**Methods**

classes are encapsulated bundles of data and the methods that work on that data. Put another way, methods give classes their behavioral characteristics, and we name methods after the actions we want the classes to carry out on our behalf.

you can retrieve only one value per method call. Obviously, calling one method for every piece of necessary data would be cumbersome in many situations. For example, let's say you have a *Color* class that represents a given color with three values by using the standard RGB (red-green-blue) model for describing colors. Using only return values, you'd be forced to write code similar to that at the top of the following page to retrieve all three values.

// Assuming color is an instance of a Color class.

int red = color.GetRed();

int green = color.GetGreen();

int blue = color.GetBlue();

But what we want is something similar to this:

int red;

int green;

int blue;

color.GetRGB(red, green, blue);

However, there's a problem here. When the *color.GetRGB* method is called, the values for the *red*, *green*, and *blue* arguments are copied into the method's local stack and any changes that the method makes will not be made to the caller's variables

In C++, this problem is circumvented by having the calling method **pass pointers or references to the variable**s so that the method works on the caller's data. The solution in C# is similar. Actually, C# offers two similar solutions. The first involves the keyword *ref*. This keyword tells the C# compiler that the arguments being passed point to the same memory as the variables in the calling code. That way, if the called method modifies these values and then returns, the calling code's variables will have been modified. The following code illustrates how to use the ***ref*** keyword with the *Color* class example:

using System;

class Color

{

public Color()

{

this.red = 255;

this.green = 0;

this.blue = 125;

}

protected int red;

protected int green;

protected int blue;

**public void GetColors(ref int red, ref int green, ref int blue)**

{

red = this.red;

green = this.green;

blue = this.blue;

}

}

class RefTest1App

{

public static void Main()

{

Color color = new Color();

int red;

int green;

int blue;

**color.GetColors(ref red, ref green, ref blue);**

Console.WriteLine("red = {0}, green = {1}, blue = {2}",

red, green, blue);

}

}

Using the *ref* keyword has one drawback, and, in fact, because of this limitation, the code above **will not compile**. **When you use the *ref* keyword, you must initialize the passed arguments before calling the method.** Therefore, for this code to work, it must be modified like this:

using System;

class Color

{

public Color()

{

this.red = 255;

this.green = 0;

this.blue = 125;

}

protected int red;

protected int green;

protected int blue;

public void GetColors(ref int red, ref int green, ref int blue)

{

red = this.red;

green = this.green;

blue = this.blue;

}

}

class RefTest2App

{

public static void Main()

{

Color color = new Color();

**int red = 0;**

**int green = 0;**

**int blue = 0;**

color.GetColors(ref red, ref green, ref blue);

Console.WriteLine("red = {0}, green = {1}, blue = {2}",

red, green, blue);

}

}

In this example, initializing the variables that are going to be overwritten seems pointless, doesn't it? Therefore, C# provides an alternative means of passing an argument whose changed value needs to be seen by the calling code: the *out* keyword. Here's the same *Color* class example using the *out* keyword:

using System;

class Color

{

public Color()

{

this.red = 255;

this.green = 0;

this.blue = 125;

}

protected int red;

protected int green;

protected int blue;

public void GetColors(out int red, out int green, out int blue)

{

red = this.red;

green = this.green;

blue = this.blue;

}

}

class OutTest1App

{

public static void Main()

{

Color color = new Color();

int red;

int green;

int blue;

**color.GetColors(out red, out green, out blue);**

Console.WriteLine("red = {0}, green = {1}, blue = {2}",

red, green, blue);

}

}

The only difference between the *ref* keyword and the *out* keyword is that the *out* keyword doesn't require the calling code to initialize the passed arguments first. So when would the *ref* keyword be used? You should use the *ref* keyword when you need to be assured that the calling method has initialized the argument. In the examples above, *out* could be used because the method being called wasn't dependent on the value of the variable being passed. But what if a parameter value is used by the called method? Take a look at this code:

using System;

class Window

{

public Window(int x, int y)

{

this.x = x;

this.y = y;

}

protected int x;

protected int y;

public void Move(int x, int y)

{

this.x = x;

this.y = y;

}

**public void ChangePos(ref int x, ref int y)**

{

this.x += x;;

this.y += y;

x = this.x;

y = this.y;

}

}

class OutTest2App

{

public static void Main()

{

Window wnd = new Window(5, 5);

int x = 5;

int y = 5;

wnd.ChangePos(ref x, ref y);

Console.WriteLine("{0}, {1}", x, y);

x = -1;

y = -1;

wnd.ChangePos(ref x, ref y);

Console.WriteLine("{0}, {1}", x, y);

}

}

As you can see, the method being called—*Window.ChangePos*—bases its work on the values being passed in. In this case, the *ref* keyword forces the caller to initialize the value so that the method will function properly.