Asynchronous Programming

Andrew Clymer
andy@rocksolidknowledge.com

Richard Blewett
richard@rocksolidknowledge.com

Course prerequisites

- Programming experience required
  - 6 months of C#

The plan

- Tasks
- Async/Await
- Thread Safety
- Simplifying Thread Safety

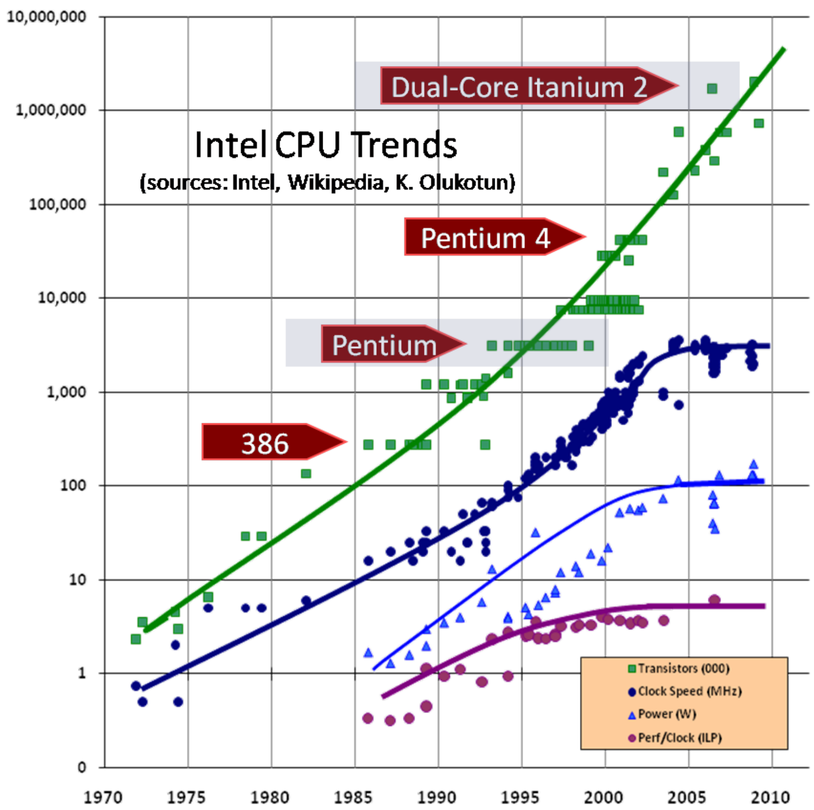
Agenda

- Why async programming
- The Task abstraction
- Creating Tasks
- Passing data into tasks and retrieving results
- Cancellation and Progress

The Benefits of Asynchronous Programming

- Single threaded applications simplest to write
  - Not practical for server side
  - Can cause responsiveness problems on client
- Running code asynchronously has many potential benefits
  - Increased throughput
  - Increased responsiveness
- Async required to take advantage of modern CPU architectures
  - Clock speeds not increasing significantly
  - Multi-core architecture now the norm

The Free Lunch is Over

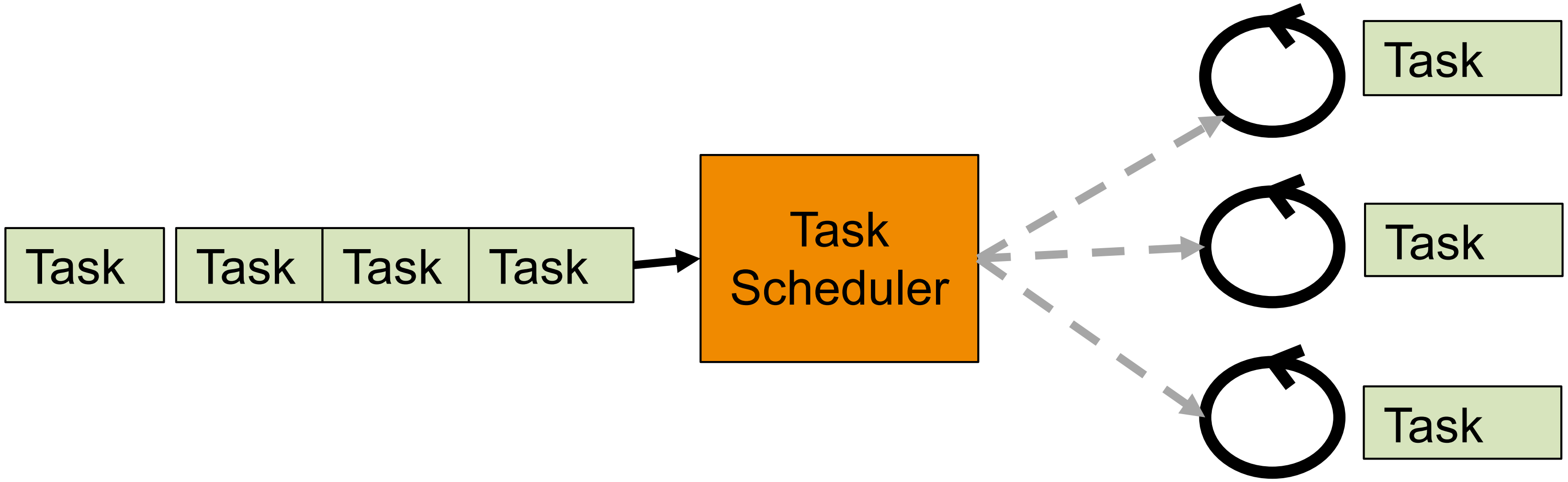


Creating Asynchronous Units of Work

- Developers indentify functionality that can run asynchronously
  - Invoking “long running” blocking operations such as network access
  - Waiting for a condition to arise such as a file to be created in a particular folder
  - Processing a request from another machine
- Work must be totally or mostly independent from main processing
  - Otherwise coordinating work removes the benefit of async
- Developer packages unit of work as a Task

What is a Task?

