Taming .NET Multithreading

Venkat Subramaniam

venkats@agiledeveloper.com

http://www.agiledeveloper.com/download.aspx

Code examples from this presentation may be downloaded from the above URL

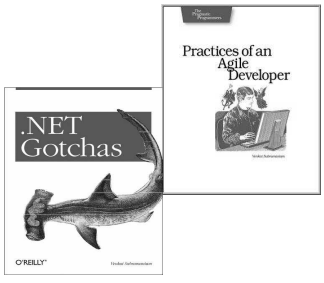
Abstract

Abstract It is easy to start a thread and then the fun starts! Developing a multithreaded application can be quite a challenge. You need to worry about thread safety, cleanup, contention, support and restrictions. This presentation starts with the features of multithreading in .NET and goes into issues of contention, performance, thread pooling, invocation restrictions, and related concepts. Issues related to how exceptions get handled in a multithreaded application will also be presented. Topics will also include issues of when to use multithreading and things we need to pay close attention to when designing and developing a multithreaded application.

About the Speaker *Dr. Venkat Subramaniam*, founder of Agile Developer, Inc., has trained and mentored thousands of software developers in the US, Canada and Europe. He has significant experience in architecture, design, and development of software applications. Venkat helps his clients effectively apply and succeed with agile practices on their software projects, and speaks frequently at conferences.

He is also an adjunct faculty at the University of Houston (where he received the 2004 CS department teaching excellence award) and teaches the professional software developer series at Rice University School of continuing studies.

Venkat has been a frequent speaker at No Fluff Just Stuff Software Symposium since Summer 2002.

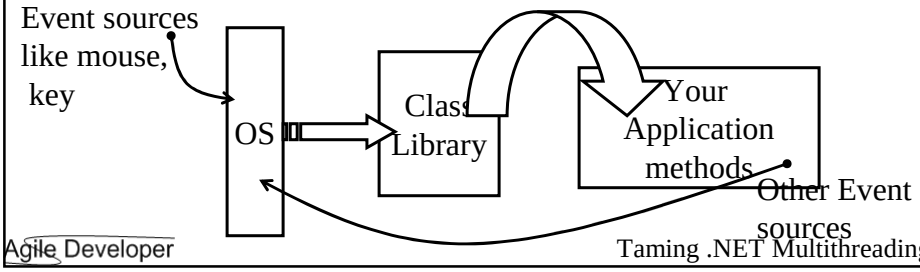


Taming .NET Multithreaing

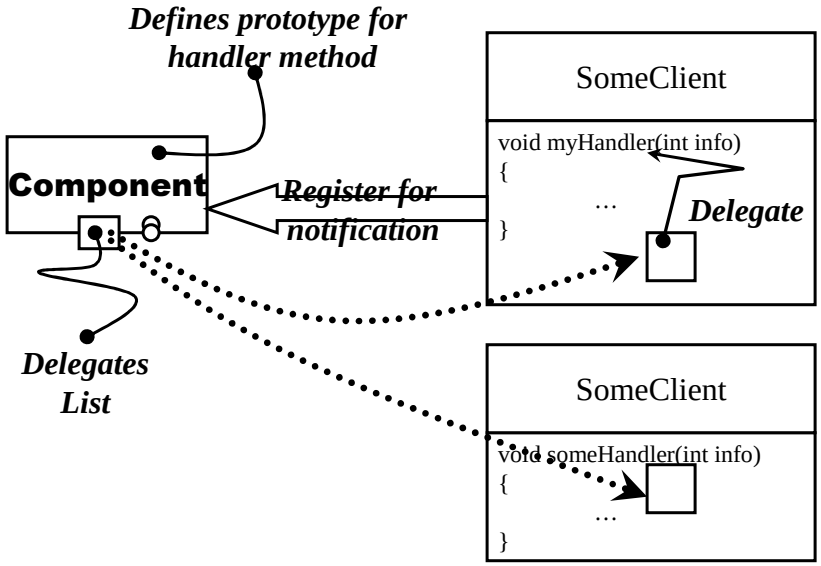
- **Event Handling**
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

Event-driven Programming

- Conventional Console Application Model
- Event
  - Signal a program receives from the operating system as a result of some (user) action
- Event-driven program
  - receives and responds to events
  - where in these events are generally asynchronous



Event Handling



Taming .NET Multithreaing

- Event Handling
- **Delegates**
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

Delegate

- Event handlers traditionally were global or static functions
- In .NET, delegates allow a lot more
- Delegates are data structures (objects) that hold
  - either a pointer to global or static function
  - or a pointer to an object's method and the object itself
  - are object-oriented, type safe and secure
- All Delegates derive from the Delegate class

