Microsoft

Async Clinic

Lucian Wischik, Stephen Toub
Visual Studio
Microsoft

Agenda

The What & Why of Async

Async Control Flow

Everything as a Task

Library Thinking

Common Problems

...but really whatever you want to talk about...

The What & Why of Async

What does asynchrony really mean?

Synchronous

Perform something here and now.

I'll regain control to execute something else *when it's done*.

Asynchronous

Initiate something here and now.

I'll regain control to execute something else *"immediately"*.

How an operation is invoked implies nothing about the implementation of the workload itself.

Sync vs Async

"Pause for 10 seconds, then output 'Hello' to the console."

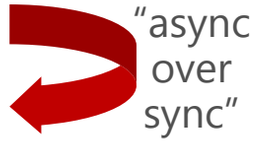
Synchronous

```
public static void PausePrint() {
    var end = DateTime.Now +
        TimeSpan.FromSeconds(10);
    while(DateTime.Now < end);
    Console.WriteLine("Hello");
}
```

```
public static void PausePrint() {
    Task t = PausePrintAsync();
    t.Wait();
}
```

Asynchronous

```
public static Task PausePrintAsync() {
    return Task.Run(() =>
        PausePrint());
}
```




```
public static async Task PausePrintAsync() {
    await Task.Delay(10000);
    Console.WriteLine("Hello");
}
```

Both "async over sync" and "sync over async" can be problematic if you don't completely understand the implementation.

Why go async?

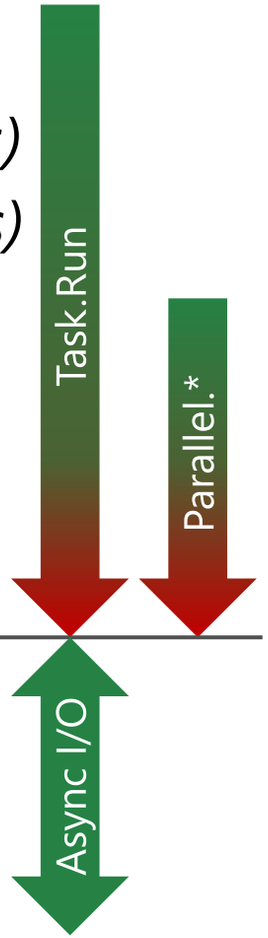
- Offloading
  - Avoid tying up the current thread *(some threads matter more than others)*
  - Do work on a specific thread *(some threads are more capable than others)*

"async over sync" often valuable

- Concurrency
  - Start multiple operations so that they may process concurrently

- Scalability
  - Don't waste resources (i.e. threads) you don't need to use

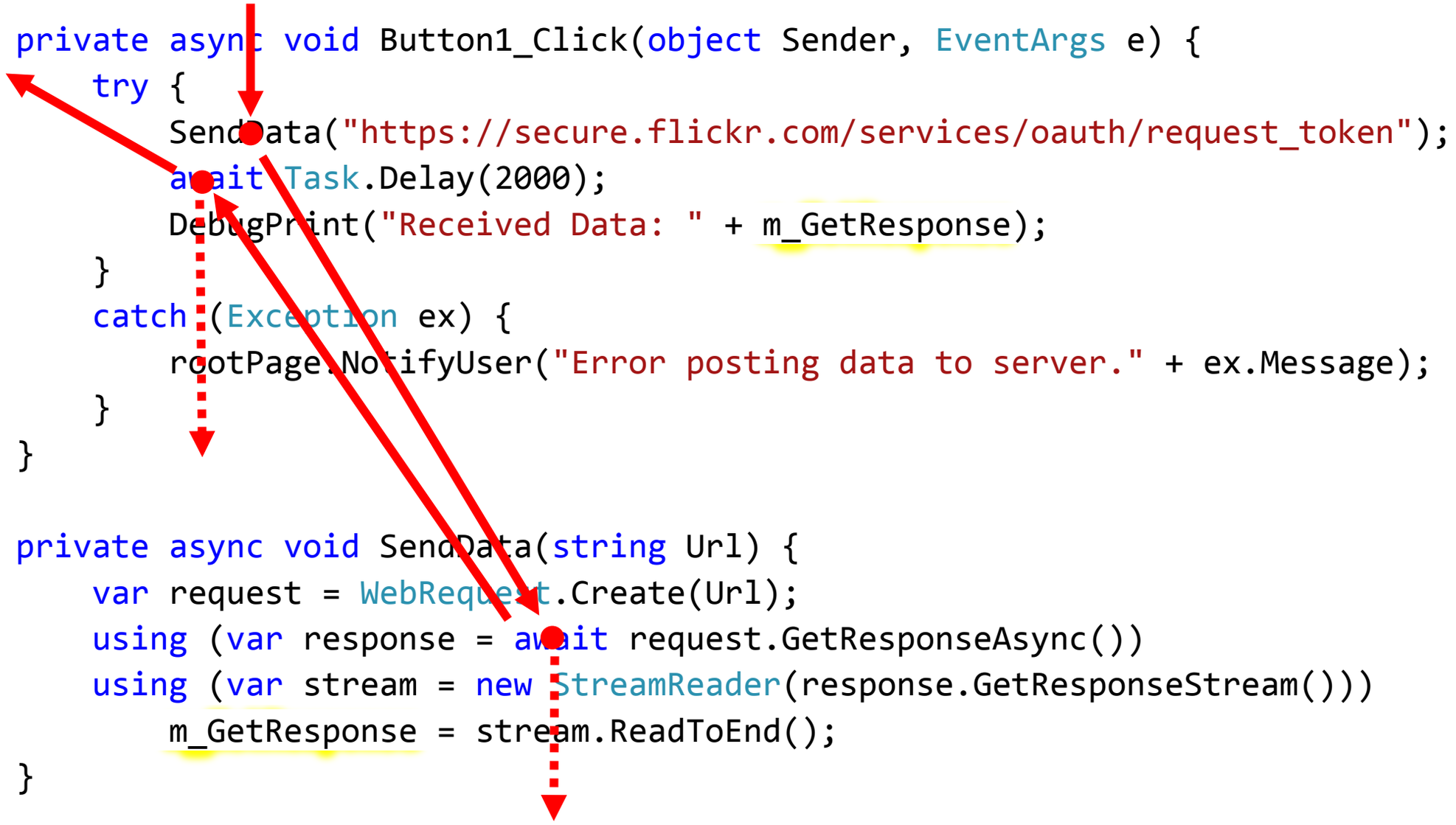
"async over sync" not useful, potentially damaging



async/await are not about "going async." They're about "composing async."

