Effective .NET programming using C# requires a thorough knowledge of the major features of the C# programming language. You have already been using some of these features without actually realizing it. Among the features that are intrinsic to the implementation of the Framework Class Library are indexers, interfaces, and enumerators. In this chapter we will take a look at the basics of each and explore examples of their use in your applications.

Indexers

We have already explored the power and use of properties in Chapter <ref>. An indexer is very similar to a property except that it is used to access an array. You probably remember that the power of properties is that they allow a controlled access to a class that allows us to abstract the internal workings of a class and the representation of its data. If we were to expose the internal data by making it public we would have no control over how it was manipulated. For example, illegal values might be stored. We would also be compelled to keep both the name and types of all such publicly exposed data unless we want to break the class. This latter point is central to one of the principles of object-oriented programming, encapsulation. That's another term for abstraction.

An indexer allows us to expose an array that is accessed just like a simple array, but a get and/or a set method is executed to validate and access the actual hidden data. In fact, we make the entire class look like an array. That's very powerful indeed. Let's start with a simple class that implements an indexer for an integer array. What will make this interesting is that the array will actually be stored internally as an array of character strings. Let's start by creating a new console application in the usual way. The easiest way to add a new class to the project is to right click the project name in the explorer window and select Add|New Item. Type the name for the class in the dialog after making sure that class is selected. Figure 5-1 shows this dialog. Here is the code for our new class:

```csharp
using System;
using System.Collections.Generic;
using System.Text;
namespace Index0
{
    class IntArray
    {
        private string[] stringInts;
        public IntArray(int n)
```
This is not a very robust implementation because there is no error detection. The first place where an error might occur is in the constructor. I am not checking to make sure the array size is greater than zero. Even if I did how would we report the error to the user? Constructors don't return arguments so we can't so it that way. An exception would be generated if the new operator is invoked with a negative value, but that would be a generic exception and not one associated with our new class. A better approach would be to create our own exception type and throw that exception for all errors. I am not going to do that because I want to focus on the indexer. The other area where exceptions will be generated is if we try to access the array with an index out of bounds. You will also notice that I have not provided a default constructor, but only the overloaded constructor taking an integer argument. This prevents the user of the class from trying to construct an instance of the IntArray class without specifying the size.

The indexer itself is declared with a special syntax using the this keyword and square brackets to specify the index. By the way, the index does not have to be an integer and can even be multidimensional. That's more advanced and something you will probably investigate as you gain some experience. The get and set methods are very similar to the counterparts for properties. The keyword value is used to represent the value to be stored in the array since it is implicitly passed to the set method. Everything else is straightforward. I am using the Convert.ToInt32 method to convert the string representation of the integer back into an int type. I don't need to worry about an exception since all the strings will be valid integers since the ToString method was used to generate them. Here is a simple program to fill an IntArray with powers of two using the Math class method Pow and the output is shown in Figure 5-2:

```csharp
namespace Index0
{
    class Program
    {
        static void Main(string[] args)
        {
            stringInts = new string[n];
        }
        public int this[int i]
        {
            get
            {
                return Convert.ToInt32(stringInts[i]);
            }
            set
            {
                stringInts[i] = value.ToString();
            }
        }
    }
}
```
```csharp
static void Main(string[] args)
{
    IntArray ia = new IntArray(10);
    for (int i = 0; i < 10; ++i)
        ia[i] = (int)Math.Pow(2, i);
    for (int i = 0; i < 10; ++i)
        Console.WriteLine(ia[i]);
}

Figure 5-1
```
Here is a more interesting class that uses just the `get` method. It constructs a set of Fibonacci numbers that can be accessed by an index:

```csharp
using System;
using System.Collections.Generic;
using System.Text;

namespace Index1
{
    class Fib
    {
        private long[] numbers;
        private int count;
        public Fib(int n)
        {
            if (n < 2) n = 2;
            count = n;
            numbers = new long[n];
            numbers[0] = 1;
            numbers[1] = 1;
            for (int i = 2; i < n; ++i)
            {
                numbers[i] = numbers[i - 2] + numbers[i - 1];
            }
        }
        public long this[int idx]
        {
            get
            {
                if (idx < 0 || idx >= count) return 0;
                return numbers[idx];
            }
        }
    }
}
```

I use a different approach to handling errors. If the argument passed to the constructor is less than two I force a set of just two numbers. Any array access that is
out of bounds returns a zero value. There is one more very subtle bug. Can you guess what it is? I will give you a little hint. What is the $100^{th}$ Fibonacci number?