- A Task is a schedulable Unit of Work
  - Wraps a delegate that is the actual work
- Task is enqueued to a TaskScheduler



Creating a Task

- **Tasks are created by passing a delegate to the constructor**
  - Call `Start` to queue the task to the scheduler
  - Can also use Factory property

```
Action a = () =>Console.WriteLine("Hello from task");
Task tsk = new Task(a);
tsk.Start();
```

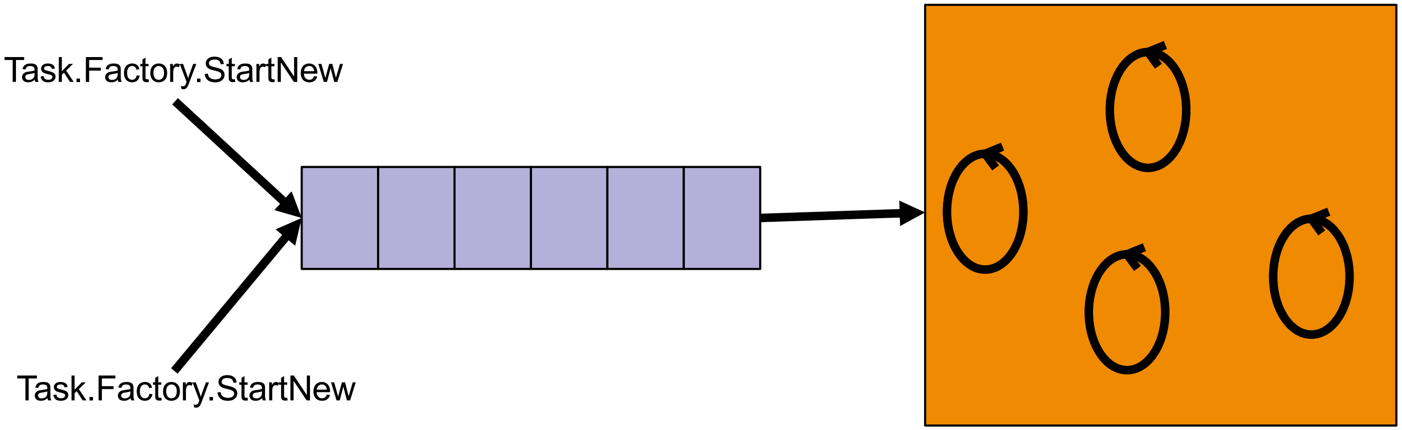
```
Action a = () => Console.WriteLine("Hello from task");
Task tsk = Task.Factory.StartNew(a);
```

Scheduling the Work

- Scheduler maps work on to threads
  - Scheduler is an abstraction that has multiple implementations
- Two main models for scheduling
  - Thread pooling
  - Dedicated threads
- Thread pooling reuses existing threads
  - Ideal for most situations
- Dedicated threads give the task its own thread
  - Is good practice for long running tasks

The System Thread Pool

- .NET has a per-process thread pool
  - Assigned work queued up
  - Threads in pool pick up tasks when idle
- Thread pool manager controls number of threads in pool
  - Number of threads depends on workload
  - Number of threads in pool is capped



A Single API

- Task provides a single API for async work
  - Long running task created by hinting to scheduler
  - Current implementation long running tasks spawn own thread otherwise Thread Pool used

```
Task t1 = new Task( DoWork,
TaskCreationOptions.LongRunning );
```

Passing Data to a Task

- Data passed explicitly using `Action<object>`
- Data can be passed implicitly using anonymous delegate rules

```
Guid jobId = Guid.NewGuid();

Action<object> a = state =>
{
  Console.WriteLine("{0}: Hello {1}", jobId, state);
};

Task tsk = new Task(a, "World");
tsk.Start();
```

Returning Data from a Task

- Generic version of Task available
  - `T` is return type
  - Accessed from the task `Result`
- Takes a Func delegate as a constructor parameter
  - `Func<T>`
  - `Func<object, T>`

```
Func<object, int> f = state =>
{
  Console.WriteLine("Hello {0}", state);
  return 42;
};
Task<int> tsk = new Task<int>(f, "World");
tsk.Start();
//...
Console.WriteLine("Task return is: {0}", tsk.Result);
```

Waiting for a Task to End

- Can wait for one or more tasks to end using `Wait`, `WaitAll` or `WaitAny`
- Can pass timeout for wait

```
Task t = new Task( DoWork );
t.Start();
t.Wait();
```

```
Task t1 = new Task( DoWork );
t1.Start();
Task t2 = new Task( DoOtherWork );
t2.Start();

if (!Task.WaitAll(new Task[]{t1, t2}, 2000))
{
  Console.WriteLine( "wait timed out" );
}
```

Task Completion States

- Tasks can end in one of three states
  - RanToCompletion: everything completed normally
  - Canceled: task was cancelled
  - Faulted: an unhandled exception occurred on the task
- Unhandled Exceptions get thrown when waiting on a task
  - thrown in the Task's finalizer if task not waited upon

```
Task t = new Task(DoWork);
t.Start();

if (!t.Wait(1000))
{
}
```

```
private static void DoWork()
{
   throw new Exception();
}
```

Cancellation

- Tasks support cancellation
  - Modelled by `CancellationToken`
- Token can be passed into many APIs
  - Task creation
  - Waiting

```
var source = new CancellationTokenSource();

Task t1 = new Task( DoWork, source.Token );
t1.Start();

t1.Wait(source.Token);
```

Triggering Cancellation

- `CancellationTokenSource` has `Cancel` method to trigger the cancellation of tasks and blocking APIs

```
var source = new CancellationTokenSource();

Task t1 = new Task( DoWork, source.Token );
t1.Start();

source.Cancel();
```

The Effects of Cancellation

- Cancellation has different effects depending on state of task
  - Unscheduled tasks are never run
  - Scheduled tasks must cooperate to end. Requires access to CancellationToken

```
private static void DoWork(object o)
{
  var tok = (CancellationToken)o;

  while (true)
  {
    Console.WriteLine("Working ...");
    Thread.Sleep(1000);
    tok.ThrowIfCancellationRequested();
  }
}
```

Cancellation of Blocking Operations

- Blocking operations throw exceptions when cancelled
  - OperationCancelledException
  - AggregateException (when more than one exception could have occurred)

```
try
{
  t1.Wait(1000, source.Token);
}
catch (AggregateException x)
{
  foreach (var item in x.Flatten().InnerExceptions)
  {
    Console.WriteLine(item.Message);
  }
}
```

Timeout via cancellation

- Can signal cancellation after a given period of time

```
var source =  new CancellationTokenSource();

source.CancelAfter(TimeSpan.FromSeconds(2));


Task t1 = new Task( DoWork, source.Token );
t1.Start();
```

Linking Cancellation

- Task cancelled when either
  - ctsToken is signaled
  - timeoutCancel source signals

```
static Task DoItAsync( in CancellationToken ctsToken){

  var timeoutCancel = CancellationTokenSource
                        .CreateLinkedTokenSource(ctsToken);

  timeoutCancel.CancelAfter(TimeSpan.FromSeconds(3));

  var result = ExecuteCallAsync(timeoutCancel.Token);

  return result;
}
```