Writing A Delegate

- Delegate classes are written with special syntax
- Compiler does special processing
  - Writes a constructor and Invoke method when compiled
- You can maintain a list of delegates by simply adding and subtracting
  - myDelegate += anotherDelegate;
    // Adds the delegate
  - myDelegate -= anotherDelegate;
    // Removes the delegate

Taming .NET Multithreaing

- Event Handling
- Delegates
- **Events**
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

Events

- Events are implemented using the Delegates
- Act like the connection point in COM
- You first declare a delegate, only if a suitable one is not available
  - must take two arguments: **Object source, EventArgs e**
  - .NET has predefined delegate EventHandler
- Any number of events can use the same Delegate definition
- Use event keyword to declare event in a class
- events appear like fields within a class
  - big difference: fields can't appear in interface, events can

Using Events

```
public class StockQuote {
      public event EventHandler highPriceEvent;
      public event EventHandler lowPriceEvent;
      private void newHighReached(double amount) {
            if (highPriceEvent != null)
                  highPriceEvent(this,
                        new PriceData(amount));
      }
}
```

Derives from EventArgs

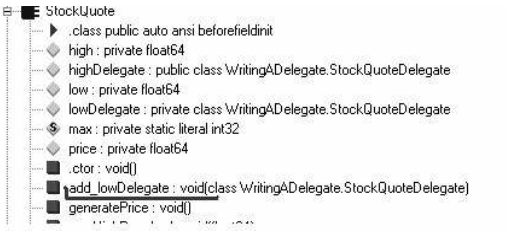
```
public static void highReport(Object source, EventArgs e){…}
…
StockQuote stkQuote = new StockQuote();

stkQuote.highPriceEvent += new
                        EventHandler(User.highReport);
```


Taming .NET Multithreaing

- Event Handling
- Delegates
- Events
- **Events vs. Delegates**
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

What's difference?

- At first sight events are nothing but delegate
- We use a special keyword event
  - so studio can recognized it as special member?
- Actually there's more to it!
- The secret is in IL



- Take a closer look at delegates and events in IL
  - Special synchronizing methods added for events

Taming .NET Multithreaing

- Event Handling
- Delegates
- Events
- Events vs. Delegates
- **Support for Multithreading**
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

Support for Multithreading

- C#'s supports multithreading through System.Threading namespace
- At the core is the Thread class
  - unlike Java, you can't derive from this, it is sealed
- Thread works with a delegate called ThreadStart
- For creating a thread, create an object of Thread, register a delegate with it and start

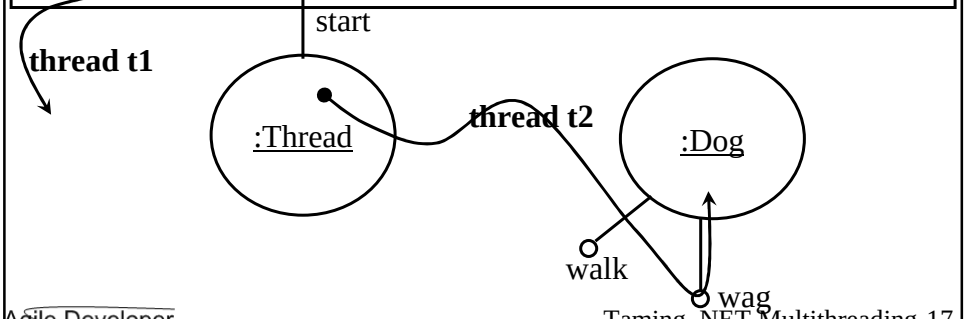
Thread Class

```
public sealed class Thread {
      public static Thread CurrentThread { get; }
      …
      public static void Sleep(int milliSeconds);
      public ApartmentState ApartmentState { get; set; }
      public string Name { get; set; }
      public ThreadState ThreadState { get; }
      public void Abort();
      public void Start();
      public void Join(…);
      public void Suspend();
      public void Resume();
}
```

Manages a thread of execution

Thread Class & Thread of Execution

```
public class Dog {
    private void keepWagging() { while(true) {…} }
    public void startToWag() {
       waggingThread =
            new Thread(new ThreadStart(keepWagging));
       waggingThread.Start();
      }
    public void stopWagging() {… waggingThread.Abort(); }
      }
}
```



Taming .NET Multithreaing

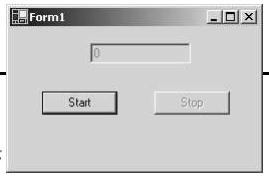
- Event Handling
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- **Starting limitations**
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

