MODULE 5

CHAPTER 19

Enumerating collections

Enumerating the elements in a collection

Collection that implementsthe *System.Collections.IEnumerable* interface. The *IEnumerable* interface contains a single method called *GetEnumerator*:

IEnumerator GetEnumerator();

The *GetEnumerator* method should return an enumerator object that implements the *System. Collections.IEnumerator* interface. The enumerator object is used for stepping through (enumerating) the elements of the collection. The *IEnumerator* interface specifies the following property and methods:

object Current { get; }

bool MoveNext();

void Reset();

Think of an enumerator as a pointer indicating elements in a list. Initially, the pointer points *before* the first element.

- You call the *MoveNext* method to move the pointer down to the next (first) item in the list; the *MoveNext* method should return *true* if there actually is another item and *false* if there isn't.
- You use the *Current* property to access the item currently pointed to.
- You use the *Reset* method to return the pointer back to *before* the first item in the list.

By creating an enumerator by using the *GetEnumerator* method of a collection and repeatedly calling the *MoveNext* method and retrieving the value of the *Current* property by using the enumerator, you can move forward through the elements of a collection one item at a time.

This is exactly what the *foreach* statement does. So, if you want to create your own enumerable collection class, you must implement the *IEnumerable* interface in your collection class and also provide an implementation of the *IEnumerator* interface to be returned by the *GetEnumerator* method of the collection class.

Microsoft .NET Framework class library also provides the generic *IEnumerator<T>* interface, which has a *Current* property that returns a *T*, instead. Likewise, there is also an *IEnumerable<T>* interface containing a *GetEnumerator* method that returns an *Enumerator<T>* object. Both of these interfaces are defined in the *System.Collections.Generic* namespace

Implementing an enumerator by using an iterator

An *iterator* is a block of code that yields an ordered sequence of values. An iterator is not actually a member of an enumerable class; rather, it specifies the sequence that an enumerator should use for returning its values. In other words, an iterator is just a description of the enumeration sequence that the C# compiler can use for creating its own enumerator.

A simple iterator

The following *BasicCollection<T>* class illustrates the principles of implementing an iterator. The classuses a *List<T>* object for holding data and provides the *FillList* method for populating this list. Notice also that the *BasicCollection<T>* class implements the *IEnumerable<T>* interface. The *GetEnumerator* method is implemented by using an iterator:

```
using System;
using System.Collections.Generic;
using System.Collections;
class BasicCollection<T> : IEnumerable<T>
{
    private List<T> data = new List<T>();
    public void FillList(params T [] items)
    {
        foreach (var datum in items)
        {
            data.Add(datum);
        }
    }
    IEnumerator<T> IEnumerable<T>.GetEnumerator()
    {
        foreach (var datum in data)
```

```
                {
                        yield return datum;
                }
        }
        IEnumerator IEnumerable.GetEnumerator()
        {
                // Not implemented in this example
                throw new NotImplementedException();
        }
}
```

The *GetEnumerator* method appears to be straightforward, but it warrants closer examination. The first thing you should notice is that it doesn't appear to return an *IEnumerator<T>* type. Instead, it loops through the items in the *data* array, returning each item in turn.

The key point is the use of the *yield* keyword. The *yield* keyword indicates the value that should be returned by each iteration. If it helps, you can think of the *yield* statement as calling a temporary halt to the method, passing back a value to the caller. When the caller needs the next value, the *GetEnumerator* method continues at the point at which it left off, looping around and then yielding the next value. Eventually, the data is exhausted, the loop finishes, and the *GetEnumerator* method terminates. At this point, the iteration is complete.

The code in the *GetEnumerator* method defines an *iterator*. The compiler uses this code to generate an implementation of the *IEnumerator<T>* class containing a *Current* method and a *MoveNext* method. This implementation exactly matches the functionality specified by the *GetEnumerator* method.
You can invoke the enumerator generated by the iterator in the usual manner, as shown in the following block of code.

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");
foreach (string word in bc)
{
        Console.WriteLine(word);
}
```

This code simply outputs the contents of the *bc* object in this order:

Twas, brillig, and, the, slithy, toves

If you want to provide alternative iteration mechanisms presenting the data in a different sequence, you can implement additional properties that implement the *IEnumerable* interface and that use an iterator for returning data. For example, the *Reverse* property of the *BasicCollection<T>* class, shown here, emits the data in the list in reverse order:

```
class BasicCollection<T> : IEnumerable<T>
{
    ...
    public IEnumerable<T> Reverse
    {
        get
        {
            for (int i = data.Count - 1; i >= 0; i--)
            {
                yield return data[i];
            }
        }

    }

}
```

You can invoke this property as follows:

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");
foreach (string word in bc.Reverse)
{
    Console.WriteLine(word);
}
```

This code outputs the contents of the *bc* object in reverse order:

toves, slithy, the, and, brillig, Twas

CHAPTER 20

Decoupling application logic and handling events

Understanding delegates

A delegate is a reference to a method. It is a very simple concept with extraordinarily powerful implications. A *delegate* is an object that refers to a method. You can assign a reference to a method to a delegate in much the same way that you can assign an *int* value to an *int* variable.

The next example creates a delegate named *performCalculationDelegate* that references the *performCalculation* method of the *Processor* object.

```
Processor p = new Processor();
delegate void performCalculationDelegate();
performCalculationDelegate = p.performCalculation;
```

It is important to understand that the statement that assigns the method reference to the delegate does not run the method at this point;

- There are no parentheses after the method name, and you do not specify any parameters (if the method takes them).
- This is just an assignment statement. Having stored a reference to the *performCalculation* method of the *Processor* object in the delegate, the application can subsequently invoke the method through the delegate, like this:

```
performCalculationDelegate();
```

This looks like an ordinary method call; if you did not know otherwise, it looks like you might actually be running a method named *performCalculationDelegate*. However, the Common Language Runtime (CLR) knows that this is a delegate, so it retrieves the method that the delegate references and runs that, instead. Later on, you can change the method to which a delegate refers, so a statement that calls a delegate might actually run a different method each time it executes. Additionally, a delegate can reference more than one method at a time (think of it as a collection of method references), and when you invoke a delegate all of the methods to which it refers will run.

Examples of delegates in the .NET Framework class library

The Microsoft .NET Framework class library makes extensive use of delegates for many of its types, the *Find* method and the *Exists* method of the *List<T>* class. If you recall, these two methods search through a *List<T>* collection, either returning a matching item or testing for the existence of a matching item.

