Module 4:

CHAPTER 15

Implementing properties to access fields

Properties:

A *property* is a cross between a field and a method—it looks like a field but acts like a method. You access a property by using exactly the same syntax that you use to access a field. However, the compiler automatically translates this field-like syntax into calls to accessor methods (sometimes referred to as *property getters* and *property setters*).

The syntax for a property declaration looks like this:

```
AccessModifier Type PropertyName
{
    get
    {
        // read accessor code
    }
    set
    {
        // write accessor code
    }
}
```

A property can contain two blocks of code, starting with the *get* and *set* keywords. The *get* block contains statements that execute when the property is read, and the *set* block contains statements that run upon writing to the property. The type of the property specifies the type of data read and written by the *get* and *set* accessors.

Example:

```
struct ScreenPosition
{
    private int _x, _y;
    public ScreenPosition(int X, int Y)
    {
```

```
            this._x = X;
            this._y = Y;
        }
        public int X
        {
            get { return this._x; }
            set { this._x = value; }
        }
        public int Y
        {
            get { return this._y; }
            set { this._y = value; }
        }
    }
```

In this example, a private field directly implements each property, but this is only one way to implement a property. All that is required is that a get accessor returns a value of the specified type. Such a value can easily be calculated dynamically rather than being simply retrieved from stored data, in which case there would be no need for a physical field.

Using properties

When you use a property in an expression, you can use it in a read context (when you are retrieving its value) and in a write context (when you are modifying its value). The following example shows how to read values from the X and Y properties of the *ScreenPosition* structure:

```
ScreenPosition origin = new ScreenPosition(0, 0);
int xpos = origin.X; // calls origin.X.get
int ypos = origin.Y; // calls origin.Y.get
```

Notice that you access properties and fields by using identical syntax. When you use a property in a read context, the compiler automatically translates your field-like code into a call to the get accessor of that property. Similarly, if you use a property in a write context, the compiler automatically translates your field-like code into a call to the set accessor of that property.

```
origin.X = 40; // calls origin.X.set, with value set to 40
origin.Y = 100; // calls origin.Y.Set, with value set to 100
```

The values being assigned are passed in to the set accessors by using the value variable, as described in the preceding section. The runtime does this automatically.

It's also possible to use a property in a read/write context. In this case, both the get accessor and the set accessor are used. For example, the compiler automatically translates statements such as the following into calls to the get and set accessors:

```
origin.X += 10;
```

Read-only properties

You can declare a property that contains only a *get* accessor. In this case, you can use the property only in a read context. For example, here's the *X* property of the *ScreenPosition* structure declared as a read-only property:

```
struct ScreenPosition
{
        private int _x;
        ...
        public int X
        {
                get { return this._x; }
        }
}
```

The *X* property does not contain a *set* accessor; therefore, any attempt to use *X* in a write context will fail, as demonstrated in the following example:

```
origin.X = 140; // compile-time error
```

Write-only properties

Similarly, you can declare a property that contains only a *set* accessor. In this case, you can use the property only in a write context. For example, here's the *X* property of the *ScreenPosition* structure declared as a write-only property:

```
struct ScreenPosition
{
        private int _x;
        ...
        public int X
        {
                set { this._x = value; }
        }
}
```

The *X* property does not contain a *get* accessor; any attempt to use *X* in a read context will fail, as illustrated here:

```
Console.WriteLine(origin.X); // compile-time error
origin.X = 200; // compiles OK
origin.X += 10; // compile-time error
```

Property accessibility

You can specify the accessibility of a property (*public*, *private*, or *protected*) when you declare it. However, it is possible within the property declaration to override the property accessibility for the *get* and *set* accessors. For example, the version of the *ScreenPosition* structure shown in the code that follows defines the *set* accessors of the *X* and *Y* properties as *private*. (The *get* accessors are *public*, because the properties are *public*.)

```
struct ScreenPosition
{
        private int _x, _y;
        ...
```

```
        public int X
        {
            get { return this._x; }
            private set { this._x = value; }
        }
        public int Y
        {
            get { return this._y; }
            private set { this._y = value; }
        }
        ...
}
```

You must observe some rules when defining accessors with different accessibility from one another:

- You can change the accessibility of only one of the accessors when you define it. It wouldn't make much sense to define a property as *public* only to change the accessibility of both accessors to *private* anyway.
- The modifier must not specify an accessibility that is less restrictive than that of the property. For example, if the property is declared as *private*, you cannot specify the read accessor as *public*. (Instead, you would make the property *public* and make the write accessor *private*.)

Property restrictions

- You can assign a value through a property of a structure or class only after the structure or class has been initialized.

```
ScreenPosition location;
location.X = 40; // compile-time error, location not assigned
```

- You can't use a property as a *ref* or an *out* argument to a method (although you can use a writable field as a *ref* or an *out* argument). This makes sense because the property doesn't really point to a memory location; rather, it points to an accessor method, such as in the following example:

```
MyMethod(ref location.X); // compile-time error
```

- A property can contain at most one *get* accessor and one *set* accessor. A property cannot contain other methods, fields, or properties.
- The *get* and *set* accessors cannot take any parameters. The data being assigned is passed to the *set* accessor automatically by using the *value* variable.
- You can't declare *const* properties, such as is demonstrated here:

```
const int X { get { ... } set { ... } } // compile-time error
```

Declaring interface properties

Interfaces can define properties as well as methods. To do this, you specify the *get* or *set* keyword, or both, but replace the body of the *get* or *set* accessor with a semicolon, as shown here:

```
interface IScreenPosition
{
    int X { get; set; }
    int Y { get; set; }
}
```

Any class or structure that implements this interface must implement the *X* and *Y* properties with *get* and *set* accessor methods.

```
struct ScreenPosition : IScreenPosition
{
    ...
    public int X
    {
        get { ... }
        set { ... }
    }
    public int Y
    {
        get { ... }
        set { ... }
```

```
        }
        ...
}
```