Async Control Flow

Problem with async void



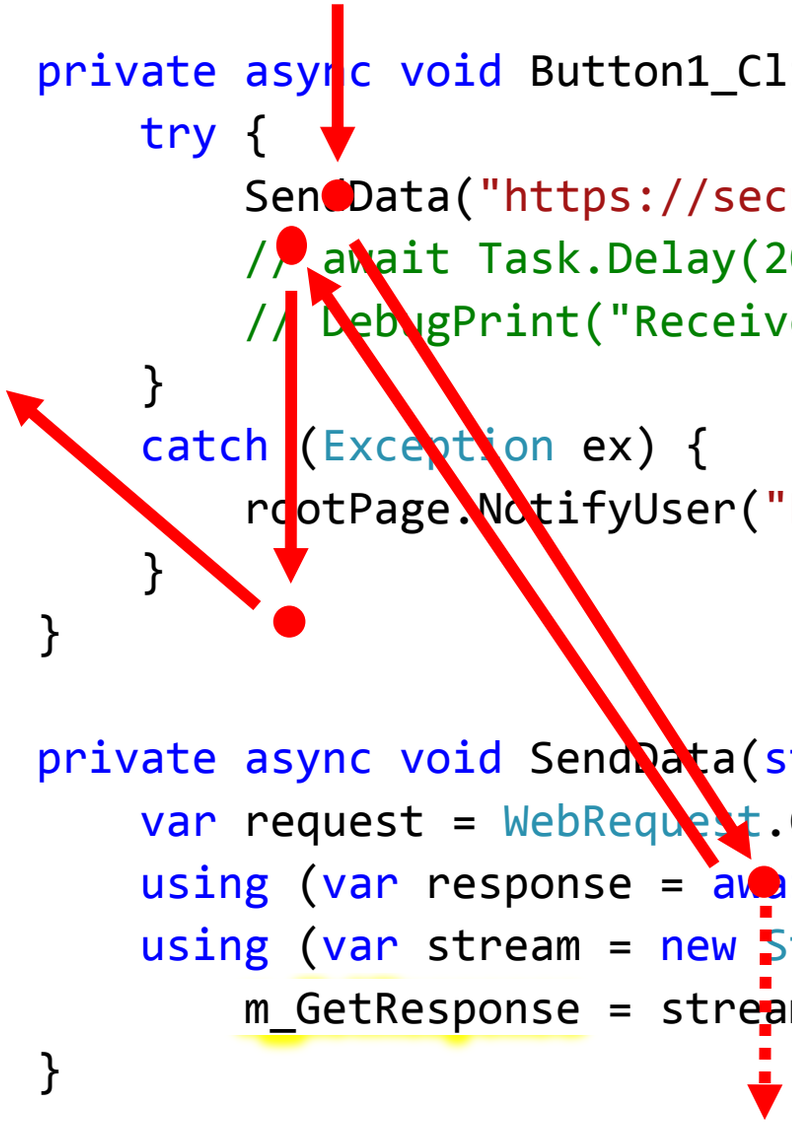
```
private async void Button1_Click(object Sender, EventArgs e) {
    try {
        SendData("https://secure.flickr.com/services/oauth/request_token");
        await Task.Delay(2000);
        DebugPrint("Received Data: " + m_GetResponse);
    }
    catch (Exception ex) {
        rootPage.NotifyUser("Error posting data to server." + ex.Message);
    }
}

private async void SendData(string Url) {
    var request = WebRequest.Create(Url);
    using (var response = await request.GetResponseAsync())
    using (var stream = new StreamReader(response.GetResponseStream()))
        m_GetResponse = stream.ReadToEnd();
}
```

Problem with async void

```
private async void Button1_Click(object Sender, EventArgs e) {
    try {
        SendData("https://secure.flickr.com/services/oauth/request_token");
        // await Task.Delay(2000);
        // DebugPrint("Received Data: " + m_GetResponse);
    }
    catch (Exception ex) {
        rootPage.NotifyUser("Error posting data to server." + ex.Message);
    }
}

private async void SendData(string Url) {
    var request = WebRequest.Create(Url);
    using (var response = await request.GetResponseAsync()) // exception on resumption
    using (var stream = new StreamReader(response.GetResponseStream()))
        m_GetResponse = stream.ReadToEnd();
}
```



For goodness' sake stop using async void

Async void is only for event handlers

Principles

Async void is a “fire-and-forget” mechanism...
The caller is *unable* to know when an async void has finished
The caller is *unable* to catch exceptions thrown from an async void
(instead they get posted to the UI message-loop)

Guidance

Use async void methods only for top-level event handlers (and their like)
Use async Task-returning methods everywhere else
When you see an async lambda, verify it

Passing Task around

```
private async void Button1_Click(object Sender, EventArgs e) {
    try {
   await SendData("https://secure.flickr.com/services/oauth/request_token");
        await Task.Delay(2000);
        DebugPrint("Received Data: " + m_GetResponse);
    }
    catch (Exception ex) {
        rootPage.NotifyUser("Error posting data to server." + ex.Message);
    }
}

private async void SendData(string Url) {
    var request = WebRequest.Create(Url);
    using (var response = await request.GetResponseAsync())
    using (var stream = new StreamReader(response.GetResponseStream()))
        m_GetResponse = stream.ReadToEnd();
}
```

