CSC350

Using Async and Await

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Introducing the `async` and `await` keywords
2. Applying `async` and `await`
3. Diving into the internals of `async` and `await`

Introducing the async and await keywords

Tasks

1. Defining asynchronous programming
2. Explore available .NET options for asynchronous programming
3. Using the **`async`** and **`await`** keywords to simplify asynchronous programming

What is asynchrony?

Synchronous vs. Asynchronous

Xamarin University

Why asynchronous programming?

- Asynchronous programming allows our mobile apps to continue to respond to user interaction while doing something else

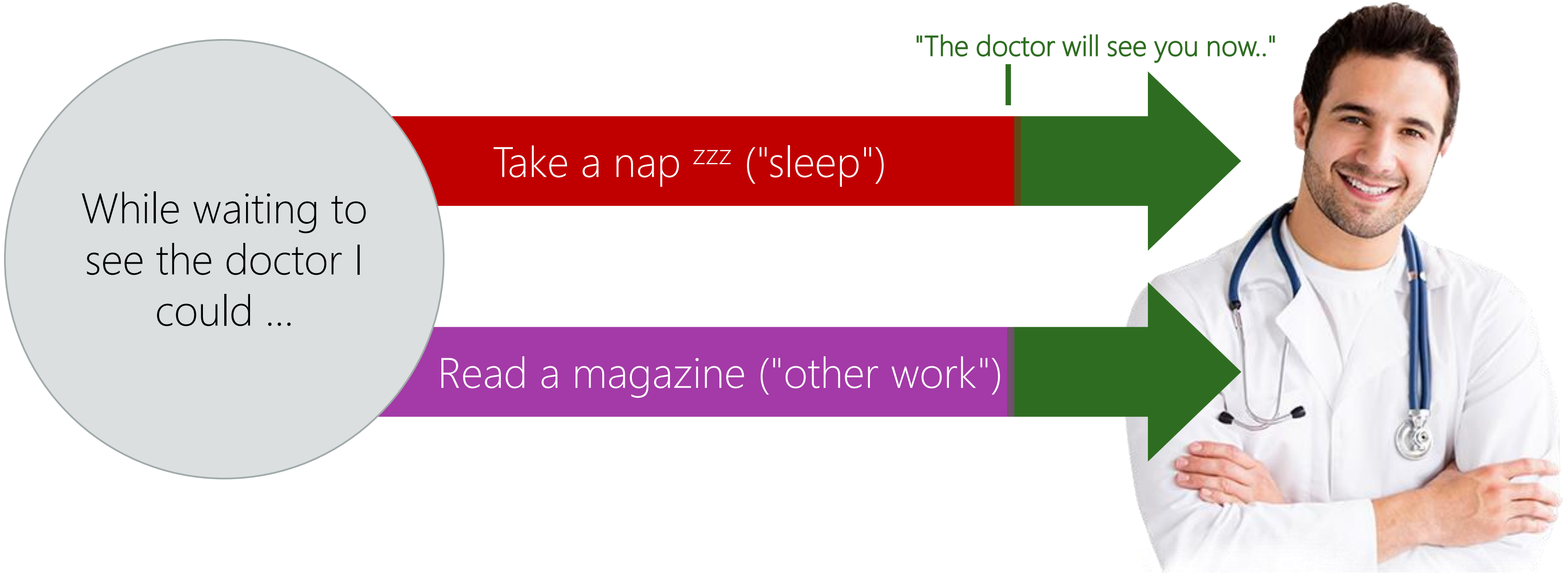
Reading or writing to a database or file

Accessing a web service

Performing data processing

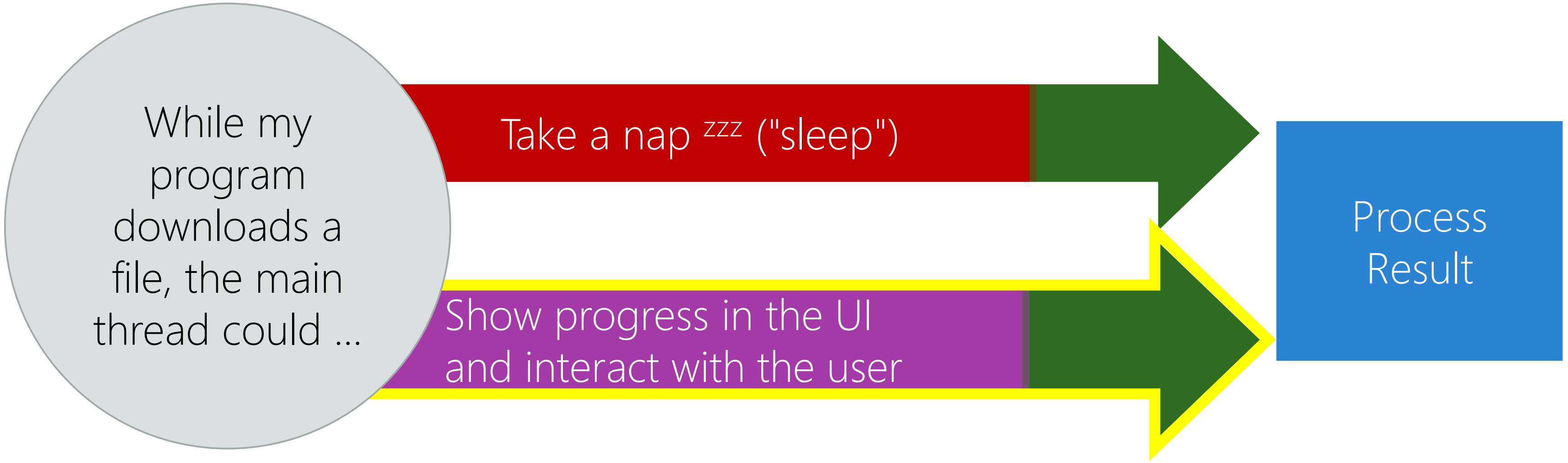
Thinking about asynchrony

- Asynchronous operations are started and then finish at some point in the future with a result … much like in the real world …


"The doctor will see you now.."
While waiting to see the doctor I could …
Take a nap zzz ("sleep")
Read a magazine ("other work")

Thinking about asynchrony

- Asynchronous operations are started and then finish at some point in the future with a result … Much like in the real world …


While my program downloads a file, the main thread could …
Take a nap zzz ("sleep")
Show progress in the UI and interact with the user
Process Result

Xamarin University

Async variations in .NET

❖ There are several asynchronous programming models to choose from:

.NET 1.0

Async Programming Model (APM)

IAsyncResult BeginMethod(...)
EndMethod(IAsyncResult)

Async variations in .NET

- There are several asynchronous programming models to choose from:

Async Programming Model (APM)

.NET 2.0

Event-based Async Pattern (EAP)

MethodAsync(...)
MethodCompleted event

Xamarin University

Async variations in .NET

- There are several asynchronous programming models to choose from:

Async Programming Model (APM)

Event-based Async Pattern (EAP)

.NET 4.0

Task-based Async Pattern (TAP)

Task MethodAsync(...)

