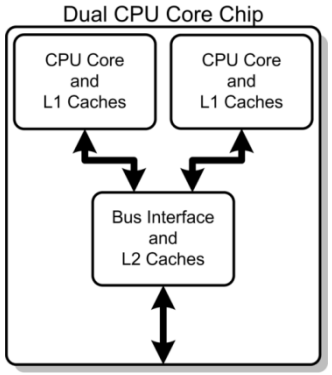
Asynchronous Calls in .Net

Agenda

- Why use Asynchronous Calls
- Revisiting Delegates
- Asynchronous Call Programming Models
- Reactive Extensions
- Asynchrony with async/await

Why use Asynchronous Calls

- Asynchronous calls allow you to improve:
  - availability
  - throughput
  - performance
  - scalability
- But asynchronous calls will only enhance your application if you have blocking calls - or the PC running your code has a multiple-Core CPU
  - The asynchronous method invocation mechanics itself isn't faster!



Requirements for an Asynchronous Mechanism

- The same component code should be used for both synchronous and asynchronous invocations.
- The client should be the one to decide whether to call a component synchronously or asynchronously.
  - This, in turn, implies that the client will have different code for invoking the call synchronously and asynchronously.
- The client should be able to issue multiple asynchronous calls and have multiple asynchronous calls in progress.
  - And it should be able to distinguish between multiple method completions.
  - The component should be able to serve multiple concurrent calls.
- If component methods have output parameters or return values, then the client should have a way to access them when the method completes.
  - Similarly, errors on the component's side should be propagated to the client side.
- The asynchronous-calls mechanism should be straightforward and simple to use.

Revisiting Delegates

- A delegate is nothing more than a type-safe method reference

```
public class Calculator
{
   public int Add(int argument1,int argument2)
   {
      return argument1 + argument2;
   }
   public int Subtract(int argument1,int argument2)
   {
      return argument1 - argument2;
   }
}
```

```
// Instead of calling the Add( ) method directly,
// you can define a delegate called BinaryOperation:
public delegate int BinaryOperation(int argument1,int argument2);

// and use BinaryOperation to invoke the method:
Calculator calculator = new Calculator( );
BinaryOperation oppDel = new BinaryOperation(calculator.Add);
int result = 0;
result = oppDel(2,3);
Debug.Assert(result == 5);
```

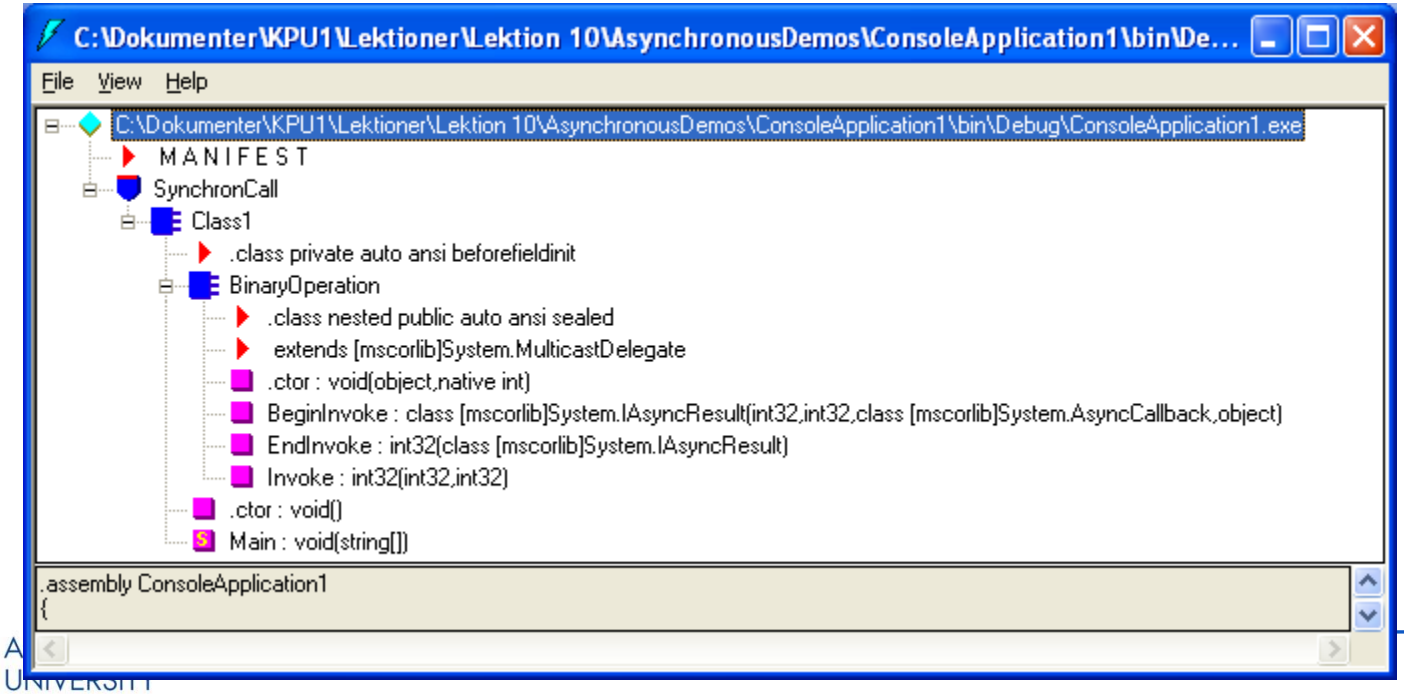
Synchronous call: the delegate blocks the caller until all target methods return

