[START with Win8 demo app running on main page]

# 0. The new async design patterns

Lucian Wischik. Techready 16, Feb 5th 2013.

\* Deep breath. Pause, look around, smile.

\* START WITH ENERGY

\* I'm Lucian Wischik, a program manager on the VB/C# language design team.

\* heavily involved with async over the past three years, new feature in VS2012

## Session objectives

\* Async involves some new concepts. Not difficult, just unfamiliar.

\* As we designed feature, knew we'd have to steer people in right direction. That's what this talk is.

\* We'll be learning "Think In Async"...

\* Only learnt to drive when I was 30, after a year of living in America.

\* Instructor kept telling me to "think in car". Never said what.

\* Long story short, I got a different instructor.

\* I'll be telling you what precisely it means to think in async

\* Important lesson: if you go away only remembering these 3 sentences, happy.

\* Async void is only for top-level event handlers.

\* Can use tasks around events.

\* Distinguish IO-bound from CPU-bound code.

\* In three sections of this talk, we'll be fleshing out those three.

## Confidential

\* I have to put this slide up.

\* Content of my talk is public. I want, NEED you to spread the word. Edited version of slides up on blog.

## Evaluation form

\* Please fill this out!

## Understanding Async

\* How many of you went to Alex's talk yesterday, or have used async in your code?

\* How many of you feel confident you understand how it works?

\* Okay, so let's get up to speed.

\* In an ideal world, async is as straightforward as this code. You use awaits for long-running tasks. You put on the async modifier, which lets a method contain awaits. You make it return Task. And that’s it.

\* But when people deviate from the straightforward pattern, that’s where they sometimes run into problems.

\* Let’s start by understanding how it works.

\* I think everything boils down to the message-loop.

\* If you get this, I think you can predict the next 75 minutes of this talk.

[CLICK]

\* In any UI application, Winforms/WPF/Silverlight/Phone/Win8, the UI thread runs a loop

\* When a click arrives, it invokes the handler.

[CLICK]

\* When async method hits first await, returns to its caller.

[CLICK]

\* When it hits first await, ... you know the drill

[CLICK]

\* Back to the message-loop. That's why it's responsive, and ready to handle more UI interactions.

\* Doesn't freeze.

[CLICK]

\* Later on, let's say the network download has finished.

\* Task gets marked as completed, and if UI thread is free, can resume where it left off.

\* That's going to be: finish the method

[CLICK]

\* And so the task gets marked as completed, and again the UI thread can resume where it left off.

\* Okay, that's the mechanism.

\* Even if you don't follow, that's fine, we'll spell out the practical ramifications.

[END]

# 4. Thread tie-up

\* That ties it up.

\* I think everything boils down to this thread of control.

\* If you understand how control flows in an async method, at the awaits,

then you can figure out the rest of this talk.

## Objectives

\* Async void, fire-and-forget, is only suitable for top-level event handlers.

\* We saw how we used TaskCompletionSource to wrap Tasks around events, to make code simpler

\* And we saw how to use background-thread for CPU-bound work,

and how to use await for IO-bound work.

## Related

\* I just want to call out some talks.

\* Dev336 already happened this morning. Paul Harrington talked about memory-bound code, which is a class of its own. Fantastic talk, and you should catch it up if you weren't there already.

\* Dev318 in just half an hour by the amazing Bart de Smet will talk about RX,

the other chief way to tame events.

\* I'd talked about F# for the analytical back-end. On Friday, Dev329, Donna will talk more about how to take advantage of that.

## Evaluation form

\* Quick reminder. Please fill out an evaluation form.

## Q&A

\* And now let's hand it over for any questions!

\* Thank you

# 1. Async void is only for event handlers

\* And so on to the first of three sections of this talk. First practical ramification.

\* Async void is only for event-handlers.

\* I'll motivate it with customer stories.

\* Actually, all the scenarios in this talk come straight from customers, and most of the code.

[CLICK]

**\* "I have a Silverlight page that uses RIA services async to load the data for the page."**

**\* "This works fine if the user waits for a few seconds before selecting the print button."**

**\* "But does \*not\* work if the user prints right away."**

**\* "If the user clicks the Print button before all of the page data is loaded, the printed output does not have all of the data."**

\* Diagnosis: she was using async void deep inside her code.

\* Fix: should return Task from her internal async methods, not void.

