

# INTERNET PROTOCOLS AND CLIENT-SERVER PROGRAMMING

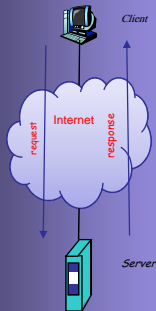
## SWE344

Fall Semester 2008-2009 (081)

**Module 3.2: Delegates, Events, GUI and Threads (Part 2)**

**Dr. El-Sayed El-Alfy**

Computer Science Department  
King Fahd University of Petroleum and Minerals  
alfy@kfupm.edu.sa

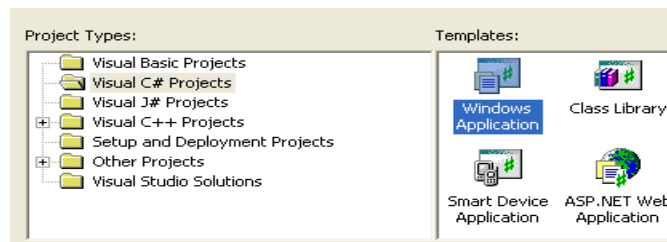


## Objectives

- ⊕ Learn about delegates, how to create them and how to use them.
- ⊕ Learn about special types of delegates called events.
- ⊕ Learn how to use the standard event handler.
- ⊕ Learn how to write GUI programs.
- ⊕ Learn how to write multi-threaded programs.

# GUI Programming

- ✦ In C#, GUI applications are created by extending the `System.Windows.Forms.Form` class.
- ✦ Developing GUI applications involves two things
  - Designing the user-interface
  - Implementing event-handling
- ✦ Visual Studio (VS) simplifies the development process



# GUI Programming ...

- ✦ Phase 1: Designing the user interface
  - In VS, create a new project and specify that you are creating a Windows Application
  - You have two views of your program: Source and Design.

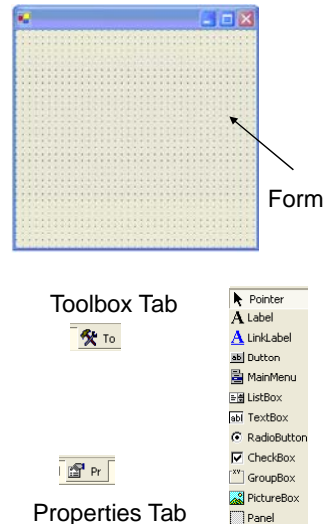


- The design view allows you to create a user interface by simply dragging the controls from the tools window into your design window

## GUI Programming ...

### ✚ Phase 1: Designing the user interface ...

- Switch to the Design tap, you will see a blank design window (Form)
- Open the **Tools** Window
- Drag the various **controls** you desire to the form design window and organize them to suit your application.
- Use the **Properties** Window to change the behavior and appearance of the controls



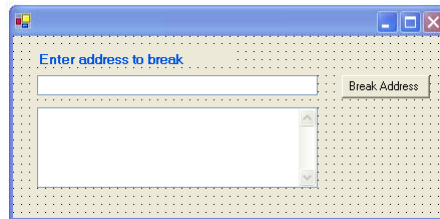
## GUI Programming ...

### ✚ Phase 1: Designing the user interface ...

- Some of the properties that are commonly changed include:
  - The variable *names* automatically generated for the controls
  - The *text* property
  - Multi-lines for TextBox (A text box is just a multi-line text field)
  - Scrollers for multi-line tools
  - Fonts (size, color, appearance, etc.)

## Sample Example

- ✚ Design the following user interface



- ✚ Switch to the source code to see the code that will be automatically generated for the above design

## Sample Example ...

```
1. using System;
2. using System.Windows.Forms;
3. namespace WindowsApp {
4.     public class BreakURL : System.Windows.Forms.Form
5.     {
6.         private System.Windows.Forms.TextBox resultBox;
7.         private System.Windows.Forms.TextBox addressBox;
8.         private System.Windows.Forms.Button button;
9.         private System.Windows.Forms.Label label;
10.        public BreakURL(){
11.            InitializeComponent();
12.        }
13.        void InitializeComponent() {
14.            this.label = new System.Windows.Forms.Label();
15.            this.button = new System.Windows.Forms.Button();
16.            this.addressBox = new System.Windows.Forms.TextBox();
17.            this.resultBox = new System.Windows.Forms.TextBox();
18.            this.SuspendLayout();
19.            // label
20.            this.label.Font = new System.Drawing.Font("Microsoft Sans Serif", 9.75F,
21.                System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point,
22.                ((System.Byte)(0)));
23.            this.label.ForeColor = System.Drawing.SystemColors.ActiveCaption;
24.            this.label.Location = new System.Drawing.Point(24, 16);
25.            this.label.Name = "label";
26.            this.label.Size = new System.Drawing.Size(200, 16);
27.            this.label.TabIndex = 0;
28.            this.label.Text = "Enter address to break";
```

