

## Microsoft Visual Studio .NET

(C) Richard R. Eckert

## The Microsoft .NET Framework

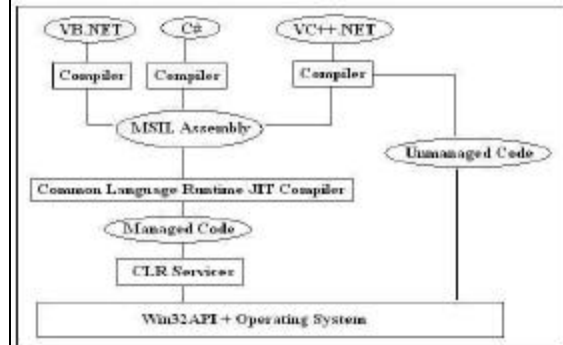
- The Common Language Runtime
- Common Language Specification
  - Programming Languages
    - C#, Visual Basic, C++, lots of others
- Managed Modules (Assemblies)
- MSIL
- The .NET Framework Class Library
  - Namespaces

(C) Richard R. Eckert

## .NET Architecture



## Compilation in the .NET Framework



## Namespaces

- A group of classes and their methods
- FCL is composed of namespaces
- Namespaces are stored in DLL files called assemblies
- Included in a C# program with the *using* keyword
  - If not included, you must give the fully qualified name of any class method or property you use
    - System.Windows.Forms.MessageBox.Show(...)
- Something like packages in Java

(C) Richard R. Eckert

## Some Important .Net Namespaces

- System Core data/auxiliary classes
- System.Collections Resizable arrays + other containers
- System.Data ADO.NET database access classes
- System.Drawing Graphical Output classes (GDI+)
- System.IO Classes for file/stream I/O
- System.Net Classes to wrap network protocols
- System.Threading Classes to create/manage threads
- System.Web HTTP support classes
- System.Web.Services Classes for writing web services
- System.Web.UI Core classes used by ASP.NET
- **System.Windows.Forms** **Classes for Windows GUI apps**
- See online help on 'Class Library'

(C) Richard R. Eckert

## C#

- A new component & object oriented language
  - Emphasis on the use of classes
- Power of C plus ease of use of Visual Basic
- Combines the best aspects of C++ and Java
  - Conceptually simpler and more clear than C++
  - More structured than Visual Basic
  - More powerful than Java
- Syntax very similar to C/C++
  - No header files
- Managed pointers only
  - “Almost no pointers”  $\approx$  “almost no bugs”

(C) Richard R. Eckert

## C# Classes

- Can contain:
  - “Fields”: Data members (like C++ variables)
  - “Methods”: Code members (like C++ functions)
  - “Properties”: In-between members that expose data
    - To user program they look like data fields
    - Within the class they look like code methods
    - Often provide controlled access to private data fields
      - Validity checks can be performed
      - Values can be obtained or set after validity checks
        - » Using Accessor methods `get()` and `set()`
  - “Events”: Define the notifications a class is capable of firing in response to user actions

(C) Richard R. Eckert

## Example: Square class

```
public class Square
{
    private int side_length = 1;           // A Field

    public int Side_length                 // A Property
    {
        get { return side_length; }
        set
        {
            if (value > 0)
                side_length = value;
            else
                throw (new ArgumentOutOfRangeException());
        }
    }

    public int area()                     // A Method
    {
        return (side_length * side_length);
    }

    public Square(int side)               // The Constructor method
    {
        side_length = side;
    }
}
```

(C) Richard R. Eckert

## Instantiating and Using the Square Class

```
Square sq = new Square(10); // Construct a Square object of
                             // side_length = 10
                             // Instantiates the object and invokes
                             // the class constructor
int x = sq.Side_length;     // Retrieve object's Side_Length Property
sq.Side_length = 15;       // Change object's Side_length Property
int sq_area = sq.area();    // Define an integer variable and use
                             // the class area() method to compute
                             // the area of the square
MessageBox.Show("Area=" + sq_area.ToString()); // Display result in a Message Box
                                                // Note use of ToString() method
                                                // to convert an integer to a string.
                                                // Show() is a static method of MessageBox
                                                // class
```