If you implement the interface properties in a class, you can declare the property implementations as *virtual*, which enables derived classes to override the implementations.

```
class ScreenPosition : IScreenPosition
{
        ...
        public virtual int X
        {
                get { ... }
                set { ... }
        }
        public virtual int Y
        {
                get { ... }
                set { ... }
        }
        ...
}
```

You can also choose to implement a property by using the explicit interface implementation syntax. An explicit implementation of a property is nonpublic and nonvirtual (and cannot be overridden).

```
struct ScreenPosition : IScreenPosition
{
        ...
        int IScreenPosition.X
        {
```

```
        get { ... }
        set { ... }
    }
    int IScreenPosition.Y
    {
        get { ... }
        set { ... }
    }
    ...
}
```

Initializing objects by using properties

```
public class Triangle
{
    private int side1Length = 10;
    private int side2Length = 10;
    private int side3Length = 10;

    public int Side1Length
    {
        set { this.side1Length = value; }
    }
    public int Side2Length
    {
        set { this.side2Length = value; }
    }
    public int Side3Length
    {
        set { this.side3Length = value; }
    }
}
```

When you create an instance of a class, you can initialize it by specifying the names and values for any public properties that have *set* accessors. For example, you can create *Triangle* objects and initialize any combination of the three sides, like this:

```
Triangle tri1 = new Triangle { Side3Length = 15 };
Triangle tri2 = new Triangle { Side1Length = 15, Side3Length = 20 };
Triangle tri3 = new Triangle { Side2Length = 12, Side3Length = 17 };
Triangle tri4 = new Triangle { Side1Length = 9, Side2Length = 12, Side3Length = 15 };
```

This syntax is known as an *object initializer*. When you invoke an object initializer in this way, the C# compiler generates code that calls the default constructor and then calls the *set* accessor of each named property to initialize it with the value specified. You can specify object initializers in combination with nondefault constructors, as well. For example, if the *Triangle* class also provided a constructor that took a single string parameter describing the type of triangle, you could invoke this constructor and initialize the other properties, like this:

```
Triangle tri5 = new Triangle("Equilateral triangle") { Side1Length = 3,Side2Length = 3,Side3Length = 3 };
```

The important point to remember is that the constructor runs first and the properties are set afterward. Understanding this sequencing is important if the constructor sets fields in an object to specific values and the properties that you specify change these values.

CHAPTER 16
Using indexers

Indexer

Indexer is a smart array which encapsulates a set of values. Indexers are used for treating an object as an array.

Defining an Indexer

- To define the indexer, you use a notation that is a cross between a property and an array.
- You introduce the indexer with the ***this*** keyword
- Specify the type of the value returned by the indexer
- Specify the type of the value to use as the index into the indexer between square brackets.

Syntax:

```
<modifier> <return type> this [argument list]
{
        get
        {
                // your get block code
        }

        set
        {
                // your set block code
        }
}
```

Example:

```
class IndexerClass
    {
       private string[] names = new string[5];

       public string this[int i]
```

```
        {
            get
            {
                return names[i];
            }
            set
            {
                names[i] = value;
            }
        }
    }
static void Main(string[] args)
    {
        IndexerClass Team = new IndexerClass();
        Team[0] = "Rocky";
        Team[1] = "Teena";
        Team[2] = "Ana";
        Team[3] = "Victoria";
        Team[4] = "Yani";

        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine(Team[i]);
        }
        Console.ReadKey();
    }
```

C# provides a set of operators that you can use to access and manipulate the individual bits in an *int*. These operators are as follows:

The NOT (~) operator This is a unary operator that performs a bitwise complement. For example, if you take the 8-bit value *11001100* (*204* decimal) and apply the ~ operator to it, you obtain the result *00110011* (*51* decimal)—all the 1s in the original value become 0s, and all the 0s become 1s.

The left-shift (<<) operator This is a binary operator that performs a left shift. The expression *204 << 2* returns the value *48*. (In binary, *204* decimal is *11001100*, and left-shifting it by two places yields *00110000*, or *48* decimal.) The far-left bits are discarded, and zeros are introduced from the right. There is a corresponding right-shift operator, >>.

The OR (|) operator This is a binary operator that performs a bitwise OR operation, returning a value containing a 1 in each position in which either of the operands has a 1. For example, the expression *204 | 24* has the value *220* (*204* is *11001100*, *24* is *00011000*, and *220* is *11011100*).

The AND (&) operator This operator performs a bitwise AND operation. AND is similar to the bitwise OR operator, except that it returns a value containing a 1 in each position where both of the operands have a 1. So, *204 & 24* is *8* (*204* is *11001100*, *24* is *00011000*, and *8* is *00001000*).

The XOR (^) operator This operator performs a bitwise exclusive OR operation, returning a 1 in each bit where there is a 1 in one operand or the other but not both. (Two 1s yield a 0—this is the "exclusive" part of the operator.) So *204 ^ 24* is *212* (*11001100 ^ 00011000* is *11010100*).

You can use these operators together to determine the values of the individual bits in an *int*. As an example, the following expression uses the left-shift (<<) and bitwise AND (&) operators to determine whether the sixth bit from the right of the *byte* variable named *bits* is set to *0* or to *1*:

bits & (1 << 5)) != 0

By compound assignment operator &= to set the bit at position 6 to *0*:

bits &= ~(1 << 5)

Similarly, if you want to set the bit at position 6 to *1*, you can use a bitwise OR (|) operator. The following complicated expression is based on the compound assignment operator |=:

bits |= (1 << 5)

```
struct IntBits
{
    private int bits;
    public IntBits(int initialBitValue)
    {
        bits = initialBitValue;
    }
    public bool this [ int index ]
    {
        get
        {
            return (bits & (1 << index)) != 0;
        }
        set
        {
            if (value) // turn the bit on if value is true; otherwise, turn it off
                bits |= (1 << index);
            else
                bits &= ~(1 << index);
        }
    }
}
```

Notice the following points:

- An indexer is not a method; there are no parentheses containing a parameter, but there are square brackets that specify an index. This index is used to specify which element is being accessed.
- All indexers use the *this* keyword. A class or structure can define at most one indexer (although you can overload it and have several implementations), and it is always named *this*.
- Indexers contain *get* and *set* accessors just like properties. In this example, the *get* and *set* accessors contain the complicated bitwise expressions previously discussed.