Delegate Dissection

```
public delegate int BinaryOperation(int argument1,int argument2);
```

- For ↑ the compiler generates this class definition:

```
public sealed class BinaryOperation : MulticastDelegate
{
        public BinaryOperation(Object target,int methodPtr) {...}
        public virtual int Invoke(int argument1,int argument2) {...}
        public virtual IAsyncResult BeginInvoke(int argument1,int argument2,
                AsyncCallback callback,object asyncState) {...}
        public virtual int EndInvoke(IAsyncResult result) {...}
}
```



Synchronous use of delegate

- When you use the delegate simply to invoke a method, such as in this code:

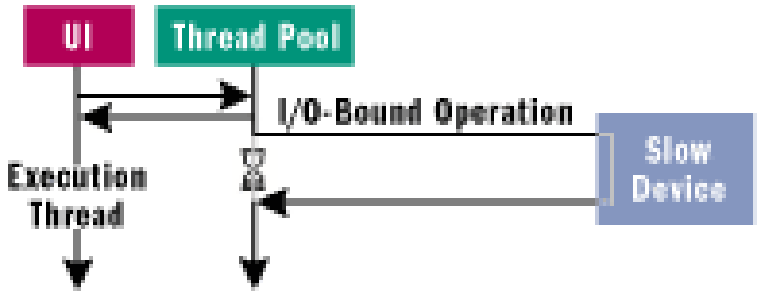
```
Calculator calculator = new Calculator( );
BinaryOperation oppDel = new BinaryOperation(calculator.Add);
int result = 0;
result = oppDel(2,3);
```

- The compiler converts the call to oppDel(2,3) to a call to the Invoke( ) method.
  - The Invoke( ) method blocks the caller, executes the method on the caller's thread, and returns control to the caller.

Asynchronous use of delegate

.NET's way of supporting asynchronous calls:

- BeginInvoke( ) initiates an asynchronous method invocation.
  - Activates a worker thread from the *.NET thread pool* and executes the delegate function on that thread.
    - It's faster to use an existing thread than to create a new!
  - The calling client is blocked for only the briefest moment.
- EndInvoke( ) manages method completion
  - specifically, retrieving output parameters and return values, and error handling.



