Properties, Events and Delegates

Taking the magic out of GUI programming

January 28, 2008

Recall last lecture:

```
// Create a form (i.e. a window)
Form theForm = new Form();

// Set the title
theForm.Text = "My Window";
```

Recall last lecture:

```
// Create a form (i.e. a window)
Form theForm = new Form();

// Set the title
theForm.Text = "My Window";
```

Q. Is theForm.Text really a member?

Recall last lecture:

```
// Create a form (i.e. a window)
Form theForm = new Form();

// Set the title
theForm.Text = "My Window";
```

Q. Is theForm.Text really a member?
A. No. theForm.Text is a *property*.

Properties provide special syntax for common methods.

- A property consists of two methods: get and set.
- Clients call set with assignment notation
  e.g. theForm.Text = "My␣Window";
- Clients call get with member read notation
  e.g. WriteLine(theForm.Text)
- Each property access runs a method.

Property Example (I/III)

```
public class Temperature{
  private double myKelvin;

  public double Kelvin{
    get{
      //Think  "public double get()"
      return myKelvin;
    }
    set{
      //Think  "public void set(double value)"
      myKelvin = value;
    }
  }
  ...
```

Property Example (II/III)

```
    ...
    public double Fahrenheit{
      get{
          return myKelvin*(9.0/5.0) - 459.67;
      }
      set{
         myKelvin = (5.0/9.0)*(value + 459.67);
      }
    }
}
```

Property Example (III/III)

```
public class Runner{
  public static void Main(string[] args)
  {
    Temperature Temp = new Temperature();
    Temp.Fahrenheit = 32.0;
    Console.Out.WriteLine(Temp.Kelvin);
  }
}
```

Output: 273.15 (that's the right answer)

C# 3.0 has special syntax for declaring simple properties.

```
public class Temperature{

  // Compiler automatically generates private
  // member, getter, and setter
  public double Kelvin { get; set; }

  public double Fahrenheit{
    get{
      return Kelvin*(9.0/5.0) - 459.67;
    }
    set{
     Kelvin = (5.0/9.0)*(value + 459.67);
    }
  }
}
```

Access limited properties.

```
public class Misc {
  int myNumber;

  // A property with a private getter.  Only
  // members of Misc can read .DropBox
  public int DropBox {
    set{
      myNumber = value;
    }
    private get{
      return myNumber;
    }
  }

  public int PrivateSet { get; private set; }
}
```

Read-only and write-only properties

```
public class GetSetOnly{
  private int myX, myY;

  // A read-only property: a common pattern
  public int X { get { return myX; } }

  // Write only patterns are considered bad style
  public int Y { set { myY = value; } }
}
```

Indexers simulate array access to a class.

```
public class BinarySearchTree<A> {
  private A Lookup(int i) { ... }

  private void SetAt(int i, A data) { ... }

  //access might throw ArgumentOutOfRangeException
  public A this[int index] {
    get { return this.Lookup(index); }
    set { this.SetAt(index, value); }
  }
}
```

Technical notes about properties

- Properties compile to method calls, not field access
- So properties can't implement fields in interfaces
- Properties are optimized to be roughly as fast as field access

When should I use a . . .

- . . . public member?
- . . . property?
- . . . indexer
- . . . method?

When should I use a . . .

- . . . public member?
  Only in trivial situations. Public members are not robust against design changes.
- . . . property?
- . . . indexer
- . . . method?

When should I use a . . .

- . . . public member?
  Only in trivial situations. Public members are not robust against design changes.
- . . . property?
  - The getter has no (observable) side effects.
  - The getter does not throw exceptions.
  - Both get and set return almost immediately (no long computations or database queries)
- . . . indexer
- . . . method?

When should I use a . . .

- . . . public member?
  Only in trivial situations. Public members are not robust against design changes.
- . . . property?
  - The getter has no (observable) side effects.
  - The getter does not throw exceptions.
  - Both get and set return almost immediately (no long computations or database queries)
- . . . indexer
  - The indexer implements an array abstraction
  - The indexer returns almost immediately
  - The indexer only raises ArugmentOutOfRangeException
- . . . method?

When should I use a . . .

- . . . public member?
  Only in trivial situations. Public members are not robust against design changes.
- . . . property?
  - The getter has no (observable) side effects.
  - The getter does not throw exceptions.
  - Both get and set return almost immediately (no long computations or database queries)
- . . . indexer
  - The indexer implements an array abstraction
  - The indexer returns almost immediately
  - The indexer only raises ArugmentOutOfRangeException
- . . . method? Any other time.

