Asynchronous models and patterns

Florian Rappl, 29 Mar 2013

An introduction to async / await, popular mistakes and solutions for asynchronous programming, as well as usages and benefits from using asynchronous programming. We will also discuss interesting patterns based on concurrency.

Download sample code - 37 Kb

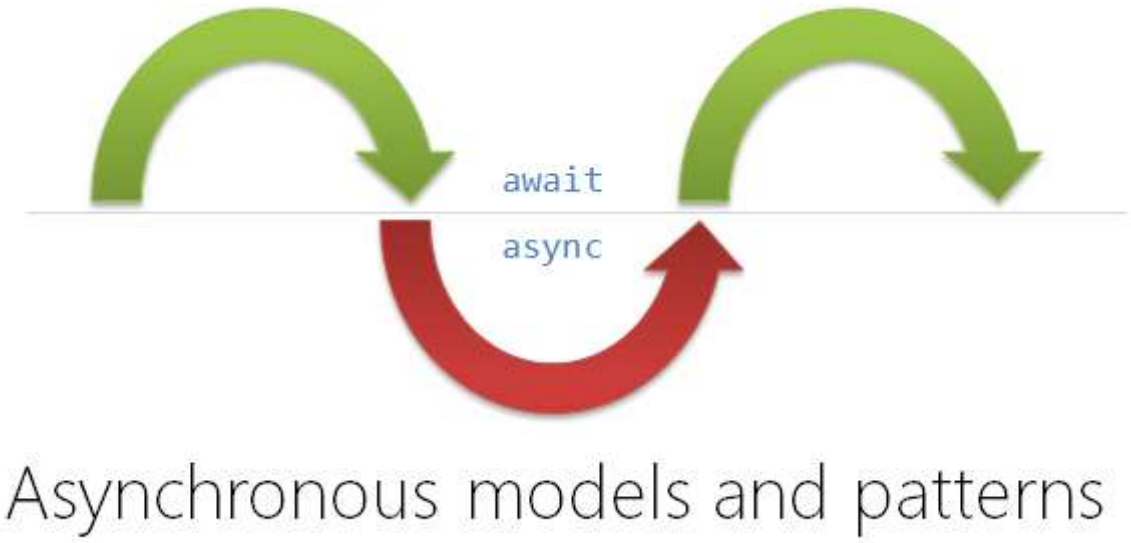


Table of Contents

Introduction

This article will discuss useful concepts and the general ideas behind using `async` / `await` in C# 5. We will investigate why this is useful and what the two keywords do exactly. Our journey will start identifying various lanes of program execution, called **threads**, and what the abstract concept of a **task** is.

All in all we will work towards some key points. The following lessons should be learned from this article:

- Avoid potential race conditions and what we can do to prevent them
- `async void` should be used exclusively for event handlers
- Asynchronous tasks can wrap around events, to make code simpler and provide locality
- Designing asynchronous APIs that will not result in potential problems
- Use threadpool for computationally-bound code - but not for memory-bound code

The initial question we have to ask, before going into the discussion is why we care about having multiple threads anyway.

Why multiple threads?

Moore's law states that the number of transistors on integrated circuits doubles approximately every two years. For decades this came along with increasing the number of operations per second of a single core. However, this was only possible, because the frequency also followed an exponential growth. Due to Dennard's law it was not possible to keep both values, frequency and density (transistor count), at an exponential growth. Therefore the frequency growing is now over and the only way to get more operations per second is to involve more cores in the computation, as well as improving the chip architecture.

In most applications we will not be computation bound (CPU-bound), i.e. we will not come close to the limits of the CPU. However, even 10 years ago, we have been memory bound (IO-bound) in most of our applications. Formerly this has been true for local IO, like memory or disk access. Nowadays more and more traffic is network IO, like downloading data from the Internet or some other computer in a network. While computation bound applications will certainly take the most benefit from multi-core CPUs, other applications can also benefit from having more cores.

Since the late 1980s every standard operating system (OS) allows us to use more than one application (even though there has been just one core). The OS does that by giving each application some time on the CPU. Once the time quota that has been given to one application is exceeded, the operating system will perform a context switch to some other application. Modern operating systems make use of multi-core CPUs by reducing necessary context switches and (additionally to the time management) CPU allocation management.

Usually the time management of modern operating systems is based on so called threads or processes. In Windows we have a thread based time management. This results in the paradigm that we should transfer CPU heavy computations to some different thread (different from the one running the GUI), such that the GUI can still do some other things. This made sense for 1 core (the OS will give the GUI thread some time and it will give the computation thread even more time), and makes even more sense for 2 cores (now it is possible that the computation thread is actually working fulltime until the job is done, while the GUI thread gets a little bit of time on the OS core). In Linux every thread is a process (simplified model as there are a few differences).

Now for memory bound methods it makes sense to use another thread in case of non-available asynchronous operations - just to keep the GUI active and responsive. If an asynchronous operation for doing the memory operation exists, this is certainly the preferred way. The argument is the same as for the computation bound method; the OS will schedule some time for the GUI thread as well as for the (waiting) worker thread. The argument for getting some benefit from using another core is also quite similar: In this case we might not need to perform context switches, giving us better performance and less wait time (however, the difference is only measurable in µs (10^-6 s)).

Therefore using threads is one way to tell the operating system which units of computation belong together. According to this the operating system will schedule computation time, CPU allocation and more. Dividing our applications into multiple threads will therefore guarantee better CPU usage, program flow and responsiveness. Using asynchronous operations when possible will give us benefits for having a responsive UI while not requiring the time to manage threads.

Race conditions

Looking at our program from the perspective of such threads will result in two different kind of variables. Those which are only in the scope of the thread, called **thread-local** (or sometimes **thread-static**) variables and those which are available to all threads. Such variables are called **global** variables. Of course we still have the local variables, but since the function's scope is always created by the thread invoking it, those variables belong also to the specific thread, as the thread-static ones do.

Already from the names we can assume that thread-local variables will not provide any problems. One thread uses its local variables until the thread is terminated. Only the thread itself can read and write the values of such variables, which is why there is no problem. However, in the case of global variables we are introduced to a new kind of problem. Depending on the structure of our application we might have the case where two or more threads access one variable. In case of reading the variable there is no problem, but once the value of the variable gets modified the problems start.

The first problem is obvious: What if multiple threads can write the value? Well, in such cases the last thread writing on the variable wins. This might not seem like a big problem at first, but it can result in unexpected behavior if the thread(s) that write the value want to use the value afterwards. In such cases the value might change and a different value is used afterwards, coming from some other thread.

The second problem is of course related to the first one. What if only one threads wants to write the value, with the other threads reading it? Even here we might have a problem, since we can not guarantee that the write happens before the reads or that the reads happen before the write.

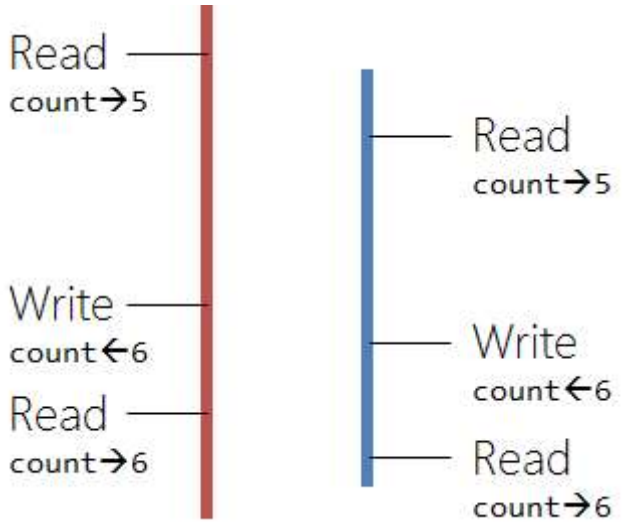
Those two problems lead to the term *Race Condition*, i.e. a state that is given when two threads do actually race for accessing a resource. Usually race conditions appear not very often (from the perspective of a computer) and unpredictable, however, given that most operations are executed in loops or regularly, it is very likely to see a race condition appearing in some codes quite often (from the perspective of the programmer). As an example: If a race condition appears once in a billion instructions (which not very often for a computer), we will see it every second (which is very often for us).

The solution to avoid race conditions is usually given by **atomic operations**. An operation is atomic if it appears to the rest of the system to occur instantaneously, i.e. if they either successfully change the state of the system, or have no apparent effect and if they can only be executed from one thread at a time. One way to force atomicity is by mutual exclusion, i.e. immutable objects. On a hardware level this has been done by a cache coherency protocol like MESI, or on the software level using semaphores or locks.

