Polymorphism (C# Programming Guide)

07/20/2015 • 7 minutes to read • Contributors all

In this article

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

- At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its run-time type.
- Base classes may define and implement virtual *methods*, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

Virtual methods enable you to work with groups of related objects in a uniform way. For example, suppose you have a drawing application that enables a user to create various kinds of shapes on a drawing surface. You do not know at compile time which specific types of shapes the user will create. However, the application has to keep track of all the various types of shapes that are created, and it has to update them in response to user mouse actions. You can use polymorphism to solve this problem in two basic steps:

1. Create a class hierarchy in which each specific shape class derives from a common base class.

2. Use a virtual method to invoke the appropriate method on any derived class through a single call to the base class method.

First, create a base class called `Shape`, and derived classes such as `Rectangle`, `Circle`, and `Triangle`. Give the `Shape` class a virtual method called `Draw`, and override it in each derived class to draw the particular shape that the class represents. Create a `List<Shape>` object and add a Circle, Triangle and Rectangle to it. To update the drawing surface, use a foreach loop to iterate through the list and call the `Draw` method on each `Shape` object in the list. Even though each object in the list has a declared type of `Shape`, it is the run-time type (the overridden version of the method in each derived class) that will be invoked.

C# Copy

```
using System;
using System.Collections.Generic;

public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
```

```
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Polymorphism at work #1: a Rectangle, Triangle and Circle
        // can all be used whereever a Shape is expected. No cast is
        // required because an implicit conversion exists from a derived
        // class to its base class.
        var shapes = new List<Shape>
        {
            new Rectangle(),
            new Triangle(),
            new Circle()
        };
```

```
        // Polymorphism at work #2: the virtual method Draw is
        // invoked on each of the derived classes, not the base class.
        foreach (var shape in shapes)
        {
            shape.Draw();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

}

/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
 */
```

In C#, every type is polymorphic because all types, including user-defined types, inherit from Object.

Polymorphism Overview

Virtual Members

When a derived class inherits from a base class, it gains all the methods, fields, properties and events of the base class. The designer of the derived class can choose whether to

- override virtual members in the base class,

- inherit the closest base class method without overriding it
- define new non-virtual implementation of those members that hide the base class implementations

A derived class can override a base class member only if the base class member is declared as virtual or abstract. The derived member must use the override keyword to explicitly indicate that the method is intended to participate in virtual invocation. The following code provides an example:

C# Copy

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Fields cannot be virtual; only methods, properties, events and indexers can be virtual. When a derived class overrides a virtual member, that member is called even when an instance of that class is being accessed as an instance of the base class. The following code provides an example:

C# Copy

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.
```

```
BaseClass A = (BaseClass)B;
A.DoWork();  // Also calls the new method.
```

Virtual methods and properties enable derived classes to extend a base class without needing to use the base class implementation of a method. For more information, see Versioning with the Override and New Keywords. An interface provides another way to define a method or set of methods whose implementation is left to derived classes. For more information, see Interfaces.

Hiding Base Class Members with New Members

If you want your derived member to have the same name as a member in a base class, but you do not want it to participate in virtual invocation, you can use the new keyword. The `new` keyword is put before the return type of a class member that is being replaced. The following code provides an example:

C# Copy

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
```

```
    }
}
```

Hidden base class members can still be accessed from client code by casting the instance of the derived class to an instance of the base class. For example:

C# Copy

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

Preventing Derived Classes from Overriding Virtual Members

Virtual members remain virtual indefinitely, regardless of how many classes have been declared between the virtual member and the class that originally declared it. If class A declares a virtual member, and class B derives from A, and class C derives from B, class C inherits the virtual member, and has the option to override it, regardless of whether class B declared an override for that member. The following code provides an example:

C# Copy

```
public class A
{
    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}
```

A derived class can stop virtual inheritance by declaring an override as sealed. This requires putting the `sealed` keyword before the `override` keyword in the class member declaration. The following code provides an example:

C# Copy

```
public class C : B
{
    public sealed override void DoWork() { }
}
```

In the previous example, the method `DoWork` is no longer virtual to any class derived from C. It is still virtual for instances of C, even if they are cast to type B or type A. Sealed methods can be replaced by derived classes by using the `new` keyword, as the following example shows:

C# Copy

```
public class D : C
{
    public new void DoWork() { }
}
```

In this case, if `DoWork` is called on D using a variable of type D, the new `DoWork` is called. If a variable of type C, B, or A is used to access an instance of D, a call to `DoWork` will follow the rules of virtual inheritance, routing those calls to the implementation of `DoWork` on class C.

Accessing Base Class Virtual Members from Derived Classes

A derived class that has replaced or overridden a method or property can still access the method or property on the base class using the `base` keyword. The following code provides an example:

C# Copy

```
public class Base
{
    public virtual void DoWork() {/*...*/ }
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
        //...
        // Call DoWork on base class
        base.DoWork();
    }
}
```

For more information, see base.

> **Note**
>
> It is recommended that virtual members use `base` to call the base class implementation of that member in their own implementation. Letting the base class behavior occur enables the derived class to concentrate on implementing behavior specific to the derived class. If the base class implementation is not called, it is up to the derived class to make their behavior compatible with the behavior of the base class.

In This Section

- Versioning with the Override and New Keywords
- Knowing When to Use Override and New Keywords
- How to: Override the ToString Method

See also

- C# Programming Guide
- Inheritance
- Abstract and Sealed Classes and Class Members
- Methods
- Events
- Properties
- Indexers
- Types