C# Concurrent Collections

Introducing the Concurrent Collections

Simon Robinson

@TechieSimon | TechieSimon.com

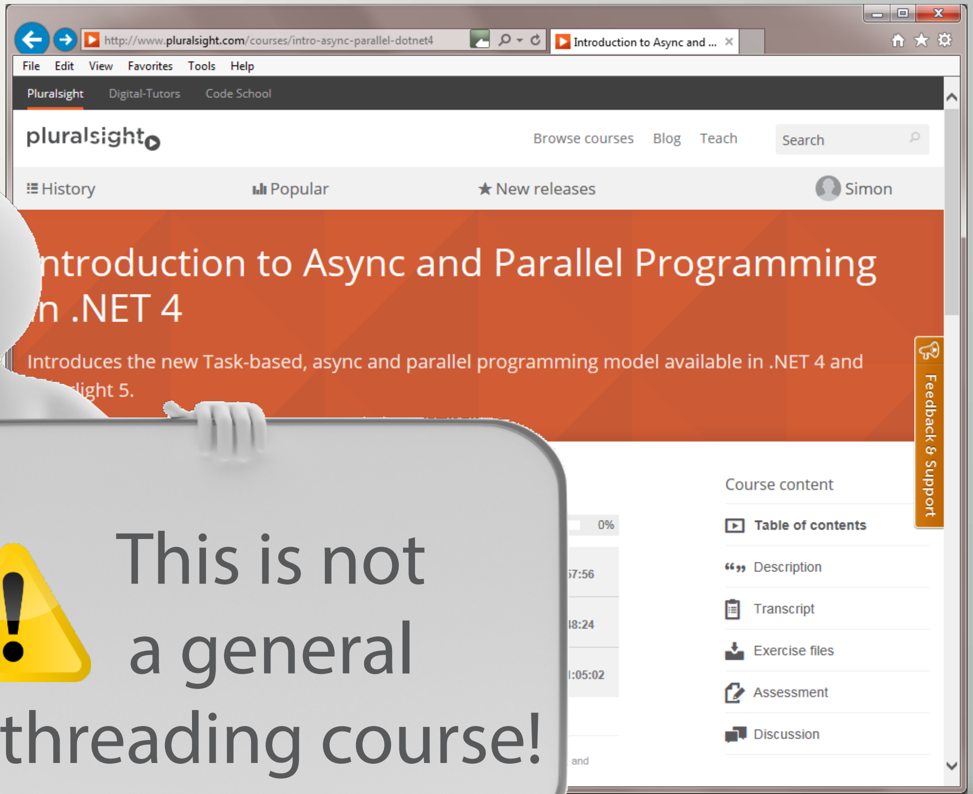
Course Overview

Concurrent dictionary	Producer-consumer	Best practices
3. Concurrent Dictionary Demo	5. Producer-Consumer and BlockingCollection Demo	
2. Introducing Concurrent Dictionary	4. Queues, Stacks and Bags	6. Some best practices

1. Introducing the Concurrent Collections

Prerequisites

Threads and tasks



http://www.pluralsight.com/courses/intro-async-parallel-dotnet4

Introduction to Async and ...

File Edit View Favorites Tools Help

Pluralsight Digital-Tutors Code School

pluralsight Browse courses Blog Teach Search

History Popular ★ New releases Simon

ntroduction to Async and Parallel Programming n .NET 4

Introduces the new Task-based, async and parallel programming model available in .NET 4 and ...light 5.

Course content

- Table of contents
- Description
- Transcript
- Exercise files
- Assessment
- Discussion

