Asynchronous Programming Model

[**http://www.codemag.com/article/0305071**](http://www.codemag.com/article/0305071)

**Early Days**

* Difficult to develop and test
* Reinventing the wheel instead of adding business value to the application
* No component-oriented programming

**Requirement**

* Client should be able to issue multiple asynchronous calls
* Manage multiple asynchronous calls & distinguish between multiple methods
* Component should be able to serve multiple concurrent calls.
* Same component code for both synchronous and asynchronous invocation

When component methods have out-going parameters or return values, these parameters are not available when control returns to the client. The client should have a way of getting these parameters or results when the method completes.

**Asynchronous programming model depends on the following:**

* A delegate that represents the address of the function to be called.
* Divide the calling functionality into 2 portions or begin/end pair
* An optional callback function.

**Delegates**

public delegate int BinaryOperation(int num1,int num2);

public class BinaryOperation: System.MulticastDelegate

{

public BinaryOperation(Object target,

int methodPtr)

{...}

public virtual int Invoke(int num1,int num2)

{...}

public virtual IAsyncResult BeginInvoke(

int num1,int num2, AsyncCallback callback,

object asyncState)

{...}

public virtual int EndInvoke(

IAsyncResult result)

{...}

}

public virtual IAsyncResult BeginInvoke(

<input and input/output parameters>,

AsyncCallback callback,

object asyncState);

public virtual <return value> EndInvoke(

<output and input/output parameters>,

IAsyncResult asyncResult);

BeginInvoke() to initiate an asynchronous method invocation

EndInvoke() to manage method completion, specifically, retrieving out-going parameters and return values as well as error handling

AsyncCallback callback  (optional)- delegate object representing a reference to a callback method

AsyncState (optional) - a generic object used to pass in whatever state information is needed by the party handling the method completion

**Getting the Results Back**

public interface IAsyncResult

{

object AsyncState{get;}

WaitHandle AsyncWaitHandle{get;}

bool CompletedSynchronously{get;}

bool IsCompleted{get;}

}

Same delegate can be used for multiple asynchronous calls on the target method; caller identifies each async call by AsyncResult object

public class AsyncResult : IAsyncResult, IMessageSink

{

//IAsyncResult implementation

public object AsyncState {virtual get;}

public WaitHandle AsyncWaitHandle{virtual get;}

public bool CompletedSynchronously{virtual get;}

public bool IsCompleted {virtual get;}

//Other properties

public bool EndInvokeCalled{get; set;}

public object AsyncDelegate{virtual get;}

/\* IMessageSink implementation \*/

}

**Options for Client**

* Perform some work while the call is in progress, and then block until completion.
* Perform some work while the call is in progress, and then poll for completion.
* Receive notification when the method has completed. The notification will be in the form of a callback on a client-provided method. The callback should contain information identifying which method has just completed and its return values.
* Perform some work while the call is in progress, and then wait only for a pre-determined amount of time and then stop waiting, even if the method execution has not completed yet.
* Wait simultaneously for completion of multiple methods. The client can choose to wait for any or all of the pending calls to complete.

**Listing 1**: Simple asynchronous execution sequence

Calculator calc = new Calculator();

BinaryOperation oppDel;

oppDel = new BinaryOperation(calc.Add);

IAsyncResult asyncResult1 = oppDel.BeginInvoke(2,3,null,null);

IAsyncResult asyncResult2 = oppDel.BeginInvoke(4,5,null,null);

/\*Do some work \*/

int result;

result = oppDel.EndInvoke(asyncResult1);

Debug.Assert(result == 5);

result = oppDel.EndInvoke(asyncResult2);

Debug.Assert(result == 9);

**Listing 2**: You can only pass the IAsyncResult object to the same delegate used to invoke the call

Calculator calc = new Calculator();

BinaryOperation oppDel1;

BinaryOperation oppDel2;

oppDel1 = new BinaryOperation(calc.Add);

oppDel2 = new BinaryOperation(calc.Add);

IAsyncResult asyncResult = oppDel1.BeginInvoke(2,3,null,null);

//This will result in InvalidOperationException

oppDel2.EndInvoke(asyncResult);

**Listing 3**: Use the AsyncDelegate property of AsyncResult to access the original delegate

using System.Runtime.Remoting.Messaging;

public class CalculatorClient

{

IAsyncResult m\_AsyncResult;

public void AsynchAdd()

{

Calculator calc = new Calculator();

DispatchAdd(calc,2,3);

/\* Do some work \*/

int result = GetResult();

Debug.Assert(result == 5);

}

protected void DispatchAdd(Calculator calc,int num1,int num2)

{

BinaryOperation oppDel;

oppDel = new BinaryOperation(calc.Add);

m\_AsyncResult = oppDel.BeginInvoke(2,3,null,null);

}

protected int GetResult()

{

int result = 0;

//Obtain original delegate

AsyncResult asyncResult = (AsyncResult)m\_AsyncResult;

BinaryOperation oppDel = (BinaryOperation)asyncResult.AsyncDelegate;

Debug.Assert(asyncResult.EndInvokeCalled == false);

result = oppDel.EndInvoke(m\_AsyncResult);

return result;

}

}

**Listing 4**: Use IAsyncResult.AsyncWaitHandle.WaitOne() to block until method completion

Calculator calc = new Calculator();

BinaryOperation oppDel;

oppDel = new BinaryOperation(calc.Add);

IAsyncResult asyncResult = oppDel.BeginInvoke(2,3,null,null);

