

Module 07:

"Tasks in TPL"



TEKNOLOGISK
INSTITUT

Agenda

- ▶ **Task Parallelism in TPL**
- ▶ Introducing Tasks
- ▶ Cancelling Tasks and Parallel Operations
- ▶ Composing Tasks
- ▶ Tasks and Exceptions

What is Task Parallelism?



"IMG_8606.jpg" by [adaenn](#) is licensed under [CC BY-NC 2.0](#)

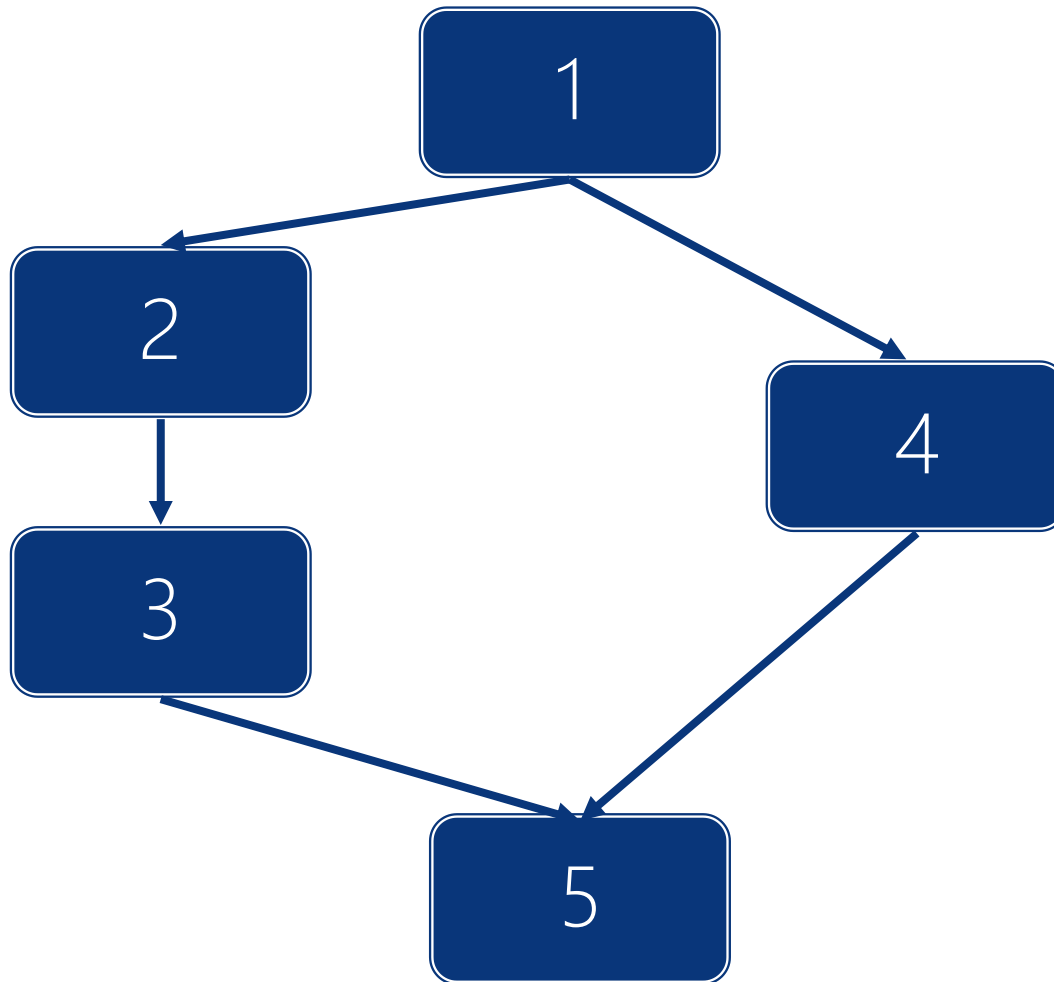


"Simmer" by [harry harris](#) is licensed under [CC BY-NC-SA 2.0](#)

Absence of Task Parallelism



Task Parallelism



Agenda

- ▶ Task Parallelism in TPL
- ▶ **Introducing Tasks**
- ▶ Cancelling Tasks and Parallel Operations
- ▶ Composing Tasks
- ▶ Tasks and Exceptions

Creating Tasks

- ▶ The Task class captures a unit of computation
- ▶ Initialized from constructor using a computation described by
 - Action delegate
 - Anonymous method
 - Lambda expression (usually preferred)

```
Task task = new Task( () =>  
    Console.WriteLine( "Hello World from Task Parallel Library" )  
);
```

- ▶ Note: Does not run automatically when created!