(C) Richard R. Eckert

## Windows Forms

- A Windows Form: just a window
- Forms depend on classes in namespace 'System.Windows.Forms'
- **Form** class is in 'System.Windows.Forms':
  - The heart of every Windows Forms application is a class derived from Form
    - An instance of this derived class represents the application's main window
    - Inherits many properties and methods from Form that determine the look and behavior of the window
      - E.g., Text property to change window's caption
- **Application**: Another important class from 'System.Windows.Forms'
  - Its static method `Run()` drives the Windows Form application
    - Argument is the Form to be run
  - Invoked in the program's entry point function: `Main()`
  - Causes the program to enter the message loop
  - Form passed to `Run()` has code to post a quit message when form is closed
  - Returns to `Main()` when done and program terminates properly

(C) Richard R. Eckert

## A Simple Windows Form App in C# -- HelloWorld

```
using System.Windows.Forms; // the namespace containing
                             // the Form class
public class HelloWorld : System.Windows.Forms.Form
{
    public HelloWorld() // our class derived from Form
    {
        this.Text = "Hello World"; // Set this form's Text Property
    }

    static void Main() // Application's entry point
    {
        Application.Run(new HelloWorld()); // Run our form
    }
}
```

(C) Richard R. Eckert

## Compiling a C# Application from the Command Line

- Start a Command Window with the proper paths to the compiler/linker set
  - Easiest way: From Task Bar:
    - 'Start' | 'All Programs' | 'Microsoft Visual Studio .NET' | 'Visual Studio .NET Tools' | 'Visual Studio .NET Command Prompt'
    - Starts the DOS Box Command Window
  - Navigate to the directory containing the source code file(s)
  - From the command prompt Invoke the C# compiler and linker
  - For example, to build an executable from the C# source file myprog.cs, type one of the following:

```
csc myprog.cs           (easiest way, creates a console app)
csc /target:exe myprog.cs (also creates a console application)
csc /t:winexe myprog.cs  (creates a Windows executable)
csc /t:winexe /r:System.dll,System.Windows.Forms.dll,System.Drawing.dll
myprog.cs                (to provide access to needed .NET DLLs)
```

(C) Richard R. Eckert

## Using Visual Studio to Develop a Simple C# Application “Manually”

- Start Visual Studio as usual
- 'File' | 'New' | 'Project' | 'Visual C# Projects' | 'Empty Project'
- To create the program
  - 'Project' | 'Add New Item'
    - Categories: 'Local Project Items'
    - Templates: 'Code File'
  - This will bring up the code editor
  - Type in or copy and paste the C# source code
- But you must also provide access to some additional .NET Common Language Runtime DLLs
- Do this by adding 'References':
  - 'Project' | 'Add Reference'
  - Select: System.dll and System.Windows.Forms.dll
- Build project as usual ('Build' | 'Build Solution')

(C) Richard R. Eckert

## Using Visual Studio's Designer to Develop a Simple C# Application

- Start Visual Studio as usual
- 'File' | 'New' | 'Project' | 'Visual C# Projects' | 'Windows Application'
  - Gives a “designer view” of the Windows Form the project will create
  - Also skeleton code: Right click on form & select 'View Code' to see it
    - Note how it's broken up into 'Regions' (+ and - boxes on the left)
    - These can be expanded and contracted
    - Expand the 'Windows Form Designer generated code' Region
      - Note the Form properties that have been preset
      - Change the 'Text' property to 'This is a Test'
    - Reactivate the Designer View by clicking on the 'Form1.cs [design]'
    - Note how the caption of the form has changed
    - Look at the 'Properties' window
    - Find the 'Text' Property and change it by Typing 'Hello World'
    - Resize the form (drag its corners) – note how the Size property changes
    - Change the Background Color in the Properties Box to red:
      - Click on BackColor | down arrow | "custom" tab | red color box
    - Go back to 'Code View' and note changes that have been made
  - Build and run the app

(C) Richard R. Eckert

## .NET Managed Modules (Assemblies)

- The result of building a program with any of the compilers capable of generating MSIL
  - Microsoft provides: C#, J#, Visual Basic, Managed C++, Jscript
  - Also ILASM (Intermediate Language Assembler)
  - Third parties provide other compilers that generate MSIL
- 'Executables' (assemblies) designed to be run by the CLR
- Contain 4 important elements stored in the “Manifest”:
  - A Windows Portable Executable (PE) file header
  - A CLR header containing important information about the module
  - Metadata describing everything inside the module and its external dependencies
    - Means every managed module is “self describing”
    - One of the keys to language interoperability
  - The MSIL instructions generated from the source code
- Can examine Assemblies with a tool called ILDASM

