.NET Conference 2016
Spain

Microsoft

#dotNetSpain2016

//async Best Practices

Lluis Franco
C# MVP & Plumber
@lluisfranco - hello@lluisfranco.com

Microsoft

#dotNetSpain2016

Microsoft

Patrocinadores

altran

avanade®

Danysoft
Haciendo visible lo invisible

DevsDNA

encamina
PIENSA EN COLORES

ilitia
technologies

Insight

intelequia
SOFTWARE SOLUTIONS

plain concepts

SOGETI

東京'TOKIOTA

Colaboradores

pue

ticjob.es

#dotNetSpain2016

About Lluis Franco...

Software Architect @ FIMARGE

Microsoft C# MVP since 2003

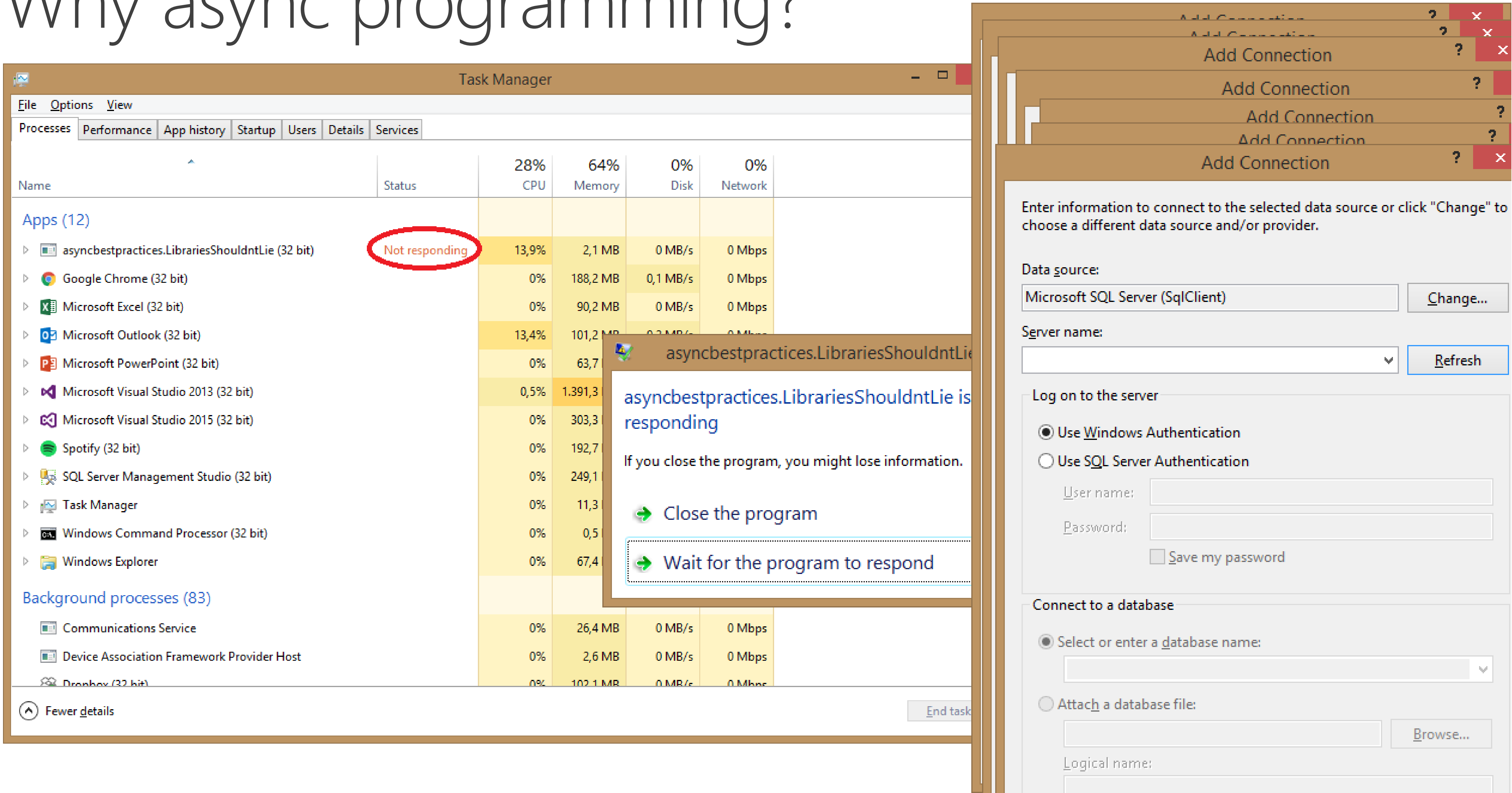
MVP of the year 2011

Founder of AndorraDotNet

Geek-a-palooza Organizer

INETA Country leader

Why async programming?



Task Manager

File Options View
Processes Performance App history Startup Users Details Services

Name	Status	28% CPU	64% Memory	0% Disk	0% Network
Apps (12)					
asyncbestpractices.LibrariesShouldntLie (32 bit)	Not responding	13,9%	2,1 MB	0 MB/s	0 Mbps
Google Chrome (32 bit)		0%	188,2 MB	0,1 MB/s	0 Mbps
Microsoft Excel (32 bit)		0%	90,2 MB	0 MB/s	0 Mbps
Microsoft Outlook (32 bit)		13,4%	101,2		
Microsoft PowerPoint (32 bit)		0%	63,7		
Microsoft Visual Studio 2013 (32 bit)		0,5%	1.391,3		
Microsoft Visual Studio 2015 (32 bit)		0%	303,3		
Spotify (32 bit)		0%	192,7		
SQL Server Management Studio (32 bit)		0%	249,1		
Task Manager		0%	11,3		
Windows Command Processor (32 bit)		0%	0,5		
Windows Explorer		0%	67,4		
Background processes (83)					
Communications Service		0%	26,4 MB	0 MB/s	0 Mbps
Device Association Framework Provider Host		0%	2,6 MB	0 MB/s	0 Mbps
Dropbox (32 bit)		0%	102,1 MB	0 MB/s	0 Mbps

asyncbestpractices.LibrariesShouldntLie

asyncbestpractices.LibrariesShouldntLie is responding

If you close the program, you might lose information.

Add Connection

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source:
Microsoft SQL Server (SqlClient)

Server name:

Log on to the server

● Use Windows Authentication
○ Use SQL Server Authentication

User name:
Password:
□ Save my password

Connect to a database

● Select or enter a database name:
○ Attach a database file:

Logical name:

Key Takeaways

Evolution of the async model

Async void is only for top-level event handlers.

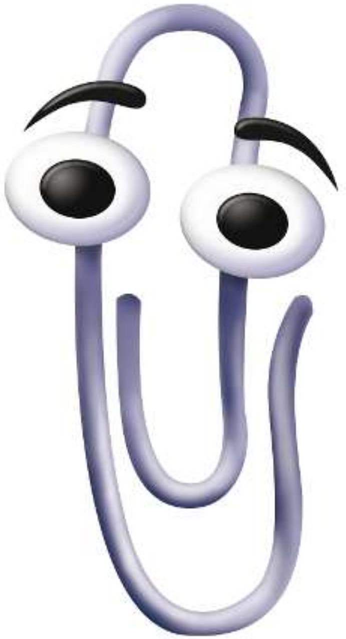
Use the threadpool for CPU-bound code, but not IO-bound.

Libraries shouldn't lie, and should be chunky.

Micro-optimizations: Consider ConfigureAwait(false)

Evolution of the async model

Evolution of the async model



It seems you’re calling an async method without awaiting…

Can I help you?

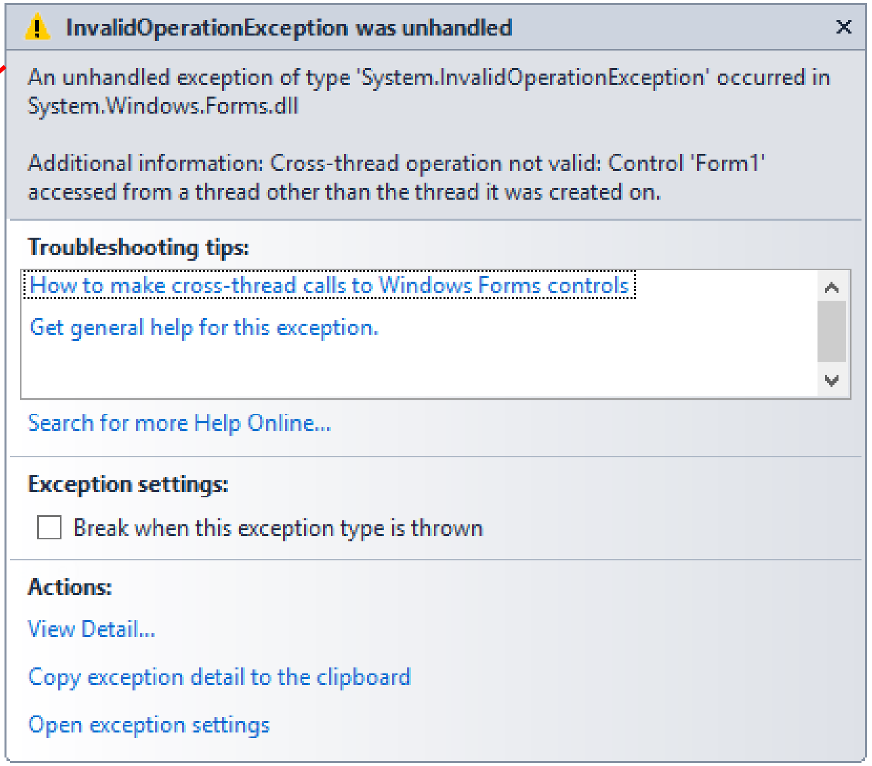
- Yep! Return a Task object plz
- Nope. Maybe latter.

