

The screenshot shows a web browser window titled ".NET Framework SDK Documentation - Random Members". The address bar shows "ms-help://MS.NETFrameworkSDK/cpref/htr". The main content area displays the "Random Members" page from the ".NET Framework Class Library". It includes sections for "Public Constructors" and "Public Methods".

**Public Constructors**

<a href="#">Random Constructor</a>	Overloaded. Initializes a new instance of the <b>Random</b> class.
------------------------------------	--

**Public Methods**

<a href="#">Equals</a> (inherited from <b>Object</b> )	Overloaded. Determines whether two <b>Object</b> instances are equal.
<a href="#">GetHashCode</a> (inherited from <b>Object</b> )	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
<a href="#">GetType</a> (inherited from <b>Object</b> )	Gets the <b>Type</b> of the current instance.
<a href="#">Next</a>	Overloaded. Returns a random number.

At the bottom, an "Index Results" table shows 1 topic found:

Title	Location
Random Members	.NET Framework Class Library

```
/*  
 * Introductory Hello CS585 project in Windows Forms C#  
 * uses console output to show program execution.  
 *  
 * Mike Barnes 9/22/02  
 */
```

```
using System;  
using System.Drawing;  
using System.Windows.Forms;
```

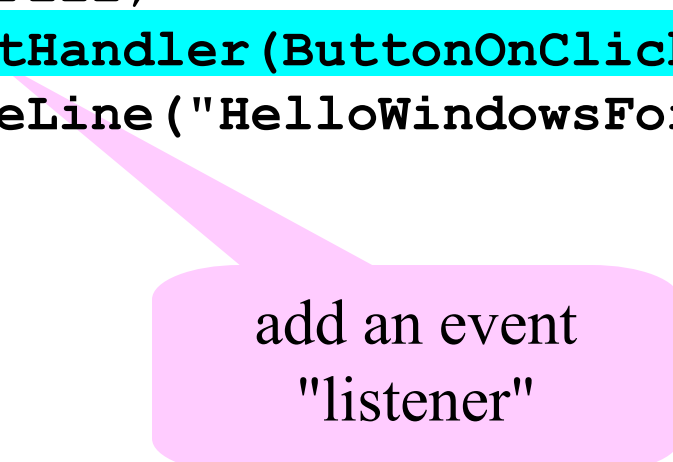
"extends"

```
class HelloWorldForm : Form {  
    readonly Button b;  
    string bLabel;  
    bool bState;
```

construct the application  
framework and execute it

```
public static void Main() {  
    System.Console.WriteLine("Main: enter");  
    Application.Run(new HelloWorldForm());  
    System.Console.WriteLine("Main: leave");  
}
```

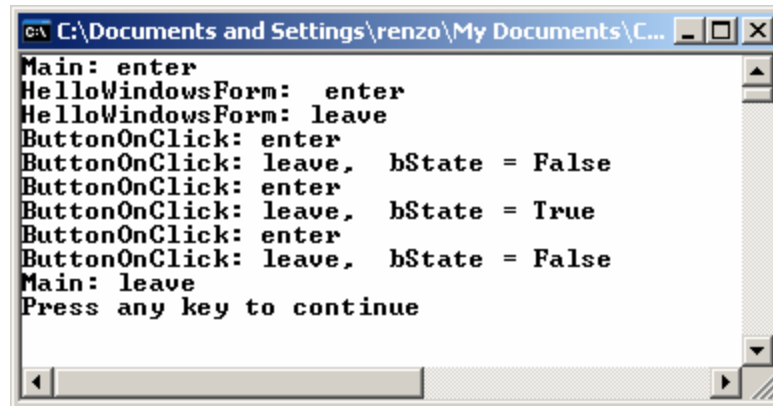
```
public HelloWorldForm() {  
    System.Console.WriteLine("HelloWindowsForm: enter");  
    Width = 100;           // set size of form  
    Height = 50;  
    bLabel = "Hello CS 585 !";    // assign string  
    bState = true;           // assign boolean state  
    b = new Button();        // create button  
    b.Parent = this;        // set button properties  
    b.Text = bLabel;  
    b.Dock = DockStyle.Fill;  
    b.Click += new EventHandler(ButtonOnClick);  
    System.Console.WriteLine("HelloWindowsForm: leave");  
}
```