Let's have a look at a classic race solution in C# using the TPL and a parallel for loop to compute some value:

```
int count = 0;
int max = 10000;
Parallel.For(0, max, m => count++);
```

If we want to draw a sample picture for this, with some arbitray sample data, then we could do it with the following diagram (we have 2 threads, marked red and blue).



Now this is a really simple code. We compute the value of `count` by incrementing it in each iteration. So in the end, the value of `count` should be equal to `max`, which is the number of iterations. In any serial code (or for-loop), this is no a problem, however, once a

certain number of operations is reached, we will definitely get race conditions.

```
       10 op. in parallel resulted in zero race-conditions (00.0%) ...
      100 op. in parallel resulted in zero race-conditions (00.0%) ...
     1000 op. in parallel resulted in zero race-conditions (00.0%) ...
    10000 op. in parallel resulted in some race-conditions (14.8%) ...
   100000 op. in parallel resulted in some race-conditions (59.7%) ...
  1000000 op. in parallel resulted in some race-conditions (60.3%) ...
 10000000 op. in parallel resulted in some race-conditions (61.3%) ...
100000000 op. in parallel resulted in some race-conditions (64.0%) ...
```

Why is there a certain offset of operations before we get some race conditions? The answer is that spawning a thread (even though the TPL is just activating a thread that is already spawned on the optimized thread-pool) takes some time. Usually the thread has to change its state from idle or sleeping and requires the function to invoke. This function (with everything that is seen by the function, like local and global variables) requires a context switch to be performed.

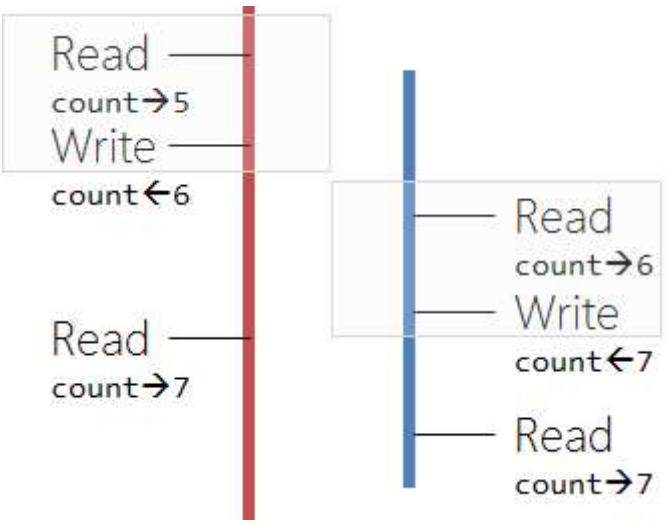
So this is a way to approximate the time for some thread to wake up. Here we see that 10000 operations are enough to get some race condition, while 1000 operations seem to be too few. Therefore we can state that the creation of a new task will take at least 1000 cycles (probably the real number is more like 5000 cycles, due to one iteration taking longer than 1 cycle).

What is the easiest way to get over this burden? Well, on a software level we already stated locks as one solution. Lucky for us that the C# team knew about this and included locks in the language. All we need is to give the `lock` an address, which will then be used to determine if the `lock` applies for the given block of statements and to wait if it is currently in use. Having a language that supports locks syntactically is a great productivity boost!

```
var lockObj = new object();
int count = 0;
int max = 10000;
Parallel.For(0, max, m => {
    lock(lockObj)
    {
        count++;
    }
});
```

Now this is working without any race condition, however, looking at the execution time, we might see that this is usually at least *N* times slower than the original version. Here *N* is the number of cores used. On my UltraBook with 4 cores I had 7804 ms compared to 1548 ms for doing the example provided in the given source code. The additional time is coming from thread creation and thread synchronization (the `lock` itself also costs some cycles, usually in the order of 1000 to 10000 cycles). Also note: depending on the current state of technologies like frequency scaling, Turbo Boost and more (additionally to the workload of the operating system) one might not see the difference of *N* times at random invocations.

Here is the picture how our two sample threads behave now:



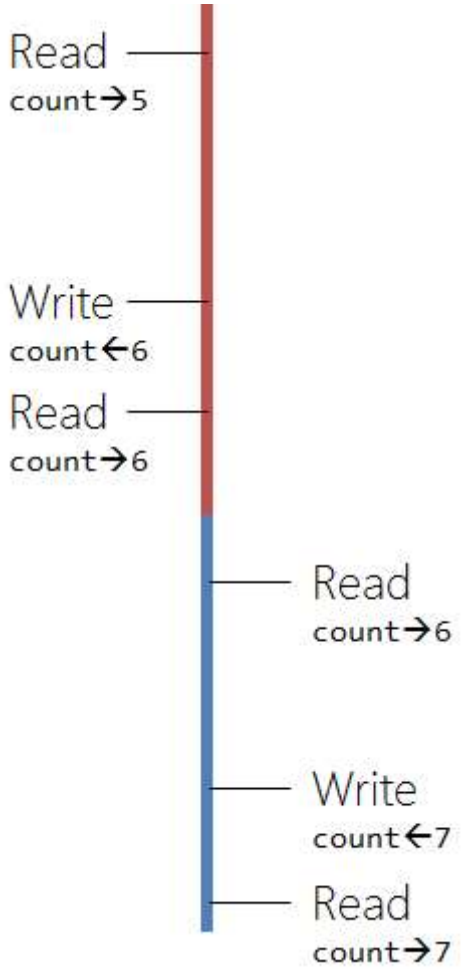
Generally it is a bad idea to make one "big" `lock` around a method that is invoked in parallel. There are two statements why:

1. If the method is the only thing that is executed by the thread, then this is a classic serial code, just with more overhead and therefore reduced performance. Not a good idea!
2. If the method is only one method that is called from a thread running in parallel, then the following code is much more convenient.

It is possible to state that a method should only run synchronized. Using our example we can do the following:

```
int max = 10000;
var sm = new SyncMethod();
Parallel.For(0, max, m => sm.Increment());
```

Running them synchronized represents the following diagram, which is basically the sequential (single-threaded) version.

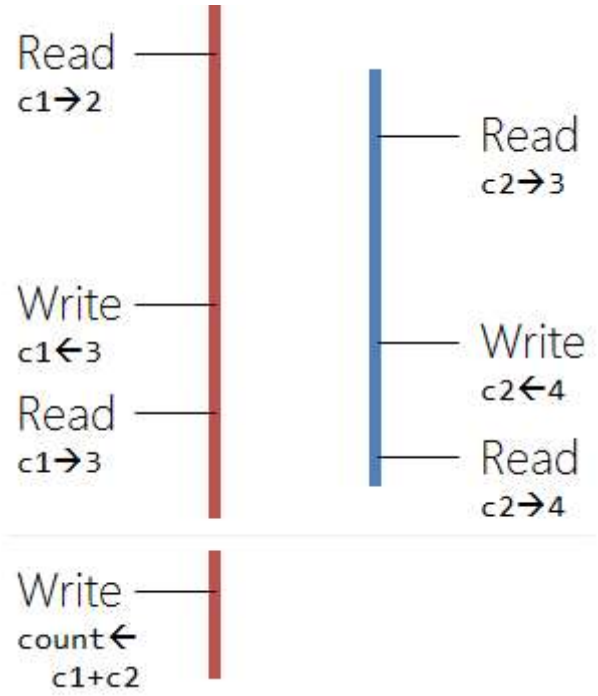


The implementation of the class `SyncMethod` looks like this:

```
class SyncMethod
{
    int count = 0;

    [MethodImpl(MethodImplOptions.Synchronized)]
    public void Increment()
    {
        count++;
    }
    public int Count
    {
        get { return count; }
    }
}
```

Generally this is a much faster and better way. It is also a nice way to keep certain operations atomic. On my machine the duration of the test method was 4478 ms - or about 33% faster than using `lock`.

The best option for our example is to use thread-local variables. Such variables make a counter for each thread, which then can be used to perform one reduction in the end. So we reduced lots of global operations to just one, which is invoked as the last step. Luckily for us the creators of the TPL knew about this problem and gave us an overload of the `Parallel.For` method, which creates and uses such a thread-local variable.