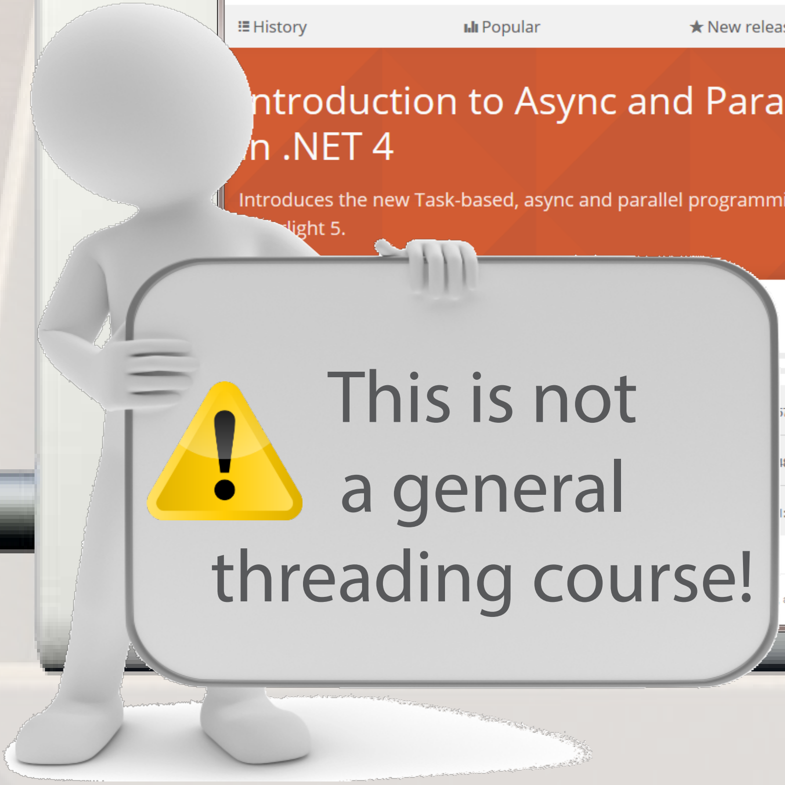
0%

57:56

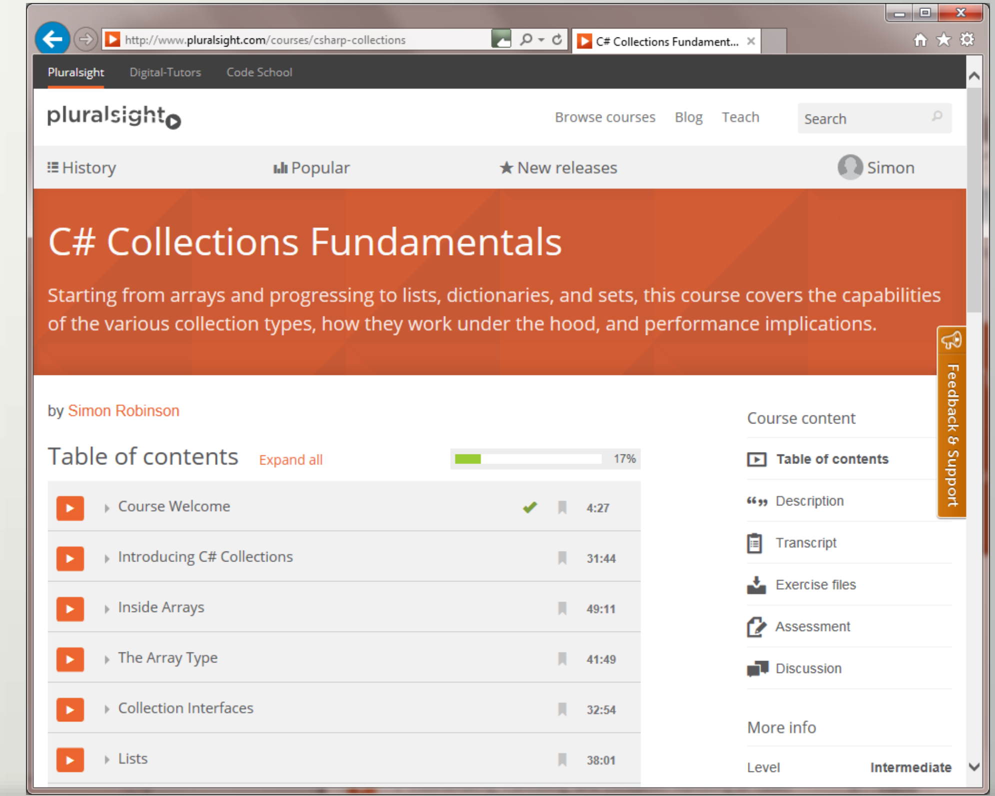
48:24

1:05:02

Feedback & Support



Collections



http://www.pluralsight.com/courses/csharp-collections

C# Collections Fundament...

Pluralsight Digital-Tutors Code School

pluralsight Browse courses Blog Teach Search

History Popular ★ New releases Simon

C# Collections Fundamentals

Starting from arrays and progressing to lists, dictionaries, and sets, this course covers the capabilities of the various collection types, how they work under the hood, and performance implications.

by Simon Robinson

Table of contents Expand all 17%

Course Welcome ✓	4:27
Introducing C# Collections	31:44
Inside Arrays	49:11
The Array Type	41:49
Collection Interfaces	32:54
Lists	38:01

Course content

- Table of contents
- Description
- Transcript
- Exercise files
- Assessment
- Discussion