Task Completion Source

- Tasks can be used to represent any asynchronous activity
- TaskCompletionSource<T> used to manage own Task lifetime
- Used to adapt non Task based asynchronous operations to Tasks

Wrapping an event

- **Single raised event wrapped as a task**

```
public static Task<FileSystemEventArgs> CompleteOnChange(
                                            this FileSystemWatcher watcher )
{
    var tcs = new TaskCompletionSource<FileSystemEventArgs>();

    void OnWatcherOnChanged(object sender, FileSystemEventArgs args)
    {
        watcher.Changed -= OnWatcherOnChanged;
        tcs.SetResult(args);
    }
    watcher.Changed += OnWatcherOnChanged;
    return tcs.Task;
}
```

Async IO

- Some .NET APIs model async IO
  - No thread is consumed while IO bound operation takes place
  - Uses IO Completion ports
- Async IO modelled with Async method
  - Long running operation has additional method that returns a Task<T>

```
WebResponse GetResponse()
```

```
Task<WebResponse> GetResponseAsync()
```

Scheduling

- TaskScheduler is an extensible abstract class
- Out of the box implementations
  - ThreadPoolTaskScheduler (Default)
  - SynchronizationContextTaskScheduler
- Can pass scheduler when starting a task

```
TaskScheduler scheduler = GetScheduler();

Task t = new Task(DoWork);

t.Start(scheduler);
```

Integrating with SynchronizationContext

- GUI applications need UI updates marshalled to the UI thread
  - SynchronizationContext is abstraction that wraps specific technology solution to thread marshalling
- Can get scheduler associated with SynchronizationContext

On UI Thread

```
TaskScheduler scheduler =
        TaskScheduler.FromCurrentSynchronizationContext();
```

On Background Thread

```
private void DoAsyncWork()
{
  Task t = new Task(() => label1.Text = "TADA!!");
  t.Start(scheduler);
}
```

Unnecessary task allocation

- **Synchronous result** still result in a GC allocation

```
public class StockDataSource {
   public Task<decimal> GetPrice(string symbol) {
      if (localPrices.TryGetValue(symbol, out decimal price)){
                 return Task.FromResult(price);
       } else {
       return GetRemotePrice(symbol);
    }
}
```

ValueTask<T>

- Value Type
- No allocation for completed tasks
  - Reduce load on GC
- Designed for high throughput scenarios were operations often completely synchronously
- Useful for abstractions that may have synchronous or asynchronous implementations

Removing unnecessary task allocation

- **Synchronous result** will not result in a GC allocation
- **Asynchronous result** wrapped by a ValueTask

```
public class StockDataSource {
   public ValueTask<decimal> GetPrice(string symbol) {
      if (localPrices.TryGetValue(symbol, out decimal price)){
                  return new ValueTask<decimal>(price);
       } else {
       return new ValueTask<decimal>(GetRemotePrice(symbol));
    }

   private Task<decimal> GetRemotePrice(string symbol) { … }
}
```

Should all asynchronous APIs be ValueTask<T>?

- Task<T> results can be safely used multiple times and concurrently
- ValueTask<T>, can **not** be safely used multiple times and concurrently
- ValueTask<T> not as feature rich as Task<T>
  - No Task.WhenAny, Task.WhenAll, Task.WaitAll …
  - Can convert to task using .AsTask()
- Simple answer is no

Summary

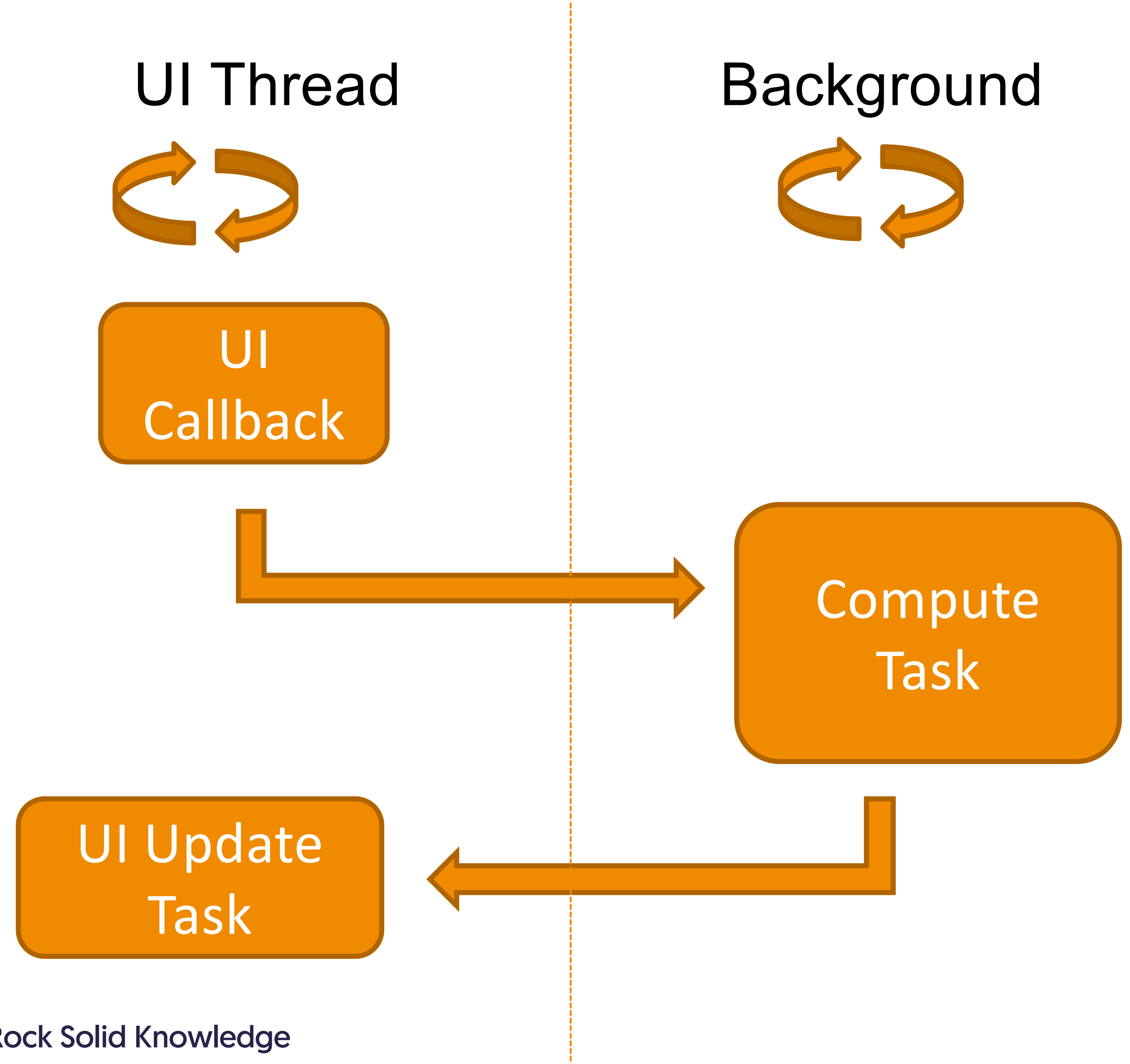
- Async programming is an important skill
- Tasks model unit of async work
  - Compute
  - IO
- Tasks support cancellation
- TaskCompletionSource useful tool for making anything a task
- Consider using ValueTask for abstractions where the implementation may often by synchronous

async and await

Objectives

- Why use Continuations
- Simple examples of async/await
- Under the hood
- Gotcha's
- Composition
- Server side async/await
- Asynchronous streams

