Distributed Computing

Lecture 4: Asynchronous Communication

Last updated: 11 August 2023

This Week

- Last week, we discussed Multi-tier architecture and started the discussion on Asynchronous communication
- This week we will explore deeper into Asynchronous communication
- We will illustrate with .NET
  - the underlying concepts apply to any language/framework

Blocking Calls and Distributed Computing

- When a function is called, the caller typically must wait (block) until the function completes & returns
- In a distributed computing system, blocking for a remote call can easily be a waste of resources
  - Especially if it is a call that could take a while
  - eg: Client waits while server performs a long job
  - Not utilising resources properly/efficiently there!

Synchronous vs Aynchronous

- Synchronous invocation = blocking call
  - Serial processing
  - Control is passed to called function
  - Caller cannot continue until called function returns
- Asynchronous invocation = non-blocking call
  - Parallel processing (or at least one way to implement it)
  - Control is returned immediately to the caller
  - Called function carries on in the background
  - At some later time, caller retrieves function’s return value

Asynchronous Calls

- Reasons for using asynchronous calls
  - Maintain GUI responsiveness
  - Utilise resources of caller more efficiently
    (eg: continue doing other work during a long-running call)
- Asynchronity traditionally implemented with threads
  - Spin off a worker thread to perform call 'in the background'

An Interlude: Threads

– Thread: an independent line of code execution within a process
  - A single process can have one or many threads within it
    - » The first thread starts at `main()`
    - » Subsequent threads can be created and start at a given function
  - All threads share the same code and memory (process space)
    - » A thread can run any function and access any data in the process
  - Each thread is scheduled independently by the O/S
    - » Thus they run concurrently (in parallel)
    - » Different threads can be updating the same data at the same time
    - » This leads to the possibility of data corruption: lost updates, etc

Threads to Asynchronous Calls

- The problem with threads is that they are hard to get working bug-free
  - The main way threads 'communicate' with each other is via updating shared data
  - But do this wrong and the data will get corrupted
    - …and be too conservative and lose all parallel performance
- Microsoft created .NET-specific approaches to (largely) take care of the threads for the user
  - Event-based: notifications sent on call completion
  - Threads still used, but hidden from the user

When to use Asynchronous Calls?

- Naïve answer: for any potentially long-running call
- But *every* RPC call is potentially long-running
  - Network/server failure only detected after timeouts expire
  - Making every RPC call asynchronous increases code complexity, just on the *chance* a network failure occurs
- So use asynchronous calls only on functions that are *expected* to take a long time
  - eg: heavy processing tasks, intensive disk I/O tasks, etc
  - For GUI clients, responsiveness is also a key issue

.NET Asynchronous Call Methods

- Blocking/polling with delegates (a special .NET concept)
  - Fairly unsophisticated: Call-DoOtherWork-WaitForReturn
  - But also simple and threads fully taken care of for you
- Completion callbacks with delegates
  - Have .NET callback your own method on completion
  - Very useful, but thread syncing is needed
- Background (worker) thread + function call
  - Manual method - create thread, do call in background
  - Flexible, but must deal directly with threads yourself

Delegates

- .NET delegates used extensively for notifications:
  - Callbacks (eg: for asynchronous completion callbacks)
  - Events (eg: OnClick event handler for Buttons)
- Delegates are like a function pointer in C/C++
  - Define a delegate, like declaring a function prototype
  - Create a variable whose type is the delegate
  - Then 'point' this variable at a real function and call the function through the delegate variable
    - » Passing this variable around is then like passing the function
- Java has no equivalent

Delegate Example

```
namespace MyExample {
public class Calculator ← Can be remote but this example won't use RPC
    {
       public int Add(int num1, int num2) { return num1 + num2; }
       public int Subtract(int num1, int num2) { return num1 - num2; }
       public int Multiply(int num1, int num2) { return num1 * num2; }
    }
}
------------------------------------------------------------------------------------------

public delegate int BinaryOperation(int operand1, int operand2); ← Delegate definition

public class DelegateTester
{
   public static void Main() {
       BinaryOperation binOp;                               ← Declare a variable of delegate type
       Calculator calc;

       calc = new Calculator();
       binOp = calc.Add;                  ← Point at Add() function of calc (the object, NOT the class!)
       System.Console.WriteLine("1 + 2 = " + binOp(1, 2)); ← binOp substitutes for calc.Add
       binOp = calc.Subtract;                              ← Point at calc.Subtract()
       System.Console.WriteLine("1 - 2 = " + binOp(1, 2)); ← binOp now substitutes for calc.Subtract
    }
}
```

Asynchronity with Blocking/Polling

- Each delegate has a BeginInvoke( ) and EndInvoke( )
  - Generated by compiler to match delegate prototype
- Use BeginInvoke( ) to start an asynchronous call
  - Call is started on a new thread
- Use EndInvoke( ) to retrieve results
  - Blocks if call hasn't completed yet
    - » NOTE: It is *still* an async call: block only occurs on EndInvoke()
  - Thread creation/syncing is done for you by .NET
- Can check if call completed before calling EndInvoke
  - In other words, can do polling – can be inefficient
  - asyncObj.IsCompleted in loop (waste processing power)

Async Call w/Blocking Example

– An example of running two calls in parallel (not RPC):

```
public delegate int BinaryOperation(int operand1, int operand2);

public class CalcClient
{
   public static void Main() {
      BinaryOperation addDel;
      Calculator calc = new Calculator();          ← Could have set up an RPC for this - has no effect on async calls
      IAsyncResult asyncObj1, asyncObj2;           ← Each IAsyncResult object tracks info for a single async call
      int iAddResult1, iAddResult2;

      addDel = calc.Add;                           ← Point delegate at function to be called asynchronously

      asyncObj1 = addDel.BeginInvoke(1, 2, null, null);      ← Start worker threads
      asyncObj2 = addDel.BeginInvoke(3, 4, null, null);        for async calls in parallel

      // Block, waiting for all async calls to complete
      iAddResult1 = addDel.EndInvoke(asyncObj1);   ← Retrieve return value from each async call tracked by the
      iAddResult2 = addDel.EndInvoke(asyncObj2);     appropriate asyncObjX

      asyncObj1.AsyncWaitHandle.Close();           ← Clean up
      asyncObj2.AsyncWaitHandle.Close();

      System.Console.WriteLine("1 + 2 = " + iAddResult1);
      System.Console.WriteLine("3 + 4 = " + iAddResult2);
   }
}
```

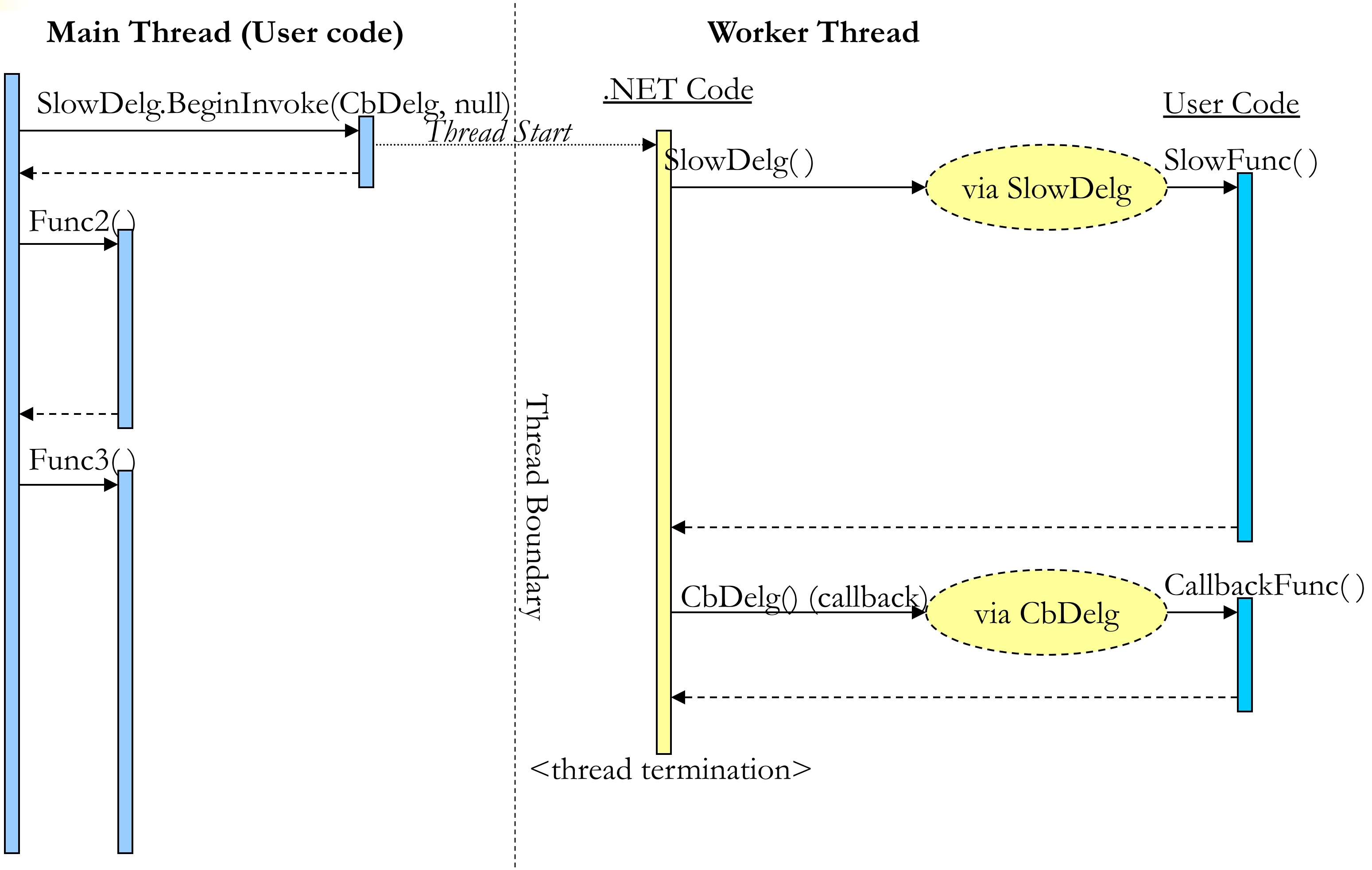
Blocking/Polling Uses

- Block/polling is 'half-synchronous'
  - Need to 'guess' when async call completes
  - But much simpler than async completion-callbacks
  - And don't have to mess with thread synchronisation
- Blocking/polling has its uses
  - Blocking is good for firing off several calls where you must wait for all calls to complete before continuing
    - » eg: for executing a task in parallel – blocking async calls are one of the few relatively elegant methods to facilitate this
  - Polling can be used to facilitate similar scenarios that have greater complexity/interplay between parallel calls