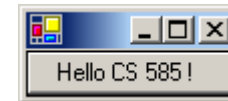
add an event  
"listener"

```
// process button clicks
```

```
void ButtonOnClick(object obj, EventArgs e) {  
    System.Console.WriteLine("ButtonOnClick: enter");  
    if (bState)  
        b.Text = "press me again";  
    else  
        b.Text = bLabel;  
    bState = !bState;  
    System.Console.WriteLine("ButtonOnClick:" +  
        "leave, bState = " + bState.ToString());  
}  
}
```



```
C:\Documents and Settings\renzo\My Documents\C...  
Main: enter  
HelloWindowsForm: enter  
HelloWindowsForm: leave  
ButtonOnClick: enter  
ButtonOnClick: leave, bState = False  
ButtonOnClick: enter  
ButtonOnClick: leave, bState = True  
ButtonOnClick: enter  
ButtonOnClick: leave, bState = False  
Main: leave  
Press any key to continue
```



To compile using the command line .NET Foundation SDK  
**csc HelloWorldsForms.cs**

## Some C# differences from C++ and Java

"using" for namespaces instead of "import" for packages

namespaces are compiled classes stored in DLLs (base classes)

"Main" is uppercase "bool" instead of "boolean"

Console.WriteLine(...) not System.out.println("...")

can have argument substitution and formatting like C printf

```
Console.WriteLine("iArray[{0}] == {1}", i, iArray[i]);
```

The arguments are substituted by index {0, 1} thus an index can be reused in the format specification string.

```
using System;
class WriteLine {
    public static void Main() {
        int i = 20;
        System.Console.WriteLine("show arg 3 times:\n" +
            "\t first = {0} \t second = {0} \t third = {0}", i); } }
```

show arg 3 times:

```
first = 20    second = 20    third = 20
```

# C# overview

C# language incorporates, builds upon, and extends concepts in C++ and Java.

Most types, arithmetic operators, and structured statements are the same.

```
char, short, int, long, double  
+ - * / %
```

Implicitly casts from a simpler type to a more complex (no loss of info).

Must explicitly cast from complex to simpler type ( possible loss).

```
float f;  
int i, j = 20;  
...  
f = j;  
i = (int) f;
```

Value types (built in, primitive) stored on run time stack (local, args)

Reference types (objects) are allocated (new) on the heap.

C# has garbage collection.

```
const type identifier = value;  
const int dimensions = 3;
```

Once declared and initialized value cannot be changed.

Enumerations: typed named constants

```
enum identifier [:base-type] { enumeration list };  
enum grades { f = 0, d = 1, c = 2, b = 3, a = 4 };
```

base-types must be of integral types

{ ushort, short, int, uint, long, ulong }

can't be char.

Useful with GUI controls settings and return values.

## Safer if && switch

```
int count = 2;
if (count = 0) ...;    // if (count == 0) ...
```

A valid C++ conditional statement that assigns 0 to count.

C# if expression accept only boolean values and there is no automatic conversion of bool to int -- compile time error.

Switch does not have automatically fall through for all cases.

```
switch ( switch expression ) {
    case case1      :           // valid
    case case2      : statement; // compile error
    case case3      : statement;
                       goto case4; // valid, goto case1 !
    case case4      : statement; break;
    case case5      : statement;
    [ default       : statement; ]
}
```

