C# allows you to define the meaning of an operator relative to a class that you create. This process is called operator overloading. By overloading an operator, you expand its usage to your class. The effects of the operator are completely under your control and may differ from class to class. For example, a class that defines a linked list might use the + operator to add an object to the list. A class that implements a stack might use the + to push an object onto the stack. Another class might use the + operator in an entirely different way.

When an operator is overloaded, none of its original meaning is lost. It is simply that a new operation, relative to a specific class, is added. Therefore, overloading the + to handle a linked list, for example, does not cause its meaning relative to integers (that is, addition) to be changed.
Operator Overloading Fundamentals

- Operator overloading is closely related to method overloading. To overload an operator, use the operator keyword to define an operator method, which defines the action of the operator relative to its class.

- There are two forms of operator methods: one for unary operators and one for binary operators. The general form for each is shown here:

```java
// General form for overloading a unary operator
public static ret-type operator op(param-type operand)
{
    // operations
}

// General form for overloading a binary operator
public static ret-type operator op(param-type1 operand1, param-type1 operand2)
{
    // operations
}
```
Here, the operator that you are overloading, such as + or /, is substituted for op. The ret-type specifies the type of value returned by the specified operation. Although it can be any type you choose, the return value is often of the same type as the class for which the operator is being overloaded. This correlation facilitates the use of the overloaded operator in expressions. For unary operators, the operand is passed in operand. For binary operators, the operands are passed in operand1 and operand2. Notice that operator methods must be both public and static.

For unary operators, the operand must be of the same type as the class for which the operator is being defined. For binary operators, at least one of the operands must be of the same type as its class. Thus, you cannot overload any C# operators for objects that you have not created. For example, you can’t redefine + for int or string.

One other point: Operator parameters must not use the ref or out modifier.
Overloading Binary Operators

To see how operator overloading works, let’s start with an example that overloads two binary operators, the + and the –. The following program creates a class called ThreeD, which maintains the coordinates of an object in three-dimensional space. The overloaded + adds the individual coordinates of one ThreeD object to another. The overloaded – subtracts the coordinates of one object from the other (Figure 10.1).
// An example of operator overloading.
using System;
// A three-dimensional coordinate class.
class ThreeD {
    int x, y, z; // 3-D coordinates
    public ThreeD() { x = y = z = 0; }
    public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }
    // Overload binary +.
    public static ThreeD operator +(ThreeD op1, ThreeD op2)
    {
        ThreeD result = new ThreeD();
        /* This adds together the coordinates of the two points
           and returns the result. */
        result.x = op1.x + op2.x; // These are integer additions
        result.y = op1.y + op2.y; // and the + retains its original
        result.z = op1.z + op2.z; // meaning relative to them.
        return result;
    }
    ... ... ...
// Overload binary -. 
public static ThreeD operator -(ThreeD op1, ThreeD op2) 
{
    ThreeD result = new ThreeD();
    /* Notice the order of the operands. op1 is the left
       operand and op2 is the right. */
    result.x = op1.x - op2.x; // these are integer subtractions
    result.y = op1.y - op2.y;
    result.z = op1.z - op2.z;
    return result;
}

// Show X, Y, Z coordinates.
public void Show()
{
    Console.WriteLine(x + ",", y + ",", z);
}

class ThreeDDemo {
    static void Main() {
        ThreeD a = new ThreeD(1, 2, 3);
        ThreeD b = new ThreeD(10, 10, 10);
        ThreeD c;
        Console.Write("Here is a: ");
        a.Show();
        Console.WriteLine();
        Console.Write("Here is b: ");
        b.Show();
        Console.WriteLine();
```csharp
c = a + b; // add a and b together
Console.Write("Result of a + b: ");
c.Show();
Console.WriteLine();
c = a + b + c; // add a, b, and c together
Console.Write("Result of a + b + c: ");
c.Show();
Console.WriteLine();
c = c - a; // subtract a
Console.Write("Result of c - a: ");
c.Show();
Console.WriteLine();
c = c - b; // subtract b
Console.Write("Result of c - b: ");
c.Show();
Console.WriteLine();
```
Figure 10.1 Overloaded binary operators
Let’s examine the preceding program carefully, beginning with the overloaded operator +. When two objects of type ThreeD are operated on by the + operator, the magnitudes of their respective coordinates are added together, as shown in operator+( ). Notice, however, that this method does not modify the value of either operand. Instead, a new object of type ThreeD, which contains the result of the operation, is returned by the method. To understand why the + operation does not change the contents of either object, think about the standard arithmetic+ operation as applied like this: 10 + 12. The outcome of this operation is 22, but neither 10 nor 12 is changed.