## For goodness' sake

\* Let me put that more strongly

\* For goodness' sake, stop using async void everywhere.

\* (At first that was going to be the title of my talk)

## Button1\_Click

\* Actually, I won't show her code because it was a bit involved.

\* I'll show someone else who made the exact same mistake.

\* Their async method returns void rather than Task.

\* This code actually comes from Microsoft's own official Win8 SDK samples!

\* Goes to show that it's a common mistake that anyone can make.

[CLICK]

\* Clicks a button, invokes the handler

[CLICK]

\* Invokes SendData, which kicks off request for data and then awaits response

\* You know what happens now. At the first await, control returns straight back to the caller.

[CLICK]

\* Normally at this point the caller would await until SendData finishes.

\* But SendData returned void, not Task, so the caller can’t do that.

\* Instead it awaits a Task Delay, so returns back to its own caller, the UI message loop

[CLICK]

\* Some time later, the response will come back. Or the delay will finish.

\* Don't know which will happen first.

\* Maybe it'll assign to m\_GetResponse first. Or maybe not.

\* That's what the customer said "My code doesn't work 100% reliably".

\* Had obviously experimented with Task.Delay until they got the right delay to work on their dev network!

\* The problem is all down to this async void SendData.

\* It's a void, right. It doesn't return anything to its caller.

\* The caller can't do anything with it.

\* The caller is UNABLE to know when SendData has finished.

\* It's basically fire-and-forget.

\* That's the crux of the problem, fire-and-forget.

## TryCatch with Button1\_Click

\* Actually, before we go on to fix it, I want to highlight another problem with fire-and-forget async voids.

\* Let's comment out the problematic race condition

\* and see how exceptions behave from a fire-and-forget method.

\* We'll focus on the try/catch, to catch exceptions arising from SendData.

[CLICK]

\* Once again, we invoke the handler

[CLICK]

\* It calls SendData.

[CLICK]

\* As you know, at the first await, it returns to its caller.

\* The thing is, at this stage, there's been no exception yet, so nothing gets caught.

\* We breeze through the catch block and return to the UI.

[CLICK]

\* Now the network request comes back, maybe with a 404 error.

\* And SendData throws an exception.

\* But where can an exception go out of a fire-and-forget method?

\* Can't go back to Button1\_Click, because that's already finished.

\* Answer is that all exceptions from these fire-and-forget async voids

get posted straight to the UI thread.

\* In Win8, terminates app. In Phone, silently swallowed. In WPF, dialog.

\* In no cases is that desirable.

## Principles & Guidance: Async void only for event handlers

\* We've seen that async void is a "fire-and-forget" mechanism

\* Meaning: the caller is \*unable\* to know when an async void has finished.

\* And the caller is \*unable\* to catch exceptions from an async void method.

\* Guidance is to use async void solely for top-level event handlers.

\* Everywhere else in code, like SendData, async methods should return Task.

\* This is an issue unique to async methods. It never came up with non-async code. It only crops up when a method has await in it.

\* There’s one other danger, about async void in lambdas, I’ll come to it in a moment.

## Fix the Button1\_Click SendData

\* But first let’s fix SendData.

\* SendData should return Task, not void.

\* Convention: every method that returns Task has a name ending with Async

\* The caller sees that name and knows he should await it.

\* And we can get rid of that goddawful Task.Delay rubbish.

## BitmapImage

\* Well, we've said async void is for fire-and-forget

\* And the only place that's appropriate is for event-handlers, or event-like things.

\* What do I mean by "event-like things"? Sometimes it's hard to know.

\* Let's look at this case.

\* I was wondering how bad the problem is of people misusing async void.

\* I looked through the MSDN forums for "async void" and "problem".

\* A lot of hits came back from this function "async void LoadState"

\* You might not know about it.

\* In Win8 apps. When you get to a page, it fires the NavigatedTo event

\* The base class handles the event with an overridable void-returning method OnNavigatedTo.

\* So that method's basically like an event-handler, fire-and-forget. It's fine to be async.

[CLICK]

\* First thing it does is call its base method.

\* If the page had already been shown before, it just returns.

\* But just for the first time that a page is shown, it invokes the virtual void method LoadState.

\* So LoadState is also basically like an event-handler, fire-and-forget. It's fine to be async void.