Asynchronity with Completion Callbacks

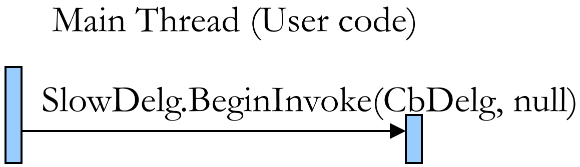
- A callback is where one of your own methods is called in response to a call that you made earlier
- .NET *completion callbacks* are callbacks designed to notify you when an asynchronous call has *completed*:
  - Approach: Define *two* delegates:
    - » A: points at the function to run asynchronously via BeginInvoke
    - » C: points at the function that is to be called *on completion* of X
  - Normal async call, but pass C as well:
    - » A.BeginInvoke(…, C, null) tells .NET to callback on C when A is done
  - When call completes, .NET notifies your method via C
    - » On the same *background worker thread* that the call executed in
    - » Use EndInvoke( ) in callback fn C to retrieve return value

Asynchronity with Completion Callbacks

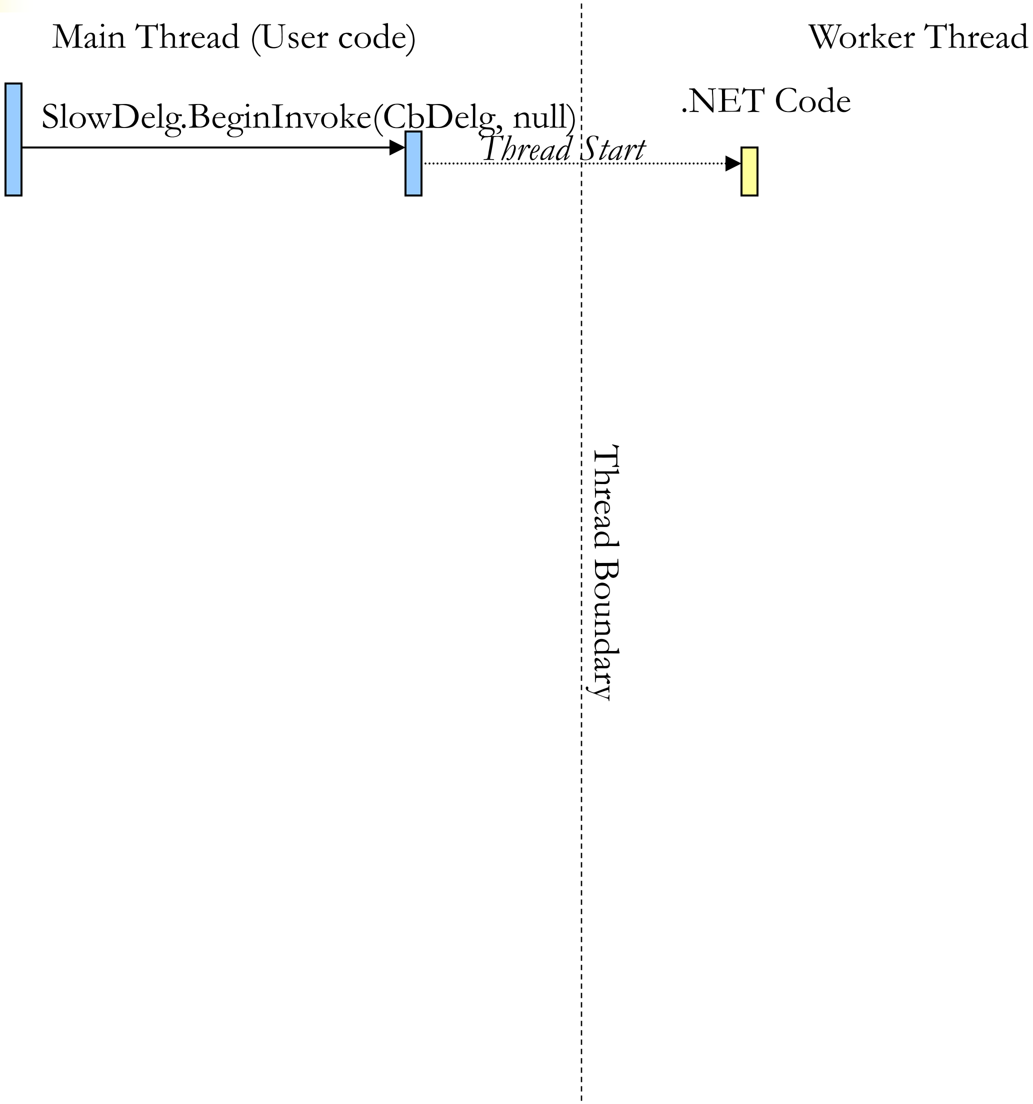


– Let's animate this over the next few slides…

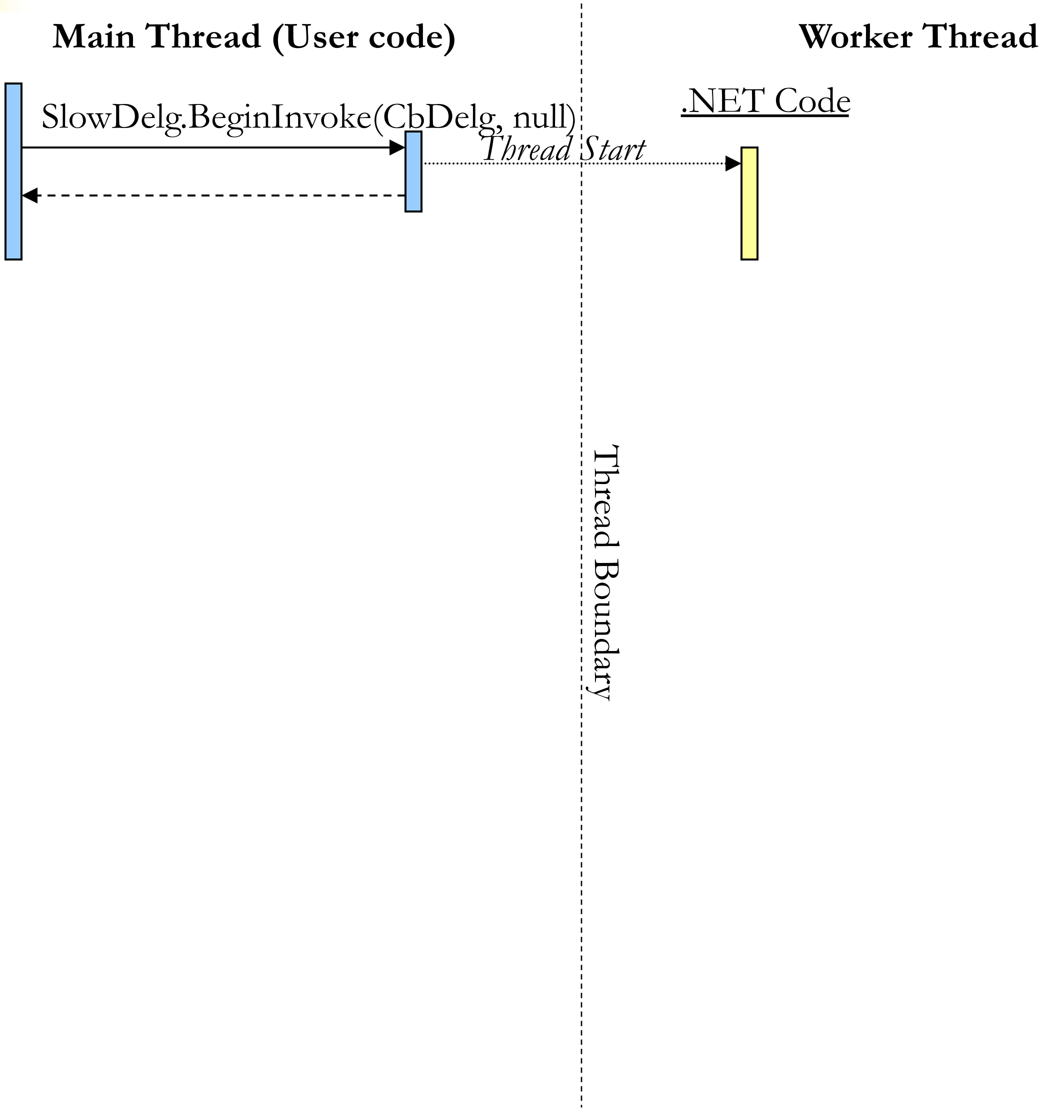
Asynchronity with Completion Callbacks



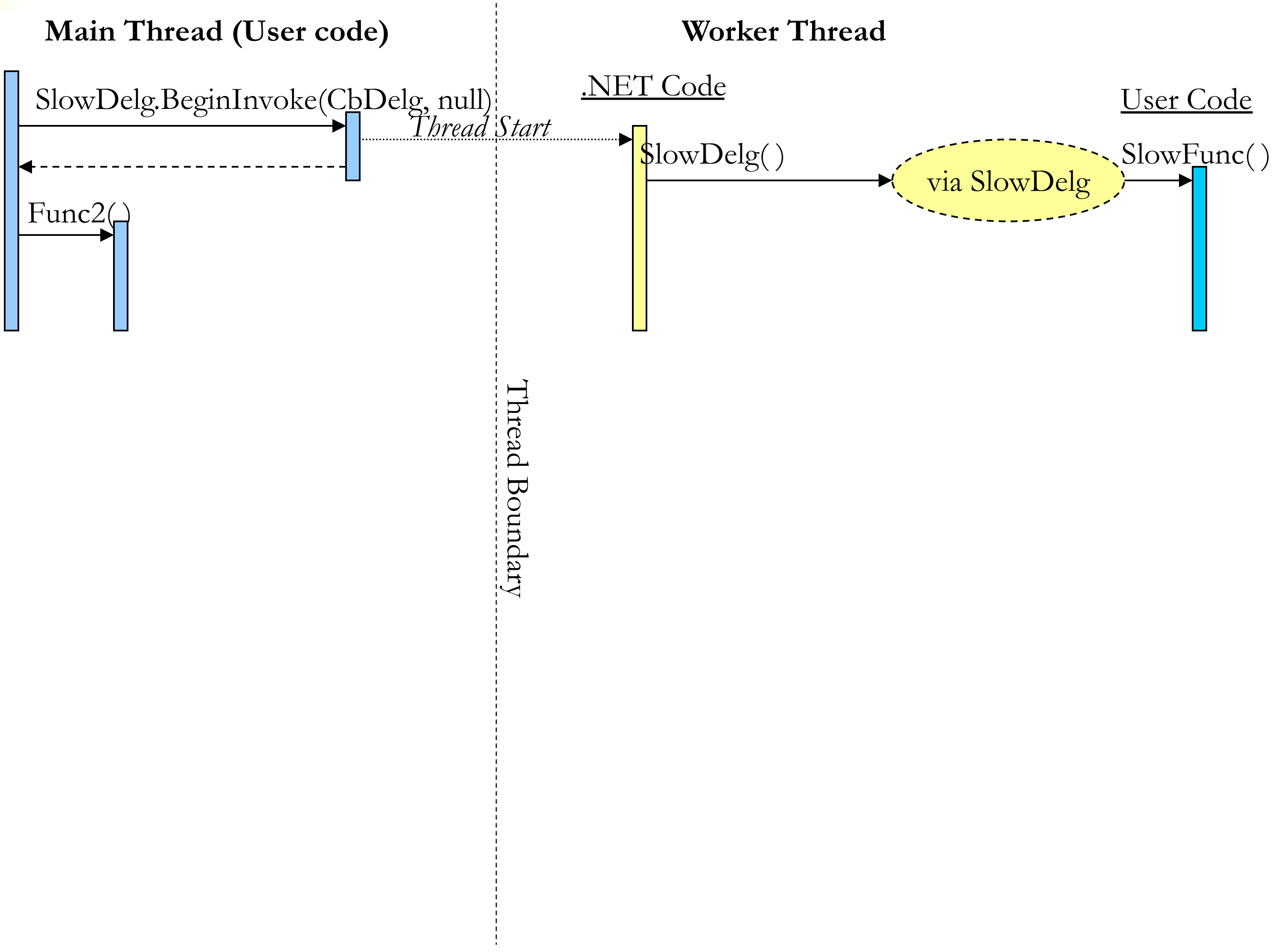
Asynchronity with Completion Callbacks



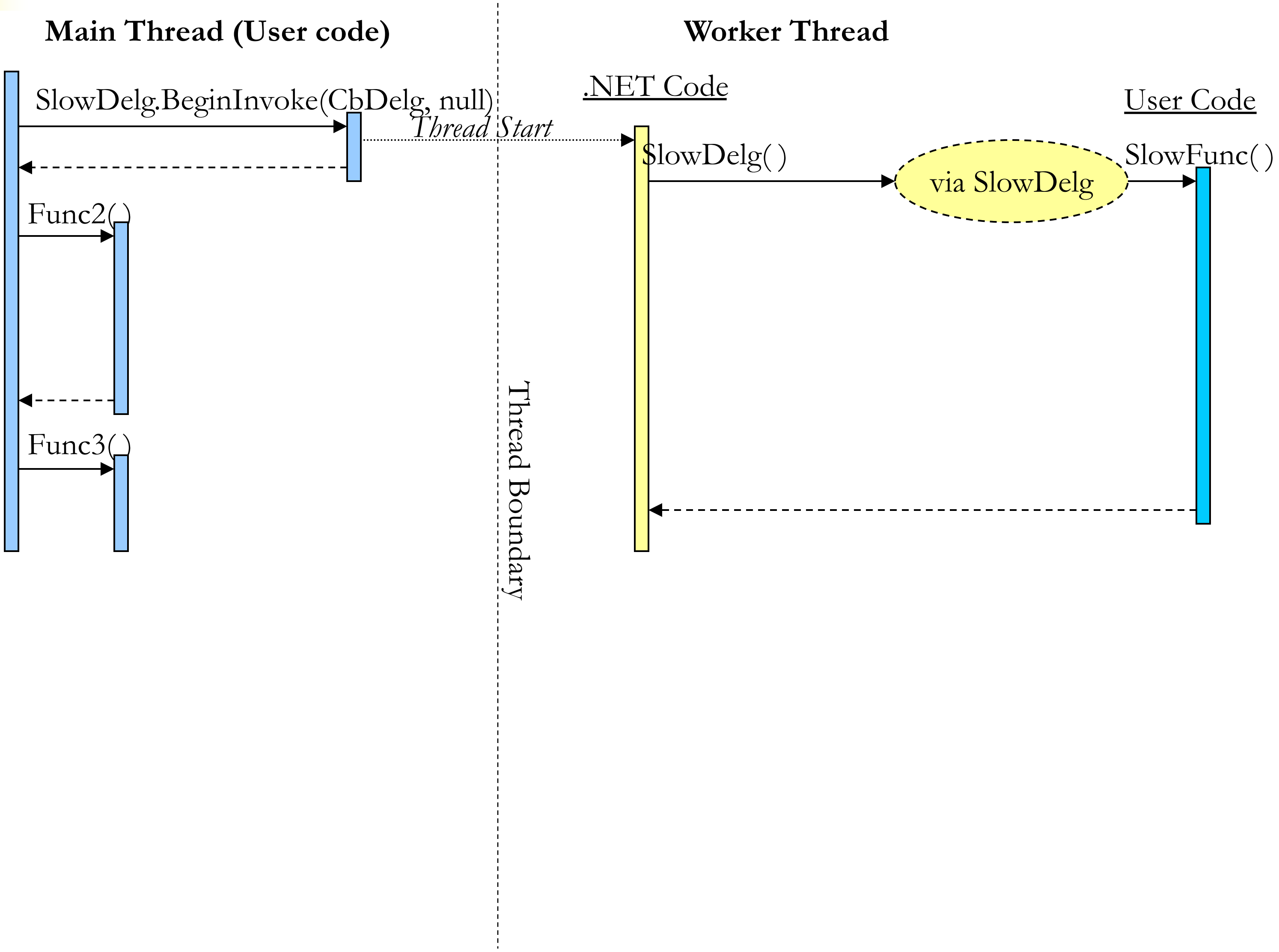
Asynchronity with Completion Callbacks



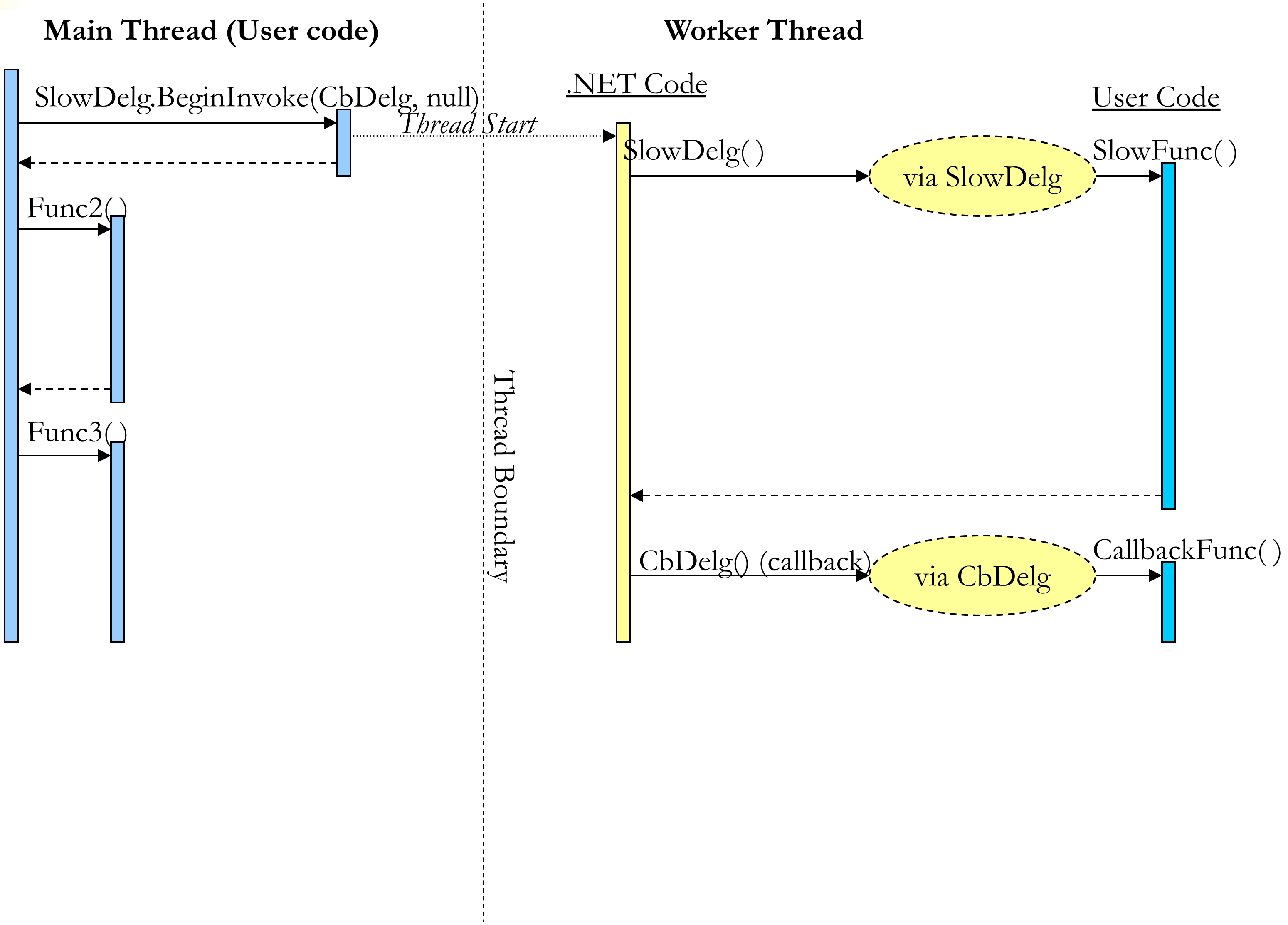
Asynchronity with Completion Callbacks



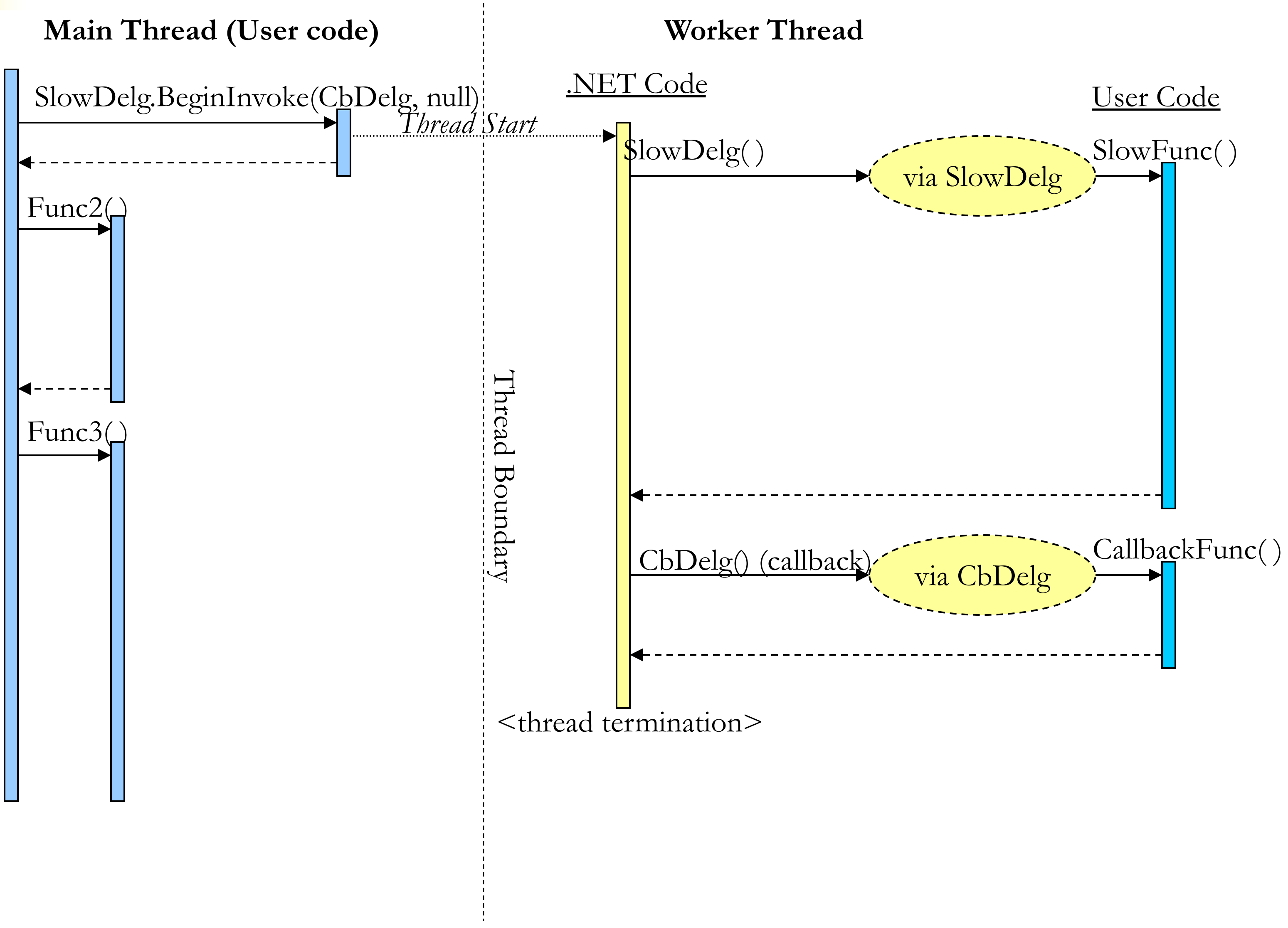
Asynchronity with Completion Callbacks



Asynchronity with Completion Callbacks



Asynchronity with Completion Callbacks



Completion Callback Example

```
public delegate int BinaryOperation(int operand1, int operand2);

public class CalcClient
{
   public static void Main() {
      BinaryOperation addDel;
      AsyncCallback callbackDel;                ← AsyncCallback is defined by .NET for BeginInvoke()
      Calculator calc = new Calculator();       ← Could have set up an RPC for this - has no effect on callbacks

      addDel = calc.Add;
      callbackDel = this.OnAddCompletion;       ← Point callback delegate at callback function
      addDel.BeginInvoke(1, 2, callbackDel, null);         ← Start worker thread (*callbackDel note*)

      System.Console.WriteLine("Waiting for completion.");  ← Race condition with result output below
      System.Console.ReadLine();
   }
   private void OnAddCompletion(IAsyncResult asyncResult) {  ← Must be same signature as AsyncCallback
      int iAddResult;
      BinaryOperation addDel;
      AsyncResult asyncObj = (AsyncResult)asyncResult;      ← So that we can get the AsyncDelegate

      if (asyncObj.EndInvokeCalled == false) {    ← Must not call EndInvoke more than once!
         addDel = (BinaryOperation)asyncObj.AsyncDelegate;  ← Gain access to delegate for EndInvoke