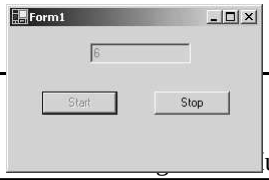
Starting Limitations

- On a thread object, start may be called only once
- What if I call start more than once?
- you get ThreadStateException
- A Thread object is not reusable
  - it represents the identity of one thread of execution only

Multithreaded StopWatch

```
public class MainForm : System.Windows.Forms.Form {…
      private int count = 0;
      private bool keepCounting = false;
      private void start(){
            keepCounting = true;
            while(keepCounting) {
                  count++;
                  TBSeconds.Text = "" + count;
                  Thread.Sleep(1000);
            }
      }
      private void stop() { keepCounting = false; }
      private void StartButton_Click(
                  object sender, EventArgs e) {…
            new Thread(new ThreadStart(start)).start();
      }
}
```





Obtaining Thread Information

- Sometimes it is necessary to find details of Thread executing some method
- Thread.CurrentThread
  - returns true identity of Thread object that represents current thread of execution
- Naming a thread may be useful for debugging
  - accessed through the Name property

Obtaining Thread ID

- Oddly, there is no way to get the Thread's ID from a thread object
- You can however, get it from the AppDomain class!
- AppDomain.GetCurrentThreadID()

Taming .NET Multithreaing

- Event Handling
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- **Background threads**
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

IsBackground

- Background thread runs only if there is a non-background thread running
- By default thread is not background
- Ask yourself if your thread keeps CLR running or does it quit?

Quiz Time

Taming .NET Multithreaing

- Event Handling
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- **Thread Interrupt and Abort**
- Background and abort
- Behavior of Exit
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

Thread Interrupt and Abort

- Important to understand these well
- Interrupt will
  - throw ThreadInterruptedException
  - **Only** is thread is blocked on a Sleep, Wait, Join
    - if not, is thrown when ever it gets blocked
- Abort will
  - throw ThreadAbortException
  - This exception propagates automatically after a catch
    - We could say thread is terminated upon Abort
    - Caution, call to ResetAbort

Taming .NET Multithreaing

- Event Handling
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- **Background and abort**
- Behavior of Exit
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

Background and Abort

- According to documentation when process exits background threads are aborted
- Not in the sense of Thread Abort however
- They are terminated – no opportunity to clean up – tough luck

Taming .NET Multithreaing

- Event Handling
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- **Behavior of Exit**
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

Know your Exit

- Exit simply terminates the process, the CLR
- No opportunity to clean up
- Any thread that has security permission to call Exit can terminate the process

Taming .NET Multithreaing

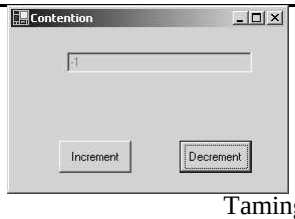
- Event Handling
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- **Contention**
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

Multithreading and Contention

- Several threads updating same data

```
public void increment() {
      int val = count;
      Thread.Sleep(5000);
      count = val + 1;
      counterTB.Text = "" + count;
}
```

```
public void decrement() {
      int val = count;
      Thread.Sleep(5000);
      count = val - 1;
      counterTB.Text = "" + count;
}
```



Taming .NET Multithreaing

- Event Handling
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- **Locking**
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

Thread Synchronization

- C# provides mutual exclusion through **lock**
- At most one thread may be in a critical section
- CLR ***locks*** the object when a thread is executing any critical section block of code *locking that* object

```
public void increment() {
      lock(this) {
            int val = count;
      }
}
```

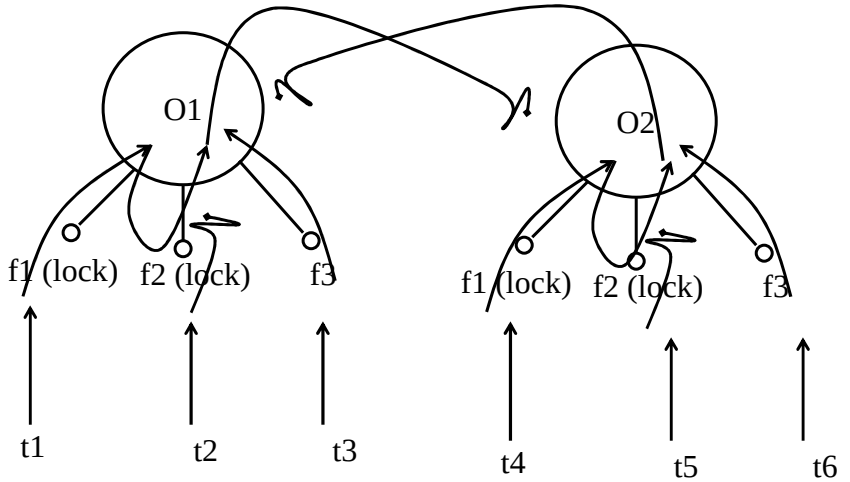
Blocks until obtaining lock on object

```
public void decrement() {
      lock(this) {
            int val = count;
            Thread.Sleep(5000);
            count = val - 1;
            counterTB.Text = "" + count;
      }
}
```

