Doing stuff at the same time in .NET 4

An introduction to parallelisation

Using the Task Parallel Library

colinmackay.co.uk @ColinMackay #dddbelfast

What to expect

- Introduction to the talk
- Overview
- Various ways of starting tasks
- Handling and Sharing Data
- Stopping Tasks
  - Handling Exceptions
  - Cancelling Tasks

Code Examples

- This talk will move quickly
- All the code examples can be found on my blog:

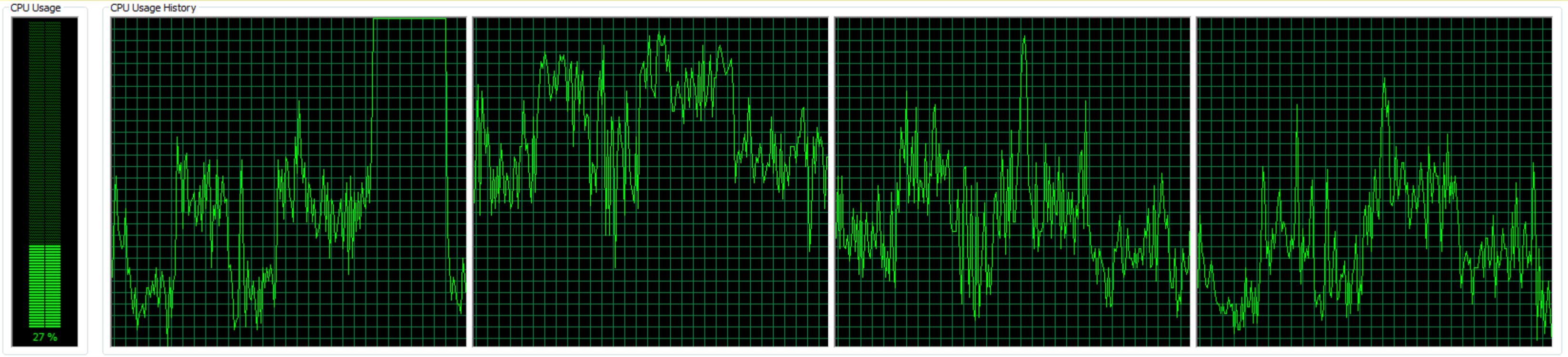
colinmackay.co.uk/blog/category/parallelisation-talk-examples/

“z” spelling also works in blog URLs

OVERVIEW

Why is Parallelisation Important?

- Multicore processors
  - Single core improvements very limited now
  - Moore’s Law requires concurrency to continue



But multithreading is hard!

- Yes – It was a real pain to work well
- Think in terms of tasks than threads
  - It makes it so much easier

So Parallel Extensions make it easy

- Easier, yes.
- Easy, no.
- Still have to consider the implications
  - Shared
    - Data
    - Resources
  - More complex than single thread
    - Bugs can be intermittent if due to clashing operations