Continuations



Control flow

- Sequential programming intent is pretty clear.
- Asynchronous programming screws with intent
- Do X Async, Then Do Y , Then Do Z Async
- How to handle errors
  - Where to place the try/catch

async/await keywords

Making async intent as clear as synchronous intent

- Two new keywords introduced in C# 5
- Enables continuations whilst maintaining the readability of sequential code
  - Automatic marshalling back on to the UI thread
- Built around Task, and Task<T>, ValueTask<T>

async and await

Example

- async method must return void or a Task/ValueTask
- async method should include an await
- await <TASK>

```
async void Button_Click(object sender, RoutedEventArgs e) {
  calcButton.IsEnabled = false;
  Task<double> piResult = CalcPiAsync(1000000000);

   // If piResult not ready returns, allowing UI to continue
   // When has completed, returns back to this thread
   // and coerces piResult.Result out
               await piResult;

   calcButton.IsEnabled = true;
   this.pi.Text = pi.ToString();
}
```

async await

Can return Task<T>

- **Code returns T, compiler returns Task<T>**

```
async Task<byte[]> DownloadDataAsync(Uri source)
{
   WebClient client =new WebClient();
   byte[] data = await client.DownloadDataTaskAsync(source);

   ProcessData(data);

   return data
}
```

Favour continuations over waiting

- Threads aren't free
- A thread waiting can't be used for anything else.
- Using continuations can reduce the total number of required threads

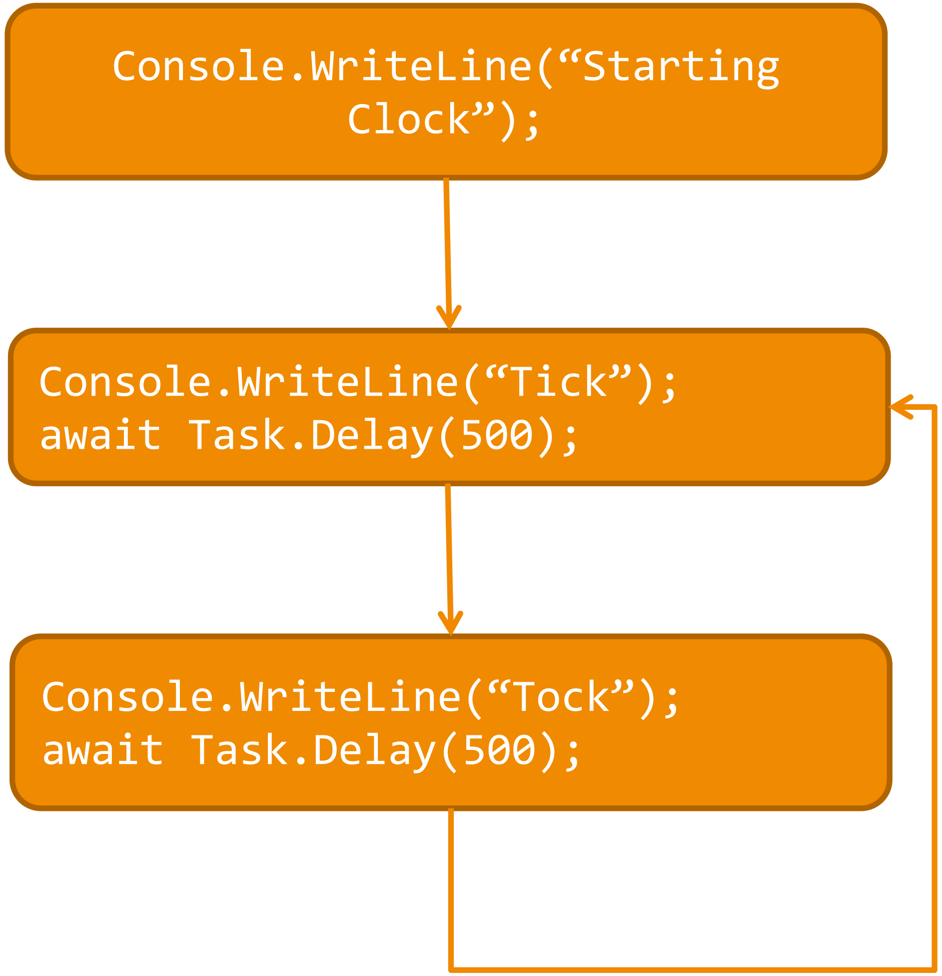
async/await under the hood

Compiler builds state machine

```
private static async void TickTockAsync()
{
 Console.WriteLine("Starting Clock");

while (true)
 {
  Console.WriteLine("Tick");
  await Task.Delay(500);

  Console.WriteLine("Tock");
  await Task.Delay(500);
 }
}
```



async/await

Gotcha

async keyword does not make code run asynchronously

```
async Task DoItAsync()
 {
     // Still on calling thread
       Thread.Sleep(5000);
       Console.WriteLine("done it..");
 }
```

async/await

Gotcha #2

Avoid async methods returning void

```
async void DownloadDataAsync(string uri){
. . .
}
```

- Better to return Task than void
- Allows caller to handle error
- void is there for asynchronous event handlers

```
async Task DownloadData(Uri source)
{
   WebClient client =new WebClient();
   byte[] data = await client.DownloadDataTaskAsync(source);

   ProcessData(data);
}
```