(C) Richard R. Eckert

## The ILDASM Disassembler

- Used to examine an assembly's metadata and code
- Start a Command Window with proper path to ILDASM set
  - Easiest way: From Task Bar:
    - 'Start' | 'All Programs' | 'Microsoft Visual Studio .NET' | 'Visual Studio .NET Tools'
    - Starts the DOS Box Command Window
  - Navigate to the directory containing the assembly (.exe)
  - Invoke ILDASM
    - e.g., for HelloWorld program:

```
ILDASM HelloWorld.exe
```
    - Displays a window showing the assembly's Manifest and the classes in the assembly

(C) Richard R. Eckert

## A Session with ILDASM

- Double Click on 'Manifest'
  - List of assemblies that module depends on
  - Assembly name
  - Modules that make up the assembly
    - Because HelloWorld is a single-file assembly, there is only one
- Expand HelloWorld class
  - Class contains two methods:
    - A constructor (.ctor)
    - Main ('S' means it's a static method)
  - Expand Main
    - .entrypoint a directive indicating it's where execution starts
    - Code instantiates a HelloWorld object and calls Application.Run for the form
  - Expand .ctor
    - Calls parent Form's constructor
    - Puts "Hello World" string on stack and calls set\_Text to set the form's Text property

(C) Richard R. Eckert

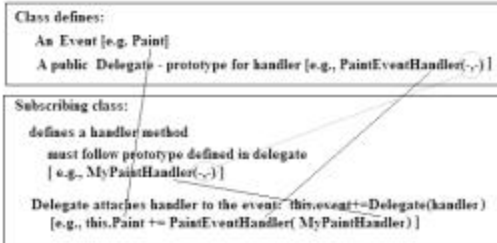
## Events, Delegates, and Handlers

- **Events:** Results of user actions
- But in .NET events are also "class notifications"
- Classes define and publish a set of events that other classes can subscribe to
  - When an object changes its state (the event occurs), all other objects that subscribe to the event are notified
- Events are processed by event **handler methods**
- The arguments to an event handler must match those of a function prototype definition called a **delegate**
  - A method to whom event handling is delegated
  - A type-safe wrapper around a callback function
  - Can be thought of as a managed (safe) function pointer
    - Not a raw memory address, but wraps the function's address
  - Helps avoid program crashes when the function is called back
  - Permits any number of handler methods for a given event

(C) Richard R. Eckert

## Events, Delegates, Handlers

### Events, Delegates, and Handlers in .NET



(C) Richard R. Eckert

## An Example – Handling a Paint Event

- **Form** class has a **Paint** event to handle window exposures
- The delegate is **PaintEventHandler**, defined as:  

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

  - First argument: sender object (where event occurred)
  - Second argument: provides event data
    - A class with properties 'Graphics' and 'ClipRectangle'
      - Graphics contains instantiation of **Graphics** class (GDI+)
        - » The class used to draw on a form (like a Device Context)
      - ClipRectangle: Specifies area of window that needs to be redrawn
- Any Paint handler method must have these arguments
- And the Paint handler must be "attached" to the Paint event of the Form class (i.e., **delegated to the handler**)

(C) Richard R. Eckert

## Defining the Event Handler and Attaching it to the Event

- Defining the form's Paint event handler method:  

```
private void MyPaintHandler(object objSender, PaintEventArgs pea)
{
    // event handling code goes here
};
```
- Attaching the handler to the form's Event (delegating it to the event handler):  

```
form.Paint += new PaintEventHandler(MyPaintHandler);
```
- A handler can also be "detached" from an event:  

```
object.event -= new delegate(method);
```