- The index specified in the indexer declaration is populated with the index value specified when the indexer is called. The *get* and *set* accessor methods can read this argument to determine which element should be accessed.

After you have declared the indexer, you can use a variable of type *IntBits* instead of an *int* and apply the square bracket notation, as shown in the next example:

int adapted = 126; // 126 has the binary representation 01111110
IntBits bits = new IntBits(adapted);
bool peek = bits[6]; // retrieve bool at index 6; should be true (1)
bits[0] = true; // set the bit at index 0 to true (1)
bits[3] = false; // set the bit at index 3 to false (0)// the value in bits is now 01110111, or 119 in decimal

Understanding indexer accessors

When you read an indexer, the compiler automatically translates your array-like code into a call to the *get* accessor of that indexer. Consider the following example:

bool peek = bits[6];

This statement is converted to a call to the *get* accessor for *bits*, and the *index* argument is set to *6*. Similarly, if you write to an indexer, the compiler automatically translates your array-like code into a call to the *set* accessor of that indexer, setting the *index* argument to the value enclosed in the square brackets, such as illustrated here:

bits[3] = true;

This statement is converted to a call to the *set* accessor for *bits* where *index* is *3*. As with ordinary properties, the data you are writing to the indexer (in this case, *true*) is made available inside the *set* accessor by using the *value* keyword. The type of *value* is the same as the type of indexer itself (in this case, *bool*). It's also possible to use an indexer in a combined read/write context. In this case, the *get* and *set* accessors are both used. Look at the following statement, which uses the XOR operator (^) to invert the value of the bit at index 6 in the *bits* variable:

bits[6] ^= true;

This code is automatically translated into the following:

bits[6] = bits[6] ^ true;

This code works because the indexer declares both a *get* and a *set* accessor.

Comparing indexers and arrays

When you use an indexer, the syntax is deliberately array-like. However, there are some important differences between indexers and arrays:

- Indexers can use non-numeric subscripts, such as a string as shown in the following example, whereas arrays can use only integer subscripts.

  public int this [string name] { ... } // OK

- Indexers can be overloaded (just like methods), whereas arrays cannot.

  public Name this [PhoneNumber number] { ... }

  public PhoneNumber this [Name name] { ... }

- Indexers cannot be used as *ref* or *out* parameters, whereas array elements can.

  IntBits bits; // bits contains an indexer

  Method(ref bits[1]); // compile-time error

Indexers in interfaces

You can declare indexers in an interface. To do this, specify the *get* keyword, the *set* keyword, or both, but replace the body of the *get* or *set* accessor with a semicolon. Any class or structure that implements the interface must implement the *indexer* accessors declared in the interface, as demonstrated here:

```
interface IRawInt
{
    bool this [ int index ] { get; set; }
}

struct RawInt : IRawInt
{
```

```
        ...
        public bool this [ int index ]
        {
                get { ... }
                set { ... }
        }
        ...
}
```

If you implement the interface indexer in a class, you can declare the indexer implementations as *virtual*. This allows further derived classes to override the *get* and *set* accessors, such as in the following:

```
class RawInt : IRawInt
{
        ...
        public virtual bool this [ int index ]
        {
                get { ... }
                set { ... }
        }
        ...
}
```

You can also choose to implement an indexer by using the explicit interface implementation syntax. An explicit implementation of an indexer is nonpublic and nonvirtual (and so cannot be overridden), as shown in this example:

```
struct RawInt : IRawInt
{
        ...
        bool IRawInt.this [ int index ]
        {
                get { ... }
                set { ... }
```

```
        }
        ...
}
```

Difference between Indexers and Properties

Indexers	Properties
• Indexers are created with this keyword.	• Properties don't require this keyword.
• Indexers are identified by signature.	• Properties are identified by their names.
• Indexers are accessed using indexes.	• Properties are accessed by their names.
• Indexer are instance member, so can't be static.	• Properties can be static as well as instance members.
• A get accessor of an indexer has the same formal parameter list as the indexer.	• A get accessor of a property has no parameters.
• A set accessor of an indexer has the same formal parameter list as the indexer, in addition to the value parameter.	• A set accessor of a property contains the implicit value parameter.

CHAPTER 17

Introducing generics

Generic Class

C# provides generics to remove the need for casting, improve type safety, reduce the amount of boxing required, and make it easier to create generalized classes and methods. Generic classes and methods accept *type parameters*, which specify the types of objects on which they operate. In C#, you indicate that a class is a generic class by providing a type parameter in angle brackets, like this:

```
class Queue<T>
{
        ...
}
```

The *T* in this example acts as a placeholder for a real type at compile time. When you write code to instantiate a generic *Queue*, you provide the type that should be substituted for *T* (*Circle*, *Horse*, *int*, and so on). When you define the fields and methods in the class, you use this same placeholder to indicate the type of these items, like this:

```
class Queue<T>
{
    private const int DEFAULTQUEUESIZE = 100;
    private int head = 0, tail = 0;
    private int numElements = 0;private T[] data; // array is of type 'T' where 'T' is the type
    parameter
    public Queue()
    {
    this.data = new T[DEFAULTQUEUESIZE]; // use 'T' as the data type
    }
    public Queue(int size)
    {

        if (size > 0)
```

```
        {
            this.data = new T[size];
        }
        else
        {
            throw new ArgumentOutOfRangeException("size", "Must be greater than zero");
        }

    }
    public void Enqueue(T item) // use 'T' as the type of the method parameter
    {
        if (this.numElements == this.data.Length)
        {
        throw new Exception("Queue full");
        }
        this.data[this.head] = item;
        this.head++;
        this.head %= this.data.Length;
        this.numElements++;
    }
    public T Dequeue() // use 'T' as the type of the return value
    {
        if (this.numElements == 0)
        {
            throw new Exception("Queue empty");
        }
        T queueItem = this.data[this.tail]; // the data in the array is of type 'T'
        this.tail++;
        this.tail %= this.data.Length;
        this.numElements--;
        return queueItem;
    }
}
```

