

Using C# Language Features with the CLR's Asynchronous Programming Model



Jeffrey Richter
Copyright © 2008
www.wintellect.com

APM Programming Model Pain Points

- The APM enables scalable and responsive applications with minimal resources (threads/context switches)
- Unfortunately, programmers don't like the programming model and so many just don't do it
 - Must split code into many callback methods
 - Can't use arguments/locals; data can't move to other method/thread
 - Can't use: try/catch/finally, lock, using, for, while, and foreach
 - Hard to implement various features:
 - Coordinating multiple concurrent operations
 - Offer cancelation and timeout
 - Marshalling work to GUI thread to update controls
- This talk examines C# language features that simplify this

Asynchronous Pattern: Passing Objects Between Methods

```
private sealed class ApmData { // Class to pass data across method/thread
    public FileStream m_fs;
    public Byte[] m_data;
}

private static void ApmPatternWithMultipleMethods(String pathname) {
    ApmData apmData = new ApmData();
    apmData.m_fs = new FileStream(pathname, FileMode.Open,
        FileAccess.Read, FileShare.Read, 8192, FileOptions.Asynchronous);
    apmData.m_data = new Byte[apmData.m_fs.Length];
    apmData.m_fs.BeginRead(apmData.m_data, 0, apmData.m_data.Length,
        ReadCompleted, apmData);
    // Method returns after issuing async I/O request
}

private static void ReadCompleted(IAsyncResult result) {
    ApmData apmData = (ApmData) result.AsyncState;
    Int32 bytesRead = apmData.m_fs.EndRead(result);
    ProcessData(apmData.m_data);
    apmData.m_fs.Close(); // No 'using'
}
```



Asynchronous Pattern: Using Fields Across Methods

```
private sealed class ApmPatternWithMultipleMethods {  
    private FileStream m_fs;  
    private Byte[] m_data;  
  
    public ApmPatternWithMultipleMethods(String pathname) {  
        m_fs = new FileStream(pathname, FileMode.Open,  
            FileAccess.Read, FileShare.Read, 8192, FileOptions.Asynchronous);  
        m_data = new Byte[m_fs.Length];  
        m_fs.BeginRead(m_data, 0, m_data.Length, ReadCompleted, null);  
        // Method returns after issuing async I/O request  
    }  
  
    private void ReadCompleted(IAsyncResult result) {  
        Int32 bytesRead = m_fs.EndRead(result);  
        ProcessData(m_data);  
        m_fs.Close(); // No 'using'  
    }  
}
```

Anonymous Methods and Lambda Expressions



APM with Anonymous Methods

- C# 2.0 has Anonymous Method feature
- Let's programmer pass code as an argument to a method;
 - Lets programmer embed one method's code in another method
- Things to notice
 - One method; other method is "passed" to BeginRead
 - Compiler turns arguments/locals into fields of a compiler-defined class; both methods refer to fields so data is shared by the methods
 - BeginRead's last argument is null
 - 'using' can't be used because there really are 2 methods
 - Methods are executed via different threads
 - Avoid thread-state: thread-local storage, priority, culture, principal

APM with Anonymous Method

```
private static void ApmPatternWithAnonymousMethod(String pathname) {  
    FileStream fs = new FileStream(pathname, FileMode.Open,  
        FileAccess.Read, FileShare.Read, 8192, FileOptions.Asynchronous);  
    Byte[] data = new Byte[fs.Length];  
  
    fs.BeginRead(data, 0, data.Length, delegate(IAsyncResult result) {  
        Int32 bytesRead = fs.EndRead(result);  
        ProcessData(data);  
        fs.Close(); // No 'using', really a separate method  
        // Thread pool thread returns to pool here  
    }, null);  
  
    // Method returns after issuing async I/O request  
}
```

APM with Lambda Expressions

- C# 3.0 has Lambda Expressions
- Anonymous Method feature with more succinct syntax
- Things to notice
 - Slightly simpler syntax compared to anonymous method version
 - Compiler infers type (IAsyncResult) of result argument