Switch expression: integral type, char, enum, or **string** type

# foreach

Easy iteration through a collection

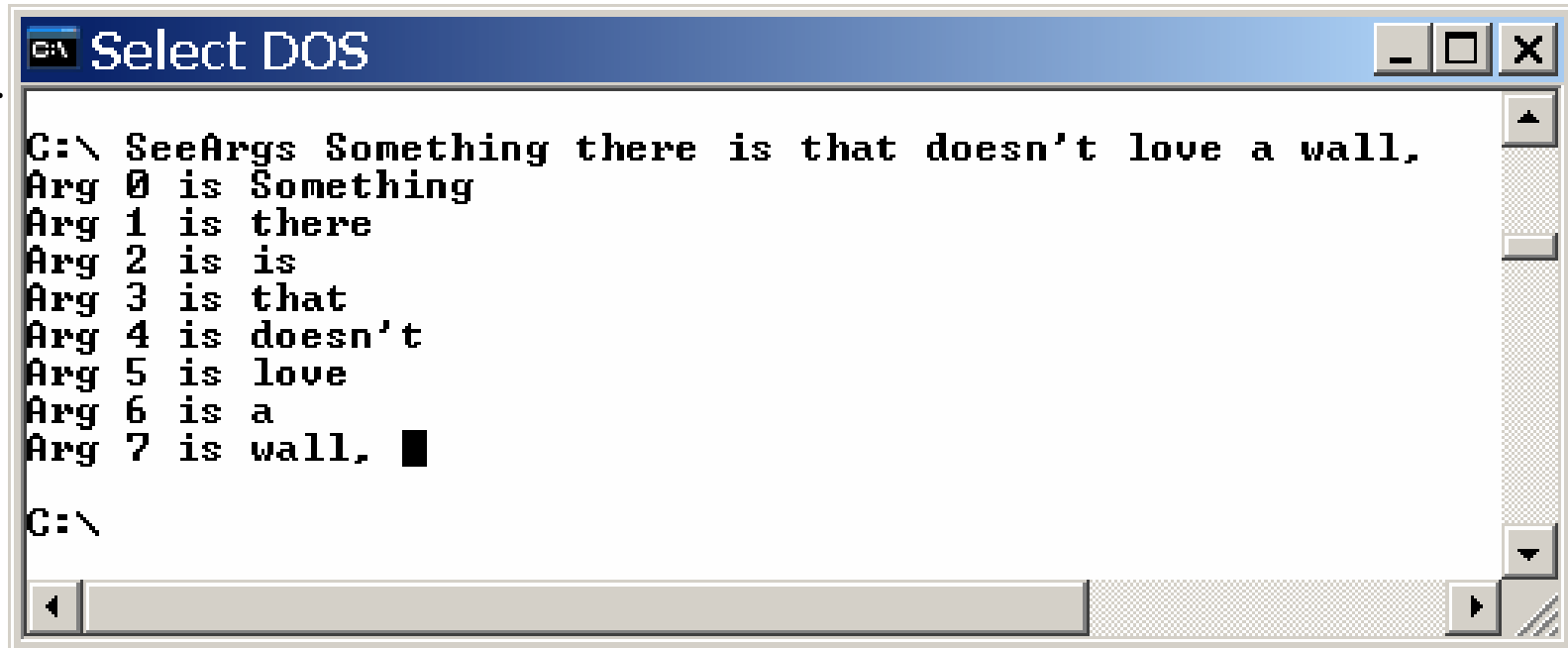
```
foreach ( type identifier in expression) statement;
```

```
using System;
```

public isn't needed for Main

```
class SeeArgs {  
    static void Main(string [] args) {  
        int i = 0;  
        foreach (string token in args)  
            Console.WriteLine("Arg {0} is {1}", i++, token);  
    }  
}
```

System isn't needed



The screenshot shows a DOS command window titled "Select DOS". The command prompt is at C:\. The user has entered the command "SeeArgs Something there is that doesn't love a wall,". The output of the program is as follows:

```
C:\> SeeArgs Something there is that doesn't love a wall,  
Arg 0 is Something  
Arg 1 is there  
Arg 2 is is  
Arg 3 is that  
Arg 4 is doesn't  
Arg 5 is love  
Arg 6 is a  
Arg 7 is wall, █  
  
C:\>
```

# Classes

C# has single inheritance

```
[access modifiers] class identifier [:base class]  
    { class body }
```

access modifiers {public, private, protected, internal, protected internal}

public, private, protected      same as Java and C++

internal                      accessible to methods in class's assembly  
                                 assemblies are \*.dll, or \*.exe files w/ class

protected internal          protected OR internal access.

private is default

Usually:    variables are private and methods are public.

Methods and Properties of a class can be either instance members or static (class) members.

Static members are accessed using the class name (shared).

Static methods cannot directly access nonstatic members.