The type parameter *T* can be any legal C# identifier, although the lone character *T* is commonly used. It is replaced with the type you specify when you create a *Queue* object. The following examples create a *Queue* of *int*s, and a *Queue* of *Horse*s:

```
Queue<int> intQueue = new Queue<int>();
Queue<Horse> horseQueue = new Queue<Horse>();
```

Additionally, the compiler now has enough information to perform strict type-checking when you build the application. You no longer need to cast data when you call the *Dequeue* method, and the compiler can trap any type mismatch errors early:

```
intQueue.Enqueue(99);
int myInt = intQueue.Dequeue(); // no casting necessary
Horse myHorse = intQueue.Dequeue(); // compiler error:
// cannot implicitly convert type 'int' to 'Horse'
```

You should be aware that this substitution of *T* for a specified type is not simply a textual replacement mechanism. Instead, the compiler performs a complete semantic substitution so that you can specify any valid type for *T*. Here are more examples:

```
struct Person
{
...
}
...
Queue<int> intQueue = new Queue<int>();
Queue<Person> personQueue = new Queue<Person>();
```

The first example creates a queue of integers, whereas the second example creates a queue of *Person* values. The compiler also generates the versions of the *Enqueue* and *Dequeue* methods for each queue. For the *intQueue* queue, these methods look like this:

public void Enqueue(int item);
public int Dequeue();

For the *personQueue* queue, these methods look like this:

public void Enqueue(Person item);
public Person Dequeue();

Contrast these definitions with those of the object-based version of the *Queue* class shown in the preceding section. In the methods derived from the generic class, the *item* parameter to *Enqueue* is passed as a value type that does not require boxing. Similarly, the value returned by *Dequeue* is also a value type that does not need to be unboxed. A similar set of methods is generated for the other two queues.

The type parameter does not have to be a simple class or value type. For example, you can create a queue of queues of integers (if you should ever find it necessary), like this:

Queue<Queue<int>> queueQueue = new Queue<Queue<int>>();

A generic class can have multiple type parameters. For example, the generic *Dictionary* class defined in the *System.Collections.Generic* namespace in the .NET Framework class library expects two type parameters: one type for keys, and another for the values.

Generics vs. generalized classes

It is important to be aware that a generic class that uses type parameters is different from a *generalized* class designed to take parameters that can be cast to different types. For example, the object-based version of the *Queue* class shown earlier is a generalized class. There is a *single* implementation of this class, and its methods take *object* parameters and return *object* types. You can use this class with *int*s, *string*s, and many other types, but in each case, you are using instances of the same class and you have to cast the data you are using to and from the *object* type.

Compare this with the *Queue<T>* class. Each time you use this class with a type parameter (such as *Queue<int>* or *Queue<Horse>*), you cause the compiler to generate an entirely new class that happens to have functionality defined by the generic class. This means that *Queue<int>* is a completely different type from *Queue<Horse>*, but they both happen to have the same behavior. You can think of a generic class as

one that defines a template that is then used by the compiler to generate new typespecific classes on demand. The type-specific versions of a generic class (*Queue<int>*, *Queue<Horse>*, and so on) are referred to as *constructed types*, and you should treat them as distinctly different types.

Creating a generic method

- With a generic method, you can specify the types of the parameters and the return type by using a type parameter in a manner similar to that used when defining a generic class.
- In this way, you can define generalized methods that are type-safe and avoid the overhead of casting (and boxing, in some cases).
- Generic methods are frequently used in conjunction with generic classes; you need them for methods that take generic types as parameters or that has a return type that is a generic type.

- You define generic methods by using the same type parameter syntax that you use when creating generic classes. (You can also specify constraints.)

- For example, the generic *Swap<T>* method in the code that follows swaps the values in its parameters. Because this functionality is useful regardless of the type of data being swapped, it is helpful to define it as a generic method:

```
static void Swap<T>(ref T first, ref T second)
{
        T temp = first;
        first = second;
        second = temp;
}
```

You invoke the method by specifying the appropriate type for its type parameter. The following examples show how to invoke the *Swap<T>* method to swap over two *int*s and two *string*s:

```
int a = 1, b = 2;
Swap<int>(ref a, ref b);
...
```

```
string s1 = "Hello", s2 = "World";
Swap<string>(ref s1, ref s2);
```

Variance and generic interfaces

Covariant interfaces

Suppose that you defined the *IStoreWrapper<T>* and *IRetrieveWrapper<T>* interfaces, shown in the following example, in place of *IWrapper<T>* and implemented these interfaces in the *Wrapper<T>* class, like this:

```
interface IStoreWrapper<T>
{
        void SetData(T data);
}
interface IRetrieveWrapper<T>
{
        T GetData();
}
class Wrapper<T> : IStoreWrapper<T>, IRetrieveWrapper<T>
{
        private T storedData;
        void IStoreWrapper<T>.SetData(T data)
        {
                this.storedData = data;
        }
        T IRetrieveWrapper
        {
                return this.storedData;
        }
}
```

Functionally, the *Wrapper<T>* class is the same as before, except that you access the *SetData* and *GetData* methods through different interfaces.

```
Wrapper<string> stringWrapper = new Wrapper<string>();
IStoreWrapper<string> storedStringWrapper = stringWrapper;
storedStringWrapper.SetData("Hello");
IRetrieveWrapper<string> retrievedStringWrapper = stringWrapper;
Console.WriteLine("Stored value is {0}", retrievedStringWrapper.GetData());
```

Thus, is the following code legal?

```
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;
```