\* OnNavigatedTo is called every time you navigate to a page. LoadState called is called only once per time the page has been constructed.

[CLICK]

\* Maybe you can see where this is going...

\* Let's trace it out.

\* We get to a page. Invoke the OnNavigatedTo handler, fire-and-forget.

\* Which calls its base.

[CLICK]

\* Which kicks off LoadState, fire-and-forget, which we’ve overridden.

\* It does an await

[CLICK]

\* Which goes back to its caller, who does an await, and returns to the UI message-loop

[CLICK]

\* But now, there are two fire-and-forget async voids in flight. Which one will go first?

\* Will it be the bottom one who loads the bitmap?

\* Or the top one you uses the bitmap, and assumes it's already loaded?

\* The customer question you hear is "Why is PixelWidth 0?"

\* It's because they're querying a bitmap that hasn't loaded yet.

## Use a task

\* Well, the answer here is to use a task.

\* It would have been easier to change LoadState to be a Task-returning async method.

\* But we can’t do that. We don’t control the signature. We’re just overriding it.

\* So instead we’ll have to pass the Task back an alternative way.

\* Here, LoadState kicks off an async method that loads a bitmap

\* But not fire-and-forget.

\* Oh no. Instead, it'll rememer that task, and save it in m\_bmpTask field.

\* That way, OnNavigatedTo can await for the same task to finish.

## Would you like to know more?

\* There are some other typical places where users run into problems with async void.

\* How to use await in constructors, or property-getters, or how to deal with re-entrancy.

\* I’ve left them as hidden slides, so if you download the slide deck you can read more.

## -> Async void constructors

\* That's all fine and well. We know how to fix up the easy case.

\* Got this question on the internal windows-developer discussion alias.

\* “I have a zillion Task-returning awaitable methods. This is mostly good, but I have some places where they I want to call them from non-awaitable functions like constructors. How?”

\* Why am I mentioning this in async void?

\* Well the customer just put the code inside an async void, and invoked that from the constructor.

\* Fire-and-forget. Terrible idea. Constructor that returns saying

\* "Hey, I might or might not have finished construction. It's fire-and-forget. You feeling lucky?"

[CLICK]

\* DON’T. Constructors in .NET have always been things that quickly initialize an object.

\* If your initialization takes a long time then you should do it lazily or use a factory.

\* Here, we have a factory method called LoadAsync. It's async, and it produces a House.

\* Okay, solved that one...

## -> Async void properties

\* The same customer asked exactly the same question about how to await something from inside a property-getter.

\* In this case, it wants a property that returns the longitude/latitude of a house.

\* Wanted to make the getter async. But compiler gave an error message.

\* Well, it should! The .NET design is that properties are lightweight.

\* If you have something heavyweight -- and a network call is very heavyweight -- then it's not a property.

\* So move it into an async method instead.

\* Here, GetLongitudeLatitudeAsync.

## -> BtnPurchase\_Click

\* Let's look at another aspect of async voids.

\* We said the caller can't tell when an async void has finished.

\* Of course it can't. It merely returns void.

\* Well, in a UI, you invoke an async void event-handler, and the UI doesn't know that.

\* Doesn't know that the handler is already in-flight.

\* User can click the button again, and invoke a second instance of the handler,

even while the first is still running.

\* Might be fine in some cases. Not in this case, where it's a button to buy stuff.

[CLICK]

\* So, we'll disable the button in the handler.

\* Then do the await. That way, user can't click the button again. Only one purchase at a time.

## C# async lambdas

\* There's just one other surprise place where async voids will bite you

\* In C#, when you write an async lambda, it can be either void-returning or Task-returning.

\* The syntax of the lambda doesn't tell you which.

\* Instead, it's the context that tells you.

\* Here I’ve assigned same async lambda to both void-returning Action delegate, and Task-returning Function delegate.

\* Look at this call to Task.Run. It passes an async lambda.

\* Will that be void-returning or Task-returning?

[CLICK]

\* Well, if both overloads are offered, it'll pick Task-returning. Good!

[CLICK]

\* In VB, the situation's different

\* Here it's not the context that decides if it's void-returning Sub or Task-returning Function.

\* Instead the expression itself says which it is.

\* But the conclusion's the same. The method you call should generally both overloads.