The following code should help to remind you of how you use the *Find* method:

```
struct Person
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
...
List<Person> personnel = new List<Person>()
{
    new Person() { ID = 1, Name = "John", Age = 47 },
    new Person() { ID = 2, Name = "Sid", Age = 28 },
    new Person() { ID = 3, Name = "Fred", Age = 34 },
    new Person() { ID = 4, Name = "Paul", Age = 22 },
};
...
// Find the member of the list that has an ID of 3
Person match = personnel.Find(p => p.ID == 3);
```

Other examples of methods exposed by the *List<T>* class that use delegates to perform their operations include *Average*, *Max*, *Min*, *Count*, and *Sum*. These methods take a *Func* delegate as the parameter. A *Func* delegate refers to a method that returns a value (a function). In the following examples, the *Average* method is used to calculate the average age of items in the personnel collection (the *Func<T>* delegate simply returns the value in the *Age* field of each item in the collection), the *Max* method is used to determine the item with the highest ID, and the *Count* method calculates how many items have an *Age* between 30 and 39 inclusive.

```
double averageAge = personnel.Average(p => p.Age);
Console.WriteLine("Average age is {0}", averageAge);
...
```

```
int id = personnel.Max(p => p.ID);
Console.WriteLine("Person with highest ID is {0}", id);
...
int thirties = personnel.Count(p => p.Age >= 30 && p.Age <= 39);
Console.WriteLine("Number of personnel in their thirties is {0}", thirties);
```

This code generates the following output:

```
Average age is 32.75
Person with highest ID is 4
Number of personnel in their thirties is 1
```

The automated factory scenario

Suppose you are writing the control systems for an automated factory. The factory contains a large number of different machines, each performing distinct tasks in the production of the articles manufactured by the factory—shaping and folding metal sheets, welding sheets together, painting sheets, and so on.

Each machine has its own unique computer-controlled process (and functions) for shutting down safely, as summarized here:

```
StopFolding(); // Folding and shaping machine
FinishWelding(); // Welding machine
PaintOff(); // Painting machine
```

Implementing the factory by using a delegate

Although the names of each method are different, they all have the same “shape”: they take no parameters, and they do not return a value. The general format of each method, therefore, is this:

```
void methodName();
```

This is where a delegate can be useful. You can use a delegate that matches this shape to refer to any of the machinery shutdown methods. You declare a delegate like this:

```
delegate void stopMachineryDelegate();
```

Note the following points:

- *Delegate* is a keyword.
- The return type (*void* in this example), a name for the delegate (*stopMachinery Delegate*), and any parameters (there are none in this case).

After you have declared the delegate, you can create an instance and make it refer to a matching method by using the += compound assignment operator. You can do this in the constructor of the controller class like this:

```
class Controller
{
    delegate void stopMachineryDelegate(); // the delegate type
    private stopMachineryDelegate stopMachinery; // an instance of the delegate
    ...
    public Controller()
    {
        this.stopMachinery += folder.StopFolding;
    }
    ...
}
```

It is safe to use the += operator on an uninitialized delegate. It will be initialized automatically. Alternatively, you can use the *new* keyword to initialize a delegate explicitly with a single specific method, like this:

```
this.stopMachinery = new stopMachineryDelegate(folder.StopFolding);
```

You can call the method by invoking the delegate, like this:

```
public void ShutDown()
{
    this.stopMachinery();
    ...
}
```

An important advantage of using a delegate is that it can refer to more than one method at the same time. You simply use the += operator to add methods to the delegate, like this:

```
public Controller()
{
    this.stopMachinery += folder.StopFolding;
    this.stopMachinery += welder.FinishWelding;
    this.stopMachinery += painter.PaintOff;
}
```

Invoking *this.stopMachinery()* in the *Shutdown* method of the *Controller* class automatically calls each of the methods in turn. The *Shutdown* method does not need to know how many machines there are or what the method names are.

You can remove a method from a delegate by using the –= compound assignment operator, as demonstrated here:

```
this.stopMachinery -= folder.StopFolding;
```

The current scheme adds the machine methods to the delegate in the *Controller* constructor. To make the *Controller* class totally independent of the various machines, you have several options:

- Make the *stopMachinery* delegate variable, public:

  ```
  public stopMachineryDelegate stopMachinery;
  ```

- Keep the *stopMachinery* delegate variable private, but create a read/write property to provide access to it:

  ```
  private delegate void stopMachineryDelegate();
  ...
  public stopMachineryDelegate StopMachinery
  {
          get
          {
                  return this.stopMachinery;
          }
          set
          {
                  this.stopMachinery = value;
          }
  }
  ```

- Provide complete encapsulation by implementing separate *Add* and *Remove* methods. The *Add* method takes a method as a parameter and adds it to the delegate, whereas the *Remove* method removes the specified method from the delegate (notice that you specify a method as a parameter by using a delegate type):

  ```
  public void Add(stopMachineryDelegate stopMethod)
  ```

```
{
        this.stopMachinery += stopMethod;
}
public void Remove(stopMachineryDelegate stopMethod)
{
        this.stopMachinery -= stopMethod;
}
```

An object-oriented purist would probably opt for the *Add*/*Remove* approach. However, the other approaches are viable alternatives that are frequently used, which is why they are shown here.

```
Controller control = new Controller();
FoldingMachine folder = new FoldingMachine();

WeldingMachine welder = new WeldingMachine();
PaintingMachine painter = new PaintingMachine();
...
control.Add(folder.StopFolding);
control.Add(welder.FinishWelding);
control.Add(painter.PaintOff);
...
control.ShutDown();
...
```

Lambda expressions and delegates

All the examples of adding a method to a delegate that you have seen so far use the method's name.
For example, returning to the automated factory scenario described earlier, you add the *StopFolding* method of the *folder* object to the *stopMachinery* delegate, like this:

```
this.stopMachinery += folder.StopFolding;
```

This approach is very useful if there is a convenient method that matches the signature of the delegate, but what if this is not the case? Suppose that the *StopFolding* method actually had the following signature:

```
void StopFolding(int shutDownTime); // Shut down in the specified number of seconds
```

This signature is now different from that of the *FinishWelding* and *PaintOff* methods, and therefore you cannot use the same delegate to handle all three methods. So, what do you do?

