Indexers, Interfaces, and Enumerators

Indexers

- An indexer is very similar to a property.
- Indexers operate on arrays of objects.
- Indexers have a *get* and *set* method.
- Indexers can *abstract* the actual storage structure used.
- For example, integers can be stored as strings internally.
- Another example is a *sparse* array.

Index0 Example

```
Index0 – IntArray.cs
using System;
using System.Collections.Generic;
using System.Text;
namespace Index0
{
    class IntArray
    {
        private string[] stringInts;
        public IntArray(int n)
        {
            stringInts = new string[n];
        }
```

Index0 Example –Contd.

```
        public int this[int i]
        {
            get
            {
                return Convert.ToInt32(stringInts[i]);
            }
            set
            {
                stringInts[i] = value.ToString();
            }
        }
    }
}
```

Issues

- Bad constructor argument.
- Index out of bounds.
- No default constructor ensures that a size is provided.
- *Throw* an exception for bad arguments!

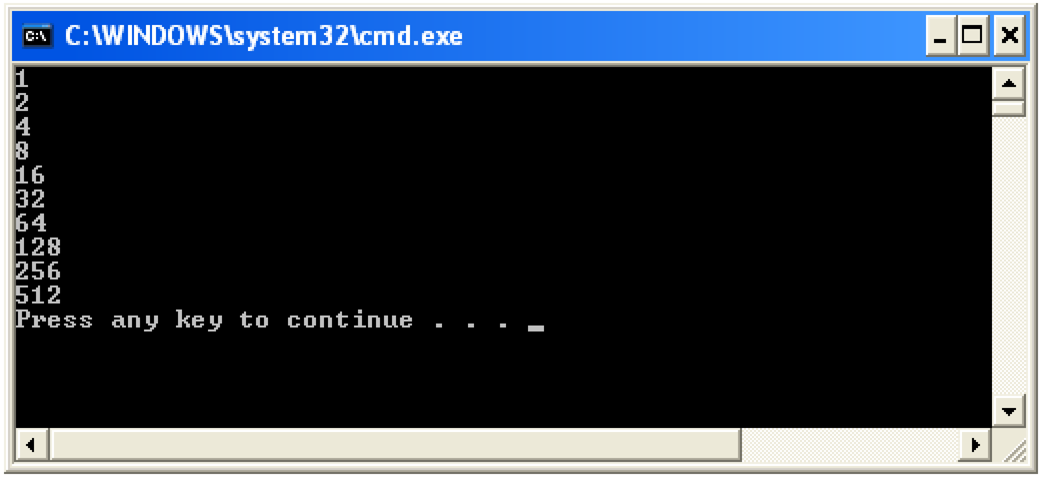
Index1 Example

```
Index0 – Program.cs
using System;
using System.Collections.Generic;
using System.Text;
namespace Index0
{
    class Program
    {
        static void Main(string[] args)
```