Looking at the picture above we see that both threads are doing independent work. Their (independent) results will then be used by one thread. It is worth noting that high performance libraries will perform this global operation (sum reduction in form of a gather) in a more sophisticated way. In networks (inter CPU) the topology of the network will be used, otherwise (if no topology is known, or we have the intra CPU case) a binary tree will give us a much better performance (the scaling using a binary tree is $log_2 N$ instead of plain sequential gathering, which scales like N).

```
var lockObj = new object();
int count = 0;
int max = 10000;
Parallel.For(0, max, () => 0,
    (m, state, local) =>
    {
        return local + 1;
    },
    local =>
    {
        lock (lockObj)
        {
            count += local;
        }
    }
);
```

On my machine the sample code runs in 461 ms, which is faster than the initial parallel code with the race condition problem. The reason for the speedup is the usage of a thread-local variable, which can be taken directly from the cores cache, instead of transferring it from some other core's cache or requiring the cache to be evicted. So we did not only get rid of the race condition, but also improved the performance.

Some languages (but not C#, since you can not overload the `=` operator) offer atomic objects. In C++ you can create objects like `atomic<int>` (or use ones provided by frameworks like the Intel Thread Building Blocks). Such objects will essentially do the same. They will be thread-local and perform global operations, once the total value is requested (in most implementations the request has to be done explicitly, in order to avoid overhead).

The .NET-Framework provides yet another way of doing some thread-safe operations for certain types and operations: the `Interlocked` class. It contains a set of helper methods like `Add()` for adding either two 4-byte integers (`Int32`) or two 8-byte integers (`Int64`). There are several other methods available, basically all to use the following assembler instructions:

- BT
- BTS
- BTR
- BTC
- XCHG
- XADD
- ADD
- OR
- ADC
- SBB
- AND
- SUB
- XOR
- NOT
- NEG
- INC
- DEC

Those instructions have locked counterparts in the X86 instruction set, i.e. they have the LOCK instruction prefix. So we could not use the `Interlocked` class for adding two doubles, but we can certainly use it in this scenario.

```
int count = 0;
int max = 10000;
Parallel.For(0, max, m => Interlocked.Increment(ref count));
```

So this is now very close to our serial code and it also performs quite well, however, as we've seen the `Interlocked` class can only be used in special scenarios. Needless to say that it should definitely be used in those scenarios (unfortunately most numeric applications require floating point arithmetic, which is excluded). The performance is certainly better than with the locked or synchronized ways, but its still way behind the optimal method of using thread-local variables to reduce communication.

Another way to avoid race conditions in more complex objects, like arrays, dictionaries and lists is to use the so called concurrent objects of the .NET-Framework. All those objects can be found in the namespace `System.Collections.Concurrent`. Here we have classes like `ConcurrentDictionary` or `ConcurrentStack` and more. All those objects provide a concurrent access to the members of the more elementary object, like the `List<T>` in case of the `ConcurrentList<T>`, however, they do not provide a thread-safe access to the underlying elements. A complete discussion of those objects will not be done in this article.

The await / async keywords

We already learned that computationally-bound tasks should be used with the TPL. Here using more cores makes sense, since it will speed up the computation. It also makes sense to use the TPL in a different thread, since then we will also keep our GUI responsive. For memory-bound tasks only the latter makes sense, i.e. we should not use the TPL, since using more cores / threads on a memory bound task only results in overhead. Just for clarification: Using a new thread for IO-bound methods makes only sense, if no non-blocking way of calling the method exists. In this case we use async over sync, which will be explained later.

In the history of .NET we will find several patterns that have been used to master the common task of spawning a new worker thread and handling results from this thread. The most known patterns are:

- Asynchronous Programming Model (APM), based on `IAsyncResult`
- Event-based Asynchronous Pattern (EAP), most known from the `BackgroundWorker` implementation
- Task-based Asynchronous Pattern (TAP), coming with the TPL and the `Task<TResult>` class

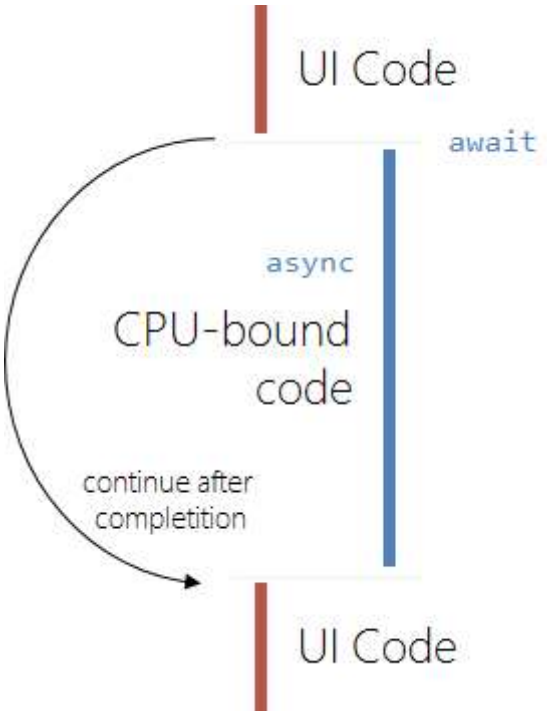
The most elegant and modern pattern is certainly TAP. There are big advantages of encapsulating a thread in a task. First of all we can connect various tasks and draw dependencies. Therefore we can create a whole tree of tasks, which results in a very elegant and expressive way of declaring a non-sequential program flow. Another feature is the improved exception handling and the optimized startup time. Tasks are managed by the TPL, i.e. from the programmer's perspective we do not care about the resources (threads) any more, but only on the subject which is the method to be invoked as an (independent) task.

The problem with tasks is, however, that they still require some lines of code. We still want the following to happen: Execute the first part of some method, invoke some other (probably memory-bound) part of the same method in a different thread and (once that task is completed) execute the last part of the method in the same thread as the first part. All those steps should still happen with exception handling and of course with our GUI still responsive.

Let's have a look at some sample code:

```
void RunSequentially(object sender, RoutedEventArgs e)
{
    //First part
    var original = Button1.Content;
    Button1.Content = "Please wait ...";
    Button1.IsEnabled = false;

    //Simulate some work
    SimulateWork();

    //Last part
    Button1.Content = original;
    Button1.IsEnabled = true;
}
```

So here we have our 3 regions. We can easily see that those 3 regions can be expanded to *N* regions. The only point we care about is the alternation of code that must be invoked from the GUI thread and code that should run in a non-GUI thread in order to keep the GUI responsive. Let's express this 3 region model in a simple yet illustrative picture.



A very simple modification of the code (no exception handling etc.) looks like the following:

```
void RunAsyncTask(object sender, RoutedEventArgs e)
{
    //First part
    var original = Button2.Content;
    Button2.Content = "Please wait ...";
    Button2.IsEnabled = false;

    //Simulate some work
    var task = Task.Factory.StartNew(() => SimulateWork());

    task.ContinueWith(t =>
    {
        //Last part
```

```
        Button2.Content = original;
        Button2.IsEnabled = true;
    }, TaskScheduler.FromCurrentSynchronizationContext());
}
```

Two points are important here. The first point is that we actually create a new running task that has to be continued with some other task. That continuation is very important, otherwise we either have two concurrently running tasks (by just creating another task) or some code that runs before the work is actually started (without wrapping the code in another `Task` at all - remember that there is some startup time for any thread). The other point is that we need to use the `TaskScheduler.FromCurrentSynchronizationContext()` method to get a `TaskScheduler` instance, which uses the current GUI thread. If we do not use this, then we will face a cross-threading exception, since we will access GUI elements from a non-GUI thread.

Now with C# 5 we can write this a lot simpler. Let's have a look at the code first:

```
async void RunAsyncAwait(object sender, RoutedEventArgs e)
{
    //First part
    var original = Button3.Content;
    Button3.Content = "Please wait ...";
    Button3.IsEnabled = false;

    //Simulate some work
    await Task.Run(() => SimulateWork());

    //Last part
    Button3.Content = original;
    Button3.IsEnabled = true;
}
```

This looks nearly identical to the sequential version. We only changed two lines - the method definition has an additional `async` keyword and the `SimulateWork()` method is now called with an encapsulated `Task`. Additionally the spawned `Task` is also awaited using the `await` keyword.

It should be noted that the `async` keyword does not spawn a new thread, nor does it do anything asynchronous out-of-the-box. The keyword just wraps the contents of the method into `Task`. This also enables the `await` keyword.