Creating a method adapter

One way around this problem is to create another method that calls *StopFolding* but that takes no parameters itself, like this:

```
void FinishFolding()
{
    folder.StopFolding(0); // Shut down immediately
}
```

You can then add the *FinishFolding* method to the *stopMachinery* delegate in place of the *StopFolding* method, using the same syntax as before:

```
this.stopMachinery += folder.FinishFolding;
```

When the *stopMachinery* delegate is invoked, it calls *FinishFolding*, which in turn calls the *Stop Folding* method, passing in the parameter of 0.

C# provides lambda expressions for situations such as this. Lambda expressions are described in Chapter 18, and there are more examples of them earlier in this chapter. In the factory scenario, you can use the following lambda expression:

```
this.stopMachinery += (() => folder.StopFolding(0));
```

When you invoke the *stopMachinery* delegate, it will run the code defined by the lambda expression, which will, in turn, call the *StopFolding* method with the appropriate parameter.

Enabling notifications by using events

The .NET Framework provides *events*, which you can use to define and trap significant actions and arrange for a delegate to be called to handle the situation. Many classes in the .NET Framework expose events.

Declaring an event

You declare an event similarly to how you declare a field. However, because events are intended to be used with delegates, the type of an event must be a delegate, and you must prefix the declaration with the *event* keyword. Use the following syntax to declare an event:

```
event delegateTypeName eventName
```

As an example, here's the *StopMachineryDelegate* delegate from the automated factory. It has been relocated to a new class called *TemperatureMonitor*, which provides an interface to the various electronic probes monitoring the temperature of the equipment (this is a more logical place for the event than the *Controller* class):

```
class TemperatureMonitor
{
        public delegate void StopMachineryDelegate();
        ...
}
```

You can define the *MachineOverheating* event, which will invoke the *stopMachineryDelegate*, like this:

```
class TemperatureMonitor
{
        public delegate void StopMachineryDelegate();
        public event StopMachineryDelegate MachineOverheating;
        ...
}
```

Subscribing to an event

Like delegates, events come ready-made with a += operator. You subscribe to an event by using this += operator. In the automated factory, the software controlling each machine can arrange for the shutdown methods to be called when the *MachineOverheating* event is raised, like this:

```
class TemperatureMonitor

{
        public delegate void StopMachineryDelegate();
        public event StopMachineryDelegate MachineOverheating;
        ...
}
...
TemperatureMonitor tempMonitor = new TemperatureMonitor();
...
tempMonitor.MachineOverheating += (() => { folder.StopFolding(0); });
tempMonitor.MachineOverheating += welder.FinishWelding;
tempMonitor.MachineOverheating += painter.PaintOff;
```

Notice that the syntax is the same as for adding a method to a delegate. You can even subscribe by using a lambda expression. When the *tempMonitor.MachineOverheating* event runs, it will call all the subscribing methods and shut down the machines.

Unsubscribing from an event

Knowing that you use the += operator to attach a delegate to an event, you can probably guess that you use the –= operator to detach a delegate from an event. Calling the –= operator removes the method from the event's internal delegate collection. This action is often referred to as *unsubscribing* from the event.

Raising an event

You can raise an event, just like a delegate, by calling it like a method. When you raise an event, all the attached delegates are called in sequence. For example, here's the *TemperatureMonitor* class with a private *Notify* method that raises the *MachineOverheating* event:

```
    class TemperatureMonitor
    {
        public delegate void StopMachineryDelegate();
        public event StopMachineryDelegate MachineOverheating;
        ...
        private void Notify()
        {
        if (this.MachineOverheating != null)
        {
            this.MachineOverheating();
        }
    }
...
}
```

The *null* check is necessary because an event field is implicitly *null* and only becomes non-*null* when a method subscribes to it by using the += operator. If you try to raise a *null* event, you will get a *NullReferenceException* exception. If the delegate defining the event expects any parameters, the appropriate arguments must be provided when you raise the event.

CHAPTER 21

Querying in-memory data by using query expressions

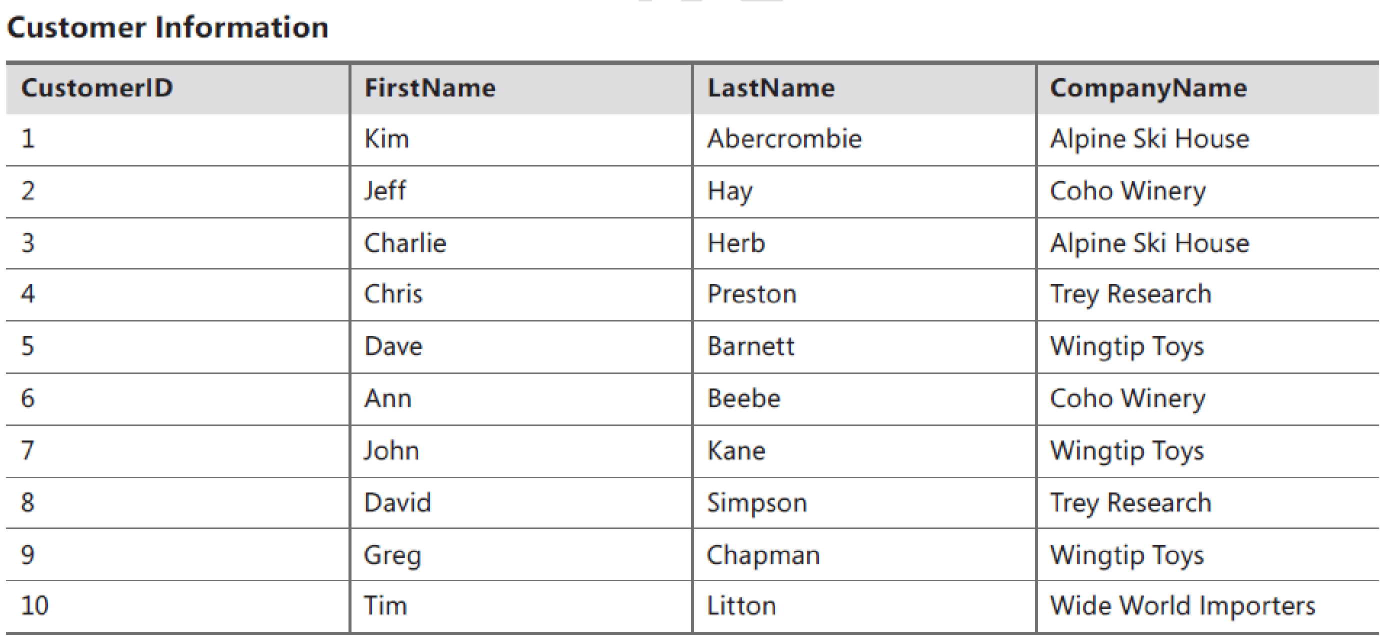
Language-Integrated Query

Set of features that abstract the mechanism that an application uses to query data from application code. LINQ provides syntax and semantics very reminiscent of SQL, and with many of the same advantages.