Consequences for the target object

- The target object (the server object) is unaware that the client is calling it asynchronously.
- As a result every .Net class supports both synchronous and asynchronous method invocating!
- The compiler will compile it
  - But there is no guarantee it will work
- The target objects code must be thread safe!
  - Eg. synchronized access to shared date

Asynchronous Call Programming Models

When the client issues an asynchronous method call, it can then choose to:

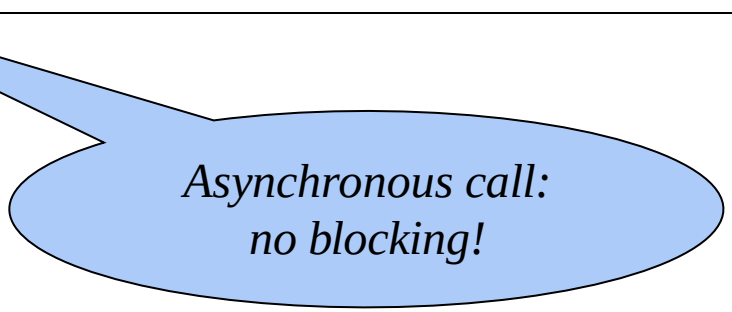
- Start and forget
  - When you don't care about the result.
- Block
  - Perform some work while the call is in progress, then block until completion.
- Poll
  - Perform some work while the call is in progress, and then poll for completion.
- Wait
  - Perform some work while the call is in progress, wait for a predetermined amount of time, and then stop waiting, even if the method execution has not yet completed.
  - Wait simultaneously for completion of multiple methods. The client can choose to wait for all or any of the pending calls to complete.
- Use a Callback
  - 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.

.NET offers all these options to clients by use of delegates

Start and forget

- If you have no interest in the result and no interest in a callback method or state information, you would write:

```
Calculator calculator = new Calculator(  );
BinaryOperation oppDel  = new BinaryOperation(calculator.Add);
oppDel.BeginInvoke(2,3,null,null);
```



Block

- Perform some work while the call is in progress, then block until completion

```
int result;
IAsyncResult asyncResult1 = oppDel.BeginInvoke(2,3,null,null);
IAsyncResult asyncResult2 = oppDel.BeginInvoke(4,5,null,null);

/* Do some work */

result = oppDel.EndInvoke(asyncResult1);
Debug.Assert(result == 5);

result = oppDel.EndInvoke(asyncResult2);
Debug.Assert(result == 9);
```

- The returned IAsyncResult object uniquely identifies the method that was invoked using BeginInvoke
- The primary use of EndInvoke( ) is to retrieve any output parameters as well as the method's return value.
- **EndInvoke( ) can be called only once for each asynchronous operation!**
- Calling BeginInvoke( ) when the delegate's list contains more than one target will result in an ArgumentException being thrown!

Poll

- Perform some work while the call is in progress, and then poll for completion

```
Calculator calculator = new Calculator();
BinaryOperation oppDel = new BinaryOperation(calculator.Add);

IAsyncResult asyncResult1 = oppDel.BeginInvoke(2, 3, null, null);
// Polling
while (!asyncResult1.IsCompleted)
   {
   /* Do some work */
   }

result = oppDel.EndInvoke(asyncResult1);    // No Blocking
```

Wait

- Perform some work while the call is in progress, wait for a predetermined amount of time, and then stop waiting, even if the method execution has not yet completed.
- Wait simultaneously for completion of multiple methods. The client can choose to wait for all or any of the pending calls to complete.

```
IAsyncResult asyncResult1 = oppDel.BeginInvoke(2, 3, null, null);
/* Do some work */

asyncResult1.AsyncWaitHandle.WaitOne(100, false);

if (asyncResult1.IsCompleted)
   {
   result = oppDel.EndInvoke(asyncResult1);   // No Blocking
   Debug.Assert(result == 5);
   Console.WriteLine("Result: " + result);
   }
else
   Console.WriteLine("Calculation not finished!");
```

Multiple Wait

- The main attraction of WaitHandles is their ability to wait for completion of multiple asynchronous methods

```
{
   Calculator calculator = new Calculator();
   BinaryOperation oppDel = new BinaryOperation(calculator.Add);
   int result = 0;

   IAsyncResult asyncResult1 = oppDel.BeginInvoke(2, 3, null, null);
   IAsyncResult asyncResult2 = oppDel.BeginInvoke(4, 5, null, null);
   WaitHandle[] handleArray = {asyncResult1.AsyncWaitHandle,
                               asyncResult2.AsyncWaitHandle};
   /* Do some work */
   WaitHandle.WaitAll(handleArray);

   result = oppDel.EndInvoke(asyncResult1);
   Console.WriteLine("Result: " + result);
   result = oppDel.EndInvoke(asyncResult2);
   Console.WriteLine("Result: " + result);
   }
```

