Threading (cont.)

Cpt S 321

Washington State University

Passing information to a thread

- Easiest way to do it is to use lambda expressions:

```
static void Main()
{
    Thread t = new Thread ( () => Print ("Hello from t!") );
    t.Start();
}
static void Print (string message) { Console.WriteLine (message); }
```

But we can also use Parameterized ThreadStart Delegate

```
public class Work {
public static void Main() {

// Start a thread that calls a parameterized static method.
  Thread newThread = new Thread(Work.DoWork);
  newThread.Start(42);

// Start a thread that calls a parameterized instance method.
  Work w = new Work();
  newThread = new Thread(w.DoMoreWork);
  newThread.Start("The answer.");
}
  public static void DoWork(object data) { // static method
    Console.WriteLine("Static thread procedure. Data='{0}'", data);
  }
  public void DoMoreWork(object data) { // instance method
    Console.WriteLine("Instance thread procedure. Data='{0}'", data);
  }
}
```

Multi-statement lambda expressions

- Actually, for simple implementations you can also wrap the method's implementation in a multi-statement lambda expression:

```
new Thread (() =>
{
    Console.WriteLine ("I'm running on another thread!");
    Console.WriteLine ("This is so easy!");
}).Start();
```

Be careful!

- Be careful about accidentally modifying *captured variables* after starting the thread:

```
for (int i = 0; i < 10; i++)
    new Thread (() => Console.Write (i)).Start();
```

- The output is nondeterministic! A typical result: 0223557799

- The **i variable** refers to the *same* memory location throughout the loop's lifetime. Therefore, each thread calls Console.Write on a variable whose value may change as it is running!

Use temporary variables to solve the problem

```
for (int i = 0; i < 10; i++) {
    int temp = i;
    new Thread (() => Console.Write (temp)).Start();
}
```

- The variable **temp** is *local* to each loop iteration. Therefore, each thread captures a different memory location.

Threads and Exceptions

```
public static void Main() {
   try {
      new Thread (Go).Start();
   }
   catch (Exception ex) {
      // We'll never get here!
      // Remember: each thread has its own independent execution path!
      Console.WriteLine ("Exception!");
   }
}
static void Go() { throw null; } // Throws a NullReferenceException
```

Threads and exceptions - solution

You need an exception handler on all thread entry methods of your application

```
public static void Main() {
        new Thread (Go).Start();
}
static void Go() {
   try {
      ...
      throw null; // The exception will get caught below
      ...
   }
   catch (Exception ex) {
      ...
   }
}
```

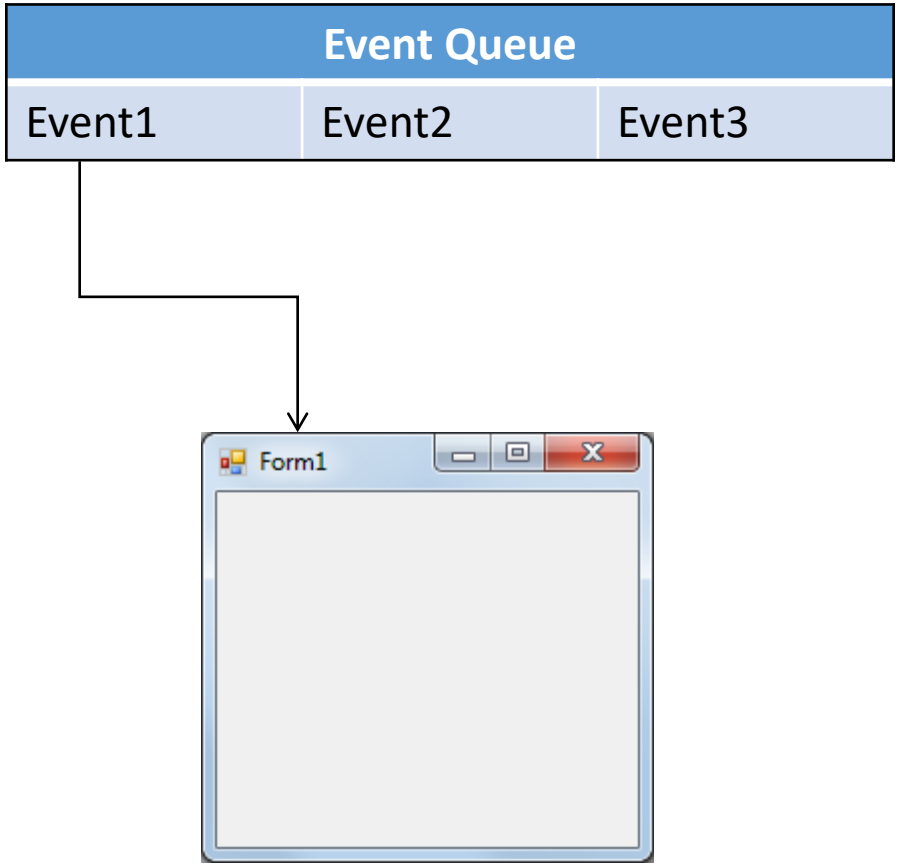
Other useful information

- Priority: how much execution time it gets relative to other active threads in the operating system

- Foreground versus Background threads: *Foreground* threads keep the application alive for as long as any one of them is running. Once all foreground threads finish, the application ends, and any background threads still running abruptly terminate.