Notice that operator+( ) returns an object of type ThreeD. Although the method could have returned any valid C# type, the fact that it returns a ThreeD object allows the + operator to be used in compound expressions, such as a+b+c. Here, a+b generates a result that is of type ThreeD.
Now, look at operator–( ). The – operator works just like the + operator except that the order of the parameters is important. Recall that addition is commutative, but subtraction is not. (That is, A – B is not the same as B – A!) For all binary operators, the first parameter to an operator method will contain the left operand. The second parameter will contain the one on the right. When implementing overloaded versions of the non commutative operators, you must remember which operand is on the left and which is on the right.

**Overloading Unary Operators**

The unary operators are overloaded just like the binary operators. The main difference, of course, is that there is only one operand. For example, here is a method that overloads the unary minus for the ThreeD class (Figure 10.2):
Figure 10.1 Overloaded unary operators

```java
// Overload unary -.
public static ThreeD operator -(ThreeD op)
{
    ThreeD result = new ThreeD();
    result.x = -op.x;
    result.y = -op.y;
    result.z = -op.z;
    return result;
}
```
Here, a new object is created that contains the negated fields of the operand. This object is then returned. Notice that the operand is unchanged. Again, this is in keeping with the usual meaning of the unary minus. For example, in an expression such as this,

\[ a = -b \]

- \(a\) receives the negation of \(b\), but \(b\) is not changed.
- In C#, overloading \(++\) and \(--\) is quite easy; simply return the incremented or decremented value, but don’t change the invoking object. C# will automatically handle that for you, taking into account the difference between the prefix and postfix forms. For example, here is an operator\(++(\ )\) method for the ThreeD class:
Here is an expanded version of the previous example program that demonstrates the unary– and the ++ operator (Figure 10.3):

```java
// Overload unary ++.
public static ThreeD operator ++(ThreeD op)
{
    ThreeD result = new ThreeD();
    // Return the incremented result.
    result.x = op.x + 1;
    result.y = op.y + 1;
    result.z = op.z + 1;
    return result;
}
```
// More operator overloading.
using System;
// A three-dimensional coordinate class.

class ThreeD {
    int x, y, z; // 3-D coordinates
    public ThreeD() { x = y = z = 0; }
    public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }
    // Overload binary +.
    public static ThreeD operator +(ThreeD op1, ThreeD op2)
    {
        ThreeD result = new ThreeD();
        /* This adds together the coordinates of the two points
         and returns the result. */
        result.x = op1.x + op2.x;
        result.y = op1.y + op2.y;
        result.z = op1.z + op2.z;
        return result;
    }
    // Overload binary -.
    public static ThreeD operator -(ThreeD op1, ThreeD op2)
    {
        ThreeD result = new ThreeD();
        /* Notice the order of the operands. op1 is the left
         operand and op2 is the right. */
        result.x = op1.x - op2.x;
        result.y = op1.y - op2.y;
        result.z = op1.z - op2.z;
        return result;
    }
}
// Overload unary -.  
public static ThreeD operator -(ThreeD op)  
{
    ThreeD result = new ThreeD();
    result.x = -op.x;
    result.y = -op.y;
    result.z = -op.z;
    return result;
}

// Overload unary ++.
public static ThreeD operator ++(ThreeD op)  
{
    ThreeD result = new ThreeD();
    // Return the incremented result.
    result.x = op.x + 1;
    result.y = op.y + 1;
    result.z = op.z + 1;
    return result;
}