You can change the underlying structure of the data being queried without needing to change the code that actually performs the queries. You should be aware that although LINQ looks similar to SQL, it is far more flexible and can handle a wider variety of logical data structures.

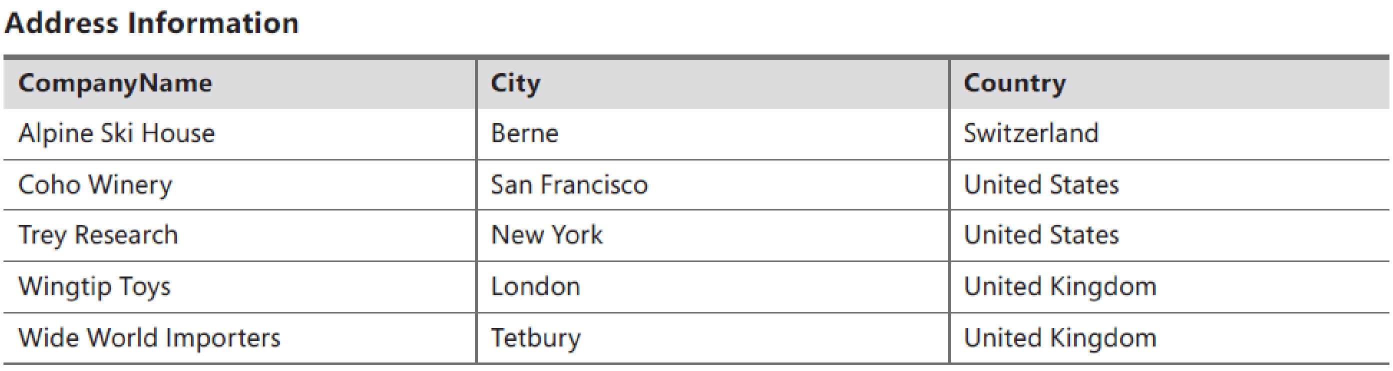
Using LINQ in a C# application

Perhaps the easiest way to explain how to use the C# features that support LINQ is to work through some simple examples based on the following sets of customer and address information:



Customer Information

CustomerID	FirstName	LastName	CompanyName
1	Kim	Abercrombie	Alpine Ski House
2	Jeff	Hay	Coho Winery
3	Charlie	Herb	Alpine Ski House
4	Chris	Preston	Trey Research
5	Dave	Barnett	Wingtip Toys
6	Ann	Beebe	Coho Winery
7	John	Kane	Wingtip Toys
8	David	Simpson	Trey Research
9	Greg	Chapman	Wingtip Toys
10	Tim	Litton	Wide World Importers



Address Information

CompanyName	City	Country
Alpine Ski House	Berne	Switzerland
Coho Winery	San Francisco	United States
Trey Research	New York	United States
Wingtip Toys	London	United Kingdom
Wide World Importers	Tetbury	United Kingdom

LINQ requires the data to be stored in a data structure that implements the *IEnumerable* or *IEnumerable<T>* interface. It does not matter what structure you use (an array, a *HashSet<T>*, a *Queue<T>*, or any of the other collection types, or even one that you define yourself) as long as it is enumerable. However, to keep things straightforward, the examples in this chapter assume that the customer and address information is held in the *customers* and *addresses* arrays shown in the following code example.

```
var customers = new[] {
new { CustomerID = 1, FirstName = "Kim", LastName = "Abercrombie",
CompanyName = "Alpine Ski House" },
new { CustomerID = 2, FirstName = "Jeff", LastName = "Hay",
CompanyName = "Coho Winery" },
new { CustomerID = 3, FirstName = "Charlie", LastName = "Herb",
CompanyName = "Alpine Ski House" },
new { CustomerID = 4, FirstName = "Chris", LastName = "Preston",
CompanyName = "Trey Research" },
new { CustomerID = 5, FirstName = "Dave", LastName = "Barnett",
CompanyName = "Wingtip Toys" },
new { CustomerID = 6, FirstName = "Ann", LastName = "Beebe",
CompanyName = "Coho Winery" },
new { CustomerID = 7, FirstName = "John", LastName = "Kane",
CompanyName = "Wingtip Toys" },
new { CustomerID = 8, FirstName = "David", LastName = "Simpson",
CompanyName = "Trey Research" },
new { CustomerID = 9, FirstName = "Greg", LastName = "Chapman",
CompanyName = "Wingtip Toys" },
new { CustomerID = 10, FirstName = "Tim", LastName = "Litton",
CompanyName = "Wide World Importers" }
};


var addresses = new[] {
new { CompanyName = "Alpine Ski House", City = "Berne",
Country = "Switzerland"},
new { CompanyName = "Coho Winery", City = "San Francisco",
Country = "United States"},
new { CompanyName = "Trey Research", City = "New York",
Country = "United States"},
new { CompanyName = "Wingtip Toys", City = "London",
Country = "United Kingdom"},
new { CompanyName = "Wide World Importers", City = "Tetbury",
Country = "United Kingdom"}
};
```

Selecting data

Suppose that you want to display a list consisting of the first name of each customer in the *customers* array. You can achieve this task with the following code:

```
IEnumerable<string> customerFirstNames = customers.Select(cust => cust.FirstName);
foreach (string name in customerFirstNames)
{
        Console.WriteLine(name);
}
```

The parameter to the *Select* method is actually another method that takes a row from the *customers* array and returns the selected data from that row. You can define your own custom method to perform this task, but the simplest mechanism is to use a lambda expression to define an anonymous method, as shown in the preceding example.

- The variable *cust* is the parameter passed in to the method. You can think of *cust* as an alias for each row in the *customers* array.

- The *Select* method does not actually retrieve the data at this time; it simply returns an enumerable object that will fetch the data identified by the *Select* method when you iterate over it later.
- The *Select* method is not actually a method of the *Array* type. It is an extension method of the *Enumerable* class.

