

Topics for today

- More on generics
- Delegates
- Events
- Graphics

```
class Stack<T>
{
    private T[] _itemsArray;
    private int _index = 0;

    public void Push(T item) { ... }
    public T Pop()
    {
        _index--;
        if (_index >= 0)
        {
            return _itemsArray[_index];
        }
        else
        {
            _index = 0;
            return ??? //What if we want to return the default value
        }
    }
}
```

```
class Stack<T>
{
    private T[] _itemsArray;
    private int _index = 0;

    public void Push(T item) { ... }
    public T Pop()
    {
        _index--;
        if (_index >= 0)
        {
            return _itemsArray[_index];
        }
        else
        {
            _index = 0;
            return default(T);
        }
    }
}
```

Returns null for reference types, and a "zero whitewash" for value types

Generics and Inheritance

```
public class BaseClass<T>
{...}
public class SubClass : BaseClass<int>
{...}
```

Non-generic

Generics and Inheritance

```
public class BaseClass<T>
{...}
public class SubClass : BaseClass<int>
{...}

public class SubClass<T> : BaseClass<T>
{...}
```

Generic

Generics and Inheritance

```
public class BaseClass<T>
{
    public virtual T SomeMethod()
    {...}
}

public class SubClass : BaseClass<int>
{
    public override int SomeMethod()
    {...}
}
```

Virtual methods are OK.

The non-generic subclass fills in the type.

Generic Methods

```
public class MyClass
{
    public void MyMethod<T>(T t)
    {...}
}
```

This allows us to call the method with different types.

```
MyClass mc = new MyClass();
mc.MyMethod<int>(3);

mc.MyMethod("hi");
```

Here the compiler infers the type.

Generic Static Methods

```
public class MyClass
{
    public static void Swap<T>(ref T item1, ref T item2)
    {
        T temp = item1;
        item1 = item2;
        item2 = temp;
    }
}
```

```
int n1 = 1, n2 = 2;
MyClass.Swap<int>(ref n1, ref n2);

decimal d1 = 0, d2 = 5.678;
MyClass.Swap<decimal>(ref d1, ref d2);

string s1 = "Bob", s2 = "Hector";
MyClass.Swap(ref s1, ref s2);
```

In the last example we are letting the compiler infer the type. Note that we can also have generic instance methods.

Why won't this work?

```
public class Calculator<T>
{
    public T Add(T arg1, T arg2)
    {
        return arg1 + arg2; //Does not compile
    }
    //Rest of the methods
}
```

The compiler can't be sure that + will be defined for every possible type that you might use in place of T.

There is no constraint that says "T supports +".

Delegates

- Simplest description: type-safe function pointers
- Declaring a delegate defines a class that derives from **System.MulticastDelegate**, which derives from **System.Delegate**

Delegates

- Delegates maintain a list of methods to be called when the delegate is invoked.
- To insure type safety, the signature (return type and arguments) of acceptable callback methods is specified when the delegate class is defined.
- We instantiate a delegate by "wrapping" the callback method.
- We can add methods to the callback chain or remove methods from the chain.