Group Exercise

Use existing async APIs in an application

The problem with async programming

- Prior to .NET 4.5, async programming was done through **callbacks** that forced developers to write code in a **non-linear** fashion

```
void OnReadDataFromUrl(object sender, EventArgs e)
{
    WebClient wc = new WebClient();
    wc.DownloadDataCompleted += (sender, e) => {
        if (e.Error == null) {
            var data = UTF8Encoding.GetString(e.Result);
            LoadData(data);
        }
    };
    wc.DownloadDataAsync(new Uri(url.Text));
}
```

How *readable* is this code

Xamarin University

The problem with async programming

- Prior to .NET 4.5, async programming was done through **callbacks** that forced developers to write code in a **non-linear** fashion

```
void OnReadDataFromUrl(object sender, EventArgs e)
{
    WebClient wc = new WebClient();
    wc.DownloadDataCompleted += (sender, e) => {
        if (e.Error == null) {
            var data = UTF8Encoding.GetString(e.Result);
            LoadData(data);
        }
    };
    wc.DownloadDataAsync(new Uri(url.Text));
}
```

Processing code is defined separately

The problem with async programming

❖ Prior to .NET 4.5, async programming was done through **callbacks** that forced developers to write code in a **non-linear** fashion

```
void OnReadDataFromUrl(object sender, EventArgs e)
{
    WebClient wc = new WebClient()
    wc.DownloadDataCompleted += (s
        if (e.Error == null) {
            var data = UTF8Encoding.GetString(e.Result);
            LoadData(data);
        }
    };
    wc.DownloadDataAsync(new Uri(url.Text));
}
```

Exceptions are reported in non-traditional fashion

The problem with async programming

- Prior to .NET 4.5, async programming was done through **callbacks** that forced developers to write code in a **non-linear** fashion

```
void OnReadDataFromUrl(object sender, EventArgs e)
{
    WebClient wc = new WebClient();
    wc.DownloadDataCompleted += (sender, e) => {
        if (e.Error == null) {
            var data = UTF8Encoding.GetString(e.Result);
            LoadData(data);
        }
    };
    wc.DownloadDataAsync(new Uri(url.Text));
}
```

Initiating method call is done *last*

Xamarin University

What we'd like to do…

- We want to write our code just as if it were going to be executed synchronously in step-by-step fashion … like this:

```
void OnReadDataFromUrl(object sender, EventArgs e)
{
    WebClient wc = new WebClient();
    try {
        byte[] result = wc.DownloadData(new Uri(url.Text));
        var data = UTF8Encoding.GetString(result);
        LoadData(data);
    }
    catch (Exception ex) { ... }
}
```

How *readable* is this code **now**

Making asynchronous code simpler

- Most UI applications benefit from asynchronous code, *but* developers struggle to write it properly; enter C# 5 and two new keywords:

async await

The new world of async + await

- C# keywords allow developers to write code in a **synchronous fashion** but have it **run asynchronously**

```
async void OnReadDataFromUrl(object sender, EventArgs e)
{
    WebClient wc = new WebClient();
    try {
        byte[] result = await wc.DownloadData(new Uri(url.Text));
        var data = UTF8Encoding.GetString(result);
        LoadData(data);
    }
    catch (Exception ex) { ... }
}
```

Processing code is defined exactly where it should be – right after the call to get the data

The new world of async + await

- C# keywords allow developers to write code in a **synchronous fashion** but have it **run asynchronously**

```
async void OnReadDataFromUrl(object sender, EventArgs e)
{
    WebClient wc = new WebClient();
    try {
        byte[] result = await wc.DownloadData(new Uri(url.Text));
        var data = UTF8Encoding.GetString(result);
        LoadData(data);
    }
    catch (Exception ex) { ... }
}
```

Errors are handled using traditional exception model

Demonstration

Look at async and await

Xamarin University

Summary

1. Defining asynchronous programming
2. Explore available .NET options for asynchronous programming
3. Using the **`async`** and **`await`** keywords to simplify asynchronous programming

Applying async and await

Xamarin University

Tasks

1. What does the `async` keyword do?
2. Applying the `await` keyword
3. Working with awaitable expressions
4. Limitations of `async` and `await`

Xamarin University

What does the async keyword do?

- The presence of **async** on a method allows the **await** keyword to be used in the method body, and indicates that some part of this method *can be executed asynchronously*

```
async void OnReadDataFromUrl(object sender, EventArgs e)
{
    ... // Now we can use await
}
```

Must be applied before the return type declaration on the method

Xamarin University

Applying the await keyword

- The `await` keyword is applied to *awaitable expressions*

```
async void OnReadDataFromUrl(object sender, EventArgs e)
{
    WebClient wc = new WebClient();
    byte[] result = await wc.DownloadDataTaskAsync(
                                        new Uri(url.Text));
...
```

An **awaitable** expression is an asynchronous operation, like this one that downloads data from a web endpoint

Xamarin University

What does the await keyword do?

- Key idea behind **await** is to *pause* forward execution of the method until *after* the asynchronous operation is complete

```
async void OnReadDataFromUrl(object sender, EventArgs e)
{
    WebClient wc = new WebClient();

    byte[] result = await wc.DownloadDataTaskAsync(
                                      new Uri(url.
    ...............................................
    var data = UTF8Encoding.GetString(result);
    LoadData(data);
...
```

This code cannot execute until the data is downloaded

Using await in lambda expressions

❖ Can also use keywords in lambda expressions and anonymous delegates

must add async keyword onto the expression definition

```
button.Clicked += async (sender,e) =>
{
    HttpClient client = new HttpClient();
    string contents = await client.GetStringAsync(...);
    welcomeLabel.Text = contents.ToLower();
};
```

await keyword goes into the method body – but is still applied to the awaitable expression

What is an awaitable expression?