APM with Lambda Expression

```
private static void ApmPatternWithLambdaExpression(String pathname) {  
    FileStream fs = new FileStream(pathname, FileMode.Open,  
        FileAccess.Read, FileShare.Read, 8192, FileOptions.Asynchronous);  
    Byte[] data = new Byte[fs.Length];  
  
    fs.BeginRead(data, 0, data.Length,  
        result => { // Type (IAsyncResult) inferred  
            Int32 bytesRead = fs.EndRead(result);  
            ProcessData(data);  
            fs.Close(); // No 'using'; really a separate method  
            // Thread pool thread returns to pool here  
        }, null);  
  
    // Method returns after issuing async I/O request  
}
```



Iterators



C# 2.0's Iterator Feature

- Lets you write single method that returns ordered sequence of values
 - Compiler turns method into IEnumerator class
 - Class object is a state machine that can be suspended/resumed!

```
private static IEnumerable<int> CountTo(int max) {  
    yield return 1;  
    for (int n = 2; n <= max; n++) yield return n;  
}
```

```
// Displays: 1 2 3 4 5  
foreach (int num in CountTo(5))  
    Console.WriteLine(num + " ");
```

Compiler Turns Iterator Into This (Approximation)

```
[CompilerGenerated]  
private sealed class <CountTo>d__0 : IEnumerator<int>, IEnumerator, IDisposable {  
  
    private int m_state = 0; // State machine location  
    private int m_current; // Last yield return value  
  
    public int max; // Arguments become fields  
    private int n; // Locals become fields  
  
    int Current { get { return m_current; } }  
  
    ...  
}
```

Compiler Turns Iterator Into This (Approximation)

```
public Boolean MoveNext() {  
    switch (m_state) {  
        case 0: m_current = 1; m_state = 1; return true;  
  
        case 1:  
            n = 2;  
            while (n <= max) {  
                m_current = n; m_state = 2; return true;  
  
                Label_ContinueLoop: n++;  
            }  
            break;  
  
        case 2: goto Label_ContinueLoop;  
    }  
    return false;  
}
```

Why Iterators are APM Greatness & Iterator Limitations

- For APM, iterators are awesome
 - Argument/local variables become fields
 - We write sequential code, compiler re-writes it allowing it to be suspended and resumed (potentially on a different thread)
 - We can use try/catch/finally, lock, using and loops (for, while, and foreach statements)
- Iterator member limitations (for reference)
 - No unsafe code
 - No out/ref parameters
 - No return statement (yield return is OK)
 - In finally block, no yield return or yield break
 - In try block with catch, no yield return (yield break is OK)
 - In catch block, no yield return (yield break is OK)

Async I/O with an Iterator

```
IEnumerator<Int32> AsyncFileRead(AsyncEnumerator ae, String pathname) {  
    using (FileStream fs = new FileStream(pathname, FileMode.Open,  
        FileAccess.Read, FileShare.Read, 8192, FileOptions.Asynchronous)) {  
  
        Byte[] data = new Byte[fs.Length];  
        fs.BeginRead(data, 0, data.Length, ae.End(), null);  
        yield return 1; // Thread returns: no blocking!  
  
        // A different thread may execute the code below!  
        Int32 bytesRead = fs.EndRead(ae.DequeueAsyncResult());  
        ProcessData(data);  
    }  
}
```

Awesome Takeaways

- Threads don't block; thread pool threads can do more work
 - Means that we don't need a lot of threads
- Multiple iterators can run simultaneously
 - Means increased scalability
- Only 1 thread at a time is in iterator code – guaranteed
 - Means no synchronization required for non-shared data
- We can easily add cool features
 - Coordination of multiple concurrent requests
 - Timeout, Cancellation, Progress reporting
 - Ability to easily update GUI components
- However, we can't use foreach to drive the iterator
 - Instead, use my AsyncEnumerator class

