Coding exercise – Async/await

The focus of this exercise is asynchronous programming in C#. You are going to use the *Task-based asynchronous pattern* (TAP). There are other patterns for asynchronous programming in C#, like the *Event-based Asynchronous Pattern* (EAP), but this exercise will only focus on TAP.

The exercise is targeted at people who want to get into asynchronous programming at a beginner level. If you are familiar and comfortable with most of the asynchronous concepts and are using them in you daily work, then this exercise is not for you. On the other hand, if you never used aync/await before, or if you have only been introduced to it but never really got comfortable with it, then this exercise is right up your alley.

|  |  |
| --- | --- |
| Synchronous | Asynchronous |
| synchronous breakfast | when any async breakfast |

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/

# The setup

Do this exercise as a **Mob**.

Since this exercise will most likely be done in Teams sessions, you should use the “rules” of Mob Programming as a guideline. Decide if you want to change driver/navigator at fixed time intervals during each session or if you want to take turns being the driver for an entire session. You should also consider how to share code, e.g. by using a network drive, creating a repo on GitHub, whatever works best for you.

# The exercise

The first part of the exercise is a step-by-step guide, to get you familiar with the basic concepts of asynchronous programming in C#. Once you have learned the basics, you’ll be introduced to more advanced topics.

Try to follow the steps to the letter and don’t skip ahead. There is quite a lot of text in this exercise, since it is both hands-on *and* a bit of theory. But don’t let that discourage you. You’ll be guided through all the concepts. Please take the time to read through every step, perhaps by having one group member read it out loud. Make sure you understand the content before moving on.

## The basics

1. Since this exercise is about asynchronous programming, the groundwork has already been done for you in terms of an application with a functioning UI. Locate the file named **AsyncKata1.zip** in the same location as where you found this document. Download and extract the zip file and open the solution in Visual Studio  
   **NB!** The solution was created in VS2017. If, for some reason, you are having issues running the solution, create a new solution and copy-paste the classes and resources from **AsyncKata1.zip** into it.
2. Let’s have a look at the program you’ll be working on. Before diving into the code, try to run the program. Hit the button and see what happens.
3. The first thing that probably comes to mind when you run the program is; holy crap, whoever wrote this must be a UX designer grand master?!? Once the initial excitement has settled, have a look at the code:
   1. First thing to notice is, that this is a .NET 4.7.2 WPF application
   2. Have a look at *App.xaml.cs*. In this class, OnStartup() has been overridden and serves as composition root. In here, a view and a viewmodel are being instantiated and the viewmodel is set as context for the view. In GPS, the framework called Prism handles this, but let’s stick to the simple approach.
   3. Have a look at MainWindowViewModel. It has 3 public properties: StartStopCommand, WorkOngoing and StatusMessage. It implements the INotifyPropertyChanged interface and raises the PropertyChanged event whenever WorkOngoing or StatusMessage changes. StartStopCommand is of type ICommand. In GPS, an implementation of this interface is provided by the Prism framework. You are not going to use Prism in this exercise, so a simple implementation of ICommand is provided by the class RelayCommand. Whenever the command fires, it updates WorkOngoing and StatusMessage.
   4. Finally, have a look at MainWindow in the file *MainWindow.xaml*. The main window contains a grid, which contains a button, an image, and a label. The button has a command binding to StartStopCommand and the label has a binding to StatusMessage. Furthermore, the button and the image each have a style, which have a binding to WorkOngoing. The value of WorkOngoing determines the color and content of the button and whether the image should rotate.
4. To illustrate the importance of asynchronous programming, you need to introduce something that takes up execution time. Create a new project in the solution of type Class Library (.NET 4.7.2) and call it *WorkerLibrary*.
5. Get rid of any template classes (e.g. *Class1.cs*)
6. Add an interface to the *WorkerLibrary* project and call it IWorker. Remember to make it public.
7. Add a method to the interface called DoWork() with return type void.
8. Add a class to the *WorkerLibrary* project and call it Worker. Make this class implement IWorker.
9. In the DoWork() method, add the following:   
     