The preceding example uses the *Select* method of the *customers* array to generate an *IEnumerable<string>* object named *customerFirstNames*. (It is of type *IEnumerable<string>* because the *Select* method returns an enumerable collection of customer first names, which are strings.) The *foreach* statement iterates through this collection of strings, printing out the first name of each customer in the following sequence:

```
Kim
Jeff
Charlie
Chris
Dave
Ann
John
David
Greg
Tim
```

Fetching more than one field of the data

Consider the example of fetching the first and last name of each customer
The important point to understand from the preceding paragraph is that the *Select* method returns an enumerable collection based on a single type. If you want the enumerator to return multiple items of data, such as the first and last name of each customer, you have at least two options:

- You can concatenate the first and last names together into a single string in the *Select* method, like this:

IEnumerable<string> customerNames =customers.Select(cust => String.Format("{0} {1}", cust.FirstName, cust.LastName));

- You can define a new type that wraps the first and last names, and use the *Select* method to construct instances of this type, like this:

```
class FullName
{
```

```
            public string FirstName{ get; set; }
            public string LastName{ get; set; }
    }
    ...
    IEnumerable<FullName> customerNames =
    customers.Select(cust => new FullName
    {
            FirstName = cust.FirstName,
            LastName = cust.LastName
    } );
```

The second option is arguably preferable, but if this is the only use that your application makes of the *Names* type, you might prefer to use an anonymous type instead of defining a new type specifically for a single operation, like this:

```
var customerNames = customers.Select(cust => new { FirstName = cust.FirstName, LastName =
cust.LastName } );
```

Notice the use of the *var* keyword here to define the type of the enumerable collection. The type of objects in the collection is anonymous, so you do not know the specific type for the objects in the collection.

Filtering data(using *Where( )* function)

For example, suppose you want to list the names of all companies in the *addresses* array that are located in the United States only. To do this, you can use the *Where* method, as follows:

```
IEnumerable<string> usCompanies =
addresses.Where(addr => String.Equals(addr.Country, "United States")).Select(usComp =>
usComp.CompanyName);
foreach (string name in usCompanies)
{
        Console.WriteLine(name);
}
```

Syntactically, the *Where* method is similar to *Select*. It expects a parameter that defines a method that filters the data according to whatever criteria you specify. This example makes use of another lambda expression. The variable *addr* is an alias for a row in the *addresses* array, and the lambda expression returns all rows where the *Country* field matches the string *"United States"*. The *Where* method returns an enumerable collection of rows containing every field from the original collection. The *Select* method is then applied to these rows to project only the *CompanyName* field from this enumerable collection to return another enumerable collection of *string* objects. (The variable *usComp* is an alias for the type of each row in the enumerable collection returned by the *Where* method.) The type of the result of this complete expression is therefore *IEnumerable<string>*. It is important to understand this sequence of operations—the *Where* method is applied first to filter the rows, followed by the *Select* method to specify the fields. The *foreach* statement that iterates through this collection displays the following companies:

```
Coho Winery
Trey Research
```

Ordering, grouping, and aggregating data

To retrieve data in a particular order, you can use the *OrderBy* method. Like the *Select* and *Where* methods, *OrderBy* expects a method as its argument. This method identifies the expressions that you want to use to sort the data. For example, you can display the name of each company in the *addresses* array in ascending order, like this:

```
IEnumerable<string> companyNames =addresses.OrderBy(addr => addr.CompanyName).Select(comp
=> comp.CompanyName);
foreach (string name in companyNames)
{
        Console.WriteLine(name);
}
```

This block of code displays the companies in the addresses table in alphabetical order.

```
Alpine Ski House
Coho Winery
Trey Research
Wide World Importers
Wingtip Toys
```

If you want to enumerate the data in descending order, you can use the *OrderByDescending* method, instead. If you want to order by more than one key value, you can use the *ThenBy* or *ThenByDescending* method after *OrderBy* or *OrderByDescending*. To group data according to common values in one or more fields, you can use the *GroupBy* method.

The following example shows how to group the companies in the *addresses* array by country:

```
var companiesGroupedByCountry =addresses.GroupBy(addrs => addrs.Country);
foreach (var companiesPerCountry in companiesGroupedByCountry)
{
        Console.WriteLine("Country: {0}\t{1}
        companies",companiesPerCountry.Key,companiesPerCountry.Count());
        foreach (var companies in companiesPerCountry)
        {
                Console.WriteLine("\t{0}", companies.CompanyName);
        }
}
```

By now, you should recognize the pattern. The *GroupBy* method expects a method that specifies the fields by which to group the data. There are some subtle differences between the *GroupBy* method and the other methods that you have seen so far, though.

You can use many of the summary methods such as *Count*, *Max*, and *Min* directly over the results of the *Select* method. If you want to know how many companies there are in the *addresses* array, you can use a block of code such as this:

```
        int numberOfCompanies = addresses.Select(addr => addr.CompanyName).Count();
        Console.WriteLine("Number of companies: {0}", numberOfCompanies);
```

Notice that the result of these methods is a single scalar value rather than an enumerable collection.

The output from the preceding block of code looks like this:

```
Number of companies: 5
```

If you wanted to find out how many different countries are mentioned in this table, you might be tempted to try this:

```
        int numberOfCountries = addresses.Select(addr => addr.Country).Count();
        Console.WriteLine("Number of countries: {0}", numberOfCountries);
```

The output looks like this:

```
Number of countries: 5
```

You can eliminate duplicates from the calculation by using the *Distinct* method, like this:

```
    int numberOfCountries =addresses.Select(addr => addr.Country).Distinct().Count();
    Console.WriteLine("Number of countries: {0}", numberOfCountries);
```

The *Console.WriteLine* statement now outputs the expected result:

Number of countries: 3

Joining data

Just like SQL, LINQ gives you the ability to join multiple sets of data together over one or more common key fields. The following example shows how to display the first and last names of each customer, together with the name of the country where the customer is located:

```
var companiesAndCustomers =
customers.Select(c => new { c.FirstName, c.LastName, c.CompanyName })
.Join(addresses, custs => custs.CompanyName, addrs => addrs.CompanyName,
(custs, addrs) => new {custs.FirstName, custs.LastName, addrs.Country });

foreach (var row in companiesAndCustomers)
{
    Console.WriteLine(row);
}
```

The customers' first and last names are available in the *customers* array, but the country for each company that customers work for is stored in the *addresses* array. The common key between the *customers* array and the *addresses* array is the company name. The *Select* method specifies the fields of interest in the *customers* array (*FirstName* and *LastName*), together with the field containing the common key (*CompanyName*). You use the *Join* method to join the data identified by the *Select* method with another enumerable collection. The parameters to the *Join* method are as follows:

- The enumerable collection with which to join
- A method that identifies the common key fields from the data identified by the *Select* method
- A method that identifies the common key fields on which to join the selected data
- A method that specifies the columns you require in the enumerable result set returned by the *Join* method

Using query operators

The designers of C# added query operators to the language with which you can employ LINQ features by using a syntax more akin to SQL.