To demonstrate this class I wrote a Windows Forms application that uses a panel for scrolling. I covered how to do that in Chapter <ref>. The code follows and the output is shown in Figure 5-3.

### Index1 – Form1.cs

```csharp
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Index1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void panel1_Paint(object sender, PaintEventArgs e)
        {
            const int count = 50;
            Graphics g = e.Graphics;
            int h = (int) Font.GetHeight();
            panel1.Height = count * h;
            Fib fibNumbers = new Fib(count);
            for (int i = 0; i < count; ++i)
                g.DrawString(fibNumbers[i].ToString(), Font, Brushes.Black, 0, i * h);
        }
    }
}
```
Interfaces are a C# language feature that is very important to users of the .NET FCL. An interface is a specification for a set of methods to be implemented by a class. An interface does not provide any actual code for these methods, but rather the parameters and return type for these methods. Of course a class that implements an interface must provide an implementation that adheres to the functionality the interface provides. This can't be enforced by any C# language features. The only enforcement is that all the methods of the interface must be implemented using the specific parameters and return type for each method. If a class implements an interface it is essentially a contract to provide functionality that is consistent across all classes that implement the interface. To use a simple analogy, consider the interface used to plug electrical appliances into the power grid. If we provide an AC outlet with a standard three prong plug we expect that the outlet will provide 120 volts alternating current. It would be a disaster to implement this outlet using 300 volt direct current. An interface improperly implemented would be similarly disastrous at the programming level.

To declare an interface we merely specify the methods using the interface keyword in a format very similar to a class. It doesn't always make sense to implement an interface for just any class. The class has to lend itself to a particular interface. Suppose we have a class that has a color associated with it. We might want an interface that returns the color as a character string. The internal representation of the color is hidden. Here is an interface that provides such a feature:

```csharp
using System;
using System.Collections.Generic;
using System.Text;
```
The name of the interface is \textit{IColorString}. By convention interfaces are normally named with a leading capital I character. Our interface has only one method, \texttt{GetColorString}. This method takes no parameters and returns a \texttt{string}. Interfaces do not have access modifiers such as \texttt{public}. I added this class to the project by using the add new item feature discussed earlier except that I selected interface instead of class.

Now let's create a class that implements this interface. Here it is:

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;

namespace Interface1
{
    class ColoredClass: IGetColorString
    {
        Color myColor = Color.Empty;
        public Color MyColor
        {
            get
            {
                return myColor;
            }
            set
            {
                myColor = value;
            }
        }
        public string GetColorString()
        {
            return myColor.ToString();
        }
    }
}
```

I needed to add the \textit{System.Drawing} namespace in order to use the \texttt{Color} structure. As you can see I have provided a property to access the internal color. The following is a form that exercises our colored class and the interface. The output is shown in Figure 5-4.
When I learned about interfaces one of the first questions I asked was why they are needed. I conjectured that if I were to just implement the methods of the interface then I would accomplish the same thing. In the previous example this is certainly true. Let's consider a slightly different scenario. Suppose we were to write a class that had methods that in some way processed objects that were yet to be defined. Classes that support data structures such as hash tables are good examples. These are commonly called container classes. Since we don't know the names (types) of the classes that will be processed by the methods of these classes we can't use their names as the types of parameters. One thing we always know is that every class derives from Object. We can always use the Object type as the parameter's type. Unfortunately, to invoke methods other than the polymorphic methods of Object we need to cast our parameter to its actual type or a known base type that supports the methods we want to invoke. But remember, we don't know the parameter's type at the time we wrote our class. Here is where interfaces really come in handy. If we were to specify that any class object