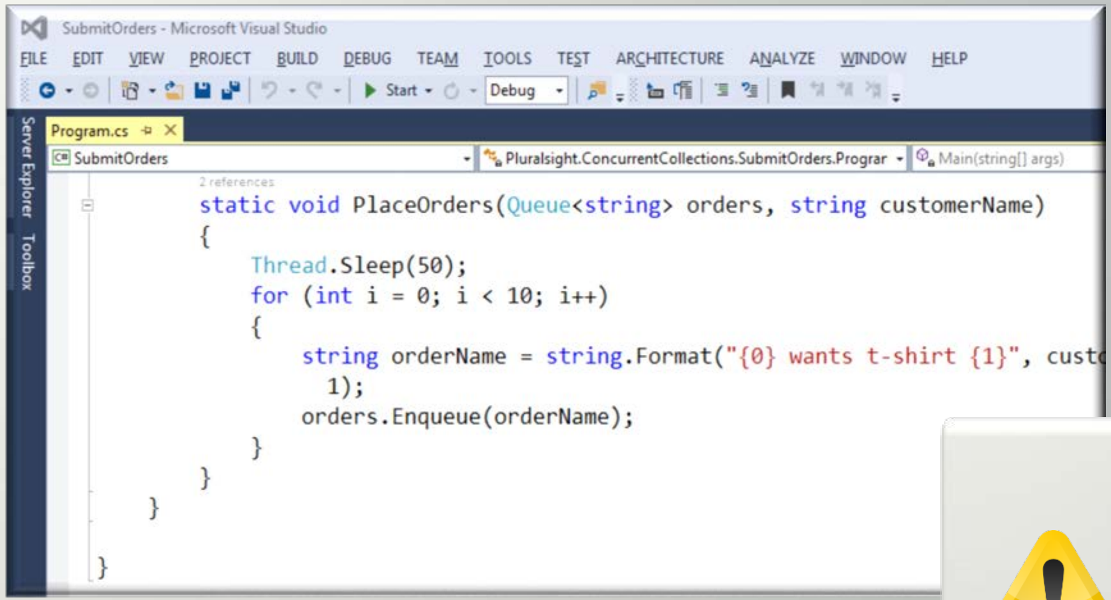
More info

Level Intermediate

Feedback & Support

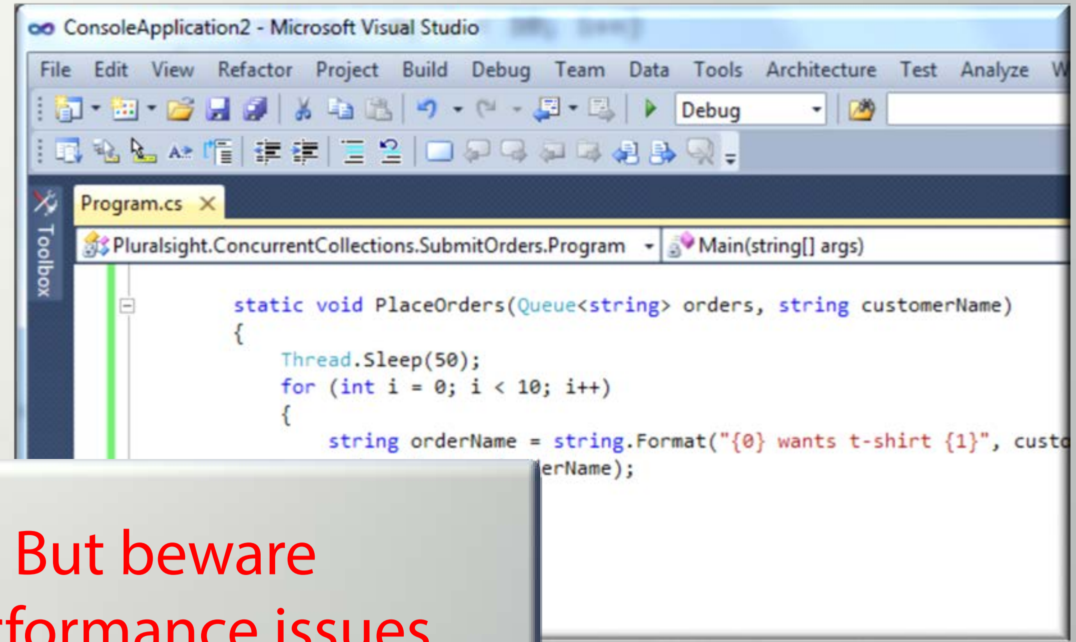
.NET Versions

We use
.NET 4.5/VS 2013


SubmitOrders - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Start
Debug
Server Explorer
Toolbox
Program.cs
SubmitOrders
Pluralsight.ConcurrentCollections.SubmitOrders.Prograr
Main(string[] args)
2 references

```
        static void PlaceOrders(Queue<string> orders, string customerName)
        {
            Thread.Sleep(50);
            for (int i = 0; i < 10; i++)
            {
                string orderName = string.Format("{0} wants t-shirt {1}", cust
                  1);
                orders.Enqueue(orderName);
            }
        }
    }
}
```

You can use
.NET 4.0/VS 2010 or later


ConsoleApplication2 - Microsoft Visual Studio
File Edit View Refactor Project Build Debug Team Data Tools Architecture Test Analyze W
Debug
Toolbox
Program.cs
Pluralsight.ConcurrentCollections.SubmitOrders.Program
Main(string[] args)

```
        static void PlaceOrders(Queue<string> orders, string customerName)
        {
            Thread.Sleep(50);
            for (int i = 0; i < 10; i++)
            {
                string orderName = string.Format("{0} wants t-shirt {1}", custo
                erName);
```

But beware
performance issues
in .NET 4.0!