As you saw in the examples shown earlier in this chapter, you can retrieve the first name for each customer, like this:

IEnumerable<string> customerFirstNames =customers.Select(cust => cust.FirstName);

from and *select* query operators

You can rephrase this statement by using the *from* and *select* query operators, like this:

var customerFirstNames = from cust in customers select cust.FirstName*;*

At compile time, the C# compiler resolves this expression into the corresponding *Select* method. The *from* operator defines an alias for the source collection, and the *select* operator specifies the fields to retrieve by using this alias. The result is an enumerable collection of customer first names. If you are familiar with SQL, notice that the *from* operator occurs before the *select* operator. to retrieve the first and last names for each customer, you can use the following statement.

var customerNames = from cust in customers select new { cust.FirstName, cust.LastName };

where operator to filter data

You use the *where* operator to filter data. The following example shows how to return the names of the companies based in the United States from the *addresses* array:

var usCompanies = from a in addresses

where String.Equals(a.Country, "United States")

select a.CompanyName;

orderby operator

To order data, use the *orderby* operator, like this:

var companyNames = from a in addresses

orderby a.CompanyName

select a.CompanyName;

group operator

You can group data by using the *group* operator in the following manner:

```
var companiesGroupedByCountry = from a in addresses
group a by a.Country;
```

You can invoke the summary functions such as *Count* over the collection returned by an enumerable collection, like this:

```
int numberOfCompanies = (from a in addresses
select a.CompanyName).Count();
```

Notice that you wrap the expression in parentheses.

If you want to ignore duplicate values, use the *Distinct* method:

```
int numberOfCountries = (from a in addresses
select a.Country).Distinct().Count();
```

join operator

You can use the *join* operator to combine two collections across a common key. The following example shows the query returning customers and addresses over the *CompanyName* column in each collection, this time rephrased using the *join* operator. You use the *on* clause with the *equals* operator to specify how the two collections are related.

```
var countriesAndCustomers = from a in addresses
join c in customers
on a.CompanyName equals c.CompanyName
select new { c.FirstName, c.LastName, a.Country };
```

LINQ and deferred evaluation

When you use LINQ to define an enumerable collection, either by using the LINQ extension methods or by using query operators, you should remember that the application does not actually build the collection at the time that the LINQ extension method is executed; the collection is enumerated only when you iterate over it. This means that the data in the original collection can change between executing a LINQ query and retrieving the data that the query identifies; you will always fetch the most up-to-date data. For example, the following query (which you saw earlier) defines an enumerable collection of companies in the United States:

```
var usCompanies = from a in addresses
where String.Equals(a.Country, "United States")
select a.CompanyName;
```

The data in the *addresses* array is not retrieved, and any conditions specified in the *Where* filter are not evaluated until you iterate through the *usCompanies* collection:

```
foreach (string name in usCompanies)
{
    Console.WriteLine(name);
}
```

If you modify the data in the *addresses* array between defining the *usCompanies* collection and iterating through the collection (for example, if you add a new company based in the United States), you will see this new data. This strategy is referred to as *deferred evaluation*. You can force evaluation of a LINQ query when it is defined and generate a static, cached collection. This collection is a copy of the original data and will not change if the data in the collection changes. LINQ provides the *ToList* method to build a static *List* object containing a cached copy of the data. You use it like this:

```
var usCompanies = from a in addresses.ToList()
where String.Equals(a.Country, "United States")
select a.CompanyName;
```

This time, the list of companies is fixed when you create the query. If you add more United States companies to the *addresses* array, you will not see them when you iterate through the *usCompanies* collection. LINQ also provides the *ToArray* method that stores the cached collection as an array.

CHAPTER 22

Operator overloading

Understanding operators

It is worth recapping some of the fundamental aspects of operators before delving into the details of how they work and how you can overload them. The following list summarizes these aspects:

- You use operators to combine operands together into expressions. Each operator has its own semantics, dependent on the type with which it works. For example, the + operator means “add” when you use it with numeric types, or it means “concatenate” when you use it with strings.
- Each operator has a *precedence*. For example, the * operator has a higher precedence than the + operator. This means that the expression *a + b * c* is the same as *a + (b * c).*
- Each operator also has an *associativity* to define whether the operator evaluates from left to right or from right to left. For example, the = operator is right-associative (it evaluates from right to left), so *a = b = c* is the same as *a = (b = c).*
- A *unary operator* is an operator that has just one operand. For example, the increment operator (++) is a unary operator.
- A *binary operator* is an operator that has two operands. For example, the multiplication operator (*) is a binary operator.

Operator constraints

- You cannot change the precedence and associativity of an operator. The precedence an associativity are based on the operator symbol (for example, +) and not on the type (for example, *int*) on which the operator symbol is being used. Hence, the expression *a + b * c* is always the same as *a + (b * c),* regardless of the types of *a, b,* and *c*.
- You cannot change the multiplicity (the number of operands) of an operator. For example, * (the symbol for multiplication) is a binary operator. If you declare a * operator for your own type, it must be a binary operator.
- You cannot invent new operator symbols. For example, you can’t create a new operator symbol, such as ** for raising one number to the power of another number. You’d have to create a method to do that.

- You can't change the meaning of operators when applied to built-in types. For example, the expression *1 + 2* has a predefined meaning, and you're not allowed to override this meaning. If you could do this, things would be too complicated.
- There are some operator symbols that you can't overload. For example, you can't overload the dot (.) operator, which indicates access to a class member. Again, if you could do this, it would lead to unnecessary complexity.

Overloaded operators

To define your own operator behavior, you must overload a selected operator. You use methodlike syntax with a return type and parameters, but the name of the method is the keyword **operator** together with the *operator* symbol you are declaring. For example, the following code shows a user-defined structure named *Hour* that defines a binary + operator to add together two instances of *Hour*:

```
struct Hour
{
        public Hour(int initialValue)
        {
                this.value = initialValue;
        }
        public static Hour operator +(Hour lhs, Hour rhs)
        {
                return new Hour(lhs.value + rhs.value);
        }
        ...
        private int value;
}
```

Notice the following:

- The operator is *public*. All operators *must* be public.
- The operator is *static*. All operators *must* be static. Operators are never polymorphic and cannot use the *virtual, abstract, override*, or *sealed* modifiers.
- A binary operator (such as the + operator shown earlier) has two explicit arguments, and a unary operator has one explicit argument. (C++ programmers should note that operators never have a hidden *this* parameter.)

