The **virtual** keyword is used to modify a method or property declaration, in which case the method or the property is called a virtual member. The implementation of a virtual member can be changed by an [overriding member](https://msdn.microsoft.com/en-us/library/ebca9ah3%28v=vs.71%29.aspx) in a 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. (For more information on run-time type and most derived implementation, see [10.5.3 Virtual methods](https://msdn.microsoft.com/en-us/library/aa645767%28v=vs.71%29.aspx).)

By default, methods are non-virtual. You cannot override a non-virtual method.

You cannot use the **virtual** modifier with the following modifiers:

**static**   **abstract**   **override**

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.

For more information on virtual methods, see [10.5.3 Virtual methods](https://msdn.microsoft.com/en-us/library/aa645767%28v=vs.71%29.aspx).

**Example**

In this example, the class Dimensions contains the two coordinates x, y, and the Area() virtual method. Different shape classes such as Circle, Cylinder, and Sphere inherit the Dimensions class, and the surface area is calculated for each figure. Each derived class has it own override implementation of Area(). The program calculates and displays the proper area for each figure by invoking the proper implementation of Area() according to the object associated with the method.

// cs\_virtual\_keyword.cs

// Virtual and override

using System;

class TestClass

{

public class Dimensions

{

public const double pi = Math.PI;

protected double x, y;

public Dimensions()

{

}

public Dimensions (double x, double y)

{

this.x = x;

this.y = y;

}

public virtual double Area()

{

return x\*y;

}

}

public class Circle: Dimensions

{

public Circle(double r): base(r, 0)

{

}

public override double Area()

{

return pi \* x \* x;

}

}

class Sphere: Dimensions

{

public Sphere(double r): base(r, 0)

{

}

public override double Area()

{

return 4 \* pi \* x \* x;

}

}

class Cylinder: Dimensions

{

public Cylinder(double r, double h): base(r, h)

{

}

public override double Area()

{

return 2\*pi\*x\*x + 2\*pi\*x\*y;

}

}

public static void Main()

{

double r = 3.0, h = 5.0;

Dimensions c = new Circle(r);

Dimensions s = new Sphere(r);

Dimensions l = new Cylinder(r, h);

// Display results:

Console.WriteLine("Area of Circle = {0:F2}", c.Area());

Console.WriteLine("Area of Sphere = {0:F2}", s.Area());

Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());

}

}

**Output**

Area of Circle = 28.27

Area of Sphere = 113.10

Area of Cylinder = 150.80

In the preceding example, notice that the inherited classes Circle, Sphere, and Cylinder are all using constructors that initialize the base class, for example:

public Cylinder(double r, double h): base(r, h) {}

This is analogous to the C++ initialization list.