Module 1 Overview

- *Demo*: Why we need concurrent collections
- What concurrent collections can/can't protect you from
- *Which collections* have concurrent equivalents
  - `ConcurrentDictionary` for most general-purpose scenarios

CODE DEMO

This slide must not appear in the recorded course

CODE DEMO

Grey area must not appear in the recorded course

But it's not so simple…

The hard part!

CODE DEMO

This slide must not appear in the recorded course

Atomicity

An **atomic** method…

Other threads never see it as part-complete

Will either finish or fail *without modifying data*

Atomicity

✗ `Queue.Enqueue()` is not atomic

✓ `ConcurrentQueue.Enqueue()` is atomic

CODE DEMO

This slide must not appear in the recorded course

Locks Are Hard…

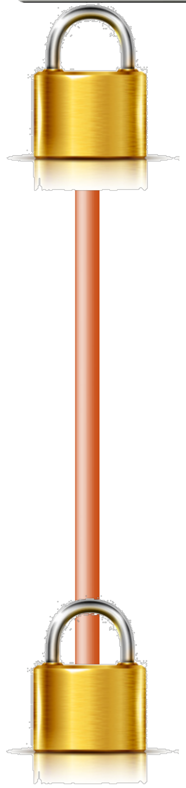
Locking only works if you lock **everywhere** that shared state is vulnerable

It's easy to forget somewhere

The logic to avoid deadlocks is hard

Locks and Scalability

```
lock (syncObj) {…
```

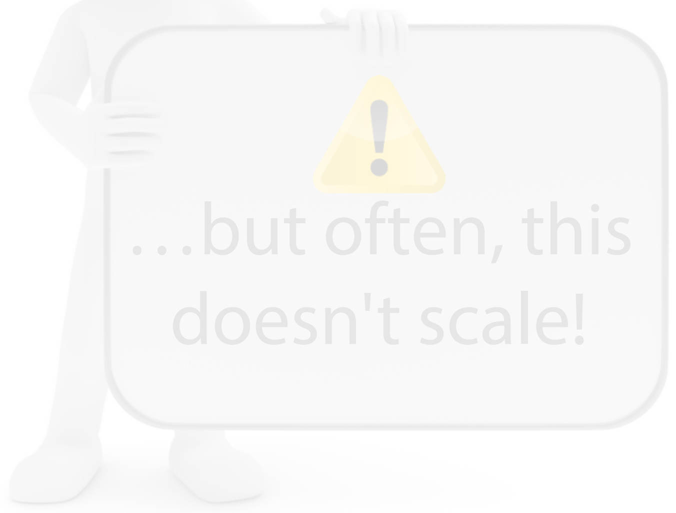


Often good on a small scale…

Locks and Scalability

```
lock (syncObj) {…
```

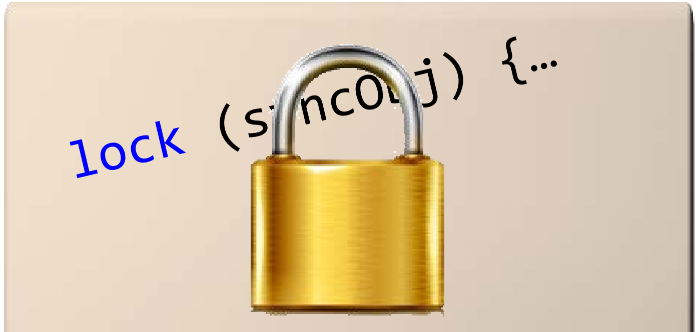
Concurrent collections avoid these problems



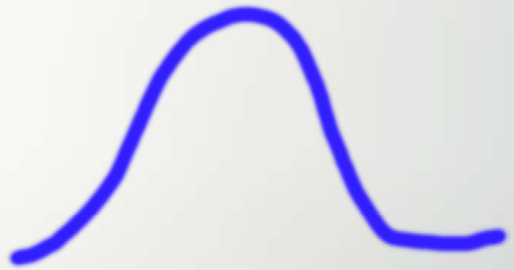
Concurrent Collection Internals

Do they always use locks under the hood?