The quick answer is no, and it fails to compile with the same error as before. But, if you think about it, although the C# compiler has deemed that this statement is not type safe, the reasons for assuming this are no longer valid. The *IRetrieveWrapper<T>* interface only allows you to read the data held in the *Wrapper<T>* object by using the *GetData* method, and it does not provide any way to change the data. In situations such as this where the type parameter occurs only as the return value of the methods in a generic interface, you can inform the compiler that some implicit conversions are legal and that it does not have to enforce strict type-safety. You do this by specifying the *out* keyword when you declare the type parameter, like this:

```
interface IRetrieveWrapper<out T>
{
    T GetData();
}
```

This feature is called *covariance*. You can assign an *IRetrieveWrapper<A>* object to an *IRetrieveWrapper<B>* reference as long as there is a valid conversion from type *A* to type *B*, or type *A* derives from type *B*. The following code now compiles and runs as expected:

```
// string derives from object, so this is now legal
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;
```

You can specify the *out* qualifier with a type parameter only if the type parameter occurs as the return type of methods. If you use the type parameter to specify the type of any method parameters, the *out* qualifier is

illegal and your code will not compile. Also, covariance works only with reference types. This is because value types cannot form inheritance hierarchies. So, the following code will not compile because *int* is a value type:

```
Wrapper<int> intWrapper = new Wrapper<int>();
IStoreWrapper<int> storedIntWrapper = intWrapper; // this is legal
...
// the following statement is not legal - ints are not objects
IRetrieveWrapper<object> retrievedObjectWrapper = intWrapper;
```

Contravariant interfaces

Contravariance follows a similar principle to covariance except that it works in the opposite direction; it enables you to use a generic interface to reference an object of type *B* through a reference to type *A* as long as type *B* derives type *A*. This sounds complicated, so it is worth looking at an example from the .NET Framework class library.

The *System.Collections.Generic* namespace in the .NET Framework provides an interface called *IComparer*, which looks like this:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

A class that implements this interface has to define the *Compare* method, which is used to compare two objects of the type specified by the *T* type parameter. The *Compare* method is expected to return an integer value: zero if the parameters *x* and *y* have the same value, negative if *x* is less than *y*, and positive if *x* is greater than *y*. The following code shows an example that sorts objects according to their hash code. (The *GetHashCode* method is implemented by the *Object* class. It simply returns an integer value that identifies the object. All reference types inherit this method and can override it with their own implementations.)

```
class ObjectComparer : IComparer<Object>
{
    int IComparer<Object>.Compare(Object x, Object y)
    {
        int xHash = x.GetHashCode();
        int yHash = y.GetHashCode();
        if (xHash == yHash)
```

```
            return 0;
        if (xHash < yHash)
            return -1;
        return 1;
    }
}
```

You can create an *ObjectComparer* object and call the *Compare* method through the *IComparer<Object>* interface to compare two objects, like this:

```
Object x = ...;
Object y = ...;
ObjectComparer objectComparer = new ObjectComparer();
IComparer<Object> objectComparator = objectComparer;
int result = objectComparator.Compare(x, y);
```

What is more interesting is that you can reference this same object through a version of the *IComparer* interface that compares strings, like this:

```
IComparer<String> stringComparator = objectComparer;
```

At first glance, this statement seems to break every rule of type safety that you can imagine. However, if you think about what the *IComparer<T>* interface does, this approach makes sense. The purpose of the *Compare* method is to return a value based on a comparison between the parameters passed in. If you can compare *Objects*, you certainly should be able to compare *Strings*, which are just specialized types of *Objects*. After all, a *String* should be able to do anything that an *Object* can do— that is the purpose of inheritance.

This still sounds a little presumptive, however. How does the C# compiler know that you are not going to perform any type-specific operations in the code for the *Compare* method that might fail if you invoke the method through an interface based on a different type? If you revisit the definition of the *IComparer* interface, you can see the *in* qualifier prior to the type parameter:

```
public interface IComparer<in T>
{
        int Compare(T x, T y);
}
```

The *in* keyword tells the C# compiler that either you can pass the type *T* as the parameter type to methods or you can pass any type that derives from *T*. You cannot use *T* as the return type from any methods. Essentially, this makes it possible for you to reference an object either through a generic interface based on the object type or through a generic interface based on a type that derives from the object type. Basically, if a type *A* exposes some operations, properties, or fields, in that case if type *B* derives from type *A*, it must also expose the same operations (which might behave differently if they have been overridden), properties, and fields. Consequently, it should be safe to substitute an object of type *B* for an object of type *A*.

- **Covariance example** If the methods in a generic interface can return strings, they can also return objects. (All strings are objects.)
- **Contravariance example** If the methods in a generic interface can take object parameters, they can take string parameters. (If you can perform an operation by using an object, you can perform the same operation by using a string because all strings are objects.)

CHAPTER 18

Using collections

Collection classes

The Microsoft .NET Framework provides several classes that collect elements together such that an application can access them in specialized ways and they live in the *System.Collections.Generic* namespace.

Following are the some of the collection classes which are defined under *System.Collections.Generic* namespace.

Collection	Description
List<T>	A list of objects that can be accessed by index, like an array, but with additional methods to search the list and sort the contents of the list.
Queue<T>	A first-in, first-out data structure, with methods to add an item to one end of the queue, remove an item from the other end, and examine an item without removing it.
Stack<T>	A first-in, last-out data structure with methods to push an item onto the top of the stack, pop an item from the top of the stack, and examine the item at the top of the stack without removing it.
LinkedList<T>	A double-ended ordered list, optimized to support insertion and removal at either end. This collection can act like a queue or a stack, but it also supports random access like a list.
HashSet<T>	An unordered set of values that is optimized for fast retrieval of data. It provides set-oriented methods for determining whether the items it holds are a subset of those in another *HashSet<T>* object as well as computing the intersection and union of *HashSet<T>* objects.
Dictionary<TKey, TValue>	A collection of values that can be identified and retrieved by using keys rather than indexes.
SortedList<TKey, TValue>	A sorted list of key/value pairs. The keys must implement the *IComparable<T>* interface.