- *Awaitable expressions* in .NET are methods that return a **`Task`** or **`Task<T>`**; this is a framework class that represents an asynchronous request

```
public Task<byte[]> DownloadDataTaskAsync(string address);
```

Returning a task from a method call indicates that some portion of the method is performed asynchronously

What is an awaitable expression?

- *Awaitable expressions* in .NET are methods that return a **`Task`** or **`Task<T>`**; this is a framework class that represents an asynchronous request

```
public Task<byte[]> DownloadDataTaskAsync(string address);
```

`Task<T>` is a generic version of **`Task`** that returns a *promise*, or *future value* that will be available when the asynchronous operation completes

What is an awaitable expression?

- *Awaitable expressions* in .NET are methods that return a **`Task`** or **`Task<T>`**; this is a framework class that represents an asynchronous request

```
public Task<byte[]> DownloadDataTaskAsync(string address);
```

By convention, methods that are executed asynchronously always have the suffix Async

Xamarin University

Be careful: Await != Thread

- `await` does not create a thread, or even require a thread be used

```
async void LetsGoAsync(...) {
   Debug.WriteLine("1.. 2.. 3..");
   await TimerDelay(1000);
   Debug.WriteLine("Go!");
}

Task TimerDelay(int msec) { ... }
```

Possibly no additional thread is used here, but the two lines will be displayed 1 second apart; `await` is all about *coordination*, whether a thread is used or not is up to the awaitable expression

Be careful: Await != Thread

- The awaitable expression must provide the asynchronous capability

```
async void button1_Click( ... )
{
   Action work = CPUWork;
   await RunWork(work);
}

async Task RunWork(Action work)
{
   work();
}
```

CPUWork will be executed synchronously!

Be careful: Await != Thread

❖ The awaitable expression must provide the asynchronous capability

```
async void button1_Click( ... )
{
   Action work = CPUWork;
   await RunWork(work);
}

async Task RunWork(Action work)
{
   return Task.Run(() => work());
}
```

✓

Flash Quiz

Flash Quiz

① Which method definition is correct?

a) `void async ReadDataFromUrl(string url) { ... }`

b) `void await ReadDataFromUrl(string url) { ... }`

c) `async void ReadDataFromUrl(string url) { ... }`

d) `await void ReadDataFromUrl(string url) { ... }`

Xamarin University

Flash Quiz

① Which method definition is correct?

a) `void async ReadDataFromUrl(string url) { ... }`

b) `void await ReadDataFromUrl(string url) { ... }`

c) `async void ReadDataFromUrl(string url) { ... }`

d) `await void ReadDataFromUrl(string url) { ... }`

Flash Quiz

② In response to the **`await`** keyword, the C# compiler will create a thread

a) True

b) False

Flash Quiz

② In response to the **`await`** keyword, the C# compiler will create a thread

a) True

b) [✓ False]

Summary

1. What does the **`async`** keyword do?
2. Applying the **`await`** keyword
3. Working with awaitable expressions
4. Limitations of **`async`** and **`await`**

Diving into the internals of async and await

Xamarin University

Tasks

1. What does the **`await`** keyword do?
2. Exploring the generated code
3. Dealing with return values

Execution progress for await

- At runtime, each **await** keyword starts the asynchronous operation and then **returns to the caller** because it cannot continue execution yet

```
async void OnClick(...)
{
    string url = ...;
✔   indicator.IsRunning = true;
    await ReadFromUrlAsync(url);
    indicator.IsRunning = false;
}
```

```
async Task ReadFromUrlAsync(string url)
{
✔   WebClient wc = new WebClient();

    byte[] result = await wc.DownloadDataTaskAsync(
                                  new Uri(url));

    var data = Encoding.ASCII.GetString(result);
    LoadData(data);
...
```

Execution progress for await

❖ … then when the awaitable expression has a result, the runtime will **return to the method where it left off** to continue execution

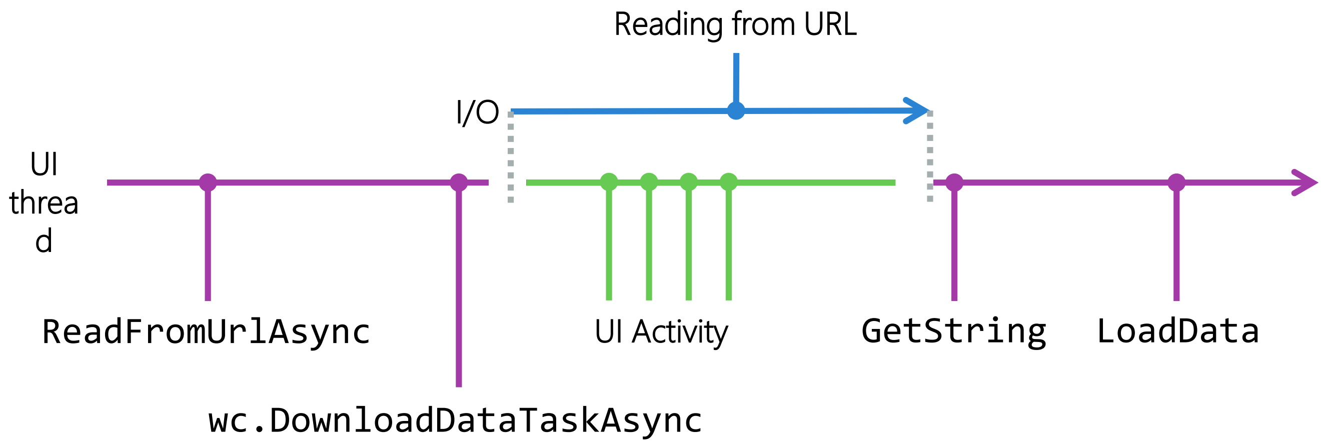
```
async void OnClick(...)
{
   string url = ...;
✔  indicator.IsRunning = true;
✔  await ReadFromUrlAsync(url);
✔  indicator.IsRunning = false;
}
```

```
async Task ReadFromUrlAsync(string url)
{
✔  WebClient wc = new WebClient();

✔  byte[] result = await wc.DownloadDataTaskAsync(
                              new Uri(url));

✔  var data = Encoding.ASCII.GetString(result);
✔  LoadData(data);
...
```

UI thread processes other UI events while waiting for the data to be downloaded

Execution progress for await

- … then when the awaitable expression has a result, the runtime will **return to the method where it left off** to continue execution



How does await *really* work?