Thread Synchronization Techniques



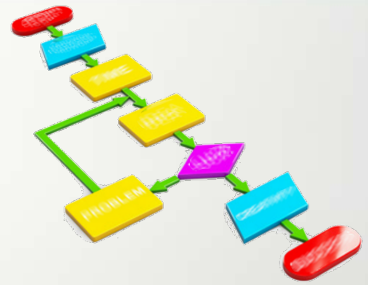
locks



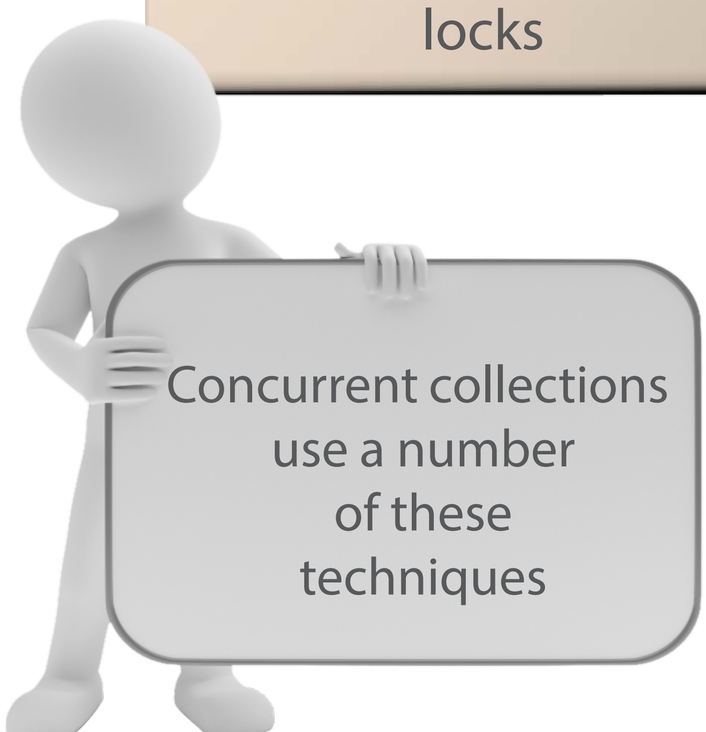
Memory barriers

Special atomic assembly instructions

Other synchronization primitives



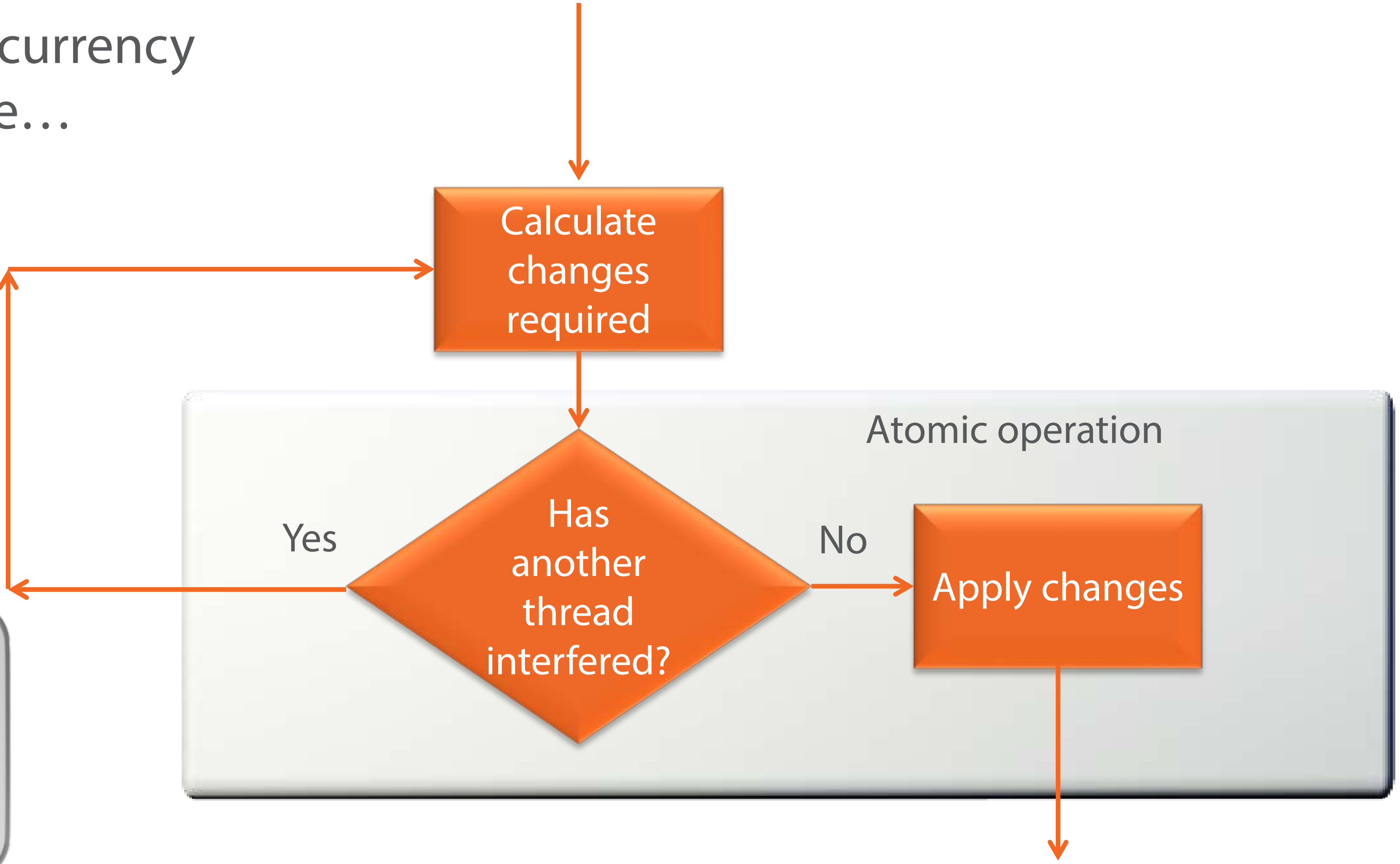
Clever algorithms



Concurrent collections use a number of these techniques