         iAddResult = addDel.EndInvoke(asyncObj);          ← Would retrieve ref or out params here too
         System.Console.WriteLine("1 + 2 = " + iAddResult); ← Race condition with result output in Main()
      }
      asyncObj.AsyncWaitHandle.Close();  ← Clean up
   }
}
```

Blocking/Polling vs Completion Callback

- Blocking advantages
  - ☑ All thread synchronisation performed for you by .NET
  - ☑ Very little extra coding over a normal synchronous call
  - ☑ Simple: no callback function involved
- Blocking disadvantages
  - ☒ Will block calling thread if async call hasn't finished by the time EndInvoke() is called
    - » May cause GUI to freeze if the thread is the event loop thread

Blocking/Polling vs Completion Callback

- Polling advantages
  - ☑ All thread synchronisation performed for you by .NET
  - ☑ Simple: no callback function involved
- Polling disadvantages
  - ☒ Inefficient use of CPU to repeatedly check if call is complete – takes CPU time away from useful processing
  - ☒ Requires coding to periodically check (poll) if complete

Blocking/Polling vs Completion Callback

- Completion Callback advantages
  - ☑ Will be notified as soon as async call is finished
    - » This avoids unnecessary waiting on the thread (blocking) and resource wasting (polling)
- Completion Callback disadvantages
  - ☒ Complex: requires a second function as the callback
  - ☒ Callback occurs on a different thread, so user must perform thread synchronisation themselves

Thread-Safety

- One problem with completion callbacks is that the callback is made on the worker thread
  - Whereas the original call was made on the main thread
  - Need some way to safely pass data between threads
    - » Using shared data is efficient, but data can be corrupted if multiple threads access/update it at the same time
  - This is the problem of thread synchronisation
    - » Coordinating two or more threads so as to avoid corrupting data
- Concept of thread-safety / re-entrancy
  - A thread-safe function guarantees that it will not corrupt data even if it is called simultaneously by multiple threads
    - » Re-entrancy is an older term for a similar concept

Need for Thread-Safety

- If you just avoid using asynchronous calls or threads, why worry about thread-safety?
  - Calls to server objects *always* run on their own thread
    - » Two simultaneous client calls = two server threads
  - Thus if server objects have any shared resources (eg: global variables), thread-safety becomes an issue