The *List<T>* collection class

- The generic *List<T>* class is the simplest of the collection classes.
- You can use it much like an array— you can reference an existing element in a *List<T>* collection by using ordinary array notation, with square brackets and the index of the element, although you cannot use array notation to add new elements.
- However, in general, the *List<T>* class provides more flexibility than arrays and is designed to overcome the following restrictions exhibited by arrays:
  - Resizing of array size
  - Removing of element

- Inserting an element

The *List<T>* collection class provides the following features

- No need to specify the capacity of a *List<T>* collection when you create it; it can grow and shrink as you add elements.
- Add an element to the end of a *List<T>* collection by using its *Add* method. You supply the element to be added. The *List<T>* collection resizes itself automatically.
- Remove a specified element from a *List<T>* collection by using the *Remove* method. The *List<T>* collection automatically reorders its elements and closes the gap. You can also remove an item at a specified position in a *List<T>* collection by using the *RemoveAt* method.
- Insert an element into the middle of a *List<T>* collection by using the *Insert* method. Again, the *List<T>* collection resizes itself.
- Easily sort the data in a *List<T>* object by calling the *Sort* method.

Here's an example that shows how you can create, manipulate, and iterate through the contents of a *List<int>* collection:

```
using System;
using System.Collections.Generic;
...
List<int> numbers = new List<int>();
foreach (int number in new int[12]{10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1})
{
        numbers.Add(number);
}
// Insert an element in the penultimate position in the list, and move the last item up
// The first parameter is the position; the second parameter is the value being inserted
numbers.Insert(numbers.Count-1, 99);
// Remove first element whose value is 7 (the 4th element, index 3)
numbers.Remove(7);
// Remove the element that's now the 7th element, index 6 (10)
numbers.RemoveAt(6);Console.WriteLine("Iterating using a for statement:");
for (int i = 0; i < numbers.Count; i++)
{
```

```
        int number = numbers[i]; // Note the use of array syntax
        Console.WriteLine(number);
    }
    Console.WriteLine("\nIterating using a foreach statement:");
    foreach (int number in numbers)
    {
        Console.WriteLine(number);
    }
```

Here is the output of this code:

Iterating using a for statement:

10

9

8

7

6

5

4

3

2

99

1

Iterating using a foreach statement:

10

9

8

7

6

5

4

3

2

99

1

The *LinkedList<T>* collection class

- The *LinkedList<T>* collection class implements a doubly linked list.
- Each item in the list holds the value for the item together with a reference to the next item in the list (the *Next* property) and the previous item (the *Previous* property).
- The item at the start of the list has the *Previous* property set to *null*, and the item at the end of the list has the *Next* property set to *null*.
- Unlike the *List<T>* class, *LinkedList<T>* does not support array notation for inserting or examining elements.

Methods defined in *LinkedList<T>* collection class

- **AddFirst(T)**:Adds a new node containing the specified value at the start of the LinkedList<T>.
- **AddLast(T)**:Adds a new node containing the specified value at the end of the LinkedList<T>.
- **AddAfter(LinkedListNode<T>, T)**:Adds a new node containing the specified value after the specified existing node in the LinkedList<T>.
- **AddBefore(LinkedListNode<T>, T)**:Adds a new node containing the specified value before the specified existing node in the LinkedList<T>.
- **Remove(T)**:Removes the first occurrence of the specified value from the LinkedList<T>.
- **RemoveFirst()**:Removes the node at the start of the LinkedList<T>.
- **RemoveLast()**:Removes the node at the end of the LinkedList<T>.

You can find the first item in a LinkedList<T> collection by querying the First property, whereas the Last property returns a reference to the final item in the list.

```
using System;
using System.Collections.Generic;
...
LinkedList<int> numbers = new LinkedList<int>();
// Fill the List<int> by using the AddFirst method
foreach (int number in new int[] { 10, 8, 6, 4, 2 })
{
        numbers.AddFirst(number);
}
```

```
// Iterate using a for statement
Console.WriteLine("Iterating using a for statement:");
for (LinkedListNode<int> node = numbers.First; node != null; node = node.Next)
{
        int number = node.Value;
        Console.WriteLine(number);
}
// Iterate using a foreach statement
Console.WriteLine("\nIterating using a foreach statement:");
foreach (int number in numbers)
{
        Console.WriteLine(number);
}
// Iterate backwards
Console.WriteLine("\nIterating list in reverse order:");
for (LinkedListNode<int> node = numbers.Last; node != null; node = node.Previous)
{
        int number = node.Value;
        Console.WriteLine(number);
}
```

Here is the output generated by this code:

```
Iterating using a for statement:
2
4
6
8
10
Iterating using a foreach statement:
2
4
6
8
10
```

Iterating list in reverse order:

10

8

6

4

2

The *Queue<T>* collection class

The *Queue<T>* class implements a first-in, first-out mechanism. An element is inserted into the queue at the back (the *Enqueue* operation) and is removed from the queue at the front (the *Dequeue* operation).

The following code is an example showing a *Queue<int>* collection and its common operations:

```
using System;
using System.Collections.Generic;
...
Queue<int> numbers = new Queue<int>();
// fill the queue
Console.WriteLine("Populating the queue:");
foreach (int number in new int[4]{9, 3, 7, 2})
{
        numbers.Enqueue(number);
        Console.WriteLine("{0} has joined the queue", number);
}
// iterate through the queue
Console.WriteLine("\nThe queue contains the following items:");
foreach (int number in numbers)
{
        Console.WriteLine(number);
}
// empty the queue
Console.WriteLine("\nDraining the queue:");
while (numbers.Count > 0)
{
```

```
        int number = numbers.Dequeue();
        Console.WriteLine("{0} has left the queue", number);
    }
```

Here is the output from this code:

Populating the queue:
9 has joined the queue
3 has joined the queue
7 has joined the queue
2 has joined the queue

The queue contains the following items:
9
3
7
2
Draining the queue:
9 has left the queue
3 has left the queue
7 has left the queue
2 has left the queue

