Module 6

"Threads and Asynchrony in WPF"

TEKNOLOGISK
INSTITUT

Agenda

- **Dispatcher**
- Data Binding
- Tasks, Async, and Await
- SynchronizationContext

TEKNOLOGISK INSTITUT

UI and Threads

- Windows UI Context
  - Notion of "Main" thread
- Message Pump
- WinForms ~ ISynchronizeInvoke
- WPF ~ Dispatcher
- Mantra:
  - "Keep Working Threads Away From UI"

WPF Class Hierarchy

- object
  - **DispatcherObject** — Access only on creating thread
    - DependencyObject
      - Freezable
      - Visual
        - UIElement — Routed events, layout, focus, …
          - FrameworkElement — Styling, data binding, …
            - Control — **Foreground**, **Background**, …
      - Visual3D
        - UIElement3D
      - ContentElement
        - FrameworkContentElement

The Dispatcher

- Any operation on **`DispatcherObject`** must happen on the UI thread
  - **`InvalidOperationException`**
- Use **`DispatcherObject.Dispatcher`** property
  - **`Invoke()`** – Synchronous
  - **`BeginInvoke()`** – Asynchronous
- WPF has two built-in "main threads"
  - Main thread
  - Render thread

DispatcherPriority

- Priority is captured by **DispatcherPriority** enumeration
  - http://msdn.microsoft.com/en-us/library/system.windows.threading.dispatcherpriority.aspx

  - **Send** Highest (= immediately)
  - **Normal**
  - **DataBind**
  - **Render**
  - …
  - Background
  - …
  - ApplicationIdle
  - SystemIdle Lowest

DispatcherTimer

- Many different timers
  - ~~System.Timers.Timer~~
  - ~~System.Threading.Timer~~

  - System.Windows.Threading.DispatcherTimer
    - Tick event
    - Interval
    - Start()
    - Stop()

TEKNOLOGISK INSTITUT

Multiple UI Threads

- More dispatcher threads can be created
- Multiple UI threads can be created
  - `Dispatcher.Run()` on separate thread
- Be careful!
  - **`Application.*`** is now misleading and dangerous!
  - Application.Windows
  - Application.Dispatcher

Agenda

- Dispatcher
- **Data Binding**
- Tasks, Async, and Await
- SynchronizationContext

Notifications and Threads

- Adding elements to **`ObservableCollection`** by other threads
  - Not directly possible
  - Needed ugly dispatching!

- WPF 4.5 adds easy-to-use Collection Synchronization
  - Provide lock for the collection
  - Enable collection synchronization
  - Update **`IEnumerable`** from any thread

```
BindingOperations.EnableCollectionSynchronization(
   _participants,   // collection
   _syncObject      // lock object
);
```

Asynchronous Data Binding

- Data binding can be evaluated asynchronously
  - Binding.IsAsync

- Is often combined with PriorityBinding

```
<PriorityBinding FallbackValue="N/A">
    <Binding Path="Slowest" IsAsync="True"/>
    <Binding Path="Slow" IsAsync="True"/>
    <Binding Path="Normal" IsAsync="True"/>
    <Binding Path="Fast" IsAsync="True"/>
    <Binding Path="Fastest" />
</PriorityBinding>
```

- Beware: Asynchronous bindings can be a sign of poor design

Agenda

- Dispatcher
- Data Binding
- **Tasks, Async, and Await**
- SynchronizationContext

Task Parallel Library

- Task Parallel Library (TPL)
  - Was introduced in .NET 4.0
  - Enhanced in .NET 4.5
    - Special keywords are included in C# 5.0
- Features
  - **Task Parallelism**
  - Data Parallelism
  - Parallel LINQ
  - Thread-safe collections
- Emerging trends leverage parallelism! Also .NET!

Creating Tasks

- The **`Task`** class captures a unit of asynchronous operation

- 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 is blocks when task is not yet completed!

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`**

Combining Tasks

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

```
Task<DateTime> t1 = new Task<DateTime>( () =>
   DateTime.Now );
Task<string> t2 = t1.ContinueWith( previous =>
   string.Format("The time is {0}!", previous.Result ) );

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

- Combinators include
  - `Task.WhenAll()` Completes when all tasks have completed
  - `Task.WhenAny()` Completes when any of the tasks completes
  - `Task.Delay()` Completes after a specified time span

- `TaskCreationOptions` allows the creation of child tasks

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`
  - …

Task Exceptions

- Task exceptions are thrown when
  - Waiting for task
  - Getting result for task
- `AggregateException` instances are thrown
  - Consists of a number of inner exceptions
  - `Flatten()`

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

C# 5.0 **`await`** Operator

- C# 5.0 introduces **`await`** keyword for methods returning `Task` or `Task<T>`
  - Yields control until awaited task completes
  - Results gets returned

- Allows you to program just like for synchronous programming…!