# Methods

primitive data members in methods have default values:

```
numeric 0    bool false    char '\0'  
enum 0      reference null
```

**this** keyword similar to Java, equivalent to Smalltalk's self  
every method has an implicit this pointer.

Method overloading is similar to C++ and Java.

methods have same type and identifier but different signatures  
argument list vary by number, type or both.

Variable number of arguments: params

compiler constructs an array to match function signature

```
access type methodName ( params type[] args) {...}
```

...

```
methodName (arg1, arg2, arg3, ... argN);
```

# Passing arguments

Value types are passed into methods by value.

Passing value types by reference

```
// declare the method  
[access] [type] identifier ( ref int a, ref double b) { ... }  
...  
// invoke the method  
[object] identifier (ref var1, ref var2);
```

C# compiler requires that var1 and var2 must have values.

Passing uninitialized value types by reference using keyword **out**

```
// declare the method  
[access] [type] identifier ( out int a, out double b) { ... }  
...  
// invoke the method  
[object] identifier (out var1, out var2);
```

# inheritance

Derived classes (subclasses) can be derived (extended) from base classes (superclasses)

Derived classes inherit all data members and member methods **except constructors.**

Base constructors can be called from a subclass's constructor using **: base**

```
access class BaseClassName {  
    public BaseClassName ( [ type1 ] [arg1] ]* ) { ... }  
}
```

```
access class DerivedClassName : BaseClassName {  
    public DerivedClassName ( [ type2 ] [arg2] ]* )  
        : base ( [ type1 ] [arg1] ]* ) { ... }  
}
```

<< show, compile inheritDemo.cs file >>

<< show ViewPoly.cs version of ViewPoly.java after next slide >>

## polymorphism & virtual methods

Virtual methods are used to define methods that will be override and used called polymorphically (in the context of the virtual method)

Virtual specifies the root of the inheritance graph for polymorphic method calls.

in base class

```
access virtual type polyMethod(...) { ... }
```

in derived<sub>i</sub> class

```
access override type polyMethod(...) { ... }
```

C# adds the use of `new` with virtual methods to indicate that a virtual method with the same name has been introduced lower in the inheritance graph. (otherwise compiler generates a warning: `new || override ??`)

in later derived<sub>i+</sub> class

```
access new virtual type polyMethod(...) { ... }
```

The new virtual method hides (shadows) the initial virtual method (higher in the inheritance graph)

OO concept of encapsulation: private data public accessor mutator methods.

- + object (data) representation is independent of client's (caller's) use. Change representation w/o changing client how it is used.
- Client can't directly operate on object directly. Client needs to know public interface of object and some idea of its concept (what is used for)

Properties are "lite, nested classes" that provide accessor / mutators methods.

- + data object is hidden
- + data usage appears to be direct.

```
anObject.Property = aValue;
```

```
aValue = anObject.Property;
```

# Property declaration

```
public class ClassIdentifier {  
    private type propertyVariable;  
    ....  
    // property definition  
    public type PropertyIdentifier {  
        get {  
            // statements to return value  
            return propertyVariable; }  
        set {  
            // statements to set value  
            propertyVariable = value; }  
        .... }  
    ....  
ClassIdentifier aClass = new ClassIdentifier();  
    ....  
type thisValue = aValue;  
aClass.PropertyIdentifier = thisValue;
```

*propertyVariable* is  
encapsulated by  
*PropertyIdentifier*

keyword `value` is an implicit argument, it is the value used in the assignment (*thisValue*).

# property example

No "( )"

```
class AClass {
    int anIntProperty;
    int min, max; // set with AClass constructor
    public int AnIntProperty {
        set {
            if (value >= min && value <= max)
                anIntProperty = value;
            else
                throw new ArgumentOutOfRangeException(
                    "anIntProptery");    }
        get { return anIntProperty; } }
    // ... assume AClass is constructed ...
    try { AnIntProperty = someIntValue; }
    catch (Exception e) {
        Console.WriteLine(e); }
}
```

# Namespaces

Namespaces have: classes, structs, interfaces, enumerations, delegate  
structs are value "object based" objects.

