Concurrent Data Structures

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Agenda

- **Concurrent data structures**

Managing concurrency in the .NET 3.5 World

- **All collections non-threadsafe**
  - Developer must select right granularity for synchronization
  - Developer must write synchronization code
  - Lock free algorithms an “arcane art”
- **Limited synchronization primitives available**
  - Basic primitives available: Monitor, Interlocked
  - Higher order primitives often implemented as wrapper over kernel
  - Higher order primitives could be built from basic ones using Monitor.Wait / Monitor.Pulse
- **PFx has changed the requirements**
  - Needs concurrent data structures and primitives internally
  - Has made them public for everyone to use

Lazy<T>

- **Common requirement for lazy initialization**
- **New type System.Lazy<T> performs lazy initialization**
  - Type created on first access of Value property
  - Uses default constructor to create type

```
class Person
{
  public string Name { get; set; }
  public int Age { get; set; }
}
```

```
Lazy<Person> myvar = new Lazy<Person>();

myvar.Value.Name = "Rich";
myvar.Value.Age = 44;
```

Lazy<T> and Thread Safety

- **By default Lazy<T> is not thread safe**
  - Could result in more than one instance of the contained object being created
  - Not necessarily an issue if creation is cheap and has no side-effects
- **Can pass a flag to the constructor to ensure thread safety**
  - Only affects the thread safety of instantiation, not of subsequent access
  - Only worthwhile if object expensive to create or has side-effects in creation and eager creation is inappropriate

```
Lazy<Person> myvar = new Lazy<Person>(true);

myvar.Value.Name = "Rich";
myvar.Value.Age = 44;
```

Overriding Default Initialization

- **Can use a delegate in place of the default constructor for initializing the contained type**

```
class Company {
  Lazy<List<Person>> employeeHolder;
  public Company() {
    employeeHolder = new Lazy<List<Person>>(GetEmployees);
  }

  public IEnumerable<Person> Employees {
    get { return employeeHolder.Value; }
  }

  List<Person> GetEmployees() {
    return new List<Person>() { //... };
  }
}
```

Concurrent Collections

- **System.Collections.Generic supports non-thread-safe general purpose collections**
  - Ideal for many uses
  - Sub-optimal for concurrent code as thread-safety is heavyweight
- **System.Collections.Concurrent introduces a group of collection designed for concurrent use**
  - ConcurrentQueue<T>
  - ConcurrentStack<T>
  - ConcurrentDictionary<T>
  - ConcurrentBag<T>
- **Collections internally thread-safe preferring lock free algorithms**

Concurrent Collection Pattern

- **All collections implement deterministic “Add”**
  - ConcurrentQueue<T>.Enqueue
  - ConcurrentStack<T>.Push
- **All Collections implement non-deterministic “Get”**
  - ConcurrentQueue<T>.TryDequeue
  - Allows non-blocking attempt at retrieval
- **Some implement atomic complex operations**
  - ConcurrentDictionary<K,V>.GetOrAdd

IProducerConsumerCollection<T>

- **Concurrent collections all implement `IProducerConsumerCollection<T>`**
  - `TryAdd` and `TryTake` model non-blocking add and remove
  - All collections add more specialized methods

```
public interface IProducerConsumerCollection<T> :
                     IEnumerable<T>, ICollection, IEnumerable
{
    void CopyTo(T[] array, int index);
    T[] ToArray();
    bool TryAdd(T item);
    bool TryTake(out T item);
}
```

Producer / Consumer Issues

- **Non blocking "Take" requires spinning or polling**

Spinning

```
while (!terminate)
{
  int val;
  if (queue.TryDequeue(out val))
  {
    ProcessData(val);
  }
}
```

Polling

```
while (!terminate)
{
  int val;
  if (queue.TryDequeue(out val))
  {
    ProcessData(val);
  }
  Thread.Sleep(200);
}
```

BlockingCollection<T>

- **Simpler programming model if “Take” blocks**
- **BlockingCollection<T> used as “decorator”**

```
ConcurrentQueue<int> queue = new ConcurrentQueue<int>();
BlockingCollection<int> col =
                    new BlockingCollection<int>(queue);
bool terminate = false;
while (!terminate)
{
  int i  = col.Take();
  ProcessData(i);
}
```

Blocks

Producer / Consumer with BlockingCollection

- **Producer / Consumer requires simple consumer model and mechanism for producer to say “production complete”**

```
ConcurrentQueue<int> q = new ConcurrentQueue<int>();
BlockingCollection<int> col = new BlockingCollection<int>(q);
```

Producer

```
Random r = new Random();

for (int i = 0; i < 10; i++)
{
  col.Add(r.Next(100));
  Thread.Sleep(500);
}

col.CompleteAdding();
```

Consumer

```
foreach (var item in
    col.GetConsumingEnumerable())
{
  Console.WriteLine(item);
}
```

Consuming enumeration completes when **`CompleteAdding`** is called (note: **`Take()`** throws an exception when **`CompleteAdding`** is called)

Summary

- **New concurrent collections**
  - Written to be as lock free as possible
  - Utilise best practice
  - Removes need for all developers to have this knowledge