   Thread.Sleep(5000);  
     
   This simulates a job that takes 5 seconds. It could be a complex calculation, a HTTP call, connecting to a hearing aid, whatever. To keep it simple, DoWork() is just going to “pretend”, by doing nothing for 5 seconds.
10. In MainWindowViewModel, add a constructor parameter of type IWorker and assign it to a private member called *m\_Worker*. You need a project reference to *WorkerLibrary* but hopefully ReSharper will fix that for you.
11. Go to *App.xaml.cs*, create a new instance of Worker and inject it into MainWindowViewModel.
12. Now, let’s make the program do something that takes time once the button is clicked. Rewrite MainWindowViewModel.StartStopCommandExecute() as follows (notice the call to m\_Worker.DoWork()):  
      
    private void StartStopCommandExecute(object obj)

{

WorkOngoing = true;

StatusMessage = "Work ongoing";

m\_Worker.DoWork();

StatusMessage = "Work done!";

WorkOngoing = false;

}

1. Run the application. Click the button.
2. What happens? The status message is updated before and after DoWork(), but the image doesn’t rotate! In fact, the entire UI becomes non-responsive when DoWork() is being executed. Try clicking the button and then immediately after drag the application window to the other side of the screen.  
   The issue lies with Thread.Sleep(). You might argue that this is a silly example but try to think of Compass GPS. What happens when you connect to a hearing aid? The entire GPS application freezes until the connection is established. Thread.Sleep() just simulates that behavior, but the mechanism is exactly the same. Doing work that takes time on the UI thread will block the UI and make it non-responsive.
3. So, how do you fix this? DoWork() has to execute while having a responsive UI. In the *Task-based asynchronous pattern* (TAP), this is done by wrapping whatever takes time in a Task. A description of Tasks can be found in the appendix *What is a Task?* on page 20. It is recommended that you spend 5 minutes reading it.
4. Let’s wrap the thing that takes time in the application. Change Worker.DoWork() to the following:  
     
   public void DoWork()

{

var t = new Task(() =>

{

Thread.Sleep(5000);

});

t.Start();  
}  
  
So what changed? A new Task object is created which takes an Action as input. In that Action, Thread.Sleep(5000) is executed. Finally, the Task is started and DoWork() returns.

1. Run the program and see what happens when you click the button.
2. Nothing happened, right? Well something did happen. Notice that the label changed to “Work done!”. The problem is that the command handler of the button click is not waiting for the result of the Task. The rotation animation has to run *while* the Task executes, and the label has to be updated *after* the 5 seconds have passed. Try to add a breakpoint in MainWindowViewModel.StartStopCommandExecute() before and after DoWork() is called. In Worker.DoWork() add a breakpoint on t.Start(); and on the closing bracket of the Task definition });. Click the button and step through the breakpoints. This gives you an idea of the execution path.
3. Let’s inject the functionality that needs to be executed *after* the 5 second sleep completes:
   1. First, update the signature of DoWork(), both in the interface IWorker and Worker class.  
        
      public interface IWorker

{

void DoWork(Action finalize);  
}

* 1. Next, call the injected functionality once the 5 seconds are up:  
       
     public void DoWork(Action finalize)  
     {  
      var t = new Task(() =>  
      {  
      Thread.Sleep(5000);  
      finalize?.Invoke();  
      });  
       
      t.Start();  
     }
  2. Add a private method in MainWindowViewModel, called FinalizeWork(), and update StartStopCommandExecute():  
       
     private void StartStopCommandExecute(object obj)

{

WorkOngoing = true;

StatusMessage = "Work ongoing";

m\_Worker.DoWork(FinalizeWork);

}

private void FinalizeWork()

{

StatusMessage = "Work done!";