- Adding **async** changes how C# compiles the method and prepares it to support one or more asynchronous operations

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

```
.method public hidebysig static void SayHello() cil managed
{
    .maxstack 8
    L_0000: nop
    L_0001: ldstr "Hello, World!"
    L_0006: call void [mscorlib]System.Console::WriteLine(string)
    L_000b: nop
    L_000c: ret
}
```

How does await *really* work?

- Adding `async` changes how C# compiles the method and prepares it to support one or more asynchronous operations

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

All we've done is add the `async` keyword to the method... but look what changes in the IL:

```
.method public hidebysig static void SayHello() cil managed
{
  .maxstack 2
  .locals init ([0] class Test.Program/<SayHello>d__1 d__)
  IL_0000:  ldloca.s   V_0
  IL_0002:  call       valuetype [mscorlib]System.Runtime.CompilerServic
[mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder::Create
  IL_0007:  stfld      valuetype [mscorlib]System.Runtime.CompilerServic
Program/'<SayHello>d__1'::'<>t__builder'
  IL_000c:  ldloca.s   V_0
  IL_000e:  call       instance void Program/'<SayHello>d__1'::MoveNext()
  IL_0013:  ret
}
```

New compiler-generated class introduced to manage any asynchronous code

Method body is compiled as a series of "steps" similar to iterator methods

```
.method public hidebysig static void SayHello() cil managed
{
  .maxstack 2
  .locals init ([0] class Test.Program/<SayHello>d__1 d__)
  IL_0000:  ldloca.s   V_0
  IL_0002:  call       valuetype [mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder
[mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder::Create()
  IL_0007:  stfld      valuetype [mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder
Program/'<SayHello>d__1'::'<>t__builder'
  IL_000c:  ldloca.s   V_0
  IL_000e:  call       instance void Program/'<SayHello>d__1'::MoveNext()
  IL_0013:  ret
}
```

async methods always return after **MoveNext** is finished

```
.method public hidebysig static void SayHello() cil managed
{
  .maxstack 2
  .locals init ([0] class Test.Program/<SayHello>d__1 d__)
  IL_0000:  ldloca.s   V_0
  IL_0002:  call       valuetype [mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder
[mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder::Create()
  IL_0007:  stfld      valuetype [mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder
Program/'<SayHello>d__1'::'<>t__builder'
  IL_000c:  ldloca.s   V_0
  IL_000e:  call       instance void Program/'<SayHello>d__1'::MoveNext()
  IL_0013:  ret
}
```

```
.method private hidebysig newslot virtual final instance void MoveNext() cil managed
{
  .override [mscorlib]System.Runtime.CompilerServices.IAsyncStateMachine::MoveNext
  .maxstack 2
  .locals init ([0] int32 num, [1] class [mscorlib]System.Exception exception)
  L_0000: ldarg.0
  L_0001: ldfld int32 Test.Program/<SayHello>d__1::<>1__state
  L_0006: stloc.0
  L_0008: ldstr "Hello, World!"
  L_000d: call void [mscorlib]System.Console::WriteLine(string)
  L_0013: leave.s
  ...
}
```

Tracks the current "state" of this async method

```
.method public hidebysig static void SayHello() cil managed
{
  .maxstack 2
  .locals init ([0] class Test.Program/<SayHello>d__1 d__)
  IL_0000:  ldloca.s   V_0
  IL_0002:  call       valuetype [mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder
[mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder::Create()
  IL_0007:  stfld      valuetype [mscorlib]System.Runtime.CompilerServices.AsyncTaskMethodBuilder
Program/'<SayHello>d__1'::'<>t__builder'
  IL_000c:  ldloca.s   V_0
  IL_000e:  call       instance void Program/'<SayHello>d__1'::MoveNext()
  IL_0013:  ret
}
```

```
.method private hidebysig newslot virtual final instance void MoveNext() cil managed
{
  .override [mscorlib]System.Runtime.CompilerServices.IAsyncStateMachine::MoveNext
  .maxstack 2
  .locals init ([0] int32 num, [1] class [mscorlib]System.Exception exception)
  L_0000: ldarg.0
  L_0001: ldfld int32 Test.Program/<SayHello>d__1::<>1__state
  L_0006: stloc.0
  L_0008: ldstr "Hello, World!"
  L_000d: call void [mscorlib]System.Console::WriteLine(string)
  L_0013: leave.s
  ...
}
```

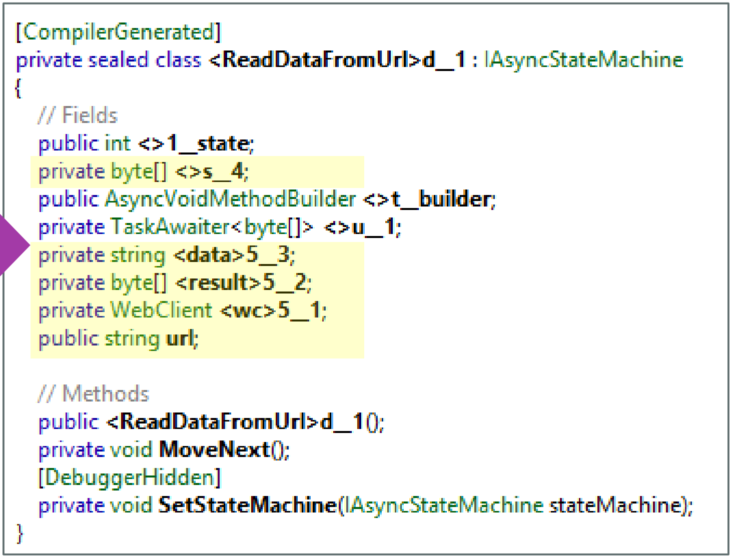
Always captures exceptions – even if you didn't ask it to!