Event Driven Programming

- Imagine a game with many actors responding to their environment.

Event Driven Programming

- Imagine a game with many actors responding to their environment.
- Polling: Every so often, each actor looks at state of environment and takes appropriate actions.

Event Driven Programming

- Imagine a game with many actors responding to their environment.
- Polling: Every so often, each actor looks at state of environment and takes appropriate actions.
- Events: Wake up actors when something interesting happens.

We can code events using basic C#...

```
public interface IEventHandler{ void DoIt(); }

public class SesameStreet{
  void RegisterForCookie(IEventHandler h){...}
  void CookieEventHappens(){...}
}

public class CookieMonster{
  class CookieEatingClass : IEventHandler{
      void DoIt() { WriteLine("Nom, Nom"); }
  }

  CookieMonster(SesameStreet s){
      s.RegisterForCookie(new CookieEatingClass());
  }
}
```

. . . but the encoding is flawed.

Problems:

- A nested class is needed to define each event handler.
- Handler has not easy access instance and local variables.
- Resulting code is hard to read.

Delegates: methods as data.

```
// Declare a new delegate type.  A binOp is a
// method that takes two ints and returns an int.
public delegate int binOp(int x, int y);

public class Demo{

    static void Main(string[] args){
        // m is stores a binOp
        binOp m = Math.Min;

        // Calling m calls the stored method,
        // Math.Min. Output is "3".
        Console.WriteLine(m(3,4));
    }
}
```

Multicasting: A delegate can call several methods. (I)

```
public delegate void Printer(string s);

public class PromptPrinter{
    private string prompt;
    public PromptPrinter(string p){ prompt=p; }
    public void Print(string s){
        Console.WriteLine(prompt + s);}
}
```

Multicasting: A delegate can call several methods. (II)

```
public class Demo{
    static Printer myPrinter;

    static void Main(string[] args){
        PromptPrinter p1 = new PromptPrinter(">>");
        PromptPrinter p2 = new PromptPrinter("#");
        myPrinter = p1.Print;
        myPrinter += p2.Print;
        myPrinter("foo");
    }
}
```

- Output is ">>foo" "#foo"
- Multicasting only makes sense for methods returning void.
- Operators =, +, -, +=, -= attach and detach delegates.

Anonymous delegates further streamline event code.

```
public delegate int binOp(int x, int y);
   ...
// C# 2.0 "Anonymous Delegate" Syntax:
binOp sum =
  delegate(int x, int y) {
    return x + y; };

// C# 3.0 "Lambda" Syntax
// (plus type inference):
binOp sum = ((x, y) => x + y );
```

How does it work?

- C# compilation translates delegate types into classes which inherit from System.MulticastDelegate.
- Delegate values are compiled to class instances.
- For multicasting, + operator builds a list of delegates objects.

Events are delegates of a standardized type.

```
public delegate HandlerType(object caller,
                            EventArgs e);

class foo{
  public event HandlerType myEvent
}
```

- Field myEvent can be updated (+=, -=) as public.
- But, the delegate stored in myEvent can only be invoked by foo
  - compiler actually makes myEvent private
  - public methods foo.add_myEvent and foo.remove_myEvent manipulate myEvent field
  - Operator syntax (+=, -=) is used to call above methods.
- By convention, foo should pass itself as caller.

Updating the cookie example (I)

```
class CookieEventArgs : System.EventArgs { };

class SesameStreet{

  delegate void CookieDelegate(object o,
                               CookieEventArgs c);
  event CookieDelegate CookieEvent;
  void DoCookie() {
      CookieEvent(this, new CookieEventArgs());}
}
```

Updating the cookie example (II)

```
class CookieMonster{

  CookieMonster(SesameStreet s){
  s.CookieEvent +=
    ((object o, CookieEventArgs c) =>
      System.Console.WriteLine("Nom, Nom") );
  }
}
```

Updating the cookie example (III)

```
public class Runner{
    static void Main(string[] args)
    {
        SesameStreet ss = new SesameStreet();
        CookieMonster cm = new CookieMonster(ss);
        ss.DoCookie();
        ss.DoCookie();
        ss.DoCookie();
    }
}
/* Output:     Nom, Nom
               Nom, Nom
               Nom, Nom      */
```

- Q) How would this change if declared DoCookie as an event?