Task Execution

- ▶ Three approaches to starting tasks
 - Create **Task** object and invoke **Task.Start()**
 - Use **Task.Factory.StartNew()** static
 - Use **Task.Run()** static

```
Task task = Task.Factory.StartNew( () =>
{
    for ( int i = 1 ; i < 100 ; i += 2 )
    {
        Console.WriteLine( "\t" + i );
    }
});
```

- ▶ Usually one of the last two options is employed

Waiting for Task Completion

► Tasks can be awaited

- `Task.Wait()`
- `Task.WaitAny()` static
- `Task.WaitAll()` static

```
Task task1 = ...;  
Task task2 = ...;  
Task task3 = ...;  
  
task1.Wait();  
  
Task.WaitAny( task1, task2, task3 );  
  
Task.WaitAll( task1, task2, task3 );
```

Tasks with Results

- ▶ **Task<T>**
 - captures a task returning a result of type **T**
- ▶ **Task.Run<T>()** and **Task.StartNew<T>()** also exist

```
Task<DateTime> t = Task.Run<DateTime>( () => DateTime.Now );  
Console.WriteLine( t.Result );
```

- ▶ Result can be explicitly retrieved via **Task.Result**
 - Note: This property blocks when task is not yet completed!

Agenda

- ▶ Task Parallelism in TPL
- ▶ Introducing Tasks
- ▶ **Cancelling Tasks and Parallel Operations**
- ▶ Composing Tasks
- ▶ Tasks and Exceptions

Cancelling Tasks

- ▶ Running tasks can be requested cancelled
 - Signal token created by **CancellationTokenSource** class
 - Other code signal token supplied to task
- ▶ Task method then
 - Checks if cancellation is requested
 - Throws **OperationCanceledException** to accept cancellation

```
task = Task.Factory.StartNew( () =>
{
    ...
    if( token.IsCancellationRequested )
    {
        throw new OperationCanceledException( token );
    }
}
```

- ▶ Check task running status via **Task.Status**

Cancelling Parallel Operations

- ▶ All operations in TPL are cancelled the same way
 - Task
 - The Parallel Class
 - Parallel LINQ

```
CancellationTokenSource cts = new CancellationTokenSource();  
...  
var even = numbers  
    .AsParallel()  
    .WithCancellation( cts.Token )  
    .Where(Filter)  
    ;
```

Agenda

- ▶ Task Parallelism in TPL
- ▶ Introducing Tasks
- ▶ Cancelling Tasks and Parallel Operations
- ▶ **Composing Tasks**
- ▶ Tasks and Exceptions

Continuation Tasks

- ▶ Tasks can be combined using `Task.ContinueWith()`

```
Task<DateTime> t1 = new Task<DateTime>( () =>
    DateTime.Now );
Task<string> t2 = t1.ContinueWith( previous =>
    $"The time is {previous.Result}!" );

t1.Start();
Console.WriteLine( t2.Result );
```

- ▶ When t1 completes, the *continuation task* executes

TaskContinuationOptions

- ▶ The behavior of `Task.ContinueWith()` and `Task<T>.ContinueWith()` can be refined
- ▶ `TaskContinuationOptions` enumeration supplied in overloads
 - `None`
 - `OnlyOnCanceled`
 - `OnlyOnFaulted`
 - `OnlyOnRanToCompletion`
 - `NotOnCanceled`
 - `NotOnFaulted`
 - `NotOnRanToCompletion`
 - ...