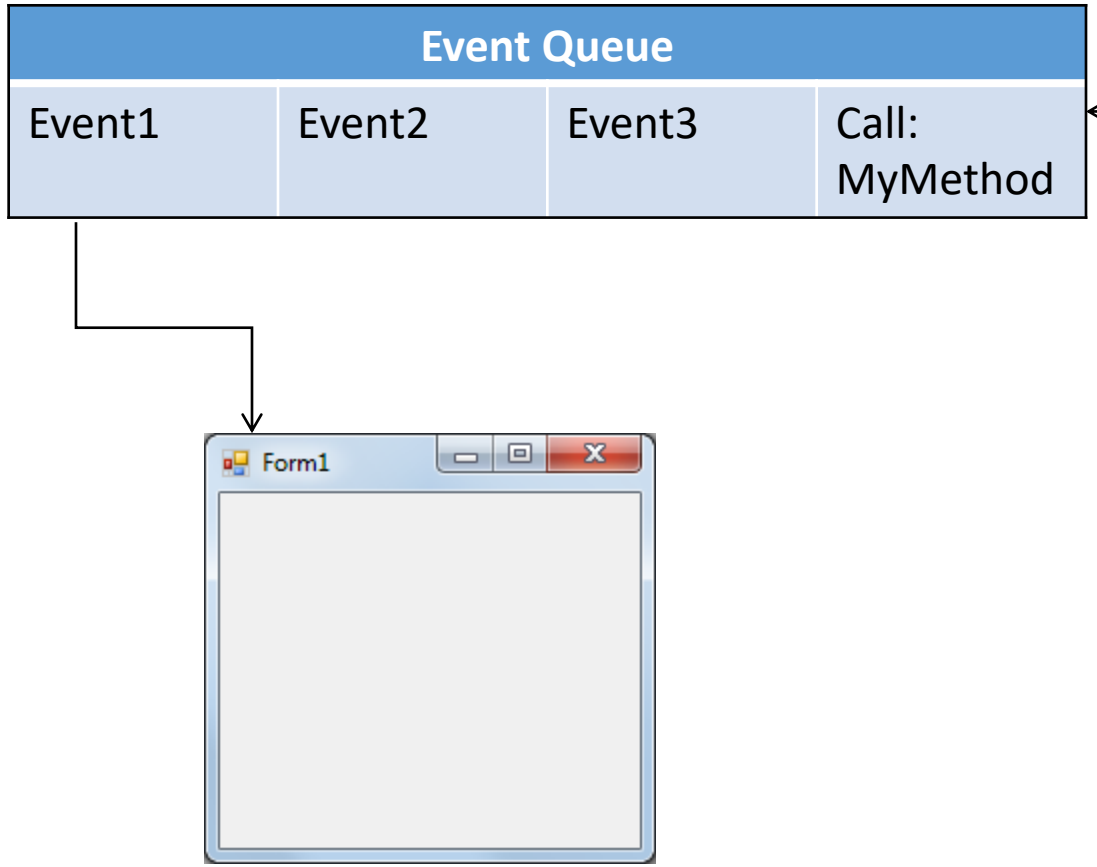
Threading in WinForm applications

- We want to avoid that our application is unresponsive due to a long/computation intensive task:
  - Create and start a “worker” thread for that task
  - The worker thread needs to update the UI when the task is complete. We do this by forwarding the request to the UI thread:
    - In Win Form applications we do this by calling BeginInvoke or Invoke on the control and passing a reference to a method that we want to execute
    - The message will be added to the UI thread’s *message queue*



Recall how GUI applications work:

```
while (!quit)
{
    WaitForMsgInQueue();
    ProcessMsgsInQueue();
}
```



Another thread calls:
Form1.BeginInvoke(MyMethod)

BeginInvoke versus Invoke

- **Invoke** blocks the worker thread until the message has been processed. Useful when you need a return value back.

- **BeginInvoke** does not block the working thread. Preferable if you don’t need a return value as it prevents possible deadlocks.