Gotcha #3

THINK before using async lambda for Action delegate

```
requests.ForEach(async request =>
{
   var client = new WebClient();
   Console.WriteLine("Downloading {0}", request.Uri);
   request.Content = await
        client.DownloadDataTaskAsync(request.Uri);
});
Console.WriteLine("All done..??");

requests.ForEach(r => Console.WriteLine(r.Content.Length));
```

async/await

Gotcha #4

- **await exception handling only delivers first exception**
- **Tasks can throw many exceptions via an AggregateException**
  - **Await re-throws only first exception from Aggregate**
- **Examine Task.Exception property for all errors**

```
Task<byte[]> loadDataTask = null;
try {
  loadDataTask = LoadAsync();
  byte[] data = await loadDataTask;
  ProcessData(data
} catch(Exception firstError) ){
     loadDataTask.Exception.Flatten().Handle( MyErrorHandler );
}
```

ConfigureAwait

Possibly the worst API ever conceived

- Not all await's need to make use of SynchronizationContext

```
public static async Task DownloadData(Uri source, string destination)
{
  WebClient client =new WebClient();
  byte[] data = await client.DownloadDataTaskAsync(source);

// DON'T NEED TO BE ON UI THREAD HERE…
  ProcessData(data);

  using (Stream downloadStream = File.OpenWrite(destination))
  {
    await downloadStream.WriteAsync(data, 0, data.Length);
  }
  // Must be back on UI thread
  UpdateUI("Download
```

ConfigureAwait

Possibly the worst API ever conceived

- First attempt, but wrong

```
 static async Task DownloadData(Uri source, string destination)
{
  WebClient client =new WebClient();
                  await client
                           .DownloadDataTaskAsync(source)
                           .ConfigureAwait(false);

   // Will continue  not on UI thread
          Stream downloadStream = File.OpenWrite(destination)) {
    await downloadStream.WriteAsync(data, 0, data.Length);
  }

  // Hmmm…Need to be back on UI thread here

  UpdateUI("All downloaded");
```

ConfigureAwait

Effective use of ConfigureAwait with composition

- **Get compiler to create Task per context**

```
static async Task DownloadData(Uri source, string destination){
   await DownloadAsync(source, destination);
   // on UI thread
   UpdateUI("All downloaded");
}

 private static async Task DownloadAsync(Uri source, string destination) {
    WebClient client = new WebClient();
    byte[] data = await client
                     .DownloadDataTaskAsync(source)
                    .ConfigureAwait(continueOnCapturedContext:false);

    using (Stream downloadStream = File.OpenWrite(destination)) {
       await downloadStream.WriteAsync(data, 0, data.Length);
    }
 }
```

Await and Monitors

- Compile will not allow an await inside a lock block
  - Lock only works if the entire block is executed on the same thread

```
lock (account)
{
    await UpdateAccount().ConfigureAwait(false);
}
```

Won't compile

```
Monitor.Enter(account)
try
{
    await UpdateAccount().ConfigureAwait(false);
}
finally{ Monitor.Exit(account); }
```

Will compile

Utilise Semaphore slim

Await compatible locking

- Semaphore with count of 1 has similar behavior to that of Mutex
- Semapore can be aquired and released around a await
- Can await on a semaphore, for non blocking synchronization
- Wrap in using pattern to maintain programming model

Awaiting with timeout

- Waiting for ever, is bad
- Awaiting for ever possibly less bad
- What if awaiting task has no cancellation?
  - await keyword has no time out
- Consider using Task.WhenAny
- .NET 6 introduces Task.WaitAsync()

Async on the server

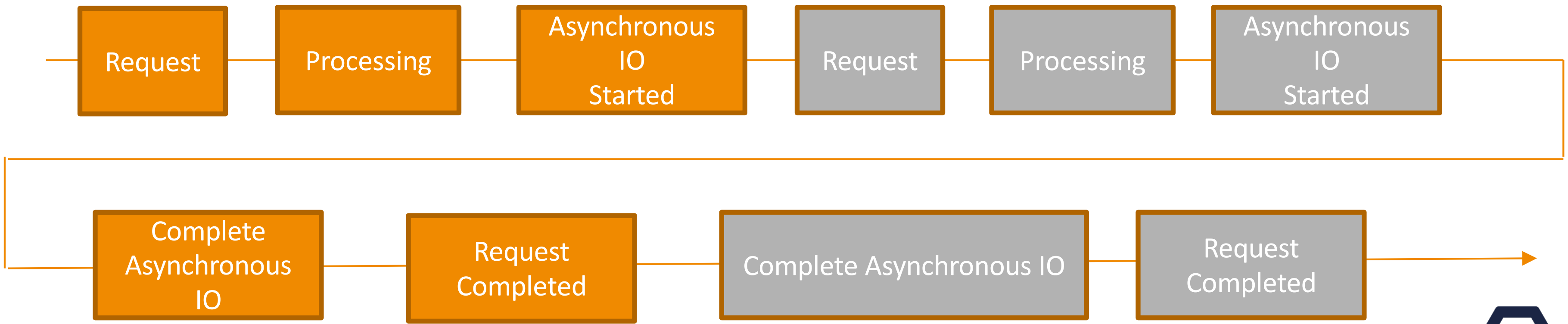
Not just client side technology

- MVC and WebAPI both understand
  - Task<T>
  - ValueTask<T>
- Blocking on a thread is harmful to your application
  - New requests may force a new thread to be created
  - New threads take time to start and consume resource
- Server threads shouldn't block
  - Release thread until they need it again
  - Allow N concurrent requests to share a single thread of execution

Async on the server

Thread re-use

- Thread starts processing a given request
- Initiates some asynchronous IO, and frees itself to perform another request
- Second request initiates asynchronous IO, and frees it self it complete the previous request



AsyncLocal<T>

- Applications sometimes need to flow ambient state
  - HttpContext
  - LoggingContext
- When using async/await can't use thread local storage
- AsyncLocal<T> used to flow ambient state across await boundaries

Async Enumerable aka Async Streams

- Why the need for asynchronous iteration
- Foreach async
- Disposable Async

DIY async iteration

- Asynchronous iteration awkward for creation and consumption

```
public static  IEnumerable<Task<string[]>> LoadCsv(string filename){
  using (var reader = new StreamReader(filename)) {
   while (!reader.EndOfStream){
    yield return LoadAndSplit(reader);
   }
  }
}

private static async Task<string[]> LoadAndSplit(StreamReader reader) {
  return (await reader.ReadLineAsync()).Split(',');
}
```

IAsyncEnumerable<T>

- **Async version of IEnumerable,IEnumerator**

```
public interface IAsyncEnumerable<out T> {
  IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken ct);
}

public interface IAsyncEnumerator<out T> :IAsyncDisposable {

  T Current {get; }
  ValueTask<bool> MoveNextAsync();
}
```