Degrees of Parallelism

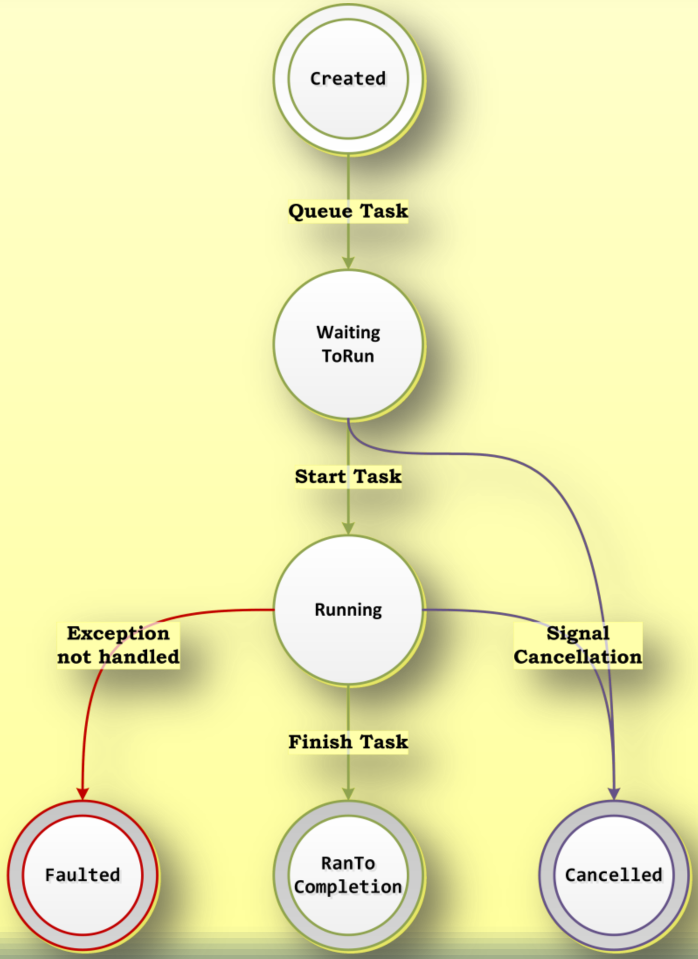
- Don’t hardcode the degree of parallelism
  - You often don’t know the hardware
  - If you do know the hardware
    - Hardware can change
  - Other software that is running may have an impact.
  - Can be useful for testing tho’
    - Simulate older hardware when set low

Tasks

- Small operations that combine in parallel to form a larger operation.
- Independent
- Granularity
  - – Too fine
    - Too much overhead
  - – Too coarse
    - Potential for cores to be idle
- Potential Parallelism

Task State Transition

- Tasks don't always start immediately
  - WaitingToRun
- Can transition directly to cancelled without running
- Faulted == Unhandled Exception



VARIOUS WAYS OF STARTING TASKS

Parallel.For

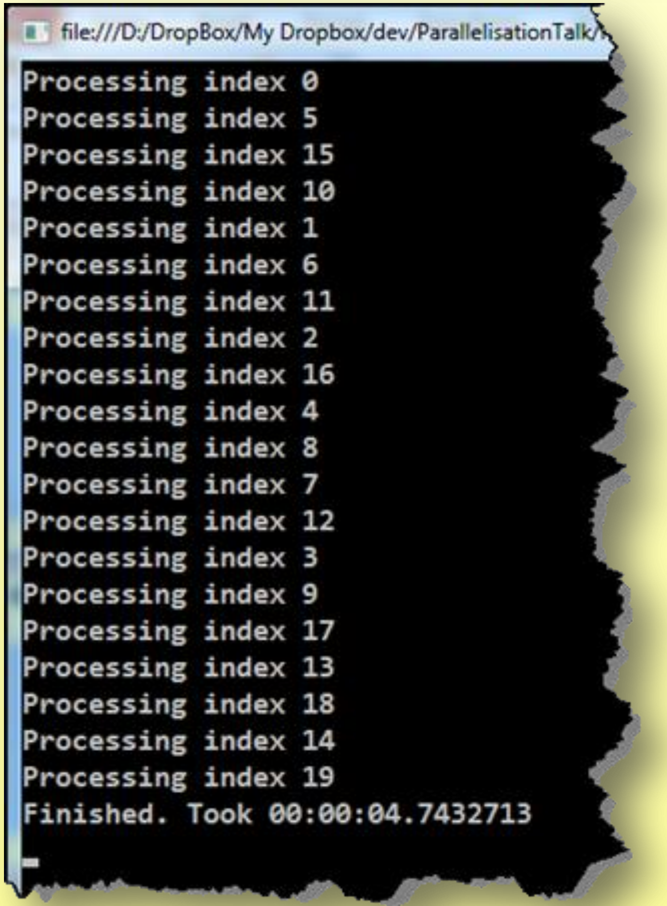
- `using System.Threading.Tasks;`

```
for (int i = 0; i < 20; i++)
    ProcessLoop(i);

Parallel.For(0, 20,
    (i) => ProcessLoop(i));
```

Example: Parallel.For

```
using System.Threading.Tasks;

Parallel.For(0, 20,
    (i) => ProcessLoop(i));
```