Async

Task

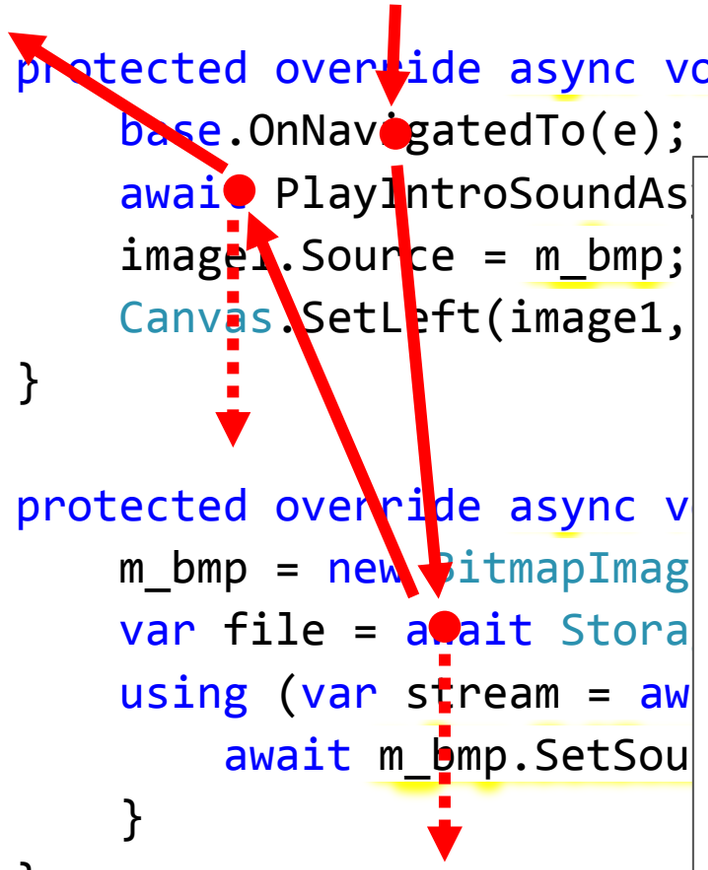
Async

Problem with async void v2

```
// Q. It sometimes shows PixelWidth and PixelHeight are both 0 ???
BitmapImage m_bmp;

protected override async void OnNavigatedTo(NavigationEventArgs e) {
    base.OnNavigatedTo(e);
    await PlayIntroSoundAs
    image1.Source = m_bmp;
    Canvas.SetLeft(image1,
}

protected override async v
    m_bmp = new BitmapImag
    var file = await Stora
    using (var stream = aw
        await m_bmp.SetSou
    }
}
```



```
class LayoutAwarePage : Page
{
    private string _pageKey;

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        if (this._pageKey != null) return;
        this._pageKey = "Page-" + this.Frame.BackStackDepth;
        ...
        this.LoadState(e.Parameter, null);
    }
}
```

Passing Task around

```
// A. Use a task
Task<BitmapImage> m_bmpTask;

protected override async void OnNavigatedTo(NavigationEventArgs e) {
    base.OnNavigatedTo(e);
    await PlayIntroSoundAsync();
    var bmp = await m_bmpTask; image1.Source = bmp;
    Canvas.SetLeft(image1, Window.Current.Bounds.Width - bmp.PixelWidth);
}

protected override void LoadState(Object nav, Dictionary<String, Object> pageState) {
    m_bmpTask = LoadBitmapAsync();
}

private async Task<BitmapImage> LoadBitmapAsync() {
    var bmp = new BitmapImage();
    ...
    return bmp;
}
```

Verify async lambdas

```
// In C#, the context determines whether async lambda is void- or Task-returning. ).
Action a1     = async () => { await LoadAsync(); m_Result="done"; };
Func<Task> a2 = async () => { await LoadAsync(); m_Result="done"; };
                    End Sub

// Q. Which one will it pick? ction()
await Task.Run( async () => { await LoadAsync(); m_Result="done"; });
                    End Function

// A. If both overloads are offered, it will pick Task-returning. Good!
class Task verloads are offered, you must give it Task-returning.
{ ait Task.Run(Async Function() ... End Function)
    static public Task Run(Action a)     {...}
    static public Task Run(Func<Task> a) {...}
    ...
}
```

Verify async lambdas

the other must be set to a value from a predefined list:

```
void OnPrintTaskRequested(PrintManager sender, PrintTaskRequestedEve
{
  // This gets invoked as soon as the Devices pane is shown
  PrintTask task = args.Request.CreatePrintTask("Document Title",
    async (taskArgs) =>
    {
      …
    });

 PrintTaskOptionDetails details =
   PrintTaskOptionDetails.GetFromPrintTaskOptions(task.Options);
```

Verify async lambdas

```
try  {
    await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, async () => {
        await LoadAsync();
        m_Result = "done";

    });
}
catch (
}
finally
    De
}

// IAsy
// delegate void DispatchedHandler();
```

```
// We want a version of Dispatcher.Run that works with async lambdas...
await Dispatcher.NewRunAsync(CoreDispatcherPriority.Normal, async () => { ... });


// Here's one easy way to provide it...
public static async Task NewRunAsync(this CoreDispatcher dispatcher, Func<Task> f)
{
    Task t = null;
    await dispatcher.RunAsync(CoreDispatcherPriority.Normal, () => t = f());
    await t;
    // TODO: use ConfigureAwait(continueOnCapturedContext:false) on both my awaits
}
```