Yield return async enumerable

- Compiler creates an async enumerable
- Methods must by marked async
- Methods must return IAsyncEnumerable<T> or IAsyncEnumerator<T>
- Yield return used as per iterator methods

```
static async IAsyncEnumerable<string[]> LoadCsv(string filename) {
   using (var reader = new StreamReader(filename))
   {
     while (!reader.EndOfStream) {
      string row = await reader.ReadLineAsync();
      yield return row.Split(',');
    }
   }
}
```

Foreach async

- Iterates through async enumerable
- Delivers each item, not Task<T>

```
IAsyncEnumerable<string[]> rows = LoadCsv(@"stockData.csv");

await foreach (string[] row in rows)
{
   Console.WriteLine(row[1]);
}
```

Async LINQ

- Nuget package System.Linq.Async
- IAsyncEnumerable<T> extension methods defined in AsyncEnumerable

```
var tradingDays = LoadCsv(@"stockData.csv")
                .Skip(1)
                .Select(row => new
                {
                    When = DateTime.Parse(row[0]),
                    Open = decimal.Parse(row[1]),
                    Close = decimal.Parse(row[4])
                });

await foreach (var tradingDay in tradingDays) {
  Console.WriteLine(tradingDay);
}
```

Foreach async cancellation

- **Iterator method can take cancellation token**
- **Problem: this is scoped for the IAsyncEnumerable not the IAsyncEnumerator**

```
 var cts = new CancellationTokenSource();
 var rows = LoadCsv("stockData.csv" , cts.Token);

await foreach (string[] row in rows) {
  Console.WriteLine(row[1]);
  cts.Cancel();
}

await foreach (string[] row in rows) {
  Console.WriteLine(row[1]); // NEVER EXECUTES
}

public static async IAsyncEnumerable<string[]> LoadCsv(
               string filename , CancellationToken ct)
```

Enumeration cancellation

- Cancellation can be scoped to the enumerator
- Iterator method parameter marked to accept cancellation token

```
 var cts = new CancellationTokenSource();
 var rows = LoadCsv("stockData.csv" , CancellationToken.None);

await foreach (string[] row in rows.WithCancellation(ct.Token) {
  Console.WriteLine(row[1]);
  cts.Cancel();
}

public static async IAsyncEnumerable<string[]> LoadCsv(
               string filename ,
               [EnumeratorCancellation]CancellationToken ct)
```

Async Disposable

- New interface IAsyncDisposable
- Using statement prefix with await uses DisposeAsync

```
public interface IAsyncDisposable  {
  ValueTask DisposeAsync();
}
```

```
await using (FileStream s = File.OpenRead("stockData.csv")){

}
```

Async iteration on the server

- **MVC and WebAPI both understand IAsyncEnumerable<T>**

```
[Route("TradingDays")]
[HttpGet]
public IAsyncEnumerable<TradingDayDTO> GetStocks()
{
  var rows = Context.TradingDays;
  return rows.Select(r => new TradingDayDTO()
    {
      When = r.When.ToShortDateString(),
      Close = r.Close,
      Open = r.Open,
      Volume = r.Volume
    }).AsAsyncEnumerable();
}
```

Summary

- Utilise async/await to
  - Simplify continuations
  - Reduce number of threads
- Use Semaphore as an await safe locking primitive
- Use ConfigureAwait to reduce work on UI thread
- Use asynchronous apis for greater scalability and performance
- Use IAsyncEnumable for streaming asynchronous results

Thread Safety

Agenda

- Highlight issues with multi threaded programming
- Introduce thread synchronization primitives
- Introduce thread safe collections

Need for Synchronization

- Creating threads is easy
- When threads share data problems can occur
  - Inconsistent reads
  - State corruption
- Synchronization fixes these problems, but potentially creates a new problem
  - Over synchronization reduces scalability
- Lots of techniques to implement synchronization
  - Each have cost and benefit
- Developers role is to write an application that scales and is thread safe, by selecting the best synchronization technique

Simple Increment

- Two threads
  - Sharing an instance of Counter.
  - Both are calling Increment 1000 times
- Question
  - What is the value of count after both threads have completed?

```
public class Counter
{
  protected int count;

  public virtual void Increment()
  {
    count++;
  }
  public int Value { get { return count; }  }
}
```

Simple Increment, NOT Atomic

- Even a simple count++ is not an atomic operation.
  - Multiple CPU instructions that could be interweaved.

time

```
T0: MOV R0,count
T0: ADD R0,1          T1: MOV R0,count
T0: MOV count,R0      T1: ADD R0,1
                               T1: MOV
count,R0
```

- Consider the possible execution below of two threads (T0, T1)
  - Assuming count=0 at the start
  - At the end of execution i would be 1 and not the desired 2.
- If two threads don't attempt to increment count at the same time not a problem. Spotting these kind of errors is hard

Interlocked

- Modern CPU's expose special instruction set to perform various operations atomically
  - Cost more than non atomic variants.

```
public class InterlockedCounter : Counter
{
  public override void Increment()
  {
    // Atomic count++
    Interlocked.Increment(ref count);
  }
}
```

- Access to these instructions via Interlocked class
  - Interlocked.Increment
  - Interlocked.Decrement
  - Interlocked.Add
  - Interlocked.Exchange, Interlocked.CompareAndExchange
    - Useful for building Spin locks

Multi step state transition

- What happens if
  - Thread A is inside ReceivePayment
  - Thread B is inside NetWorth
- Can Interlocked help ?

```
 class SmallBusiness {
   decimal Cash = 0;
   decimal Receivables = 1000;

   public void ReceivePayment(decimal amount){
     Cash += amount;
     Receivables -= amount;
   }

   public decimal NetWorth {
     get { return Cash + Receivables; }
   }
}
```

Sequential access

- To fix the problem
  - Sequentialise access to the object state
- How
  - Each instance of a reference type has a Monitor
  - CLR guarantees that only one thread can own the monitor
  - If a thread can't acquire the monitor it enters a wait state
  - When the monitor is available it is woken up and proceeds
- Critical areas of code can therefore be protected by using a monitor.