WorkOngoing = false;  
}

1. Run the application and try to click the button. It works! Right?
2. Well, besides not handling what happens when you click the stop button (don’t worry, that comes later), there is another major problem with this implementation. The UI is being updated from within a Task. Now, in this case it doesn’t crash, since updating the UI is decoupled from the logic, i.e. only the viewmodel properties are updated in the Task, not the UI elements themselves. Still, you should **never ever** update the UI from anywhere besides the UI thread. If you want to explore how this can go horribly wrong, pause the exercise for a minute and do the bonus assignment in the appendix, *Updating the UI in a Task* on page 21.
3. The old-fashioned way of updating the UI after a Task completes, is to introduce a continuation for that Task. Whatever is defined in the continuation is executed once the Task completes. However, you need to execute the continuation on the context of the UI thread, not the context of the Task (which is the ThreadPool context).  
   Let’s remove the invocation of the injected Action (finalize) from the Task body and define a continuation for the Task instead:

public void DoWork(Action finalize)  
{  
 var t = new Task(() =>  
 {  
 Thread.Sleep(5000);  
 });

t.ContinueWith(

x => finalize?.Invoke(),

TaskScheduler.FromCurrentSynchronizationContext());

t.Start();  
}

1. Run the application again. Now it works and without the risk of exceptions.
2. In the code in step 22, the *TaskScheduler* of the continuation has been specified to that of the UI *SynchronizationContext*. The concept of context is quite difficult to get your head around but is a key concept of asynchronous programming. It is highly recommended that you spend 5-10 minutes, reading the appendix *Context, SynchronizationContext and TaskScheduler* on page 20. If not now, then perhaps before the next session. As a minimum, you need to know, that the UI thread runs in a context, and a Task running on a ThreadPool thread runs in a different context. All UI elements are created in the context of the UI thread, hence only the UI thread is allowed to update those elements (objects). Which is why the continuation has to switch back to the UI context.
3. The code works now, but it’s quite cumbersome and doesn’t scale nicely if you need to offload multiple work items to the TreadPool. This is where async/await comes into play. It offers a simple way to write asynchronous programs without the hassle of writing Task continuations and switching context manually. The compiler will handle all of that for you.

Let’s see what all the fuzz is about. First off, revert the injection of the finalizeAction in DoWork() and have it return a Task instead of void.

public interface IWorker

{

Task DoWork();

}

public Task DoWork()

{

var t = new Task(() =>

{

Thread.Sleep(5000);

});

t.Start();  
   
 return t;

}

1. Next, let’s introduce the preferred way of creating Tasks when using async/await, i.e. by using Task.Run():  
     
   public Task DoWork()

{

return Task.Run(() => Thread.Sleep(5000));

}

Task.Run() creates a Task and starts it immediately upon creation. This is preferred since you should **always** return a Task that has been started. Otherwise the client code has to guess if the Task has been started or not.

1. Next, lets update MainWindowViewModel.StartStopCommandExecute() to make use of the async/await pattern. First, get rid of the method FinalizeWork() and put the content of that method back into StartStopCommandExecute()

private void StartStopCommandExecute(object obj)

{

WorkOngoing = true;

StatusMessage = "Work ongoing";

m\_Worker.DoWork();

StatusMessage = "Work done!";

WorkOngoing = false;

}

Then, introduce the async and await keywords:

private async void StartStopCommandExecute(object obj)

{

WorkOngoing = true;

StatusMessage = "Work ongoing";

await m\_Worker.DoWork();

StatusMessage = "Work done!";

WorkOngoing = false;

}

1. Run the program and see if everything works.
2. Let’s dive into what happens in StartStopCommandExecute(). The *async* keyword just indicates to the compiler, that this method is asynchronous and can *await* Tasks. When you click the button, StartStopCommandExecute() starts executing on the UI thread, until it hits the await statement. When that happens, the execution is suspended until the Task being awaited completes, i.e. the Task returned by m\_Worker.DoWork(). To do this in a non-blocking way, the context is captured and stored when execution hits the await statement, and control is handed back to the UI thread so it may do other work. When the Task completes, the remaining work after the await statement is scheduled on the captured context, in this case the UI thread. This is exactly the same as before with the continuation (ContinueWith) of the Task. This syntax just makes it a lot easier.  
     