Async/await are not magic

```
// ORIGINAL CODE
async Task FooAsync()
{
    BODY
}
```

```
// APPROXIMATELY WHAT THE COMPILER GENERATES...
Task FooAsync()
{
    var sm = new FooAsyncStruct();
    sm.tcs = new TaskCompletionSource();
    sm.MoveNext();
    return sm.tcs.Task;
}
```

```
struct FooAsyncStruct {
    public TaskCompletionSource tcs;

    private void MoveNext() {
        try {
            TRANSFORMED_BODY
        }
        catch (Exception ex) {
            tcs.SetException(ex); return;
        }
        tcs.SetResult();
    }
}
```

Async/await are not magic

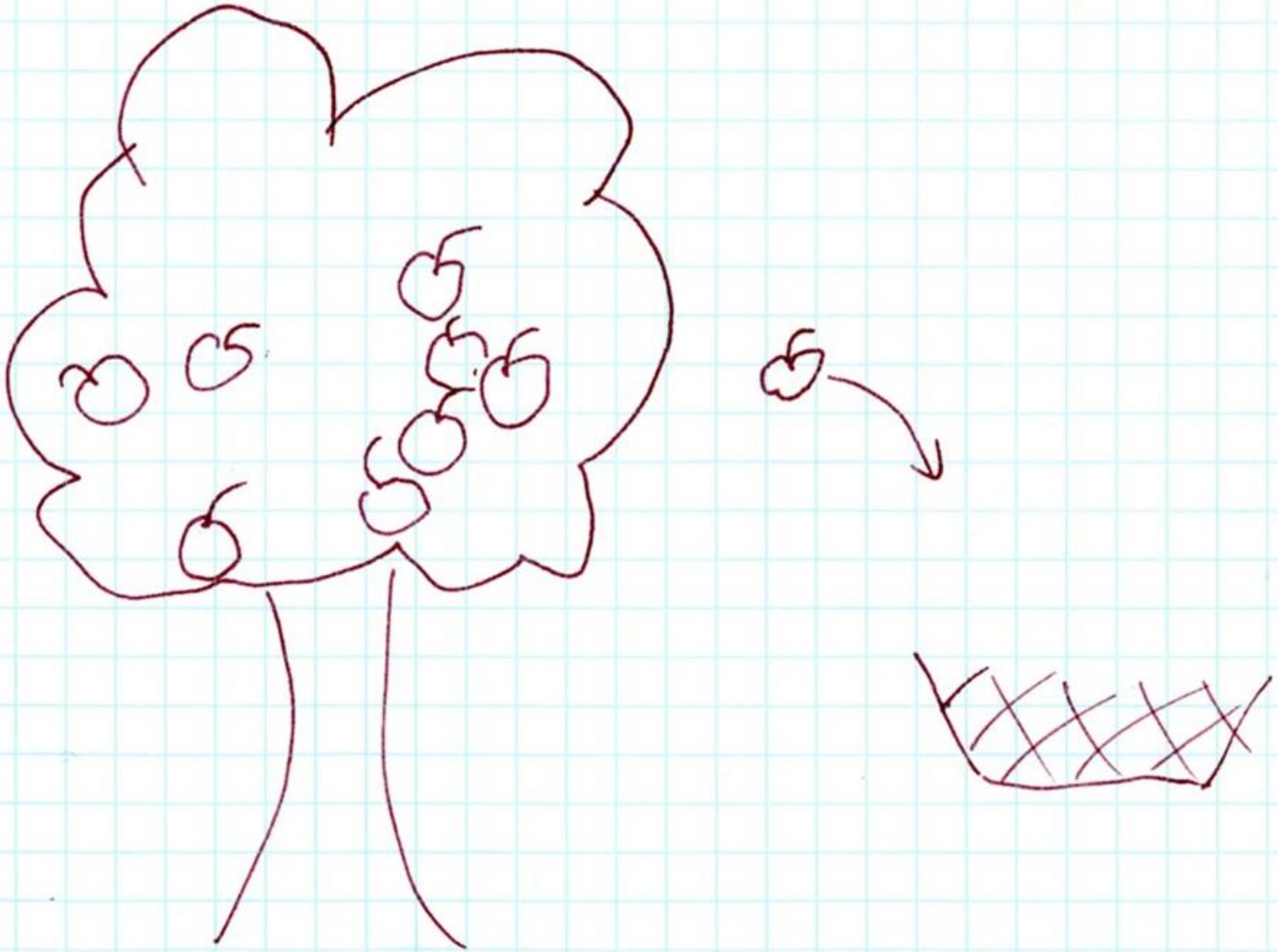
```
// ORIGINAL CODE
var r = await t;


// APPROXIMATELY WHAT THE COMPILER GENERATES...
var tmp = t.GetAwaiter();
if (!tmp.IsCompleted) {
    var ec = ExecutionContext.Capture()
    ((INotifyCompletion)tmp).OnCompleted(_ => ExecutionContext.Run(ec, K1));
    return;
}
K1:
var r = tmp.GetResult();

// The two lines in italics are executed from within a routine inside mscorlib
// The argument "K1" represents a delegate that, when executed, resumes execution at the label.
// If "tmp" is a struct, any mutations after IsCompleted before GetResult may be lost.
// If tmp implements ICriticalNotifyCompletion, then it calls tmp.UnsafeOnCompleted instead.
// The variable "ec" gets disposed at the right time inside the lambda
// If ec is null, then it invokes K1 directly instead of via ExecutionContext.Run
// There are optimizations for the case where ExecutionContext is unmodified.
```