// Show X, Y, Z coordinates.
public void Show()  
{
    Console.WriteLine(x + ", " + y + ", " + z);
}
```csharp
class ThreeDDemo {
    static void Main() {
        ThreeD a = new ThreeD(1, 2, 3);
        ThreeD b = new ThreeD(10, 10, 10);
        ThreeD c = new ThreeD();
        Console.Write("Here is a: ");
        a.Show();
        Console.WriteLine();
        Console.Write("Here is b: ");
        b.Show();
        Console.WriteLine();
        c = a + b; // add a and b together
        Console.Write("Result of a + b: ");
        c.Show();
        Console.WriteLine();
        c = a + b + c; // add a, b, and c together
        Console.Write("Result of a + b + c: ");
        c.Show();
        Console.WriteLine();
        c = c - a; // subtract a
        Console.Write("Result of c - a: ");
        c.Show();
        Console.WriteLine();
    }
}
```
```c
    c = c - b; // subtract b
    Console.Write("Result of c - b: ");
    c.Show();
    Console.WriteLine();
    c = -a; // assign -a to c
    Console.Write("Result of -a: ");
    c.Show();
    Console.WriteLine();
    c = a++; // post-increment a
    Console.WriteLine("Given c = a++");
    Console.Write("c is ");
    c.Show();
    Console.Write("a is ");
    a.Show();
    // Reset a to 1, 2, 3
    a = new ThreeD(1, 2, 3);
    Console.Write("\nResetting a to ");
    a.Show();
    c = ++a; // pre-increment a
    Console.WriteLine("\nGiven c = ++a");
    Console.Write("c is ");
    c.Show();
    Console.Write("a is ");
    a.Show();
```
Figure 10.3 More overloaded operator
Handling Operations on C# Built-in Types

For any given class and operator, an operator method can, itself, be overloaded. One of the most common reasons for this is to allow operations between a class type and other types of data, such as a built-in type. For example, once again consider the ThreeD class. To this point, you have seen how to overload the + so that it adds the coordinates of one ThreeD object to another.

```csharp
public static ThreeD operator +(ThreeD op1, int op2)
{
    ThreeD result = new ThreeD();
    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;
    return result;
}
```

Notice that the second parameter is of type int. Thus, the preceding method allows an integer value to be added to each field of a ThreeD object. This is permissible because, as explained earlier, when overloading a binary operator, one of the operands must be of the same type as the class for which the operator is being overloaded. However, the other operand can be of any other type. Here is a version of ThreeD that has two overloaded + methods (Figure 10.4):
// Overload addition for ThreeD + ThreeD, and for ThreeD + int.
using System;
// A three-dimensional coordinate class.

class ThreeD {
    int x, y, z; // 3-D coordinates
    public ThreeD() { x = y = z = 0; }
    public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }
    // Overload binary + for ThreeD + ThreeD.
    public static ThreeD operator +(ThreeD op1, ThreeD op2)
    {
        ThreeD result = new ThreeD();
        /* This adds together the coordinates of the two points
           and returns the result. */
        result.x = op1.x + op2.x;
        result.y = op1.y + op2.y;
        result.z = op1.z + op2.z;
        return result;
    }
    // Overload binary + for object + int.
    public static ThreeD operator +(ThreeD op1, int op2)
    {
        ThreeD result = new ThreeD();
        result.x = op1.x + op2;
        result.y = op1.y + op2;
        result.z = op1.z + op2;
        return result;
    }
}
```csharp
// Show X, Y, Z coordinates.
public void Show()
{
    Console.WriteLine(x + "", " + y + ", " + z);
}

class ThreeDDemo {
    static void Main()
    {
        ThreeD a = new ThreeD(1, 2, 3);
        ThreeD b = new ThreeD(10, 10, 10);
        ThreeD c = new ThreeD();
        Console.Write("Here is a: ");
        a.Show();
        Console.WriteLine();
        Console.Write("Here is b: ");
        b.Show();
        Console.WriteLine();
        c = a + b; // ThreeD + ThreeD
        Console.Write("Result of a + b: ");
        c.Show();
        Console.WriteLine();
        c = b + 10; // ThreeD + int
        Console.Write("Result of b + 10: ");
        c.Show();
    }
}
As the output confirms, when the + is applied to two ThreeD objects, their coordinates are added together. When the + is applied to a ThreeD object and an integer, the coordinates are increased by the integer value.

While the overloading of + just shown certainly adds a useful capability to the ThreeD class, it does not quite finish the job. Here is why. The operator+(ThreeD, int) method allows statements like this:

```cpp
ob1 = ob2 + 10;
```

It does not, unfortunately, allow ones like this:

```cpp
ob1 = 10 + ob2;
```
The reason is that the integer argument is the second argument, which is the right-hand operand, but the preceding statement puts the integer argument on the left. To allow both forms of statements, you will need to overload the + yet another time. This version must have its first parameter as type int and its second parameter as type ThreeD. One version of the operator+( ) method handles ThreeD + integer, and the other handles integer + ThreeD. Overloading the + (or any other binary operator) this way allows a built-in type to occur on the left or right side of the operator. Here is a version ThreeD that overloads the + operator as just described (Figure 10.5):
```java
public static ThreeD operator +(ThreeD op1, ThreeD op2)
{
    ThreeD result = new ThreeD();
    /* This adds together the coordinates of the two points
    and returns the result. */
    result.x = op1.x + op2.x;
    result.y = op1.y + op2.y;
    result.z = op1.z + op2.z;
    return result;
}
// Overload binary + for object + int.
public static ThreeD operator +(ThreeD op1, int op2)
{
    ThreeD result = new ThreeD();
    result.x = op1.x + op2;
    result.y = op1.y + op2;
    result.z = op1.z + op2;
    return result;
}
// Overload binary + for int + ThreeD.
public static ThreeD operator +(int op1, ThreeD op2)
{
    ThreeD result = new ThreeD();
    result.x = op2.x + op1;
    result.y = op2.y + op1;
    result.z = op2.z + op1;
    return result;
}
```
```csharp
class ThreeDDemo {
    static void Main() {
        ThreeD a = new ThreeD(1, 2, 3);
        ThreeD b = new ThreeD(10, 10, 10);
        ThreeD c = new ThreeD();
        Console.Write("Here is a: ");
        a.Show();
        Console.WriteLine();
        Console.Write("Here is b: ");
        b.Show();
        Console.WriteLine();
        c = a + b; // ThreeD + ThreeD
        Console.Write("Result of a + b: ");
        c.Show();
        Console.WriteLine();
        c = b + 10; // ThreeD + int
        Console.Write("Result of b + 10: ");
        c.Show();
        c = 15 + b; // int + ThreeD
        Console.Write("Result of 15 + b: ");
        c.Show();
    }
}
```
Overloading the Relational Operators

- The relational operators, such as `==` or `<`, can also be overloaded and the process is straightforward. Usually, an overloaded relational operator returns a true or false value. This is in keeping with the normal usage of these operators and allows the overloaded relational operators to be used in conditional expressions. If you return a different type result, then you are greatly restricting the operator’s utility.

- Here is a version of the ThreeD class that overloads the `<` and `>` operators. In this example, these operators compare ThreeD objects based on their distance from the origin. One object is greater than another if its distance from the origin is greater. One object is less than another if its distance from the origin is less than the other. Given two points, such an implementation could be used to determine which point lies on the larger sphere. If neither operator returns true, then the two points lie on the same sphere. Of course, other ordering schemes are possible (Figure 10.6).
// Overload < and >.
using System;
// A three-dimensional coordinate class.

class ThreeD {
    int x, y, z; // 3-D coordinates
    public ThreeD() { x = y = z = 0; }
    public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }
    // Overload <.
    public static bool operator <(ThreeD op1, ThreeD op2)
    {
        if(Math.Sqrt(op1.x * op1.x + op1.y * op1.y + op1.z * op1.z) <
            Math.Sqrt(op2.x * op2.x + op2.y * op2.y + op2.z * op2.z))
            return true;
        else
            return false;
    }
    // Overload >.
    public static bool operator >(ThreeD op1, ThreeD op2)
    {
        if(Math.Sqrt(op1.x * op1.x + op1.y * op1.y + op1.z * op1.z) >
            Math.Sqrt(op2.x * op2.x + op2.y * op2.y + op2.z * op2.z))
            return true;
        else
            return false;
    }
    // Show X, Y, Z coordinates.
    public void Show()
    {
        Console.WriteLine(x + ", " + y + ", " + z);
    }
}
```csharp
class ThreeDDemo {
    static void Main() {
        ThreeD a = new ThreeD(5, 6, 7);
        ThreeD b = new ThreeD(10, 10, 10);
        ThreeD c = new ThreeD(1, 2, 3);
        ThreeD d = new ThreeD(6, 7, 5);
        Console.WriteLine("Here is a: ");
        a.Show();
        Console.WriteLine("Here is b: ");
        b.Show();
        Console.WriteLine("Here is c: ");
        c.Show();
        Console.WriteLine("Here is d: ");
        d.Show();
        Console.WriteLine();
        if (a > c) Console.WriteLine("a > c is true");
        if (a < c) Console.WriteLine("a < c is true");
        if (a > b) Console.WriteLine("a > b is true");
        if (a < b) Console.WriteLine("a < b is true");
        if (a > d) Console.WriteLine("a > d is true");
        else if (a < d) Console.WriteLine("a < d is true");
        else Console.WriteLine("a and d are same distance from origin");
    }
}
```
An important restriction applies to overloading the relational operators: You must overload them in pairs. For example, if you overload <, you must also overload >, and vice versa. The operator pairs are