The secret behind await

- Under the covers, the compiler turns the method into a *state machine* in anticipation of having portion(s) be executed in steps

```
async void ReadDataFromUrl(string url)
{
   WebClient wc = new WebClient();
   var result = await wc.DownloadDataTaskAsync(url);
   var data = Encoding.ASCII.GetString(result);
   LoadData(data);
}
```

All of the variables in the method are captured as fields in the state machine and used to manage the local state



```
[CompilerGenerated]
private sealed class <ReadDataFromUrl>d__1 : IAsyncStateMachine
{
    // Fields
    public int <>1__state;
    private byte[] <>s__4;
    public AsyncVoidMethodBuilder <>t__builder;
    private TaskAwaiter<byte[]> <>u__1;
    private string <data>5__3;
    private byte[] <result>5__2;
    private WebClient <wc>5__1;
    public string url;

    // Methods
    public <ReadDataFromUrl>d__1();
    private void MoveNext();
    [DebuggerHidden]
    private void SetStateMachine(IAsyncStateMachine stateMachine);
}
```

What's in MoveNext?

- State Machine is coded into a **`MoveNext`** method which is called for each step and tracks an integer state to execute the code

```
async void ReadDataFromUrl(string url)
{
   WebClient wc = new WebClient();
   var result = await wc.DownloadDataTaskAsync(url);
   var data = Encoding.ASCII.GetString(result);
   LoadData(data);
}
```

1 Create the **`WebClient`** and issue the download async request

```
public void MoveNext()
{
   uint num = (uint)this.$PC;
   this.$PC = -1;
   try {
      switch (num) {
         case 0:
            this.<wc>__0 = new WebClient();
            this.$awaiter0 = this.<wc>                    s.url).GetAwaiter();
            this.$PC = 1;
            ...
            return;
            break;
         case 1:
            this.<result>__1 = this.$awaiter0.GetResult();
            this.<data>__2 = Encoding.ASCII.GetString(this.<result>__1);
            this.$this.LoadData(this.<data>__2);
            break;
         default:
            return;
      }
   }
   catch (Exception exception) { ... }
   this.$PC = -1;
   this.$builder.SetResult();
}
```

Move to next state

What's in MoveNext?

- State Machine is coded into a **`MoveNext`** method which is called for each step and tracks an integer state to execute the code

```
async void ReadDataFromUrl(string url)
{
   WebClient wc = new WebClient();
   var result = await wc.DownloadDataTaskAsync(url);
   var data = Encoding.ASCII.GetString(result);
   LoadData(data);
}
```

1 Create the **`WebClient`** and issue the download async request

```
public void MoveNext()
{
   uint num = (uint)this.$PC;
   this.$PC = -1;
   try {
      switch (num) {
         case 0:
            this.<wc>__0 = new WebClient();
            this.$awaiter0 = this.<wc>__0.DownloadDataTaskAsync(this.url).GetAwaiter();
            this.$PC = 1;
            ...
            return;
            break;
         case 1:
            this.<result>__1 = th
            this.<data>__2 = Enc                                 sult>__1);
            this.$this.LoadData(this.<data>__2);
            break;
         default:
            return;
      }
   }
   catch (Exception exception) { ... }
   this.$PC = -1;
   this.$builder.SetResult();
}
```

Notice the return!

What's in MoveNext?

- State Machine is coded into a **`MoveNext`** method which is called for each step and tracks an integer state to execute the code

```
async void ReadDataFromUrl(string url)
{
   WebClient wc = new WebClient();
   var result = await wc.DownloadDataTaskAsync(url);
   var data = Encoding.ASCII.GetString(result);
   LoadData(data);
}
```

2 Process the results from the async call

```
public void MoveNext()
{
   uint num = (uint)this.$PC;
   this.$PC = -1;
   try {
      switch (num) {
         case 0:
            this.<wc>__0 = new WebClient();
            this.$awaiter0 = this.<wc>__0.DownloadDataTaskAsync(this.url).GetAwaiter();
            this.$PC = 1;
            ...
            return;
            break;
         case 1:
            this.<result>__1 = this.$awaiter0.GetResult();
            this.<data>__2 = Encoding.ASCII.GetString(this.<result>__1);
            this.$this.LoadData(this.<data>__2);
            break;
         default:
            return;
      }
   }
   catch (Exception exception) { ... }
   this.$PC = -1;
   this.$builder.SetResult();
}
```

Xamarin University

What's in MoveNext?

- State Machine is coded into a **`MoveNext`** method which is called for **each step** and tracks an integer state to execute the code

```
async void ReadDataFromUrl(string url)
{
   WebClient wc = new WebClient();
   var result = await wc.DownloadDataTaskAsync(url);
   var data = Encoding.ASCII.GetString(result);
   LoadData(data);
}
```

3 Catches all exceptions; this is done even if your method does not have a `try` / `catch` handler and is how `await` is able to re-throw them in the client code.

```
public void MoveNext()
{
   uint num = (uint)this.$PC;
   this.$PC = -1;
   try {
      switch (num) {
         case 0:
            this.<wc>__0 = new WebClient();
            this.$awaiter0 = this.<wc>__0.DownloadDataTaskAsync(this.url).GetAwaiter();
            this.$PC = 1;
            ...
            return;
            break;
         case 1:
            this.<result>__1 = this.$awaiter0.GetResult();
            this.<data>__2 = Encoding.ASCII.GetString(this.<result>__1);
            this.$this.LoadData(this.<data>__2);
            break;
         default:
            return;
      }
   }
   catch (Exception exception) { ... }
   this.$PC = -1;
   this.$builder.SetResult();
}
```

Flash Quiz

Flash Quiz

① The `await` keyword causes the current thread to block waiting for the asynchronous operation to complete

a) True

b) False

Xamarin University

Flash Quiz

① The **`await`** keyword causes the current thread to block waiting for the asynchronous operation to complete

a) True

b) [✓ <u>False</u>]

Xamarin University

Flash Quiz

② Adding **`async`** to a method definition doesn't change anything until the **`await`** keyword is used

a) True

b) False

Flash Quiz

② Adding **`async`** to a method definition doesn't change anything until the **`await`** keyword is used

a) True

b) [✓ <u>False</u>]

