

Windows Forms

C# Programming

January 10

Part I

Windows Forms

Event-Driven Model

- The `System.Windows.Forms` library contains controls for building GUI applications
- The actual window for a GUI application is an instance of `System.Windows.Forms.Form`
- Controls (such as buttons, text boxes, file menus) get added to the form, where they sit and wait for events (such as the user clicking them or mousing over them)
- Each control responds to various user and other inputs (such as the timer tick of an internal clock) by **raising events**
- **Event handlers** can be attached to these events so that they are executed when they are raised
- **Delegates** are the means by which we can attach handlers to events

Delegates

- You can think of a delegate as a type-safe function pointer
- I.e., a variable that holds a function of a particular return type and parameter list
- Delegates decouple the class that declares the delegate from the class that uses the delegate
- After a delegate has been defined, it can be instantiated, assigned to, and then the contents of this instance (a reference to a function) can be invoked

```
using System;
namespace FirstDelegate
{
    public delegate void MyDel(int a, int b);

    class Program {
        private static MyDel someFunction;
        private static void PrintSum(int x, int y)
            { Console.WriteLine(x+y); }
        static void Main(string[] args) {
            someFunction = PrintSum;
            someFunction(2, 3);
        }
    }
}
```

Multicasting

- Oftentimes you want to attach several functions to an event (sometimes said *subscribe* several functions to an event)
- You can attach (and detach) additional functions to an instance of a delegate using +, -, +=, and -=

```
static void Main(string[] args) {  
    someFunction = PrintSum;  
    someFunction += PrintSum;  
    someFunction(2, 3);  
}
```

Hello GUI World

[demo]

Adding Controls Programmatically

- Although the Form Designer is great for rapid development and prototyping, organizing the UI solely through it can become messy
- You can, of course, explicitly write the code to create GUI elements and modify their properties

```
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 3; j++) {  
        Button b = new Button();  
        b.Size = new Size(40, 40);  
        b.Location = new Point(i * 40, j * 40);  
        this.Controls.Add(b);  
    }  
}
```

Custom Controls

- To customize the behavior a regular control, you can extend that control class
- To create a custom control, create a new **Windows Control Library** project
- By default, the class (with default name `UserControl1`) will be declared to extend `UserControl`
- Instead, choose the type of control you want to extend (eg, `Button`, `TextBox`, `Label`, etc)
- You will then have to delete a line assigning to the `AutoScaleMode` property, which `UserControl` has but some other controls do not
- Add whatever custom event handlers and logic you need to the class definition

Custom Controls

- Once you compile the custom control project, you can use it from a Windows Forms application
- To be able to access the custom control class from your project, you need to add a reference to it
- In the Solution Explorer, right-click the project name (or the References folder underneath it), and select **Add Reference**
- Locate the custom control project folder, and then select the .dll file (this should be in the bin subdirectory)
- Once the reference has been added, the namespace of the custom control project (and all its classes, etc.) will be in scope

Custom Controls

- You can also add a custom control to the Form Designer's Toolbox
- Right-click anywhere in the Toolbox and select **Choose Items**
- It takes a long time for this dialog box to appear
- The first time you select Choose Items, it takes a *really* long time, so be patient
- Locate the .dll for your custom control
- An entry for the custom control will be added to the Toolbox, and you can drag instances of it onto a form

Part II

Separation of UI and Logic

Partial Classes

- The Form Designer generates the plumbing code to set up the window and its controls
- This can often grow large and clutter the rest of your code
- To remedy this situation, **partial classes** were introduced in C# 2.0 to allow a class definition to span multiple files
- For a form class `Form1`, the automatically generated code resides in `Form1.Designer.cs`
- Note: you should not modify this file!
- The rest of your own logic can go in a separate file, likely `Form1.cs`, apart from the messy GUI setup code

Separation of UI and Logic

- You must be careful to organize your own logic in an intelligent way
- The Form Designer is convenient in that it automatically generates stubs for event handlers
- It is *not* a good style, however, to put complex logic in these handlers
- If in the future you decide to reorganize your GUI and change the way the user interacts with it, you would not want to do a lot of copying and pasting of logic in the event handlers
- Instead, you want event handlers to be fairly straightforward calls to functions that take care of the heavy duty work

Finite State Diagrams

- The complexity of the state of a GUI application can grow quickly, even with a seemingly small number of tasks an application needs to complete
- The logic for maintaining state and appropriately responding to events fired can easily become convoluted
- It is a good idea to think carefully about all possible states your GUI application can be in and all the events that can be raised in each
- Creating a **finite state diagram** before you begin coding will make the developing and maintaining the code much easier
- Getting into this good habit early will make development of medium and large scale projects much more efficient and effective

Traffic Light Example

[demo]