Async over sync

The example shows something that is considered **async over sync**. Additionally we will have a look at **sync over async**. The other two ways are (trivially) sync over sync (just call it sync!) and async over async (just call it async!). Using the `Task.Run()` method to get some async over sync is definitely a possibility. We can also use this to make an old and out-dated (sync) API async. Consider the old method:

```
void SimulateWork()
{
    //Simulate work
    Thread.Sleep(5000);
}
```

Now we can wrap this sync method to become async:

```
async Task SimulateWorkAsync()
{
    await Task.Run(() => SimulateWork());
}
```

Generally it is a good practice (or to be more precise: the only practice one should consider) to return a `Task` from an async method. Usually one would prefer a `Task<TResult>`, i.e. something like the following:

```
double ComputeNumber()
{
```

```
    //Simulate work
    Thread.Sleep(5000);
    //Return some number
    return 1.0;
}

async Task<double> ComputeNumberAsync()
{
    return await Task.Run(() => ComputeNumber());
}
```

That way the task had some meaningful goal - computing a double number. Giving tasks results is not always possible and does not always make sense, however, if possible the technique should definitely be applied.

Sync over async

The other way is also possible. If one has a real async process (like network or file system operations) then exposing the usage as a sync method does not make much sense. Instead one should always expose method calls in an async way - just to keep going with the flow. If some user now wants to call the async method sequentially, then this is easily possible by using some features of the `Task` class.

Let's have a look at a simple sample code:

```
public async Task<string> ReadUrlAsync(string url)
{
    var request = WebRequest.Create(url);

    using (var response = await request.GetResponseAsync())
        using (var stream = new StreamReader(response.GetResponseStream()))
            return stream.ReadToEnd();
}
```

Here we are using the `GetResponseAsync()` method of the `WebRequest` class. It would make no sense to hide this async behavior of the lower level. Now the user has basically two options. The usual and preferred option is to use the async way.

```
async Task ReadUrlIntoTextBoxAsync()
{
    Output.Text = string.Empty;
    ButtonSync.IsEnabled = false;

    string response = await ReadUrlAsync("http://www.bing.com");

    Output.Text = response;
    ButtonSync.IsEnabled = true;
}
```

Now we can also use this in a sync way, naming the call sync over async.

```
void ReadUrlIntoTextBox()
{
    Output.Text = string.Empty;
    ButtonSync.IsEnabled = false;

    var task = ReadUrlAsync("http://www.google.com");
    task.Wait();
    var response = task.Result;

    //Or alternatively just use:
    //var response = ReadUrlAsync("http://www.google.com").Result;

    Output.Text = response;
    ButtonSync.IsEnabled = true;
}
```

This way seems trivial at first, but our API needs to be redesigned in order to support it correctly. We will discuss the mistake in depth in the Common mistakes section. For now all we need to know is that the following API would work without any problems:

```
public async Task<string> ReadUrlAsync(string url)
{
    var request = WebRequest.Create(url);

    using (var response = await request.GetResponseAsync().ConfigureAwait(false))
        using (var stream = new StreamReader(response.GetResponseStream()))
            return stream.ReadToEnd();
}
```

So using `Wait()` we can block one thread to wait for completion of some task. Alternatively we can use the `Result` property, which implicitely uses `Wait()` until the result of the task is set. There is also a method called `RunSynchronously()`, however, this one cannot be performed in this case, since `RunSynchronously()` may not be called on a task which is not bound to a delegate, such as the task returned from an asynchronous method.

Usages and benefits

The most obvious usage of using `await` is to keep the GUI responsive. However, we will see that there is a lot more possible. In the patterns section we will discover ways of improving our code by keeping local states local.

Keeping the GUI responsive is a good argument for using `await`, but what about `async`? Obviously `async` is required for using `await`. But there is much more behind it. Marking functions as `async` in some API should only be done with real asynchronous functions, otherwise (if it is in fact just a async over sync function) the API has been designed wrong. Keep in mind that every programmer could use async over sync, hence providing additional functions which just use async over sync is definitely wrong.

So what is the lesson here? Do not lie about your APIs. If you are doing some sync stuff, then expose it as such. If you are doing some really async stuff, then expose it with `async` (and following the convention: the method name should end with *Async*). If you want to expose a sync version of an async function, then do not just use sync over async, but in a real sequential way of writing the function.

The benefit is quite clear: If the creator of an API did not lie, the user will always know what's going on behind the curtain. If it is some new task that will be spawned, or not, and if it should be awaited or not. Always think about those 70 ms, which is approx. the max. time that any method call should take. If its longer, then your UI will most probably not feel very responsiveness. Therefore APIs should be designed in a way to let the user know which version to take.

The right SynchronizationContext

A big issue is the synchronization context, i.e. the thread which performs the work. Most of the time we will not specify a particular context, however, in case of GUI applications we want all actions that change the state of controls to happen in the default synchronization context.

This is actually a big point in favor of using `async` and `await`. Usually this will just work, since the default setting is to switch the context after the `await`statement. This ensures that the code after the task will actually run from the GUI thread. However, we might not want that. There are several reasons to avoid this, most notable performance and reliability. Let's see how we can control this:

```
//This code runs in the GUI thread
var request = WebRequest.Create(url);
//This code runs in a different thread
var response = await request.GetResponseAsync().ConfigureAwait(true);
//This code will run in the GUI thread
/* ... */
```

The call of `ConfigureAwait(true)` is somewhat unnecessary, since this is already the default value. Nevertheless, we also have a glance at the other possibility:

```
//This code runs in the GUI thread
var request = WebRequest.Create(url);
//This code runs in a different thread
var response = await request.GetResponseAsync().ConfigureAwait(false);
```

```
//This code will generally run in a different thread than the GUI thread
/* ... */
```

This way we keep the main benefit from `await`, which is using an asynchronous method without getting headaches in terms of exception handling, continuation and more, while not using the UI thread for the continuation. This is actually a best practice for writing APIs, since we do not want to unnecessarily block the UI.

The issue with the synchronization context is not new and Microsoft has a generic answer since a long time: the `SynchronizationContext` class! In the past I've written a number of APIs that had restrictions on the framework version or profile. Some of those restrictions made it impossible to expose an asynchronous API or to use tasks / threads, however, it was quite obvious that the API will be used (in some cases) asynchronously. In this scenario there are only two options:

1. Being lazy and let the programmer who actually uses the API find out that all events inside the API must be re-routed to the GUI synchronization context.
2. Being nice and already fire the events from the same synchronization context as the GUI.

While number 1 is obviously the preferred option for some programmers (also from some .NET-Framework developers in the early versions; where even some async exposed APIs do not fire the event in the right context), option 2 is much less work than it sounds. Basically the following construct is required:

```
class SomeAPIClass
{
    SynchronizationContext context;

    public event EventHandler Changed;

    public SomeAPIClass()
    {
        //We either take the currently default one or we create a new one
        //if no default sync. context exists (e.g. console projects, ...)
        context = SynchronizationContext.Default ?? new SynchronizationContext();
    }

    public void SomeMaybeAsyncCalledMethodWithAnEvent()
    {
        /* Do the work */
        RaiseEventInContext();
    }

    void RaiseEventInContext()
    {
        //Do all the "normal" work
        if(Changed != null)
        {
            //Use the synchronization context to fire the event in
            //the right context
            context.Post(_ => Changed(this, EventArgs.Empty), null);
        }
    }
}
```

It is worth noting that `SynchronizationContext.Default` will automatically be set by instantiating a new `Form` in case of Windows Forms or by the `Application` class in WPF. Is is also important to remember that this default context is thread-bound, i.e. if our API class is already instantiated in some thread, there is no way to get the default synchronization object for the GUI thread. So this way only works if the objects have been instantiated from the GUI thread, but (some) methods of the object have been called from a different thread.

This also brings up the problem that the API has to be thread-safe. If some methods are not thread-safe, then either the API design is wrong, or accessing them from various threads should be restricted. In either way we probably made some programming or design mistakes, which should be handled. Most problems occur from `static` function calls that rely (read and probably write) on `static` members.

The `SynchronizationContext` offers two methods: `Send()` and `Post()`. Since we are using it with event handlers, we should always prefer the `Post()` method. It is asynchronous and does not catch on exceptions. This "fire-and-forget" method is exactly what

we want in our case. Otherwise it might make sense to use `Send()`. Here we can catch exceptions from the method invocation and we would be able to run commands sequentially.

It is important to understand that the `SynchronizationContext` class does not perform all the magic. Instead it is a specific implementation that is really useful. Sometimes it could make sense to use our own implementation of the class with the `Send()` and `Post()` methods, or to just use our own definition in case of `SynchronizationContext.Default` being `null`. More information on the `SynchronizationContext` can be found in this article by Leslie Sanford.

