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

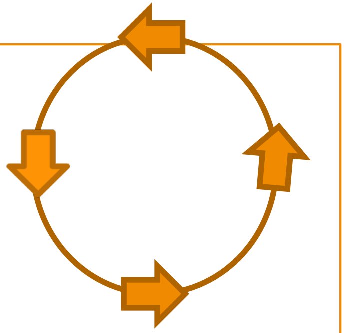
Concurrent Collections

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

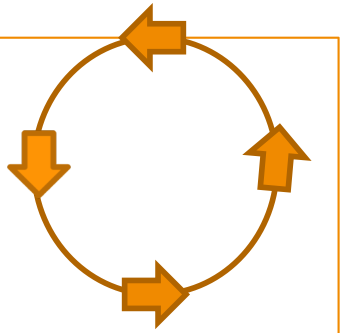
Consider this

```
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

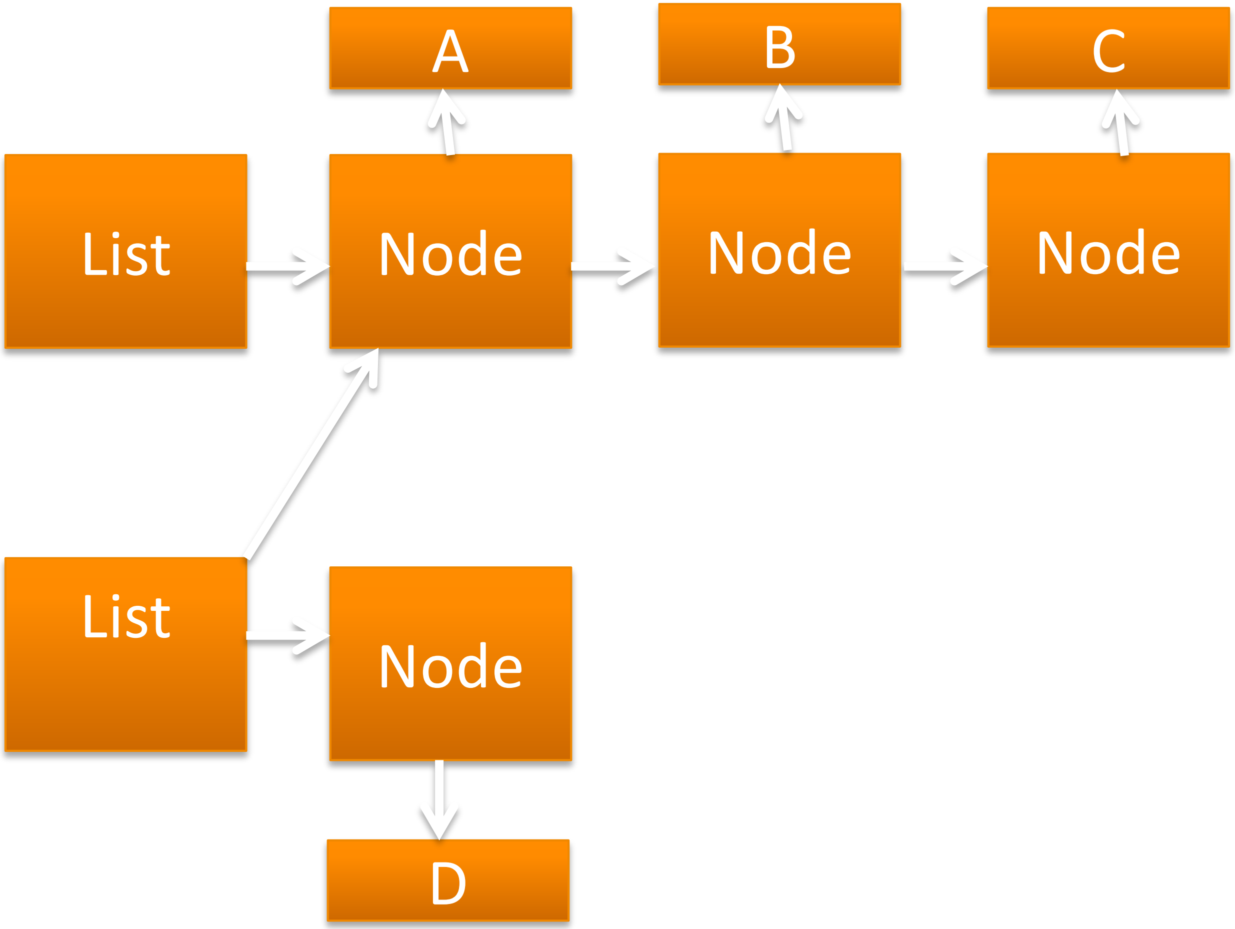
Channel Types

- 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

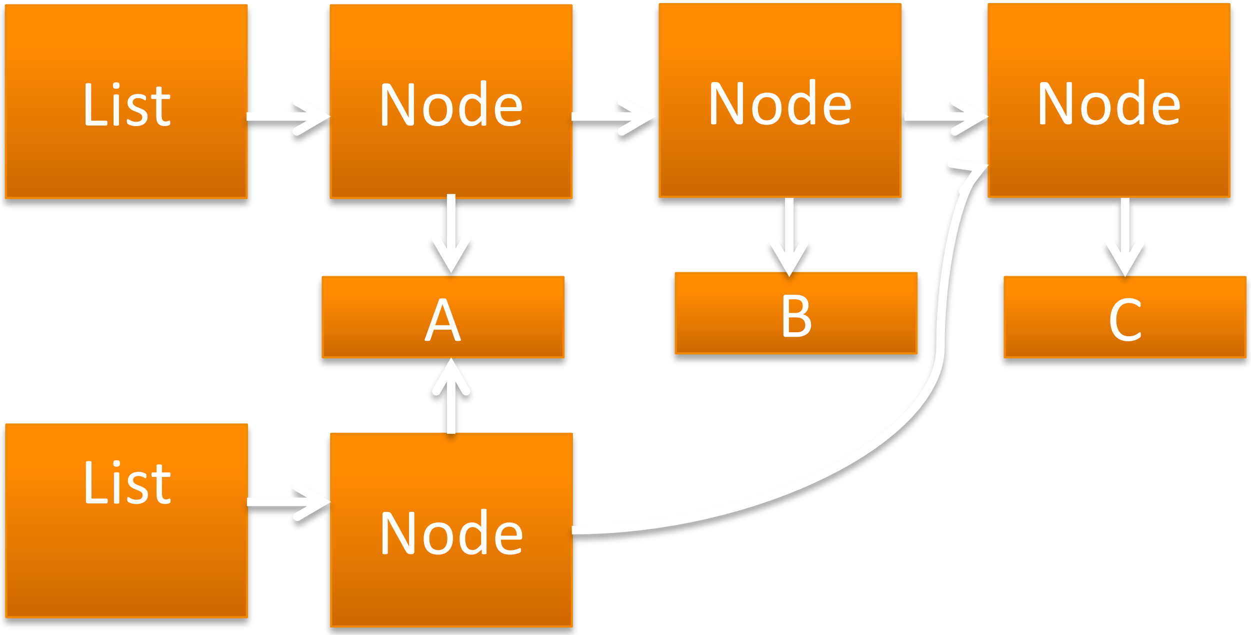
Immutable Collections

- 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