    - » eg: Say the server host has a GUI showing how many clients are connected. Server objects updating this form must synchronise or else risk corrupting the connection count.
- Moreover, most GUI clients use worker threads or asynchronous calls to keep the GUI responsive
  - I won't require this, but your employer ***definitely*** will!

Thread Synchronisation in .NET

- Two ways to perform thread synchronisation
- ‘Automatic’ (via .NET-specific attribute keywords)
  - SynchronizationContext (controlled by UseSynchronizationContext)
    - » Essentially forces single-threaded access to a *synchronisation domain* (a set of objects that were all created on same thread)
  - Synchronised methods (your likely mainstay)
    - » Calling a synchronised method of an object locks all other threads from accessing any other *synchronised* method of the obj
- Manual: locks, mutexs, waits/signals (underpins auto methods)
  - We won’t explore these in this unit
  - Fairly easy to use, but hard to keep bug-free!

.NET Thread Synchronisation Example

```
public class CalculatorEx : Calculator
{
   private int m_iNumOperationsCompleted;  ← A bit contrived, but we need some kind of shared data

   public CalculatorEx() {
      m_iNumOperationsCompleted = 0;
   }

   public int GetNumOperationsCompleted() {
      return m_iNumOperationsCompleted;     ← Read only: no need to synchronise this
   }

   [MethodImpl(MethodImplOptions.Synchronized)]     ← Synchronise access to member var(s)
   public int Add(int operand1, int operand2) {
      m_iNumOperationsCompleted++;
      return operand1 + operand2;
   }

   [MethodImpl(MethodImplOptions.Synchronized)]     ← Synchronise access to member var(s)
   public int Subtract(int operand1, int operand2) {
      m_iNumOperationsCompleted++;
      return operand1 - operand2;
   }
}
```

.NET Thread Syncing with Lock

– Locking in the previous example could have been done manually with the lock keyword too (not recommended):

```
public int Add(int operand1, int operand2) {
    lock(m_iNumOperationsCompleted) {   ← Synchronise access to var m_iNumOpsCompleted
        m_iNumOperationsCompleted++;
    }
    return operand1 + operand2;
}
```

– The fact that we only locked on a single variable increases the risk of accidentally creating race conditions or deadlocks
  - Must be careful that you lock the right variable(s) in the right order
  - `lock(this)` is safer, but same as a Synchronized MethodImpl
    - » Locking the whole object means locking access to *any* shared variable (safer)
  - It can be more efficient to use lock(), but at the cost of more bugs
    - » And there is no harder bug to find than a threading bug