WaitHandle.WaitAny() is another possibility

Use a Callback

- Receive notification when the method has completed.
  - The notification will be in the form of a callback on a client-provided method.
  - The callback contain information identifying which method has just completed and its return values.
  - The callback is run on the background thread!!!

```
{
    Calculator calculator = new Calculator();
    BinaryOperation oppDel = new BinaryOperation(calculator.Add);
    int asyncState = 27;
    oppDel.BeginInvoke(2, 3, OnMethodCompletion, asyncState);
    /* Do some work */
}

void OnMethodCompletion(IAsyncResult asyncResult)
{
    int asyncState = (int)asyncResult.AsyncState;
    Debug.Assert(asyncState == 27);
    AsyncResult resultObj = (AsyncResult)asyncResult;
    BinaryOperation oppDel = (BinaryOperation)resultObj.AsyncDelegate;
    int result = oppDel.EndInvoke(asyncResult);
    Console.WriteLine("Operation returned " + result);
}
```

Error Handling

- What happens if an exception is thrown in the asynchronous method?

- .Net catches the exception and re-throws that exception object when `EndInvoke()` is called.
- And if a callback method is provided, .Net calls the callback immediately after the exception is thrown.

- So remember to put a try-catch block around `EndInvoke()` calls.

Asynchronous Operations Without Delegates

- Most framework classes where you expect long response times like disk operations, network access, web requests etc. have build in support for asynchronous operations.
- Those classes will have functions with names and signatures similar to delegates:
  - Begin<*operation*>
  - End<*operation*>
- *Eg.:*

```
Public void AsyncRead()
{
  bool useAsync = true;
  Stream stream = new FileStream("MyFile.bin", FileMode.Open,
                  FileAccess.Read, FileShare.None, 1000, useAsync);
  using(stream)
  {
    stream.BeginRead(m_Array, 0, 10, OnMethodCompletion, null););
  }
}
```

REACTIVE EXTENSIONS

Curing your Asynchronous Programming Blues

- The Reactive Extensions (Rx) is a library to compose asynchronous and event-based programs using observable collections and LINQ-style query operators.
- You can represent multiple asynchronous data streams as an observable collections (implements IObservable<T> interface) e.g.:
  - stock quote,
  - tweets,
  - computer events,
  - web service requests,
  - etc.
- And subscribe to the event stream using the IObserver<T> interface.
- The IObservable<T> interface notifies the subscribed IObserver<T> interface whenever an event occurs.

Use LINQ To Query Rx

- Because observable sequences are data streams, you can query them using standard LINQ query operators implemented by the Observable type.
- Thus you can filter, project, aggregate, compose and perform time-based operations on multiple events easily by using these static LINQ operators.
- In addition, there are a number of other reactive stream specific operators that allow powerful queries to be written.
- Cancellation, exceptions, and synchronization are also handled gracefully by using the extension methods provided by Rx.

IObservable

- The two core interfaces around which Rx is built:

```
public interface IObservable<out T>
{
    IDisposable Subscribe(IObserver<T> observer);
}
public interface IObserver<in T>
{
    void OnCompleted();
    void OnError(Exception error);
    void OnNext(T value);
}
```

- To receive notifications from an observable collection you uses the Subscribe method to hand it an IObserver<T> object.
- The Subscribe method returns an IDisposable object that acts as a handle for the subscription.
- Calling Dispose on this object will detach the observer from the source such that notifications are no longer delivered.