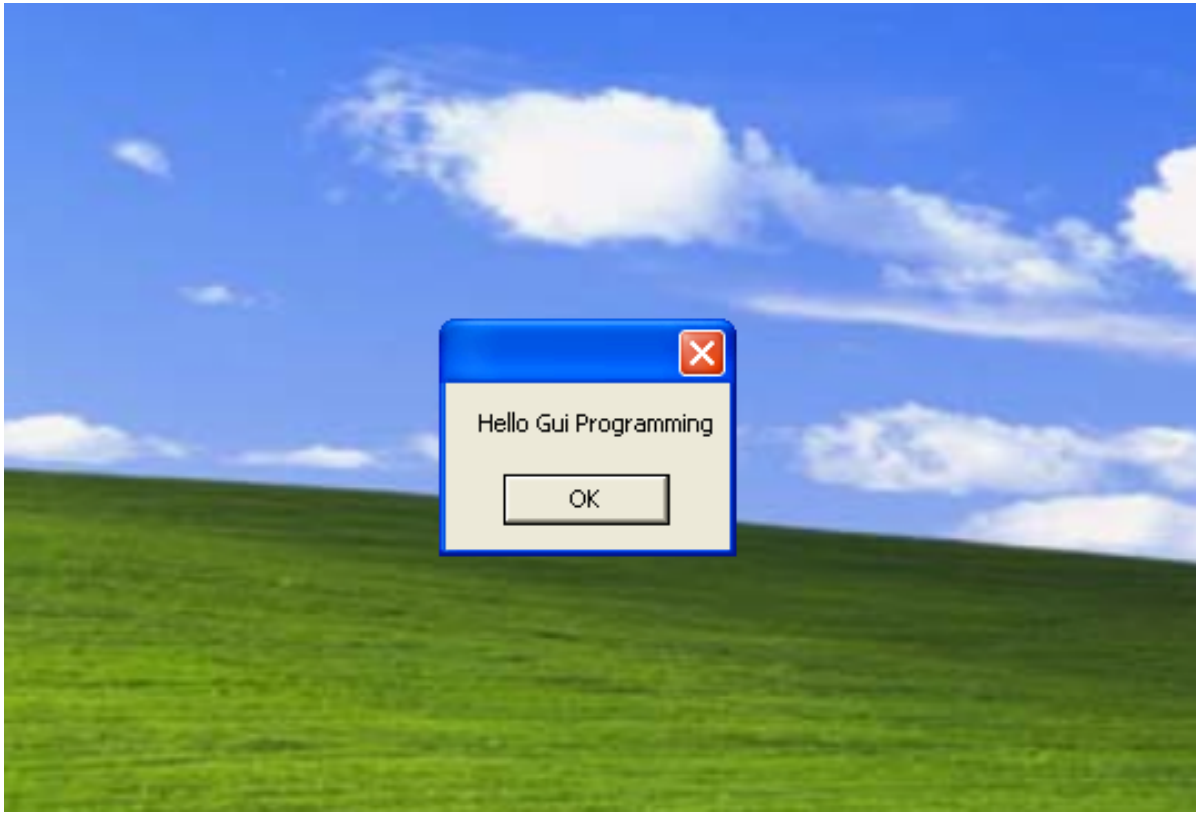
Gui programs are not special.

- Execution starts at Main
- Events model used to get inputs from controls
- Fancy designers just a convenient way to generate code
- (One caveat coming up)

A Simple Gui Program (I)