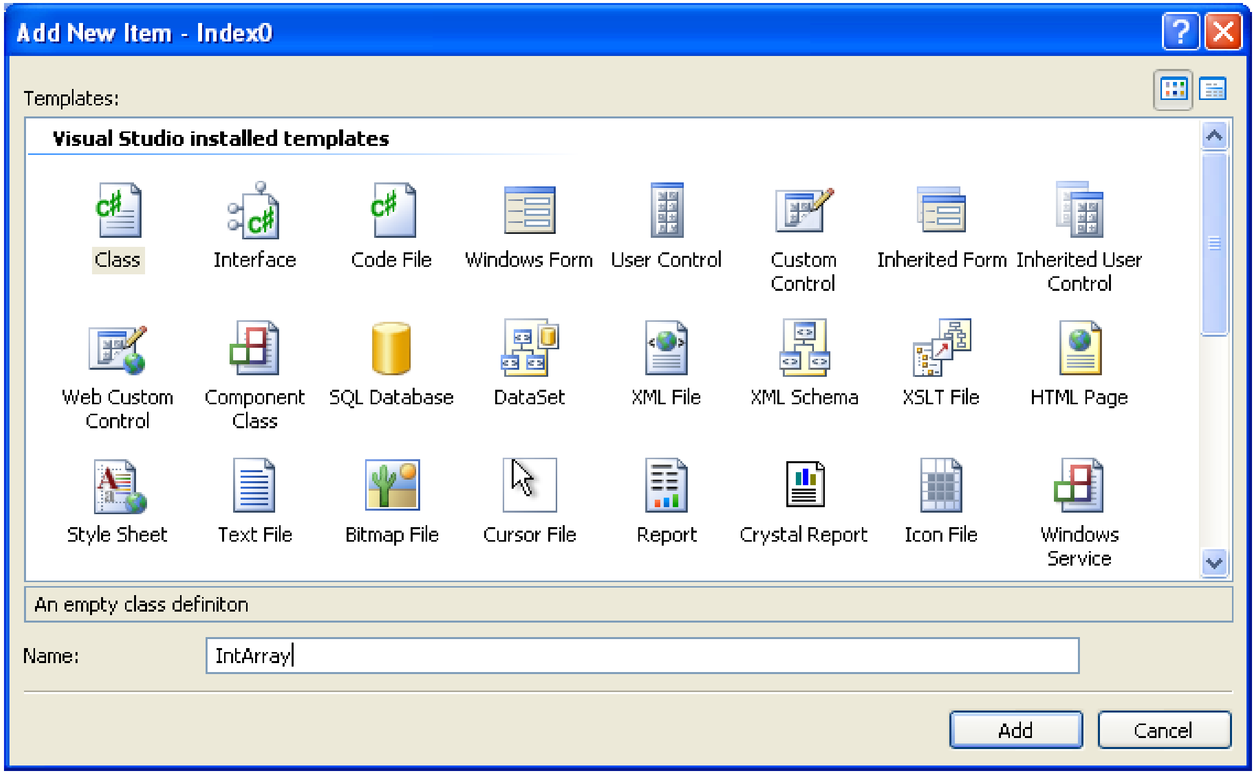
Index1 Example –Contd.

```
        {
            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]);
        }
    }
}
```

Output


C:\WINDOWS\system32\cmd.exe
1
2
4
8
16
32
64
128
256
512
Press any key to continue . . .

Adding a Class



Fibonacci Class

```
Index1 – Fib.cs
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;
```

Fibonacci Class – Contd.

```
        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];
        }
    }
}
```

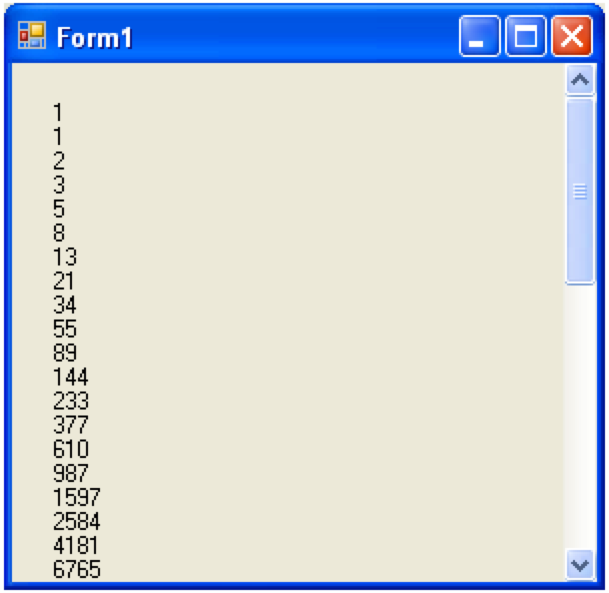
Windows Form for Fibonacci Class

```
Index1 – Form1.cs
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()
        {
```

Windows Form for Fibonacci Class – Contd.

```
            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);
        }
    }
}
```

Output


Form1
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765

Interfaces

- An interface is a *contract* to implement certain methods in a class.
- Interfaces have no method definitions, only declarations.
- Interfaces are similar to *abstract* classes but more restricted.
- By convention interfaces start with I.

A Simple Interface

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Interface1
{
    interface IGetColorString
    {
        string GetColorString();
    }
}
```

Implementing IGetColorString

```
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
            {
```

Implementing IGetColorString – Contd.