Everything as a Task

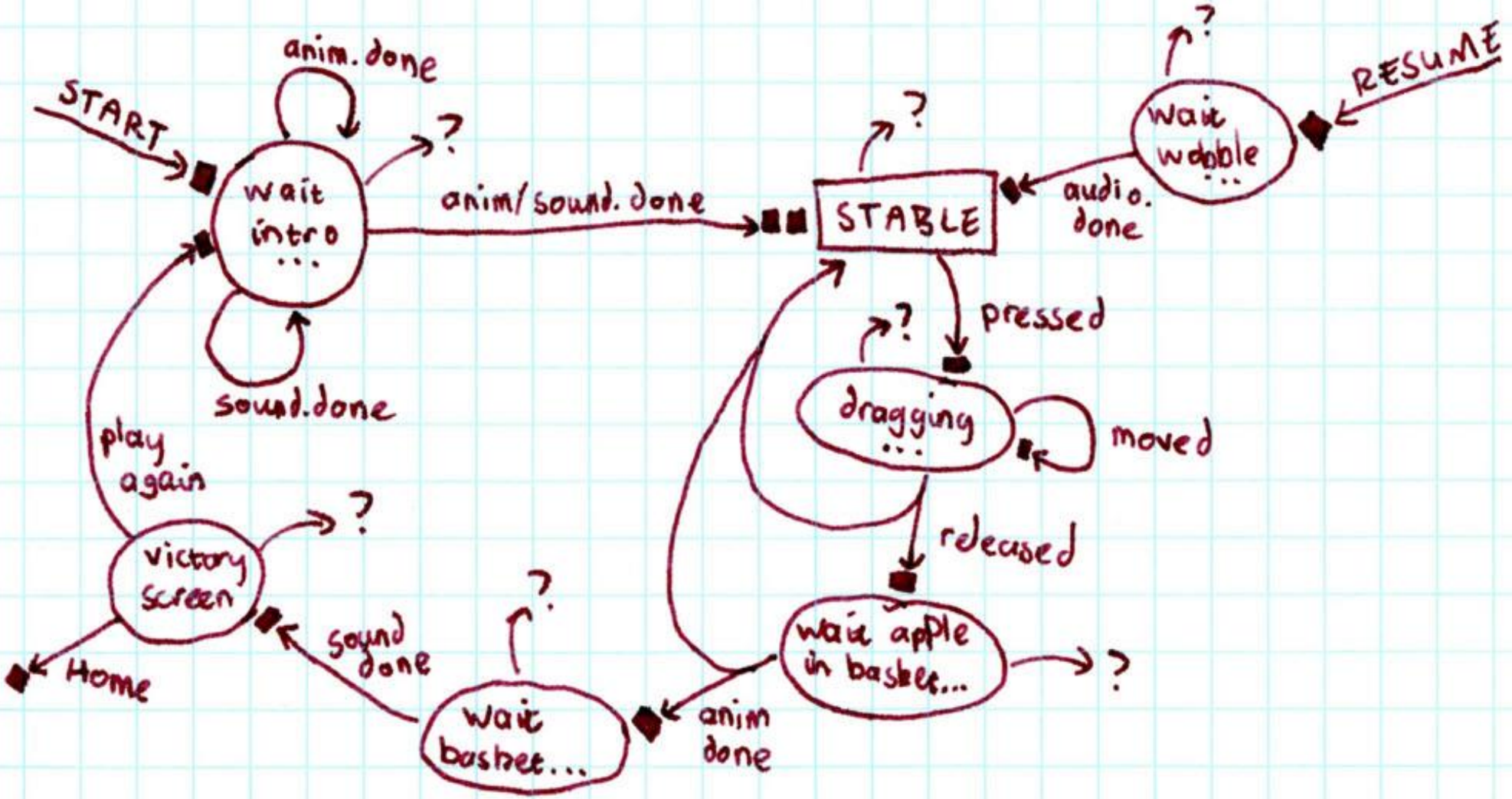
Event-based code…



Event-based code…

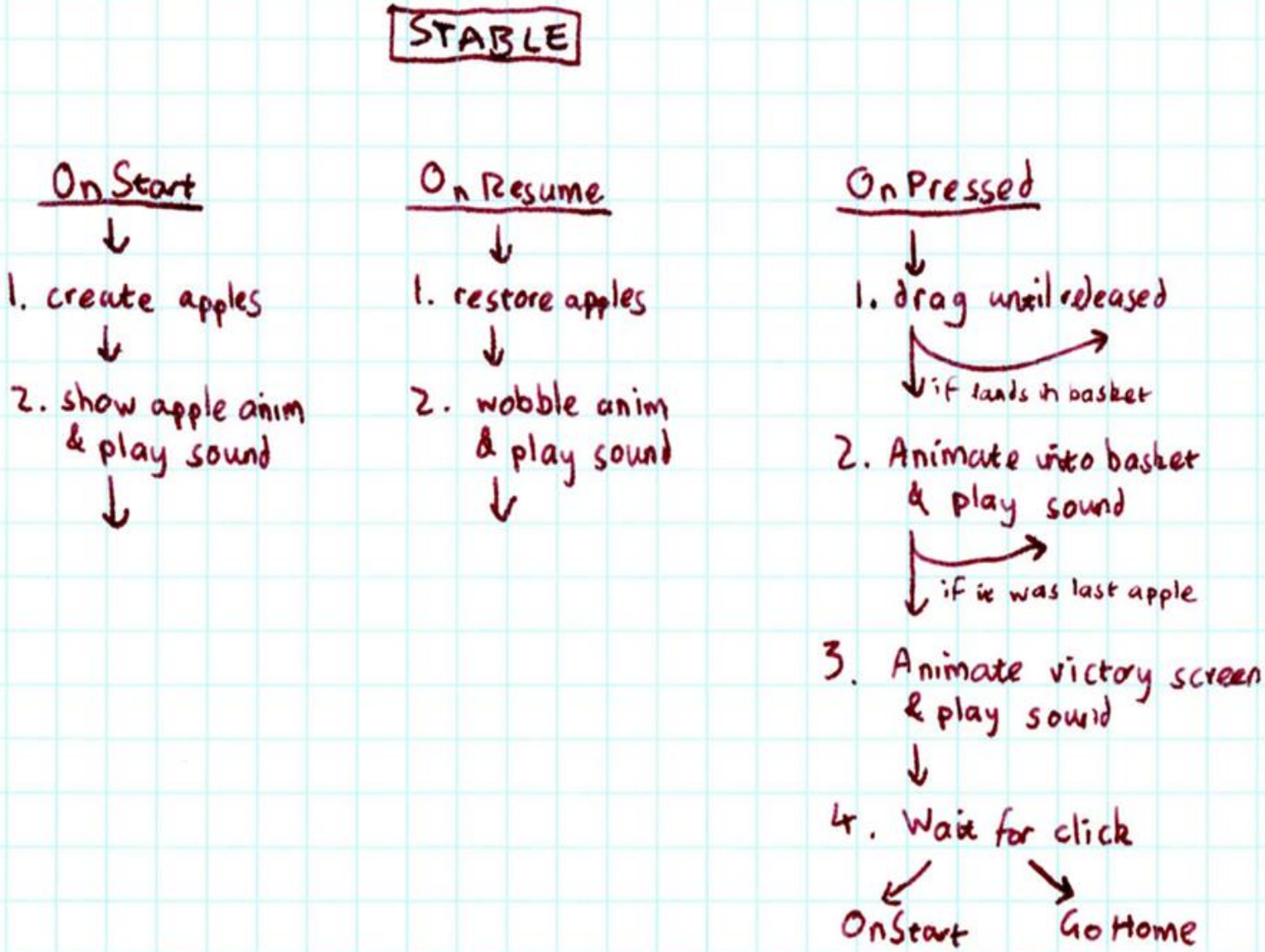
```
Protected Overrides Sub OnPointerPressed(e As PointerRoutedEventArgs)
    Dim apple = CType(e.OriginalSource, Image)
    AddHandler apple.PointerReleased,
        Sub(s, e2)
            Dim endpt = e2.GetCurrentPoint(Nothing).Position
            If Not BasketCatchmentArea.Bounds.Contains(endpt) Then Return
            Canvas.SetZIndex(apple, 1) ' mark apple as no longer free
            If FreeApples.Count > 0 Then
                m_ActionAfterAnimation = Sub()
                                             WhooshSound.Stop()
                                             ShowVictoryWindow()
                                             AddHandler btnOk.Click, Sub()
                                                                         ....
                                                                     End Sub
                                         End Sub
            End If
            WhooshSound.Play()
            AnimateThenDoAction(AppleIntoBasketStoryboard)
```