Component  
declarations

Component  
Initialization

## Sample Example ...

```
29.         // button
30.         this.button.Location = new System.Drawing.Point(328, 40);
31.         this.button.Name = "button";
32.         this.button.Size = new System.Drawing.Size(88, 23);
33.         this.button.TabIndex = 3;
34.         this.button.Text = "Break Address";
35.         // addressBox
36.         this.addressBox.Location = new System.Drawing.Point(24, 40);
37.         this.addressBox.Name = "addressBox";
38.         this.addressBox.Size = new System.Drawing.Size(280, 20);
39.         this.addressBox.TabIndex = 1;
40.         this.addressBox.Text = "";
41.         // resultBox
42.         this.resultBox.BackColor = System.Drawing.Color.White;
43.         this.resultBox.Location = new System.Drawing.Point(24, 72);
44.         this.resultBox.Multiline = true;
45.         this.resultBox.Name = "resultBox";
46.         this.resultBox.ScrollBars = System.Windows.Forms.ScrollBars.Both;
47.         this.resultBox.Size = new System.Drawing.Size(280, 80);
48.         this.resultBox.TabIndex = 2;
49.         this.resultBox.Text = "";
```

## Sample Example ...

```
50.         // CreatedForm
51.         this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
52.         this.ClientSize = new System.Drawing.Size(432, 182);
53.         this.Controls.AddRange(new System.Windows.Forms.Control[] {
54.             this.button,
55.             this.resultBox,
56.             this.addressBox,
57.             this.label});
58.         this.Name = "CreatedForm";
59.         this.ResumeLayout(false);
60.     }
61.     public void BreakAddress(Object source, EventArgs arg) {
62.     }
63.
64.     public static void Main() {
65.         Application.Run(new BreakURL());
66.     }
67. }
68. }
```

EventHandler

Main()

## GUI Programming ...

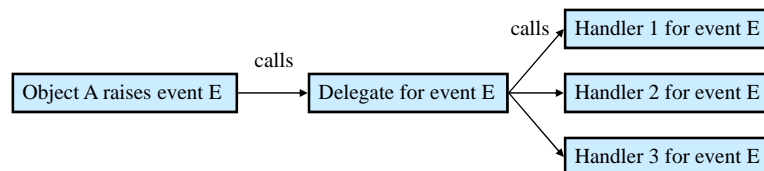
### ✚ Important Note:

- In .NET 2.0 (Visual Studio 2005), the concept of partial class was introduced.
  - This allows a class to be broken into more than one source files.
- When you create a GUI application, the system breaks your class (Form) into two partial classes, `Form.cs` and `Form.Designer.cs`
- The code generated by the IDE is stored in `Form.Designer.cs`. You should not change the code in this file manually. If you wish to make any changes, use the design window.
  - The IDE may override your changes if you go back to the design window.
- Any additional user code should be added inside the partial class, `Form.cs`

## GUI Programming ...

### ✚ Phase 2: Handling Events

- GUIs are event driven
- Event handling model using delegates



- Event handlers
  - Methods that process events and perform tasks.
- Associated event delegates
  - Objects that reference methods
  - Contain lists of method references
    - Must have same signature and return type


## GUI Programming ...

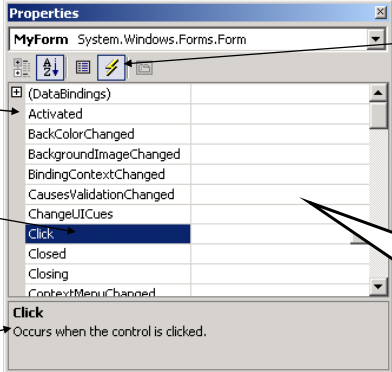
### ✚ Phase 2: Handling Events ...

- Each control has a number of event delegates of type, **EventHandler**, to which you can register your event handling methods.
- EventHandler delegate expects methods with signature **void MethodName(Object source, EventArgs arg)**
- Common events that many of the controls have are:
  - Click
  - Closed
  - Closing
  - Leave
  - etc.