Example: Update ConcurrentDictionary

Optimistic concurrency technique…



Concurrent Collections

They work

They are thread-safe
(against internal data corruption)

They are scalable

Concurrent Collections

✓

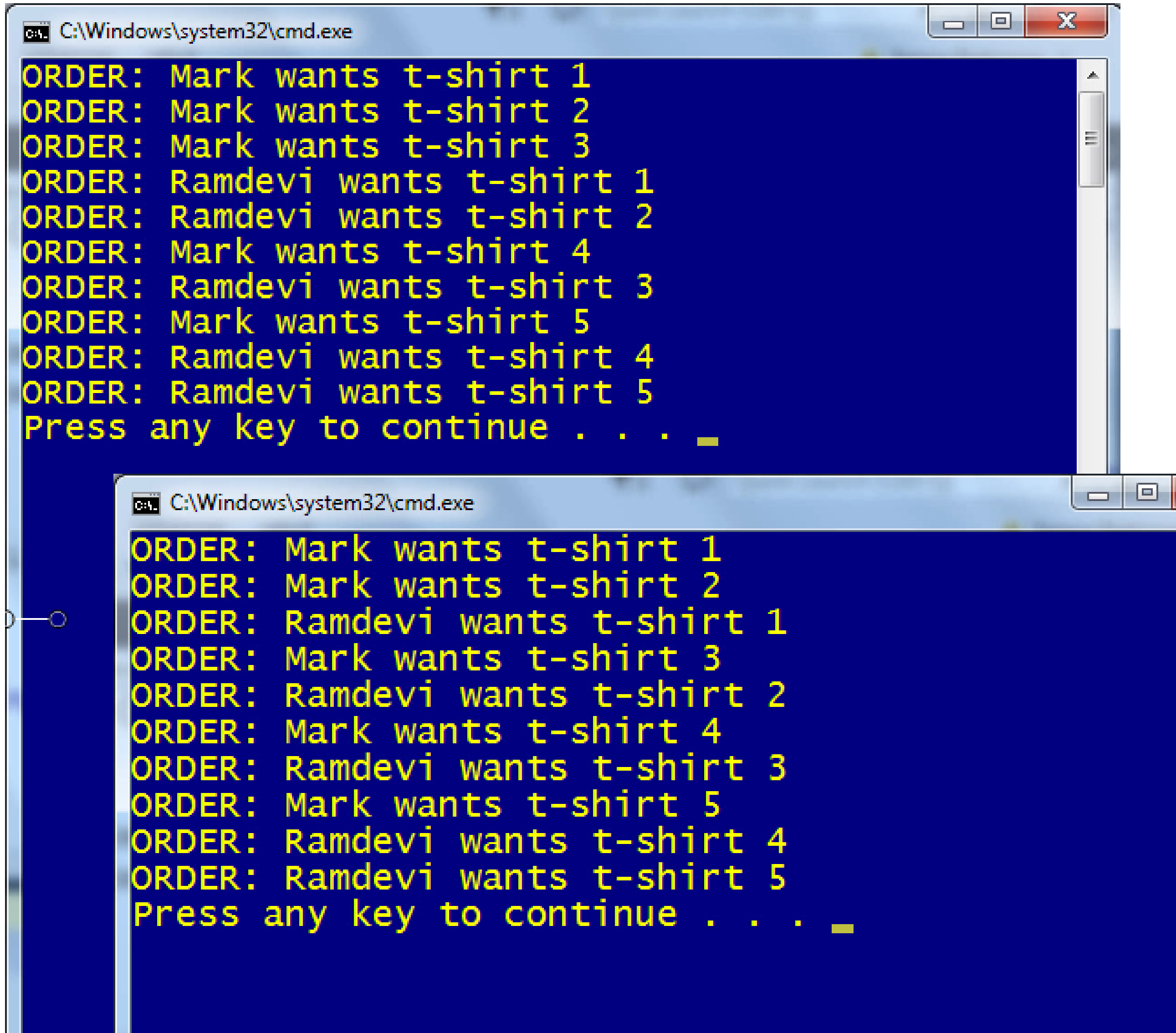
Will protect you from Internal data corruption