Common mistakes

The biggest mistake that people make when using the new keywords is to use `async void`. There is only one reason to use `async void` and that is the case if somebody wants to enable `await` in the code of an event handler. Since handlers have a fixed signature, we cannot change the return type of `void` here. Usually this is no problem, since the only methods that should call such methods are wired up internally and would not benefit from awaiting the handler call.

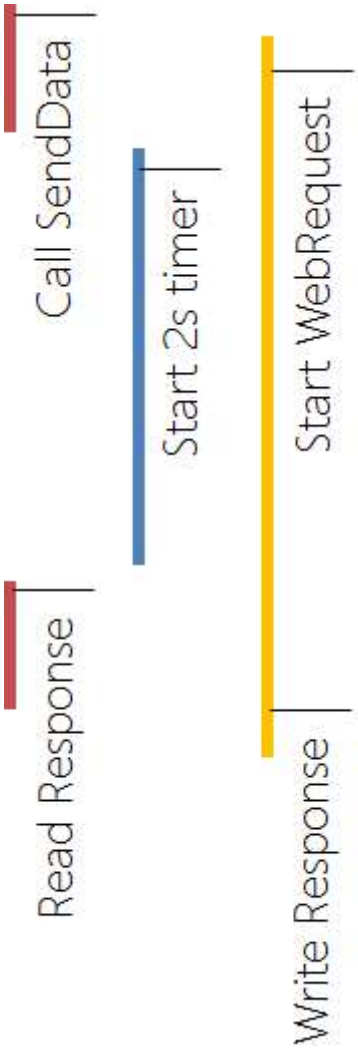
Let's have a look at some problems that might arise using `async void`. Let's have a look at the following code:

```
string m_GetResponse;

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);
    }
}

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();
}
```

There are two major problems here:

1. Since we are not returning a `Task`, we will not be able to catch exceptions!
2. We are not awaiting the web request... What if the request takes longer than those 2 seconds?

Illustrating this in a picture yields the following result:



This has actually been copied from a Windows store app example on MSDN. Since then they obviously got informed about this buggy code and replaced it. The cure is also quite straight forward:

```
string m_GetResponse;

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

async Task SendDataAsync(string url)
{
    var request = WebRequest.Create(url);
    using (var response = await request.GetResponseAsync())
        using (var stream = new StreamReader(response.GetResponseStream()))
            m_GetResponse = stream.ReadToEnd();
}
```

Another popular example from StackOverflow is the following code:

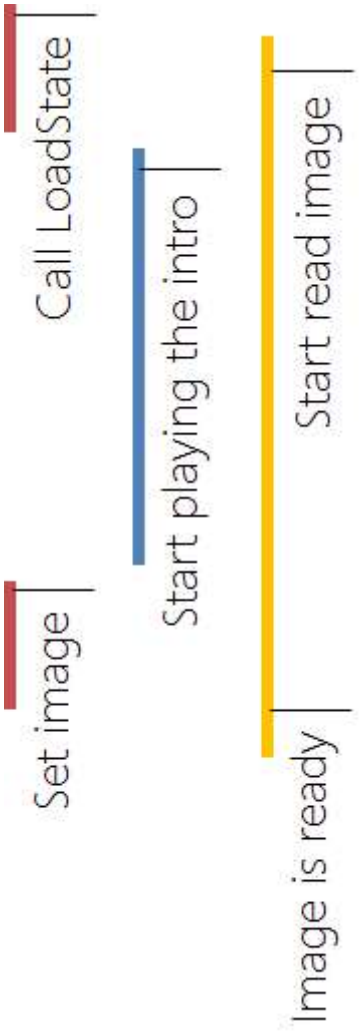
```
BitmapImage m_bmp;

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

protected override async void LoadState(Object nav, Dictionary<String, Object> pageState)
{
    m_bmp = new BitmapImage();
    var file = await StorageFile.GetFileFromApplicationUriAsync("ms-appx:///pic.png");
    using (var stream = await file.OpenReadAsync())
        await m_bmp.SetSourceAsync(stream);
}
```

The problem with this code snippet has been that it sometimes shows that both, `PixelWidth` and `PixelHeight`, are 0. To see the problem one has to know a little bit more about the base class (not the current one):

```
class LayoutAwarePage : Page
{
    string _pageKey;

    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        if (this._pageKey != null)
            return;

        this._pageKey = "Page-" + this.Frame.BackStackDepth;
        /* ... */
        this.LoadState(e.Parameter, null);
    }
}
```

So now this is pretty obvious. The code is actually calling the `OnNavigatedTo()` method of the base class, which is then calling the current `LoadState()` implementation. Nothing is awaited of course, so our code is continuing with awaiting the `PlayIntroSoundAsync()`. Now this is a kind of race condition. If the `PlayIntroSoundAsync()` method finished before the `LoadState()` method, we will see that the width and height of the image are still 0. Otherwise the code will work.

Illustrating this in a picture yields quite the same result as before:



Again we can construct a very simple cure. Of course we cannot touch the internals of the `Page` class - and we also should not touch the internals of the class `LayoutAwarePage`. Then a solution looks like the following:

```
Task<BitmapImage> m_bmpTask;

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

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

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

So we are still starting the task in the `LoadState()` method, but we are not relying on that task being finished in the `OnNavigatedTo()` method. Instead we are awaiting the end of that task (in case of that task being ended already, there is no wait time). So again, using a `Task` as return type did solve this problem quite elegantly.

That being said it is obvious that `async void` is a "fire-and-forget" mechanism. The caller is unable to know when an `async void` has finished and is unable to catch exceptions thrown from it. Instead exceptions get posted to the UI message-loop. Therefore we should use `async void` methods only for top-level event handlers (and their like) and return `async Task` everywhere else.

To close the discussion about `async void`, we have to realize that we will face this disaster in some places, where we did not expect it. So even though we believe to do everything right, we might still get in trouble. Let's consider the following code:

```
try
{
    await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, async () =>
    {
        await LoadAsync();
        m_Result = "done";
        throw new Exception();
    });
}
catch (Exception ex)
{ }
finally
{
    DebugPrint(m_Result);
}
```

Everything seems alright does not it? Usually an `async` lambda expression returns a task, however, lambda expressions are also restricted to their proposed (or required) signature(s). In this case the signature is given by the `DispatchedHandler` delegate. This delegate, however, returns `void`. So we will not catch the exception and we will also have a kind of race condition, since the finally block is entered at the same time as the `m_Result` variable is set.

The interaction model of those instructions can be represented as the following diagram:

Therefore: **Watch out for lambdas**, i.e. verify the signature of the lambda and be sure that it returns a `Task`.

Another common mistake has already been mentioned while discussing proper API design. Consider the following piece of code:

```
public async Task<string> ReadUrlAsync(string url)
{
    var request = WebRequest.Create(url);

    using (var response = await request.GetResponseAsync())
        using (var stream = new StreamReader(response.GetResponseStream()))
            return stream.ReadToEnd();
}

void ReadUrlIntoTextBox()
{
    Output.Text = string.Empty;
    ButtonSync.IsEnabled = false;

    var task = ReadUrlAsync("http://www.google.com");
    task.Wait();
    var response = task.Result;

    Output.Text = response;
    ButtonSync.IsEnabled = true;
}
```

What's wrong here? Well, obviously the return type of the `async` method is fine, since we are returning a `Task<string>`. In the `ReadUrlIntoTextBox()` method we are using this method sequentially, by calling the `Wait()` method of the task. Here is where the problem lies (and calling the `Result` property would result in the same disaster). What are we doing exactly?

1. First we just setting some UI components.
2. Then we are calling the `ReadUrlAsync()` method.
3. We create a `WebRequest` instance and start a new task.
4. By using the `await` we are placing a callback (continuation) **in the GUI thread**.
5. Now we are waiting until the task completes, by **blocking the GUI thread**.
6. This is it, since the GUI thread is now blocked and no continuation will be able to run there.

This is kind of a chicken and egg problem. The continuation (required for the task to finish) requires the GUI thread, which is blocked until the task is finished. The solution has already been presented:

```
public async Task<string> ReadUrlAsync(string url)
{
    var request = WebRequest.Create(url);

    using (var response = await request.GetResponseAsync().ConfigureAwait(false))
        using (var stream = new StreamReader(response.GetResponseStream()))
            return stream.ReadToEnd();
}
```

This way the continuation does not require the GUI thread. Therefore the continuation is not blocked by the `Wait()` method and everything works as expected. The question here is - why is this the default way? Well, obviously for the standard programmer (mostly consuming APIs) it is much more convenient to always come back to the GUI thread. Hence such programmers have a lot less work. On the other side API developers are considered more sensitive about such problems, which is why the default has been in favor of API-consumer programmers.

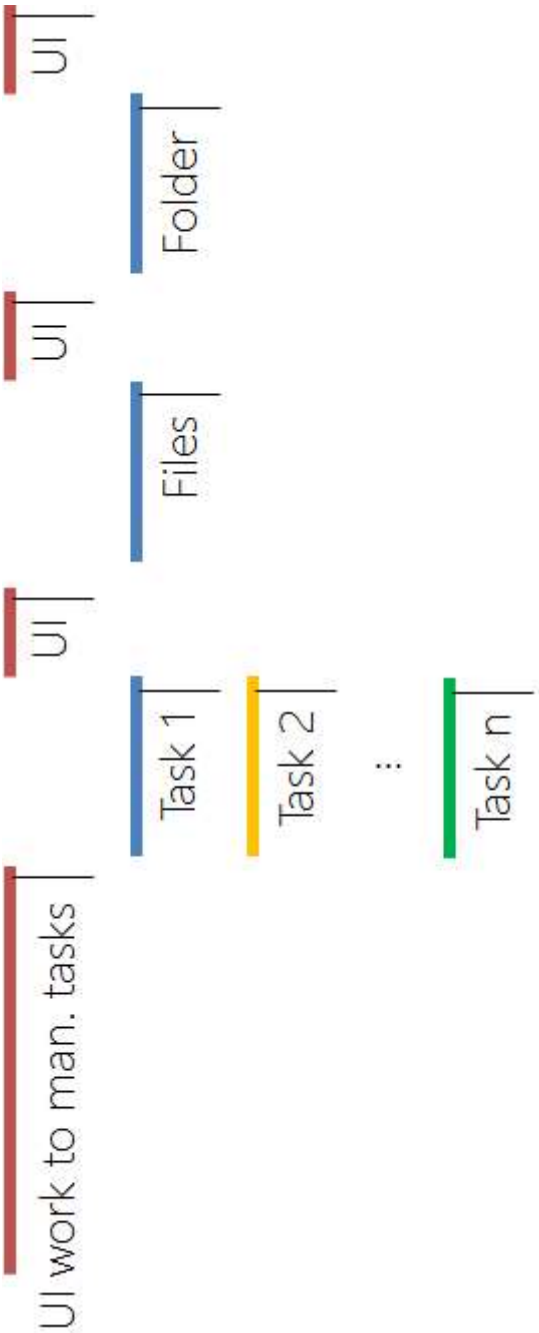
The last common mistake lies again in the difference of memory-bound and CPU-bound tasks. If we have some legacy code that should run in a different thread, or some CPU-bound task, then we should use `Task.Run()`. It will create a background-thread, i.e. it wil use one of the threadpool threads. This is alright. But if we just want to start some memory-bound operation, until it is done, then the right thing to do is to use `TaskCompletionSource<T>`.

The following code shows a scenario where we artificially put stress on the CPU, just by opening too many threads at a time. This will eventually lead to a scenario where memory-bound work is transformed to partly CPU-bound work, which is then handled on the GUI thread:

```
protected override async void LoadState(Object nav, Dictionary<String, Object> pageState)
{
```

```
    var pictures = new ObservableCollection<House>();
    listbox.ItemsSource = pictures;

    var folder = await Package.Current.InstalledLocation.GetFolderAsync("pictures");
    var files = await folder.GetFilesAsync(CommonFileQuery.OrderByName);
    var tasks = from file in files select LoadHouseAsync(file, pictures);

    await Task.WhenAll(tasks);
}
```

Let's try to create a picture of what's going on first:



The code looks really nice, but it has one major drawback. In the real code several thousand pictures have been loaded. Now this resulted in several thousand `await` statements (and as many threads). This caused a lot of overhead and the UI was not responsive for more than a second. Not very nice! The solution is quite simple - don't try to perform huge quantities of `await` at the same time; process them sequentially.

Asynchronous Design Patterns

For years we've been waiting for this: Having multi-threaded code that looks like sequential code. This all is possible by using `async` in combination with `await`. It is probably the right place to say, that this is not parallel code which looks like sequential or serial code. There is a huge difference. Parallel code is realized by using the TPL (or self-spawning all the workers, most efficiently over an optimized thread-pool) and requires a different kind of synchronization. We've seen a very simple (and useless) parallel code in the beginning with the increment of the `count` variable.

The question now is: What else can we do with this new elegance of writing multi-threaded code? In the next subsections I try to come up with some design patterns, which could reduce complexity, strengthen maintenance and provide an elegant solution to some problems.

Async Event Pattern

This pattern has been introduced by Lucian Wischik. For more details about him consider the points of interest section.

Sometimes state machines (even for really simple systems) tend to blow up. They are expanding, and become more and more complicated. Suddenly the whole code looks like madness, with no one knowing what's the purpose of the most methods there. The problem here is usually over-generalization. The solution has been quite hard, since keeping parts of the code more local required even more classes, restrictions and more. The main problem, however, was that for keeping something more local, one had to use different threads (since the UI thread is blocked otherwise).

Now this problem is quite easy to solve. We just introduce extension methods like the following:

```
public static async Task WhenClicked(this Button button)
{
    var tcs = new TaskCompletionSource<bool>();
    RoutedEventHandler ev = (sender, evt) => tcs.TrySetResult(true);

    try
    {
        button.Click += ev;
        await tcs.Task;
    }
    finally
    {
        button.Click -= ev;
    }
}
```

Now we can easily use some (existing) button to come back to our (local) code, once it has been clicked:

```
/* Update UI */
await button.WhenClicked();
/* Update UI */
```

Before we would have had 2 methods and one handler for the button. The first method would have been the initial state, the second method the final state and the event handler would have been required to go from the initial to the final state, once the initial state has been entered. If that does not sound complicated to you then I don't know - for me it sounds too complicated, especially since (usually) we will have several states and those 2 states (initial, final) would be just 2 states of out *n* states (*n* is usually bigger than 10 in such scenarios).

Now we could just introduce it to one method. This one method is one of the top level states. Hence we have a nice grouping.

We can also use tasks to make our media interactions a lot easier to manage. Let's consider the following code:

```
protected async void OnResume(Navigation.NavigationEventArgs e)
{
    //Start playing sound
    var rustleSoundTask = rustleSound.PlayAsync();
```

```
    using(var wobbleCancel = new CancellationTokenSource())
    {
        //Animation
        foreach(var apple in freeApples)
            AnimateAppleWobbleAsync(apple, wobbleCancel.Token);