/\*Do some work \*/

asyncResult.AsyncWaitHandle.WaitOne(); //This may block

int result;

result = oppDel.EndInvoke(asyncResult); //This will not block

Debug.Assert(result == 5);

**Listing 5**: Use WaitOne() to specify wait timeout

Calculator calc = new Calculator();

BinaryOperation oppDel;

oppDel = new BinaryOperation(calc.Add);

IAsyncResult asyncResult = oppDel.BeginInvoke(2,3,null,null);

while(asyncResult.IsCompleted == false)

{

asyncResult.AsyncWaitHandle.WaitOne(10,false); //This may block

/\*Do some work \*/

}

int result;

result = oppDel.EndInvoke(asyncResult); //This will not block

**Listing 6**: Wait for completion of multiple methods

Calculator calc = new Calculator();

BinaryOperation oppDel1;

BinaryOperation oppDel2;

oppDel1 = new BinaryOperation(calc.Add);

oppDel2 = new BinaryOperation(calc.Add);

IAsyncResult asyncResult1 = oppDel1.BeginInvoke(2,3,null,null);

IAsyncResult asyncResult2 = oppDel2.BeginInvoke(4,5,null,null);

WaitHandle[] handleArray =

{asyncResult1.AsyncWaitHandle,asyncResult2.AsyncWaitHandle};

WaitHandle.WaitAll(handleArray);

int result;

//These calls to EndInvoke() will not block

result = oppDel1.EndInvoke(asyncResult1);

Debug.Assert(result == 5);

result = oppDel2.EndInvoke(asyncResult2);

Debug.Assert(result == 9);

**Listing 7**: Manage asynchronous completion using a completion method

public class CalculatorClient

{

public void AsynchAdd()

{

Calculator calc = new Calculator();

AsyncCallback callback;

BinaryOperation oppDel;

callback = new AsyncCallback(OnMethodCompletion);

oppDel = new BinaryOperation(calc.Add);

oppDel.BeginInvoke(2,3,callback,null);

}

protected void OnMethodCompletion(IAsyncResult asyncResult)

{

int result = 0;

AsyncResult resultObj = (AsyncResult)asyncResult;

Debug.Assert(resultObj.EndInvokeCalled == false);

BinaryOperation oppDel = (BinaryOperation)resultObj.AsyncDelegate;

result = oppDel.EndInvoke(asyncResult);

Trace.WriteLine("Operation returned " + result.ToString());

}

}

**Listing 8**: Pass additional parameter to the callback completion method using a state object

public class CalculatorClient

{

public void AsynchAdd()

{

Calculator calc = new Calculator();

AsyncCallback callback;

BinaryOperation oppDel;

int asyncState = 4;

callback = new AsyncCallback(OnMethodCompletion);

oppDel = new BinaryOperation(calc.Add);

oppDel.BeginInvoke(2,3,callback,asyncState);

}

protected void OnMethodCompletion(IAsyncResult asyncResult)

{

int asyncState;

asyncState = (int)asyncResult.AsyncState;

Debug.Assert(asyncState == 4);

/\*Rest of the callback \*/

}

}

**Listing 9**: Asynchronous Web-service call with a completion callback

public class CalculatorWebServiceClient

{

public void AsynchAdd()

{

//Calculator here is the auto-generated wrapper class

Calculator calc;

calc = new Calculator();

AsyncCallback callback;

callback = new AsyncCallback(OnMethodCompletion);

calc.BeginAdd(2,3,callback,null);

}

protected void OnMethodCompletion(IAsyncResult asyncResult)

{

//Calculator here is the auto-generated wrapper class

Calculator calc;

calc = new Calculator();

int result;

result = calc.EndAdd(asyncResult);

Trace.WriteLine("Operation returned " + result.ToString());

}

}

**Listing 10**: Asynchronous stream read with a completion callback

public class FileStreamClient

{

public void AsynchRead()

{

bool useAsync = true;

Stream stream = new FileStream("MyFile.bin",FileMode.Open,FileAccess.Read,

FileShare.None,1000,useAsync);

AsyncCallback callback;

callback = new AsyncCallback(OnMethodCompletion);

stream.BeginRead(m\_Array,0,10,callback,null);

stream.Close();

}

protected void OnMethodCompletion(IAsyncResult asyncResult)

{

bool useAsync = true;

Stream stream = new FileStream("MyFile.bin",FileMode.Open,FileAccess.Read,

FileShare.None,1000,useAsync);

int bytesRead = stream.EndRead(asyncResult);

stream.Close();

//Access m\_Array

}

Byte[] m\_Array = new Byte[2000];

**Listing 11**: Asynchronous error handling

public class CalculatorClient

{

public void AsyncDivide()

{

Calculator calc = new Calculator();

AsyncCallback callback;

BinaryOperation oppDel;

callback = new AsyncCallback(OnMethodCompletion);

oppDel = new BinaryOperation(calc.Divide);

oppDel.BeginInvoke(2,0,callback,null);

}

private void OnMethodCompletion(IAsyncResult asyncResult)

{

AsyncResult resultObj = (AsyncResult)asyncResult;

Debug.Assert(resultObj.EndInvokeCalled == false);

BinaryOperation oppDel = (BinaryOperation)resultObj.AsyncDelegate;

try

{

int result = 0;

result = oppDel.EndInvoke(asyncResult);

Trace.WriteLine("Operation returned " + result.ToString());

}

catch(DivideByZeroException exception)

{

Trace.WriteLine(exception.Message);

}

}

}