```csharp
namespace Interface1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            int cy = Font.Height;
            ColoredClass c = new ColoredClass();
            g.DrawString(c.GetColorString(), Font, Brushes.Black, 10, 0);
            c.MyColor = Color.DodgerBlue;
            g.DrawString(c.GetColorString(), Font, Brushes.Black, 10, cy);
            c.MyColor = Color.Gold;
            g.DrawString(c.GetColorString(), Font, Brushes.Black, 10, 2*cy);
        }
    }
}
```
passed as a parameter was to implement a specific interface then we can cast the parameter to this interface and invoke the methods of the interface.

We could try to solve this problem with conventional classes or abstract base classes. I discuss abstract base classes in Chapter <ref>. There is one major limitation if we were to do this. The C# language does not support multiple inheritance. That means that we must always have a single derivation path from any class back to Object. There are also some other drawbacks that I will not discuss at this time. A class may implement more than one interface. This really comes in handy. The FCL has many classes that support multiple interfaces. So while C# doesn't support multiple inheritance as does C++, it does support multiple interfaces.

**Is and As Operators**

A risk is involved when we cast one type to another. If the type can't be cast because it isn’t an instance of the type we are casting it to, an exception will be generated if the type is not known at compile time. If the type is known at compile time then the compiler will report an error. We can obviously catch this exception and take appropriate action. This isn't always the best approach since it isn't the most efficient. The C# language provides a couple of very handy operators to assist us in casting.

The **is** operator can be used to test if one type can be cast to another. For example:

```csharp
if (t is type2)
{
    type2 t2 = (type2) t;
    //use t2
}
```

The **as** operator simplifies the above to a conditional cast.

```csharp
type2 t2 = t as type2;
if (t2 != null)
{
    //use t2
}
```

If the cast fails then `t2` will be null, otherwise it will be a reference to a `type2` object. Here is a version of the previous example demonstrating how we can use the **as** operator:

```csharp
Interface2
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
```
namespace Interface2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            ColoredClass c = new ColoredClass();
            Draw(c, g);
            c.MyColor = Color.DodgerBlue;
            Draw(c, g);
            c.MyColor = Color.Gold;
            Draw(c, g);
            Object o = new Object();
            Draw(o, g);
        }
        private void Draw(Object o, Graphics g)
        {
            ypos += Font.Height;
            IGetColorString lo = o as IGetColorString;
            if (lo != null)
            {
                g.DrawString(lo.GetColorString(), Font, Brushes.Black, 10, ypos);
            }
            else
            {
                g.DrawString("No color!", Font, Brushes.Black, 10, ypos);
            }
        }
        private int ypos = 0;
    }
}

Notice that Drawis passed a parameter of type Object and so it could be virtually anything. We expect the object to implement the IGetColorString interface but we use the as operator just to make sure. The output is shown in Error! Reference source not found..
**IEnumerable, an Important Interface**

For classes that support collections of objects it is often a desire to iterate over all the objects in the collection. We can use an indexer for this as long as we have a way to determine the number of objects in the collection. This is normally a property provided by the designer of the class. Code such as the following is standard fare:

```csharp
CollectionClass coll = new CollectionClass();
//fill the collection
CCType item;
for (int i=0; i<coll.Count; ++i)
{
    item = coll[i];
    //use item
}
```

Since this type of access is so common the C# language includes the `foreach` statement that allows us to write the above this way:

```csharp
CollectionClass coll = new CollectionClass();
//fill the collection
foreach (CCType item in coll)
{
    //use item
}
```

In both examples `item` is declared as the type of the objects stored in the collection.

Unfortunately we can't use the `foreach` statement on just any class we define. For the `foreach` statement to work the class must implement the `IEnumerable` interface. Fortunately virtually all collection type classes in the .NET FCL implement the `IEnumerable` interface. The `IEnumerable` interface has exactly one method with the following declaration:

```csharp
IEnumerator GetEnumerator();
```

`IEnumerator` is another interface that actually does the work. This interface has the methods shown in Table 5-1.

<table>
<thead>
<tr>
<th>IEnumerator Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object Current {get;}</td>
<td>Return the object at the current index.</td>
</tr>
<tr>
<td>bool MoveNext();</td>
<td>Increment the index. Return false if the index is advanced beyond the end of the collection else return true.</td>
</tr>
<tr>
<td>void Reset();</td>
<td>Reset the index to -1.</td>
</tr>
</tbody>
</table>
The actual implementation does not require that the index is actually an integer. I am using an integer index to make it easier to understand what is going on. The enumerator is never used with an explicit index value but rather the concept of the current position is utilized. Reset and MoveNext are used to control the position within the collection.

A totally correct implementation also requires that the InvalidOperationException exception be thrown if the collection changes during the time period we are iterating through the collection. I am going to take the liberty of ignoring this requirement since we haven't covered the concepts of multi-threaded programming. Unless the collection is accessed in a multi-threaded program this exception is never capable of being thrown.

Let's modify the Fibonacci class we wrote earlier in the chapter to support the IEnumerable and IEnumerator interfaces. New and modified code is shown in bold:

```csharp
namespace IEnumerable1
{
    class Fib : IEnumerable, IEnumerator
    {
        private long[] numbers;
        private int count;
        public Fib(int n)
        {
            if (n < 2) n = 2;
            count = n;
            numbers = new long[n];
            numbers[0] = 1;
            numbers[1] = 1;
            for (int i = 2; i < n; ++i)
                numbers[i] = numbers[i - 2] + numbers[i - 1];
        }
        public long this[int idx]
        {
            get
            {
                if (idx < 0 || idx >= count) return 0;
                return numbers[idx];
            }
        }
        private int index = -1;

        public IEnumerator GetEnumerator()
        {
            return this;
        }
        public void Reset()
        {
            index = -1;
        }
    }
}
```
I needed to change the collections namespace that was automatically generated by Visual Studio 2005. The default is to use the generic collection classes. I will cover just what generic classes are in Chapter <ref>. The other change was to add the two interfaces to the line declaring our class. The code necessary to implement the two interfaces follows the original code. Again, I want to indicate that this implementation does not meet the exact specification since we are not checking for the collection changing from call to call.

The following is the modified code for the form. Note that now we are using the `foreach` statement. Again the changes and added code is in bold:

```csharp
namespace Ienumerable1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void panel1_Paint(object sender, PaintEventArgs e)
        {
            const int count = 50;
            Graphics g = e.Graphics;
            int h = (int) Font.GetHeight();
            panel1.Height = count * h;
            int i = 0;
            Fib fibNumbers = new Fib(count);
            foreach (long n in fibNumbers)
            {
```
The output is exactly the same as in Figure 5-3.

**Iterators**

C# 2.0 makes the process of providing an enumerator for a collection very easy through the addition of a new compiler level feature, the *iterator*. Iterators are essentially a shortcut method to have the compiler generate code that you would normally type. No run-time support is involved. Recall that the three methods we need to implement for the `IEnumerator` interface are `Reset`, `MoveNext`, and `Current`. If you think about it these three methods implement a type of for loop where the index is retained between calls to `Current`. What if we could write code that looked exactly like a for loop and have the compiler generate the code required by `IEnumerator`? Well that’s exactly the way Iterators work in C# 2.0; The following code is a rewrite of our Fibonacci class using an iterator:

```csharp
using System;
using System.Collections;
using System.Text;

namespace Ienumerable2
{
    class Fib: IEnumerable
    {
        private long[] numbers;
        private int count;
        public Fib(int n)
        {
            if (n < 2) n = 2;
            count = n;
            numbers = new long[n];
            numbers[0] = 1;
            numbers[1] = 1;
            for (int i = 2; i < n; ++i)
                numbers[i] = numbers[i - 2] + numbers[i - 1];
        }
        public long this[int idx]
        {
            get
            {
                if (idx < 0 || idx >= count) return 0;
                return numbers[idx];
            }
        }
        public IEnumerator GetEnumerator()
        {
```
The new code is shown in bold. It is our implementation of GetEnumerator. As you can see it is a simple for loop with the exception of the keyword yield. This is a new keyword in C# 2.0. Don't worry if you are using this keyword as an identifier somewhere else in your program as might be the case if you are updating older programs written for C# 1.0. The compiler is smart enough to use the context to distinguish between the two uses of the keyword.

There are two details that you should be aware of in using Iterators. The Reset method is not implemented as well as detection of collection changes made during an iteration. I discussed this latter issue above. The absence of the Reset method is not really a major drawback. It is not used all that often and the workaround is to merely re-instantiate the enumerator which resets the index. There are some additional features of Iterators that you can explore, but I have just presented the most often used capabilities.