

Multiple Forms

C# Programming

January 24

Projects

- Calculator due tonight at midnight
- Instructions for submitting on the webpage
- Project 2 will be posted today and will be due Wednesday, February 7

Any questions?

Note on access modifiers

- Recall that the five access modifiers are:
 - `public`
 - `private`
 - `protected`
 - `internal`
 - `protected internal`
- The default visibility *for top-level types* is `internal`
- The default visibility *for instance variables and methods* is `private`

Part I

Multiple Forms

Multiple forms

- So far, our example applications have only used one window
- Most non-trivial applications use several windows
- These windows often need to communicate

Opening other forms

- Suppose the `Main()` method calls:
`Application.Run(new MainForm());`
- This creates an instance of `MainForm` and displays the window
- This is the top-level window for this WinForms app
- When this window is closed, the application terminates

Opening other forms

- The MainForm object can open other windows in two ways:
- `new ChildForm().Show();`
- `new ChildForm().ShowDialog();`
- `ShowDialog()` requires that the window be closed before focus can return to the caller

Communicating forms

- Consider an example where `MainForm` has a button and a textbox, and `ChildForm` has a textbox
- The button launches a `ChildForm` window
- We will set up two simple communication channels:
 1. When the parent's text box is updated, the child's gets updated to the same value
 2. When the child's text box is updated, the parent's gets updated to the same value
- We will look at three ways to implement these, with increasingly good design

Example 1a

- Our first attempt to communicate updates to the child form is by directly accessing its text box
- Since the child's controls are `private` by default, we need to either change it to `public` or `internal` so the main form can access it

ChildForm.Designer.cs:

```
internal System.Windows.Forms.TextBox txt1;
```

MainForm.cs:

```
ChildForm child;
```

```
private void btn1_Click(object sender, EventArgs e) {  
    child = new ChildForm();  
    child.Show();  
}
```

```
private void txt1_TextChanged(  
    object sender, EventArgs e) {  
    if (child != null)  
        child.txt1.Text = this.txt1.Text;  
}
```

Example 1b

- Instead of accessing the child form's control directly, we can instead keep the text box `private` and define a `public` or `internal` method in `ChildForm` that gets called when the parent's text box changes

ChildForm.Designer.cs:

```
private System.Windows.Forms.TextBox txt1;
```

ChildForm.cs:

```
internal void OnParentTextChanged(string s) {  
    this.txt1.Text = s;  
}
```

MainForm.cs:

```
private void txt1_TextChanged(  
    object sender, EventArgs e) {  
    if (child != null)  
        child.OnParentTextChanged(this.txt1.Text);  
}
```

Example 1c

- Defining the `OnParentTextChanged` function in the previous example feels like an event handler
- So let's use a delegate instead of calling the function directly

ChildForm.Designer.cs and ChildForm.cs: same as before

MainForm.cs:

```
private delegate void SetTextDelegate(string s);
private SetTextDelegate SetTextCallback;

private void btn1_Click(object sender, EventArgs e) {
    child = new ChildForm();
    this.SetTextCallback +=
        new SetTextDelegate(child.OnParentTextChanged);
    child.Show();
}

private void txt1_TextChanged(
    object sender, EventArgs e) {
    if (SetTextCallback != null)
        SetTextCallback(this.txt1.Text);
}
```

Example 2a/2b

- To communicate changes from the child to the main form, our first two approaches would require a reference to the main form itself
- The parent form would need to pass a reference of itself to the child so that it could manipulate its controls or call its methods
- (The code on the next slide skips the version where the child updates the parent's text box directly)

MainForm.cs:

```
private void btn1_Click(object sender, EventArgs e) {
    child = new ChildForm(this);
    child.Show();
}
internal void OnChildTextChanged(string s) {
    this.txt1.Text = s;
}
```