Automatic vs Manual Syncing

- Automatic
  - ☑ Very easy - just add a line of code
    - » Easier to see what is thread-safe too
  - ☒ Inefficient for objects that have many methods used by many threads, especially if they use different private data
    - » A call to one method locks *all* methods of that object
- Manual
  - ☑ Fine control - can lock at the variable level (or lower)
    - » ie: can be more efficient (IF you do it properly)
  - ☒ More coding - must lock wherever variable is used
  - ☒ Much more prone to errors

WARNING

- Never underestimate thread concurrency issues
  - **IF YOU DO NOT FEAR THREADS, YOU DO NOT KNOW THREADS**
    - » It's like plutonium: *never ever* be less than cautious!
    - » Debugging race conditions is the stuff of nightmares
- Race conditions:
  - when two threads are in a race to update shared resources
- Types of errors from race conditions
  - Lost update – one thread overwrites the change another thread made
  - Out-of-date data
    - » eg: Memory access errors: one thread deletes data another uses
  - Inconsistent state
    - » Two threads interleave their updates on related member vars

How to Approach Thread Safety

- Avoid mixing auto and manual syncing in one class
  - Auto-syncing won't be aware of your manual syncing
- Components should be (externally) thread-safe
  - Data integrity: Can't assume clients will ensure thread-safe access
- Minimise number of potentially-shared variables
  - Use local variables, or use arrays with one entry for each thread
- Better safe than sorry: err on the side of inefficiency
  - eg: lock entire object rather than one variable
- BUT: Don't just synchronise everything
  - Frequent thread [b]locking = slower than single-threaded
  - Synchronise at the business logic level, not the low level classes like Lists (Java learned this the hard way with Vector, hence ArrayList)
  - No need to lock read-only accesses to shared variables (usually)

Oneway Calls

- What if we just need to quickly fire off a message to the remote end but don't need anything back?
  - Blocking would be a waste of time: there's no return val
  - An async RPC call wouldn't block, but it adds MT issues
- One solution is to use oneway calls
  - Introduced back in CORBA (and probably earlier)
  - Still a synchronous (non-threaded) call, but the client only blocks long enough to ensure the call started on the server
  - As soon as the server ACKs the initial call, the client continues even though the server has not completed

Oneway Calls

- Oneway calls must be void and with no out/ref parameters
  - ie: no returned values
- In .NET, a oneway call is defined by marking it with a parameter on the OperationContract attribute:
  - [OperationContract(IsOneWay=true)]
- Their behaviour is such that they are a much simpler substitute for full-blown asynchronous calls

Oneway Calls vs Async Calls

- However, oneway calls aren't *quite* immediate return
  - Will block until server ACKs the call
  - This could be a while if the server is down and times out
  - Still, it would be rare to have a oneway call last a noticeable amount of time
- Hence network exceptions can occur on the call
  - Network errors, timeout errors
- In comparison, async calls return *before* the RPC call is made
  - A thread is spun off to deal with the call

Remote Callbacks

- The concept of callbacks is broader than just their use as completion callbacks in asynchronous calls
  - Callback refers to the idea of calling back (or on behalf of) the caller via the function they provided to you
- Callback are frequently used in distributed computing
- Examples:
  - Progress updates from server to client
  - Publisher/subscriber systems
  - Peer-to-peer applications, where there is no server/client
    - » or more accurately, both sides act as both client and server

Remote Callbacks

- Typical case is that server needs to notify the client
  - eg: when the server is processing a long job and must periodically update the client on its progress
- In most RPC frameworks (but not WCF) the approach is to have the *client* implement a server object
  - Client passes this object to the server when starting the job
    - » The object is passed by reference (as a kind of 'network pointer')
  - Server does callback by invoking on the passed object ref
    - » Object knows where it came from; no need to set up connection
    - » Server must have been compiled with the client's interface

Remote Callbacks

- However, WCF tries to be *service*-oriented, not O-O
  - Hence passing object references around is not allowed
  - Instead, server objects have an optional *callback channel* that can be defined and associated with an interface
    - » Set up on connection initialisation
    - » This callback channel is requested by the *client*
      - – Called a duplex channel; duplex = two-way communication
  - On the client, we then have a class that implements the callback interface
    - » This class is effectively acting as a server object for the true server to call back on

Remote Callbacks – Duplex Channels

- Duplex channels are an explicit callback mechanism
  - As opposed to other frameworks that allow you to pass object references across the network, where callbacks are just invocations on a pointer to the client's 'server' object
- There are a few limitations of this approach:
  - Only one callback interface per server interface
    - » Although can have an inheritance hierarchy for your callbacks, so that the client can choose the base interface that fits it needs
  - .NET only provides access to the client's callback impl object during the server's RPC handling of a client
    - » You must save or pass this reference around to use it later
    - » (since it is a reference to a proxy, it is just a normal pointer)

Remote Callbacks – Server Side

```
// IServer.cs
[ServiceContract]
public interface IServerCallback {                ← Callback interface
   [OperationContract(IsOneWay=true)]              ← Callbacks are often a good candidate for oneway methods
   void ProgressUpdate(int iteration);
}
[ServiceContract(CallbackContract=typeof(IServerCallback)]   ← Define the associated callback
public interface IServer {                                        ← Server interface
   [OperationContract]
   void LongRunningJob();
}
-------------------------------------------------------------------------------------------
// ServerImpl.cs
[ServiceBehavior(ConcurrencyMode=ConcurrencyMode.Multiple,
                 UseSynchronizationContext=false)]
internal class ServerImpl : IServer {
   public void LongRunningJob() {
      // Get a reference to the client-side callback object
      IServerCallback cb = OperationContext.Current.GetCallbackChannel<IServerCallback>();
                        ← The cb ref can only be obtained in LongRunningJob (but can be stored)
      // Do server processing of the long-running job
      int ii = 0;
      while (DoProcessing() == true) {
         ii++;
         cb.ProgressUpdate(ii);                    ← Callback: let the client know of the progress
      }
   }
}
```

Remote Callbacks – Server Side

```
--------------------------------------------------------------------------------
// ServerHost.cs is the same as usual - see Week 2 lecture notes
// The use of a duplex channel is defined on the client side
--------------------------------------------------------------------------------
```

Remote Callbacks – Client Side

```
// ClientObj.cs
using IServer;                                                  ← For definition of IServerCallback
[CallbackBehavior(ConcurrencyMode=ConcurrencyMode.Multiple,      ← Same as ServiceBehaviour
                  UseSynchronizationContext=false)]
internal class ClientObj : IServerCallback {                    ← Implementation class for callback
   public void ConnectToServer() {
      DuplexChannelFactory<IServer> serverFactory;               ← Note Duplex, to allow for callbacks
      IServer theServer;
      NetTcpBinding tcpBinding = new NetTcpBinding()
      string sURL = "net.tcp://localhost:8005/Server";
      serverFactory = new DuplexChannelFactory<IServer>(this, tcpBinding, sURL);
                                                                 ← Pass ourselves to be callback handler
      theServer = serverFactory.CreateChannel();
      theServer.LongRunningJob()                                 ← Will do callbacks to ProgressUpdate
   }
   public void ProgressUpdate() {
      MessageBox.Show("Progress is occurring.");
   }
}
------------------------------------------------------------------------------------------
// ClientMain.cs
public class TheClient {
   public static void Main() {
      ClientObj client = new ClientObj();
      client.ConnectToServer
   }
}
```

Remote Callbacks - Notes

– Things to note:
  - We made the callback a oneway call
    - » This means that the server will not be blocked by the client
    - » Important since the server is probably doing processing, so waiting for the client to tell the user is a waste of server time
    - » And *much* simpler than writing an async call for the callback!
  - We marked the impl class with `[CallbackBehavior(...)]`
    - » Force .NET to be multi-threaded and that <u>we</u> handle synch-ing
    - » If the client is a GUI, then in truth a single-threaded, .NET-synchronised callback is actually a good idea
      - – GUIs run on an event loop, so the callback will just add itself to the event queue and act like any other fired event – convenient!
    - » But since we are learning, we'll stick with the harder MT option

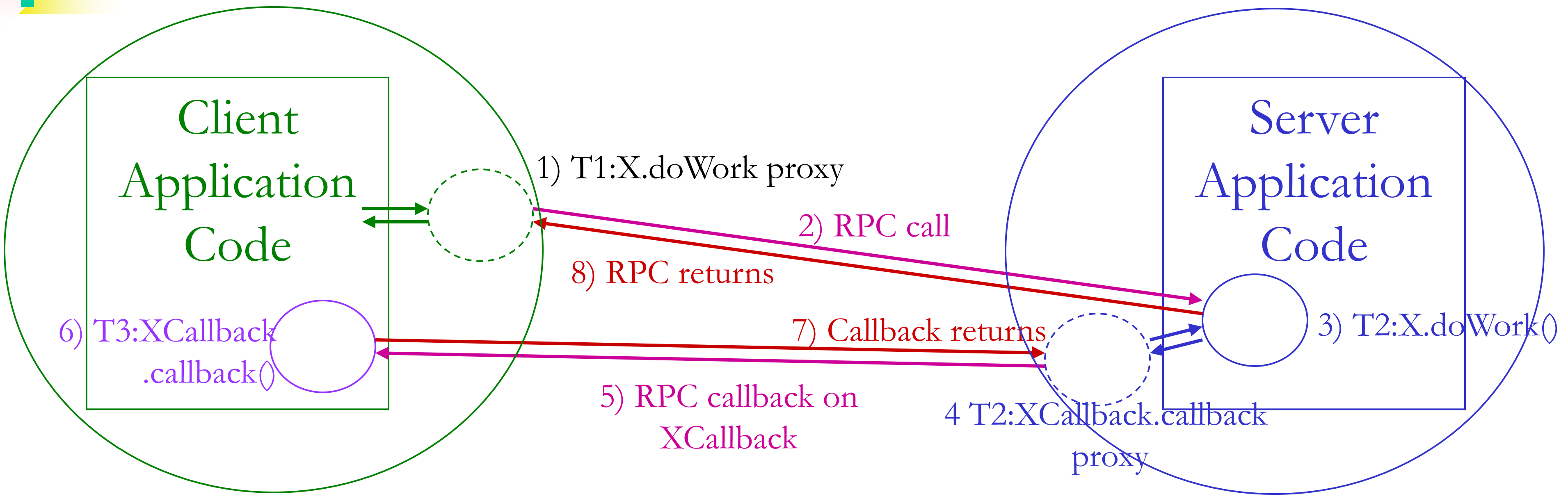
Remote Callbacks - Duplex

- The duplex channel setup is nearly identical to the normal channel setup
  - Just change ChannelFactory to DuplexChannelFactory, AND pass in a reference to the callback handler object
- Note that the callback class can be *any* class that implements IServerCallback
  - It didn't have to be the ClientObj, that was just convenient
    - » It reduced the number of classes and keeps the callback code in the same class as where the original call was made
    - » In the pracs, we will make the MainWindow (a GUI class!) to be our handler