Flash Quiz

③ What side effects does using `async/await` always have on a method? (Select all that apply)

a) The method will be broken into multiple steps

b) Exceptions will be caught, and *possibly* re-thrown

c) Local variables will be captured and moved to the GC heap

d) It will cause the method to use multiple threads

Flash Quiz

③ What side effects does using **`async`/`await`** always have on a method? (Select all that apply)

a) The method will be broken into multiple steps

b) Exceptions will be caught, and *possibly* re-thrown

c) Local variables will be captured and moved to the GC heap

d) It will cause the method to use multiple threads

Leaking abstractions

- Notice that the value being **consumed** and the value being **returned** are not quite the same

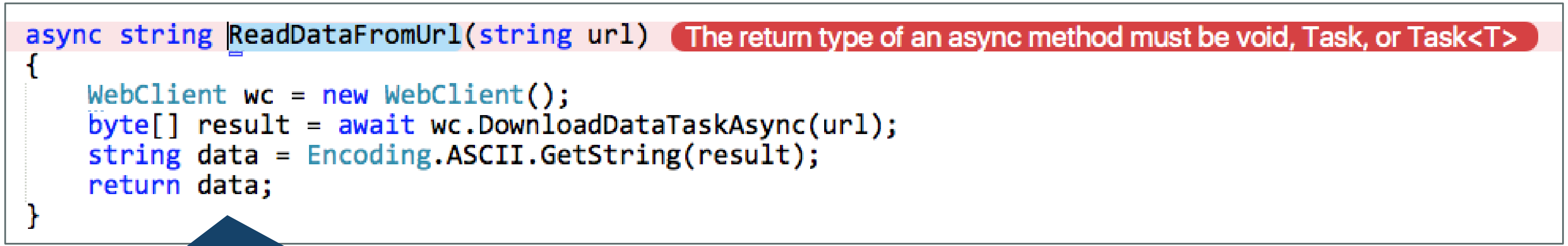
```
byte[] result = await wc.DownloadDataTaskAsync( ... );
```

Compiler interprets the `await` keyword to mean "get the result from the task"

```
public Task<byte[]> DownloadDataTaskAsync(string address);
```

Dealing with return values

- This becomes evident in the return value from methods that *use* `await`



```
async string ReadDataFromUrl(string url)
{
    WebClient wc = new WebClient();
    byte[] result = await wc.DownloadDataTaskAsync(url);
    string data = Encoding.ASCII.GetString(result);
    return data;
}
```

The return type of an async method must be void, Task, or Task<T>

What are we returning here?
Or perhaps a better question is *where are we returning from this method?*

Dealing with return values

❖ This becomes evident in the return value from methods that *use* **`await`**

```
async string ReadDataFromUrl(string url)
{
   WebClient wc = new WebClient();
   byte[] result = await wc.DownloadDataTaskAsync(url);
   string data = Enc[illegible]ng.ASCII.GetString(result);
   return da
}
```

The method is *really* returning to the caller **here** – before we ever hit an actual return keyword .. What must this return?

Returning a "future" value

- `Task<T>` represents a "future" value – something that will eventually produce either a value or exception; this is what we must return from the method in order for the compiler to produce legitimate code

```
async Task<string> ReadDataFromUrl(string url)
{
    WebClient wc = new WebClient();
    var result = await wc.DownloadDataTaskAsync(url);
    string data = Encoding.ASCII.GetString(result);
    return data;
}
```

✓

Valid return values for async methods

❖ Since `await`ed methods return before the entire method is complete, they must have a **specific return type**, one of three valid values:

`Task<T>` if it returns a value

```
async Task<double> CalculatePiAsync()
```

Xamarin University

Valid return values for async methods

- Since `await`ed methods return before the entire method is complete, they must have a **specific return type**, one of three valid values:

`Task<T>` if it returns a value

`Task` for no return value (e.g. *void*)

```
async Task WriteToLogAsync(...)
```

Valid return values for async methods

- Since `await`ed methods return before the entire method is complete, they must have a **specific return type**, one of three valid values:

`Task<T>` if it returns a value

`Task` for no return value (e.g. *void*)

`void` for event handlers

```
async void OnClick(...)
```

Beware void returns

❖ You cannot `await` a `void`-return async method since they don't return a `Task`

❖ Without an `await`, you cannot catch any exceptions that occur in the task

❖ You should **never** use a `void`-return async method *unless* it's an event handler, or a virtual method override where you have no choice

Individual Exercise

Convert app to use async and await

Xamarin University

When is await unnecessary?

- If you do not need to process the results from a **`Task`**, then you can just return the task directly – no need to await it

```
async Task<byte[]> ReadDataFromUrlAsync(string url)
{
    WebClient wc = new WebClient();
    byte[] result = await wc.DownloadDataTaskAsync(
                              new Uri(url));
    return result;
}
```

Think about what happens when the compiler does with the `async` keyword and what it does when it compiles the **`await`** keyword?

When is await unnecessary?

- If you do not need to process the results from a **`Task`**, then you can just return the task directly – no need to await it

```
Task<byte[]> ReadDataFromUrlAsync(string url)
{
    WebClient wc = new WebClient();
    return wc.DownloadDataTaskAsync(new Uri(url));
}
```

This is more efficient because we avoid the state machine logic *and* we don't come back to this method when the download is complete!

Xamarin University

Mapping await to threads

- Sometimes it's beneficial to continue the work on the task, rather than switching back to the original thread

```
async Task ReadDataFromUrl(string url)
{
    WebClient wc = new WebClient();
    byte[] result = await wc.DownloadDataTaskAsync(
                                        new Uri(url));

    var data = Encoding.ASCII.GetString(result);
    LoadData(data);
...
```

Do we need to be back on thread #1 when we run this code?