- Each index takes almost 1s
- Total time for 20 iterations: 5s
  - I have a 4 core processor
- Loop index is not sequential
- Cannot depend on result of previous iteration



```
Processing index 0
Processing index 5
Processing index 15
Processing index 10
Processing index 1
Processing index 6
Processing index 11
Processing index 2
Processing index 16
Processing index 4
Processing index 8
Processing index 7
Processing index 12
Processing index 3
Processing index 9
Processing index 17
Processing index 13
Processing index 18
Processing index 14
Processing index 19
Finished. Took 00:00:04.7432713
_
```

Anti-pattern parallel for

```
Task[] tasks = new Task[20];
for (int i = 0; i < 20; i++)
    tasks[i] = Task.Factory.StartNew(
        () => Console.WriteLine(
            "The loop index is {0}", i));
Task.WaitAll(tasks);
```

Refactoring Tips

- If you require
  - Step values other than one
  - Or going in reverse

Possibly got dependencies on other iterations

- If you can
  - reverse the sequential iteration
  - And have no ill effects

Possibly make parallel without issues

Parallel.ForEach

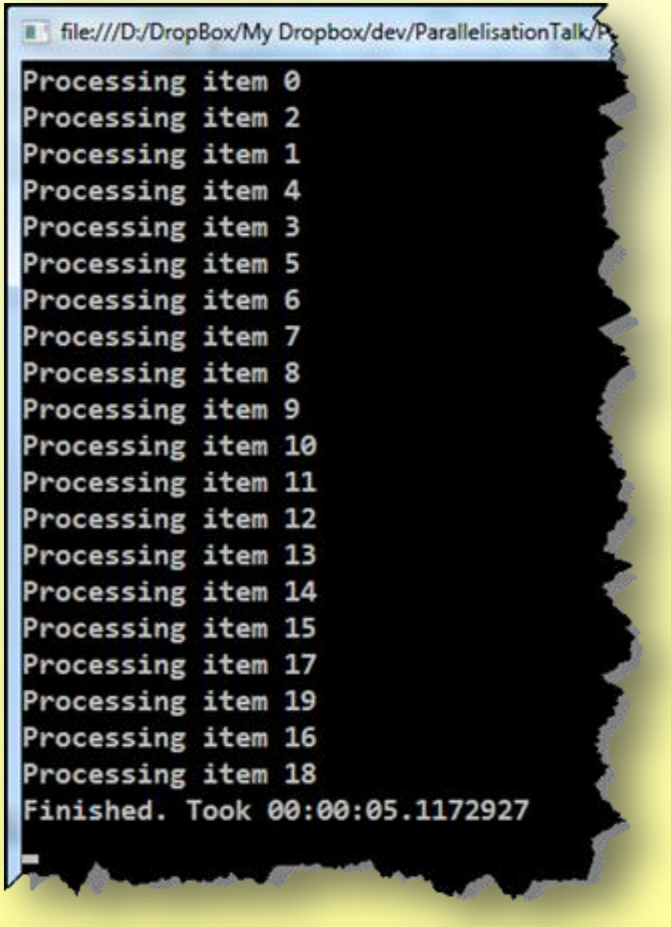
- using System.Threading.Tasks;

```
foreach(var item in items)
    ProcessLoop(item);

Parallel.ForEach(items,
    (item) => ProcessLoop(item));
```

Example: Parallel.ForEach

```
using System.Threading.Tasks;

Parallel.ForEach(items,
    (item) => ProcessLoop(item));
```

- Each item takes almost 1s
- Total time for 20 iterations: 5s
  - I have a 4 core processor
- Looping is not sequential
- Cannot depend on result of previous iteration



```
Processing item 0
Processing item 2
Processing item 1
Processing item 4
Processing item 3
Processing item 5
Processing item 6
Processing item 7
Processing item 8
Processing item 9
Processing item 10
Processing item 11
Processing item 12
Processing item 13
Processing item 14
Processing item 15
Processing item 17
Processing item 19
Processing item 16
Processing item 18
Finished. Took 00:00:05.1172927
_
```

