

# More Language Features and Windows Forms

C# Programming

January 12

# Part I

## Some Language Features

# Inheritance

- To extend a class A: `class B : A { ... }`
- B inherits all instance variables and methods of A
- Which ones it can access depends on the visibility levels
- Unlike in Java, **method overriding is explicit**
- To allow a method to be overridden, define the method as **virtual** in the base class
- To override the method, define the method as **override** in the derived class
- To *hide* a function of the base class, define the method as **new** in the derived class

In the following examples,

```
A a = new A();
```

```
B b = new B();
```

# Inheritance

```
class A {  
    ...  
    public string m1() { return "a1"; }  
    ...  
}  
class B {  
    ...  
}
```

```
b.m1(); // "a1"
```

- B has inherited the method m1 from A

# Inheritance

```
class A {  
    ...  
    public virtual string m2() { return "a2"; }  
    ...  
}  
class B {  
    ...  
}
```

```
b.m2(); // "a2"
```

- Even though `m1` is defined to be `virtual`, since `B` does not override it, the definition in `A` gets executed

# Inheritance

```
class A {  
    ...  
    public virtual string m3() { return "a3"; }  
    ...  
}  
class B {  
    ...  
    public override string m3() { return "b3"; }  
    ...  
}
```

```
b.m3(); // "b3"
```

- B has overridden m3, so its execution its **dynamically dispatched**

# Inheritance

```
class A {  
    ...  
    public string m4() { return "a4"; }  
    ...  
}  
class B {  
    ...  
    public new string m4() { return "b4"; }  
    ...  
}
```

```
b.m4(); // "b4"
```

- B has *hidden* A's definition of m1 (but this is not dynamic dispatch)

# Inheritance

- m4 is **statically dispatched**
- `((A)b).m4(); // "a4"`
  
- On the other hand...
- `((A)b).m3(); // "b3"`
- ...since b's type at runtime is B

# Inheritance

- To require that a subclass provide an implementation for a method, declare it as **abstract**
- An abstract method declaration does not provide an implementation, just the method signature
- A class with one or more abstract methods must be declared abstract and cannot be instantiated
- To prevent a class from being extended, mark the class as **sealed** (like a final class in Java)

# Interfaces

- An interface is a collection of abstract definitions, of events, methods, properties, and indexers (we won't talk about indexers)

```
interface IGo {
    string Text {
        get;
        set;
    }
    void PrintText();
}
```

# Interfaces

- An implementing class would be declared

```
class MyGoo : IGoo { ... }
```
- MyGoo would need to provide concrete definitions for all abstract members defined in IGoo
- A class can implement multiple interfaces

# Arrays

- As in Java, arrays are reference types
- `int[] arr1 = new int[5];`
- `arr1.Length` is 5
- Possible ways to initialize:
  - `int[] arr1 = new int[5]{ 1, 2, 3, 4, 5 };`
  - `int[] arr1 = new int[] { 1, 2, 3, 4, 5 };`
  - `int[] arr1 = { 1, 2, 3, 4, 5 };`

# Multidimensional Arrays

- Unlike Java, there is a syntactical difference between **rectangular arrays** and **jagged arrays**

## Rectangular Arrays

- `int[,] arr2 = new int[3,2];`
- `arr2.Length` is 6
- `arr2.GetLength(0)` is 3
- `arr2.GetLength(1)` is 2
  
- Possible way to initialize:
- `int[,] arr2 = { { 1, 2 }, { 3, 4 }, { 5, 6 } };`

## Jagged Arrays

- `int[] [] arr3 = int[3] [];`
- `arr3[0] = new int[1];`
- `arr3[1] = new int[4];`
- `arr3[2] = new int[2];`
- `arr3.Length` is 3
- `arr3[0].Length` is 1
- `arr3[1].Length` is 4
- `arr3[2].Length` is 2

## Jagged Arrays

- Possible way to initialize:

```
int[] [] arr3 = new int[] [] {  
    new int[] { 1 },  
    new int[] { 1, 2, 3, 4 },  
    new int[] { 1, 2 }  
};
```