Structs can't inherit (except object)

You can derive (subclass) from a struct

**delegates** (used as event handlers) are like interfaces except  
only 1 method to implement

created at runtime (interfaces are at compile time)

2 args: object that raised event, object that defines the event

Classes (or structs) have

fields data members, objects

constructors and methods

properties

operators (can be overridden)

indexers, reference objects like arrays

events

embedded (nested) classes, structures, interfaces, enumerations,  
delegates.

# MessageBox

A convenience dialog class

Petzold's hello world:

```
public static void Main() {  
    System.Windows.Forms.MessageBox.Show("Hello World"); }  
}
```

DialogResult Show (string str)

(..., string title)

(..., MessageBoxButtons mbb)

(..., MessageBoxIcon mbi)

(..., MessageBoxDefaultButton mbdb)

(..., MessageBoxOptions mbo)

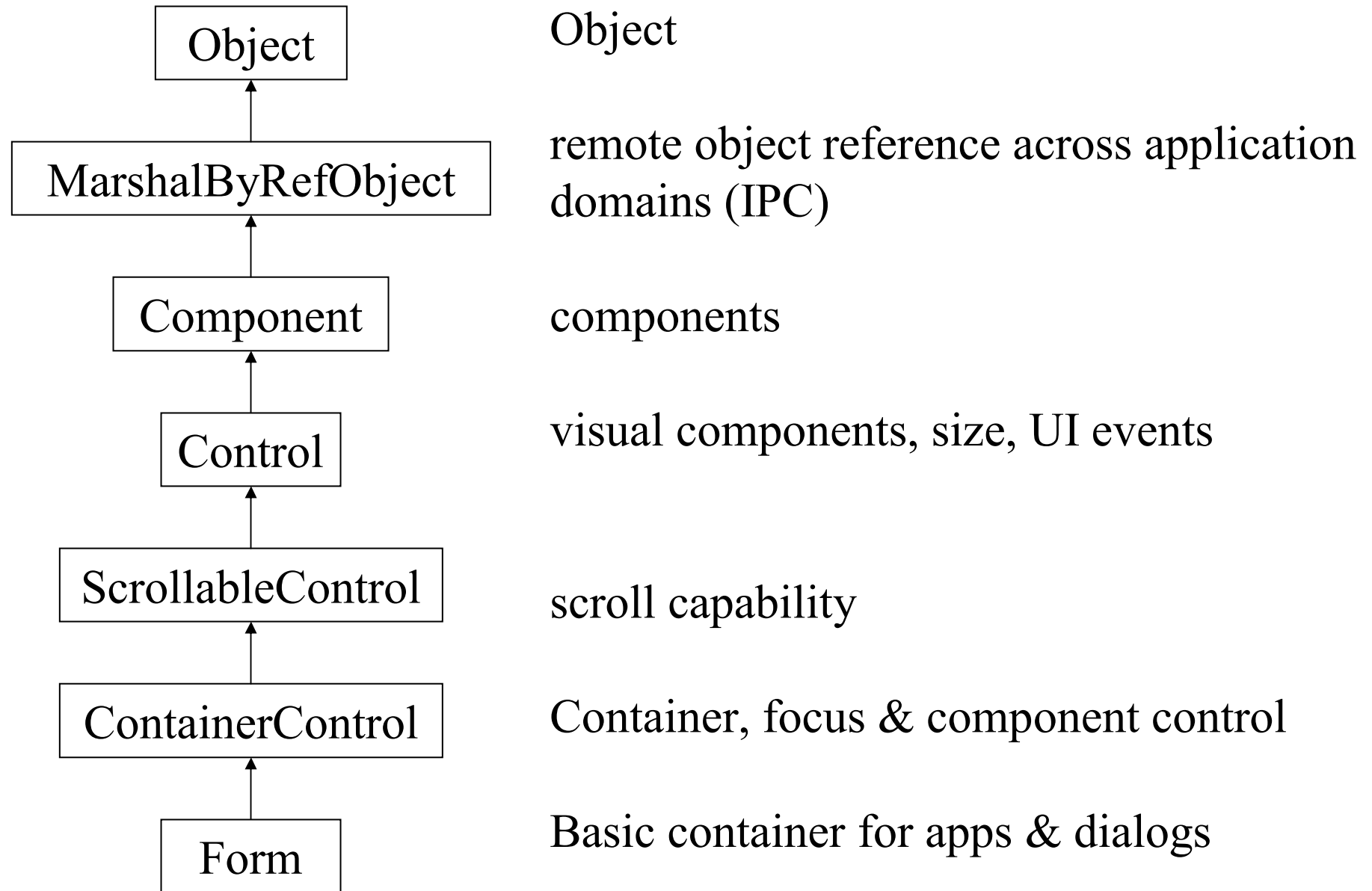
Enumerations are used for results and to set display arguments options