Event-based code...



The problem is events.
They're not going away.

Async over events



Async over events

```
Protected Async Sub OnPointerPressed(e As PointerRoutedEventArgs)
    ' Let user drag the apple
     Dim apple = CType(e.OriginalSource, Image)
     Dim endpt = Await DragAsync(apple)
     If Not BasketCatchmentArea.Bounds.Contains(endpt) Then Return

     ' Animate and sound for apple to whoosh into basket
     Dim animateTask = AppleStoryboard.PlayAsync()
     Dim soundTask = WhooshSound.PlayAsync()
     Await Task.WhenAll(animateTask, soundTask)
     If FreeApples.Count = 0 Then Return

     ' Show victory screen, and wait for user to click button
     Await StartVictoryScreenAsync()
     Await btnPlayAgain.WhenClicked()
     OnStart()
End Sub
```

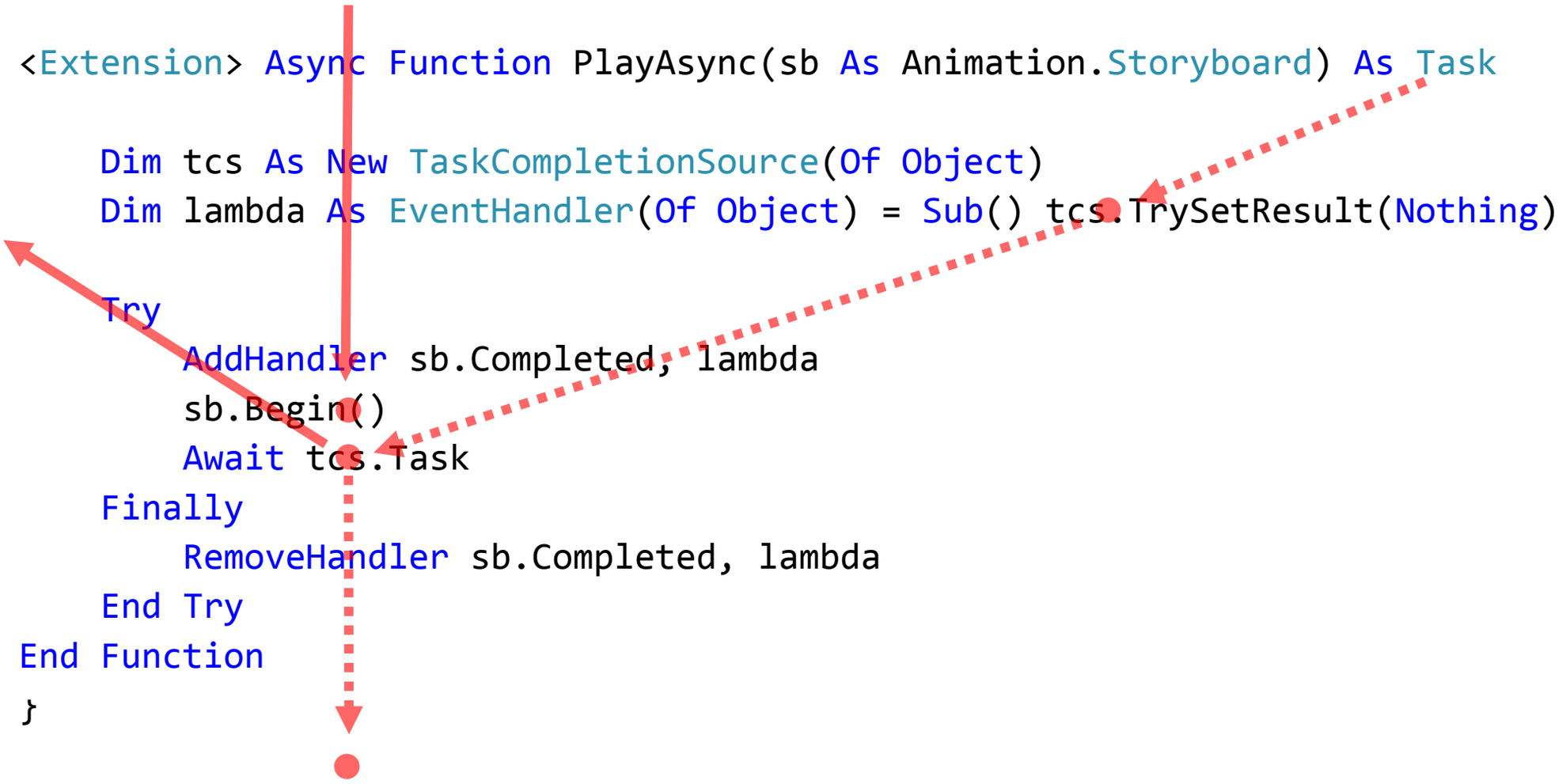
Async over events

```
'  Usage: Await storyboard1.PlayAsync()

<Extension> Async Function PlayAsync(sb As Animation.Storyboard) As Task

    Dim tcs As New TaskCompletionSource(Of Object)
    Dim lambda As EventHandler(Of Object) = Sub() tcs.TrySetResult(Nothing)

    Try
        AddHandler sb.Completed, lambda
        sb.Begin()
        Await tcs.Task
    Finally
        RemoveHandler sb.Completed, lambda
    End Try
End Function
}
```