## Boxing / Unboxing

- **Boxing** is supplying a value type where a reference type is expected
- Boxing is implicit in C#
- `object o = 5;`
  
- **Unboxing** is supplying a reference type where a value type is expected
- Unboxing must be explicit in C#
- `int i = (int)o;`

## Parsing Numbers

- To parse a string into an integer:
- `int i = Int32.Parse("13");`
- Parsing other types of numbers is similar
- `Int32.Parse` can throw a `System.FormatException`

## Exception Handling Example

```
int i;  
try  
{  
    i = Int32.Parse("1a3");  
}  
catch (FormatException)  
{  
    Console.WriteLine("Error parsing");  
    i = -1;  
}
```

## Part II

# Event Arguments

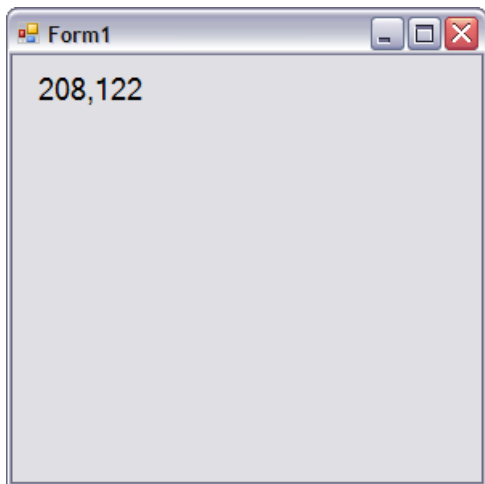
# Event Arguments

- When an object raises an event, it can send arguments along with it
- From looking at the event handlers that get attached to control events, we see that these events get packaged with two arguments
  - `object sender` – a reference to the control that fired the event
  - `EventArgs e` – or a subclass of `EventArgs` depending on the type of event

## MouseDown Arguments

- As an example, let's look at some of the information that is contained in e when the MouseDown event is fired
- We notice two properties, X and Y that represent the coordinates of the cursor when the mouse was clicked

```
void Form1_MouseDown(object sender, MouseEventArgs e) {  
    label1.Text =  
        e.X.ToString() + "," + e.Y.ToString();  
}
```



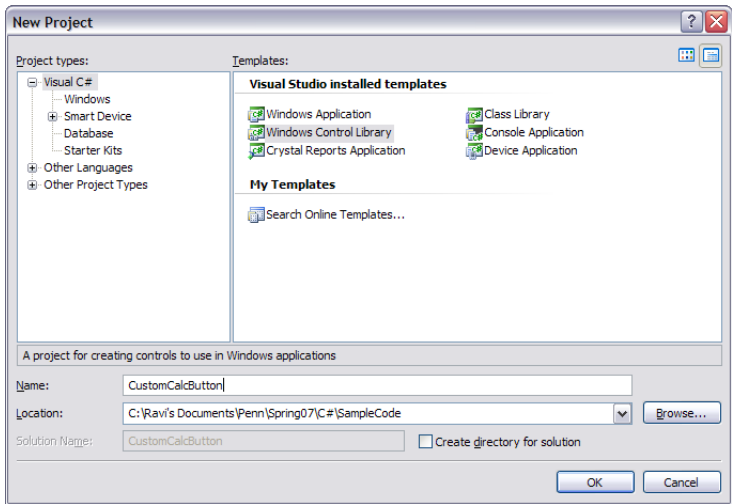
## Part III

# Custom Controls

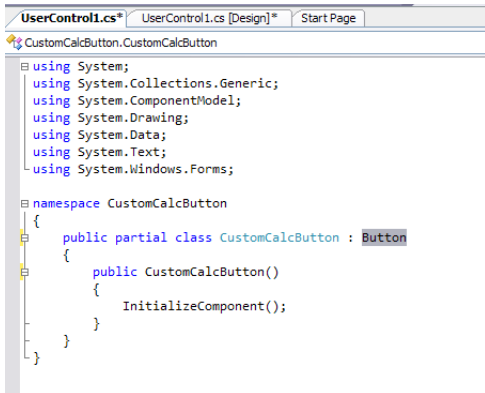
# Custom Controls

- Here is a short demonstration of how to define a custom control

- Create a **Windows Control Library** project



- Choose an existing control class to extend



```
UserControl1.cs* | UserControl1.cs [Design]* | Start Page
CustomCalcButton.CustomCalcButton
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;

namespace CustomCalcButton
{
    public partial class CustomCalcButton : Button
    {
        public CustomCalcButton()
        {
            InitializeComponent();
        }
    }
}
```