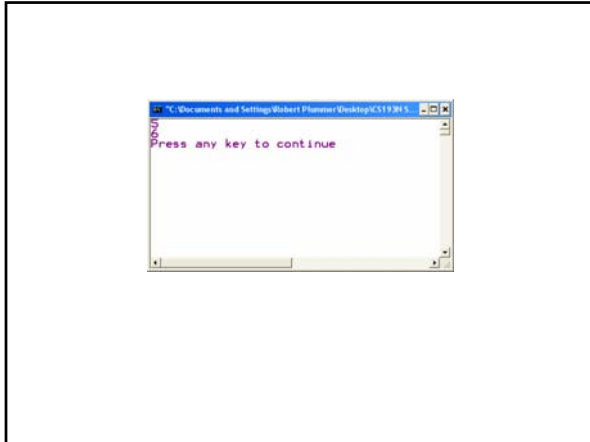
```
delegate int MathOp(int x, int y); //Declare delegate type

class Class1
{
    static void Main(string[] args)
    {
        MathOp f1 = new MathOp(Class1.Add); //Instantiate
        MathOp f2 = new MathOp(Class1.Mult); // delegates

        Console.WriteLine(DoSomeMath(f1, 2, 3));
        Console.WriteLine(DoSomeMath(f2, 2, 3));
    }

    public static int DoSomeMath(MathOp fn, int x, int y)
    {
        return fn(x, y); //Invoke the delegate
    }

    public static int Add(int a, int b)
    {
        return a + b;
    }
}
```



```

delegate int MathOp(int x, int y); //Declare delegate type

class Class1
{
    static void Main(string[] args)
    {
        MathOp f1 = null;
        MathOp f2 = null;

        f1 += Class1.Add;
        f2 += Class1.Mult;

        Console.WriteLine(DoSomeMath(f1, 2, 3));
        Console.WriteLine(DoSomeMath(f2, 2, 3));
    }

    public static int DoSomeMath(MathOp fn, int x, int y)
    {
        return fn(x, y); //Invoke the delegate
    }
}
    
```

```

delegate int MathOp(int x, int y); //Declare delegate type

class Class1
{
    static void Main(string[] args)
    {
        MathOp f1 = Class1.Add;
        MathOp f2 = Class1.Mult;

        Console.WriteLine(DoSomeMath(f1, 2, 3));
        Console.WriteLine(DoSomeMath(f2, 2, 3));
    }

    public static int DoSomeMath(MathOp fn, int x, int y)
    {
        return fn(x, y); //Invoke the delegate
    }
}
    
```

```

delegate int MathOp(int x, int y); //Declare delegate type

class Class1
{
    static void Main(string[] args)
    {
        MathOp f1 = null;

        f1 += Class1.Add;
        f1 += Class1.Mult;

        Console.WriteLine(DoSomeMath(f1, 2, 3));
    }

    public static int DoSomeMath(MathOp fn, int x, int y)
    {
        return fn(x, y); //Invoke the delegate
    }
}
    
```

Only the last result would be written to the console, so normally we only add multiple methods if the return type is void.

```

class DemoClass
{
    private string _name;
    private int _number;

    public DemoClass(string str, int x)
    {
        _name = str;
        _number = x;
    }

    public void ShowMyName()
    {
        Console.WriteLine("My name is " + this._name);
    }

    public void ShowMyNumber()
    {
        Console.WriteLine("My number is " + this._number);
    }
}
    
```

```

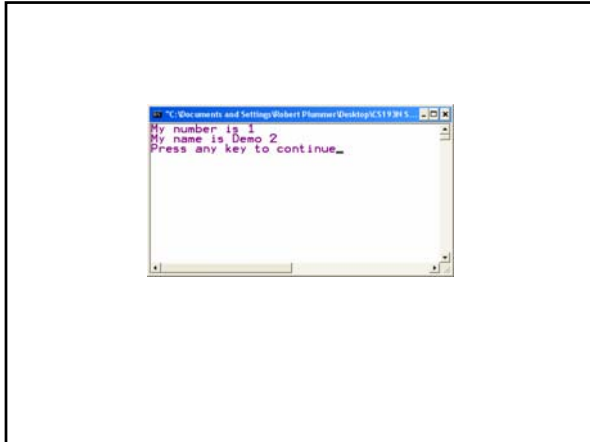
delegate void DemoOp(); //Declare delegate type

static void Main(string[] args)
{
    //Create some objects
    DemoClass dc1 = new DemoClass("Demo 1", 1);
    DemoClass dc2 = new DemoClass("Demo 2", 2);

    DemoOp demoDel = null; //Declare an empty delegate

    demoDel += dc1.ShowMyNumber; //Add a method to list
    demoDel += dc2.ShowMyName; //Add another one

    demoDel(); //Invoke the delegate
}
    
```



```
class DemoClass
{
    private string _name;
    private int _number;

    public DemoClass(string str, int x)
    {
        _name = str;
        _number = x;
    }

    public void ShowMyName()
    {
        Console.WriteLine("My name is " + this._name);
    }

    public void ShowMyNumber()
    {
        Console.WriteLine("My number is " + this._number);
    }
}
```