        //Wait for the sound to finish
        await rustleSoundTask;
        //And cancel all outstanding tasks
        wobbleCancel.Cancel();
    }
}
```

So here we are starting the playback of sound. Usually we would require some event to recognize the end of the playback. Now we are just doing it using the `await` mechanism. While the sound starts playing we are busy spawning more tasks, which will perform some animation (shaking the free hanging apples). Now that all our media is wrapped as tasks and busy running, we will wait for the sound task to stop. Once this is done we will just cancel the animation (enough shaking). Keeping all those actions in sync would have been quite a headache, but `async` and `Task` made it quite easy.

One remark to the code above: The usage of `AnimateAppleWobbleAsync()` without `await` will end up in a warning. To get rid of the warning one has to do something with the `Task` instance. Usually we would assign the instance to some variable, but in this case this does not make much sense. Therefore it might be meaningful to use something like the following:

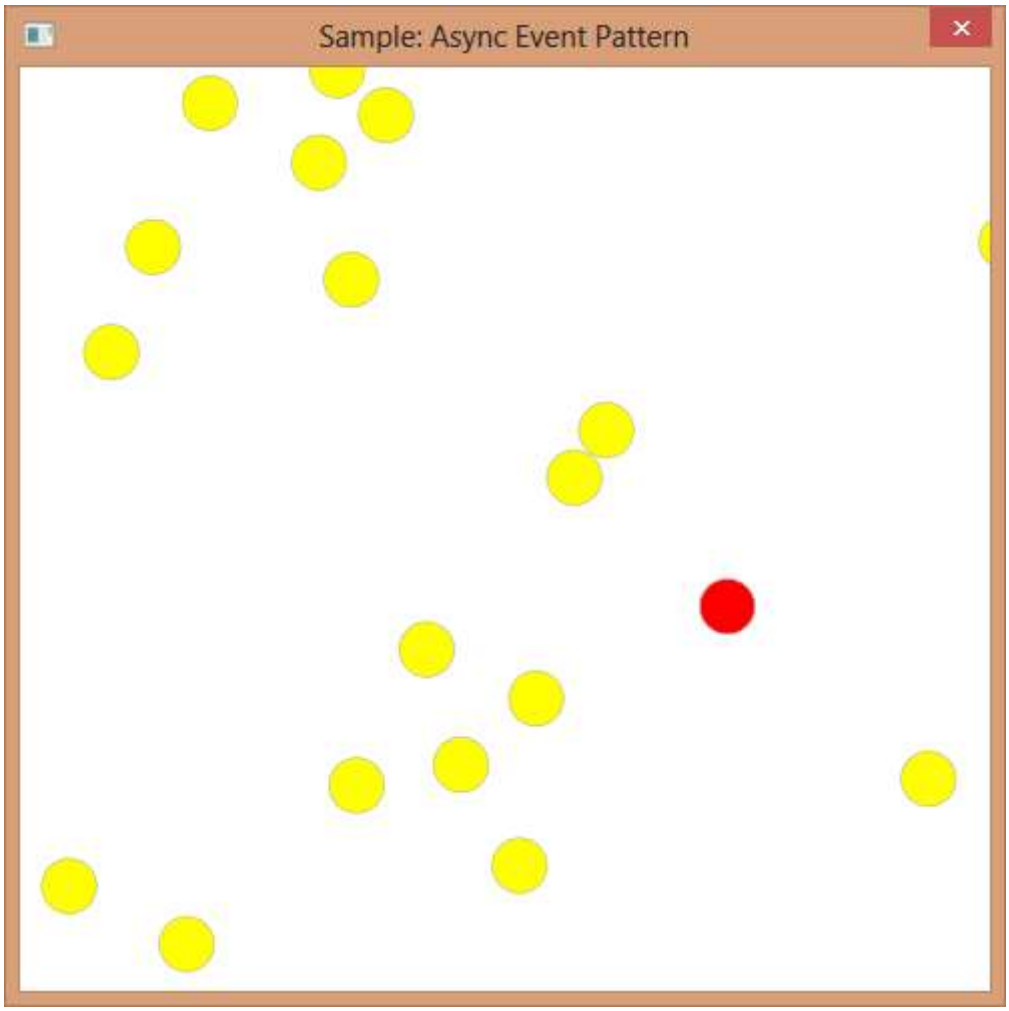
```
public static void FireAndForget(this Task task)
{
}

public static void FireAndForget<T>(this Task<T> task)
{
}
```

Those extension methods do nothing other than telling the compiler that we actually use the instance for something. For us more important is that they tell the user, that the "fire-and-forget" mechanism has been used intentionally.

The example code for this pattern will contain a small game, which has the objective of clicking at randomly moving yellow circles. Clicking on a red circle will decrease the amount of circles. Once all circles have been clicked away, the game is over. The whole game is basically boiled down to one method:

```
async void ButtonClicked(object sender, RoutedEventArgs e)
{
    Random r = new Random();
    var active = new List<GameEllipse>(ellipses);
    var cts = new CancellationTokenSource();
    startButton.Visibility = Visibility.Hidden;
    ShowEllipses();

    Task movingEllipses = MoveEllipses(cts);
    var time = DateTime.Now;
    movingEllipses.Start();

    while (active.Count > 0)
    {
        int index = r.Next(0, active.Count);
        var selected = active[index];
        selected.Shape.Fill = Brushes.Red;
        await selected.Shape.WhenClicked();
        selected.Shape.Fill = Brushes.Yellow;
        selected.Shape.Visibility = System.Windows.Visibility.Hidden;
        active.RemoveAt(index);
    }

    var ts = DateTime.Now.Subtract(time);
    cts.Cancel();
    MessageBox.Show("Your time: " + ts.Seconds + " seconds.");
    startButton.Visibility = Visibility.Visible;
}
```

The following picture shows a screenshot of the sample:



We do not care about the button click, which is why we just have a "fire-and-forget" method here. The important part is in the code. We could make the game a lot more complicated and still preserve a kind of overview, just by localizing states in methods. Here variables that are only used when the balls are moving are placed in the method.

The `MoveEllipses()` method call returns a task. That task will continuously move the ellipses, until the cancellation token is used.

```
Task MoveEllipses(CancellationTokenSource cancel)
{
    return new Task(() =>
    {
        while (!cancel.Token.IsCancellationRequested)
        {
            Dispatcher.Invoke(() =>
            {
                for (int i = 0; i < ellipses.Count; i++)
                {
                    var ellipse = ellipses[i];
                    ellipse.UpdatePosition();
                    Canvas.SetLeft(ellipse.Shape, ellipse.X);
                    Canvas.SetBottom(ellipse.Shape, ellipse.Y);
                }
            });

            Thread.Sleep(10);
        }
    }, cancel.Token);
}
```

We see that all ellipses are actually moved, even though only active ones are visible. So this implementation could be improved in that area. On the other side we use the `Dispatcher` to move the ellipses in the GUI thread. Every 10 ms such a move is actually performed.

Async Model Binding

We all know and love MVVM, or at least the binding capabilities of WPF. A possible drawback is that binding usually has to happen sequentially, i.e. there is no direct way of telling the binding engine to perform the binding in a different thread, with the interaction to the control being handled in the GUI thread.

However, we might want to bind our data sources or some fields to some web requests or other sources, which might take a while to transmit the required data to our client. In any case we do not want to lose responsiveness in our application, due to the binding engine.

There are many stages to async model binding. This section will NOT be about how to create `async` properties - this is not possible. It would be nice to bind such values, but do not think about it as a restriction, but more as a motivation and guideline. It should motivate us to use caching and to request changes over methods. Such (asynchronous) methods should then change the value of the properties.

The same is true for constructors. Both, constructors and properties, should be lightweight. There is a simple way around using an asynchronous constructor - that is a factory method or something that follows this private constructor pattern:

```
class MyLazyClass
{
    private MyLazyClass()
    {
        /* some light initializations */
    }

    async Task DoSomeHeavyWork()
    {
        /* the long taking initializations */
    }

    public async static Task<MyLazyClass> Create()
    {
        var m = new MyLazyClass();
        await m.DoSomeHeavyWork();
        return m;
    }
}
```

There are other advantages of this pattern (like the ability to return `null` in case of failure or the ability to return a more specialized version if certain criteria are met), but this section will be more about the model binding issues. The asynchronous factory and similar helpers are presented for completeness only.

The interesting fact about MVVM in WPF is that in WPF, if a background thread modifies a databound property, the binder itself will marshal that change back to the UI. Even more interesting: New in .NET 4.5 is the ability to use the `BindingOperations.EnableCollectionSynchronization()` method for update (add, change, delete) values in databound collections like `ObservableCollection<T>`. This way we can easily add values to the collection from another thread.

So having no problems with collections is one problem less. However, most of the times we will spawn new threads or wait for some request in case of user-input, i.e. of some button is pressed or more generally if a command is invoked. The problem is, that `ICommand` implements everything in a "fire-and-forget" paradigm (returning `void`).

The key is therefore to avoid relying on the code spawned by the command and use the command only to trigger an asynchronous code in the view model. The view model is then controlling the execution state by changing some property. This property is then used by the command to determine if the command is executed at the moment. If it is executed, then we should skip execution (i.e. the buttons associated to that command should be disabled), otherwise we would allow it (i.e. the corresponding buttons would be enabled).

Let's see some sample code, which depends on a class called `AsyncViewModel`:

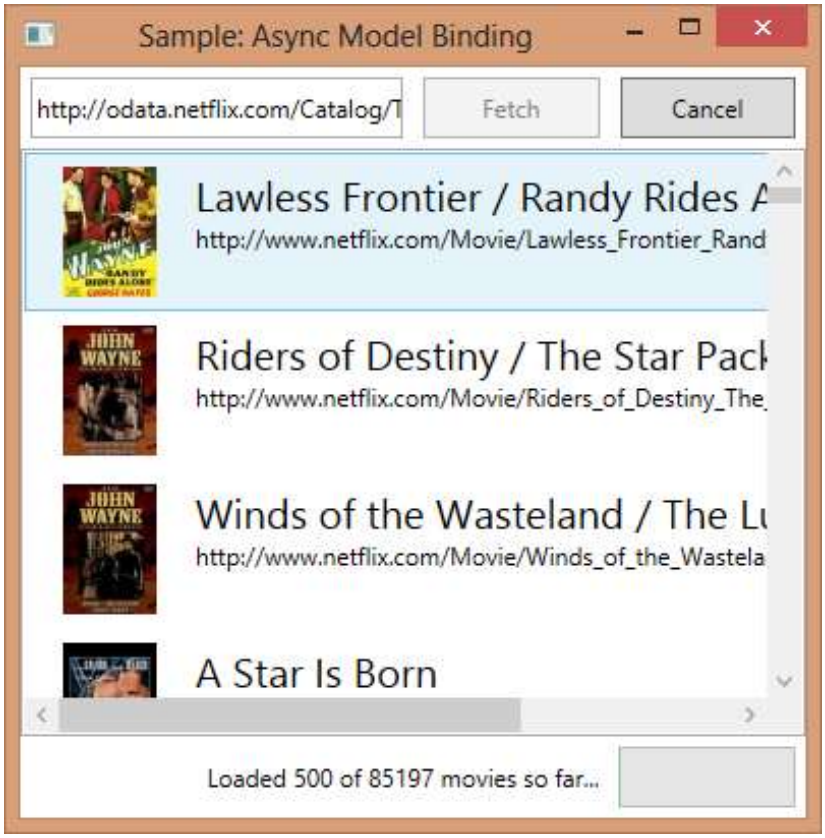
```
public class FetchCommand : ICommand
{
```