PLINQ

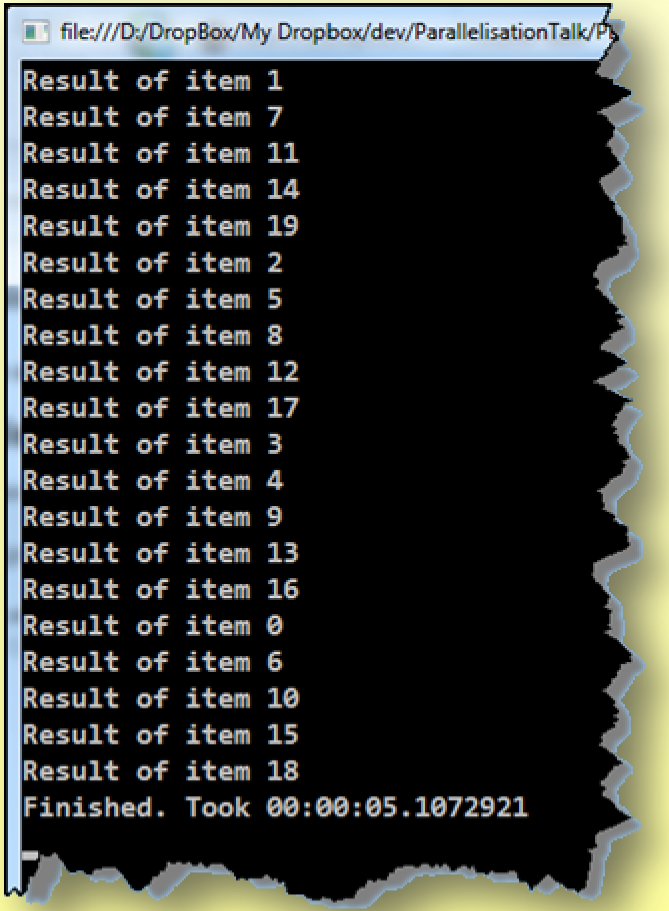
```
var results = items
    .Select(ProcessItem);
```

```
var results = items
    .AsParallel()
    .Select(ProcessItem);
```

Example: PLINQ

```
var results = items
    .AsParallel()
    .Select(ProcessItem);
```

- Each item takes almost 1s
- Total time for 20 iterations: 5s
  - I have a 4 core processor
- Processing is not sequential
- Uses a `Func<TResult, …>`
  - Instead of `Action<…>`
  - Data can be returned



```
Result of item 1
Result of item 7
Result of item 11
Result of item 14
Result of item 19
Result of item 2
Result of item 5
Result of item 8
Result of item 12
Result of item 17
Result of item 3
Result of item 4
Result of item 9
Result of item 13
Result of item 16
Result of item 0
Result of item 6
Result of item 10
Result of item 15
Result of item 18
Finished. Took 00:00:05.1072921
```

Tasks

- Don’t necessarily start immediately
  - Queued if insufficient cores are available
  - Scheduler optimises degree of concurrency
- Scales well without dependencies
  - Locking mechanisms will cause bottlenecks
  - Independent data structures work well

Parallel.Invoke

- Starts multiple tasks via delegates
  - `params Action<…>`
- Method blocks until all tasks complete

```
using System.Threading.Tasks;

Parallel.Invoke(TaskOne, TaskTwo);
```

Demo showing Parallel.Invoke

PARALLEL.INVOKE

Task.Factory.StartNew

```
Parallel.Invoke(TaskOne, TaskTwo);

                    ==

Task t1 = Task.Factory.StartNew(TaskOne);
Task t2 = Task.Factory.StartNew(TaskTwo);
Task.WaitAll(t1, t2);
```

Task.Factory.StartNew

- Schedules a Task to start
  - May start in the future
  - Depends on the Scheduler
- Overloaded allowing a lot of configuration
  - Creation options
  - Action<…> or Func<…>

Long Running Tasks

- If task time >500ms
  - Use: `TaskCreationOptions.LongRunning`
- Scheduler thinks tasks >500ms are blocked
  - Will start more tasks
  - If long running task processor intensive…
    - more tasks = more context switching.

Tasks within Tasks

- A task can generate further work
- `Task.Factory.StartNew()`
- `TaskCreationOptions.AttachToParent`
    - – Launching task won’t complete until child tasks complete

Demo showing tasks being launched within another task.