What about race conditions?

Race Conditions

Race condition:

Results are sensitive
to precise timing of threads



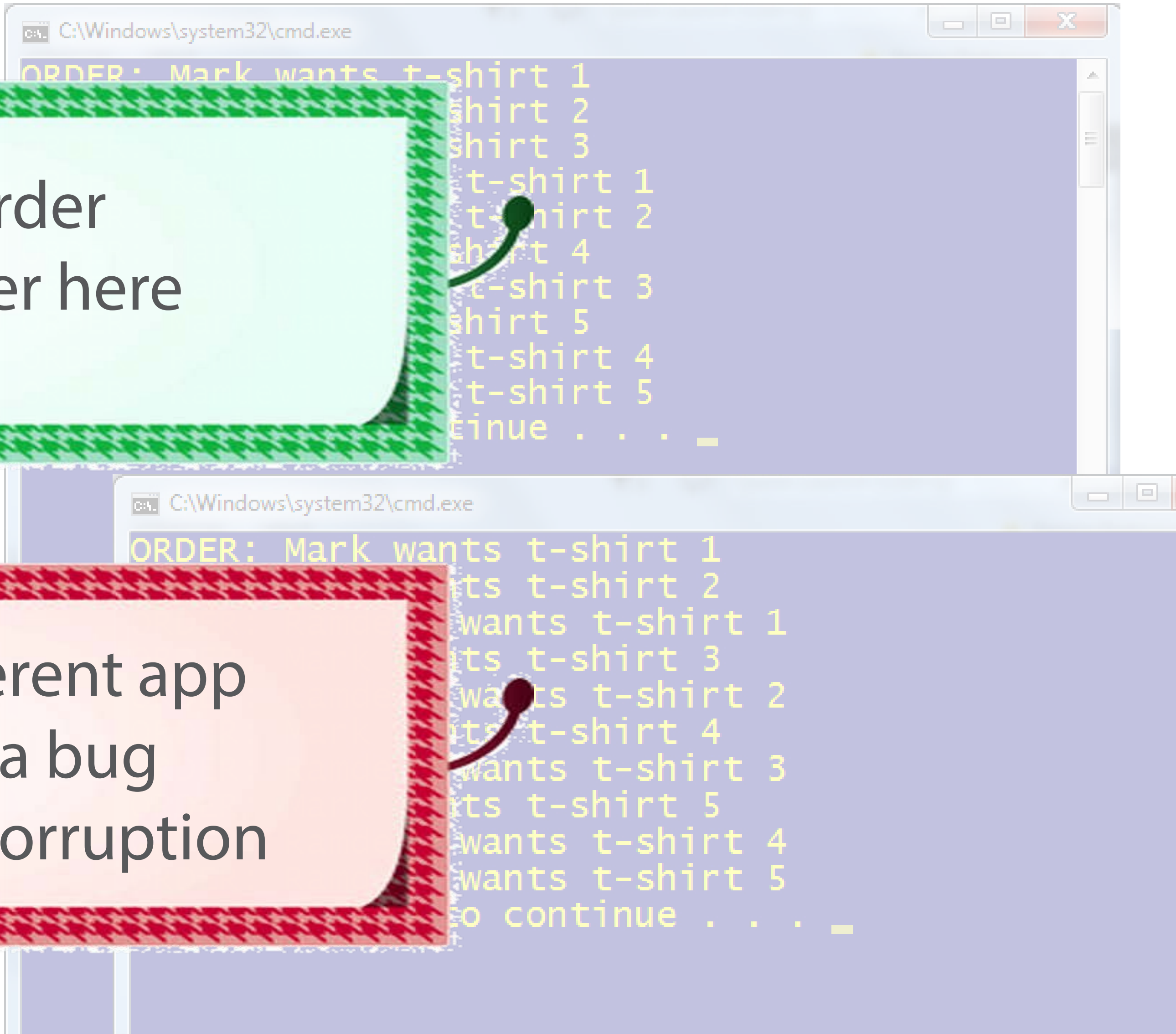
```
ORDER: Mark wants t-shirt 1
ORDER: Mark wants t-shirt 2
ORDER: Mark wants t-shirt 3
ORDER: Ramdevi wants t-shirt 1
ORDER: Ramdevi wants t-shirt 2
ORDER: Mark wants t-shirt 4
ORDER: Ramdevi wants t-shirt 3
ORDER: Mark wants t-shirt 5
ORDER: Ramdevi wants t-shirt 4
ORDER: Ramdevi wants t-shirt 5
Press any key to continue . . .
```

```
ORDER: Mark wants t-shirt 1
ORDER: Mark wants t-shirt 2
ORDER: Ramdevi wants t-shirt 1
ORDER: Mark wants t-shirt 3
ORDER: Ramdevi wants t-shirt 2
ORDER: Mark wants t-shirt 4
ORDER: Ramdevi wants t-shirt 3
ORDER: Mark wants t-shirt 5
ORDER: Ramdevi wants t-shirt 4
ORDER: Ramdevi wants t-shirt 5
Press any key to continue . . .
```