## GUI Programming ...

### ✚ Phase 2: Handling Events ...

- To register for events, select the events tab,  , from within the properties window.



List of events supported by control

Selected event

Event description

Events icon

For each event that you wish to handle, type the method name to be executed when the event occurs.

## GUI Programming ...

- If the method does not exist, the system will automatically create it with an empty body, which you can then fill, e.g., in the example above, an empty method:

`void BreakAddress(Object source, EventArgs arg)`

is created and registered with the Click method of the button control.

- The final thing you need to do for your program to execute is to create a main method.
  - In which, the static Run method of the Application class is called, passing to it, an instance of the form as argument.

## Threads

- ✚ An application consists of a single process but it can have one or more threads
- ✚ Threads provide a way to execute different tasks simultaneously
  - The CPU switches back and forth between different threads
- ✚ Threads do not increase the amount of work your computer can do but they
  - can share the resources more efficiently
    - Executing other tasks while one task is waiting for I/O
  - can help the computer to appear more responsive
    - e.g., background printing, spell checking, etc
- ✚ However, thread scheduling is by nature nondeterministic and incurs extra overhead
  - special attention is needed to avoid resulting problems



## Writing Multi-Threaded Applications

- ✦ Enclose the code to be executed in a separate thread into a method with the signature  
`void MethodName()`
- ✦ Create an instance of the `System.Threading.ThreadStart` delegate and pass the method as parameter
- ✦ Create an instance of the `System.Threading.Thread` class and pass the delegate as a parameter
  - Once an instance of Thread is created, it can be started by calling its `Start()` method.
  - It can be suspended and resumed using `Suspend()` and `Resume()`
  - It can be aborted using `Abort()`
- ✦ Example

Thread `thread1` = new Thread(new ThreadStart(`Counter1`));

## Sample Example

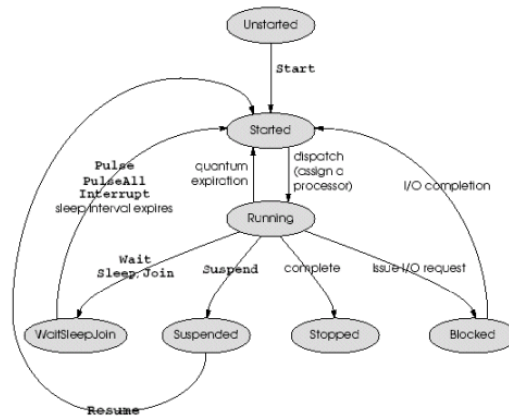
```
1. using System;
2. using System.Threading;
3. public class ThreadedCounters {
4.     public static void Main(){
5.         Thread thread1 = new Thread(new ThreadStart(Counter1));
6.         thread1.Start();
7.         Thread thread2 = new Thread(new ThreadStart(Counter2));
8.         thread2.Start();
9.     }
10.    public static void Counter1() {
11.        for (int i = 0; i<10; i++) {
12.            Console.WriteLine("Counter 1: "+i);
13.            Thread.Sleep(35);
14.        }
15.    }
16.    public static void Counter2() {
17.        for (int i = 0; i<10; i++) {
18.            Console.WriteLine("Counter 2: "+i);
19.            Thread.Sleep(20);
20.        }
21.    }
22. }
```

```
Counter 1: 0
Counter 2: 0
Counter 2: 1
Counter 1: 1
Counter 2: 2
Counter 2: 3
Counter 1: 2
Counter 2: 4
Counter 2: 5
Counter 1: 3
Counter 2: 6
Counter 1: 4
Counter 2: 7
Counter 1: 5
Counter 2: 8
Counter 1: 6
Counter 2: 9
Counter 1: 7
Counter 1: 8
Counter 1: 9
Press any key to continue . . . _
```

## Lifetime Cycle of a Thread

- A thread object can have many possible states, e.g.,
  - Unstarted: When it is created
  - Started: After Start() is called
  - Running: When it gets a processor
  - Suspended: by calling Suspend ()
  - A suspended thread is resumed when the Resume() is called.
  - The Sleep method is often used, as the name implies, to send a thread to sleep for a specified period of time.
  - After the time elapses, the thread wakes up and continues running automatically.
  - A Thread object should be terminated when the application exits using Abort()

These states are members of the **System.Threading.ThreadState** enumeration



State Transition Diagram

## Lifetime Cycle of a Thread ...

State	Description
Aborted	The thread is in the Stopped state.
AbortRequested	ThreadAbort() method has been invoked on the thread, but the thread has not yet received the System.Threading.ThreadAbortException that will try to stop it.
Background	The thread is being executed as a background thread, as opposed to a foreground thread. This state is controlled by setting the IsBackground property of the Thread class.
Running	The thread has been started, it is not blocked, and there is no pending ThreadAbortException.
Stopped	The thread has stopped.
StopRequested	The thread is being requested to stop. This is for internal use only.
Suspended	The thread has been suspended.
SuspendRequested	The thread is being requested to suspend
Unstarted	The Thread.Start method has not been invoked on the thread.
WaitSleepJoin	The thread is blocked as a result of a call to Wait, Sleep or Join methods.

