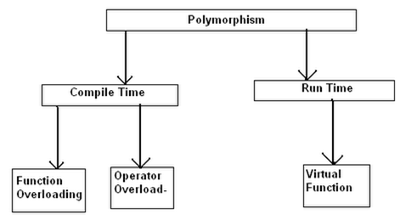
**Polymorphism**

The word **polymorphism** means having many forms.

In Polymorphism we have 2 different types those are

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

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



1. Static Polymorphism

Early Binding

Overloading

same method names with different signatures in same class

Example:

public class Class1

{

public void NumbersAdd(int a, int b)

{

Console.WriteLine(a + b);

}

public void NumbersAdd(int a, int b, int c)

{

Console.WriteLine(a + b + c);

}

}

They are:

* Function overloading
* Operator overloading

## Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

The following example shows using function **print()** to print different data types:

using System;

namespace PolymorphismApplication

{

class Printdata

{

void print(int i)

{

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

}

void print(double f)

{

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

}

void print(string s)

{

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

}

static void Main(string[] args)

{

Printdata p = new Printdata();

// Call print to print integer

p.print(5);

// Call print to print float

p.print(500.263);

// Call print to print string

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

Console.ReadKey();

}

}

}

When the above code is compiled and executed, it produces the following result:

Printing int: 5

Printing float: 500.263

Printing string: Hello C++

Compiler checks the type and number of parameters passed on to the method and decides which method to call at compile time and it will give an error if there are no methods that match the method signature of the method that is called at compile time.

2)Dynamic Polymorphism

Late Binding

Overriding

same method names with same signatures.in different class

Examle:

//Base Class

public class Bclass

{

public virtual void Sample1()

{

Console.WriteLine("Base Class");

}

}

// Derived Class

public class DClass : Bclass

{

public override void Sample1()

{

Console.WriteLine("Derived Class");

}

}

// Using base and derived class

class Program

{

static void Main(string[] args)

{

// calling the overriden method

DClass objDc = new DClass();

objDc.Sample1();

// calling the base class method

Bclass objBc = new DClass();

objBc.Sample1();

((Bclass)objDc).Sample1(); // Typecasting

}

}

Output

----------------------------------

Derived Class

Derived Class

Base class

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

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

using System;

namespace PolymorphismApplication

{

abstract class Shape

{

public abstract int area();

}

class Rectangle: Shape

{

private int length;

private int width;

public Rectangle( int a=0, int b=0)

{

length = a;

width = b;

}

public override int area ()

{

Console.WriteLine("Rectangle class area :");

return (width \* length);

}

}

class RectangleTester

{

static void Main(string[] args)

{

Rectangle r = new Rectangle(10, 7);

double a = r.area();

Console.WriteLine("Area: {0}",a);

Console.ReadKey();

}

}

}

**No need to use Virtual keyword when using abstract class.**

### **Real World Example of Polymorphism**

#### Example 1

* A Teacher behaves with student.
* A Teacher behaves with his/her seniors.

Here teacher is an object but the attitude is different in different situations.

#### Example 2

* Person behaves as a SON in house, at the same time that person behaves like an EMPLOYEE in the office.

#### Example 3

Your mobile phone, one name but many forms:

* As phone
* As camera
* As mp3 player
* As radio

### **Virtual Keyword**

According to MSDN, “The virtual keyword is used to modify a method, property, indexer or event declaration, and allow it to be overridden in a derived class.”

### **Virtual Method**

Virtual method is a method whose behavior can be overridden in derived class. Virtual method allows declare a method in base class that can be redefined in each derived class.

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

* By default, methods are non-virtual. You cannot override a non-virtual method.
* You cannot use the virtual modifier with the static, abstract, private or override modifiers.
* Virtual properties behave like abstract methods, except for the differences in declaration and invocation syntax.
* It is an error to use the virtual modifier on a static property.
* A virtual inherited property can be overridden in a derived class by including a property declaration that uses the override modifier.

Virtual method solves the following problem:

In OOP, when a derived class inherits from a base class, an object of the derived class may be referred to (or cast) as either being the base class type or the derived class type. If there are base class methods overridden by the derived class, the method call behavior is ambiguous.

In C#, polymorphism is explicit - you must have a virtual (or abstract) modifier on the base class method (member) and an override on the derived class method, which you probably already know.

If you don't put a modifier on a base class method, polymorphism can't ever happen. If you then add a non-modified method to the derived class with the same signature as the non-modified base class method, the compiler will generate a Warning message.

namespace PolymorphismByManishAgrahari

{

class Program

{

public class Base

{

public void Show()

{

Console.WriteLine("Show From Base Class.");

}

}

public class Derived : Base

{

*//Following line will Give an Warning*

*/\**

*'PolymorphismByManishAgrahari.Program.Derived.Show()'*

*hides inherited member*

*'PolymorphismByManishAgrahari.Program.Base.Show()'.*

*Use the new keyword if hiding was intended.*

*\*/*

public void Show()

{

Console.WriteLine("Show From Derived Class.");

}

}

static void Main(string[] args)

{

Base objBase = new Base();

objBase.Show();*// Output ----> Show From Base Class.*

Derived objDerived = new Derived();

objDerived.Show();*//Output--> Show From Derived Class.*

Base objBaseRefToDerived = new Derived();

objBaseRefToDerived.Show();*//Output--> Show From Base Class.*

Console.ReadLine();

}

}

}

In other languages, take Java for instance, you have what is called "implicit" polymorphism where just putting the method in the derived class with the same signature as a base class method will enable polymorphism.