```
WebClient client = new WebClient();
string result = await client.DownloadStringTaskAsync( ... );

Console.WriteLine( result );
```

- Really complex control flow under the hood is made stunningly simple by compiler

C# 5.0 **async** Modifier

- C# 5.0 introduces **async** keyword
  - Marks method or lambda as asynchronous
  - Note: Methods making use of **await** must be marked "**async**"

- You can now easily define your own asynchronous methods

```
async static void DoStuff()
{
   // ...

   string result = await client.DownloadStringTaskAsync( ... );

   // ...
}
```

- Can create async methods returning **void**, **Task**, or **Task<T>**

TEKNOLOGISK INSTITUT

Exceptions Thrown by Tasks and Awaitable Methods

- Observe and catch exceptions "as usual" when awaiting tasks

```
try
{
   string data = await client.DownloadStringTaskAsync( ... );
}
catch ( WebException ex ) { ... }
```

- Subscribe to unobserved exceptions through the `TaskScheduler.UnobservedTaskException` event

```
TaskScheduler.UnobservedTaskException +=
   ( object s, UnobservedTaskExceptionEventArgs ute ) => {
      foreach( Exception e in ute.Exception.InnerExceptions )
      {
          ...
      };
```

Dispatcher vs. Task

- The **async** and **await** keywords in C# mix perfectly with WPF

- WPF 4.5 also adds many new **Dispatcher** methods
  - **Dispatcher.Invoke<T>()**
  - **Dispatcher.InvokeAsync()**
  - **Dispatcher.InvokeAsync<T>()**
- These are basically just rehashings of **Dispatcher.BeginInvoke()**
  - Can return values as well

```
await Dispatcher.InvokeAsync(
   () => txtResult.Text = DateTime.Now.ToString()
);
...
string old = await Dispatcher.InvokeAsync<string>(
   () => txtResult.Text
);
```

Agenda

- Dispatcher
- Data Binding
- Tasks, Async, and Await
- SynchronizationContext

What is a SynchronizationContext?

- Context handling synchronization of (a)synchronous operations
  - In general a many-to-many relationship with threads

```
public class SynchronizationContext
{
   public virtual void OperationCompleted() { ... }
   public virtual void OperationStarted() { ... }
   public virtual void Post(SendOrPostCallback d, object state)
   {
      // Perform operation asynchronously
   }
   public virtual void Send(SendOrPostCallback d, object state)
   {
      // Perform operation synchronously
   }
}
```

Built-in SynchronizationContexts

- **`WindowsFormsSynchronizationContext`**
  - Executes on a specific UI thread
  - Executes in the order they were queued.
- **`DispatcherSynchronizationContext`**
  - Queues delegates to a specific UI thread with `Normal` priority.
  - Executes in the order they were queued
  - Installed as current context by `Dispatcher.Run()`
- Default (Thread Pool) **`SynchronizationContext`**
  - if a thread's current Synchronization Context is null, then it implicitly has this default Synchronization Context.
  - Queues its asynchronous delegates to the Thread Pool but executes its synchronous delegates directly on the calling thread.

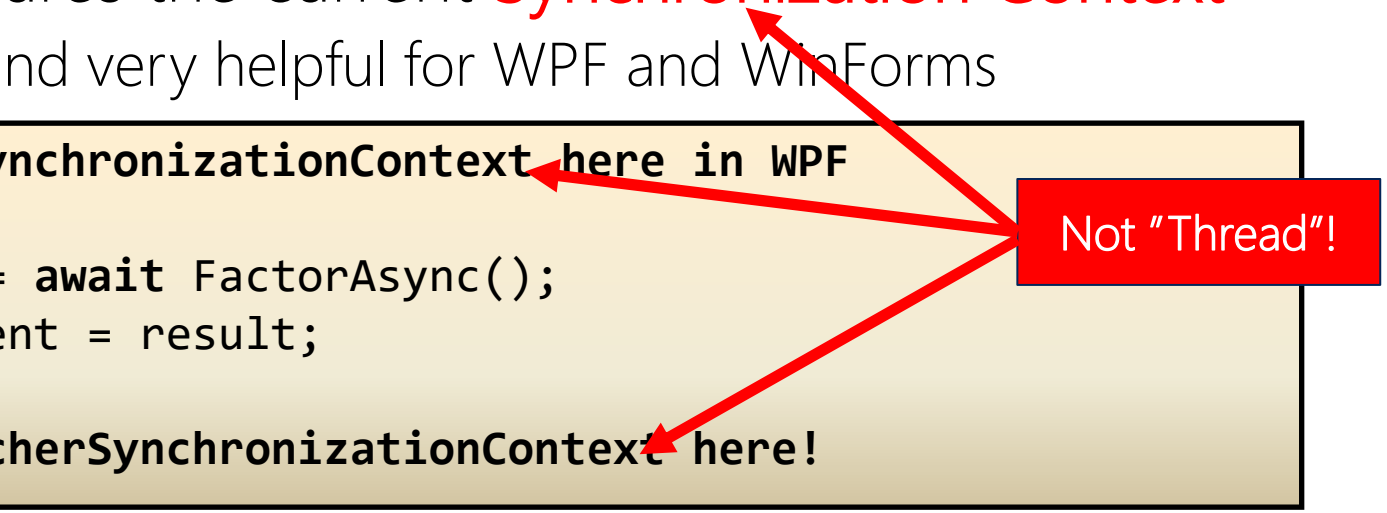
Await and SynchronizationContext

- Await captures the current Synchronization Context
  - Essential and very helpful for WPF and WinForms

```
// DispatcherSynchronizationContext here in WPF

string result = await FactorAsync();
lblResult.Content = result;

// Also DispatcherSynchronizationContext here!
```



ConfigureAwait()

- By default execution continues on the current Synchronization Context after **`await`**
- Optionally, this requirement can be manually relaxed by **`Task.ConfigureAwait(false)`**

```
// DispatcherSynchronizationContext here in WPF

string result = await FactorAsync().ConfigureAwait( false );
lblResult.Content = result;

// Not DispatcherSynchronizationContext here!
```

Summary

- Dispatcher
- Data Binding
- Tasks, Async, and Await
- SynchronizationContext


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