## Dispatcher.RunAsync

\* Let's see async lambda problems in practice

\* Here I'm writing a Win8 app which invokes Dispatcher.RunAsync

\* I'm passing it an async lambda.

\* Whenever I see an async lambda being passed to a function, I always check that function.

\* Look at the bottom of the slide.

\* In this case, it takes something called a DispatchedHandler,

\* which is void-returning.

\* So it's passing a void-returning async.

[CLICK]

\* We can imagine what will happen.

\* The dispatcher will kick off the lambda, fire-and-forget.

\* The lambda will get to the first await.

[CLICK]

\* It'll return it its caller. And the dispatcher will think it's done.

[CLICK]

\* So our await on the dispatcher will finish, and we'll plow through the rest of our method.

[CLICK]

\* Meanwhile, m\_Result doesn't get set until too late,

\* and the exception from our async lambda was never caught.

\* That's because our async lambda was void-returning, fire-and-forget.

[CLICK]

\* You understand the problem.

\* We’ll touch on a solution in next section. It’s subtle.

## Principles & Guidance

\* But first, let's sum up.

\* "For goodness' sake, stop using async void"

\* That's because async void means fire-and-forget.

\* And fire-and-forget is only appropriate for event-handlers.

# 2. Async Over Events

\* Section two out of three.

\* This actually came from the same customer who motivated the previous section.

[CLICK]

**\* "I have a Silverlight page that uses RIA services async to load the data for the page."**

**\* "It fires an OnWhatever method when the load is complete."**

**\* "If the user clicks the Print button before all of the page data is loaded, the printed output does not have all of the data."**

\* And then she asked

**\* "Can I use Await in some way to wait for the OnWhatever event?"**

\* You know, that kind of forms-over-data doesn't make very exciting demos.

\* (although that customer did have a fantastic user-friendly UI)

\* I'm going to work through the code of a different customer.

\* His code was awful, like spaghetti. Lost ability to maintain it or add features. Wanted help cleaning it up.

\* The problem again was that he was building a complicated UI around events.

\* The fix was to wrap them up in tasks so he could await for the events.

## Tree sketch

\* This is how it started,

\* with an idea for an educational game for young children on phone and tablet.

\* There's an apple tree, and the child drags apples into a basket,

and learns numbers and counting.

\* Like The Count from Sesame St. One apple. ha ha ha ha. Two apples. ha ha ha.

[CLICK]

\* Let's see the game in action...

## DEMO

\* Start the game

\* Drag all apples into the cart

\* Watch it zoom off

\* Click Play Again

\* Move just one apple

\* Go back to Main, then play the apples again.

## Sub OnPointerPressed

\* This is what the customer's code looked like.

\* He was writing in VB.

\* Don't worry, I'm not going to ask you to understand it.

\* He didn't either. Nor did I.

\* What I did see was a nightmare of nested lambdas, even lambdas assigned to fields.

\* It was impossible to know what the code was doing, impossible to debug,

impossible to add features.

## State-machine diagram

\* So we started from the beginning.

\* Get at the architecture of the game.

\* Drew a state machine of how the game's meant to work.

\* Well, when it starts, it kicks off the animation and sound to put the apples into the tree.

\* (I represented that piece of code with the little solid square)

\* Then it waits until the animation and sound fire their OnCompleted events.

\* When both of them are done, it sets up the game, and waits for interactions

\* If it gets a PointerPressed event, then dragging starts.

\* In this state it can get a PointerMoved or PointerReleased event...

\* Look, this isn't the right way to go about it.

\* You know what’s the worst? Those solid black squares are the heart of our program, they’re what DOES stuff.

\* But the entire architecture is spent solely on WHEN it does stuff!

\* We tried to think of the problem more cleanly than the nested lambdas,

but we didn't help matters any.

## The problem is events

\* Basically the problem boils down to events.

\* Building a complicated UI that responds to events.

\* The customer had tried one approach, keeping the event handlers local.

\* That led to a nightmare of nested lambdas.

\* We tried another approach, figuring out the game's state machine, and how it responds to events.

\* That ended up being too global. That is, things that should have been local ended up being part of the global state machine. What should have been local variables were promoted to class fields.

\* The thing is, we have to figure out a way to tame events.

\* Events have been with us for a long time, and they'll be with us for a long time to come.

\* WPF, Silverlight, even in Windows8, they're still full of events.

\* The challenge is how to tame them.