Monitor based solution

- Only one thread in any critical region at any point in time

```
private object _lock = new object();

public void ReceivePayment(decimal amount)
{
  Monitor.Enter(_lock);
    Cash += amount;
    Receivables -= amount;
  Monitor.Exit(_lock);
}
```

Could be an issue with exceptions

```
public decimal NetWorth
{   get {
    Monitor.Enter(_lock);
    try { return Cash + Receivables; }
    finally { Monitor.Exit(_lock);
    }
   }
}
```

Deals better with exceptions

Lock keyword

- Enter, try, finally , Exit common pattern
  - C# language offers lock keyword to assist
  - Compiler emits try, finally logic
- Use of Monitor.Enter and lock can lead to deadlocks
  - Prefer Montior.TryEnter which takes a timeout
- Avoid using lock(this) and lock(typeof(X))
  - Less control over objects use for synchronization.
  - Prefer creation of object for sole purpose of

```
public void ReceivePayment(decimal amount){
 lock (_lock)
 {
   Cash += amount;
   Receivables -= amount;
 }
}
```

High Read to Write Ratio

- Monitor provides mutual exclusion behaviour
  - Excluding readers and writers
- Thread safety not an issue if all threads read.
- Better throughput may be achieved with a Synchronization primitive that ensures
  - There can be Many Readers, Zero Writer
  - Or One Writer, Zero Readers
- This is known as a ReaderWriterLock
  - .NET 3.5 and above prefer ReaderWriterLockSlim
  - Pre 3.5, ReaderWriterLock
    - – Not well implemented, can result in writer being denied access for long periods of time.
- Often used for caching

Reader Writer Lock

```
public class SimpleCache
{
 private Dictionary<int, string> cache=
              new Dictionary<int, string>();
 private ReaderWriterLockSlim _lock = new ReaderWriterLockSlim();

 public string Get(int key) {
   _lock.EnterReadLock();
    try { return cache[key]; }
    finally { _lock.ExitReadLock(); }
}

 public void Set(int key, string val)
 {
   _lock.EnterWriteLock();
   try { cache.Add(key,val) }
   finally { _lock.ExitWriteLock(); }
 }
}
```

Many threads can can read from the cache

When one thread has the write lock no other thread can obtain read or write lock.

Semaphores

- Primitive that can be acquired for a set number of times
  - Often used to implement throttling
- Semaphore can be released from any thread, unlike Monitor
  - Semaphore count of one similar to semantics to Monitor
- Two implementations in the framework
  - SemaphoreSlim
  - Semaphore, wraps Kernel level primitive

Synchronization across process

- Managed synchronization primitives only allow synchronization inside a single process
- How to control access to a shared file ?
  - Requires Kernel based synchronization
- Kernel synchronization can be achieved via managed wrappers
  - Mutex
  - Semaphore
  - AutoResetEvent
  - ManualResetEvent
- These synchronization primitives are orders of magnitude more expensive than managed ones

Summary

- A variety of ways to perform synchronization, the skill is picking the correct one
- Concurrent collections make it simpler to write efficient thread safe code
- Only use kernel synchronization primitives when absolutely necessary
- Analyse code and imagine worse possible race conditions

Simplifying thread safety

Agenda

- Greater concurrency without complexity
- Lazy<T>
- Concurrent Collections
  - Blocking Collections
- Channels .NET Core
- Immutable Collections

Thread Safe code can be complex

- Single threaded algorithms if written well simple to understand
- Add threads possibly require locking
  - Simple lock can be inefficient
  - Perhaps ReaderWriter Locks
  - Perhaps double check locking
- RESULT: Complex code hard to maintain original intent often lost

Need for locks

- Mutable shared state
- Shared state often takes the form of
  - Collections (List, Dictionary , Queue , Stack)
- Solutions
  - Thread safe collections, hide synchronization
    - Concurrent collections
  - All shared state is immutable
    - Immutable collections

Lazy<T>

- Provides thread safe on first read initialisation
  - Cheap stand in
- Useful
  - For delay loading the contents of a collection
  - Thread safe Virtual Proxy

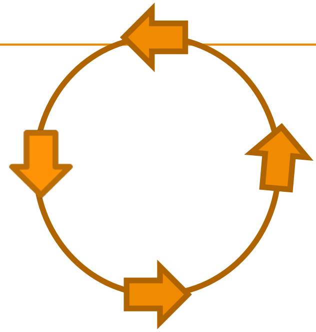
Concurrent Collections

- Collections are the bedrock of most apps
  - List, Dictionary, Queue, Stack
- Problem, not thread safe
- Synchronized proxies/wrappers don't cut it

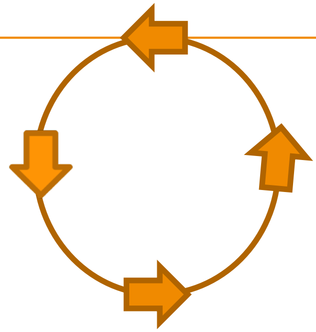
Consider this

```
Queue<int> queue = new Queue<int>();
queue.Enqueue(1);
. . .
```

```
if (queue.Count > 0)
{
  int val = queue.Dequeue();
}
```



```
if (queue.Count > 0)
{
  int val = queue.Dequeue();
}
```



Concurrent Collections
Concurrent API

- If/do, introduces race conditions
- Concurrent collection API remove if/do
  - TryXXXX
  - More complex atomic operations
    - AddOrUpdate
    - GetOrAdd
- WARNING…Be careful when using extension methods based on non-concurrent interfaces.
  - ToList()

ConcurrentDictionary<K,V>

- 30-40% insert speed improvement in 4.5
  - Re-use Nodes for reference and small value types
  - Number of locks change as structure grows
- Initialise with potential size and level of concurrency for best performance

ConcurrentBag<T>

- List keeps items in order
- Bag keeps items
- What is it NOT
  - IT IS NOT A THREAD SAFE UNORDERED LIST
- It is ideally for load balancing divide/conquer

What if I need to block

- Concurrent data structures don't block
  - Highly concurrent
- If require value before proceeding consider blocking
- BlockingCollection<T>
  - Adds block semantics to implementors of
    - IProducerConsumerCollection <T>

Issue with blocking collections

- Blocking a thread pool thread is RUDE
- To scale well
    - Minimum number of threads maximum concurrency
- async/await provides convent programming model to release and resume thread usage

Asynchronous queue

.NET Core

- Channel<T> for producer consumer pattern
  - Supports asynchronous reads and writes
  - Support IAsyncEnumerable
- Supports bounded and unbounded queues
  - `Channel.CreateUnbounded<T>();`
  - `Channel.CreateBounded<T>(size);`
- Can be optimized for
  - Single Reader
  - Single Writer
  - Synchronous writes and reads

