Poliformismo

El polimorfismo suele considerarse el tercer pilar de la programación orientada a objetos, después de la encapsulación y la herencia. Polimorfismo es una palabra griega que significa "con muchas formas" y tiene dos aspectos diferentes:

- En tiempo de ejecución, los objetos de una clase derivada pueden ser tratados como objetos de una clase base en lugares como parámetros de métodos y colecciones o matrices. Cuando ocurre, el tipo declarado del objeto ya no es idéntico a su tipo en tiempo de ejecución.
- Las clases base pueden definir e implementar métodos virtuales, y las clases derivadas pueden invalidarlos, lo que significa que pueden proporcionar su propia definición e implementación. En tiempo de ejecución, cuando el código de cliente llama al método, CLR busca el tipo en tiempo de ejecución del objeto e invoca esa invalidación del método virtual. Por lo tanto, en el código fuente puede llamar a un método en una clase base y hacer que se ejecute una versión del método de la clase derivada.

Los métodos virtuales permiten trabajar con grupos de objetos relacionados de manera uniforme. Por ejemplo, supongamos que tiene una aplicación de dibujo que permite a un usuario crear varios tipos de formas en una superficie de dibujo. En tiempo de compilación, no sabe qué tipos específicos de formas creará el usuario. Sin embargo, la aplicación tiene que realizar el seguimiento de los distintos tipos de formas que se crean, y tiene que actualizarlos en respuesta a las acciones del mouse del usuario. Para solucionar este problema en dos pasos básicos, puede usar el polimorfismo:

1. Crear una jerarquía de clases en la que cada clase de forma específica deriva de una clase base común.
2. Usar un método virtual para invocar el método apropiado en una clase derivada mediante una sola llamada al método de la clase base.

Primero, cree una clase base llamada `Shape` y clases derivadas como `Rectangle`, `Circle` y `Triangle`. Dé a la clase `Shape` un método virtual llamado `Draw` e invalídelo en cada clase derivada para dibujar la forma determinada que la clase representa. Cree un objeto `List<Shape>` y agregue Circle, Triangle y Rectangle a él. Para actualizar la superficie de dibujo, use un bucle foreach para iterar por la lista y llamar al método `Draw` en cada objeto `Shape`de la lista. Aunque cada objeto de la lista tenga un tipo declarado de `Shape`, se invocará el tipo en tiempo de ejecución (la versión invalidada del método en cada clase derivada).

```
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
 */
```

En C#, cada tipo es polimórfico porque todos los tipos, incluidos los definidos por el usuario, heredan de Object.

Introducción al polimorfismo

Miembros virtuales

Cuando una clase derivada hereda de una clase base, obtiene todos los métodos, campos, propiedades y eventos de la clase base. El diseñador de la clase derivada tiene las siguientes opciones:

- Invalidar los miembros virtuales de la clase base.

- Heredar el método de la clase base más próximo sin invalidarlo.
- Definir una nueva implementación no virtual de esos miembros que oculte las implementaciones de la clase base.

Una clase derivada puede invalidar un miembro de la clase base si este se declara como virtual o abstracto. El miembro derivado debe usar la palabra clave override para indicar explícitamente que el propósito del método es participar en una invocación virtual. El siguiente fragmento de código muestra un ejemplo:

```
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; }
    }
}
```

Los campos no pueden ser virtuales; solo los métodos, propiedades, eventos e indizadores pueden ser virtuales. Cuando una clase derivada invalida un miembro virtual, se llama a ese miembro aunque se acceda a una instancia de esa clase como una instancia de la clase base. El siguiente fragmento de código muestra un ejemplo:

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

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

Los métodos y propiedades virtuales permiten a las clases derivadas extender una clase base sin necesidad de usar la implementación de clase base de un método. Una interfaz proporciona otra manera de definir un método o conjunto de métodos cuya implementación se deja a las clases derivadas. Para más información, vea el módulo de Interfaces.

Ocultar miembros de clase base con nuevos miembros

Si quiere que el miembro derivado tenga el mismo nombre que un miembro de una clase base, pero no quiere que participe en la invocación virtual, puede usar la palabra clave new. La palabra clave `new` se coloca antes que el tipo devuelto del miembro de la clase que se está reemplazando. El siguiente fragmento de código muestra un ejemplo:

```
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; }
    }
}
```

Aún se puede acceder a los miembros de la clase base ocultos desde el código de cliente convirtiendo la instancia de la clase derivada en una instancia de la clase base. Por ejemplo:

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

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

Evitar que las clases derivadas invaliden los miembros virtuales

Los miembros virtuales permanecen virtuales indefinidamente, independientemente de cuántas clases se hayan declarado entre el miembro virtual y la clase que originalmente la declaró. Si la clase A declara un miembro virtual y la clase B deriva de A, y la clase C deriva de B, la clase C hereda el miembro virtual y tiene la opción de invalidarlo, independientemente de que la

clase B declarara una invalidación para ese miembro. El siguiente fragmento de código muestra un ejemplo:

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

Una clase derivada puede detener la herencia virtual al declarar una invalidación como sealed. Para ello, es necesario colocar la palabra clave `sealed` antes de la palabra clave `override` en la declaración del miembro de la clase. El siguiente fragmento de código muestra un ejemplo:

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

En el ejemplo anterior, el método `DoWork` ya no es virtual para ninguna clase que derive de C. Sigue siendo virtual para instancias de C, aunque se conviertan al tipo B o A. Los métodos sellados pueden reemplazarse por las clases derivadas al usar la palabra clave `new`, como se muestra en el ejemplo siguiente:

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

En este caso, si se llama a `DoWork` en D usando una variable de tipo D, se llama al nuevo `DoWork`. Si se usa una variable de tipo C, B o A para acceder a una instancia de D, la llamada a `DoWork` seguirá las reglas de herencia virtual y enrutará dichas llamadas a la implementación de `DoWork` en la clase C.

Acceder a miembros virtuales de clases base desde clases derivadas

Una clase derivada que ha reemplazado o invalidado un método o propiedad puede seguir accediendo al método o propiedad en la clase base usando la siguiente palabra clave `base`. El siguiente fragmento de código muestra un ejemplo:

```
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();
    }
}
```