```
==   !=
<    >
<=   >=
```

Properties

Another type of class member is the property. As a general rule, a property combines a field with the methods that access it. Properties are similar to indexers. A property consists of a name along with **get** and **set** accessors. The accessors are used to get and set the value of a variable. The key benefit of a property is that its name can be used in expressions and assignments like a normal variable, but in actuality the **get** and **set** accessors are automatically invoked. This is similar to the way that an indexer’s get and set accessors are automatically used.
The general form of a property is shown here:

```csharp
typename {
    get {
        // get accessor code
    }
    set {
        // set accessor code
    }
}
```

Here, type specifies the type of the property, such as int, and name is the name of the property. Once the property has been defined, any use of name results in a call to its appropriate accessor. The set accessor automatically receives a parameter called value that contains the value being assigned to the property.

Here is a simple example that defines a property called MyProp, which is used to access the field prop. In this case, the property allows only positive values to be assigned.

Let’s examine this program carefully. The program defines one private field, called prop, and a property called MyProp that manages access to prop. As explained, a property by itself does not define a storage location. Instead, most properties simply manage access to a field. Furthermore, because prop is private, it can be accessed only through MyProp (Figure 10.12).
// A simple property example.
using System;

class SimpProp
{
    int prop; // field being managed by MyProp
    public SimpProp() { prop = 0; }
    /* This is the property that supports access to
     * the private instance variable prop. It
     * allows only positive values. */
    public int MyProp
    {
        get
        {
            return prop;
        }
        set
        {
            if (value >= 0) prop = value;
        }
    }
}
// Demonstrate a property.

class PropertyDemo
{
    static void Main()
    {
        SimpProp ob = new SimpProp();
        Console.WriteLine("Original value of ob.MyProp: " + ob.MyProp);
        ob.MyProp = 100; // assign value
        Console.WriteLine("Value of ob.MyProp: " + ob.MyProp);
        // Can't assign negative value to prop.
        Console.WriteLine("Attempting to assign -10 to ob.MyProp");
        ob.MyProp = -10;
        Console.WriteLine("Value of ob.MyProp: " + ob.MyProp);
    }
}

Figure 10.12 Using property
The property **MyProp** is specified as public so it can be accessed by code outside of its class. This makes sense because it provides access to prop, which is private. The get accessor simply returns the value of `prop`. The set accessor sets the value of prop if and only if that value is positive. Thus, the **MyProp** property controls what values prop can have. This is the essence of why properties are important.

The type of property defined by MyProp is called a read-write property because it allows its underlying field to be read and written. It is possible, however, to create read-only and write-only properties. To create a read-only property, define only a **get** accessor. To define a write-only property, define only a **set** accessor.
Use Access Modifiers with Accessors

By default, the set and get accessors have the same accessibility as the indexer or property of which they are a part. For example, if the property is declared public, then by default the get and set accessors are also public. It is possible, however, to give set or get its own access modifier, such as private. In all cases, the access modifier for an accessor must be more restrictive than the access specification of its property or indexer.

There are a number of reasons why you may want to restrict the accessibility of an accessor. For example, you might want to let anyone obtain the value of a property, but allow only members of its class to set the property. To do this, declare the set accessor as private. For example, here is a property called MyProp that has its set accessor specified as private (Figure 10.14).
// Use an access modifier with an accessor.
using System;

class PropAccess {
    int prop; // field being managed by MyProp
    public PropAccess() { prop = 0; }
    /* This is the property that supports access to
       the private instance variable prop. It allows
       any code to obtain the value of prop, but only
       other class members can set the value of prop. */
    public int MyProp {
        get {
            return prop;
        }
        private set { // now, private
            prop = value;
        }
    }
    // This class member increments the value of MyProp.
    public void IncrProp()
    {
        MyProp++; // OK, in same class.
    }
}
// Demonstrate accessor access modifier.

class PropAccessDemo
{
    static void Main()
    {
        PropAccess ob = new PropAccess();
        Console.WriteLine("Original value of ob.MyProp: " + ob.MyProp);
        // ob.MyProp = 100; // can't access set
        ob.IncrProp();
        Console.WriteLine("Value of ob.MyProp after increment: "
                          + ob.MyProp);
    }
}

Figure 10.14 Private set accessor