and get the required data.

1. **public** **static** **int** Main(string[] args) {
2. Department d = **new** Department();
3. d.SetDepartname("ELECTRONICS");
4. Console.WriteLine("The Department is :" + d.GetDepartname());
5. **return** 0;
6. }

In the above example we can't access the private data departname from an object instance. We manipulate the data only using those two methods.

**ENCAPSULATION USING PROPERTIES**

Properties are a new language feature introduced with C#. Only a few languages support this property. Properties in C# helps in protect a field in a class by reading and writing to it. The first method itself is good but Encapsulation can be accomplished much smoother with properties.

Now let's see an example.

1. using system;
2. **public** **class** Department {
3. **private** string departname;
4. **public** string Departname {
5. get {
6. **return** departname;
7. }
8. set {
9. departname = value;
10. }
11. }
12. }
13. **public** **class** Departmentmain {
14. **public** **static** **int** Main(string[] args) {
15. Department d = **new** Department();
16. d.departname = "Communication";
17. Console.WriteLine("The Department is :{0}", d.Departname);
18. **return** 0;
19. }
20. }

From the above example we see the usage of Encapsulation by using properties. The property has two accessor get and set. The get accessor returns the value of the some property field. The set accessor sets the value of the some property field with the contents of "value". Properties can be made read-only. This is accomplished by having only a get accessor in the property implementation.

**READ ONLY PROPERTY**

1. using system;
2. **public** **class** ReadDepartment {
3. **private** string departname;
4. **public** ReadDepartment(string avalue) {
5. departname = avalue;
6. }
7. **public** string Departname {
8. get {
9. **return** departname;
10. }
11. }
12. }
13. **public** **class** ReadDepartmain {
14. **public** **static** **int** Main(string[] args) {
15. ReadDepartment d = **new** ReadDepartment("COMPUTERSCIENCE");
16. Console.WriteLine("The Department is: {0}", d.Departname);
17. **return** 0;
18. }
19. }

In the above example we see how to implement a read-only property. The class ReadDepartment has a Departname property that only implements a get accessor. It leaves out the set accessor. This particular class has a constructor, which accepts a string parameter. The Main method of the ReadDepartmain class creates a new object named d. The instantiation of the d object uses the constructor of the ReadDepartment that takes a string parameter. Since the above program is read-only, we cannot set the value to the field departname and we only read or get the value of the data from the field. Properties can be made also Write-only. This is accomplished by having only a set accessor in the property implementation.

**WRITE ONLY PROPERTY**

1. using system;
2. **public** **class** WriteDepartment {
3. **private** string departname;
4. **public** string Departname {
5. set {
6. departname = value;
7. Console.WriteLine("The Department is :{0}", departname);
8. }
9. }
10. }
11. **public** **class** WriteDepartmain {
12. **public** **static** **int** Main(string[] args) {
13. WriteDepartment d = **new** WriteDepartment();
14. d.departname = "COMPUTERSCIENCE";
15. **return** 0;
16. }
17. }

In the above example we see how to implement a Write-only property. The class WriteDepartment has now has a Departname property that only implements a set accessor. It leaves out the get accessor. The set accessor method is varied a little by it prints the value of the departname after it is assigned.

**CONCLUSION**The Encapsulation is the first footstep towards the object-oriented programming. This article gives you a little bit information about Encapsulation. Using accessor and mutator methods we can make encapsulation. Another one method is using a named property. The benefit of properties is that the users of your objects are able to manipulate the internal data point using a single named item.

**Definition - What does *Encapsulation* mean?**

Encapsulation, in the context of C#, refers to an object's ability to hide data and behavior that are not necessary to its user. Encapsulation enables a group of properties, methods and other members to be considered a single unit or object.  
  
The following are the benefits of encapsulation:

* Protection of data from accidental corruption
* Specification of the accessibility of each of the members of a class to the code outside the class
* Flexibility and extensibility of the code and reduction in complexity
* Lower coupling between objects and hence improvement in code maintainability

Encapsulation is used to restrict access to the members of a class so as to prevent the user of a given class from manipulating objects in ways that are not intended by the designer. While encapsulation hides the internal implementation of the functionalities of class without affecting the overall functioning of the system, it allows the class to service a request for functionality and add or modify its internal structure (data or methods) to suit changing requirements.  
  
Encapsulation is also known as information hiding.

**Techopedia explains *Encapsulation***

Encapsulation in C# is implemented with different levels of access to object data that can be specified using the following access modifiers:

* Public: Access to all code in the program
* Private: Access to only members of the same class
* Protected: Access to members of same class and its derived classes
* Internal: Access to current assembly
* Protected Internal: Access to current assembly and types derived from containing class

Encapsulation can be illustrated with an example of an Employee object that stores details of that object. By using encapsulation, the Employee object can expose the data (like Name, EmployeeID, etc.) and methods (like GetSalary) necessary for using the object, while hiding its irrelevant fields and methods from other objects. It is easy to see a situation in which all users could access basic information about an employee while restricting salary information.   
  
C# allows encapsulation of data through the use of accessors (to get data) and mutators (to modify data), which help in manipulating private data indirectly without making it public. Properties are an alternate mechanism for private data to be encapsulated in a C# object and accessed in either read-only mode or in read-write mode. Unlike the accessor and mutator, a property provides a single point of access to an object's "set" and "get" values.