Clippy-driven development

Thread class

```
private async void Button1_Click(object Sender, EventArgs e) {
    Thread oThread = new Thread new ThreadStart(myExpensiveMethod));
    oThread.Start();
    ...
  ● oThread.Abort();
    ...
    if(oThread.IsAlive) {
        ...
    }
}

private static void myExpensiveMethod() {
    //Some expensive stuff here...
    //Read from Database/Internet
    //Perform some calculations
  ● salaryTextBox.Text = result;
}
```



InvalidOperationException was unhandled

An unhandled exception of type 'System.InvalidOperationException' occurred in System.Windows.Forms.dll

Additional information: Cross-thread operation not valid: Control 'Form1' accessed from a thread other than the thread it was created on.

Troubleshooting tips:

How to make cross-thread calls to Windows Forms controls

Get general help for this exception.

Search for more Help Online...

Exception settings:

□ Break when this exception type is thrown

Actions:

View Detail...

Copy exception detail to the clipboard

Open exception settings

#dotNetSpain2016

ThreadPool

```
private async void Button1_Click(object Sender, EventArgs e) {

    //Thread oThread = new Thread(new ThreadStart(myExpensiveMethod));
    //oThread.Start();

    ThreadPool.QueueUserWorkItem(p => myExpensiveMethod());



}

private static void myExpensiveMethod() {
    //Some expensive stuff here...
    //Read from Database/Internet
    //Perform some calculations
    if (salaryTextBox.InvokeRequired)
        salaryTextBox.Invoke(new Action(() => salaryTextBox.Text = result));
}
```

Update UI from other Thread

IAsyncResult

```
private int myExpensiveMethod()
{
    ...
    return 42; //the answer to the life the universe and everything
}

private void Button1_Click(object sender, EventArgs e)
{
    var function = new Func<int>(myExpensiveMethod);
    IAsyncResult result = function.BeginInvoke(whenFinished, function);
}

private void whenFinished(IAsyncResult ar)
{
    var function = ar.AsyncState as Func<int>;
    int result = function.EndInvoke(ar);
    resultTextBox.Text = string.Format("The answer is... {0}!", result);
}
```

Task Parallel Library

Highlights

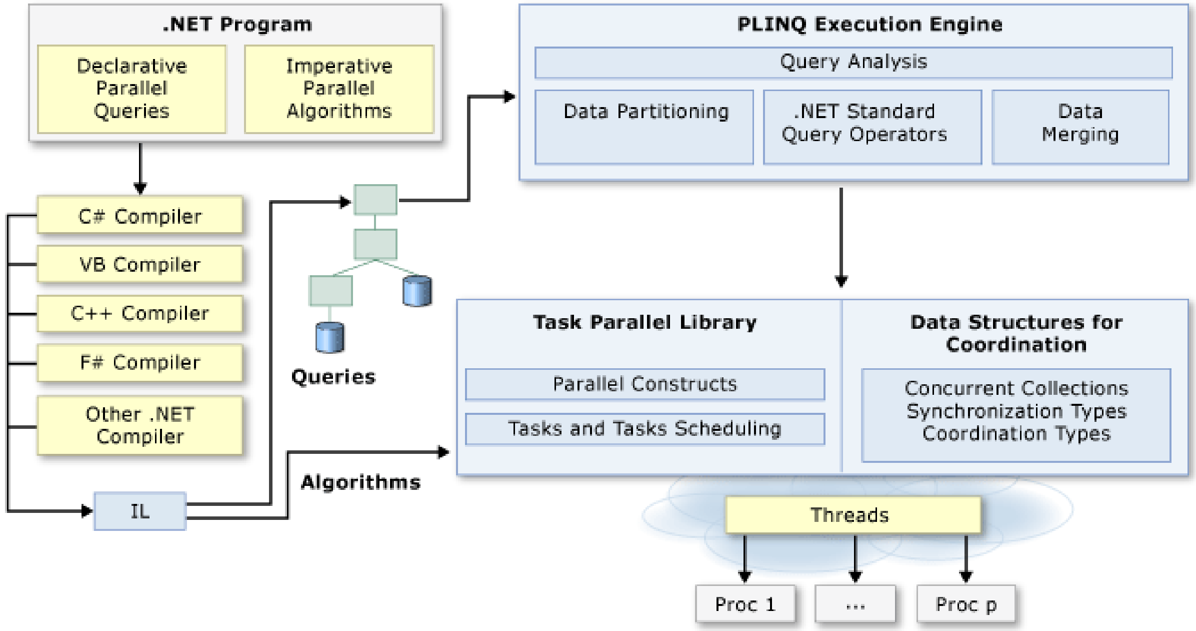
- New in .NET 4.0 (Visual Studio 2010)
- High level: We talk about Tasks, not Threads.
- New mechanisms for CPU-bound and IO-bound code (PLINQ, Parallel & Task class)
- Cancellations with token, Continuations and Synchronization between contents

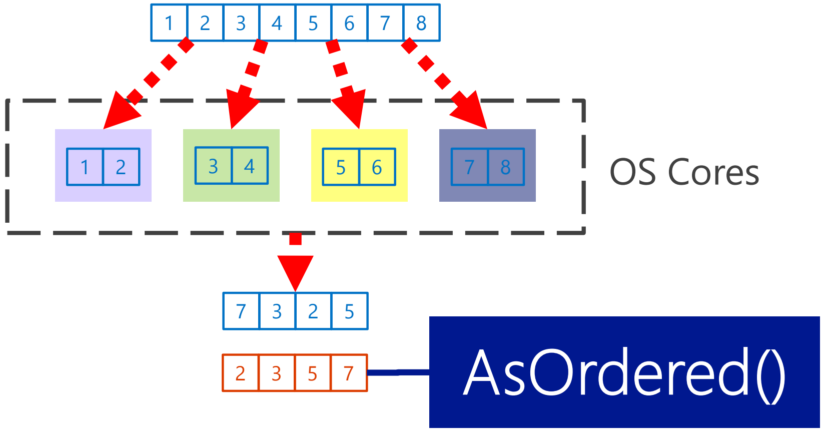
Task class

- Code that can be executed asynchronously
- In another thread? It doesn’t matter…

Task Parallel Library - PLINQ

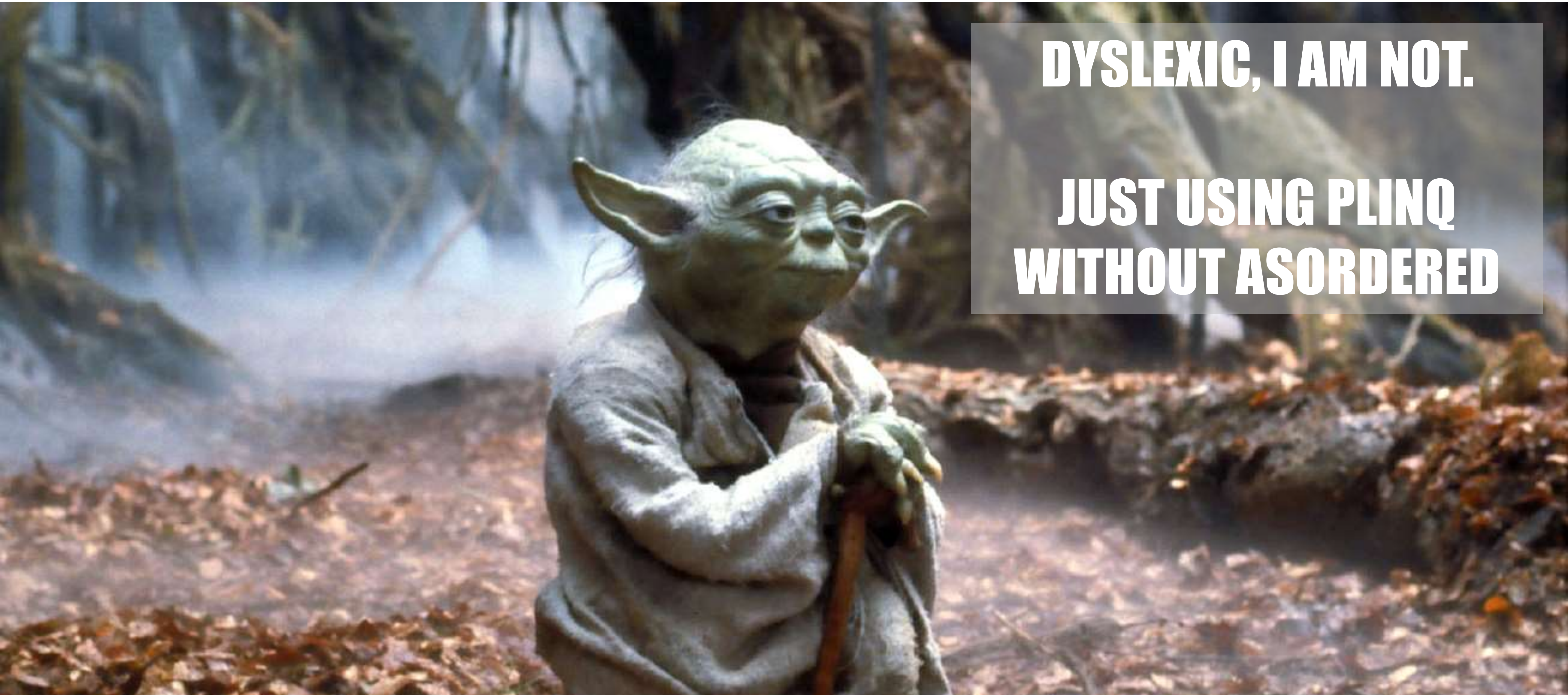
```
var numbers = Enumerable.Range(1, 10000000);
var query = numbers.AsParallel().Where(n => n.IsPrime());
var primes = query.ToArray();
```