Observing the Passing of Time

```
static void Main(string[] args)
{
    IObservable<long> source =
                    Observable.Interval(TimeSpan.FromSeconds(1));
    //IObserver<int> handler = null;

    IDisposable subscription = source.Subscribe(
            x => Console.WriteLine("OnNext: {0}", x),
            ex => Console.WriteLine("OnError: {0}", ex.Message),
            () => Console.WriteLine("OnCompleted")
            );

    Console.WriteLine("Press ENTER to unsubscribe...");
    Console.ReadLine();
    subscription.Dispose();
}
```

ASYNCHRONY WITH ASYNC/AWAIT

New in C# 5

Task-based Asynchronous Pattern

- .Net v. 4.5 exposes asynchronous versions of a great many operations, following a newly-documented **Task-based asynchronous pattern**.
- The WinRT framework used to create Windows Store Apps enforces asynchrony for all long-running (or potentially long-running) operations.

- In short:

the future is asynchronous

Asynchronous Functions

- C# 5 introduces the concept of an **asynchronous function** in the language.
- This is always either a method or an anonymous function which is declared with the `async` modifier,
  - and can include `await` expressions.

```
private async void btnGetHtml_Click(object sender,
                                    RoutedEventArgs e)
{
    tbxLength.Text = "Fetching...";
    string url = tbxUrl.Text;
    HttpClient client = new HttpClient();
    string text = await client.GetStringAsync(url);
    tbxLength.Text = text.Length.ToString();
}
```

Task<TResult>

- If we split the call to HttpClient.GetStringAsync from the await expression we can see the types involved:

```
HttpClient client = new HttpClient();
Task<string> task = client.GetStringAsync(url);
string text = await task;
```

- The type of `GetStringAsync` is `Task<string>`.
- But the type of the `await` task expression is just string.
  - **The `await` expression performs an "unwrapping" operation.**

await

- A new language construct that allows you to "await" an asynchronous operation.
- **The main purpose of await is to avoid blocking while we wait for a time-consuming operation to complete.**
- This "awaiting" looks very much like a normal blocking call, in that the rest of your code won't continue until the operation has completed
  - ***but it manages to do this without actually blocking the currently executing thread***.

await uncovered

- Await's trick is the method actually returns as soon as we hit the await expression.
  - Up until that point, it executes synchronously on the UI thread just as any other event handler.
- When await is reached, the code checks whether the result is already available, and if it's not it schedules a **continuation** to be executed when the awaited operation has completed.
- The continuation is executed on the GUI thread.

```
private async void btnGetHtml_Click(object s, RoutedEventArgs e)
{
    string url = tbxUrl.Text;
    HttpClient client = new HttpClient();
    string text = await client.GetStringAsync(url);
    tbxLength.Text = text.Length.ToString();
}
```

Continuation

Return types from async methods

- Async methods are limited to the following return types:
  - `void`
  - `Task`
  - `Task<TResult>`

- Task and Task<TResult> types, represent an operation which may not have completed yet.
  - Task<TResult> represents an operation which returns a value of type T.
  - Task will not produce a result.

- Note:
  - You cannot use the out or ref modifiers on the parameters to an async operation declaration.

Async on WinRT

- The Windows Runtime was built with the concept of asynchronous operations.
- It was also built with the idea that these asynchronous operations must work not only for C#, but also for JavaScript, C++, and any number of other, very different languages.
- For those reasons, the team didn't take a dependency on the Task Parallel Library in .NET, but instead created an interface-based approach consistent with the rest of the Windows Runtime.
- Every asynchronous operation implements, at a minimum, the `IAsyncInfo` interface.

Async support interfaces in WinRT.

Interface	Description
IAsyncAction	The most basic async support interface beyond IAsyncInfo. This defines an asynchronous method which does not have a return type.
IAsyncActionWithProgress<TProgress>	An interface which supports progress reporting of a supplied type.
IAsyncOperation<TResult>	Interface which supports an asynchronous method which has a return value.
IAsyncOperationWithProgress<TResult, TProgress>	Interface which supports an asynchronous method with both a return value, and progress reporting.

References and Links

- **Task-based Asynchronous Pattern (async – await)**
  http://www.microsoft.com/en-us/download/details.aspx?id=19957
- Reactive Extensions
  http://msdn.microsoft.com/en-us/data/gg577609
- .NET Delegates: Making Asynchronous Method Calls in the .NET Environment
  http://msdn.microsoft.com/en-us/magazine/cc301332.aspx