\* I'm going to show how we can tame them by wrapping them up with tasks.

## Nicer state diagram

\* Look, here's the architecture diagram I want to draw.

\* I want to write the game where it has only one single state.

\* And I'll write three event handlers, OnStart, OnResume, OnPressed

\* What happens OnPressed?

\* 1. We do a drag operation. 2. Then we do an animation. 3. Then zoom off the cart and bring in a victory screen. 4. Then wait for the button to be clicked.

\* But I want to write as sequential code, not as callbacks, not as events.

That's because I think of them sequentially.

## Principles and guidance

\* The pattern here is that if our code feels like a state-machine,

going to one state after another, then awaiting is easier.

\* Let's see how that works in practice

## OnPointerPressed

\* Here's my implementation of that ideal architecture diagram.

\* This is the OnPressed handler.

\* 1., I'll await for a drag operation to complete.

DragAsync a helper I wrote for this game to tame move/released events.

\* 2. If the apple gets dragged to the bucket's mouth, then I'll

kick off sound and animation as it whooshes inside, and wait until

both sound and animation have finished.

\* 3. if last apple remaining, I'll animate in the victory screen with a storyboard

\* 4. And then I'll await until the user clicks the PlayAgain button.

And I'll trigger the OnStart event handler, fire-and-forget.

\* It's beautiful, isn't it!

\* I'm going to show you how I wrote PlayAsync for this game.

## Await Storyboard.PlayAsync

\* The thing is, storyboards are all about events.

\* I wanted to be able to write “await storyboard.PlayAsync()”.

\* I wanted to wrap up the event as a task, so I could await it.

[CLICK]

\* I know some of you aren’t as familiar with VB syntax, so here’s the same in C# syntax.

\* This is the method. Let's step through.

[CLICK]

\* When someone awaits Storyboard.PlayAsync, it invokes this method,

and runs through.

\* We create this tcs, TaskCompletionSource.

\* You can’t just new up a Task. Newing up a task is used to run it on the threadpool.

\* We don’t want that. We’re not running any code in our task.

We just want a plain task object, basically a promise, or a signal.

It can be in one of two states: starts in NotYetCompleted, and

we can make it transition into the Completed state.

\* We sign up a lambda for storyboard's the Completed event.

\* We kicks off the storyboard, sb.Begin()

[CLICK]

\* Then we await until the task we created completes.

\* Has it completed? Not yet, so we'll return for now to our caller.

[CLICK]

\* Some time later, the storyboard will finish.

\* It'll invoke the Completed handler that we signed up.

\* And this handler will transition our tcs task into the Completed state.

[CLICK]

\* You know what that means.

\* It means that our await will now finish.

[CLICK]

\* That's our opportunity to remove event-handler that we signed up,

\* and complete our async method.

\* And once we've completed, then anyone who was awaiting us will be able to resume.

\* That's it!

\* Now you might think I’ve pulled a bait-and-switch. You might think I’m still using events, callbacks, lambdas.

\* That’s true. As I said, events will be with us always.

\* But what we’ve done is encapsulate them in just 9 lines of small, easy-to-audit code that an expert writes only once. The rest of the program doesn’t need lambdas, or events, or callbacks. The rest of the program is all sequential.

\* And we can use exactly this technique anytime we want to wrap events up with a Task, so that someone can await them.

## Awaiting Button.WhenClicked()

\* Here, I wanted to show another example of wrapping up events.

\* Here, I want to support the syntax “await button1.WhenClicked()”

\* I’m using exactly the same technique. And pretty much the same code.

\* It’s a common pattern that you’ll use again and again.

## Would you like to know more?

\* There are some other nice wrappings we can do with TaskCompletionSource

\* How to wrap PointerMoved/PointerReleased into just “await DragAsync”

\* How to wrap Dispatcher.RunAsync so it works properly with Task-returning async lambdas.

\* Also, how to kick of the wobble animations but then stop them as soon as a sound has finished.

\* Again I’ve left them as hidden slides, so if you download the slide deck you can read more.

## -> DragAsync

\* This one’s a bit more complicated.

\* What’s interesting is that it’s dealing with two events, not just one:

PointerMoved and PointerReleased.

\* But still, follows exactly the same pattern.

\* In WinRT, also have to listen for PointerCancelled and PointerCaptureLost.