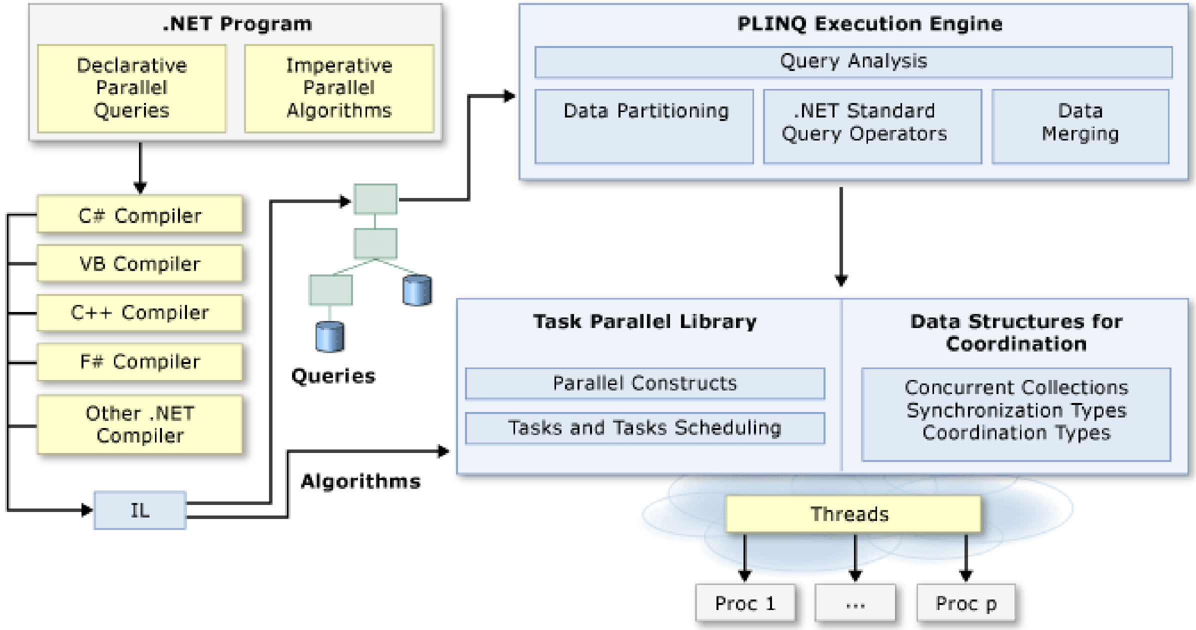
#dotNetSpain2016

“Judge me by my size, do you?”



Task Parallel Library – Parallel class

```
var customers = Customer.GetSampleCustomers();
Parallel.ForEach(customers, c => {
    if(!c.IsActive) c.Balance = 0;
});
```



PLINQ and Parallel are not Async!

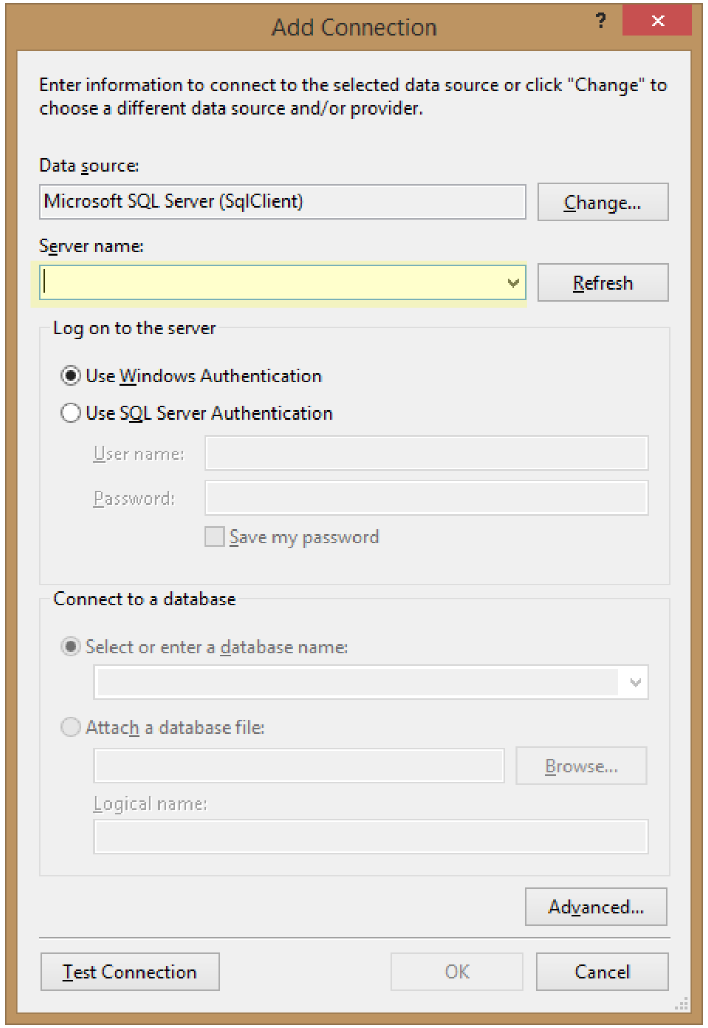
#dotNetSpain2016

Task Parallel Library - Task class (1)

```
public static List<string> GetNetworkSQLServerInstances() {
    //Get local network servers
    return servers;
}

private void updateServersList(List<string> severs) {
    comboBox1.Items.AddRange(servers.ToArray());
}

//SYNC version
private void Button1_Click(object sender, EventArgs e) {
    var servers = GetNetworkSQLServerInstances();
    updateServersList(servers);
}
```



Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: Microsoft SQL Server (SqlClient)

Server name:

Log on to the server

● Use Windows Authentication

○ Use SQL Server Authentication

User name:

Password:

□ Save my password

Connect to a database

● Select or enter a database name:

○ Attach a database file:

Logical name:

#dotNetSpain2016

Task Parallel Library - Task class (2)

```
public static List<string> GetNetworkSQLServerInstances() {
    //Get local network servers
    return servers;
}

private void updateServersList(List<string> severs) {
    comboBox1.Items.AddRange(servers.ToArray());
}

//ASYNC version
private void Button1_Click(object sender, EventArgs e) {
    var serversTask = Task.Factory.StartNew(() => GetNetworkSQLServerInstances());
    serversTask.ContinueWith(t => updateServersList(serversTask.Result));
}
```

Task Continuations

Task Parallel Library - Task class (3)

```
public static List<string> GetNetworkSQLServerInstances() {
    //Get local network servers
    return servers;
}

private void updateServersList(List<string> severs) {
    comboBox1.Items.AddRange(servers.ToArray());
}

//ASYNC version + context synchronization
private void Button1_Click(object sender, EventArgs e) {
    var serversTask = Task.Factory.StartNew(() => GetNetworkSQLServerInstances());
    serversTask.ContinueWith(t => updateServersList(serversTask.Result),
        TaskScheduler.FromCurrentSynchronizationContext());
}
```

Update UI from Task

async/await

```
public static List<string> GetNetworkSQLServerInstances() {
    //Get local network servers
    return servers;
}

private void updateServersList(List<string> severs) {
    listBox1.Items.AddRange(servers.ToArray());
}

//ASYNC/AWAIT version
private async void Button1_Click(object sender, EventArgs e) {
    var servers = await Task.Run(() => GetNetworkSQLServerInstances());
    updateServersList(servers);
}
```

Async MAGIC!

async/await

Highlights

Syntax sugar for invoking and chaining Tasks.

Pros

Mega-Easy syntax
Without callbacks
Allows update UI

Cons

More overhead than sync methods
Typically the overhead is negligible… this talk is about when it isn’t

async/await REAL

```
public static async Task<List<string>> GetNetworkSQLServerInstancesAsync() {
    //Get local network servers using async APIs
    return servers;
}

//ASYNC/AWAIT version 1 (previous)
private async void Button1_Click(object sender, EventArgs e) {
    var servers = await Task.Run(() => GetNetworkSQLServerInstances());
    updateServersList(servers);
}

//ASYNC/AWAIT version 2 (REAL async)
private async void Button1_Click(object sender, EventArgs e) {
    var servers = await GetNetworkSQLServerInstancesAsync();
    updateServersList(servers);
}
```

async/await

```
//ASYNC/AWAIT version
private async void Button1_Click(object sender, EventArgs e) {
    var servers = await GetNetworkSQLServerInstancesAsync();
    updateServersList(servers);
}
```

async

↓ FALSE ‘This method is asynchronous’

↑ TRUE ‘In this method we will call asynchronous methods’

await

↓ FALSE ‘Call this asynchronous method and wait until the method ends’

↑ TRUE ‘Call this method and return control immediately to the caller, when the method ends, continue the execution from that point’

#dotNetSpain2016

Mental Model (sync)

We all “know” sync methods are “cheap”

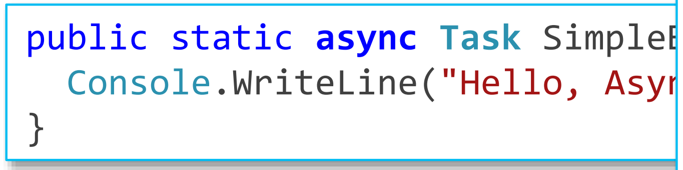
- Years of optimizations around sync methods
- Enables refactoring at will

```
public static void SimpleBody() {
  Console.WriteLine("Hello, Async World!");
}
```

