C# Programming in Depth

Prof. Dr. Bertrand Meyer

March 2007 – May 2007

Lecture 5: Delegates, Events and GUI

Lisa (Ling) Liu
Overview

- Delegates
- Events
- GUI
Delegate

- A delegate is a type-safe object that points to another (or possibly multiply methods) in the application, which can be invoked at a late time.

- A delegate type maintains three pieces of information:
  - The *name* of the method which it makes calls
  - The *argument* (if any) of this method
  - The *return value* (if any) of this method
Defining a delegate in C#

- A delegate type should be defined to match the signature of the method it points to.

  ```csharp
  public delegate int BinaryOp (int x, int y);
  ```

- A C# delegate definition results in a sealed class with three compiler-generated methods whose parameter and return types are based on the delegate’s declaration.

  ```csharp
  //pseudo-code behind
  public sealed class DelegateName : System.MulticastDelegate
  {
      public DelegateName (object target, uint functionAddress);
      public delegateReturnValue Invoke (allDelegateInputParams);
      public IAsyncResult BeginInvoke (allDelegateInputRefOutParams,
                                      AsynCallback cb, object state);
      public delegateReturnValue EndInvoke (allDelegateRefOutParams,
                                             IAsyncResult result);
  }
  ```
public delegate int BinaryOp (int x, int y);
public class SimpleMath
{
    public static int Add (int x, int y)
    {
        return x+y;
    }
    public int Subtract (int x, int y)
    {
        return x – y;
    }
}

static void Main ()
{
    int result;
    BinaryOp b = new BinaryOp(SimpleMath.Add);
    result = b(10, 10);
}
Investigate a delegate object

C# programming lecture 5: Delegates, Events and GUI
Event

- **Events** provide a way for a class or object to notify other classes or objects when something of interest happens.

- The class that sends (or *raises*) the event is called the *publisher* and the classes that receive (or *handle*) the event are called *subscribers*. 
Properties of events

- The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
- An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
- Events that have no subscribers are never called.
- Events are commonly used to signal user actions such as button clicks or menu selections in graphical user interfaces.
- When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised.
- Events can be used to synchronize threads.
- In the .NET Framework class library, events are based on the EventHandler delegate and the EventArgs base class.
Defining a event

1. Define a delegate that points to the method to be called when the event is fired

   public delegate void CarEventHandler(string msg);

2. Declare events in terms of the related delegate

   public event CarEventHandler Exploded;
Send event

Call the method (event handler) pointed by the event.

```
public void SpeedUp (int delta)
{
    // If the car is dead, fire Exploded event.
    if (carIsDead)
    {
        if (Exploded != null)
            Exploded("Sorry, this car is dead...");
    }
}
```
Register event handler

```csharp
Car.CarEventHandler d = new Car.CarEventHandler(CarExploded);
c1.Exploded += d;

public static void CarExploded (string msg)
{
    Console.WriteLine(msg);
}
```
Building a Main Window

- Derive a new class from System.Windows.Forms.Form

- Configure your application’s Main() method to invoke Application.Run(), passing an instance of your Form-derived type as an argument
Event-driven applications

- Idea is very simple:
  - individual user actions are translated into "events"
  - events are passed, one by one, to application for processing

- this is how most GUIs are programmed...
GUI-based events

- Mouse move
- Mouse click
- Mouse double-click
- Key press
- Button click
- Menu selection
- Change in focus
- Window activation
- etc.
Events are handled by methods that live behind visual interface
- known as "code-behind"
- our job is to program these methods...
Call-backs

- Events are a call from object back to us...
- How is connection made?
  - setup by code auto-generated by Visual Studio
Example: a windowing application

- GUI apps are based on the notion of forms and controls...
  - a form represents a window
  - a form contains 0 or more controls
  - a control interacts with the user

- Let's create a GUI app in a series of steps...
Step 1

- Create a new project of type "Windows Application"
  - a form will be created for you automatically...
Step 2 — GUI design

- Select desired controls from toolbox...
  - hover mouse over toolbox to reveal
  - drag-and-drop onto form
  - position and resize control
GUI design cont’d…

- A simple calculator:

- Position and configure controls
  - click to select
  - set properties via Properties window
Step 3 — code design

- “Code behind” the form...
- Double-click the control you want to program
  ➢ reveals coding window

```csharp
// Code for button click event
private void button1_Click(object sender, System.EventArgs e)
{
    int i, j, k;
    i = System.Convert.ToInt32(this.textBox1.Text);
    j = System.Convert.ToInt32(this.textBox2.Text);
    k = i + j;
    System.Windows.Forms.MessageBox.Show("Sum = " + k);
}
```
Step 4 — run mode

- Run!
Break mode?