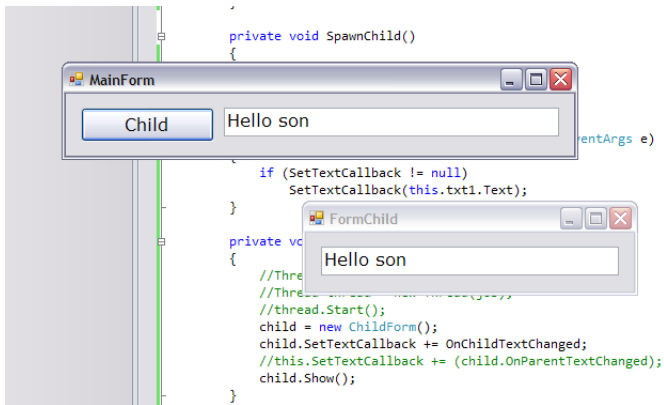
ChildForm.cs:

```
private MainForm f;
public ChildForm(MainForm f) {
    InitializeComponent(); this.f = f;
}
private void txt1_TextChanged(
    object sender, EventArgs e) {
    f.OnChildTextChanged(this.txt1.Text);
}
```

```
private void SpawnChild()
{
    if (SetTextCallback != null)
        SetTextCallback(this.txt1.Text);
}

private void SpawnChild(EventArgs e)
{
    if (SetTextCallback != null)
        SetTextCallback(this.txt1.Text);
}

private void SpawnChild(EventArgs e)
{
    //Thread t = new Thread(SpawnChild);
    //Thread t.Start();
    //thread.Start();
    child = new ChildForm();
    child.SetTextCallback += OnChildTextChanged;
    //this.SetTextCallback += (child.OnParentTextChanged);
    child.Show();
}
```



The image displays a Windows application interface with two overlapping windows. The top window, titled "MainForm", features a button labeled "Child" and a text box containing the text "Hello son". Below it, a smaller window titled "FormChild" also contains a text box with "Hello son". The background shows a code editor with C# code defining a `SpawnChild()` method and a `SpawnChild(EventArgs e)` method. The code includes comments about threads and shows the creation and configuration of a `ChildForm` object, including setting a text callback and displaying the window.

Example 2c

- We can also set up this callback through a delegate
- Since this time the delegate needs to be accessed by both classes, it is defined at the top-level of the namespace

```
namespace MultipleForms
{
    public delegate void SetTextDelegate(string s);
    ...
}
```

ChildForm.cs:

```
public SetTextDelegate SetTextCallback;
private void txt1_TextChanged(
    object sender, EventArgs e) {
    if (SetTextCallback != null)
        SetTextCallback(this.txt1.Text);
}
```

MainForm.cs:

```
internal void OnChildTextChanged(string s) {
    this.txt1.Text = s;
}
private void btn1_Click(object sender, EventArgs e) {
    child = new ChildForm();
    child.SetTextCallback += OnChildTextChanged;
    child.Show();
}
```

MainForm

Child Hello father

FormChild

Hello father|

Summary

- Facilitating communication between forms using `public` variables and methods is not a good idea
- For applications with tightly coupled behavior between forms, it is sometimes useful to facilitate communication using `internal` variables and methods freely
- For most applications, especially when communicating with separate code, controlling communication channels through delegates is the best choice

Part II

Multi Threading

Multi threading

- With GUI applications, we want the interface to remain responsive at all times
- For example, if the application is performing expensive I/O or network operations, the GUI should not lock up
- To decouple the execution of different tasks (like background computation versus a form responding to user events), these need to be split up into different threads
- If they are not (if the application is single-threaded instead), the rest of the application can be blocked and appear frozen while some expensive operation is executing

Multi threading

- A thread can be created and set to run an arbitrary function
- For now, we will look at an example of creating the `ChildForm` object in a separate thread

Single threaded issue

- In our examples, a MainForm object has created a ChildForm object
- It then calls Show() to display the form
- Show() **does not** start the child form in a new thread
- Therefore, any undue delay in the child form's execution will prevent the execution of the main form

Single threaded issue

- For example, consider the following silly behavior of the `ChildForm` constructor:

```
public ChildForm() {  
    InitializeComponent();  
    while (true) {  
        int i = 0;  
    }  
}
```

- When the button on the `MainForm` is clicked, this infinitely-looping constructor is called
- Since the child form is running in the same thread, the main form becomes unresponsive

Multi threading

- We can spawn a new thread for the child form to run in using classes in the `System.Threading` namespace

MainForm.cs:

```
private void SpawnChild() {  
    child = new ChildForm();  
    this.SetTextCallback += (child.OnParentTextChanged);  
    child.Show();  
}
```

```
private void btn1_Click(object sender, EventArgs e) {  
    ThreadStart job = new ThreadStart(SpawnChild);  
    Thread thread = new Thread(job);  
    thread.Start();  
}
```

Multi threading

- Now, while the child form is spinning its wheels, the main form remains responsive

Windows/.NET threading model

- We will return to multi threading in future assignments
- But there are many more details about how threading in Windows and .NET works than we will cover in this class
- If you are interested, there are some links posted to the Resources page that go into greater detail

Part III

About Boxes

About boxes

- It is good practice to have an About box for each Windows application you write
- The user expects this to be accessible from the Help -> About menu
- You can create any Form to serve as a template for your about boxes
- Or you can use built-in template

About boxes

- Add -> New Item -> AboutBox
- You can set the text that appears in the about box from the constructor