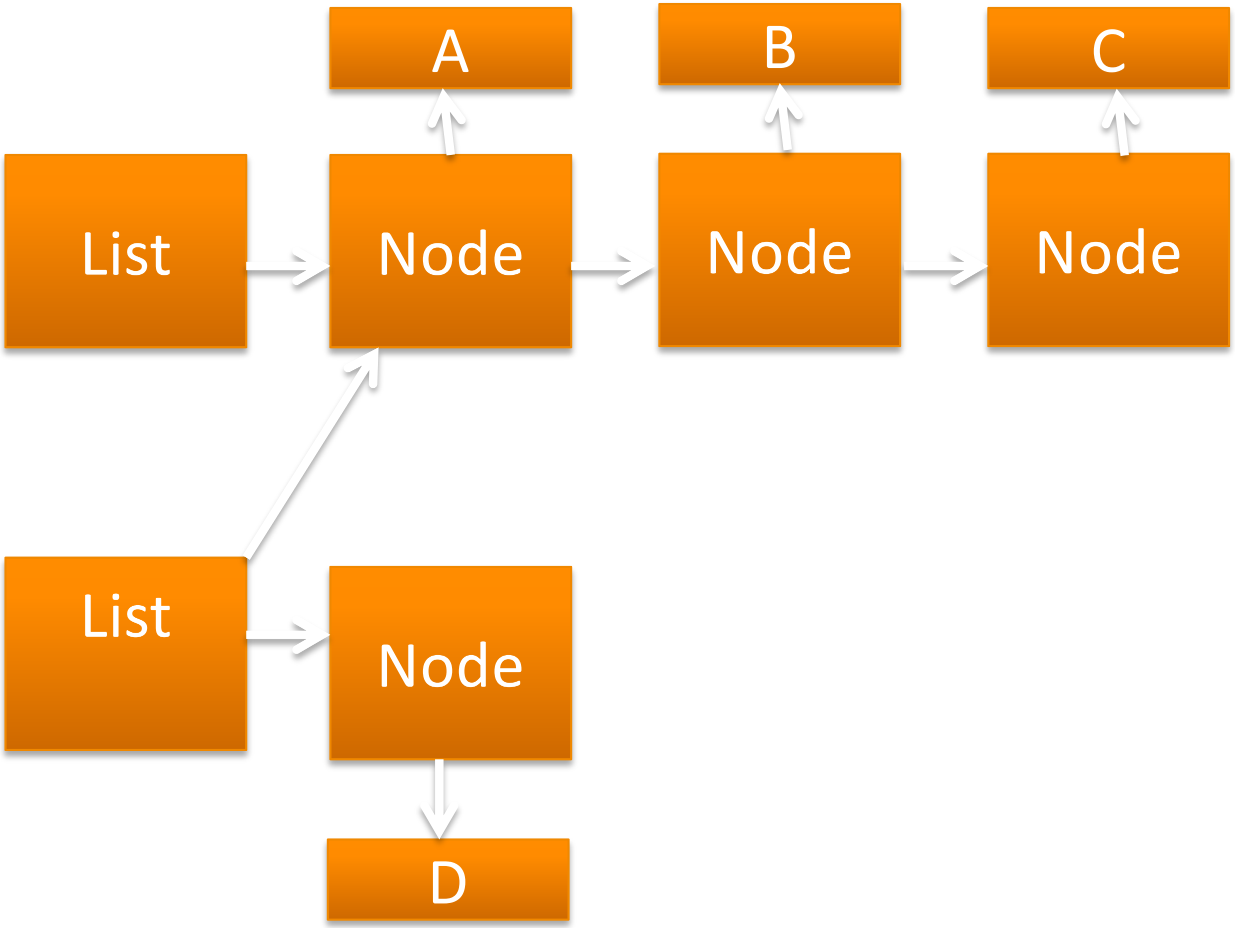
Channel Types

- Unbounded channel
  - Assumes memory never runs out
  - Best for performance
- Bounded channel
  - Constrain number of items in the channel
  - Configurable when full behavior
    - Wait
    - Drop Newest
    - Drop Oldest
    - Drop Write

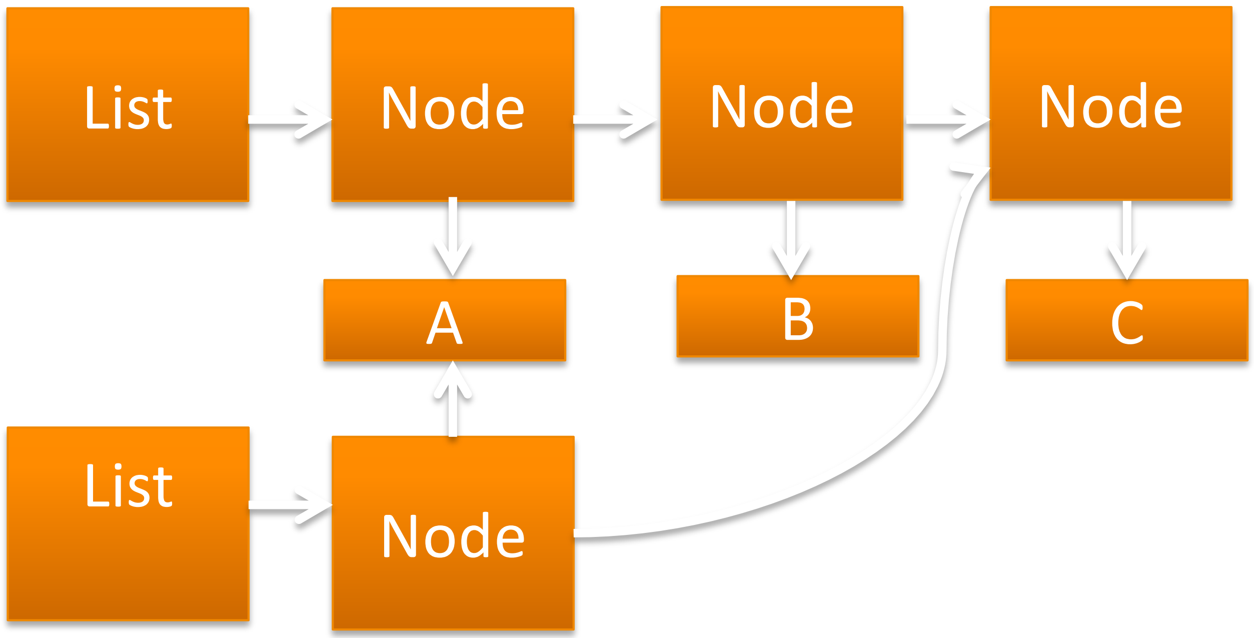
Immutable Collections

- Thread Safety can be hard with mutable data It's a breeze with immutable data
- Not easy to achieve
- NuGet Microsoft Immutable collections
- Mutable operations results in efficient creation of new collection

Add item D to Immutable list



Remove item B to Immutable list



Summary

- Thread safety now achieved with high level abstraction
  - Maintains readability
  - Greater confidence it works
  - Leverage on going development