## More on Threads

- ✦ Thread class has the **IsAlive** property that can be used to inquire about the state of a Thread object.
  - If IsAlive property is True, the thread has been started and has not been aborted.
- ✦ A thread can also run in background or foreground
  - A background thread is the same as a foreground thread, except that background threads do not prevent a process from terminating.
- ✦ A Thread is put in background state by setting the Thread **IsBackground** property to True.

## Example

- ✦ It shows how to use threads to write GUI applications which are *responsive*.
- ✦ Without using thread, a GUI control could be inactive while some computation is going on, which is an undesirable behavior.



## Example

```
1. using System;
2. using System.ComponentModel;
3. using System.Drawing;
4. using System.Text;
5. using System.Windows.Forms;
6. using System.Threading;

7. namespace CarRace
8. {
9.     public partial class Form1 : Form
10.    {
11.        int leftX, rightX, X, Y;
12.        Thread driver;
13.        bool start = false;
14.
15.        public Form1()
16.        {
17.            InitializeComponent();
18.            Control.CheckForIllegalCrossThreadCalls = false;
19.            X = leftX = Bounds.Left;
20.            Y = car1.Location.Y;
21.            rightX = Bounds.Right;
22.            car1.Location = new Point(X, Y);
23.            driver = new Thread(new ThreadStart(DriveCar));
24.        }
```

Associate a method with the thread

## Example ...

```
25. private void startStop_Click(object sender, EventArgs e) {
26.     if (!start) {
27.         start = true;
28.         startStop.Text = "Stop";
29.         if (driver.IsAlive)
30.             driver.Resume();
31.         else
32.             driver.Start();
33.     }
34.     else {
35.         start = false;
36.         startStop.Text = "Start";
37.         driver.Suspend();
38.     }
39. }
40. void DriveCar() {
41.     while (true) {
42.         while (X < rightX) {
43.             X += 10;
44.             car1.Location = new Point(X, Y);
45.             Refresh();
46.             Thread.Sleep(30);
47.         }
48.         X = leftX - 100;
49.     }
50. }
```

Changes the thread state in response to button clicks

## Example ...

```
51. protected override void OnClosing(CancelEventArgs e)
52. {
53.     if (driver.IsAlive)
54.         if (start)
55.             driver.Abort();
56.         else
57.         {
58.             driver.Resume();
59.             driver.Abort();
60.         }
61.     base.OnClosing(e);
62. }
63.
64. }
```

## Thread Problems

- ⊕ The nondeterministic nature of thread scheduling can lead to many problems in the code
  - Resource contention issues
    - Occur when more than one thread needs to modify a certain object at a time.
    - Coordination is needed to resolve these problems
  - Deadlocks
    - A deadlock occurs when there are two threads each of them waiting for an object locked by the other
    - Handled by modern OS
  - Race conditions
    - A race condition occurs when your code depends on one thread completing some work before another thread is called however the second thread starts before the first complete the required work
    - Handled by modern OS

## Managing Threads

- ✚ There are ways to manage the behavior of threads
  - Using **Lock**, **Interlocked**, **Monitor**, **Mutex** classes to coordinate the activities and resource usage of multiple threads (thread synchronization)
  - Using **Timer** class to run threads at specific intervals (thread scheduling)
  - Using **Join** method to make one thread wait for another thread to complete

## Example

- ✚ Thread Synchronization
  - Consider the Unsafe banking example below

```
1. using System;
2. using System.Threading;
3. public class BankAccount {
4.     int balance = 0;
5.     public BankAccount(int initial) {
6.         balance = initial;
7.     }
8.     public void Deposit(int amount) {
9.         balance+=amount;
10.    }
11.    public void Withdraw(int amount) {
12.        balance-=amount;
13.    }
14.    public int GetBalance() {
15.        return balance;
16.    }
17. }
```