My AsyncEnumerator Class



AsyncEnumerator API

```
public class AsyncEnumerator {  
    // Methods called by code outside the iterator  
    public AsyncEnumerator();  
    public void Execute(IEnumerable<Int32> enumerator);  
  
    // Methods called by code inside the iterator  
    public AsyncCallback End();  
    public IAsyncResult DequeueAsyncResult();  
}
```

```
// How to Invoke an iterator  
AsyncEnumerator ae = new AsyncEnumerator();  
ae.Execute(AsyncFileRead(ae, pathname));
```

How It Works

```
IEnumerator<Int32> FileIO(AsyncEnumerator ae, String filename) {  
    using (FileStream fs = new FileStream(...)) {  
        fs.BeginWrite(..., ae.End(), null);  
        yield return 1; // Number ops to wait for  
        fs.EndWrite(ae.DequeueAsyncResult());  
    }  
}
```

AsyncEnumerator Object

- IEnumerator<Int32> m_enumerator;
- waitCount = 1
- inboxCount = 1

Inbox (List<IAsyncResult>)
IAsyncResult (Completed Write)

Cool AsyncEnumerator Features



Execute Supports the APM

- Use BeginExecute/EndExecute instead of Execute
 - Perfect for Windows Forms/WPF so GUI thread doesn't block
 - Prefer BeginExecute/EndExecute over Execute for improved responsiveness and scalability

```
public class AsyncEnumerator {  
    public IAsyncResult BeginExecute(IEnumerator<Int32> enumerator,  
        AsyncCallback callback, Object state);  
  
    public void EndExecute(IAsyncResult result);  
}
```

Use AsyncEnumerator<TResult> to Return a Value

```
public sealed class AsyncEnumerator<TResult> : AsyncEnumerator {
    public AsyncEnumerator();
    public TResult Result { get; set; }

    new public TResult Execute(IEnumerator<int32> enumerator);
    new public TResult EndExecute(IAsyncResult result);
}
```

```
// Invoking the iterator
AsyncEnumerator<String> ae = new AsyncEnumerator<String>();
String result = ae.Execute(AsyncOp(ae));
```

```
// Implementing the iterator
IEnumerator<int32> AsyncOp(AsyncEnumerator<String> ae) {
    ...
    ae.Result = ...;
}
```

SynchronizationContext

- Iterator code is executed by various thread pool threads
 - Bad for WF/WPF because GUI thread must manipulate controls
 - Bad for ASP.NET because Culture and IPincipal are not set
- AsyncEnumerator uses SynchronizationContext to fix this
- Every thread has a SC associated with it (or null)
 - If null (CUI or Service app), AE uses TP thread to MoveNext
 - If non-null, AE uses SC to call MoveNext
 - For WF: No need to call Control's Invoke/BeginInvoke
 - For WPF: No need to call Dispatcher's Invoke/BeginInvoke
 - For ASP: Culture & IPincipal are set before calling MoveNext

```
public class AsyncEnumerator {
    public SynchronizationContext SyncContext { get; set; }
}
```

AsyncEnumerator Usage Patterns



Examples of Common Usage Patterns

- Simple AsyncEnumerator Patterns
 - Issue one and process when complete
 - Issue many and process after all complete
 - Issue many and process each as each completes
 - Issue other iterators (composition/subroutines)
 - Several iterators access shared data in a thread-safe way
 - Via ReaderWriterGate & anonymous methods/lambda expressions
- AsyncEnumerator with Discard-Support Patterns
 - Issue many and process only the first; discard the rest
 - Issue many and process each until timeout
 - Issue many and process each until cancel

References

- For Anonymous Methods, Lambda Expressions, and Iterators in general
 - See the C# Language Specification
- Using these constructs with asynchronous programming
 - <http://msdn.microsoft.com/en-us/magazine/cc163323.aspx>
 - <http://msdn.microsoft.com/en-us/magazine/cc546608.aspx>
 - MSDN Magazine August 2008 Concurrent Affairs column
- Richter/Wintellect Power Threading Library & Group
 - Library: <http://wintellect.com/PowerThreading.aspx>
 - Group: <http://tech.groups.yahoo.com/group/PowerThreading/>

Discussion