```
                return myColor;
            }
            set
            {
                myColor = value;
            }
        }
        public string GetColorString()
        {
            return myColor.ToString();
        }
    }
}
```

Associated Form

```
Interface1
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace Interface1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
```

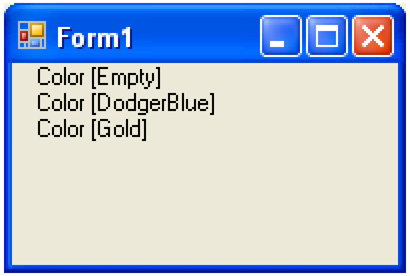
Associated Form – Contd.

```
        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);
```

Associated Form – Contd.

```
            c.MyColor = Color.Gold;
            g.DrawString(c.GetColorString(), Font,
               Brushes.Black, 10, 2*cy);
        }
    }
}
```

Output



Why Interfaces?

- Isn’t it sufficient to just implement the methods of an interface without using and interface declaration?
- Not if we don’t know the name of the class at compile time.
- Virtual methods is an alternative, but not ideal because it is not generalized but specific to a base class.
- We can’t add virtual methods to *object*.
- We can use the methods of an interface even if we don’t know the name of the class by casting the reference to the interface.
- We can then call the methods.

Is and As

- To use an interface we often cast the class to the interface.
- If, however, the class does not support the interface and we try to cast we will get an exception or compiler error depending on whether or not the class is known to the compiler.

Example

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

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

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

Example

```
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);
```

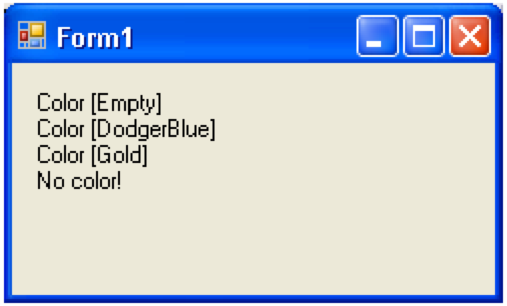
Example – Contd.

```
        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;
```

Example – Contd.

```
            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;
    }
}
```

Output



Foreach

```
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:

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

IEnumerable Interface

- IEnumerator GetEnumerator() returns a reference to another very important interface, IEnumerastor.
- IEnumerator allows us to iterate through the items in the class.
- The following slide shows the three methods used with this interface.
- They are self explanatory.

IEnumerator Interface

IEnumerator Method	*Description*
object Current {get;}	Return the object at the current index.
bool MoveNext();	Increment the index. Return false if the index is advanced beyond the end of the collection else return true.
void Reset();	Reset the index to -1.

Fibonacci Example

```
//Ienumerable1
using System;
using System.Collections;
using System.Text;
namespace Ienumerable1
{
    class Fib: IEnumerable, IEnumerator
    {
        private long[] numbers;
        private int count;
        public Fib(int n)
        {
```

Fibonacci Example – Contd.

```
            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;
```

Fibonacci Example – Contd.

```
                return numbers[idx];
            }
        }
        private int index = -1;
        public IEnumerator GetEnumerator()
        {
            return this;
        }
        public void Reset()
        {
            index = -1;
        }
        public bool MoveNext()
```

Fibonacci Example – Contd.

```
        {
            if (index < count) ++index;
            return index < count;
        }
        public Object Current
        {
            get
            {
                return numbers[index];
            }
        }
    }
}
```

Modified Form

```
//Ienumerable1
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
namespace Ienumerable1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
```

Modified Form – Contd.

```
        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)
            {
                g.DrawString(n.ToString(), Font, Brushes.Black,
                       0, i * h);
                ++i;
            }
        }
    }
}
```

Iterators

- C# version 2.0 makes setting up iterators very easy.
- Iterators are essentially a shortcut method to have the compiler generate code that you would normally type.
- The entire GetEnumerator method can be trivially implemented.
- The *yield* keyword is the way we simplify the loop code.

Using an Iterator

```
Ienumerable2
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)
```

Using an Iterator – Contd.

```
                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()
        {
            for (int index = 0; index < count; ++index)
            {
                yield return numbers[index];
            }
        }
    }
}
```