What is "this" when the method is called via a delegate?

```
delegate void DemoOp(); //Declare delegate type

static void Main(string[] args)
{
    //Create some objects
    DemoClass dc1 = new DemoClass("Demo 1", 1);
    DemoClass dc2 = new DemoClass("Demo 2", 2);

    //Declare an empty delegate
    DemoOp demoDel = null;

    //Add a method to list
    demoDel += dc1.ShowMyNumber;

    //Add another one
    demoDel += dc2.ShowMyName;

    //Invoke the delegate
    demoDel();
}
```

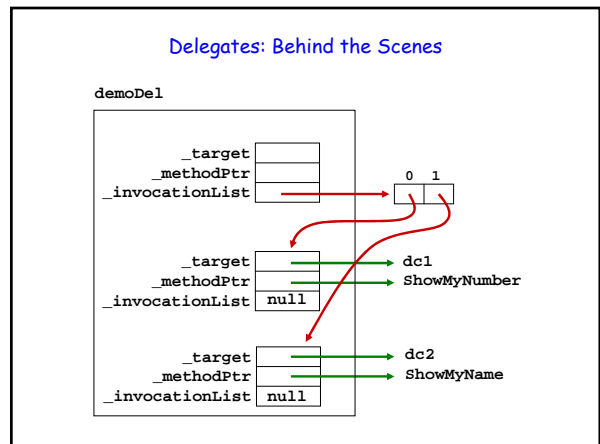
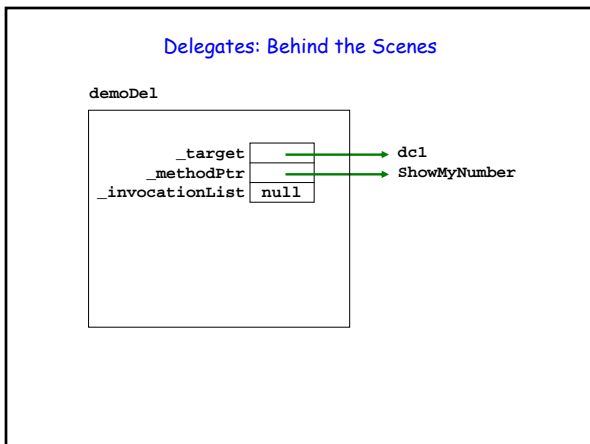
```
delegate void DemoOp(); //Declare delegate type

static void Main(string[] args)
{
    //Create some objects
    DemoClass dc1 = new DemoClass("Demo 1", 1);
    DemoClass dc2 = new DemoClass("Demo 2", 2);

    //Declare an empty delegate
    DemoOp demoDel = null;

    //Add a method to list
    demoDel += dc1.ShowMyNumber;
    //Add another one
    demoDel += dc2.ShowMyName;
    //Invoke the delegate
    demoDel();

    //Remove a method from list
    demoDel -= dc1.ShowMyNumber;
    demoDel();
}
```



Using Anonymous Methods with Delegates

For simple tasks like adding two integers, there is no need to define a named method like Add. We can use a delegate to call an "anonymous" method that we define with inline code.

```

delegate int MathOp(int x, int y);

MathOp f3 = delegate(int a, int b) { return a + b; };

Console.WriteLine(DoSomeMath(f3, 2, 3));

MathOp f4 = delegate(int a, int b) {
    int t = a;
    if (b > a) t = b;
    return t;
};

Console.WriteLine(DoSomeMath(f4, 2, 3));
    
```