## Example ...

```
18. public class UnsafeBanking {
19.     static Random randomizer = new Random();
20.     static BankAccount account = new BankAccount(100);
21.     public static void Main() {
22.         Thread[] banker = new Thread[10];
23.         for (int i=0; i<10; i++) {
24.             banker[i] = new Thread(new ThreadStart(DepositWithdraw));
25.             banker[i].Start();
26.         }
27.     }
28.     public static void DepositWithdraw() {
29.         int amount = randomizer.Next(100);
30.         account.Deposit(amount);
31.         Thread.Sleep(100);
32.         account.Withdraw(amount);
33.         Console.WriteLine(account.GetBalance());
34.     }
35. }
```

the amount being deposited is the same as the amount withdrawn, one would expect the balance to remain unchanged

```
486
405
326
279
241
194
164
139
102
100
Press any key to continue . . .
```

Wrong output!!

## Using the Monitor Class

- ⊕ The Monitor class provides two static methods,
  - **Enter** method
    - used to obtain a lock on an object that the monitor guards and is called before accessing the object.
    - If the lock is currently owned by another thread, the thread that calls Enter blocks—that is, is taken off the processor and placed in a very efficient wait state—until the lock becomes free.
  - **Exit** method
    - frees the lock after the access is complete so that other threads can access the resource.
- ⊕ Example:
  - The following uses Enter/Exit methods to synchronize access to the account object

## Using the Monitor Class ...

```
1. public static void DepositWithdraw() {
2.     int amount = randomizer.Next(100);
3.     Monitor.Enter(account);
4.     try {
5.         account.Deposit(amount);
6.         Thread.Sleep(100);
7.         account.Withdraw(amount);
8.         Console.WriteLine(account.GetBalance());
9.     }
10.    finally {
11.        Monitor.Exit(account);
12.    }
13. }
```

- ⊕ Note that calls to Exit are enclosed in finally blocks to ensure that they're executed even when there are exceptions.
- ⊕ Always use finally blocks to exit monitors or else you run the risk of causing other threads to hang indefinitely.

## Using the lock keyword

- ⊕ An alternative to using the Enter/Exit method is to use the **lock** keyword around the code that should be accessed by many threads

```
1. public static void DepositWithdraw() {
2.     int amount = randomizer.Next(100);
3.     lock(account) {
4.         account.Deposit(amount);
5.         Thread.Sleep(100);
6.         account.Withdraw(amount);
7.         Console.WriteLine(account.GetBalance());
8.     }
9. }
```

```
100
100
100
100
100
100
100
100
100
100
100
100
Press any ke
```



## Synchronizing Access to Collections

- ✦ The collections classes in the .NET Framework class library are not thread-safe.
  - For example, when sharing an ArrayList between a reader thread and a writer thread, it's important to synchronize access to the ArrayList so that one thread can't read from it while another thread writes to it.
  - One way to synchronize access to an ArrayList is to use a monitor or lock

```
ArrayList list = new ArrayList ();
// Thread A
lock (list) {
    list.Add ("Fender Stratocaster");
}
// Thread B
lock (list) {
    string item = (string) list[list.Count - 1];
}
```

## Synchronizing Access to Collections

- ✦ Another approach
  - use the **Synchronized()** method implemented in ArrayList, Hashtable, Queue, Stack, and selected other standard classes
  - It returns a thread-safe wrapper around the object passed to it.

```
// Create the ArrayList and a thread-safe wrapper for it
ArrayList list = new ArrayList ();
ArrayList safelist = ArrayList.Synchronized (list);
// Thread A
safelist.Add ("Some Item");
// Thread B
string item = (string) safelist[safelist.Count - 1];
```

- ✦ Advantages
  - Using thread-safe wrappers created with the Synchronized() method shifts the burden of synchronization from your code to the framework
  - It can also improve performance because a well-designed wrapper class can use its knowledge of the underlying class to lock only when necessary.

## Resources

### ✦ MSDN Library

- <http://msdn.microsoft.com/en-us/default.aspx>

### ✦ Books

- C# 3.0 The Complete Reference, 3E, 2005
- C# 3.0 in a Nutshell: A Desktop Quick Reference, 2007
- Pro C# 2008 and the .NET 3.5 Platform, 4E, 2007
- C# How to Program, By Deitel
- Richard Blum, C# Network Programming. Sybex 2002.

### ✦ Lecture notes of previous offerings of SWE344 and ICS343

### ✦ Some other web sites and books; check the course website at

- <http://faculty.kfupm.edu.sa/ics/alfy/files/teaching/swe344/index.htm>