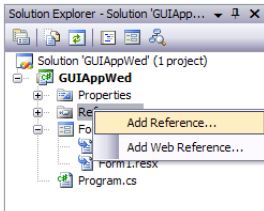
- Define the custom behavior

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;

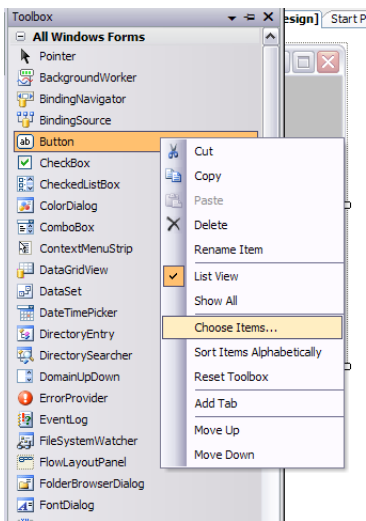
namespace CustomCalcButton
{
    public partial class CustomCalcButton : Button
    {
        public CustomCalcButton()
        {
            InitializeComponent();
            this.Click += new EventHandler(CustomCalcButton_Click);
        }

        void CustomCalcButton_Click(object sender, EventArgs e)
        {
            MessageBox.Show("custom control!");
        }
    }
}
```

- To use this from a Windows Forms application, you need to add a reference to the compiled .dll from your custom control project



- Or if you want to be able to drag-and-drop your custom control from the Toolbox, add your control to it



## Part IV

# Maintaining State

# Maintaining State

- It is a good idea to maintain a state variable
- What should the type of this be?
- A good option is to define an **enum** whose values are all possible states your application can be in
- The state variable can be an instance of this enum
- Code to update the GUI can perform a **switch** on the state variable

## A trivial example

- Consider a simple application to model a traffic light
- The empty form's background color changes from red to green to yellow according to a fixed timer
- We start by defining the states of the application, and an instance variable to hold the current state:

```
public partial class Form1 : Form {  
    public enum State { RED, GREEN, YELLOW }  
    private State state = State.RED;  
    ...  
}
```

- Now we define a method to transition from state to state...

```
private void ChangeColor() {  
    switch (this.state) {  
        case State.RED:  
            this.BackColor = Color.Green;  
            this.state = State.GREEN;  
            break;  
        case State.YELLOW:  
            this.BackColor = Color.Red;  
            this.state = State.RED;  
            break;  
        case State.GREEN:  
            this.BackColor = Color.Yellow;  
            this.state = State.YELLOW;  
            break;  
    }  
}
```

- And finally create a timer to drive state transitions

```
public Form1() {
    InitializeComponent();
    Timer timer1 = new Timer();
    timer1.Interval = 2000;
    timer1.Tick += new EventHandler(timer1_Tick);
    timer1.Start();
}

void timer1_Tick(object sender, EventArgs e) {
    ChangeColor();
}
```

## Maintaining State

- Since this is such a simple example, it is hardly worth the effort to explicitly define each state in an enum
- For applications that can be in several complicated states, however, organizing the GUI update logic in this manner can be very useful
- Also note that since the logic for updating the GUI (changing from red to green to yellow) would be the same no matter how we chose to “run” the traffic light, we placed that logic in a separate function
- If we wanted to change the application so that, say, a user clicked a button to change the state of the traffic light color, we would only have to make a call to `ChangeColor()` in the button’s event handler (instead of having to copy-and-paste the color update logic)