The *Stack<T>* collection class

The *Stack<T>* class implements a last-in, first-out mechanism. An element joins the stack at the top (the push operation) and leaves the stack at the top (the pop operation).

```
using System;
using System.Collections.Generic;
...
Stack<int> numbers = new Stack<int>();
// fill the stack
Console.WriteLine("Pushing items onto the stack:");
foreach (int number in new int[4]{9, 3, 7, 2})
```

```
{
        numbers.Push(number);
        Console.WriteLine("{0} has been pushed on the stack", number);
}
// iterate through the stack
Console.WriteLine("\nThe stack now contains:");
foreach (int number in numbers)
{
        Console.WriteLine(number);
}
// empty the stack
Console.WriteLine("\nPopping items from the stack:");
while (numbers.Count > 0)
{
        int number = numbers.Pop();
        Console.WriteLine("{0} has been popped off the stack", number);
}
```

Here is the output from this program:

```
Pushing items onto the stack:
9 has been pushed on the stack
3 has been pushed on the stack
7 has been pushed on the stack
2 has been pushed on the stack
The stack now contains:
2
7
3
9
Popping items from the stack:
2 has been popped off the stack
7 has been popped off the stack
3 has been popped off the stack
9 has been popped off the stack
```

The Dictionary<TKey, TValue> collection class

- The *Dictionary<TKey, TValue>* class implements this functionality by internally maintaining two arrays, one for the *keys* from which you're mapping and one for the *values* to which you're mapping.
- When you insert a key/value pair into a *Dictionary<TKey, TValue>* collection, it automatically tracks which key belongs to which value and makes it possible for you to retrieve the value that is associated with a specified key quickly and easily.

The design of the *Dictionary<TKey, TValue>* class has some important consequences:

- A *Dictionary<TKey, TValue>* collection cannot contain duplicate keys. If you call the *Add* method to add a key that is already present in the keys array, you'll get an exception.

- Internally, a *Dictionary<TKey, TValue>* collection is a sparse data structure that operates most efficiently when it has plenty of memory with which to work. The size of a *Dictionary<TKey, TValue>* collection in memory can grow quite quickly as you insert more elements.

- When you use a *foreach* statement to iterate through a *Dictionary<TKey, TValue>* collection, you get back a *KeyValuePair<TKey, TValue>* item. This is a structure that contains a copy of the key and value elements of an item in the *Dictionary<TKey, TValue>* collection, and you can access each element through the *Key* property and the *Value* properties. These elements are read-only;

Here is an example that associates the ages of members of my family with their names and then prints the information:

```
using System;
using System.Collections.Generic;
...
Dictionary<string, int> ages = new Dictionary<string, int>();
// fill the Dictionary
ages.Add("John", 47); // using the Add method
ages.Add("Diana", 46);
ages["James"] = 20; // using array notation
ages["Francesca"] = 18;
// iterate using a foreach statement
```

```
// the iterator generates a KeyValuePair item
Console.WriteLine("The Dictionary contains:");
foreach (KeyValuePair<string, int> element in ages)
{
        string name = element.Key;
        int age = element.Value;
        Console.WriteLine("Name: {0}, Age: {1}", name, age);
}
```

Here is the output from this program:

The Dictionary contains:

Name: John, Age: 47

Name: Diana, Age: 46

Name: James, Age: 20

Name: Francesca, Age: 18

The SortedList<TKey, TValue> collection class

- The *SortedList<TKey, TValue>* class is very similar to the *Dictionary<TKey, TValue>* class in that you can use it to associate keys with values.
- The main difference is that the keys array is always sorted. When you insert a key/value pair into a *SortedList<TKey, TValue>* collection, the key is inserted into the keys array at the correct index to keep the keys array sorted. The value is then inserted into the values array at the same index.
- The *SortedList<TKey, TValue>* class automatically ensures that keys and values maintain synchronization, even when you add and remove elements.
- This means that you can insert key/value pairs into a *SortedList<TKey, TValue>* in any sequence; they are always sorted based on the value of the keys.
- Like the *Dictionary<TKey, TValue>* class, a *SortedList<TKey, TValue>* collection cannot contain duplicate keys.

Here is the example that associates the ages of members of my family with their names and then prints the information

```
using System;
using System.Collections.Generic;
...
SortedList<string, int> ages = new SortedList<string, int>();
// fill the SortedList
ages.Add("John", 47); // using the Add method
ages.Add("Diana", 46);
ages["James"] = 20; // using array notation
ages["Francesca"] = 18;
// iterate using a foreach statement
// the iterator generates a KeyValuePair item
Console.WriteLine("The SortedList contains:");
foreach (KeyValuePair<string, int> element in ages)
{
        string name = element.Key;
        int age = element.Value;
        Console.WriteLine("Name: {0}, Age: {1}", name, age);
}
```

The output from this program is sorted alphabetically by the names of my family members:

The SortedList contains:
Name: Diana, Age: 46
Name: Francesca, Age: 18
Name: James, Age: 20
Name: John, Age: 47

The *HashSet<T>* collection class

- The *HashSet<T>* class is optimized for performing set operations such as determining set membership and generating the union and intersect of sets.
- You insert items into a *HashSet<T>* collection by using the *Add* method, and you delete items by using the *Remove* method. However, the real power of the *HashSet<T>* class is provided by the *IntersectWith*, *UnionWith*, and *ExceptWith* methods.

- These methods modify a *HashSet<T>* collection to generate a new set that either intersects with, has a union with, or does not contain the items in a specified *HashSet<T>* collection.
- You can also determine whether the data in one *HashSet<T>* collection is a superset or subset of another by using the *IsSubsetOf*, *IsSupersetOf*, *IsProperSubsetOf*, and *IsProperSupersetOf* methods.