## -> DispatcherRunAsync

\* Remember the problem with the built-in Dispatcher.RunAsync is that it didn’t work well with async lambdas. That’s because it treated them as void-returning asyncs, not as Task-returning.

\* Here we fixed that.

\* Subtle thing is: we don’t even need a TaskCompletionSource to invent a new Task. That’s because asyncFunc parameter already gives us a Task!

## -> OnResume

\* Let's see a little bit more about how we implemented the interactive UI, with await

\* When you resume the game, remember, it plays a sound, and wobbles the apples

\* Once the sound stops, then it stops wobbling the apples.

\* So, we'll kick off the sound.

\* Then we'll kick off each apple wobbling.

\* I wrote this extension method, "FireAndForget", because even though AnimateAppleWobbleAsync

returned a task, this time I really don't need to wait until it's finished, and

I'm confident it won't throw an extension.

\* I could have turned AnimateAppleWobbleAsync into an async void method.

\* But that's just bad practice.

\* Instead I wrote this extension method, as a kind of self-documenting style,

so everyone reading the code knows that I deliberately want to fire-and-forget.

\* Well, then I await until the sound finishes.

\* And at that time I cancel all the apple-wobbling.

\* This CancellationTokenSource, from .NET4, is how we do cancellation of async methods.

## Principles and Guidance

\* So what we've seen here is a very general technique

to wrap up event-based things into Task-returning async methods

\* We used it to make our interactive game easier to architect.

\* And whenever we see our event-based code starts to feel like a state-machine,

that's a good time to think whether awaiting for events might be easier.

\* And can turn so many things into awaitable Task w/ TaskCompletionSource.

# 3. Threads and databinding

\* Now the final of three sections of this talk.

\* I want to talk about threads and databinding, about IO- and CPU-bound workloads.

\* Let's hear what the customer had to say. He said:

[CLICK]

**\* "I'm now looking at the biggest user complaint about a slow running operation in an ASP.NET WebForms page."**

**\* "Essentially, the page loads some data and I'm wondering if it'd be the best approach to use the Task Parallel Library."**

**\* "The method itself deserializes an object and depending on user choices can call the method in a foreach 26+ times, the result of which I bind to a gridpanel."**

**\* "The deserialization itself is where 99% of the time is being spent."**

\* Well, that's brilliant! He used profiling first. He identified the problem area.

\* The punchline is that is code turned out not to be CPU-bound, and so he should have been using await.

## LoadHousesSequentially

\* But let’s look at what the customer started with.

\* It's a zillow-like housing app. He's deserializing a load of houses,

and databinding them to his webform.

\* And we’ll start by taking him at his word that the deserialization work is CPU-bound.

[CLICK]

\* If we draw a flow chart of it, the request comes in, then it does

one house after another, and then finishes.

\* If each house takes 100ms to deserialize, and he does five houses, then

it'll be 500ms before the user sees anything in his web-browser.

## LoadHousesInParallel

\* This is what the customer tried using the TaskParallelLibrary.

\* He used Parallel.For, to deserialize all the houses in parallel.

\* This lambda is the work for each house that has to be done.

[CLICK]

\* Let's draw a flow-chart for it.

\* First a request comes in.

\* Then he does Parallel.For, which means that five lambdas will have to be executed eventually.

\* Then the threadpool does those five pieces of work.

\* The threadpool will run them on as many threads as will be fastest.

\* My laptop has two CPU cores -- a boy and a girl, you can see, so two cores will be fastest.

\* If each request takes 100ms, then we'll get the answer out in 300ms.

That's an improvement!

\* Actually, it might not be. If we're running on a server that has other workload as well,

then one of the cores will probably be taken, so we'll only have one.

## Is it CPU-bound or IO-bound?

\* Oh. Just hold on there a moment.

\* What the heck kind of deserialization takes so long? 100ms per house? That's an eternity.

\* Well, I checked with the customer.

\* Turns out his deserialization wasn't really what I'd call deserialization.

\* It was looking up tables in a database.

\* That's why it took so long. It was network-bound, not CPU-bound.

## Sequential diagram

\* So this is what his first sequential code was actually doing.

\* It was downloading data for each house, one after the other.

\* But it only took a miniscule amount of time to kick off each request,

then it was idle for about 100ms,

then it got back the response from the network.