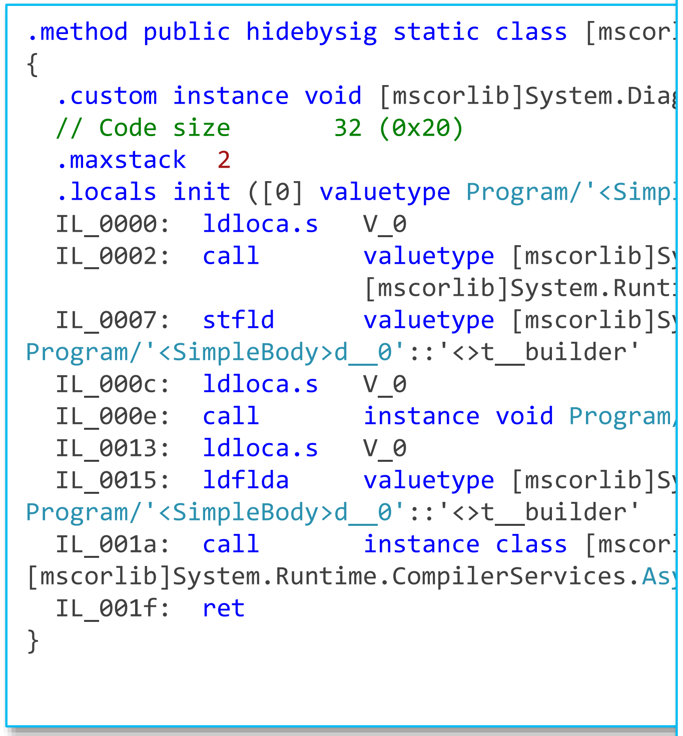
```
.method public hidebysig static void SimpleBody() cil managed
{
    .maxstack 8
    L_0000: ldstr "Hello, Async World!"
    L_0005: call void [mscorlib]System.Console::WriteLine(string)
    L_000a: ret
}
```

Mental Model (async)

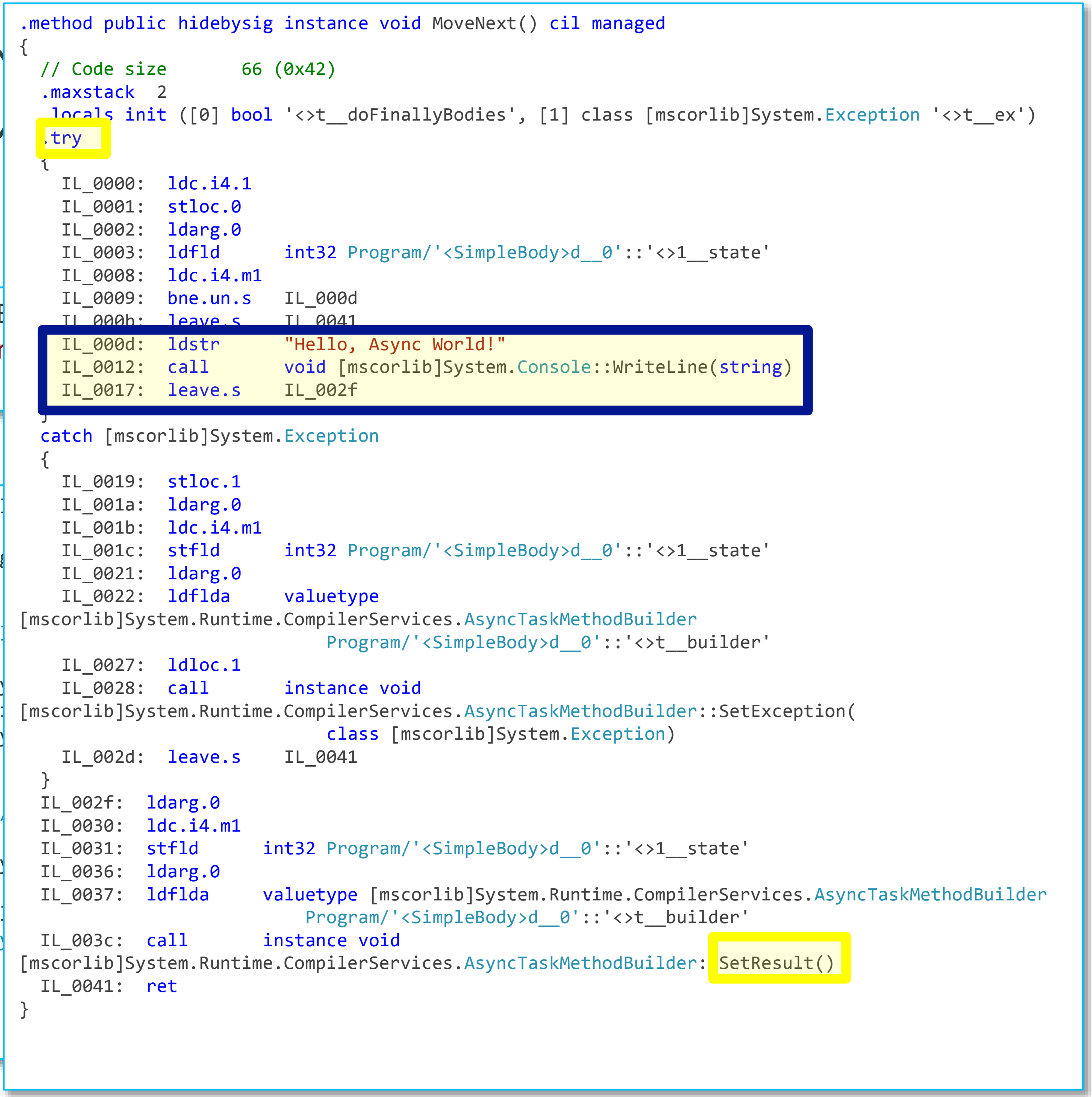
Not so for asynchronous methods



```
public static async Task SimpleB
  Console.WriteLine("Hello, Asyn
}
```



```
.method public hidebysig static class [mscor
{
  .custom instance void [mscorlib]System.Diag
  // Code size       32 (0x20)
  .maxstack  2
  .locals init ([0] valuetype Program/'<Simp
  IL_0000:  ldloca.s   V_0
  IL_0002:  call       valuetype [mscorlib]Sy
                       [mscorlib]System.Runti
  IL_0007:  stfld      valuetype [mscorlib]Sy
Program/'<SimpleBody>d__0'::'<>t__builder'
  IL_000c:  ldloca.s   V_0
  IL_000e:  call       instance void Program/
  IL_0013:  ldloca.s   V_0
  IL_0015:  ldflda     valuetype [mscorlib]Sy
Program/'<SimpleBody>d__0'::'<>t__builder'
  IL_001a:  call       instance class [mscor
[mscorlib]System.Runtime.CompilerServices.Asy
  IL_001f:  ret
}
```

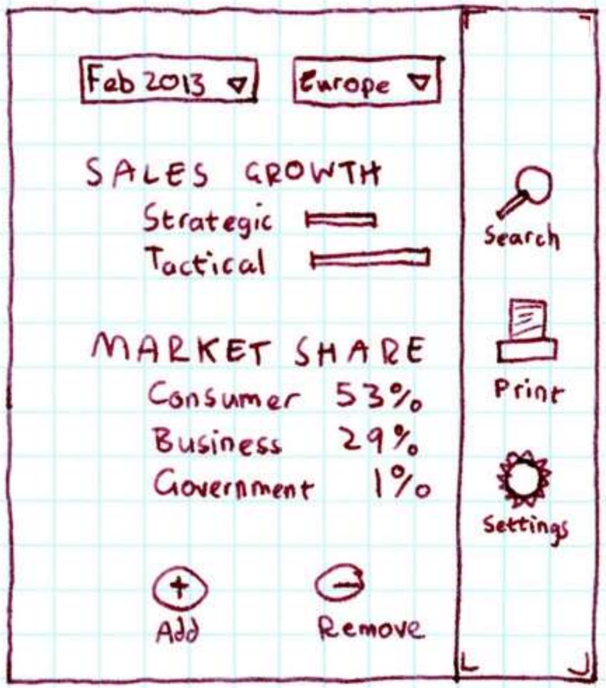


```
.method public hidebysig instance void MoveNext() cil managed
{
  // Code size       66 (0x42)
  .maxstack  2
  .locals init ([0] bool '<>t__doFinallyBodies', [1] class [mscorlib]System.Exception '<>t__ex')
  .try
  {
    IL_0000:  ldc.i4.1
    IL_0001:  stloc.0
    IL_0002:  ldarg.0
    IL_0003:  ldfld      int32 Program/'<SimpleBody>d__0'::'<>1__state'
    IL_0008:  ldc.i4.m1
    IL_0009:  bne.un.s   IL_000d
    IL_000b:  leave.s    IL_0041
    IL_000d:  ldstr      "Hello, Async World!"
    IL_0012:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_0017:  leave.s    IL_002f
  }
  catch [mscorlib]System.Exception
  {
    IL_0019:  stloc.1
    IL_001a:  ldarg.0
    IL_001b:  ldc.i4.m1
    IL_001c:  stfld      int32 Program/'<SimpleBody>d__0'::'<>1__state'
    IL_0021:  ldarg.0
    IL_0022:  ldflda     valuetype
[mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder
                           Program/'<SimpleBody>d__0'::'<>t__builder'
    IL_0027:  ldloc.1
    IL_0028:  call       instance void
[mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder::SetException(
                           class [mscorlib]System.Exception)
    IL_002d:  leave.s    IL_0041
  }
  IL_002f:  ldarg.0
  IL_0030:  ldc.i4.m1
  IL_0031:  stfld      int32 Program/'<SimpleBody>d__0'::'<>1__state'
  IL_0036:  ldarg.0
  IL_0037:  ldflda     valuetype [mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder
                         Program/'<SimpleBody>d__0'::'<>t__builder'
  IL_003c:  call       instance void
[mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder::SetResult()
  IL_0041:  ret
}
```

DEMO

Sync vs Async
Method Invocation Overhead

Async void is only for event handlers



User:

“It mostly works, but not 100% reliably.”