The following example shows how to populate a *HashSet<T>* collection and illustrates the use of the *IntersectWith* method to find data that overlaps two sets:

```
using System;
using System.Collections.Generic;
...
HashSet<string> employees = new HashSet<string>(new string[] {"Fred","Bert","Harry","John"});
HashSet<string> customers = new HashSet<string>(new string[] {"John","Sid","Harry","Diana"});
employees.Add("James");
customers.Add("Francesca");
Console.WriteLine("Employees:");
foreach (string name in employees)
{
        Console.WriteLine(name);
}
Console.WriteLine("\nCustomers:");
foreach (string name in customers)
{
        Console.WriteLine(name);
}
Console.WriteLine("\nCustomers who are also employees:");
customers.IntersectWith(employees);
foreach (string name in customers)
{
        Console.WriteLine(name);
}
```

This code generates the following output:

Employees:

Fred

Bert

Harry

John

James

Customers:

John

Sid

Harry

Diana

Francesca

Customers who are also employees:

John

Harry

Using collection initializers

The examples in the preceding subsections have shown you how to add individual elements to a collection by using the method most appropriate to that collection (*Add* for a *List<T>* collection, *Enqueue* for a *Queue<T>* collection, *Push* for a *Stack<T>* collection, and so on). You can also initialize *some* collection types when you declare them, using a syntax similar to that supported by arrays. For example, the following statement creates and initializes the *numbers List<int>* object shown earlier, demonstrating an alternate technique to repeatedly calling the *Add* method:

List<int> numbers = new List<int>(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};

Internally, the C# compiler actually converts this initialization to a series of calls to the *Add* method. For more complex collections that take key/value pairs, such as the *Dictionary<TKey, TValue>* class, you can specify each key/value pair as an anonymous type in the initializer list, like this:

Dictionary<string, int> ages =new Dictionary<string, int>(){{"John", 44}, {"Diana", 45}, {"James", 17}, {"Francesca",15}};

The first item in each pair is the key, and the second is the value.

The *Find* methods, predicates, and lambda expressions

- *Find()* method supports non-keyed classes(*List<T> and LinkedList<T>)* to find the item.
- **Predicates** are passed as an argument to provide the search criteria.
- If the criteria matches the predicate returns the Boolean value.
- The *Find()* method returns the corresponding item.
- *List<T> and LinkedList<T> also supports FindLast() which returns the last occurrence of matching object.*
- *List<t> supports FindAll() method –returns all matching object*

The easiest way to specify the predicate is to use a *lambda expression*. A lambda expression is an expression that returns a method. Typical method consists of four elements: a return type, a method name, a list of parameters, and a method body. A lambda expression contains two of these elements: a list of parameters and a method body. Lambda expressions do not define a method name, and the return type (if any) is inferred from the context in which the lambda expression is used. In the case of the *Find* method, the predicate processes each item in the collection in turn; the body of the predicate must examine the item and return true or false depending on whether it matches the search criteria. The example that follows shows the *Find* method (highlighted in bold) on a *List<Person>* collection, where *Person* is a struct. The *Find* method returns the first item in the list that has the *ID* property set to 3:

```
struct Person
{
        public int ID { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
}
...
// Create and populate the personnel list
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((Person p) => { return p.ID == 3; });
Console.WriteLine("ID: {0}\nName: {1}\nAge: {2}", match.ID, match.Name, match.Age);
```

Here is the output generated by this code:

ID: 3

Name: Fred

Age: 34

In the call to the *Find* method, the argument ***(Person p) => { return p.ID == 3; }*** is a lambda expression that actually does the work. It has the following syntactic items:

- A list of parameters enclosed in parentheses. As with a regular method, if the method you are defining (as in the preceding example) takes no parameters, you must still provide the parentheses. In the case of the *Find* method, the predicate is provided with each item from the collection in turn, and this item is passed as the parameter to the lambda expression.
- The => operator, which indicates to the C# compiler that this is a lambda expression.
- The body of the method. The example shown here is very simple, containing a single statement that returns a Boolean value indicating whether the item specified in the parameter matches the search criteria. However, a lambda expression can contain multiple statements, and you can format it in whatever way you feel is most readable. Just remember to add a semicolon after each statement, as you would in an ordinary method.

Strictly speaking, the body of a lambda expression can be a method body containing multiple statements or it can actually be a single expression. If the body of a lambda expression contains only a single expression, you can omit the braces and the semicolon (but you still need a semicolon to complete the entire statement). Additionally, if the expression takes a single parameter, you can omit the parentheses that surround the parameter. Finally, in many cases, you can actually omit the type of the parameters because the compiler can infer this information from the context from which the lambda expression is invoked. A simplified form of the *Find* statement shown previously looks like this (this statement is much easier to read and understand):

```
Person match = personnel.Find(p => p.ID == 3);
```

The forms of lambda expressions:

```
x => x * x // A simple expression that returns the square of its parameter
           // The type of parameter x is inferred from the context.

x => { return x * x ; } // Semantically the same as the preceding
                        // expression, but using a C# statement block as
                        // a body rather than a simple expression

(int x) => x / 2 // A simple expression that returns the value of the
                 // parameter divided by 2
                 // The type of parameter x is stated explicitly.

() => folder.StopFolding(0) // Calling a method
                            // The expression takes no parameters.
                            // The expression might or might not
                            // return a value.

(x, y) => { x++; return x / y; } // Multiple parameters; the compiler
                                 // infers the parameter types.
                                 // The parameter x is passed by value, so
                                 // the effect of the ++ operation is
                                 // local to the expression.

(ref int x, int y) => { x++; return x / y; } // Multiple parameters
                                             // with explicit types
                                             // Parameter x is passed by
                                             // reference, so the effect of
                                             // the ++ operation is permanent.
```

Features of lambda expressions

- If a lambda expression takes parameters, you specify them in the parentheses to the left of the "=>" *operator*.
- Lambda expressions can return values, but the return type must match that of the delegate to which they are being added.
- The body of a lambda expression can be a simple expression or a block of C# code.
- Variables defined in a lambda expression method go out of scope when the method finishes.
- A lambda expression can access and modify all variables outside the lambda expression that are in scope when the lambda expression is defined.

Comparing arrays and collections

Array	Collection Class
• An array instance has a fixed size and cannot grow or shrink.	• A collection can dynamically resize itself as required.
• An array can have more than one dimension.	• A collection is linear.
• You store and retrieve an item in an array by using an index.	• Not all collections support this notion.