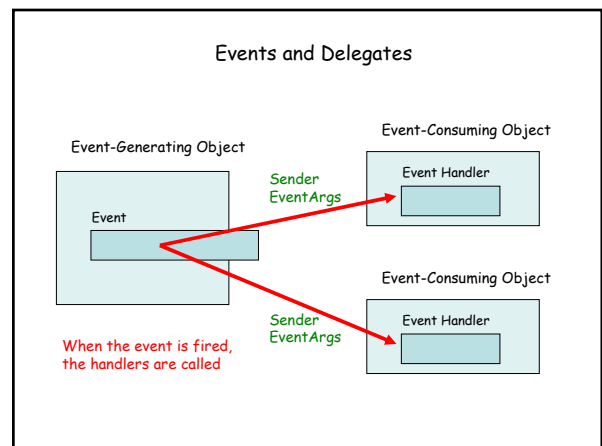
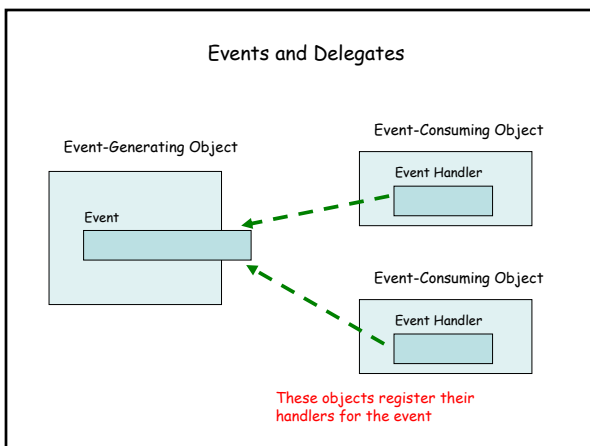
What if one of the methods attached to a delegate throws an exception?

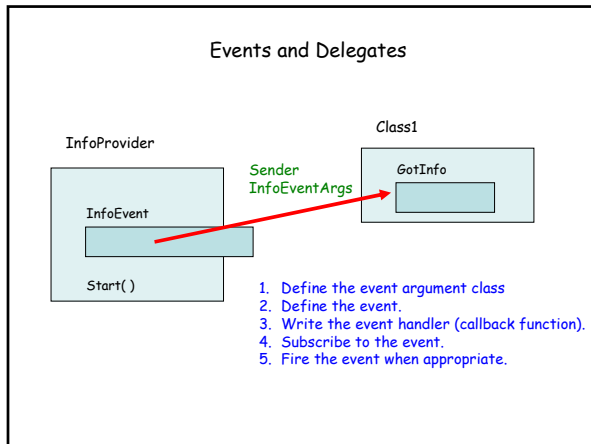
Visual Studio demo

What does an anonymous method have access to?

Visual Studio demo

- ### The Microsoft .NET Event Model
-
- Events provide a way for one object to notify another object that something has happened
 - Events use the **delegate** mechanism for invoking callback functions
 - Visual Studio simplifies the use of events with GUI elements





```
public class InfoEventArgs : EventArgs
{
    private DateTime date;

    public InfoEventArgs(DateTime date)
    {
        this.date = date;
    }

    public DateTime Date
    {
        get {return date;}
    }
}
```

```
public class InfoProvider
{
    public event EventHandler<InfoEventArgs> InfoEvent;

    public InfoProvider()
    {
    }

    public void Start()
    {
        . . .
    }
}
```

This is a predefined delegate for event handler methods. Its definition is:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e)
    where TEventArgs : EventArgs;
```

```
public void GotInfo(Object sender, InfoEventArgs e)
{
    textBox1.Text = e.Date.ToString();
}

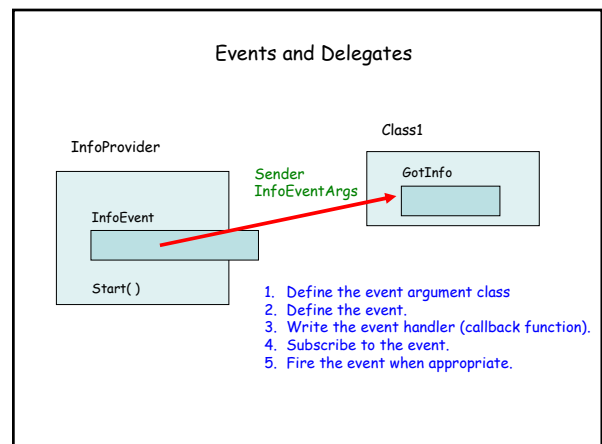
static void Main()
{
    InfoProvider ip = new InfoProvider();

    ip.InfoEvent += GotInfo;