   If you want a more detailed introduction to async/await, head to the article linked below:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/>

1. To get a better understanding of the execution path, try to add breakpoints in StartStopCommandExecute() on the line containing the await statement as well as the line above and below it. Run the program, click the button, and observe when the breakpoints are being hit. Add a few breakpoints in Worker.DoWork() too if that helps.
2. Let’s go back to Worker.DoWork(). Imagine that this method itself had to call a subroutine asynchronously. This is where async await really comes in handy. Add a method called BackgroundWork() and call it from DoWork():

public async Task DoWork()

{

await BackgroundWork();

}

private static Task BackgroundWork()

{

return Task.Run(() => Thread.Sleep(5000));

}

1. Run the program and check that it still works (of course it does!).
2. Let’s have a quick look at the code above. The actual creation of the Task has just been moved to BackgroundWork(). However, notice that DoWork() uses the async keyword and still returns a Task, although it doesn’t actually have a return statement. Async methods that don't contain a return statement usually have a return type of Task. Such methods return void if they run synchronously. If you use a Task return type for an async method, a calling method can use an await operator to suspend the caller's completion until the called async method has finished. This makes it easy to nest asynchronous work. Just imagine the nightmare it would be to do this with *ContinueWith* statements and context switching.
3. Let’s explore the concept of *async Task* in a bit more detail. Try to copy paste the line   
   await BackgroundWork() a few times in DoWork() to see how it works. Use breakpoints in DoWork(), BackgroundWork() and StartStopCommandExecute().   
   **Revert** it when you’re done, so that await BackgroundWork() only appear ones in DoWork().
4. So far, you’ve used the return type *void* and *Task* in your async methods. There is a third option, which is *Task<TResult>*. Often a Task produces some result, like a boolean, an integer or even a complex reference type. This is where *Task<TResult>* comes into play. Unlike async methods with return type *Task*, async methods of return type *Task<TResult>* contains a return statement in which the operand is of type *TResult*. Let’s see what that looks like. Change the return type of DoWork() and BackgroundWork() to Task<string> and don’t forget to update the interface IWorker. Then, return a string in BackgroundWork() and return that same string in DoWork().  
     
     
     
   public async Task<string> DoWork()

{

string result = await BackgroundWork();

return result;

}

private static Task<string> BackgroundWork()

{

return Task.Run(() =>

{

Thread.Sleep(5000);

return "Work Done!";

});

}

1. DoWork() has been made explicit to illustrate the concept. However, you would probably write it like this:

public async Task<string> DoWork()

{

return await BackgroundWork();

}

1. So, what’s going on? When DoWork() hits the await statement, it suspends execution until the Task that is being awaited completes. When that happens, the result of that Task, in this case a string, is returned to the caller. So now, all that’s left to do, is to update the top-most caller in the stack to actually use this return value. Change MainWindowViewModel.StartStopCommandExecute() as follows:

private async void StartStopCommandExecute(object obj)

{

WorkOngoing = true;

StatusMessage = "Work ongoing";

StatusMessage = await m\_Worker.DoWork();

WorkOngoing = false;

}

1. Now StartStopCommandExecute() suspends execution until DoWork() completes, then takes the result and updates the StatusMessage property, which in turn updates the UI. The await statement captures the context before suspending and returns to that context once the awaited Task completes. In this case, the captured context is the UI thread. Hence StatusMessage is being updated on the UI thread, exactly as it should be.  
   Run the application and check out the result. If you struggle to understand the concept, breakpoints can really help in understanding the execution path.
2. This concludes the basics. A final word of advice; you have now used all 3 return types of an async method (*Task*, *Task<TResult>* and *void*). However, you should **never** use *async void* except for event handlers and command handlers. Which is exactly what you’ve used it for in this exercise. Had DoWork() been defined as *async void* instead of *async Task*, you wouldn’t have been able to await the result. Hence *async void* becomes a “fire and forget” mechanism in which you don’t know when the job has finished. This works fine for event handlers, since whoever fired the event shouldn’t care when listeners are done handling that event. But in all other cases, use *Task* or *Task<TResult>*.

## Cancelling a Task

You might have been wondering why the current implementation never bothered to handle what happens when you click “Stop” in the application. You’ll be handling this now!