(C) Richard R. Eckert

## Drawing Text in response to a Paint Event

- **Drawing** namespace contains many classes and structures for drawing on a window
- Some of them:
  - Bitmap, Brush, Brushes, Color, Font, **Graphics**, Icon, Image, Pen, Pens, Point, Rectangle, Size
  - See online help: 'ClassName class' 'all members'
- **Graphics** Class
  - Represents a GDI+ drawing surface
    - Like a device context
  - Contains many graphics drawing methods
    - See Help on 'Graphics class', 'all members'
  - Obtaining a graphics object:
    - In Paint event handler, use second argument:
      - PaintEventArgs is a Graphics object
      - Code: Graphics g = pea.Graphics

(C) Richard R. Eckert

## Using DrawString() to Draw Text

- **Graphics.DrawString()** has lots of overloaded versions
- Simplest:  

```
DrawString(string str, Font font, Brush brush, float x, float y);
```

  - string class: an alias for System.String
    - Defines a character string
    - Has many methods to manipulate a string
  - Font class: gives a Windows Form program access to many fonts with scalable sizes
    - A Form has a default Font: It's one of the Form's properties
    - Or you can instantiate a new Font object: Lots of possibilities (we'll see later)
  - Brush or Brushes class: color/style of characters
    - Lots of different color properties, e.g. Brushes.Black
    - Or can create one of a specified Color
      - Brush br = new SolidBrush(Color.FromArgb(r,g,b));
      - Brush br = new SolidBrush(Color.Red);
  - x,y : Location to draw string on window client area

(C) Richard R. Eckert

### **Hello\_in\_window Example Program**

- Responds to Paint Event by displaying 'Hello World' in window's client area using several different Brushes
- Manual Project
  - Define Handler and Attach it to Paint event manually
- Designer Project
  - Select the Paint event in the form's Properties window
    - Click on lightning bolt
  - Attachment of handler done automatically
  - Skeleton handler code generated automatically

(C) Richard R. Eckert

### **An Alternative to Installing Event Handlers**

- In any class derived from 'Control' (e.g. 'Form') its protected OnPaint() and other event handlers can be overridden:

```
protected override void OnPaint(PaintEventArgs pea)
{
    // Painting code goes here
};
```

  - Avoids having to attach the handler to the event
- See HelloWorld\_override example program

(C) Richard R. Eckert

### **A Separate Class for Main()**

- An alternative way of organizing a Windows Form application:
  - Define the Form in one class
  - Place the Main() function in another class
  - Must be done manually
    - Designer gives the single class program template
  - See SeparateMain1 example program

(C) Richard R. Eckert

### **Inheriting Form Classes**

- Just as your Form inherits from 'System.Windows.Forms.Form', you can set up a new Form that inherits from a previously defined Form
- Be sure its Main() includes keyword 'new'
- And that Visual Studio knows which class' Main() is the entry point:
  - In project's Properties box select 'Property Pages' icon
    - 'Common Properties' | 'General' | Application' | 'Startup Object'
    - Select 'InheritHelloWorld'
- See HelloWorld\_inherit example

(C) Richard R. Eckert

### **Multiple Handlers**

- An advantage of the delegate mechanism is that multiple handlers of the same event can be used
- Just attach each handler to the event
  - For example:

```
Form.Paint += new PaintEventHandler(PaintHandler1);
Form.Paint += new PaintEventHandler(PaintHandler2);
```
- And then write the handlers
- Each time the event occurs, all handlers will be called in sequence
- See TwoPaintHandlers example

(C) Richard R. Eckert

### **Some other GDI+ Drawing Methods**

- DrawArc( );
- DrawEllipse( );
- DrawLine( );
- DrawPolygon( );
- DrawRectangle( );
- FillEllipse( );
- FillPolygon( );
- FillRectangle( );
- Lots of others in 'Graphics' class
  - See online help on various overloaded forms of calling these functions

(C) Richard R. Eckert

### Random Rectangles Example Program

- Makes use of FillRectangle() GDI+ method
- 'Random' class contains many methods to generate random numbers

Random r = new Random();

- Instantiates a new Random object and seeds the pseudo-random number generator

- The 'Next()' method actually generates the number
  - Many overloaded forms of Next()

- Getting a random color:

Color c = Color.FromArgb(r.Next(256), r.Next(256), r.Next(256));

- Use Form's ClientSize Property to get width and height of window
- Draw filled rectangle with random size and color:
  - Use FillRectangle() and Math.Min(), Math.Abs()

(C) Richard R. Eckert