Remote Callbacks

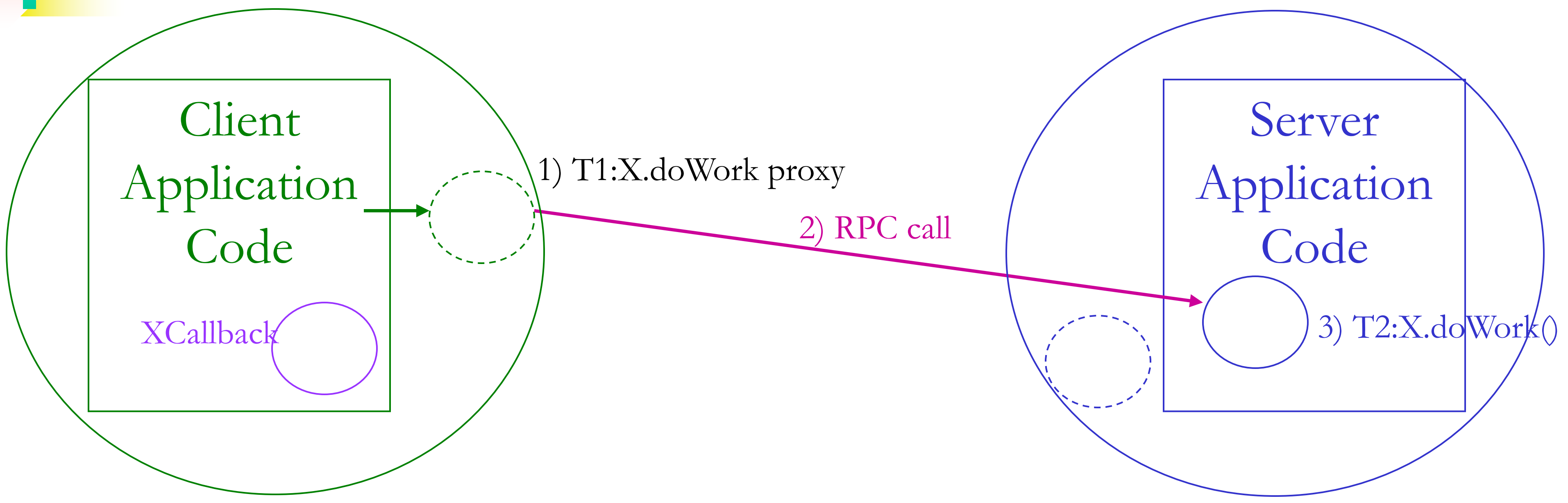
- Moreover, the callback ref can be stored by the server
      - » eg: in a member field
  - Would allow the server to call the client *at any time*, not just when handling the client's call
      - » In other words, the client becomes like a server
      - » Good for peer-to-peer applications where both sides can initiate new calls to the remote end depending on their respective users
- In any case, thread synchronisation is still needed
  - An RPC callback on a client object will be run in a new thread, just like an RPC call to a server object
      - » It can't just interrupt the client's main thread!
  - Most callbacks will need to update shared resources
      - » eg: update a progress bar
      - » Hence thread synchronisation becomes an issue