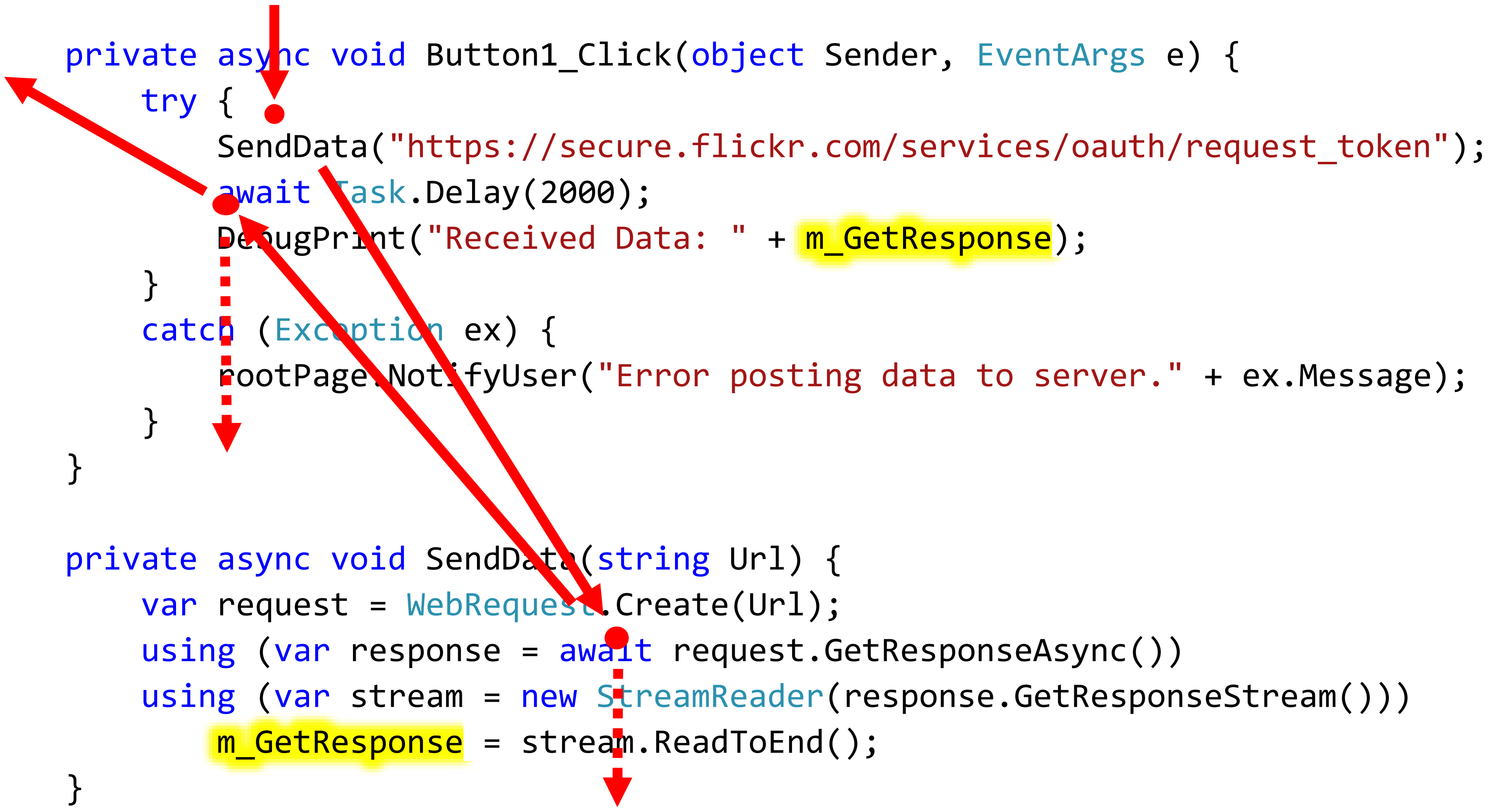
Diagnosis & Fix:

Probably was using async void.

Should return Task not void.

For goodness’ sake, stop using async void!

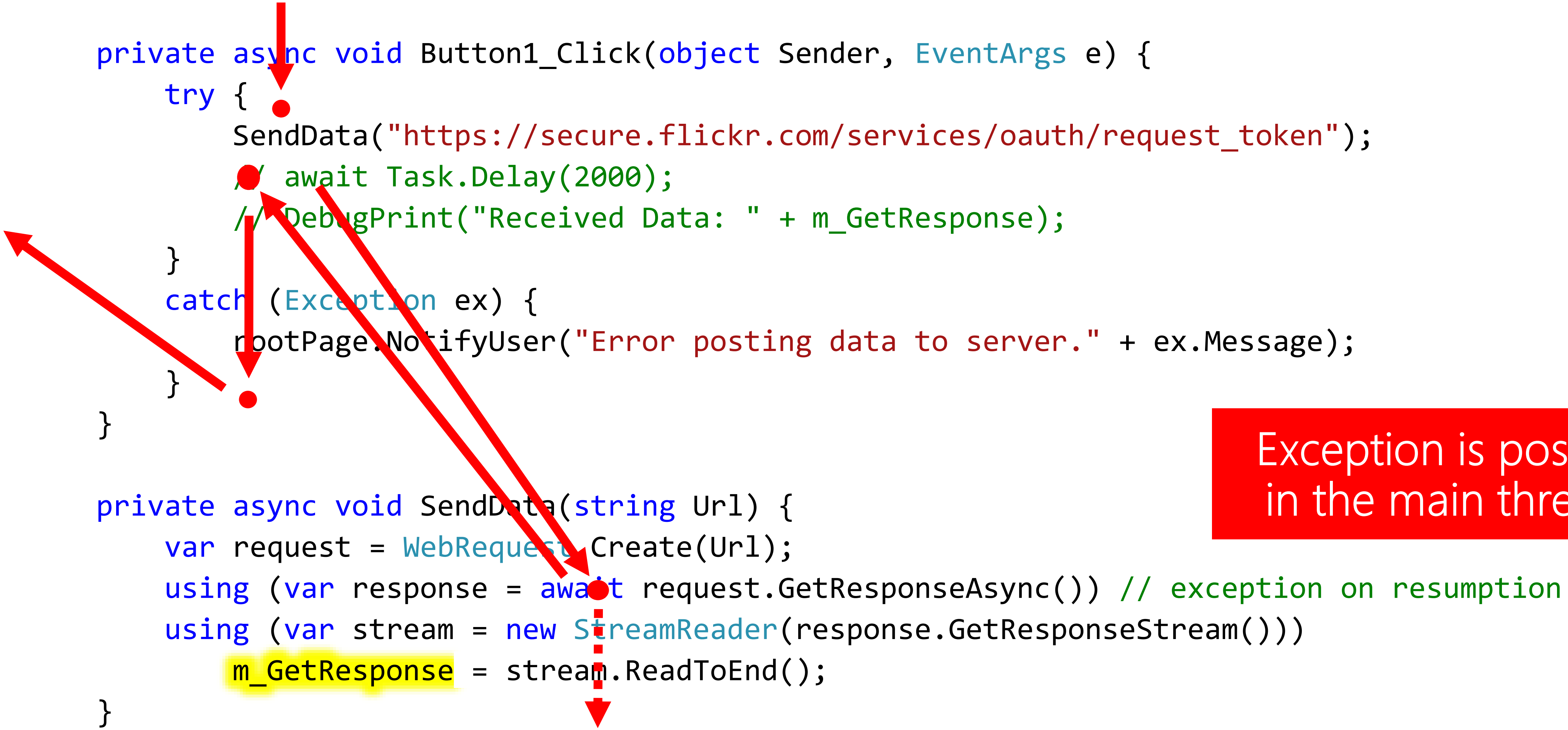
Async void is only for event handlers



```
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();
}
```

Async void is only for event handlers



```
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();
}
```

Exception is posted in the main thread

#dotNetSpain2016

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

If you need fire-and-forget elsewhere, indicate it explicitly e.g. “FredAsync().FireAndForget()”

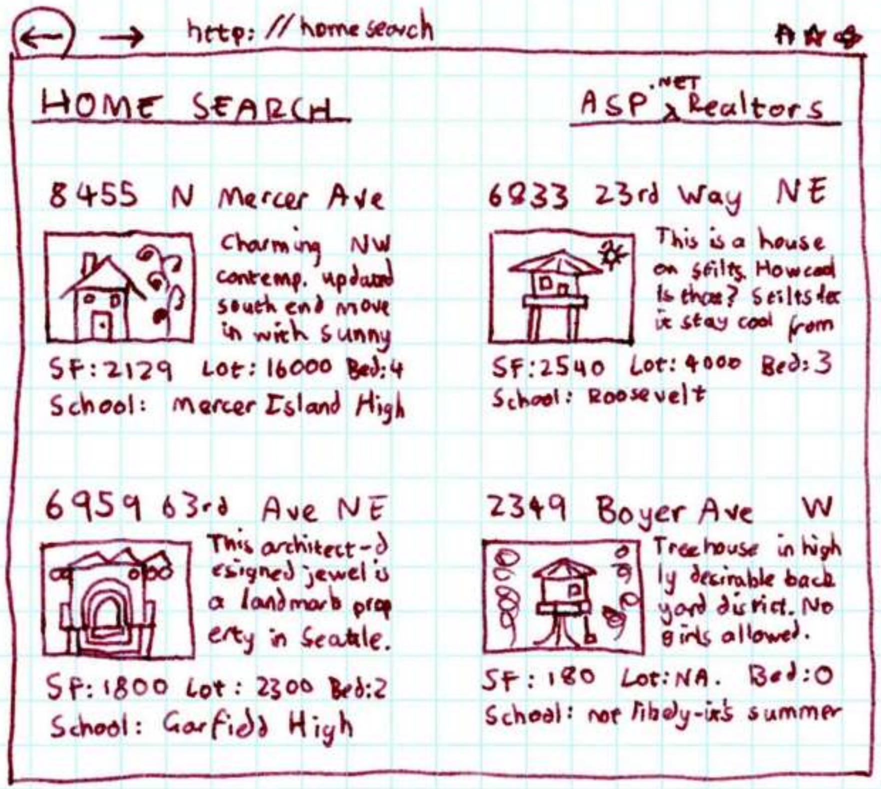
Async void is only for event handlers

```
private async void Button1_Click(object Sender, EventArgs e) {
    try {     Async
   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);
    }
}

              Task      Async
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();
}
```

#dotNetSpain2016

Threadpool


http: //home search
HOME SEARCH
ASP.NET Realtors
8455 N Mercer Ave
Charming NW contemp. updated south end move in with sunny
SF:2129 Lot: 16000 Bed:4
School: Mercer Island High
6833 23rd Way NE
This is a house on stilts. How cool is that? Stilts let it stay cool from
SF:2540 Lot: 4000 Bed:3
School: Roosevelt
6959 63rd Ave NE
This architect-designed jewel is a landmark property in Seattle.
SF:1800 Lot: 2300 Bed:2
School: Garfield High
2349 Boyer Ave W
Treehouse in highly desirable backyard district. No girls allowed.
SF: 180 Lot:NA. Bed:0
School: not likely-it's summer

User:

"How to parallelize my code?"

