Microsoft Visual Studio 2005/2008 and the .NET Framework

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

(C) Richard R. Eckert
.NET Architecture

Compilation in the .NET Framework
Namespace

- A collection of related classes and their methods
- FCL is composed of namespaces
- Namespaces are stored in DLL assembly files
- .NET applications must have “references” to these DLLs so that their code can be linked in
- Also should be included in a C# program with the `using` declaration
  - e.g. `using System.Windows.Forms;`
  - If left out, you must give the fully qualified name of any class method or property you use, e.g. `System.Windows.Forms.MessageBox.Show(...);`
- Something like a package in Java

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

- See online help on ‘Class Library’
C#

- A new component & object oriented language
  - Emphasis on the use of classes
- Power of C++ and 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” ⇔ “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 the user program they look like data fields
    - Within the class they look like code methods
    - Often they provide controlled access to private data fields
      - Validity checks can be performed
      - Values can be obtained or set after validity checks
        - Properties use Accessor methods `get()` and `set()`
        - `get()` to retrieve the value of a data field … `return` *data-field*;
        - `set()` to change the value of a data field … `data-field =` *value*;
      - Other classes use Properties just like data fields
  - **Events**: Define the notifications a class is capable of firing in response to user actions

(C) Richard R. Eckert
Example: Square class

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

    public int Side_length // A Property
    {
        get { return side_length; } // "return": specifies value going out

        set
        {
            if (value>0)
                side_length = value; // "value": specifies value that came in
            else
                throw (new ArgumentOutOfRangeException());
        }
    }

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

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

Instantiating and Using the Square Class

```csharp
Square sq = new Square(10); // Construct a Square object called sq
// 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: In .NET it’s just a window
- Forms depend on classes in the 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 the 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 create the form passed to it and enter the message loop
    - Implies form’s constructor will run (typically code to set initial window properties)
  - The form passed to Run() has code to post a QUIT message when form is closed
  - Returns to Main() when done and program terminates properly

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
{                   // our class is derived from Form
    public HelloWorld() // our class constructor
    {
        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:
    • 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)
    (to provide access to needed .NET DLLs)

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

• Start Visual Studio as usual
• To create the program
  – ‘Project’ | ‘Add New Item’
    • Visual Studio installed templates: ‘C# 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’ … ‘.NET tab
  – Select: System and System.Windows.Forms
• 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
  – 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
  – This is only part of the code
  – To see code generated by the Visual Studio designer:
    • In Solution Explorer, expand Form1.cs & double click on Form1.Designer.cs
    • Expand the ‘Windows Form Designer generated code’ Region

Where is Main()? 

- Go to Class View and expand the project’s classes
  – Note that there are two classes: the Form and the Program
  – Expand the Program class
    • That is where Main() is
    • It runs the Form just as in our manual code
Changing Form Properties

- In Form1.Designer.cs, note the Form’s properties that have been preset
  - Change code so the ‘Text’ property is “This is a Test”
- Reactivate the Designer View by clicking on the ‘Form1.cs [design]’ tab
  - 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’
  - Activate Form1.Designer.cs and note how code has changed
- In Designer View resize the form (drag its corners)
  - note how the ClientSize property changes in Form1.Designer.cs code
- Change the Background Color in the Properties Box to red:
  - Click on ‘BackColor’ | down arrow | “custom” tab | red color box
  - Go back to Form1.Designer.cs and note changes in code
- Build and run the application

.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 ILDASMDM
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 managed pointer to a function
  - A type-safe wrapper around an event handler callback function
    - Handler function must use parameters specified in delegate args
  - “Attaches” the handler function to the event
  - Permits any number of handler methods for a given event

(C) Richard R. Eckert
Events and Delegates

Event-Generating Object

Event-Consuming Object

Event-Handler

How does this object know who to notify?

These objects must register for the event

(C) Richard R. Eckert
Event-Handling Model

Objects A raises event E  
\[ \text{Delegate for event E} \]
\[ \text{Handler 1 for event E} \]
\[ \text{Handler 2 for event E} \]
\[ \text{Handler 3 for event E} \]

Events, Delegates, Handlers

Events, Delegates, and Handlers in .NET

Class defines:
- An Event [e.g., Paint]
- A public Delegate - prototype for handler [e.g., PaintEventHandler(\( \_\_\_\_\_ \))]

Subscribing class:
- defines a handler method
- must follow prototype defined in delegate [e.g., MyPaintHandler(\( \_\_\_\_\_ \))]

Delegate attaches handler to the event: this.event+=Delegate(handler)  
[e.g., this.Paint+=PaintEventHandler(MyPaintHandler)]
An Example – Handling a Paint Event

- Form class has a Paint event to notify of window exposures
- The delegate is PaintEventHandler, defined as:
  ```csharp
  public delegate void PaintEventHandler(object objSender, PaintEventArgs pea);
  ```
  - First argument: sender “object” (where event occurred)
  - Second argument “PaintEventArgs”: provides event data
    - A class with properties ‘Graphics’ and ‘ClipRectangle’
      - ‘Graphics’ property: contains an instantiation of the Graphics class (GDI+)
        » The class is used to draw on a form (like a Device Context)
      - ClipRectangle: Specifies the area of the 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)

---

Defining the Paint Event Handler and Attaching it to the Event

- Defining the form’s Paint event handler method:
  ```csharp
  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):
  ```csharp
  form.Paint += new PaintEventHandler (MyPaintHandler );
  ```
  - From now on MyPaintHandler(-,-) will be called any time the Paint event occurs

- A handler can also be “detached” from an event:
  ```csharp
  object.event -= new delegate(method);
  ```

(C) Richard R. Eckert
Drawing Text in Response to a Paint Event

- **System.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
- **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 pea` provides a Graphics object
      - Get it with following code: `Graphics g = pea.Graphics`

Using DrawString() to Draw Text

- Graphics DrawString() method has lots of overloads
- Simplest:
  - `DrawString(string str, Font font, Brush brush, float x, float y);`
  - string class: an alias for System.String
    - Defines a character string
    - Also 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 static color properties, e.g.
      - `Brushes.Black`, `Brushes.Red`
    - Or we can create one of a specified Color
      - `Brush br = new SolidBrush(Color.FromArgb(r,g,b));`
      - `Brush br = new SolidBrush(Color.Red);`
    - Color structure has many static methods and properties
  - `x,y : Location to draw string on window client area`
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
    • Double click on “Paint” event
  – Attachment of handler using its delegate is done automatically
  – Skeleton handler code generated automatically

(C) Richard R. Eckert

An Alternative to Installing Event Handlers: Overriding instead of Attaching

• 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 using the delegate

• 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
  - See SeparateMain example program

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:
    ```csharp
    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

---

**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

---

(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()