Xamarin University

Staying on the task thread

- Use **`ConfigureAwait(false)`** on the task to tell the API to *not* switch back to the original thread context

```
WebClient wc = new WebClient();
Task<byte[]> task = wc.DownloadDataTaskAsync(new Uri(url));

byte[] result = await task.ConfigureAwait(false);

var data = Encoding.ASCII.GetString(result);
```

Staying on the task thread

- Use `ConfigureAwait(false)` on the task to tell the API to *not* switch back to the original thread context

```
WebClient wc = new WebClient();
Task<byte[]> task = wc.DownloadDataTaskAsync(new Uri(url));

byte[] result = await task.ConfigureAwait(false);

var data = Encoding.ASCII.GetString(result);
```

🟥 Thread #1
🟩 Async Op

This can be more efficient when data processing does not need to be on the original thread (e.g. UI) because we avoid the cost of switching threads

Xamarin University

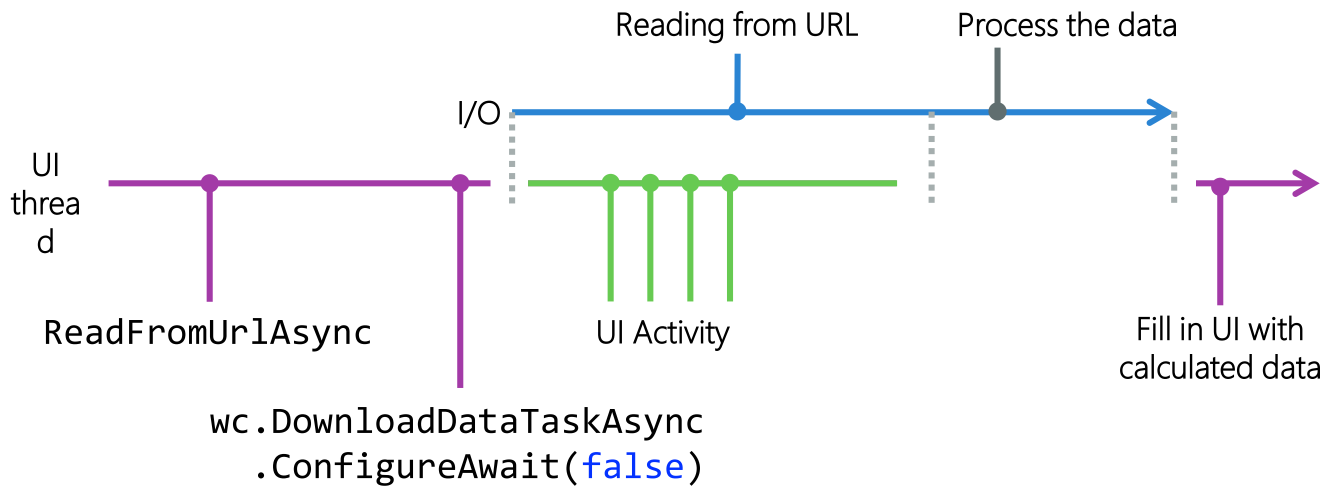
Staying on the task thread

- Beware: `ConfigureAwait` can return back on the *original* thread if the task completes immediately (e.g. no asynchrony occurred)

```
// Start on Thread #1 (UI thread)
await Task.FromResult(0).ConfigureAwait(false);

// WARNING: Still on Thread #1!
await Task.Delay(0).ConfigureAwait(false);

// WARNING: Still on Thread #1!
await Task.Delay(1000).ConfigureAwait(false);

// OK: Now on some other thread..
```

Manually switching back to the UI thread

- Can get the best of both worlds by pushing the **majority of the work** onto the background/worker thread and **switching to the UI thread** to update UI



Reading from URL
Process the data
I/O
UI threa d
ReadFromUrlAsync
UI Activity
Fill in UI with calculated data
wc.DownloadDataTaskAsync .ConfigureAwait(false)

Staying on the task thread

- Each method that utilizes await has it's own context – so the *calling* method is <u>not</u> affected by **`ConfigureAwait`**

```
async Task RefreshMovieList() {
   var movieList = await ReloadMovies ();
   RefreshUIWithMovies (movieList);
}

async Task<List<Movie>> ReloadMovies() {
   var result = await new WebClient()
         .DownloadDataTaskAsync(new Uri(AmazonMovies))
         .ConfigureAwait(false);
   var json = Encoding.ASCII.GetString (result);
   return Newtonsoft.Json.JsonConvert.DeserializeObject<List<Movie>> (json);
}
```

What thread will this method be executed on?

Xamarin University

Staying on the task thread

- Each method that utilizes await has it's own context – so the *calling* method is not affected by **ConfigureAwait**

🟥 Main thread
🟩 Bkgnd thread

```
async Task RefreshMovieList() {
   var movieList = await ReloadMovies ();
   RefreshUIWithMovies (movieList);
}

async Task<List<Movie>> ReloadMovies() {
   var result = await new WebClient()
         .DownloadDataTaskAsync(new Uri(AmazonMovies))
         .ConfigureAwait(false);
   var json = Encoding.ASCII.GetString (result);
   return Newtonsoft.Json.JsonConvert.DeserializeObject<List<Movie>> (json);
}
```

Switching to the UI thread

- Each platform has a unique API to get to the UI thread, Xamarin.Forms abstracts this into a single static method on the **`Device`** class

```
Task.Run(() => { // Long running work in a loop
   ...
   while (!haveFinalValue) {
      calculatedValue = RefineCalculation(calculatedValue);
      // Update the UI
      Device.BeginInvokeOnMainThread(() =>
         resultLabel.Text = calculatedValue.ToString();
      });
      haveFinalValue = ...;
  }
});
```

Switching to the UI thread

- `SynchronizationContext` allows you to switch to an associated thread; this works on all platforms *and* can be mocked out for unit tests!

```
// Must get context on the thread we want to return to (UI)
SynchronizationContext ctx = SynchronizationContext.Current;
Task.Run(() => {
   ...
   // Now let's update the UI
   ctx.Post(unused => {
      resultLabel.Text = calculatedValue.ToString();
   }, null);
});
```

Flash Quiz

Flash Quiz

① The **`await`** keyword can be applied to ____________

a) any method call we want to run on a different thread

b) really fast code we want to slow down

c) methods that return a **`Task`** or **`Task<T>`**

d) All of the above

Xamarin University

Flash Quiz

① The `await` keyword can be applied to ___________

a) any method call we want to run on a different thread

b) really fast code we want to slow down

c) <u>methods that return a **`Task`** or **`Task<T>`**</u>

d) All of the above

Flash Quiz

② The **`await`** keyword should always be added to a method that returns a **`Task`** type

a) True

b) False

Flash Quiz

② The **`await`** keyword should always be added to a method that returns a **`Task`** type

a) True

b) False

Flash Quiz

③ Using **`ConfigureAwait(false)`** in Xamarin.Forms causes await to return on which thread?