```
    AsyncViewModel vm;

    internal FetchCommand(AsyncViewModel vm)
    {
        this.vm = vm;
    }

    internal void Invalidate()
    {
        CanExecuteChanged(this, EventArgs.Empty);
    }

    public bool CanExecute(object parameter)
    {
        return !vm.Fetching;
    }

    public event EventHandler CanExecuteChanged;

    public void Execute(object parameter)
    {
        vm.DoFetch();
    }
}
```

Obviously we use the property `Fetching` to see if the code is already executing. If it is, then we can not execute the command. The `Invalidate()` method is used to invoke the `CanExecuteChanged` event by the view model. It should be called when the `Fetching` property changes. A sample implementation of the property is as easy as the following code snippet:

```
public bool Fetching
{
    get { return (bool)GetValue(FetchingProperty); }
    set { SetValue(FetchingProperty, value); }
}

public static readonly DependencyProperty FetchingProperty =
    DependencyProperty.Register("Fetching", typeof(bool), typeof(AsyncViewModel), new PropertyMetadata(
        new PropertyChangedCallback((d, e) =>
        {
            var vm = (AsyncViewModel)d;
            vm.Fetch.Invalidate();
            vm.Cancel.Invalidate();
        })));
```

The given example in the source code is a modified version of Lucian Wischiks sample code for obtaining the list of movies from Netflix.



The picture above is a screenshot of the sample application.

Async Pipelining

Another really useful option for using a `Task` is given when continuous work is required. An example that is quite illustrative is a queue that will process images - processing an image might take a while, however, once processing an image is done either another one should be processed (if available) or nothing should be done. This is the case where we just put in some object in a queue, that will get handled once we have resources for handling it.

Everyone has already used something quite like this, the message loop in a Windows application. Here we might do some heavy processing (as we have learned, the processing should not be too heavy in order to keep the UI responsive), however, once the processing is done we are receiving the next instructions. In this case those instructions might end up in calling an event handler.

The queue must be concurrent, otherwise we cannot process incoming images and work on a current image simultaneously. The pattern requires the following ingredients:

- A processing "kernel", i.e. a method that will do some work.
- An invocation point, i.e. a method or constructor that manages the queue.
- An `async` queue handler, i.e. a method that will actually push the work.

In case of the example provided in the source code the `async` queue handler looks like the following. We just simulate some work on a class called `WorkItem`, which only has two fields. One field called `WorkTime` carries a random generated number, which basically tells our queue how long the (simulated) work should be. Additionally we put in options to cancel the current process as well as the overall process and we write some log. So the kernel is in this case just the `Delay()` method, however, in reality it could have been something a lot more sophisticated.

```
async Task QueueHandler()
{
    RaiseLog("Kernel started.");

    do
    {
        WorkItem current = null;

        if (!queue.TryDequeue(out current))
```

```
            break;

        cancelCurrent = new CancellationTokenSource();
        RaiseLog("Element {0} has been dequeued.", current.Id);

        try
        {
            //Here we actually call the kernel
            await Task.Delay(current.WorkTime, cancelCurrent.Token);
            RaiseLog("Element {0} has been completely processed.", current.Id);
        }
        catch (TaskCanceledException ex)
        {
            Debug.WriteLine(ex.Message);
            RaiseLog("Processing element {0} was cancelled.", current.Id);
        }

        cancelCurrent = null;
    }
    while (!cancelAll.IsCancellationRequested);

    cancelAll = null;
    RaiseLog("Kernel stopped.");
}
```

The invocation point is called `Push()`. Here we enqueue new items and (re-)start the handler, if necessary.

```
public void Push(WorkItem item)
{
    queue.Enqueue(item);
    RaiseLog("Item {0} enqueued.", item.Id);

    if (cancelAll == null)
    {
        cancelAll = new CancellationTokenSource();
        kernel = QueueKernel();
    }
}
```

So where does `async` get handy in here? Writing a continuation loop without `await` has been a real pain. Now we get exceptions handling and everything for free, and we do not have to care about blocking our UI. The blocking parts have are wrapped in `async` methods which are just awaited. We do not care about the internal structure, like a `while` loop, or the exception handling, but we focus purely on the queue logic.

Promises

I will hopefully find the time to include this soon.

Points of Interest

Some of the work presented here is based on a talk of Lucian Wischik, who works on the C# language design team and was heavily involved in introducing the new keywords. He also inspired me for the asynchronous patterns section. His blog is available at blogs.msdn.com/b/lucian/.

The WinRT API includes a lot of `async` marked methods. Some of them do not offer sequential versions. It is therefore important to understand, that `async` / `await` are two very important keywords, which make it impossible for any programmer to blame the programming language for not implementing multi-threaded code. More than a decade ago, multi-threaded code should already have been mandatory for any programmer (slow disk access, even slower network times, ...), but nowadays with super-responsive apps and more network access than ever, this seems like the only right thing to do.

The minimum requirement for `async` / `await` has been set to .NET-Framework 4.5 with C# 5. However, it is possible to install C# 5 on development computers, which do not have access to Visual Studio 2012. The async CTP gave every computer with Visual Studio 2010 a

preview version of C# 5, which is nearly identical to the release version.

Finally also the framework requirement (and that is much bigger problem, since this is not only required for compilation, but also for running the application) has been lowered from .NET-Framework 4.5 to .NET-Framework 4. It is possible to access the `async` / `await` feature in .NET 4 by using the **AsyncTargetingPack** (see NuGet, Microsoft.CompilerServices.AsyncTargetingPack). The requirement is, however, that Visual Studio 2010 (VS11) is installed.

Conclusion

Working with asynchronous methods has gained a lot of interest and momentum since `async` / `await` have been introduced. A lot of good APIs, applications and helpers have been written with the new keywords. Most of the problems that have been seen can be cured by not using `async void` and by drawing a picture of the program flow. Realizing that (even though with the `await` keyword the code looks sequential) the program flow is not sequential anymore will help in detecting most problems.

There are several reasons for using the new keywords and multi-threading in general, but two reasons are more important than others:

- The ability to reduce complexity by keeping local states local.
- The power to have an ultra-responsive UI.

Therefore it does not only make a lot of sense to use the new abilities for creating new patterns, improving UI performance and having more options (like stopping running tasks), but also to design APIs with asynchronous operations in mind.

History

- **v1.0.0** | Initial Release | 14.03.2013
- **v1.0.1** | Some minor clarifications and corrections | 15.03.2013
- **v1.1.0** | Typos, source update and async pipelining | 28.03.2013
- **v1.1.1** | Corrected a little typo | 28.03.2013

License

About the Author

Florian Rappl

Architect

Germany

Florian lives in Munich, Germany. He started his programming career with Perl. After programming C/C++ for some years he discovered his favorite programming language C#. He did work at Siemens as a programmer until he decided to study Physics.

During his studies he worked as an IT consultant for various companies. After graduating with a PhD in theoretical particle Physics he is working as a senior technical consultant in the field of home automation and IoT.

Florian has been giving lectures in C#, HTML5 with CSS3 and JavaScript, software design, and other topics. He is regularly giving talks at user groups, conferences, and companies. He is actively contributing to open-source projects. Florian is the maintainer of AngleSharp, a completely managed browser engine.

Comments and Discussions

52 messages have been posted for this article Visit **https://www.codeproject.com/Articles/562021/Asynchronous-models-and-patterns** to post and view comments on this article, or click **here** to get a print view with messages.