When you use the + operator on two expressions of type *Hour*, the C# compiler automatically converts your code to a call to your *operator* + method. The C# compiler transforms this code

```
Hour Example(Hour a, Hour b)
{
    return a + b;
}
```

to this:

```
Hour Example(Hour a, Hour b)
{
    return Hour.operator +(a,b); // pseudocode
}
```

There is one final rule that you must follow when declaring an operator: at least one of the parameters must always be of the containing type. In the preceding *operator*+ example for the *Hour* class, one of the parameters, *a* or *b*, must be an *Hour* object. In this example, both parameters are *Hour* objects.

Creating symmetric operators

In the preceding section, you saw how to declare a binary + operator to add together two instances of type *Hour*. The *Hour* structure also has a constructor that creates an *Hour* from an *int*. This means that you can add together an *Hour* and an *int*; you just have to first use the *Hour* constructor to convert the *int* to an *Hour*, as in the following example:

```
Hour a = ...;
int b = ...;
Hour sum = a + new Hour(b);
```

This is certainly valid code, but it is not as clear or concise as adding together an *Hour* and an *int* directly, like this:

```
Hour a = ...;
int b = ...;
Hour sum = a + b;
```

To make the expression *(a + b)* valid, you must specify what it means to add together an *Hour* (*a*, on the left) and an *int* (*b*, on the right). In other words, you must declare a binary + operator whose first parameter is an *Hour* and whose second parameter is an *int*. The following code shows the recommended approach:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static Hour operator +(Hour lhs, Hour rhs)
    {
        return new Hour(lhs.value + rhs.value);
    }
    public static Hour operator +(Hour lhs, int rhs)
    {
        return lhs + new Hour(rhs);
    }
    ...
    private int value;
}
```

Notice that all the second version of the operator does is construct an *Hour* from its *int* argument and then call the first version.

This *operator+* declares how to add together an *Hour* as the left operand and an *int* as the right operand. It does not declare how to add together an *int* as the left operand and an *Hour* as the right operand:

```
int a = ...;
Hour b = ...;
Hour sum = a + b; // compile-time error
```

This is counterintuitive. If you can write the expression *a + b*, you expect to also be able to write *b + a*. Therefore, you should provide another overload of *operator+*:

```
struct Hour
{
        public Hour(int initialValue)
        {
            this.value = initialValue;
        }
        ...
        public static Hour operator +(Hour lhs, int rhs)
        {
            return lhs + new Hour(rhs);
        }
        public static Hour operator +(int lhs, Hour rhs)
        {
            return new Hour(lhs) + rhs;
        }
        ...
        private int value;
    }
```

Understanding compound assignment evaluation

A compound assignment operator (such as +=) is always evaluated in terms of its associated simple operator (such as +). In other words, the statement

a += b;

is automatically evaluated like this:

a = a + b;

In general, the expression *a @= b* (where @ represents any valid operator) is always evaluated as *a = a @ b*. If you have overloaded the appropriate simple operator, the overloaded version is automaticall called when you use its associated compound assignment operator, as is shown in the following example:

```
    Hour a = ...;
    int b = ...;
    a += a; // same as a = a + a
    a += b; // same as a = a + b
```

The first compound assignment expression *(a += a)* is valid because *a* is of type *Hour*, and the *Hour* type declares a binary *operator+* whose parameters are both *Hour*. Similarly, the second compound assignment expression *(a += b)* is also valid because *a* is of type *Hour* and *b* is of type *int*. The *Hour* type also declares a binary *operator+* whose first parameter is an *Hour* and whose second parameter is an *int*.

Be aware, however, that you cannot write the expression *b* += *a* because that's the same as

b = *b* + *a*. Although the addition is valid, the assignment is not, because there is no way to assign an *Hour* to the built-in *int* type.

Declaring increment and decrement operators

With C#, you can declare your own version of the increment (++) and decrement (– –) operators. The usual rules apply when declaring these operators: they must be public, they must be static, and they must be unary (they can take only a single parameter). Here is the increment operator for the *Hour* structure:

```
struct Hour
    {
    ...
    public static Hour operator ++(Hour arg)
    {
        arg.value++;
        return arg;
    }
    ...
    private int value;
}
```

The increment and decrement operators are unique in that they can be used in prefix and postfix forms. C# cleverly uses the same single operator for both the prefix and postfix versions. The result of a postfix expression is the value of the operand *before* the expression takes place. In other words, the compiler effectively converts the code

```
Hour now = new Hour(9);
Hour postfix = now++;
```

to this:

```
Hour now = new Hour(9);
Hour postfix = now;
now = Hour.operator ++(now); // pseudocode, not valid C#
```

The result of a prefix expression is the return value of the operator, so the C# compiler effectively transforms the code

```
Hour now = new Hour(9);
Hour prefix = ++now;
```

to this:

```
Hour now = new Hour(9);
now = Hour.operator ++(now); // pseudocode, not valid C#
Hour prefix = now;
```

This equivalence means that the return type of the increment and decrement operators must be the same as the parameter type.

Comparing operators in structures and classes

Be aware that the implementation of the increment operator in the *Hour* structure works only because *Hour* is a structure. If you change *Hour* into a class but leave the implementation of its increment operator unchanged, you will find that the postfix translation won't give the correct answer. If you remember that a class is a reference type and if you revisit the compiler translations explained earlier, you can see in the following example why the operators for the *Hour* class no longer function as expected:

```
Hour now = new Hour(9);
Hour postfix = now;
now = Hour.operator ++(now); // pseudocode, not valid C#
```

If *Hour* is a class, the assignment statement *postfix* = now makes the variable *postfix* refer to the same object as *now*. Updating *now* automatically updates *postfix*! If *Hour* is a structure, the assignment statement makes a copy of *now* in *postfix*, and any changes to *now* leave *postfix* unchanged, which is what you want. The correct implementation of the increment operator when *Hour* is a class is as follows:

```
class Hour
{
        public Hour(int initialValue)
        {
                this.value = initialValue;
        }
        ...
        public static Hour operator ++(Hour arg)
        {
                return new Hour(arg.value + 1);
        }
        ...
        private int value;
    }
```

Notice that *operator* ++ now creates a new object based on the data in the original. The data in the new object is incremented, but the data in the original is left unchanged. Although this works, the compiler translation of the increment operator results in a new object being created each time it is used. This can be expensive in terms of memory use and garbage collection overhead. Therefore, it is recommended that you limit operator overloads when you define types. This recommendation applies to all operators, not just to the increment operator.

Defining operator pairs

Some operators naturally come in pairs. For example, if you can compare two *Hour* values by using the *!=* operator, you would expect to be able to also compare two *Hour* values by using the == operator. The C# compiler enforces this very reasonable expectation by insisting that if you define either *operator* == or *operator !=*, you must define them both. This neither-or-both rule also applies to the < and > operators and the <= and >= operators. The C# compiler does not write any of these operator partners for you. You must write them all explicitly yourself, regardless of how obvious they might seem. Here are the == and *!=* operators for the *Hour* structure:

```
    struct Hour
    {
        public Hour(int initialValue)
        {
            this.value = initialValue;
        }
        ...
        public static bool operator ==(Hour lhs, Hour rhs)
        {
            return lhs.value == rhs.value;
        }
        public static bool operator !=(Hour lhs, Hour rhs)
        {
            return lhs.value != rhs.value;
        }
        ...
        private int value;
    }
```

The return type from these operators does not actually have to be Boolean. However, you would need to have a very good reason for using some other type, or these operators could become very confusing.

Understanding conversion operators

Sometimes, you need to convert an expression of one type to another. For example, the following method is declared with a single *double* parameter:

```
    class Example
    {
        public static void MyDoubleMethod(double parameter)
        {
            ...
        }
    }
```

You might reasonably expect that only values of type *double* could be used as arguments when calling *MyDoubleMethod*, but this is not so. The C# compiler also allows *MyDoubleMethod* to be called with an argument of some other type, but only if the value of the argument can be converted to a *double*. For example, if you provide an *int* argument, the compiler generates code that converts the value of the argument to a *double* when the method is called.

Providing built-in conversions

The built-in types have some built-in conversions. For example, as mentioned previously, an *int* can be implicitly converted to a *double*. An implicit conversion requires no special syntax and never throws an exception.

```
Example.MyDoubleMethod(42); // implicit int-to-double conversion
```

An implicit conversion is sometimes called a *widening conversion* because the result is *wider* than the original value—it contains at least as much information as the original value, and nothing is lost. In the case of *int* and *double*, the range of *double* is greater than that of *int*, and all *int* values have an equivalent *double* value. However, the converse is not true, and a *double* value cannot be implicitly converted to an *int*:

```
class Example
{
    public static void MyIntMethod(int parameter)
    {
        ...
    }
}
...
Example.MyIntMethod(42.0); // compile-time error
```

When you convert a *double* to an *int*, you run the risk of losing information, so the conversion will not be performed automatically. (Consider what would happen if the argument to *MyIntMethod* were 42.5: How should this be converted?) A *double* can be converted to an *int*, but the conversion requires an explicit notation (a cast):

```
Example.MyIntMethod((int)42.0);
```

An explicit conversion is sometimes called a *narrowing conversion* because the result is *narrower* than the original value (that is, it can contain less information) and can throw an *OverflowException* exception if the resulting value is out of the range of the target type. In C#, you can create conversion operators for your own user-defined types to control whether it is sensible to convert values to other types, and you can also specify whether these conversions are implicit or explicit.

Implementing user-defined conversion operators

The syntax for declaring a user-defined conversion operator has some similarities to that for declaring an overloaded operator, but also some important differences. Here’s a conversion operator that allows an *Hour* object to be implicitly converted to an *int*:

```
struct Hour
{
    ...
    public static implicit operator int (Hour from)
    {
        return from.value;
    }
    private int value;
}
```

A conversion operator must be *public* and it must also be *static*. The type from which you are converting is declared as the parameter (in this case, *Hour*), and the type to which you are converting is declared as the type name after the keyword *operator* (in this case, *int*). There is no return type specified before the keyword *operator*. When declaring your own conversion operators, you must specify whether they are implicit conversion operators or explicit conversion operators. You do this by using the *implicit* and *explicit* keywords. For example, the *Hour* to *int* conversion operator mentioned earlier is implicit, meaning that the C# compiler can use it without requiring a cast.

```
    class Example
    {
        public static void MyOtherMethod(int parameter) { ... }
        public static void Main()
        {
            Hour lunch = new Hour(12);
            Example.MyOtherMethod(lunch); // implicit Hour to int conversion
        }
    }
```

If the conversion operator had been declared *explicit*, the preceding example would not have compiled, because an explicit conversion operator requires a cast.

```
Example.MyOtherMethod((int)lunch); // explicit Hour to int conversion
```

When should you declare a conversion operator as explicit or implicit? If a conversion is always safe, does not run the risk of losing information, and cannot throw an exception, it can be defined as an *implicit* conversion. Otherwise, it should be declared as an *explicit* conversion. Converting from an *Hour* to an *int* is always safe—every *Hour* has a corresponding *int* value—so it makes sense for it to be implicit. An operator that converts a *string* to an *Hour* should be explicit because not all strings represent valid *Hours*.

Creating symmetric operators, revisited

Conversion operators provide you with an alternative way to resolve the problem of providing symmetric operators. For example, instead of providing three versions of *operator+* (*Hour* + *Hour*, *Hour* + *int*, and *int* + *Hour*) for the *Hour* structure, as shown earlier, you can provide a single version of *operator+* (that takes two *Hour* parameters) and an implicit *int* to *Hour* conversion, like this:

```
struct Hour
        {
        public Hour(int initialValue)
        {
                this.value = initialValue;
        }
        public static Hour operator +(Hour lhs, Hour rhs)
        {
                return new Hour(lhs.value + rhs.value);
        }
        public static implicit operator Hour (int from)
        {
                return new Hour (from);
        }
        ...
        private int value;
}
```

If you add an *Hour* to an *int* (in either order), the C# compiler automatically converts the *int* to an *Hour* and then calls *operator+* with two *Hour* arguments, as demonstrated here:

```
void Example(Hour a, int b)
{
        Hour eg1 = a + b; // b converted to an Hour
        Hour eg2 = b + a; // b converted to an Hour
}
```