Async over events

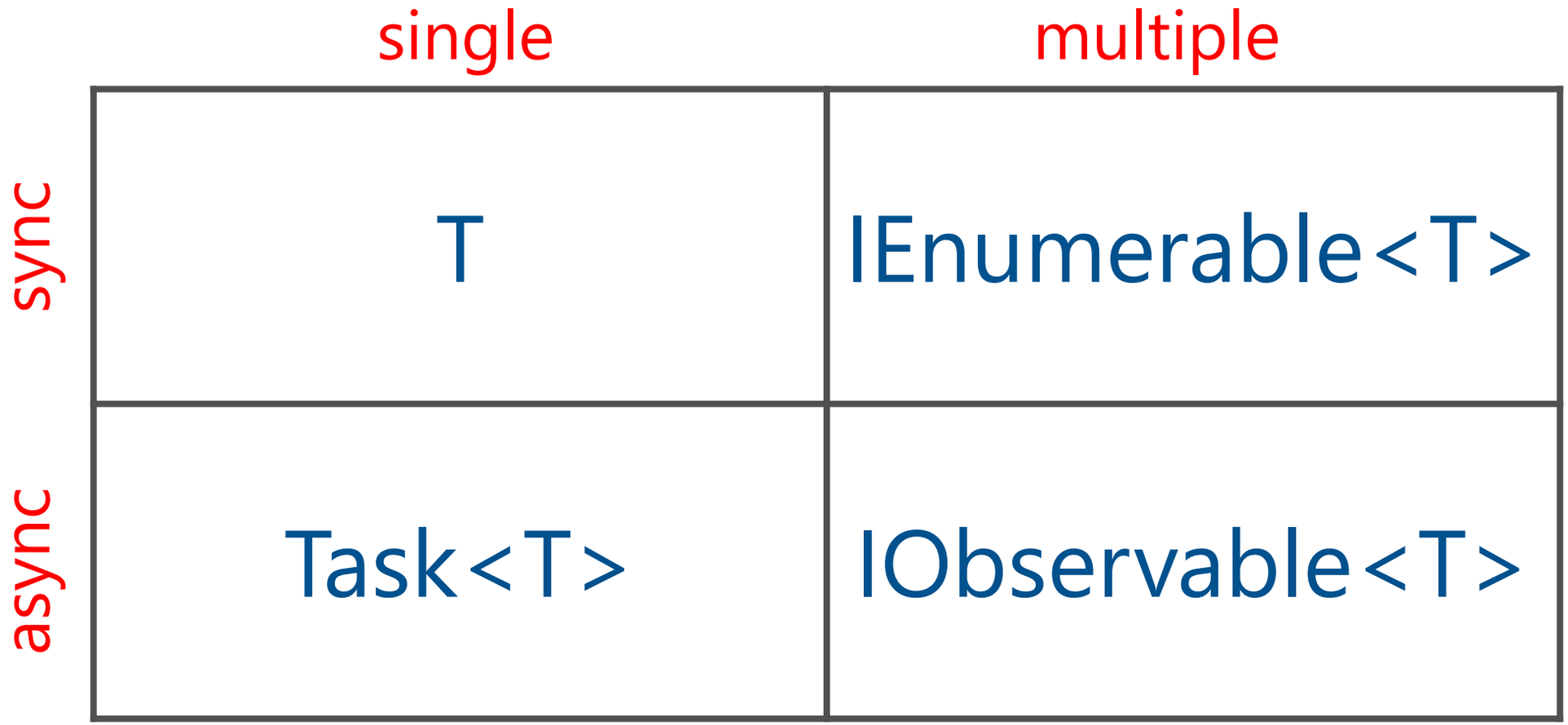
```
// Usage: await button1.WhenClicked();

public static async Task WhenClicked(this Button button)
{
    var tcs = new TaskCompletionSource<object>();
    RoutedEventHandler lambda = (s,e) => tcs.TrySetResult(null);

    try {
        button.Click += lambda;

        await tcs.Task;
    }
    finally {
        button.Click -= lambda;
    }
}
```

Async and RX


single
multiple
sync
async
T
IEnumerable<T>
Task<T>
IObservable<T>

Library Thinking

Be a responsible library developer

Library methods shouldn’t lie.