TASKS WITHIN TASKS

HANDLING AND SHARING DATA

Handling and Sharing Data

- Independent object graphs between tasks
  - Each task updates only its own object graph
- Immutable/read-only shared data
- Sharing variables require locks
  - Locks serialise access to the shared data
  - Scalability issues due to contention
    - Too many locks
    - Too many cores
  - Can be easy to accidentally use locks incorrectly
    - Problems include deadlocking.

Independent Object Graphs

- Each task has its own object graph
- The task can do what it wants with the graph
- No interference with other tasks
- No dependencies on other tasks
- No locking/serialising required

Demo showing each task performing operations in its own object graph independently of other tasks.

INDEPENDENT OBJECT GRAPHS

Locking

- Safe access to shared data or resources
- Contention an issue
  - When another thread attempts to gain access to previously locked resource.
  - Get more performance by designing for parallel
    - Don’t just add locks to sequential code then run it in parallel.

The Concurrent… Dictionary / Bag / Queue / Stack

- A number of Concurrent… classes in .NET 4.0
- Provide the locking mechanism built in
  - Reduces the possibility of mistakes.

```
using System.Collections.Concurrent;
```

ConcurrentBag

- Unordered collection
- Allows duplicates
- Enumerating creates a snapshot
  - Removals and Additions won’t affect it
- Main methods
  - Add
  - TryTake
  - TryPeek
  - Count
  - IsEmpty

Demo showing adding items to the concurrent bag in a background task while enumerating over them in the main thread.

CONCURRENT BAG

ConcurrentDictionary

- Collection of Key/Value pairs
- Main methods
  - TryAdd
  - TryUpdate
  - TryGetValue
  - TryRemove

ConcurrentDictionary: TryUpdate

```
TryUpdate(key, newValue, comparisonValue)
```

- If the comparisonValue doesn’t match existing value then update fails
  - Ensures no overwrites of other task’s updates

Demo showing a ConcurrentDictionary being used to count the number of each type of word in a document.

CONCURRENT DICTIONARY

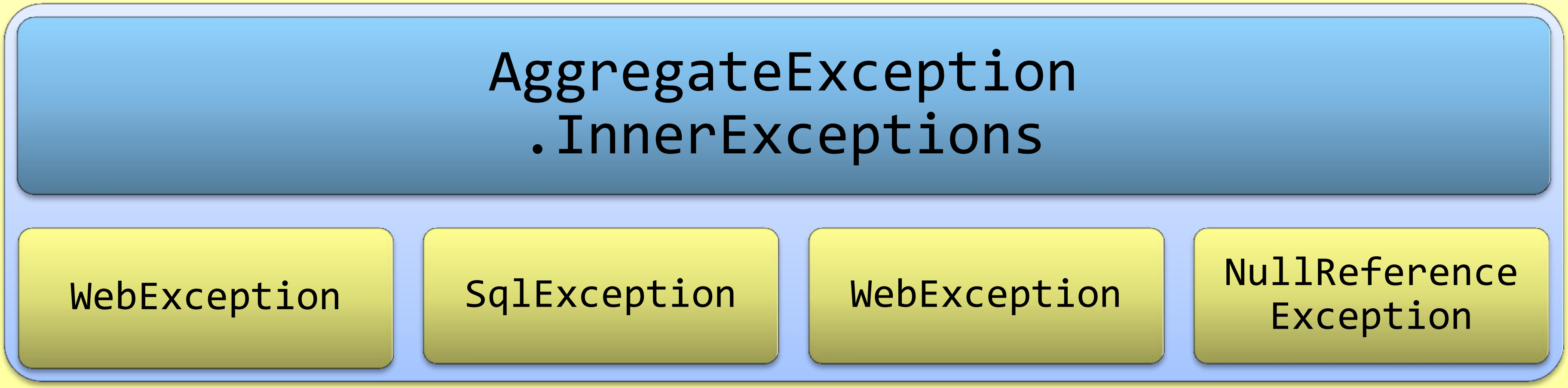
HANDLING EXCEPTIONS

Handling Exceptions

- Within a task handle as normal
- Uncaught exceptions bubble out
  - Put try/catch around WaitAll
  - Implicit WaitAll in Parallel.Invoke, Parallel.For, Parallel.ForEach & PLINQ
- Multiple tasks → Multiple exceptions
  - `AggregateException`