Threading in Avalonia

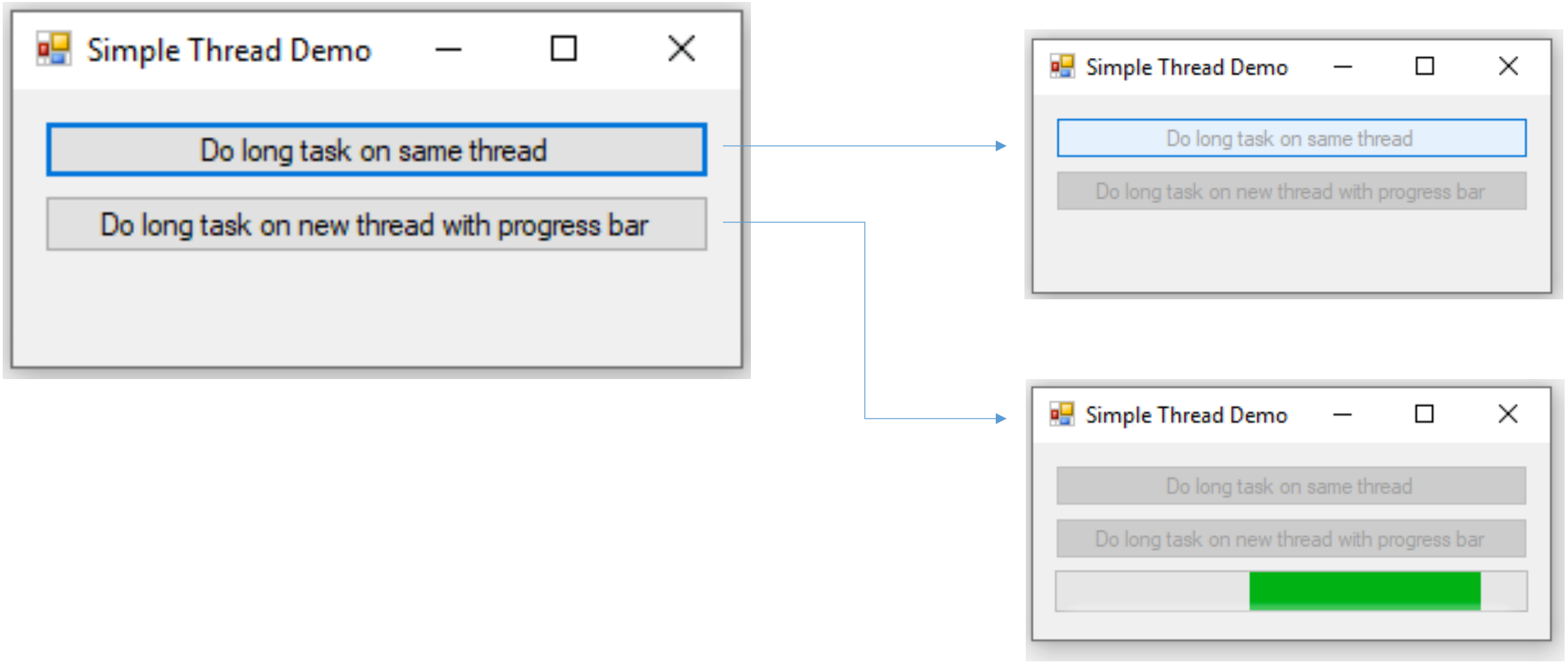
- Similarities with WinForms:
  - Create and start a “worker” thread for that task
  - The worker thread updates the UI when the task is complete (by forwarding the request to the UI thread).
  - The message will be added to the UI thread’s message queue
- In Avalonia:
  - We access the UI thread through a Dispatcher (usually have only one – the one that handles the UI thread): **Dispatcher.UIThread**
  - To run a job on the UI thread, we call InvokeAsync (when you need to wait for the result) or Post (when you don’t need a result) on the UI thread and passing a reference to a method that we want to execute and a priority.

Typical Scenario - we will also use it for the coding demo

- We want to do some slow task (can simulate just by waiting for a period of time – 10s) and have the UI show something that lets the user know that the application is working on the task and not frozen.

1. **// On the UI thread, in response to a button or menu-item click:**
   - **Show that the application received the request and is working on it**
   - **Create new thread – the worker thread that will do the actual task**
   - **"Wait" for notification from thread that it is finished**
2. // On newly created thread
   - Do the computationally intensive task
   - Notify UI thread when we are done