Be honest. Use “XxAsync” if, and only if, you’re not thread-bound (with a few notable exceptions).
Suffix should help caller to understand implementation.

Be a responsible library developer

Library methods shouldn’t lie.

Be honest. Use “XxAsync” if, and only if, you’re not thread-bound (with a few notable exceptions).
Suffix should help caller to understand implementation.

Library methods might be called from various environments.

Context becomes critical. Be agnostic whenever possible.
ConfigureAwait(continueOnCapturedContext:false) is your best friend.

Be a responsible library developer

Library methods shouldn't lie.

Be honest. Use "XxAsync" if, and only if, you're not thread-bound (with a few notable exceptions).
Suffix should help caller to understand implementation.

Library methods might be called from various environments.

Context becomes critical. Be agnostic whenever possible.
ConfigureAwait(continueOnCapturedContext:false) is your best friend.

Library methods might be used from perf-sensitive code.

Performance becomes critical.

- Design for chunky instead of chatty.
- Optimize the synchronously completing cases.

Common Problems

Symptom: Not Running Asynchronously

```
async void button1_Click(…)
{
    Action work = CPUWork;
    await Task.Run(() => work());
    …
}
```

Problem:
Thinking ‘async’ forks.

Solution:
Use Task.Run around a synchronous method if you need to offload… but don’t expose public APIs like this.

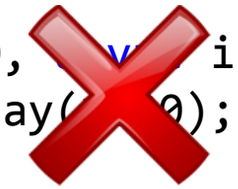
Symptom: Completing Too Quickly

```
…;
await Task.Delay(1000);
…;
```

Problem:
Neglecting to await.

Solution:
Await your awaitables.

▲ 1 of 12 ▼ ParallelLoopResult Parallel.For(**int fromInclusive**, int toExclusive, Action<int> body)
Executes a for (For in Visual Basic) loop in which iterations may run in parallel.
fromInclusive: *The start index, inclusive.*



```
Parallel.For(0, 10, async i => {
    await Task.Delay(1000);
});
```

Problem:
Async void lambda.

Solution:
Use extreme caution (& knowledge) when passing around async lambdas as void-returning delegates.

▲ 2 of 16 ▼ (awaitable) Task<Task> TaskFactory.StartNew<Task>(Func<Task> **function**)
Creates and starts a System.Threading.Tasks.Task<TResult>.

Usage:
Task x = await StartNew(...);
function: *A function delegate that returns the future result to be available through the S…*

```
await Task.Run(
    async () => { await Task.Delay(1000); });
```

Problem:
Task<Task> instead of Task.

Solution:
Make sure you're unwrapping your Tasks, explicitly (Unwrap) or implicitly.

Symptom: Method Not Completing

```
async void button1_Click(…) {
    await FooAsync();
}
```

Problem:
Deadlocking UI thread.

Solution:
Don't synchronously wait on the UI thread.

```
async Task FooAsync()  {
    await Task.Delay(1000).ConfigureAwait(false);
}
```

Problem:
Deadlocking UI thread.

Solution:
Use ConfigureAwait(false) whenever possible.

```
var tcs = new TaskCompletionSource<T>();
Task.Run(delegate {
    try {
        T result = Foo();
        tcs.SetResult(result);
    }
    catch(Exception e) { tcs.SetException(e); }
});
return tcs.Task;
```

Problem:
Awaited task never completes.

Solution:
Always complete Tasks when the underlying operation has ended. Always.

That’s a Wrap…

Key Takeaways

- Understand why you're "going async," then code accordingly.
- 'async' doesn't force asynchrony.
- 'async void' is only for top-level event handlers.
- 'async'/'await' are not magic.
- 'async'/'await' are about composition, not the leaves.
- Implementing a library? Be kind to your consumers.

Follow-up Reading

Async/Await FAQ:
http://blogs.msdn.com/b/pfxteam/archive/2012/04/12/async-await-faq.aspx

Task-based Async Pattern:
http://aka.ms/tap

"Async over Sync" & "Sync over Async":
http://blogs.msdn.com/b/pfxteam/archive/2012/03/24/10287244.aspx
http://blogs.msdn.com/b/pfxteam/archive/2012/04/13/10293638.aspx

Forking:
http://blogs.msdn.com/b/pfxteam/archive/2011/10/24/10229468.aspx
http://blogs.msdn.com/b/pfxteam/archive/2012/09/11/forking-in-async-methods.aspx

Common problems:
http://blogs.msdn.com/b/pfxteam/archive/2013/01/28/psychic-debugging-of-async-methods.aspx
http://blogs.msdn.com/b/pfxteam/archive/2012/02/08/10265476.aspx
http://blogs.msdn.com/b/pfxteam/archive/2011/10/02/10218999.aspx
http://blogs.msdn.com/b/pfxteam/archive/2012/04/12/10293249.aspx
http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115163.aspx

Performance:
http://blogs.msdn.com/b/pfxteam/archive/2012/02/03/10263921.aspx
http://blogs.msdn.com/b/pfxteam/archive/2012/03/03/10277034.aspx
http://blogs.msdn.com/b/pfxteam/archive/2011/10/24/10229662.aspx

Await Anything
http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115642.aspx
http://blogs.msdn.com/b/lucian/archive/2012/11/28/how-to-await-a-storyboard-and-other-things.aspx
http://blogs.msdn.com/b/lucian/archive/2012/11/28/how-to-await-a-button-click.aspx

WinRT & Async
http://blogs.msdn.com/b/windowsappdev/archive/2012/04/24/diving-deep-with-winrt-and-await.aspx
http://blogs.msdn.com/b/windowsappdev/archive/2012/06/14/exposing-net-tasks-as-winrt-asynchronous-operations.aspx

Parallel Patterns
http://www.microsoft.com/en-us/download/details.aspx?id=19222
http://msdn.microsoft.com/en-us/library/ff963553.aspx