Aggregate Exceptions

- Has an InnerException like other exceptions
- Also has InnerExceptions
  - A ReadOnlyCollection<Exception>



AggregateException
.InnerExceptions

WebException

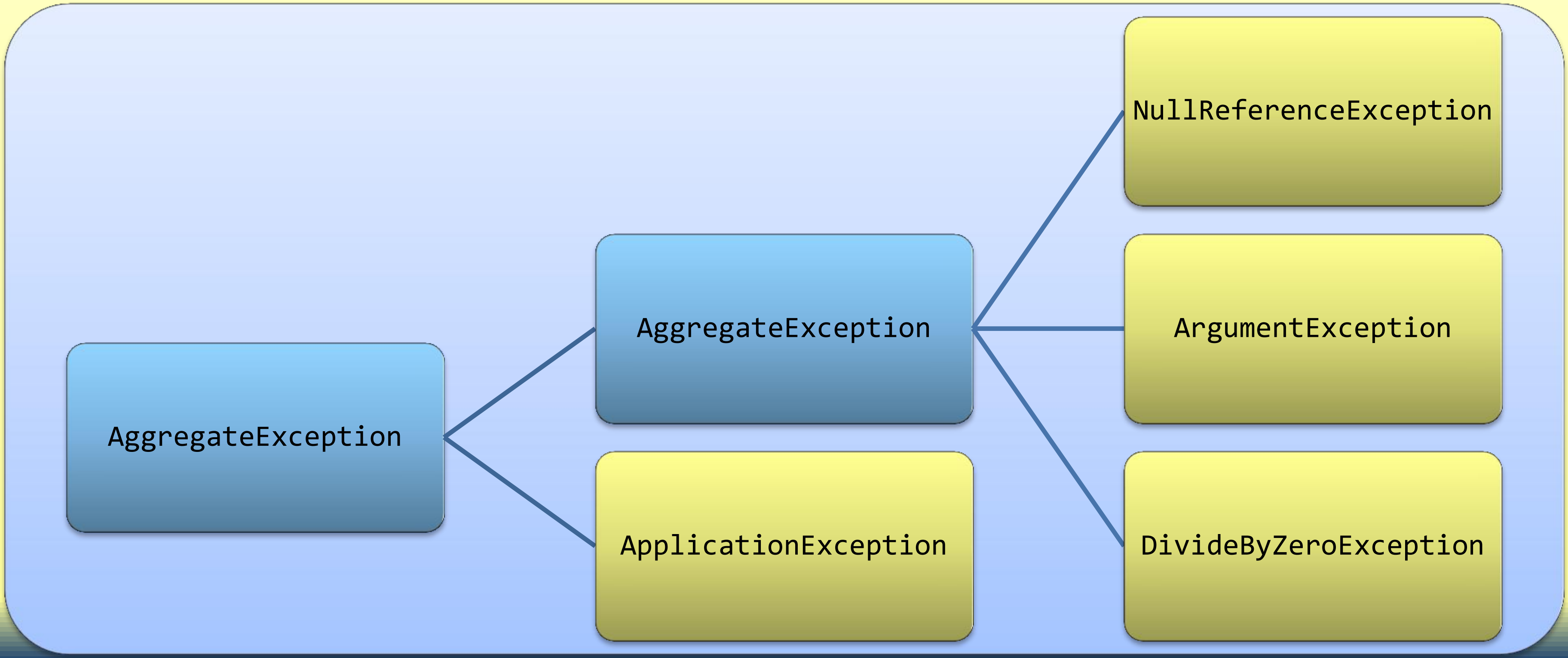
SqlException

WebException

NullReference
Exception

Aggregate Exceptions

- Tasks within Tasks can lead to
  - AggregateException within AggregateException


AggregateException

AggregateException

ApplicationException

NullReferenceException

ArgumentException

DivideByZeroException

Shows what happens when an exception is thrown inside a task.