Easily triggered in this application via invalid input...

```csharp
private void button1_Click(object sender, System.EventArgs e)
{
    int i, j, k;
    i = System.Convert.ToInt32(this.textBox1.Text);
    j = System.Convert.ToInt32(this.textBox2.Text);
    k = i + j;
    System.Windows.Forms.MessageBox.Show("Sum = " + k);
}
```
WinForms

- Another name for traditional, Windows-like GUI applications
  - vs. WebForms, which are web-based

- Implemented using FCL (Framework Class Library)
  - hence portable to any .NET platform
Abstraction

- FCL acts as a layer of abstraction
  - separates WinForm app from underlying platform
Form properties

- Form properties typically control visual appearance:
  - AutoScroll
  - BackgroundImage
  - ControlBox
  - FormBorderStyle (sizable?)
  - Icon
  - Location
  - Size
  - StartPosition
  - Text (i.e. window's caption)
  - WindowState (minimized, maximized, normal)

```csharp
Form1 form;
form = new Form1();
form.WindowState = FormWindowState.Maximized;
form.Show();
```
Form methods

- **Actions you can perform on a form:**
  - **Activate:** give this form the focus
  - **Close:** close & release associated resources
  - **Hide:** hide, but retain resources to show form later
  - **Refresh:** redraw
  - **Show:** make form visible on the screen, & activate
  - **ShowDialog:** show modally

```csharp
form.Hide();
form.Show();
```
Form events

- Events you can respond to:
  - bring up properties window
  - double-click on event name

Load: occurs just before form is shown for first time
Closing: occurs as form is being closed (ability to cancel)
Closed: occurs as form is definitely being closed
Resize: occurs after user resizes form
Click: occurs when user clicks on form's background
KeyPress: occurs when form has focus & user presses key
private void Form1_Closing(object sender, 
    System.ComponentModel.CancelEventArgs e) 
{
    DialogResult r;

    r = MessageBox.Show("Do you really want to close?", 
                        "MyApp", 
                        MessageBoxButton.YesNo, 
                        MessageBoxIcon.Question, 
                        MessageBoxDefaultButton.Button1);

    if (r == DialogResult.No)
        e.Cancel = true;
}
Controls

- User-interface objects on the form:
  - labels
  - buttons
  - text boxes
  - menus
  - list & combo boxes
  - option buttons
  - check boxes
  - etc.
Abstraction

- Like forms, controls are based on classes in the FCL:
  - `System.Windows.Forms.Label`
  - `System.Windows.Forms.TextBox`
  - etc.

- Controls are instances of these classes
Who creates all these objects?

- Who is responsible for creating control instances?
  - code is auto-generated by Visual Studio
  - when form object is created, controls are then created…
Naming conventions

- Set control's name via Name property
- A common naming scheme is based on prefixes:
  - `cmdOK` refers to a command button control
  - `lstNames` refers to a list box control
  - `txtFirstName` refers to a text box control
Labels

- For static display of text
  - used to label other things on the form
  - used to display read-only results

- Interesting properties:
  - Text: what user sees
  - Font: how he/she sees it
Command buttons

- For the user to click & perform a task

- Interesting properties:
  - Text: what user sees
  - Font: how he/she sees it
  - Enabled: can it be clicked

- Interesting events:
  - Click: occurs when button is "pressed"
Text boxes

- **Most commonly used control!**
  - for displaying text
  - for data entry

- **Lots of interesting features...**
Text box properties

- **Basic properties:**
  - **Text:** denotes the entire contents of text box (a string)
  - **Modified:** has text been modified by user? (True / False)
  - **ReadOnly:** set if you want user to view text, but not modify

- **Do you want multi-line text boxes?**
  - **MultiLine:** True allows multiple lines of text
  - **Lines:** array of strings, one for each line in text box
  - **ScrollBars:** none, horizontal, vertical, or both
Text box events

- **Interesting events:**
  - Enter, Leave: occurs on change in focus
  - KeyPress: occurs on ascii keypress
  - KeyDown, KeyUp: occurs on any key combination
  - TextChanged: occurs whenever text is modified

- **Validating and Validated**
  - Validating gives you a chance to cancel on invalid input
List Boxes

- Great for displaying / maintaining list of data
  - list of strings
  - list of objects (list box calls ToString() to display)

```csharp
Customer[] customers;

// create & fill array with objects...

// display customers in list box
foreach (Customer c in customers)
    this.listBox1.Items.Add(c);

// display name of selected customer (if any)
Customer c = (Customer) this.listBox1.SelectedItem;
if (c == null)
    return;
else
    MessageBox.Show(c.Name);
```
Just the tip of the iceberg...

- Menus, dialogs, toolbars, etc.

- Thousands of additional controls
  - .NET and ActiveX
  - right-click on Toolbox
  - "Customize Toolbox"
Summary

- Event-driven programming is very intuitive for GUI apps

- Forms are the first step in GUI design
  - each form represents a window on the screen
  - form designer enables drag-and-drop GUI construction
  - Users interact primarily with form's controls
  - labels, text boxes, buttons, etc.
  - implies that GUI programming is control programming
Questions?