1. There are a few ways to stop a running Task, the most widely used of which, is to use a *CancellationToken*. A CancellationToken enables you to cancel a Task from the caller and at the same time determine how that cancel operation should be handled within the Task itself.  
   Pass a CancellationToken as argument in DoWork() (remember to update the interface):  
     
   public async Task<string> DoWork(CancellationToken cancellationToken)

{

return await BackgroundWork();

}

1. Pass the token on to BackgroundWork() and then on to Task.Run() as argument. Having a CancellationToken as argument to Task.Run() ensures, that the Task is not started if cancel has already been requested on the CancellationToken, but it doesn’t handle cancellation requests once the Task is running!

public async Task<string> DoWork(CancellationToken cancellationToken)

{

return await BackgroundWork(cancellationToken);

}

private static Task<string> BackgroundWork(CancellationToken cancellationToken)

{

return Task.Run(() =>

{

Thread.Sleep(5000);

return "Work Done!";

}, cancellationToken);  
}

1. Now you need to define what happens, if the CancellationToken gets a cancellation request. Of course, in this example the Task doesn’t do much but in real life scenarios, it is very convenient to be able to finish up what you are doing, before finishing the Task. Let’s make something simple to see how it works. Rewrite BackgroundWork() as follows:

private static Task<string> BackgroundWork(CancellationToken cancellationToken)

{

return Task.Run(() =>

{

var count = 0;

while (count < 10)

{

if (cancellationToken.IsCancellationRequested)

{

return "Work cancelled!";

}

Thread.Sleep(500);

count++;

}

return "Work Done!";

}, cancellationToken);

}

1. Instead of sleeping for 5 seconds straight, BackgroundWork() now sleeps for 500ms at a time and checks if cancellation was requested before the next sleep. If cancellation was indeed requested, the Task returns. This is not too different from how you would write something like a polling Task. Say you want to poll a piece of hardware for 5 seconds at a 500ms interval. You want to be able to cancel that operation, but not in the middle of a poll. That could have unexpected behavior. Instead, this gives you a mechanism to check for cancellation requests between each poll.
2. MainWindowViewModel.StartStopCommandExecute() is still not handling clicks on the Stop button. So, let’s fix that now. In MainWindowViewModel, define a private member of type *CancellationTokenSource*:  
     
   private CancellationTokenSource m\_Cancel;  
     
   A CancellationTokenSource contains a CancellationToken. You don’t call *Cancel* on the token itself but on its source.
3. Next, you need to rewrite StartStopCommandExecute() to be able to handle both Start and Stop. Go ahead and copy-paste the code below. Then go through it afterwards in the next step.  
     
   private async void StartStopCommandExecute(object obj)

{

if (WorkOngoing)

{

// Make sure cancel hasn't already been requested

if (m\_Cancel is null || m\_Cancel.IsCancellationRequested)

{

Stop

return;

}

m\_Cancel.Cancel(); // Cancel the DoWork() Task

return;

}

WorkOngoing = true;

m\_Cancel = new CancellationTokenSource();

StatusMessage = "Work ongoing";

Start

StatusMessage = await m\_Worker.DoWork(m\_Cancel.Token);

m\_Cancel?.Dispose();

WorkOngoing = false;

}

1. Okay, so it looks a little complicated. Don’t worry, it is not that bad. When you click the button, StartStopCommandExecute() is executed. You can think of it as one handler for both the Start and Stop button (it is actually just one button which changes color and content). Hence, you need to know whether the button click was Start or Stop. WorkOngoing is being used for this. If WorkOngoing is true, then the Stop button was clicked. Let’s skip that for a second and pretend that WorkOngoing is false, meaning of course that the Start button was clicked. In this case, m\_Cancel is instantiated, and its token passed to m\_Worker.DoWork(). And then, execution is suspended until the Task completes. If Stop is clicked, the UI thread enters StartStopCommandExecute() a second time but now, WorkOngoing is true. This means that cancel will be requested on the CancellationTokenSource , which stops the Task in an orderly fashion and the first run-through of StartStopCommandExecute() resumes execution after the await statement. Notice that the CancellationTokenSource is disposed! That is important to remember, to avoid memory leaks.
2. That was a lot of reading, right? Sorry about that. Try to run the program and see what happens when you click Start and Stop. Try adding breakpoints in StartStopCommandExecute() to be able to observe the execution path.