Example Remote Callback

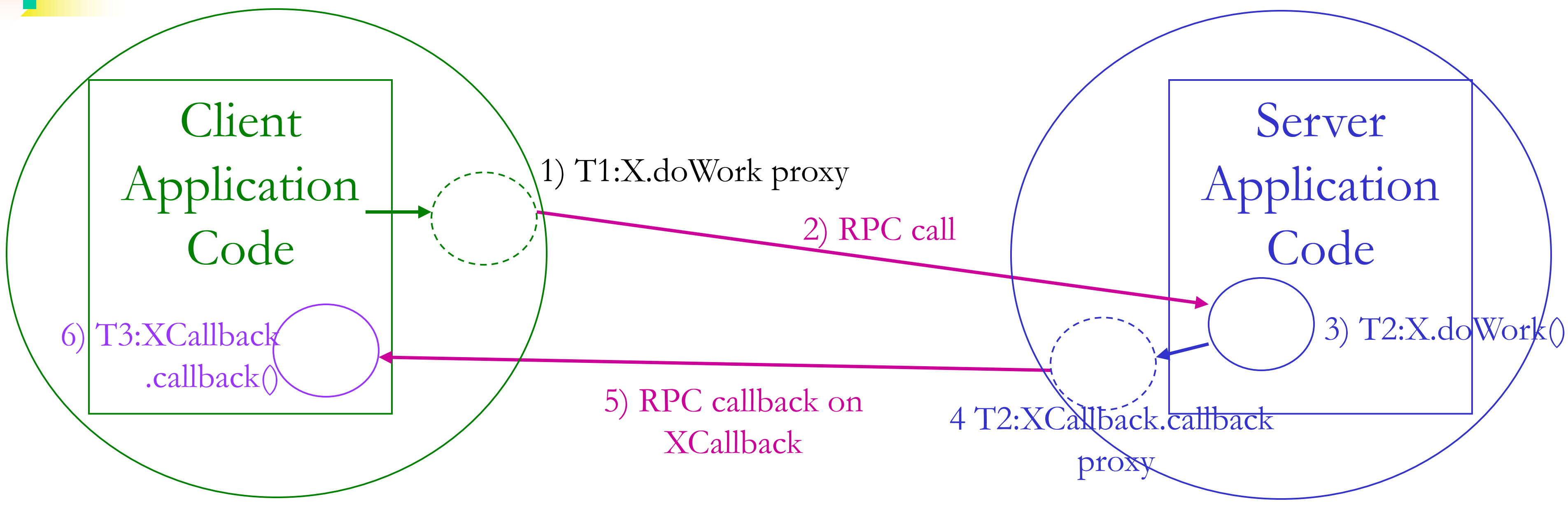


– We'll do it in animation over the next few slides

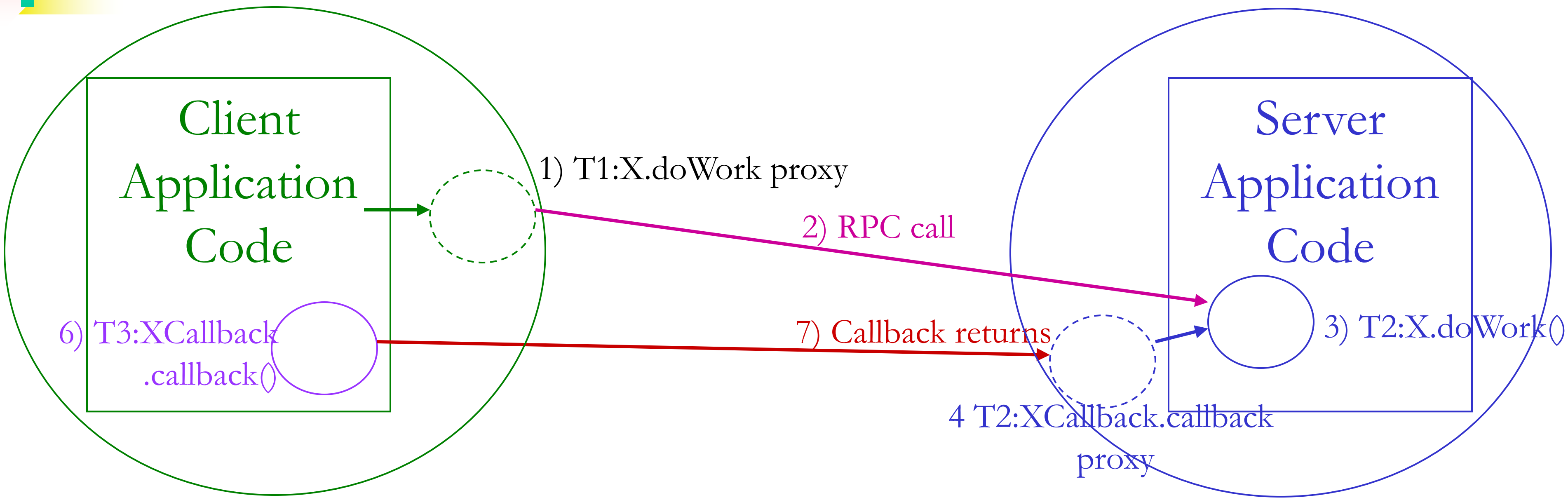
Example Remote Callback



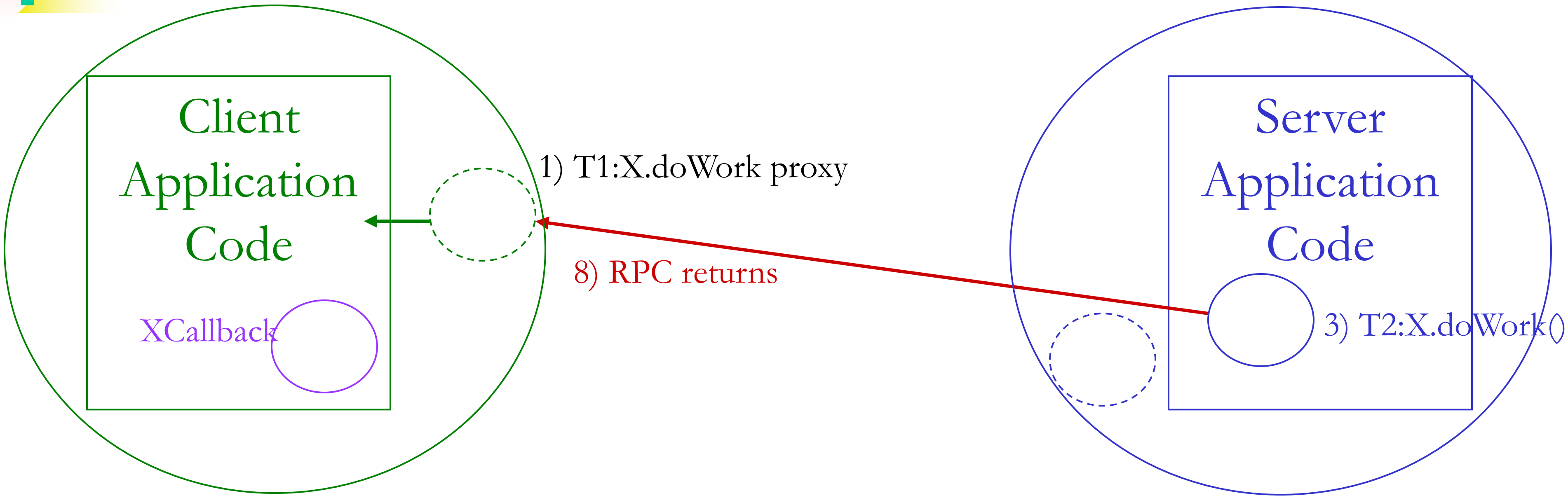
Example Remote Callback



Example Remote Callback



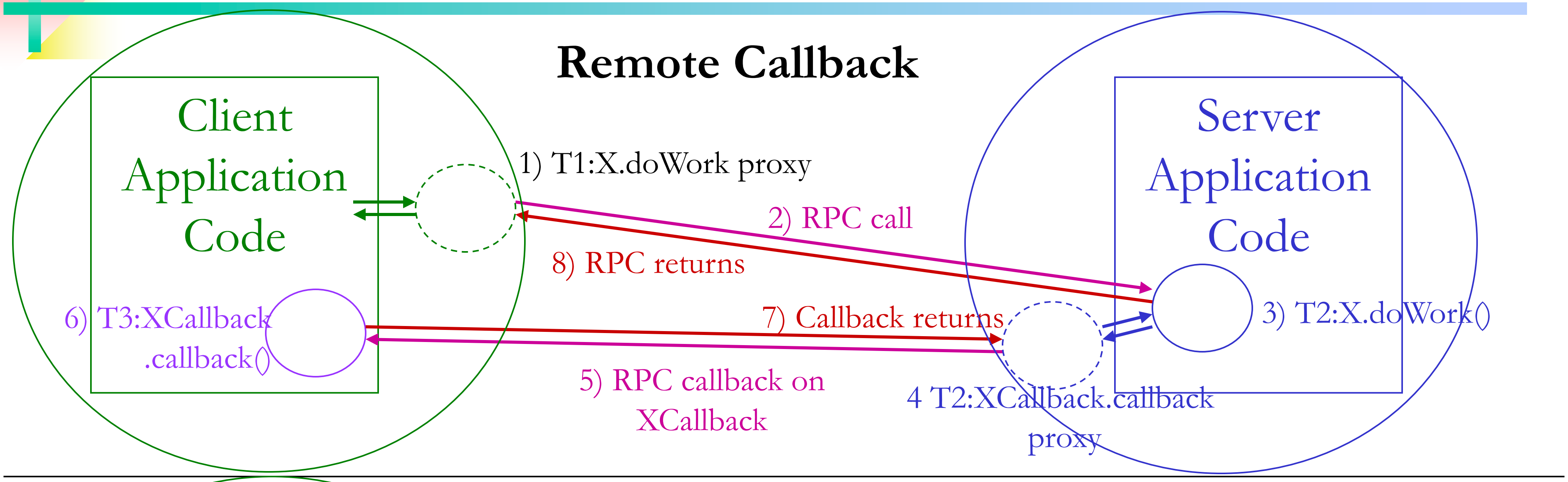
Example Remote Callback

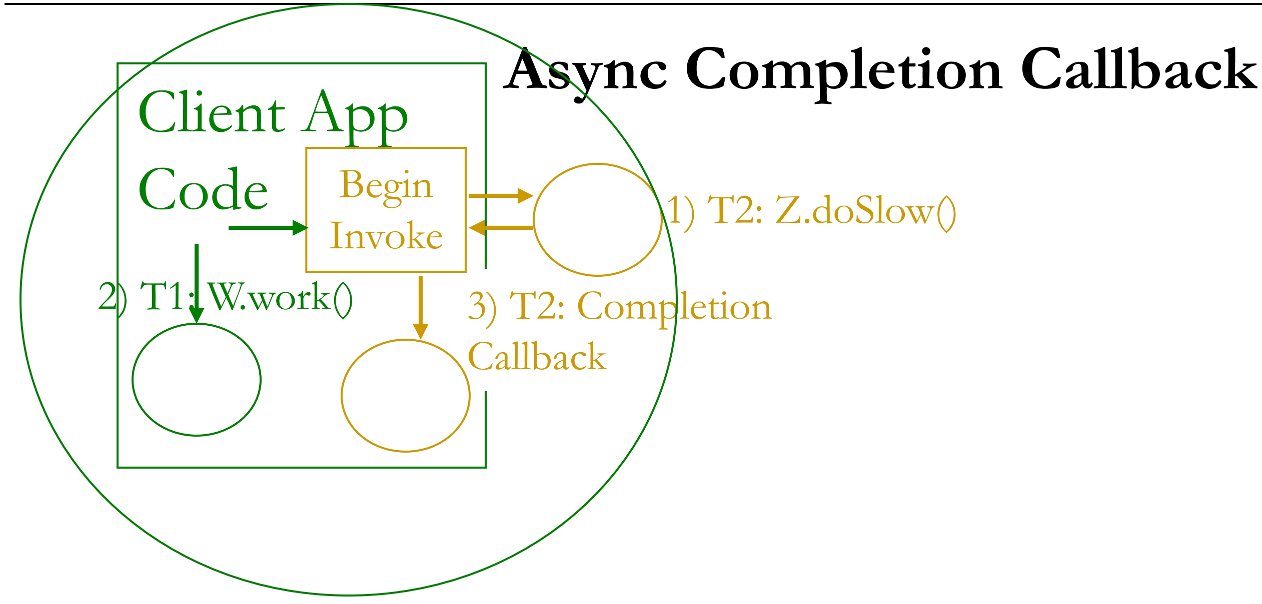


Remote Callback Example

- When X.doWork( ) is called on Duplex channel, the server is aware of the XCallback object
  - On the server, .NET silently creates a proxy to the XCallback object client object so that it can get RPC calls
- The server understands XCallback because it is an extended part of the interface definition of X
- Remote callbacks have nothing to do with async completion callbacks, even if it is a *remote* async call
  - The former is the server calling back to the client
  - The latter is .NET telling you an async call has *finished*