AGGREGATE EXCEPTION DEMO

CANCELLING TASKS

Cancelling Tasks - Setup

- `CancellationTokenSource`
  - Source of the cancellation request
  - Provides the token
- `CancellationToken`
  - Passed in to each task that can be cancelled
  - Has facilities for tasks to use when cancelled.
- `Task.Factory.StartNew(… , token)`

Cancelling tasks – In the task

- Cooperative model. When appropriate:
  - `token.ThrowIfCancellationRequested();`

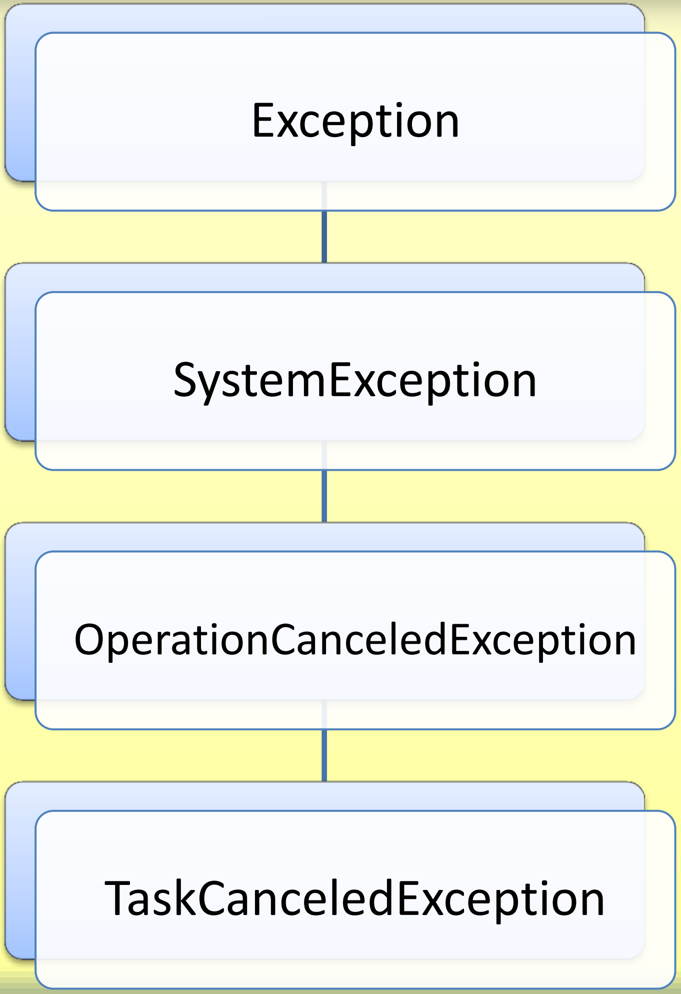
```
catch (OperationCanceledException)
{
    // Any clean up code goes here.
    throw; // Must rethrow
}
```

Cancelling Tasks: In calling code

```
try
{
    Task.WaitAll(tasks); // Exceptions thrown at this point
}
catch (AggregateException aex)
{
    aex.Handle(ex =>
    {
        TaskCanceledException tcex = ex as TaskCanceledException;
        if (tcex != null)
        {
            // Do stuff to handle the cancellation of a task
            return true;
        }
        return false;
    });
}
```

Cancelling Tasks Gotcha

- Exception inside the task is: **`OperationCanceledException`**
- Exception outside the task in the `AggregateException` is: **`TaskCanceledException`**
  - has no stack trace!
  - Has a reference to the Task object
  - Inherits from `OperationCanceledException`



Exception

SystemException

OperationCanceledException

TaskCanceledException

Shows what happens when a task is cancelled.

CANCELLATION DEMO

Further Reading

- colinmackay.co.uk/blog/tag/parallelisation/
- colinmackay.co.uk/blog/category/parallelisation-talk-examples/

“z” spelling also works in blog URLs

Parallelisation

Question Time

colinmackay.co.uk/blog/category/parallelisation-talk-examples/

Parallelisation

Fill in feedback

colinmackay.co.uk/blog/category/parallelisation-talk-examples/