Diagnosis & Fix:

User's code was not CPU-bound: should use await, not Parallel.For.

#dotNetSpain2016

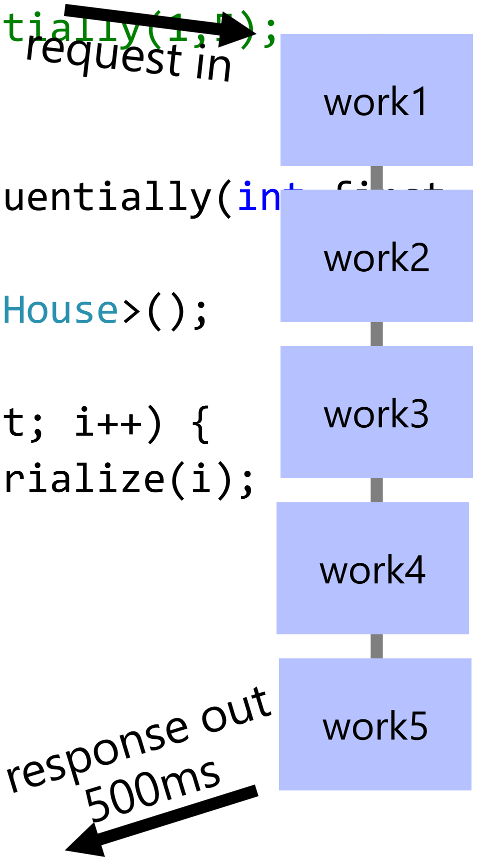
Threadpool

```
// var houses = LoadHousesSequentially(1,5);
// Bind houses to UI control;

public List<House> LoadHousesSequentially(int first, int last)
{
    var loadedHouses = new List<House>();

    for (int i = first; i <= last; i++) {
        House house = House.Deserialize(i);
        loadedHouses.Add(house);
    }

    return loadedHouses;
}
```



#dotNetSpain2016

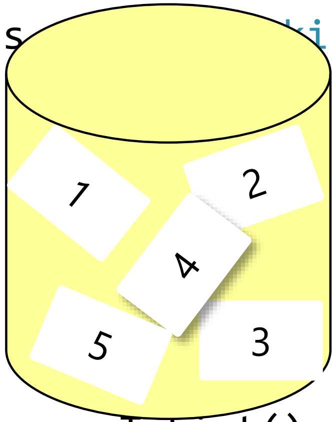
Threadpool

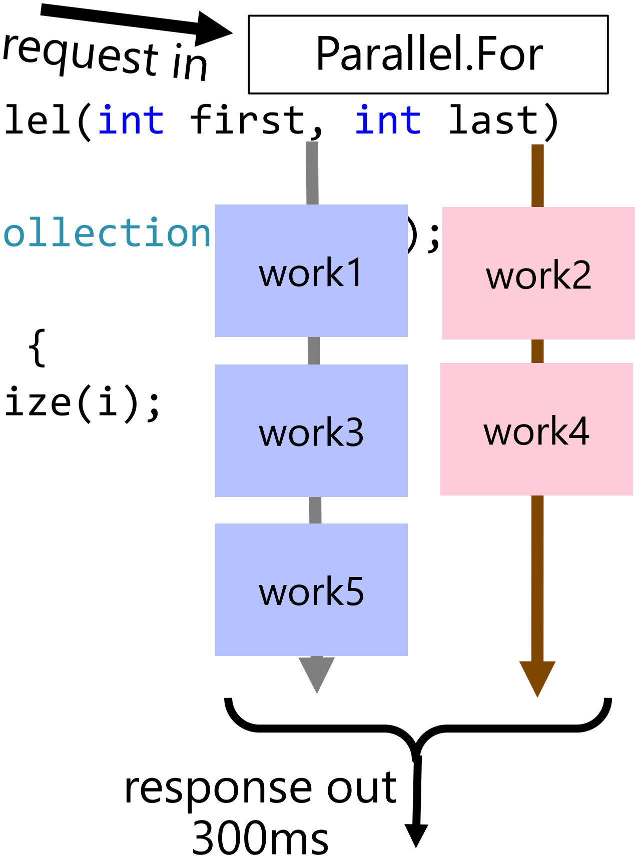
```
// var houses = LoadHousesInParallel(1,5);
// Bind houses to UI control;

public List<House> LoadHousesInParallel(int first, int last)
{
    var loadedHouses              ingCollection          );

    Parallel.For(fi                          => {
        House house                   ialize(i);
        loadedHouse
    });

    return loadedHouses.ToList();
}
```



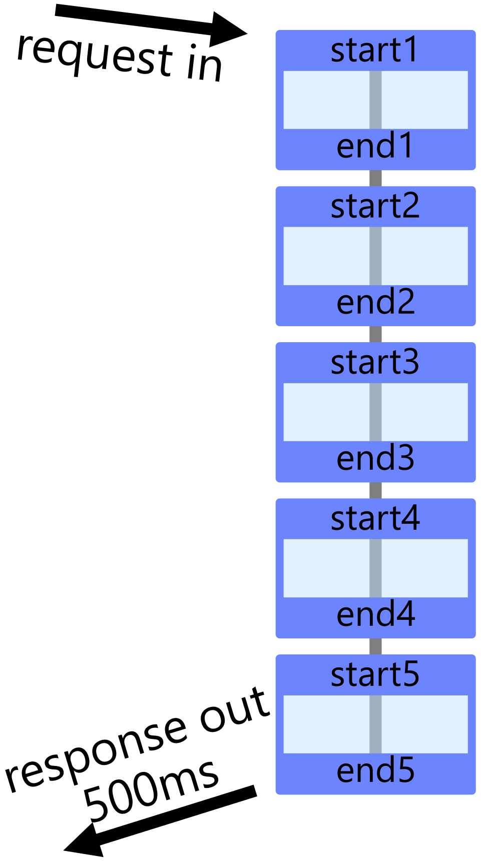


Parallelization hurts Scalability!

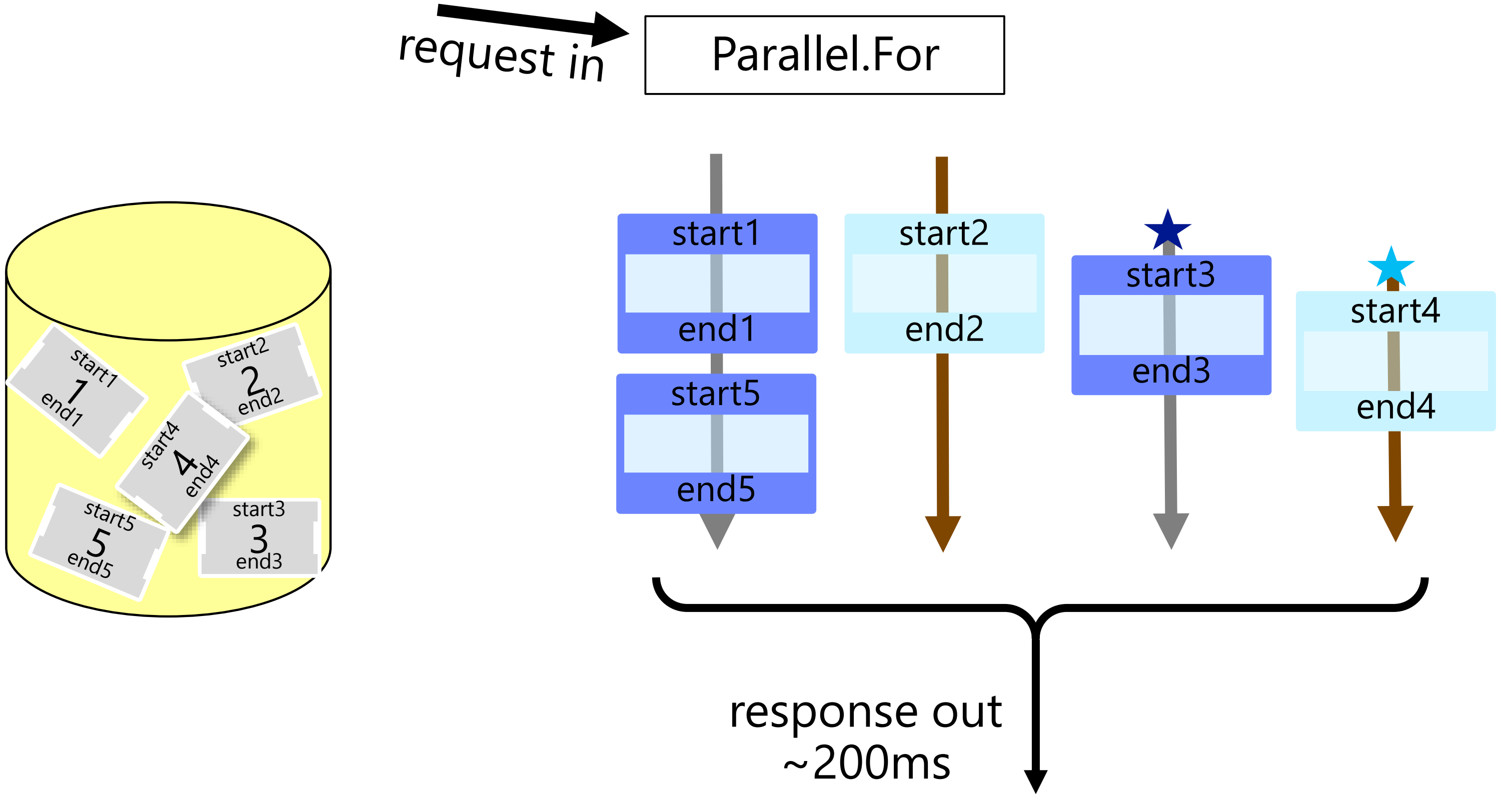
Is this code CPU-bound, or I/O-bound?