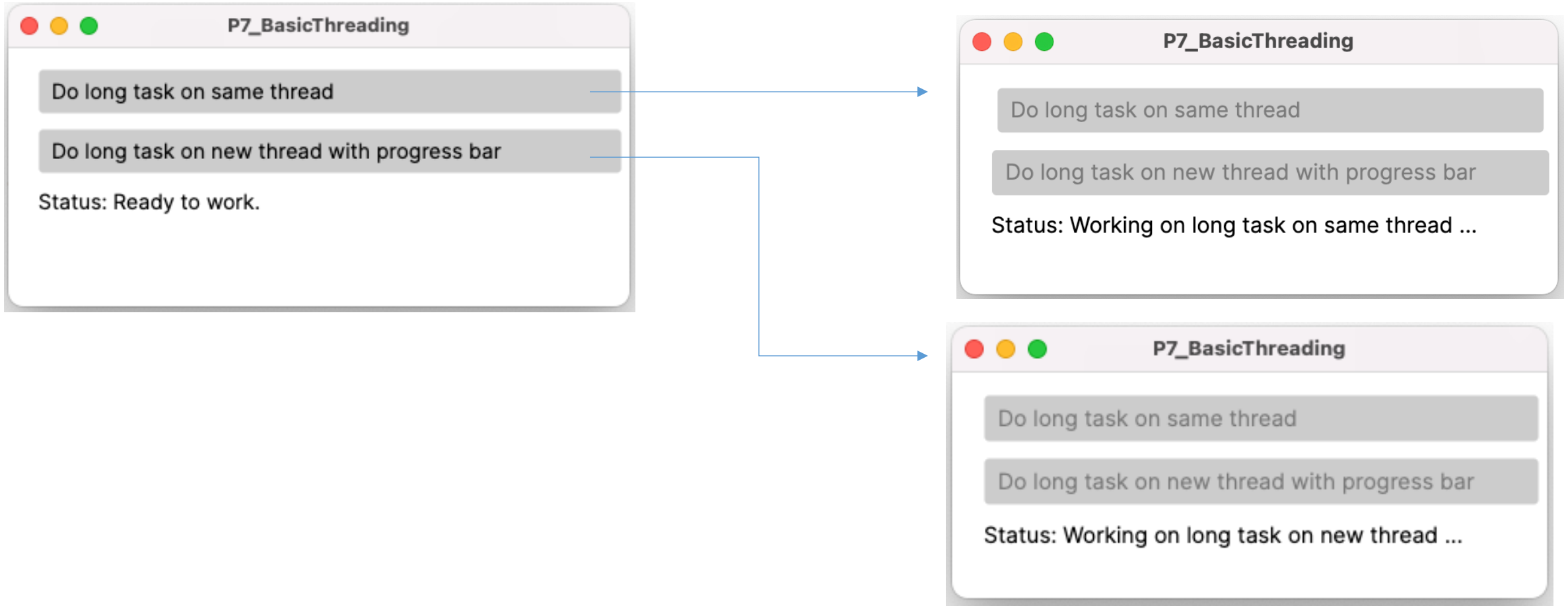
Coding Demo: Let’s do this - WinForm



Steps - WinForm

1. You can use a StackPanel with two Buttons and a TextBlock
2. Implement a ThreadCompleted() method which reinitializes the UI (i.e., enables the buttons and makes the progress bar invisible)
3. Implement DoTask(TimeSpan howLong) as follows:
   - Wait for howLong
   - Notify the UI thread that the task is complete by passing a reference to ThreadCompleted: new Action(ThreadCompleted)
4. Create btnTaskOnUIThread_Click(object sender, EventArgs e)
   - Disable the buttons and make the progress bar visible
   - Call do task with 10s
5. Create btnTaskOnNewThread_Click(object sender, EventArgs e)
   - Disable the buttons and make the progress bar visible
   - Create and start a new thread with DoTask for 10s

Coding Demo: Let’s do this - Avalonia



Steps - Avalonia

1. Use the graphical designer to create the form as shown on the previous slide.
2. Implement a ThreadCompleted() method which reinitializes the UI, i.e., enables the buttons and resets the text, e.g.:

```
Dispatcher.UIThread.Post(() => this.FindControl<Button>("ButtonSameThread").IsEnabled = true);
```

3. Implement DoTask(TimeSpan howLong) as follows:
   - Wait for howLong
   - Notify the UI thread that the task is complete by passing a reference to ThreadCompleted:

```
Dispatcher.UIThread.Post(() => DoTask(new TimeSpan(0, 0, 10)), DispatcherPriority.ContextIdle);
```

4. Create ButtonSameThreadOnClickCommand(object? sender, RoutedEventArgs routedEventArgs) and bind it to the first Button: `<Button Name="ButtonSameThread" Click="ButtonSameThreadOnClickCommand"`
   - Disable the buttons and show a message that the application is working
   - Call do task with 10s
5. Create ButtonNewThreadOnClickCommand (similar to the previous method)
   - Disable the buttons and show a message that the application is working
   - Create and start a new thread with DoTask for 10s

Further reading

- Check the concurrent collections already in .NET