TaskCreationOptions

- ▶ **TaskCreationOptions** allows the creation of child tasks
 - Allows distinguishing between nested and child tasks
- ▶ **TaskCreationOptions** enumeration supplied in overloads
 - None
 - PreferFairness
 - LongRunning
 - **AttachedToParent**
 - DenyChildAttach
 - HideScheduler
 - RunContinuationsAsynchronously

Agenda

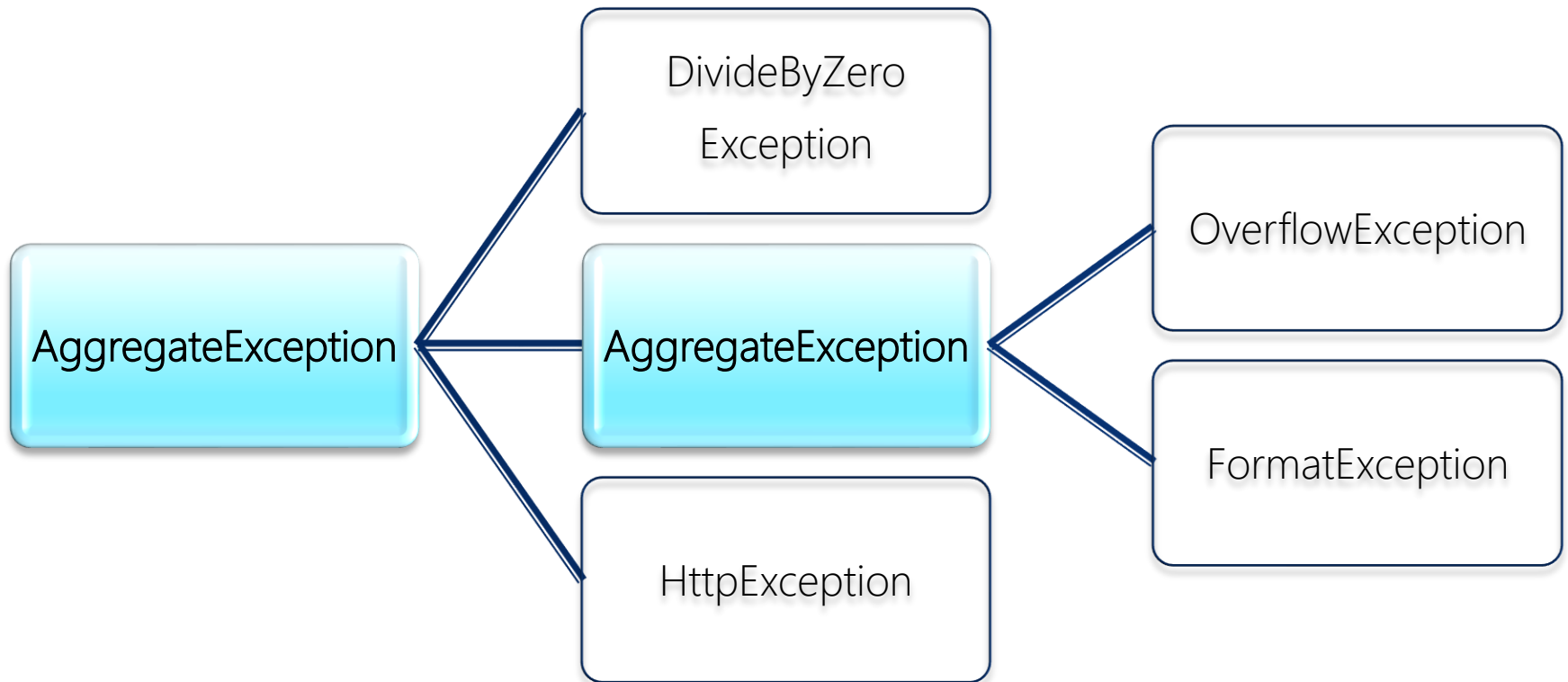
- ▶ Task Parallelism in TPL
- ▶ Introducing Tasks
- ▶ Cancelling Tasks and Parallel Operations
- ▶ Composing Tasks
- ▶ **Tasks and Exceptions**

