Module 6

”Threads and Asynchrony in WPF [Foundation]”

TEKNOLOGISK
INSTITUT

Agenda

- **Dispatcher**
- Tasks, Async, and Await in WPF
- Synchronization Context

UI and Threads

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

Image by victor408 is licensed under CC BY-NC 2.0

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 ”emulates” two built-in main threads
  - Main thread
  - Render thread

DispatcherPriority

- Priority is captured by **`DispatcherPriority`** enumeration
  - **`Send`** Highest (= immediately)
  - **`Normal`**
  - **`DataBind`**
  - **`Render`**
  - …
  - `Background`
  - …
  - `ApplicationIdle`
  - `SystemIdle` Lowest
- Best practice
  - Always make this Normal (unless you have a very good reason not to!)

DispatcherTimer

- We have previously covered two threading timers:
  - System.Timers.Timer ~ Thread Pool
  - System.Threading.Timer ~ Thread Pool

- But… Perfectly suited for WPF UI:
  - System.Windows.Threading.DispatcherTimer ~ Dispatcher
    - Tick event
    - Interval
    - Start()
    - Stop()

TEKNOLOGISK
INSTITUT

Agenda

- Dispatcher
- **Tasks, Async, and Await in WPF**
- Synchronization Context

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!

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>**

Best Practices for Task Methods

- Microsoft recommends that the name of methods returning `Task` or `Task<T>` should be postfixed with …`Async`
  - Regardless of whether it is marked with async modifier…!

```
async Task<string> DoStuffAsync()
{
   // ...

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

   return result;
}
```

```
Task<string> GetSimpleAsync()
{
   return Task.CompletedTask; // <-- We will see this later
}
```

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 ) { ... }
```

- Note that
  - `Task.WaitXxx()` throws an `AggregateException`
  - `Task.Result` throws an `AggregateException`
  - Awaiting a `Task` throws exceptions "as usual", however!

Agenda

- Dispatcher
- Tasks, Async, and Await in WPF
- Synchronization Context

TEKNOLOGISK
INSTITUT

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!
```

Not "Thread"!

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!
```

Dispatcher vs. Task

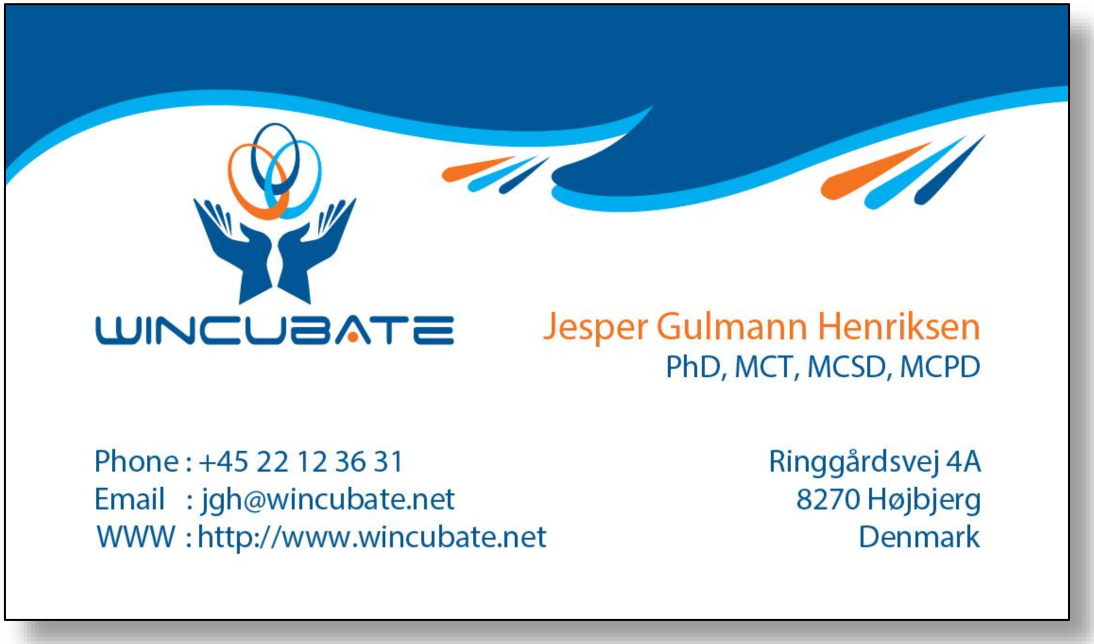
- The **`async`** and **`await`** keywords in C# mix perfectly with WPF

- 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
);
```

Summary

- Dispatcher
- Tasks, Async, and Await in WPF
- Synchronization Context



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