## Parallel.For

\* But let's look back at how his Parallel.For code was behaving.

\* Well, as we said, it had five workitems in the threadpool.

\* Let's say the threadpool started with two threads, because of my two cores.

[CLICK]

\* Gradually it'll realize that its threads aren't really being used,

and it'll add an extra thread to do some more work.

[CLICK]

\* Maybe an extra thread as well.

\* The threadpool will gradually find the optimum number of threads to run

a given workload, but it's fairly slow to respond.

[CLICK]

\* In this case maybe it only ended up growing by two extra threads.

\* Well, this result came in about 200ms.

\* In general, threadpool growth isn't the right way to get responsive code.

\* That's because it does take time to get there.

\* Sometimes you'll see it adding just one new thread a second.

## Async diagram

\* Let's draw a flow diagram about how this code should ideally work.

\* We should kick off all five requests in one go.

\* We might as well issue the requests in sequence, since it's so quick to issue a request.

\* Later on, about 100ms later, the responses trickle in.

\* They might come out of order. That doesn't matter. We'll get them all.

\* And we should have them all done within about 100ms.

\* That's the fastest "Time To First Byte" of all our solutions.

## Principles and guidance

\* So let's review.

\* It's vital to distinguish between what is CPU-bound work and what is IO-bound work.

\* CPU-bound means things like LINQ-to-objects, or iterations, or computationally-intensive inner loops.

\* Parallel.ForEach and Task.Run are good ways to put these CPU-bound workloads on the threadpool.

\* But it's important to understand that threads haven't increased throughput

\* Throughput means “what’s the highest number of requests per second we can handle?”

\* Each CPU-bound work item still takes a given number of milliseconds on a processor core, and that’s the bottom line, and that’s the limiting factor on throughput of CPU-bound work.

\* As for IO-bound work, well, threadpool did help some, but it didn’t respond quickly to load. The best solution is async.

## LoadHousesAsync

\* Back to the customer's scenario.

\* This is the code that the customer should have used

\* He can kick off tasks for all the database loads.

\* And then await Task.WhenAll, until they're all finished.

## btnPayout

\* Well, that code wasn't CPU-bound.

\* But I want to show you an example of best-practice with CPU-bound code.

\* To come clean, this is the only example in this talk that didn't come from customers.

\* Well, it's a desktop app for doing financial analysis. Option-pricing.

\* So what financial firms do is they simulate a million different ways

that the stock might move over the coming year, each equally likely.

That's the function Quant.SimulateStockPrices.

\* They figure whether the option will be a good deal or a bad deal over all those million possibilities, and how much it will payout on each.

That's the function Quant.Payout\_AsianCallOption.

[CLICK]

\* Actually, I did the numerical back-end code in F#.

I think you'll be seeing more and more of this, the front-end UI in VB or C#,

and the back-end in F#, because analysists and rocket-scientists can use F# really effectively.

[CLICK]

\* Back to the code.

\* It's clearly very CPU-bound. It's an inner computational loop.

\* So, it \*HAS\* to go onto the threadpool.

[CLICK]

\* We have two ways to put work on the threadpool. One is Parallel.For, and one is Task.Run. I could have used either, but I’ll use Task.Run.

\* Sometimes people ask, “Why not just put it into an async method?”

\* But remember, when we make a method async, all that means is that it’s allowed to use the await keyword.

\* It certainly doesn’t mean that the thread will run on the threadpool.

\* The compiler will never implicitly run anything on the threadpool.

\* If we want it to run our CPU-bound work on the threadpool, we have to explicitly opt into it, by calling Task.Run or Parallel.For.

\* And here I chose to use Task.Run, and I gave it a lambda to run on the threadpool.

\* The best-practice here is that the body of Task.Run is as small as possible,

only for the CPU-bound bit of code.

\* It doesn't try to update the UI.

\* It merely returns the answer at the end.

\* That way, I can await for it, get the result into this expectedPayout variable,

and update the UI myself.

\* It's great that the background thread doesn't have to interact with the UI.

[CLICK]

\* Oh, in the UI, I really wanted a progress bar, to see how much it's done.

\* Once again, you generally can't update the UI from a background thread,

so this is the new standard async design pattern for it.

\* We create this variable "progress" of type IProgress, a new type in .NET45.

\* The background thread can report its progress to it.