Remote Callback vs Completion Callback

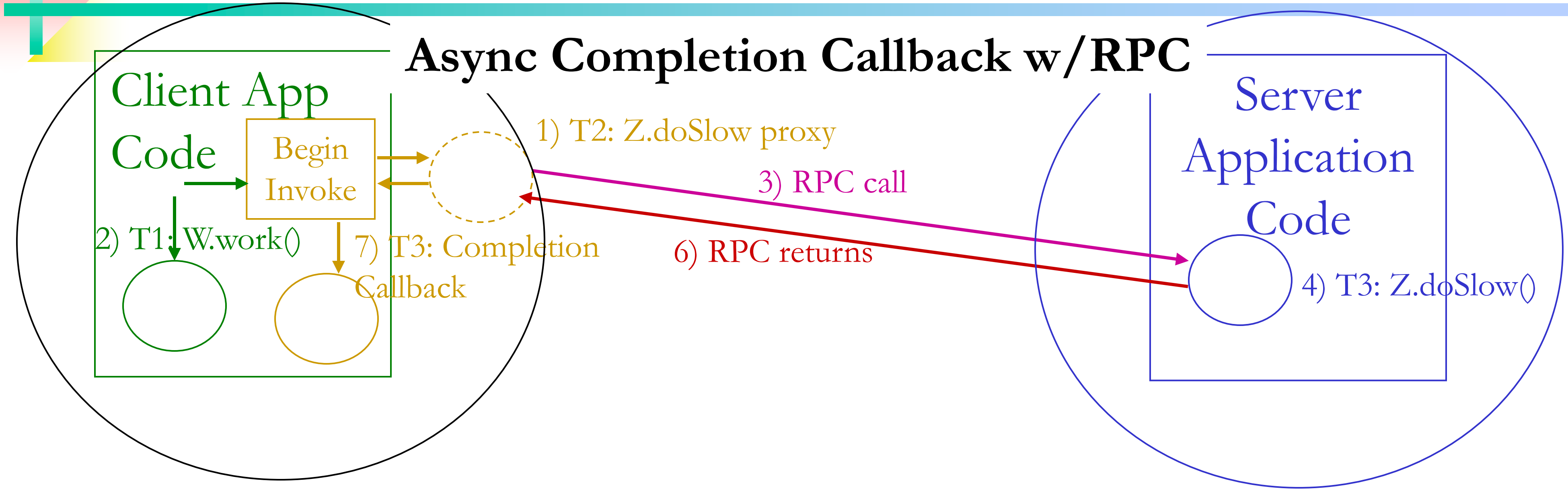


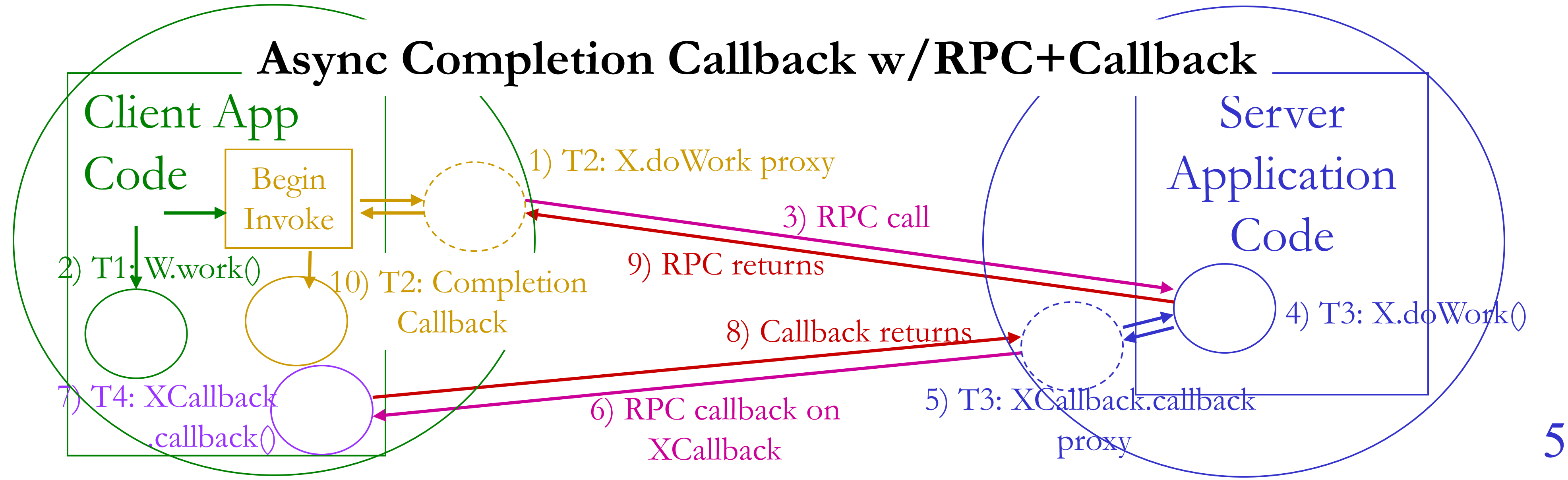


Remote Callback vs Completion Callback

- Note the two types *have nothing to do with each other*
  - The remote callback is about having the server callback to the client during its execution of X.doWork()
  - The completion callback is all about informing whoever called Z.doSlow() that the latter finished. Whether Z is remote or not is irrelevant
- Since the two are independent concepts, they can be used in combination
  - eg 1: Perform the async call to Z.doSlow as an RPC
  - eg 1: Perform the call to X.doWork asynchronously and have it do a callback
  - eg 3: Perform the *callback* asynchronously

Remote Callback vs Completion Callback





Cross-Thread Method Invocation

- Sometimes it's necessary for one thread to ask another thread to call a method
  - In particular, for methods of GUI controls - all updates to GUI controls must be done on the GUI event thread
    - » Ensures that the GUI will never be corrupted
  - .NET WPF uses delegates and the Dispatcher class for this
    - » In particular, by calling Application.Current.Dispatcher.Invoke() passing it a delegate to execute
  - Although it looks like you are invoking a function, you are in fact putting a message onto the GUI event queue
    - » This message will later be dispatched by the queue, thus 'calling' the method

GUI Event Loop

- Every GUI program executes an event loop
  - Most new languages (eg: Java) try to hide it from the coder
- The event loop iterates through a queue of events that need processing (called 'dispatching')
  - Events come from the operating system (eg: mouse click), or can be generated by the programmer
- An event is processed by calling the function(s) that have asked to be notified about a given event
  - eg: your own OnClick code for a Button

GUI Event Loop

- When an event is dispatched, the event handling method is given control *on the GUI event loop thread*
  - No other events can be processed until the method returns
    - » This includes painting the GUI or animating a button click
  - Thus a slow event will make the GUI appear to freeze
    - » Events still get buffered (queued), but not processed
- All updates of any GUI element *must* be processed via the event queue *and* on the event loop thread
  - Otherwise it is far too hard to guarantee thread-safety for the critical job of GUI event dispatching
  - Dispatcher is one way; there are a couple of others…

Dispatcher Invoke Example

```
partial public class wdwCalculatorGUI : Window
{
   // The following assumes there is a status bar called stbStatus on the Window...

   protected delegate void SetStatusDelg(string text);

   void SetStatus(string text) {
      ((StatusBarItem)stbStatus.Items[0]).Content = text;
   }

   [MethodImpl(MethodImplOptions.Synchronized)]      ← Let's say this function will be called from multiple threads
   public void SetMessage(string msg) {
      SetStatusDelg setSts = SetStatus;    ← Delegate for Dispatcher Invoke()

      object[] prms;                                 ← Set up params for SetStatusDelg()
      textParams = new object[1];
      textParams[0] = msg;

      Application.Current.Dispatcher.Invoke(setSts, prms);   ← Invoke SetStatus on GUI event thread
                                                               Blocks until invocation completes
                                                               (use BeginInvoke for async invocation)

   }
}
```

Anonymous Methods and Delegates

- The previous example can be simplified by the use of anonymous methods
  - The problem was that we had to define a separate SetStatus method and create an associated delegate type
  - Anonymous methods are methods which have no name and can be declared at any point *within* another method
    - » This allows you to put the update code close to where it is called
  - Can make the code *much* more compact and readable
- Anonymous methods also facilitate interesting tricks with using variables from outside the anon. method
  - From the concept of closures

Dispatcher w/Anon. Method

```
partial public class wdwCalculatorGUI : Window
{
   // The following assumes there is a status bar called stbStatus on the Window...

   [MethodImpl(MethodImplOptions.Synchronized)]
   public void SetMessage(string msg) {
      Action<string> a = delegate() {
         ((StatusBarItem)stbStatus.Items[0]).Content = msg;
      };

      Application.Current.Dispatcher.Invoke(a);
   }
}
```

← Let’s say this function will be called from multiple threads

← Define an anonymous function that takes no params and has no return val

← Can use outer variable! (‘closures’)

← No params needed: uses closure on msg variable

.NET Built-in Delegate Types

- Here, Action is a .NET built-in delegate type to point at methods with no parameters and no return value
  - ie .NET defines: `delegate void Action();`
- Other standard delegate types include:
  - Action<T>, Action<T1,T2>, Action<T1,T2,T3>
    - » Define void functions that receive one, two or three params whose types are defined by the generics
  - Func<TRes>, Func<T, TRes>, Func<T1,T2,TRes> …
    - » Similar to Action, but with a return value whose type is <TRes>
  - These are only provided for your convenience: you could easily have defined your own, but why not use these?

Closures

- Note that `msg` is made available to the anon method, via the concept of a 'closure' on that variable:
  - A closure is a compiler trick that 'captures' (keeps) variables for use in another method, even though those variables are not local or passed to that method
  - A complex topic with many subtleties in implementation

Lambda Expressions

- A related topic to anonymous methods and closures is the concept of Lambda Expressions
  - From functional programming, where everything is a function that transforms the results of other functions
  - Versus the more familiar imperative programming where data is state to be manipulated by statements
- A Lambda expression could be viewed as a kind of anonymous method
  - C# uses the => operator to define a lambda
    - » Define the parameter vars, then the =>, then the function
  - Data types are all inferred by the compiler (more tricks!)

Dispatcher w/Lambda Expression

```
partial public class wdwCalculatorGUI : Window
{
   // The following assumes there is a status bar called stbStatus on the Window...

   [MethodImpl(MethodImplOptions.Synchronized)]
   public void SetMessage(string msg) {
      Action<string> a = () => {
         ((StatusBarItem)stbStatus.Items[0]).Content = msg;
      };

      Application.Current.Dispatcher.Invoke(a);
   }
}
```

← Let's say this function will be called from multiple threads

← Define a lambda expression that has no parameters (the () =>)

← Can use outer variable! ('closures')

← No params needed: uses closure on msg variable

Lambda Expressions

- A different example where a parameter is passed:
  - `x => { 2.0 * x }`
  - From this, the compiler infers that x is a double and the function returns a double
- Lambdas are probably easiest to understand as a shorthand for defining an anonymous method
  - You can also define a lambda with multiple parameters, and even out or ref parameters
  - However, anything you can do with lambdas can also be done with the **`delegate()`** syntax explored earlier
    - » Or the long delegates-and-separate-methods for that matter!

Updating GUI – Deligate Example

```
private Search search; //reference to method

public MainWindow()
{…
}

private void GoButton_Click(object sender, RoutedEventArgs e)
{…
}

private void SearchButthon_Click(object sender, RoutedEventArgs e)
{
    search = SearchDB;
    AsyncCallback callback;
    callback = this.OnSearchCompletion;
    IAsyncResult result = search.BeginInvoke(SearchBox.Text, callback, null);

}
```

Updating GUI – Deligate Example

```
private Student SearchDB(string value)
{
    string name = null;
    int id = 0;
    string universityName = null;
    foob.GetValuesForSearch(value, out name, out id, out universityName);
    if (id != 0)
    {
        Student aStudent = new Student();
        aStudent.Name= name;
        aStudent.Id= id;
        aStudent.University = universityName;
        return aStudent;
    }
    return null;
}

private void OnSearchCompletion(IAsyncResult asyncResult)
{
    Student iStudent=null;
    Search search=null;
    AsyncResult asyncobj = (AsyncResult)asyncResult;
    if (asyncobj.EndInvokeCalled == false)
    {
        search = (Search)asyncobj.AsyncDelegate;
        iStudent = search.EndInvoke(asyncobj);
        UpdateGui(iStudent);
    }

    asyncobj.AsyncWaitHandle.Close();
```

Updating GUI – Delegate Example

```
private void UpdateGui(Student aStudent)
{
    SiD.Dispatcher.Invoke(new Action(() =>SiD.Text = aStudent.Id.ToString()));
    SName.Dispatcher.Invoke(new Action(() => SName.Text = aStudent.Name));
    SUni.Dispatcher.Invoke(new Action(() => SUni.Text = aStudent.University));

}
```

Updating GUI – Async Example

```
private async void SearchButthon_Click(object sender, RoutedEventArgs e)
{
    searchvalue = SearchBox.Text;
    Task<Student> task = new Task<Student>(SearchDB);
    task.Start();
    statusLabel.Content = "Searching starts.....";
    Student student = await task;
    UpdateGui(student);
    statusLabel.Content = "Searching ends.....";
}
```

```
private void UpdateGui(Student aStudent)
{
    SiD.Text = aStudent.Id.ToString();
    SName.Text = aStudent.Name;
    SUni.Text = aStudent.University;
}
```