Note: Check it with the developer

Threadpool

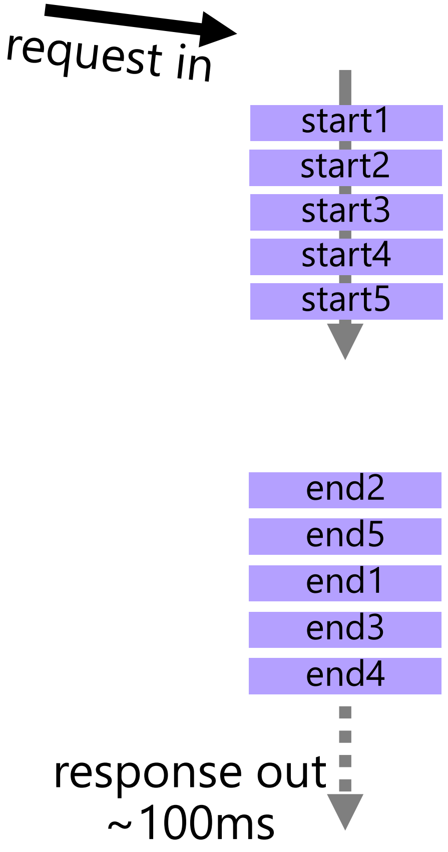

request in
start1
end1
start2
end2
start3
end3
start4
end4
start5
end5
response out
500ms

Threadpool



request in
Parallel.For
start1
1
end1
start2
2
end2
start4
4
end4
start5
5
end5
start3
3
end3
start1
end1
start2
end2
start3
end3
start4
end4
start5
end5
response out
~200ms

Threadpool


request in
start1
start2
start3
start4
start5
end2
end5
end1
end3
end4
response out
~100ms

Threadpool

```
// var houses = LoadHousesInParallel(1,5);
// Bind houses to UI control;

public async Task<List<House>> LoadHousesAsync(int first, int last)
{
    var tasks = new List<Task<House>>();

    for (int i = first; i <= last; i++)
    {
        var t = House.LoadFromDatabaseAsync(i);
        tasks.Add(t);
    }

    var loadedHouses = await Task.WhenAll(tasks);
    return loadedHouses.ToList();
}
```

You can call await 5 times

or…

When… methods minimize awaits + exceptions

Threadpool

Principles

CPU-bound work means things like: LINQ-over-objects, or big iterations, or computational inner loops.

Parallel.ForEach and Task.Run are a good way to put CPU-bound work onto the thread pool.

Thread pool will gradually feel out how many threads are needed to make best progress.

Use of threads will ***never*** increase throughput on a machine that's under load.

Guidance

For IO-bound "work", use await rather than background threads.

For CPU-bound work, consider using background threads via Parallel.ForEach or Task.Run, unless you're writing a library, or scalable server-side code.

Library methods shouldn't lie

Library methods shouldn't lie…

Only expose an async API
if your implementation is truly async.

Don't "fake it" through internal use of Task.Run.

Library methods shouldn’t lie

```
Foo();
```

This method’s signature is ***synchronous***:

I expect it will **perform** something here and now. I’ll regain control to execute something else **when it’s done**.

It will probably be using the CPU flat-out while it runs… or not.

```
var task = FooAsync();
...
await task;
```

This method’s signature is ***asynchronous***

I expect it will **initiate** something here and now. I’ll regain control to execute something else **immediately**.

It probably won’t take significant threadpool or CPU resources. ***

I could even kick off two FooAsyncs() to run them in parallel.

Library methods shouldn't lie

"Pause for 10 seconds, then print 'Hello'."

Synchronous

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

Asynchronous

```
public static Task PausePrint2Async() {
    return Task.Run(() 
```

USING A THREAD

"Should I expose async wrappers for synchronous methods?" – no!

```
// I want to offer an async signature,
// but my underlying library is synchronous"
```

```
public static void PausePrint2() {
    Task t = PausePrintAsync();
```

"How can I expose sync wrappers for async methods?" – if you absolutely have to, you can use a nested message-loop…

```
// but my underlying library is async
```

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

TRUE ASYNC.

#dotNetSpain2016

DEMO

The dangers of Task.Run in libraries

The dangers of

LIBRARIES THAT USE TASK.RUN
(looks async, but it wraps a sync implementation)

The threadpool is an app-global resource

The number of threads available to service work items varies greatly over the life of an app

The thread pool adds and removes threads using a hill climbing algorithm that adjusts slowly

In a server app, spinning up threads hurts scalability

A high-traffic server app may choose to optimize for scalability over latency

An API that launches new threads unexpectedly can cause hard-to-diagnose scalability bottlenecks

The app is in the best position to manage its threads

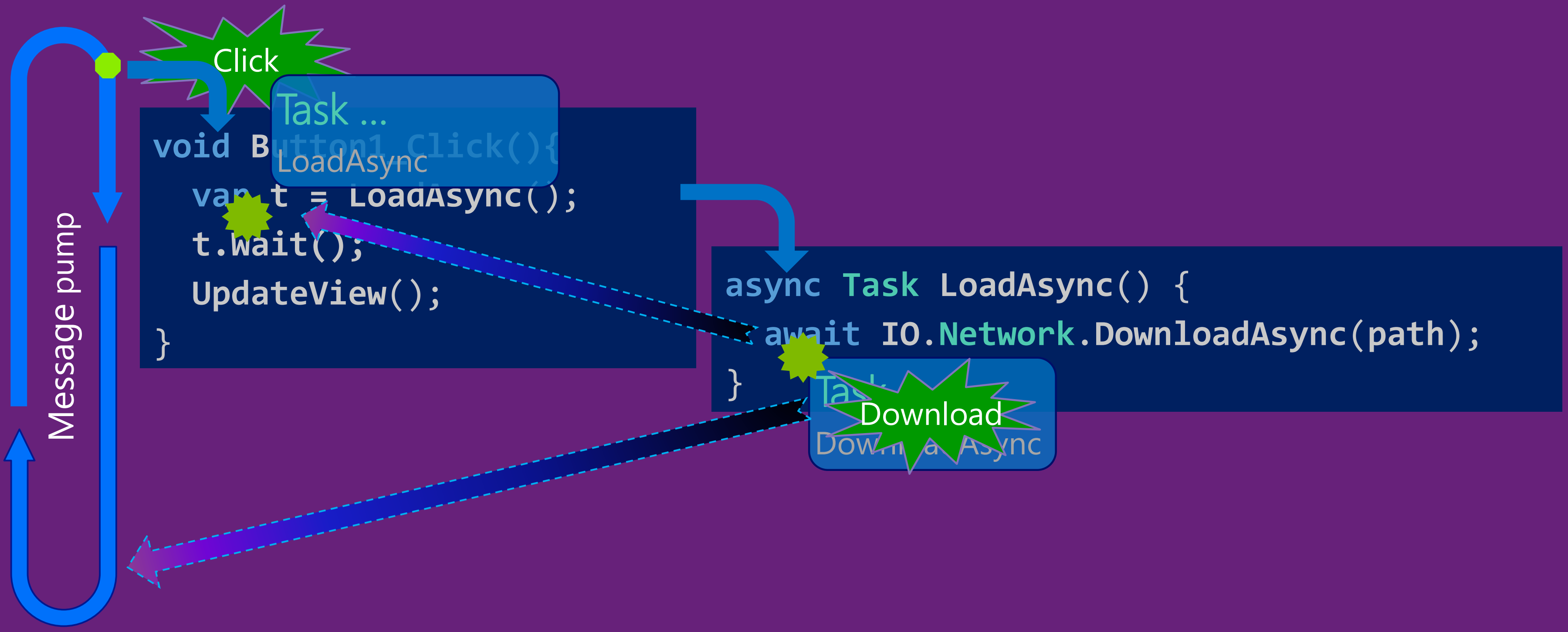
Provide **synchronous** methods when you do CPU-work that **blocks the current thread**

Provide **asynchronous** methods when you can do so **without spawning new threads**

Let the app that called you use its domain knowledge to manage its threading strategy (Task.Run)

The dangers of

BLOCKING
(looks sync, but it wraps an async/await method)



```
void Button1_Click(){
  var t = LoadAsync();
  t.Wait();
  UpdateView();
}
```

```
async Task LoadAsync() {
  await IO.Network.DownloadAsync(path);
}
```

Library methods shouldn't lie

Principles

The threadpool is an app-global resource.

Poor use of the threadpool hurts **server scalability.**

Guidance

Help your callers understand how your method behaves:

Libraries shouldn't use the threadpool in secret;

Use async signature only for truly async methods.

Should expose chunky async APIs

Async perf overhead is fine, unless you have a chatty API.

Don't await 10 million times in an inner loop.

But if you have to, then optimize using fast-path and caching.