Race Conditions

Different order doesn't matter here

…but in a different app it might be a bug or cause data corruption



C:\Windows\system32\cmd.exe

```
ORDER: Mark wants t-shirt 1
                  shirt 2
                  shirt 3
                  t-shirt 1
                  t-shirt 2
                  shirt 4
                  t-shirt 3
                  shirt 5
                  t-shirt 4
                  t-shirt 5
                  tinue . . . _
```

C:\Windows\system32\cmd.exe

```
ORDER: Mark wants t-shirt 1
            ts t-shirt 2
            wants t-shirt 1
            ts t-shirt 3
            wants t-shirt 2
            ts t-shirt 4
            wants t-shirt 3
            ts t-shirt 5
            wants t-shirt 4
            wants t-shirt 5
            o continue . . . _
```

Concurrent Collections

✓ Will protect you from:

Internal data corruption

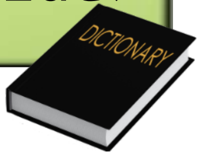
Race conditions
inside method calls

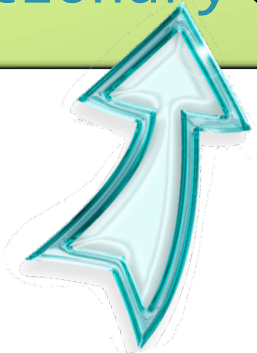
✗ Won't protect you from:

Race conditions
between method calls

What Concurrent Collections Are There?

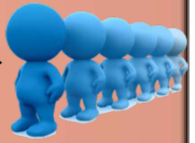
`ConcurrentDictionary<TKey, TValue>`





The only general purpose thread-safe collection!

`ConcurrentQueue<T>`



`ConcurrentStack<T>`

`ConcurrentBag<T>`

Specialised scenarios only

Using a Concurrent Dictionary

You want…

Thread-safe version of `List<T>` or `T[]`

You can use…

`ConcurrentDictionary<int, T>`

The index

You want…

Thread-safe version of `HashSet<T>`

You can use…

`ConcurrentDictionary<T, T>`

You want…

Thread-safe version of `SortedList<TKey, TValue>` or `SortedDictionary<TKey, TValue>`

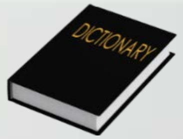
You can use…

`ConcurrentDictionary<TKey, TValue>`

(But sort before enumerating)

Concurrent Collections – The Full List…

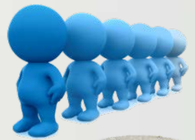
General-purpose



`ConcurrentDictionary<TKey, TValue>`

Producer-consumer

`ConcurrentQueue<T>`



`ConcurrentStack<T>`

`ConcurrentBag<T>`

`BlockingCollection<T>`

`IProducerConsumerCollection<T>`

Partitioners

`Partitioner<T>`

`OrderablePartitioner<T>`

`Partitioner`

`EnumerablePartitionerOptions`

Concurrent Collections – The Full List…

General-purpose

`ConcurrentDictionary<TKey, TV`

- Abstract base classes for partitioners

Partitioners

```
Partitioner<T>
OrderablePartitioner<T>
Partitioner
EnumerablePartitionerOptions
```

```
ConcurrentBag<T>
BlockingCollection<T>
IProducerConsumerCollection<T>
```

CODE DEMO

This slide must not appear in the recorded course

Module 1 Summary

- Concurrent collections can be invoked on multiple threads without internal data corruption
  - But don't protect you from race conditions between method calls
- For most general purposes – use `ConcurrentDictionary`
- Concurrent queue, stack and bag for producer-consumer scenarios
- Concurrent collections don't rely exclusively on blocking threads