Show returns DialogResult enumeration {0..7}

None, Ok, Cancel, Abort, Retry, Ignore, Yes, No

Enumerations are used for MessageBoxButtons

OK, OKCancel, AbortRetryIgnore, YesNoCancel, YesNo, RetryCancel



Display or hide a Form (good for dialogs)

```
aForm.Show();           // or aForm.Hide();  
aForm.Visible = true;   // or false
```

When forms are the base container for an application use:

```
Application.Run(new aForm());
```

Application is a static class (like MessageBox).

Run takes a form object, makes it visible, and passes control to Window's message loop.

The form class can post a "quit" message to terminate the program.

FormBorderStyle property (enumeration)

0 none	1 FixedSingle	2 Fixed3D
3 FixedDialog	4 Sizable (default)	
5 FixedToolWindow	smaller caption bar, no control box	
6 SizableToolWindow		

# Form's Properties

often set by Designer

Text	title
BackColor	background -- also BackgroundImage
Width, Height	size
Cursor	see Cursors class for static values
StartPosition	Center.Parent, Center.Screen, Manual, WindowsDefaultBounds, WindowsDefaultLocation (default).
Anchor	Bottom, Left, None, Top, Right (Top, Left default)
Bounds	location and size of control
ClientSize	Area of client (! TitleBar and Borders)
ClientRectangle	Area of client (! title, menu, borders, scrollbars)
Icon	image used in task bar and control box
Parent	
Owner	
Location	position of origin

Forms has

```
public delegate void PaintEventHandler(object obj,  
    PaintEventArgs pea);
```

In a Form class add the handler

```
aForm.Paint += new PaintEventHandler(aPainter);
```

write the handler

```
static void aPainter(object obj, PaintEventArgs pea)  
{  
    Graphics g = pea.Graphics;  
    g.DrawString("Hi CS 585!" Font, Brushes.Black, 20, 20);  
}
```

Paint events on open, resize, minimize | restore, exposure

To cause a Paint event use  
`aForm.Invalidate()`

Inheriting is a better approach

## Inherit painting

Inherit Form's OnPaint (a virtual protected method)

OnPaint is called when Control is invalid

The OnPaint method in Control is overridden to call  
HiAgain's OnPaint.

```
class HiAgain : Form {  
    public static void Main() {  
        Application.Run(new HiAgain()); }  
  
    protected override void OnPaint(PaintEventArgs pea) {  
        Graphics g = pea.Graphics;  
        g.DrawString("Hi again", Font, Brushes.Black,  
                    40, 40); }  
}
```

OnPaint is called when client area is invalid

Note use of keyword "**override**" for use when overriding virtual methods.

OnResize method is invoked when the form's window is resized. This method is often overridden

on overridden methods you often need to call the base class's method first, the keyword "**base**" is used (like java's super)

```
protected override void OnResize(EventArgs ea) {  
    base.OnResize(ea);  
    // do class specific resize behaviors  
    ...  
}
```

The property `ResizeRedraw` can be set true to force the entire client area to be invalidated on resize events.

To force repainting of the form after an invalidate call `Update()`.

```
...  
Invalidate();           // posts paint event  
Update();               // executes pending paint events
```

## structs

A value-type object-based abstract type.

can't inherit from any type (implicitly inherits from object)

can't be subclassed.

can't have a default constructor -- one w/ no args

If new isn't called on a struct an instance w/ all fields zeroed is created.