a) UI thread (calling thread)

b) Always a thread pool thread

c) The thread that did the background work (which might be the UI thread)

Flash Quiz

③ Using `ConfigureAwait(false)` in Xamarin.Forms causes await to return on which thread?

a) UI thread (calling thread)

b) Always a thread pool thread

c) <u>The thread that did the background work (which might be the UI thread)</u>

Xamarin University

Flash Quiz

④ If you want to use a cross-platform approach to switching back to the UI thread, you should use ______________

a) `Dispatcher.BeginInvoke`

b) `SynchronizationContext`

c) `Device.BeginInvokeOnMainThread`

d) `RunOnUIThread`

Flash Quiz

④ If you want to use a cross-platform approach to switching back to the UI thread, you should use ______________

a) `Dispatcher.BeginInvoke`

b) **<u>`SynchronizationContext`</u>**

c) `Device.BeginInvokeOnMainThread`

d) `RunOnUIThread`

Coordinating multiple tasks

- Never forget that **await** *coordinates activity* – it can inadvertently **reduce** parallelism in your code

```
Task priceAA = ..., priceDelta = ..., priceUnited = ...;
List<Task> runningTasks = { priceAA, priceDelta, ... };

for (int i = 0; i < 3; i++) {
   await runningTasks[i];
   ... // Process results
}
```

What's the problem with this code?

Xamarin University

Coordinating multiple tasks

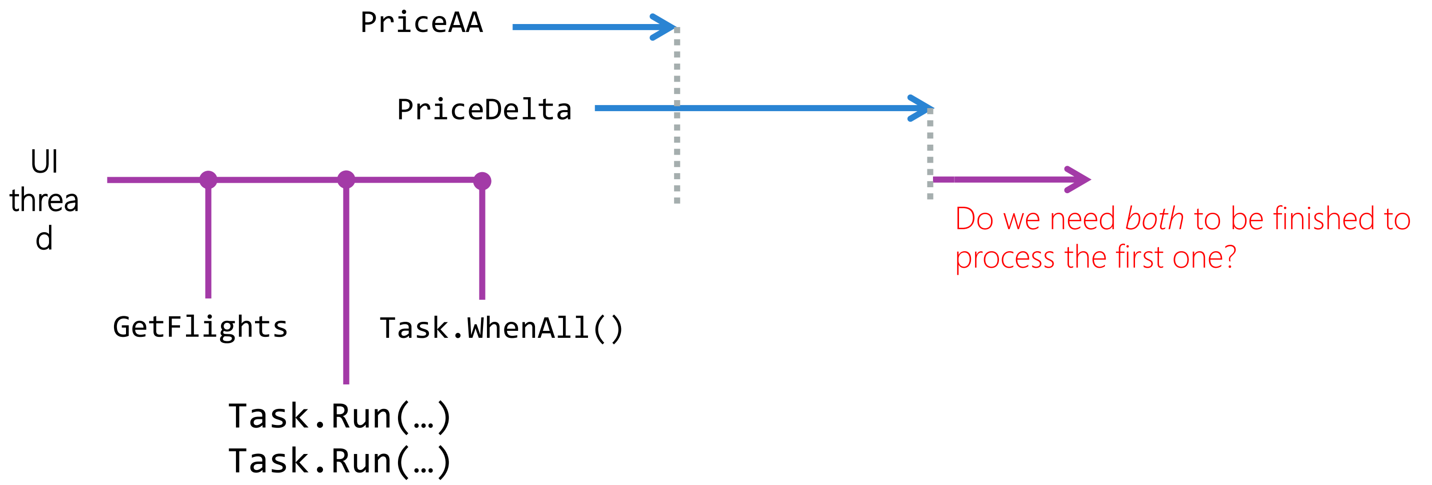
- Better to start multiple, related tasks to get data and then efficiently wait for all of them to be finished before processing the results

```
Task priceAA = ...;
Task priceDelta = ...;
Task priceUnited = ...;

// Wait for all prices to be available before updating UI
await Task.WhenAll(priceAA, priceDelta, priceUnited);

// All tasks are                                    to user
...
```

WhenAll returns a new **Task** which completes when all the passed tasks are finished

Efficient processing of multiple tasks

❖ Sometimes returning results can be processed as soon as the task is complete – independent of the other tasks we are waiting on



Efficient processing of multiple tasks

- Can use **`Task.WhenAny`** to return control when the *first* task is done – use a loop to continue processing the results in the order they are finished

```
List<Task> runningTasks = ...;
while (runningTasks.Any()) {
   // Wait for the first task to finish
   Task completed = await Task.WhenAny(runningTasks);
   // Remove from our running list
   runningTasks.Remove(completed);
   // Process the completed task
   CheckIfLowestPrice(completed);
}
```

Make sure to use **`await`**! **`Task.WhenAny`** returns a **`Task<Task>`** so code will compile *without* await but won't run properly!

Async / Await limitations

- Several restrictions placed on `async` / `await` usage by the compiler

No `out` or `ref` parameters

Return complex types from the Task instead – such as `Tuple<T1,T2>`

Async / Await limitations

- Several restrictions placed on `async` / `await` usage by the compiler

No `out` or `ref` parameters

Cannot use in constructors or property getters

Ctors and properties must return an immediate value, which is not possible

Async / Await limitations

- Several restrictions placed on `async` / `await` usage by the compiler

No `out` or `ref` parameters

Cannot use in constructors or property getters

Not allowed while in sync block (`lock`)

Can use different synchronization mechanism, or restructure code

Xamarin University

Async / Await limitations

- Several restrictions placed on async / await usage by the compiler

No out or ref parameters

Cannot use in constructors or property getters

Not allowed while in sync block (lock)

Limited support in LINQ queries

Can only use await in first select or group

Summary

1. What does the `await` keyword do?
2. Exploring the generated code
3. Dealing with return values

Where are we going from here?

- Async / Await are really nice ways to provide a convenient structure for *consuming* async code

- Next we will look at how to *write* async code using the Task Parallel Framework in **CSC351**

Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile

Microsoft