In Java, all methods of a class are virtual by default unless the developer decides to use the final keyword thus preventing subclasses from overriding that method. In contrast, C# adopts the strategy used by C++ where the developer has to use the virtual keyword for subclasses to override the method. Thus all methods in C# are non virtual by default.

The C# approach is more explicit for the purpose of making the code safer in versioning scenarios, i.e., you build your code based on a 3rd party library and use meaningful, but common, method names. The 3rd party library upgrades, using the same common method name. With implicit polymorphism the code would break, but with C#, you would receive a compiler warning so you can double check to see if polymorphism was something you wanted to do.

**Interview questions for calling a method from base class**

namespace PolymorphismByManishAgrahari

{

class Program

{

public class Base

{

public void Show()

{

Console.WriteLine("Show From Base Class.");

}

}

public class Derived : Base

{

*//Following line will Give an Warning*

*/\**

*'PolymorphismByManishAgrahari.Program.Derived.Show()'*

*hides inherited member*

*'PolymorphismByManishAgrahari.Program.Base.Show()'.*

*Use the new keyword if hiding was intended.*

*\*/*

public void Show()

{

Console.WriteLine("Show From Derived Class.");

}

}

static void Main(string[] args)

{

Base objBase = new Base();

objBase.Show();*// Output ----> Show From Base Class.*

Base class method has no virtual keyword and derived class has not override that method

Derived objDerived = new Derived();

objDerived.Show();*//Output--> Show From Derived Class.*

Base objBaseRefToDerived = new Derived();

objBaseRefToDerived.Show();*//Output--> Show From Base Class.*

Console.ReadLine();

}

}

}

### **Difference between Method Overriding and Method Hiding**

Method overriding allows a subclass to provide a specific implementation of a method that is already provided by base class. The implementation in the subclass overrides (replaces) the implementation in the base class.

namespace PolymorphismByManishAgrahari

{

class Program

{

public class Base

{

public virtual void Show()

{

Console.WriteLine("Show From Base Class.");

}

}

public class Derived : Base

{

*//the keyword "override" change the base class method.*

public override void Show()

{

Console.WriteLine("Show From Derived Class.");

}

}

static void Main(string[] args)

{

Base objBaseRefToDerived = new Derived();

objBaseRefToDerived .Show();*//Output--> Show From Derived Class.*

Console.ReadLine();

}

}

}

**Override keyword override a previous implementation due to having virtual keyword of method.**

**Method hiding does not have a relationship between the methods in the base class and derived class. The method in the derived class hides the method in the base class.**

Hide   Shrink http://www.codeproject.com/images/arrow-up-16.png   Copy Code

namespace PolymorphismByManishAgrahari

{

class Program

{

public class Base

{

public virtual void Show()

{

Console.WriteLine("Show From Base Class.");

}

}

public class Derived : Base

{

public new void Show()

{

Console.WriteLine("Show From Derived Class.");

}

}

static void Main(string[] args)

{

Base objBaseRefToDerived = new Derived();

objBaseRefToDerived .Show();*//Output--> Show From Base Class.*

Console.ReadLine();

}

}

}

Output is:? Show From Base Class.

In the preceding example, Derived.Show will be called; because, it overrides Base.Show.

**C# language specification states that "You cannot override a non-virtual method**

namespace PolymorphismByManishAgrahari

{

class Program

{

public class Base

{

public void Show()

{

Console.WriteLine("This is Base Class.");

}

}

public class Derived : Base

{

*//Following Line will give error.*

*/\**

*Error:- 'PolymorphismByManishAgrahari.Program.Derived.Show()'*

*cannot override inherited member 'PolymorphismByManishAgrahari.Program.Base.Show()'*

*\* because it is not marked virtual, abstract, or override*

*\*/*

public override void Show()

{

Due to base class method is Non virtual ,it can not be overridden

Console.WriteLine("This is Derived Class.");

}

}

static void Main(string[] args)

{

Base objBase = new Base();

objBase.Show();*// Output ----> This is Base Class.*

Derived objDerived = new Derived();

objDerived.Show();*//Output--> This is Derived Class.*

Base objBaseRefToDerived = new Derived();

objBaseRefToDerived.Show();*//Output--> This is Base Class.*

Console.ReadLine();

}

}

}

## Summary

1. It is not compulsory to mark the derived/child class function with override keyword while base/parent class contains a virtual method.
2. Virtual methods allow subclasses to provide their own implementation of that method using theoverride keyword.
3. Virtual methods can't be declared as private.
4. You are not required to declare a method as virtual. But, if you don't, and you derive from the class, and your derived class has a method by the same name and signature, you'll get a warning that you are hiding a parent's method.
5. A virtual property or method has an implementation in the base class, and can be overridden in the derived classes.
6. We will get a warning if we won't use Virtual/New keyword.
7. Instead of Virtual, we can use New keyword.