Releases lock automatically

Synchronization and Threads

Two objects of same class, f3 has no critical section



t1 calls f1, f2 on O1 and f1 on O2; similarly t4…

Taming .NET Multithreaing

- Event Handling
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- Locking
- **Improper locking**
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

Contention for Class (static) variables

- Bacteria Example

  Keeps tract of number of Bacterial objects

```
public Bacteria()
{
        lock(typeof(Bacteria)) {
                bacteriaCount++;
        }
}
```

This is too sweeping.

- If all that you want to do is increment or decrement a value, you may also use the InterlockedIncrement or InterlockedDecrement

Closer look at *lock*

- lock is actually to short form/wrapper
- lock(obj) { … code … } is equivalent to

```
System.Threading.Monitor.Enter(obj);
try
{
    … code …
}
finally
{
    System.Threading.Monitor.Exit(obj);
}
```


The Monitor

- This is the central class that provides synchronization in .NET
- It obtains or releases lock on any given object
- Blocks threads requesting lock until they gain exclusive access – providing mutual exclusion

```
public class Monitor {
      public static void Enter(Object obj);
      public static void Exit(Object obj);
      public static bool TryEnter(Object obj);  …
      public static bool Wait(Object obj);      …
      public static void Pulse(Object obj);
      public static void PulseAll(Object obj);
}
```

Taming .NET Multithreaing

- Event Handling
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- Locking
- Improper locking
- **Thread Pool behavior**
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

Thread Pool

- Should you use a thread from thread pool or create a thread yourself?
- Using thread pool is so much easier, isn't it?
- Gives better performance as well
- But,
  - at most 25 threads per process per processor by default
  - You may holdup start of some tasks – use caution

Taming .NET Multithreaing

- Event Handling
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- **Delegate BeginInvoke**
- Lost Exceptions
- WinForm Controls Thread Safety
- Conclusion

Delegate BeginInvoke

- Calling methods which take parameter in another thread is hard
- But, look how easy it is to start a method using the Delegate BeginInvoke method
- Use caution in understanding where the thread is executing
  - Thread Pool thread or not

Taming .NET Multithreaing

- Event Handling
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- **Lost Exceptions**
- WinForm Controls Thread Safety
- Conclusion

Thread Pool Exceptions

- What happens when an exception is thrown from a thread in thread pool?
- .NET Framework quietly suppresses it!
- This is what happens when you invoke a web service using asynch call and it breaks
- Also when you use Delegate.BeginInvoke

Taming .NET Multithreaing

- Event Handling
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- **WinForm Controls Thread Safety**
- Conclusion

WinForm Controls

- Most of the methods on WinForm controls are not thread safe
  - This is actually better for performance
  - But that's also like having un-gated rail road crossings
- Ask InvokeRequired and jump threads before accessing the control

Quiz Time

Taming .NET Multithreaing

- Event Handling
- Delegates
- Events
- Events vs. Delegates
- Support for Multithreading
- Starting limitations
- Background threads
- Thread Interrupt and Abort
- Background and abort
- Behavior of Exit
- Contention
- Locking
- Improper locking
- Thread Pool behavior
- Delegate BeginInvoke
- Lost Exceptions
- WinForm Controls Thread Safety
- **Conclusion**

Conclusion

- Starting thread is easy
- Getting it right is hard

- A number of excellent facilities provided in .NET
- A few Gotchas as well

- Only one way to master Multithreading… understand it well

References

1. .NET Gotchas, Venkat Subramaniam, O'Reilly.

2. Microsoft Developer Network (MSDN). http://msdn.microsoft.com

3. Essential .NET, Volume I: The Common Language Runtime, Don Box, Addison-Wesley.

4. Concurrent Programming in Java, Doug Lea, Addison-Wesley.

5. http://www.AgileDeveloper.com/download.aspx

Thanks!

Please fill out your evaluations!