\* And this Progress type does the magic of marshaling from the background thread

to the UI thread, so it can update the UI of the progress-bar.

\* Well, that's the standard way to get updates from the background thread onto the UI.

## Would you like to know more?

\* WPF has its own data-bound approaches to updating the UI from a background thread, including some enhancements new in WPF4.5.

\* I won’t go into them because they’re WPF-specific.

\* Again, these are hidden slides you can read if you download the deck.

## <- Databinding

\* But, you ask, can we use data-binding instead?

\* The answer is that if you're using WPF, then you can.

\* In WPF, if a background thread modifies a databound property,

then the binder itself will marshal that back to the UI.

\* In this case, the opt.ExpectedPayout property is databound to the UI,

and I can happily update it from a background thread.

\* That's easy. Alas it doesn't work on Silverlight or Phone or WinRT,

so you have to fall back to the previous solution.

[CLICK]

\* That's fine for databound properties.

\* But what about databound collections?

How about an ObservableCollection that's databound to a listbox?

\* Can you update that ObservableCollection from a background thread?

\* Well, new in WPF4.5, you can.

\* Use this new function BindingOperations.EnableCollectionSynchronization.

\* That way, if you add stuff to the collection from a background thread,

it takes care of marshaling over to the UI thread to make the change.

\* That's the good news.

\* The bad news is that this only works in WPF4.5. For WPF4, or Silverlight/Phone/WinRT,

you have to implement it yourself. Not particularly easy.

## <- LoadHousesWithTask

\* Actually, heh, that's not quite what the customer came up with.

\* He knew he should use tasks somehow.

\* But he only had VS2010, so he couldn't use the await keyword.

\* And he came up with this.

\* I can see at a glance that it's not right. How?

\* Well, he uses Task.Factory.StartNew.

That's not like TaskCompletionSource.

Instead like Task.Run, for putting CPU-bound piece of work into threadpool.

\* Also I see he's still calling House.LoadFromDatabase.

If it's not async, then it must be blocking the thread.

\* And finally I see he used Task.WaitAll rather than await Task.WhenAll.

WaitAll, from .NET4, is the synchronous version - it blocks until all tasks have finished.

He should have used Task.WhenAll, the asynchronous version.

\* So what he achieved was to limit himself to a maximum of two threads.

Not so good!

## <- LoadState

\* If there's time, I want to show you one last sneaky example.

\* This came from a customer who was writing a WinRT app.

\* He had about 600 JPEGs on disk, each geotagged with longitude and latitude.

\* Because you can store geotag information inside the JPEG itself.

\* On a map, he wanted to draw an ellipse for the geographical location of each JPEG.

\* Well, that should be easy.

\* Is loading files off disk an IO-bound or a CPU-bound operation?

\* It's IO-bound! So he doesn't need any background threads.

\* And that's just as well because he wanted to use ObservableCollection databinding

to render those ellipses.

\* So this is the code, and it looks great, and best-practice.

\* Let's take a look at what happened.

## <- DEMO

\* Launch the Map app. Move cursor around.

\* It's loading files off disk in the background.

\* Suddenly, there's a 2-second hang of the UI!

\* Repeat.

## <- LoadState

\* So what happened?

\* Well, there was CPU-bound work hiding there.

\* First, it takes some CPU just to resume an await on the UI thread.

\* In WinRT, the guidance is that just a hundred or so awaits resuming on the UI thread per second

will be fine, but a thousand per second will be bad.

It wasn't designed for that.

\* Also, with .NET on WinRT, just kicking off an asynchronous WinRT thread takes some CPU overhead when it kicks off, and again when it resumes. Just a couple of ms.

\* Normally these are insignificant.

\* But when 600 WinRT async operations all finish in quick succession,

and each of them takes 2ms of CPU time, then it adds up.

\* The moral here is that if you're just doing sequential async,

awaiting async APIs immediately as you invoke them, then you'll be fine.

\* But when you start kicking off a large number of work items in parallel,

then you might face stealth CPU-loads.

## Principles & Guidance

\* Let's recap.

\* We saw that it's crucial to figure out whether your workload is CPU-bound or IO-bound.

\* For IO-bound stuff, we used await to manage it, and we didn't need background threads (unless there was just a huge number of async operations).

\* For CPU-bound stuff, we saw best practice in using Task.Run to put it on the threadpool.