Task Exceptions

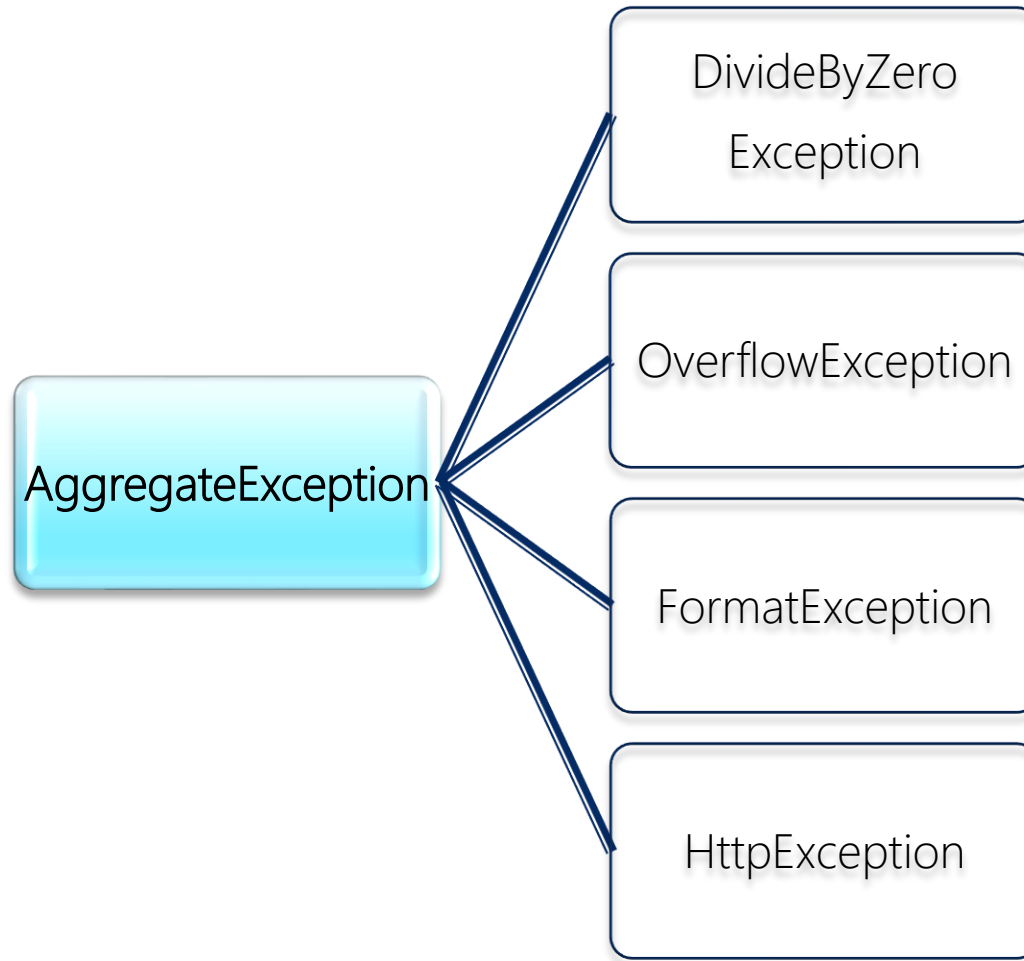
- ▶ Task exceptions are thrown when
 - Waiting for task
 - Getting result for task
- ▶ **AggregateException** instances are thrown everywhere in TPL
 - Consists of a number of inner exceptions
- **Flatten()**
is important!

```
try
{
    t.Wait();
}
catch ( AggregateException ae )
{
    foreach( Exception e in ae.InnerExceptions )
    {
        Console.WriteLine( e.Message );
    }
}
```

Before Flattening



After Flattening



Agenda

- ▶ Task Parallelism in TPL
- ▶ Introducing Tasks
- ▶ Cancelling Tasks and Parallel Operations
- ▶ Composing Tasks
- ▶ Tasks and Exceptions



WINCUBATE

Jesper Gulmann Henriksen

PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31

Email : jgh@wincubate.net

WWW : <http://www.wincubate.net>

Ringgårdsvej 4A

8270 Højbjerg

Denmark