Properties are class members that allow you to expose a characteristic of an object. Indexers are members that allow you to treat a class as though it were an array, using the same index syntax ([ ]) that is described in Chapter 13.

Properties and indexers both allow you to control access to the fields in your class and give you more control than exposing a field directly. As you’ll see when we get into the detail, properties and indexers are very flexible; in fact, they are so rich that they can sometimes blur the distinction between fields and methods.

Custom operators allow you to implement custom operations using standard notation such as + and ++ and allow you to control the implicit and explicit conversion from one type to another.

If you are coming to C# from a language that doesn’t have anything like these features, you might wonder why you should use them. I certainly felt that way when I started writing in C# after years of Java coding. The truth is that these features are largely syntactic sugar; they are nice-to-have features that reduce the code clutter that methods can cause. But they are nice-to-have. I am a firm convert, especially to properties and indexers. Give them a try in your programs, and you might become a convert too. Table 8-1 provides the summary for this chapter.

Table 8-1. Quick Problem/Solution Reference for Chapter 8

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
<th>Listings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use a property to mediate access to a field.</td>
<td>Use a field-backed property.</td>
<td>8-2</td>
</tr>
<tr>
<td>Use a property to validate values for a field.</td>
<td>Add validation code to the set accessor of a field-backed property.</td>
<td>8-2</td>
</tr>
<tr>
<td>Implement a property without defining a field.</td>
<td>Use an automatically implemented property.</td>
<td>8-3, 8-4</td>
</tr>
<tr>
<td>Create a read-only or write-only property.</td>
<td>Implement only the get or set accessor in a property.</td>
<td>8-5</td>
</tr>
<tr>
<td>Compute the value of a property on demand.</td>
<td>Add code to compute the result in the accessors.</td>
<td>8-6</td>
</tr>
<tr>
<td>Map the type of a property to a differently typed field.</td>
<td>Add translation code to the accessors.</td>
<td>8-7</td>
</tr>
</tbody>
</table>
Restrict the access to a property. | Apply an access modifier to the entire property or apply a more restrictive modifier to one of the accessors. | 8-8

Create a virtual, abstract, sealed or static property. | Apply the appropriate keyword to the property. | 8-9 through 8-12

Add array-like support for a class. | Add an indexer. | 8-13, 8-14, 8-16

Use an indexer that validates access to a field. | Add validating code statements in the get and set accessors. | 8-15

Implement custom unary and binary operators for custom types. | Use the operator keyword. | 8-17 through 8-20

Provide support for converting between different types. | Use the conversion keyword. | 8-21, 8-22

Creating a Property

Properties are very flexible. You can use this language feature to achieve several different results. In fact, properties are so flexible that you can end up creating something that might be mistaken for a method. In the following sections, I show you how to use the different models available for properties and how you can apply the same keywords available for other member types to get fine-grained control over how your properties function.

Creating a Field-Backed Property

The standard way of using a property is to use it as a mediator to a field, typically to ensure that other classes cannot set the field to a value that would create an exceptional state for your class. Listing 8-1 demonstrates the problem that this kind of property can help avoid.

Listing 8-1. The Illegal Field Value Problem

using System;

class Product {
    public int ItemsInStock;
    public double PricePerItem;

    public double GetTotalValueOfStock() {
        return ItemsInStock * PricePerItem;
    }
}