Mental Model (sync)

We all “know” sync methods are “cheap”

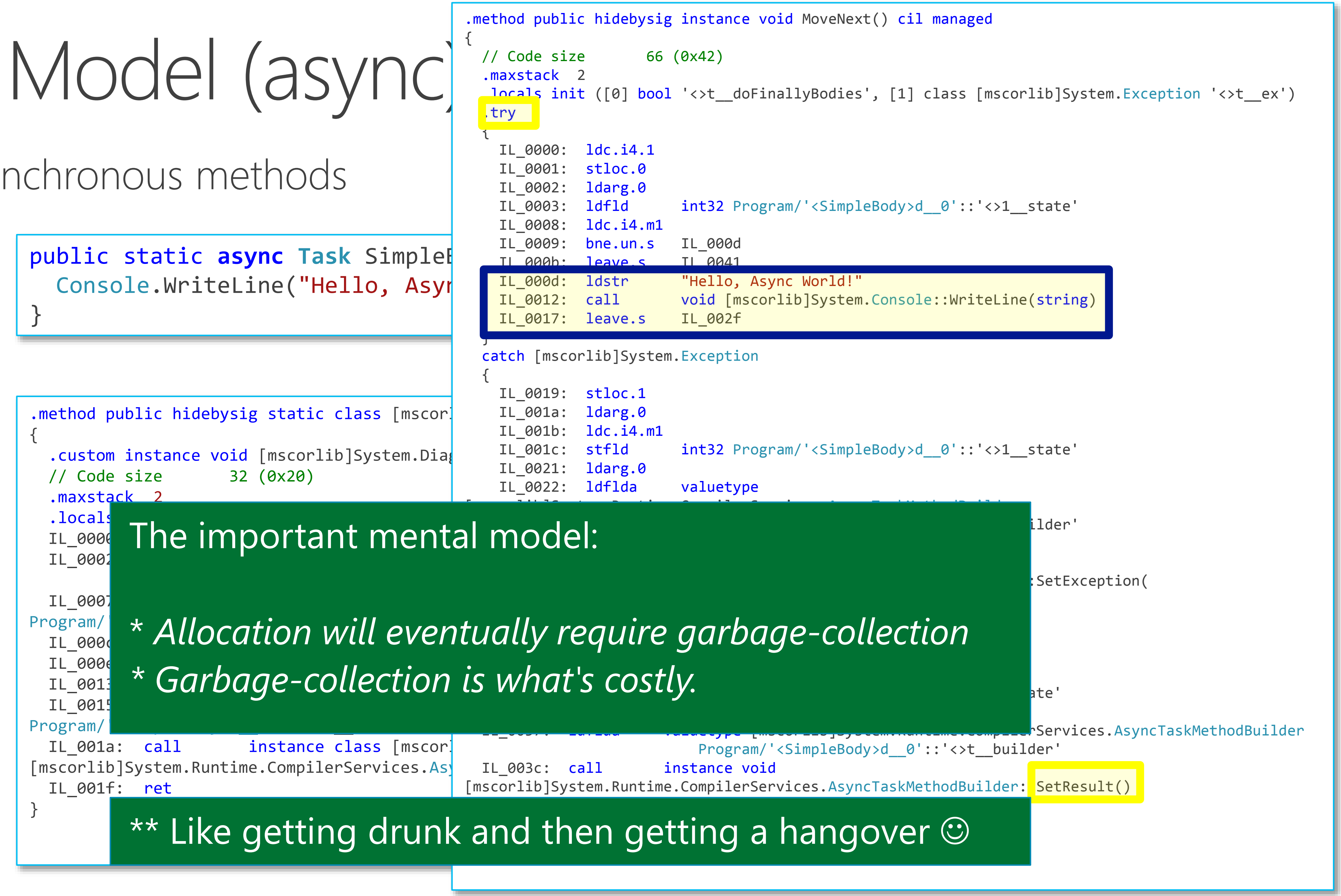
- Years of optimizations around sync methods
- Enables refactoring at will

```
public static void SimpleBody() {
  Console.WriteLine("Hello, Async World!");
}
```

```
.method public hidebysig static void SimpleBody() cil managed
{
    .maxstack 8
    L_0000: ldstr "Hello, Async World!"
    L_0005: call void [mscorlib]System.Console::WriteLine(string)
    L_000a: ret
}
```

Mental Model (async)

Not so for asynchronous methods



```
public static async Task SimpleB
  Console.WriteLine("Hello, Asyn
}
```

```
.method public hidebysig static class [mscor
{
  .custom instance void [mscorlib]System.Dia
  // Code size       32 (0x20)
  .maxstack  2
  .locals
  IL_0000
  IL_0002

  IL_0007
Program/
  IL_000c
  IL_000e
  IL_0013
  IL_0015
Program/
  IL_001a:  call       instance class [mscor
[mscorlib]System.Runtime.CompilerServices.Asy
  IL_001f:  ret
}
```

```
.method public hidebysig instance void MoveNext() cil managed
{
  // Code size       66 (0x42)
  .maxstack  2
  .locals init ([0] bool '<>t__doFinallyBodies', [1] class [mscorlib]System.Exception '<>t__ex')
  .try
  {
    IL_0000:  ldc.i4.1
    IL_0001:  stloc.0
    IL_0002:  ldarg.0
    IL_0003:  ldfld      int32 Program/'<SimpleBody>d__0'::'<>1__state'
    IL_0008:  ldc.i4.m1
    IL_0009:  bne.un.s   IL_000d
    IL_000b:  leave.s    IL_0041
    IL_000d:  ldstr      "Hello, Async World!"
    IL_0012:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_0017:  leave.s    IL_002f
  }
  catch [mscorlib]System.Exception
  {
    IL_0019:  stloc.1
    IL_001a:  ldarg.0
    IL_001b:  ldc.i4.m1
    IL_001c:  stfld      int32 Program/'<SimpleBody>d__0'::'<>1__state'
    IL_0021:  ldarg.0
    IL_0022:  ldflda     valuetype
                                                                      ilder'

                                                                      :SetException(

                                                                      ate'
                                                                      rServices.AsyncTaskMethodBuilder
                            Program/'<SimpleBody>d__0'::'<>t__builder'
  IL_003c:  call       instance void
[mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder::SetResult()
```

The important mental model:

* *Allocation will eventually require garbage-collection*

* *Garbage-collection is what's costly.*

** Like getting drunk and then getting a hangover ☺

Fast Path in awaits

Each async method involves allocations

- ~~For "state machine" class holding the method's local variables~~
- ~~For a delegate~~
- For the returned Task object

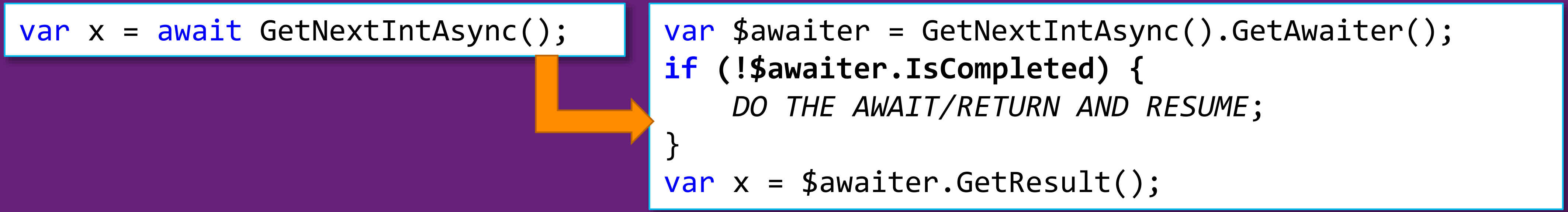
Avoided if the method skips its awaits

```
public static async Task<int> GetNextIntAsync()
{
    if (m_Count == m_Buf.Length)
    {
        m_Buf = await FetchNextBufferAsync();
        m_Count = 0;
    }
    m_Count += 1;
    return m_Buf[m_Count - 1];
}
```

Fast Path in awaits

If the awaited Task has already completed...

...then it skips all the await/resume work!



```
var x = await GetNextIntAsync();
```

```
var $awaiter = GetNextIntAsync().GetAwaiter();
if (!$awaiter.IsCompleted) {
    DO THE AWAIT/RETURN AND RESUME;
}
var x = $awaiter.GetResult();
```

Fast Path in awaits

Each async method involves allocations

- ~~For "state machine" class holding the method's local variables~~
- ~~For a delegate~~
- ~~For the returned Task object~~

Avoided if the method skips its awaits

Avoided if the method took fast path,
AND the returned value was "common" …

0, 1, true, false, null, ""

For other returns values, try caching yourself!

```
public static async Task<int> GetNextIntAsync()
{
    if (m_Count == m_Buf.Length)
    {
        m_Buf = await FetchNextBufferAsync();
        m_Count = 0;
    }
    m_Count += 1;
    return m_Buf[m_Count - 1];
}
```

DEMO

Tracking the Garbage Collector

Should expose chunky async APIs

Principles

The heap is an app-global resource.

Like all heap allocations, async allocations can contributing to hurting **GC perf.**

Guidance

Libraries should expose chunky async APIs. If GC perf is a problem, and the heap has lots of async allocations, then optimize the fast-path.

Use .ConfigureAwait(false) in libraries

Library methods might be called from different contexts:

Consider .ConfigureAwait(false)

Use .ConfigureAwait(false) in libraries

Sync context represents a “target for work”

e.g. WindowsFormsSynchronizationContext, whose .Post() does Control.BeginInvoke
e.g. DispatcherSynchronizationContext, whose .Post() does Dispatcher.BeginInvoke
e.g. AspNetSynchronizationContext, whose .Post() ensures one-at-a-time

“Await task” uses the sync context

1. It captures the current SyncContext before awaiting.
2. Upon task completion, it calls SyncContext.Post() to resume **“where you were before”**

For app-level code, this is fine. **But for library code, it’s rarely needed!**
You can use “await task.ConfigureAwait(false)”
This suppresses step 2; instead if possible it resumes **“on the thread that completed the task”**

Result: slightly better performance. Also can avoid deadlock if a badly-written user blocks.

DEMO

Consider .ConfigureAwait in Libraries

Use .ConfigureAwait(false) in libraries

Principles

The UI message-queue is an app-global resource.

To much use will hurt **UI responsiveness**.

Guidance

If your method calls chatty async APIs, but doesn’t touch the UI, then use ConfigureAwait(false)

Q&A

Special thanks to:

Stephen Toub
Lucian Wischik
Mads Torgersen
Stephen Cleary

MSDN Blogs & Channel 9

http://aka.ms/DOTNETT7S3

#dotNetSpain2016

¡Gracias!

No olvides realizar la encuesta

Lluis Franco
C# MVP & Plumber
@lluisfranco - hello@lluisfranco.com

http://aka.ms/DOTNETT7S3

** BONUS LEVEL **

Look behind you... a three headed monkey!