NET Foundation defines a richer Point struct than this example....

```
struct Point {  
    public int x, y;  
  
    public Point (int X, int Y) {  
        x = X; y = Y; }  
  
    public override string ToString() {  
        return(String.Format("{0}, {1}", x, y)); }  
  
}  
  
...  
Point pt = new Point(100, 100);
```

## Useful structures in System.Drawing: Point, Size, Rectangle, Color

Point, a 2D coordinate point, has two properties X and Y

```
Point pt = new Point(30, 20);  
Point pt2 = new Point(); pt2.X = 40; pt2.Y = 60;  
  
Console.WriteLine(pt2); // invokes ToString()  
if (pt == pt2)... // same as if (pt.Equals(pt2)) ...  
  
Point [] pts4 = { new Point(10, 10), new Point(20, 10),  
                 new Point(20, 20), new Point(10, 20);
```

Size has Width and Height properties.

sizes and points can be cast into each other and used to construct each other.

"+" and "-" operators are overridden for both points and sizes

There are also float versions: PointF and.SizeF.

# Rectangle

Rectangle struct is a point (origin) with a size (extent). Also RectangleF

```
Rectangle(Point p, Size s);  
Rectangel(int x, int y, int w, int h);
```

Rectangle properties:

get/set Point, Size, X, Y, Width, Height,

get Left, Top, Right, Bottom, isEmpty

Left is X, Top is Y

Right is X + Width, Bottom is Y + Height

Operators are: == !=

Equals(Rectangle)

# Color

Color struct represents color with ARGB (alpha is transparency level).  
has 141 static color names properties

Pen class used for drawing lines, curves

Brush class used for drawing filled areas and text

**Pens** and **Brushes** classes have 141 static read only color properties that are easier usage than Pen and Brush.

SystemColors class has 26 read only properties for most (26 / 29) user settable colors in the Windows interface.

Window (back), WindowText(fore), Control, ControlText, ...

SystemPens and SystemBrushes have 15 and 21 of these properties

KnownColors enumeration has all color names and system colors

Use SystemColors values if you want user settable interface colors to be used in your application (next slide).

```
class HiYetAgain : Form {  
    public static void Main() {  
        Application.Run(new HiYetAgain()); }  
  
    public HiYetAgain() {  
        ...  
        BackColor = SystemColors.Window;  
        ForeColor = SystemColors.WindowText;  
        ... }  
  
    protected override void OnPaint(PaintEventArgs pea) {  
        Graphics g = pea.Graphics;  
        g.DrawString("Hi yet again", Font,  
            new SolidBrush(ForeColor), 40, 40); }  
    ...  
}
```

## Form and Client

Client area is the internal form area (excludes caption bar, borders, menu bar, scroll bars)

`ClientSize` property returns the `Size` value

`ClientRectangle` property returns a `Rectangle` value

Can't set a property of a property

```
ClientSize.Width += 100;           // doesn't work
ClientSize += new Size(100,0);     // works
```

Drawing in the Client (outside of an `OnPaint`, `PaintHandler` or constructor method)

```
...
Graphics g = CreateGraphics();
// drawing statements here
g.Dispose();
...
```

# Scrolling

ScrollableControl get/set properties

bool AutoScroll, HScroll, VScroll

Size AutoScrollMargin, AutoScrollMinSize, AutoScrollPosition

Scrolling in a panel

Panel is a control used to contain other controls or to paint into

```
public aFormConstructor() {  
    ...  
    AutoScroll = true;  
    ... // create a panel  
    Panel panel = new Panel();  
    panel.Parent = this; // adds the panel to the form  
    panel.Paint += new PaintEventHandler(PanelOnPaint)  
    panel.Size = new Size (...);  
    ... }
```

```
void PanelOnPaint(object obj, PaintEventArgs pea) {  
    ... // do all the painting operations into the panel  
}
```

## Scroll in the Form

In the constructor

- set `AutoScroll` true

- set `AutoScrollMinSize` to the necessary size for all controls

In `OnPaint`

- perform all drawing requests

This is inefficient

- often drawing into an area that is clipped (by the `ClientArea`).

- or, need to calculate what part of form is displayed with `AutoScrollPosition` and then draw only parts visible

- The `ClipRectangle` property of the `PaintEventArgs` returns the smallest `Rectangle` in client area that is the invalid region.