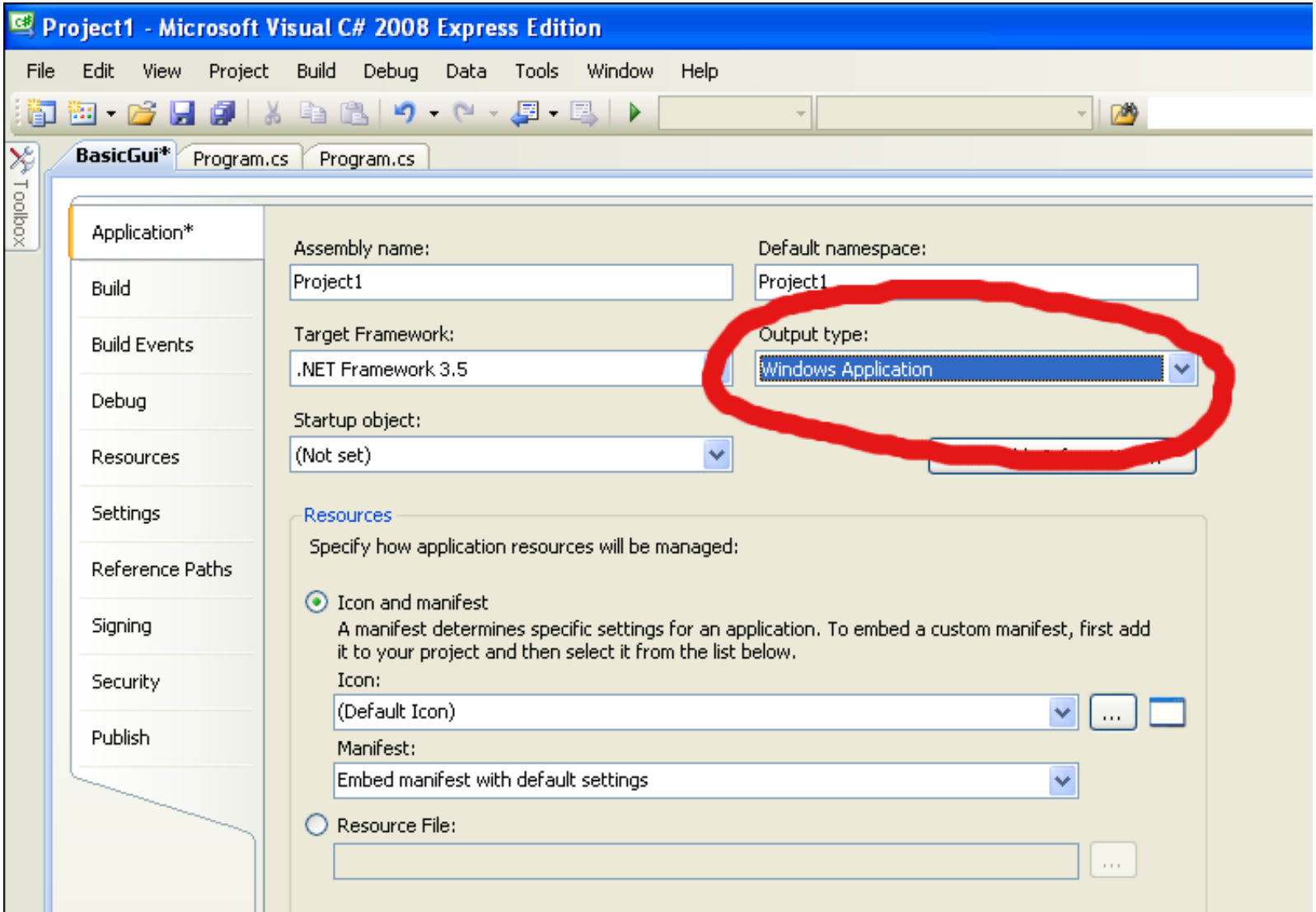
```
using System.Windows.Forms;

// Simplest GUI program.
// Compile as a "Windows Application"
class Program
{
    static void Main(string[] args)
    {
        MessageBox.Show("Hello Gui Programming");
    }
}
```

A Simple Gui Program (II)



A Simple Gui Program (III)

The caveat: I had to change the project's output type to "Windows Application". This stops the program from popping up a command prompt.



Event Driven Gui Programming

- All screen elements are represent by objects.
- Interesting user activities trigger events.
- Handling these event lets your program update it's state.

- Windows are instances of System.Windows.Forms.Form
- Buttons are instances of System.Windows.Controls.Button

Finally: A Gui That Does Something! (I)

```
static void Main() {
    RandColorPicker cp = new RandColorPicker();

    Form theForm = new Form();
    theForm.Text = "My Window";

    //Event handlers here
    theForm.MouseClick +=
      ((x,y) => theForm.BackColor = cp.GetRand());
    theForm.MouseEnter +=
      (delegate(object x, EventArgs y) {
        theForm.BackColor = cp.GetRand(); });

    theForm.ShowDialog();
}
```

Finally: A Gui That Does Something! (II)

Finally: A Gui That Does Something! (II)

Finally: A Gui That Does Something! (II)