    ip.Start();
}
```

```
class InfoProvider
{
    public event EventHandler<InfoEventArgs> InfoEvent;
    . . .
    public void Start()
    {
        do the following every second for 5 seconds
        {
            DateTime date = DateTime.Now;
            InfoEventArgs args = new InfoEventArgs(date);
            HaveInfo(args);
        }
    }

    protected void HaveInfo(InfoEventArgs args)
    {
        if (InfoEvent != null)
            InfoEvent(this, args);
    }
}
```



Graphics

GDI+ is the .NET Framework class library for graphical programming.

It provides:

- Drawing surfaces—windows, bitmaps, printers
- Tools for 2-D drawing—shapes, polygons, curves, brushes, pens
- Text-drawing features
- Image and bitmap support—read, draw onto any surface, draw into image
- Print and print preview
- Ability to work with any kind of .NET application—WinForm or WebForm

Graphics

In order to draw on a surface, we need an object of type `Graphics` associated with the surface.

The `Graphics` object:

- "Encapsulates" the surface
- Provides the "device context" for the surface
- Preserves graphics state for the surface
- Provides methods for 2D drawing


Some methods of the `Graphics` object:

<code>Clear</code>	<code>DrawPolygon</code>
<code>DrawArc</code>	<code>DrawRectangle</code>
<code>DrawBezier</code>	<code>DrawString</code>
<code>DrawCurve</code>	<code>FillClosedCurve</code>
<code>DrawEllipse</code>	<code>FillEllipse</code>
<code>DrawIcon</code>	<code>FillPath</code>
<code>DrawImage</code>	<code>FillPie</code>
<code>DrawLine</code>	<code>FillPolygon</code>
<code>DrawPath</code>	<code>FillRectangle</code>
<code>DrawPie</code>	<code>FillRegion</code>

- To draw on a surface, we need the `Graphics` object for the surface.
- `Graphics` are not persistent, so we need to redraw the image if the surface becomes invalid.
- To redraw our picture, we can handle the **Paint** event of the form, OR
- To redraw our picture, we can override the virtual function **OnPaint**, which **Form** inherits from **Control**
- A collection class may be needed to store items that need to be redrawn.

Graphics

If your form contains controls such as buttons, textboxes, and labels:



The controls are redrawn automatically when the form receives a **Paint** event.

In your **Blackjack** program, you are only responsible for drawing the hands.

Graphics

In the examples shown in class, there were two ways to get the form repainted:

- Write a handler for the **Paint** event and register for the event
- Override the **OnPaint** method that is inherited from the class **Control**

The second approach is suggested by many authors, but either will work.

Graphics

In the examples shown in class, there were two ways to get the form repainted:

- Write a handler for the Paint event and register for the event
- Override the OnPaint method that is inherited from the class Control

The second approach is suggested by many authors, but either will work. Note that your code should do this:

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    ...
}
```

This guarantees that any registered delegates receive the event.

Graphics

Question: what do you do when you need to redraw the screen?

Answer: force a Paint event.

```
Invalidate();
Update();
```

Causes an immediate call to OnPaint.

Graphics

To read a bitmap from a file, use the static method `FromFile` of the `Image` class:

```
Image myImage = Image.FromFile(filename);
```

Graphics

To read a bitmap from a file, use the static method `FromFile` of the `Image` class:

```
Image myImage = Image.FromFile(filename);
```

string

Graphics

To read a bitmap from a file, use the static method `FromFile` of the `Image` class:

```
Image myImage = Image.FromFile(filename);
```

If you have a `Graphics` object `g`, you can draw the image at location `x, y` with:

```
g.DrawImage(myImage, x, y);
```

How do you get a Graphics object?

- When you override the `OnPaint` method or install a handler for the `Paint` event, you will receive a `PaintEventArgs` object that has a `Graphics` property.
- To paint on a form (or control) at other times, you can call the form's `CreateGraphics` method. You should call `Dispose` on `Graphics` objects that you create.
- To draw on a bitmap in memory, you can obtain a `Graphics` object from the static method `Graphics.FromImage`. When you are done, you would call `Dispose`. To display the bitmap on a surface, you then use the `Graphics` object for the surface.