## ConfigureAwait – How to avoid deadlocks

A risk that comes with async/await is the potential for deadlocks. To understand how that works, you are going to introduce a deadlock and afterwards solve the problem.

1. Let’s say that you don’t want to run DoWork() asynchronously in MainWindowViewModel. You want to run it synchronously and block the UI until the job is done. You should of course try to avoid this, but you could imagine a situation, where you need the result of a Task to actually show a meaningful UI. In this case, you need to wait synchronously for that Task to finish. In StartStopCommandExecute(), remove the async keyword, remove the await keyword and add .Result at the end of the DoWork() call:

private void StartStopCommandExecute(object obj)

{

if (WorkOngoing)

{

// Make sure cancel hasn't already been requested

if (m\_Cancel is null || m\_Cancel.IsCancellationRequested)

{

return;

}

m\_Cancel.Cancel(); // Cancel the DoWork() Task

return;

}

WorkOngoing = true;

m\_Cancel = new CancellationTokenSource();

StatusMessage = "Work ongoing";

StatusMessage = m\_Worker.DoWork(m\_Cancel.Token).Result;

m\_Cancel?.Dispose();

WorkOngoing = false;

}

1. Calling .Result on a Task will block the caller until the Task completes. Okay, that was the goal. Run the program, click the button, and see what happens.
2. Deadlock! The application enters a deadlock of which there is no escape. Let’s explore what’s going on. Per default, awaiting a task will capture the context of the caller and return to that context upon completion. However, continuation can only run once the context is freed up. In this case, the context is the UI thread, which is captured at the await statement in Worker.DoWork(). StartStopCommandExecute() waits for the task to complete while the Task, waits for the context to be free. This is what causes the deadlock.
3. The way to fix this is to use *ConfigureAwait* in Worker.DoWork(). ConfigureAwait has one argument named *continueOnCapturedContext*. It is default true which means, that after an await statement the execution returns to the captured context, in this case the UI thread. By setting this to false, you are indicating that you don’t want to go back to the captured context. Instead, whatever comes after the await statement is executed on the ThreadPool. Let’s see what that looks like:

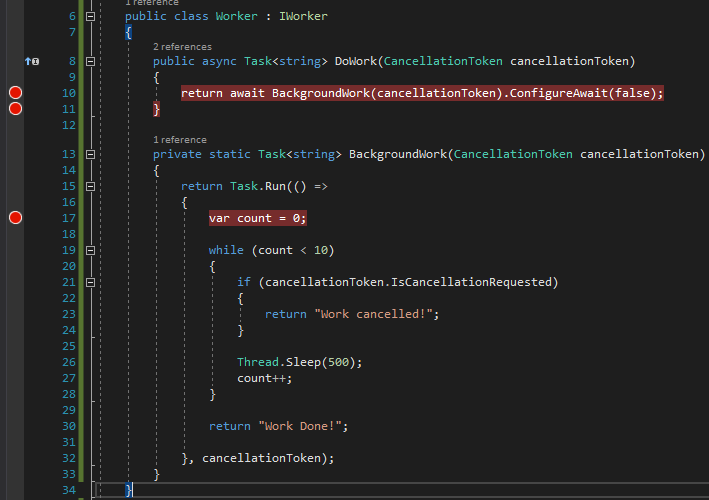
public async Task<string> DoWork(CancellationToken cancellationToken)

{

return await BackgroundWork(cancellationToken).ConfigureAwait(false);

}

1. Run the program and see what happens when you click the button.
2. It should now run synchronously and block the UI when you click Start, but the deadlock is avoided. This serves as a very important lesson. In your library code, use ConfigureAwait(false) as much as possible. You don’t know how the consumer of your library will interact with it. Whether they call your async methods synchronously or asynchronously. In your UI code, you should also try to use ConfigureAwait(false) whenever you don’t need the result of the Task to update the UI. If you always return to the captured context, i.e. the UI thread, after every await statement, it could hurt the performance of you program.
3. Roll back the synchronous behavior by reintroducing async/await in StartStopCommandExecute() and by removing .Result. Keep the ConfigureAwait() in DoWork(). Run the program and check that everything works.
4. Now, you’ll be using a handy tool in Visual Studio, to see the effects of ConfigureAwait(). Run the program, then, in Visual Studio, go to “Debug à Windows à Threads”. This window shows a list of threads used by the program. Add the following breakpoints:



1. Run the program and click the button. At each breakpoint, observe which thread is active (indicated by a small arrow in the Threads window).
2. In the above example, the breakpoint on the closing bracket of DoWork() executes on a ThreadPool thread. Now, try to change the argument of ConfigureAwait() to true in DoWork(), thereby forcing execution back to the captured context (UI/main thread). Try to run the program, click the button, and observe the active thread at each breakpoint.

## Error handling

Handling exceptions in asynchronous programming is somewhat different from synchronous programming. This exercise highlights some of the key aspects of that.

1. To avoid confusion, let’s add a second button with its own handler. In MainWindowViewModel, add a new ICommand property, initialize it from the constructer and define its handler:

public MainWindowViewModel(IWorker worker)

{

m\_Worker = worker;

StartStopCommand = new RelayCommand(StartStopCommandExecute);

CrashCommand = new RelayCommand(CrashCommandExecute);

}

public ICommand StartStopCommand { get; }

public ICommand CrashCommand { get; }

.

.

.

private void CrashCommandExecute(object obj)

{

}

1. In MainWindow (*MainWindow.xaml*), add a button with a command binding to CrashCommand:

.  
.  
.  
<Label

Grid.Column="0"

Grid.ColumnSpan="2"

Grid.Row="1"

Height="30"

HorizontalAlignment="Center"

VerticalAlignment="Stretch"

Content="{Binding StatusMessage}"/>

<Button

Grid.Row="1"

Grid.Column="0"

Content="Crash!"

Command="{Binding CrashCommand}"/>

</Grid>  
.  
.  
.

1. Run the program and check that the button is placed below the Start/Stop button and that clicking it fires the command handler.
2. Introduce a private async **void** method in MainWindowViewModel that awaits a dummy Task and then throws an exception. Call this method from CrashCommandExecute:

private void CrashCommandExecute(object obj)

{

JobThatThrows();

}

private async void JobThatThrows()

{  
 await Task.Run(() => { }); // dummy task

throw new Exception("Big explosion");

}

1. Run the program and click the crash button.
2. The program crashes, as you might expect. The natural thing to do, is to wrap JobThatThrows() in a try-catch. Go ahead and do that:

private void CrashCommandExecute(object obj)

{

try

{

JobThatThrows();

}

catch (Exception e)

{

Console.WriteLine(e);

}

}

1. Add a breakpoint within the catch statement, run the program and click the crash button.
2. Okay, so the program still crashes and the breakpoint in the catch statement wasn’t hit. What the hell is going on? JobThatThrows() is marked as async with the return type void. If you recall, you should **never** use *async void*, except for event handlers. This is another example of why that is. Since JobThatThrows() is marked as async, it throws the exceptions directly on the SynchronizationContext that was active when the *async void* method started. Which in this case, is the UI thread. If the method is *async Task* instead (or *async Task<TResult>*), the exception will be thrown on the Task object. That sounds promising! Change JobThatThrows() to async Task:

private async Task JobThatThrows()

{

await Task.Run(() => { }); // dummy task

throw new Exception("Big explosion");

}

1. Run the program again and click the crash button.
2. Now, nothing happens! Or so it seems. Check the Output window when you click the crash button. The exception is thrown on the Task object, which then dies in silence. Since the SynchronizationContext of the UI wasn’t captured beforehand, the exception is never propagated. This is a prime example of why you should always await methods that return a Task object. That way, the context is captured and whatever comes after the await statement is executed on the captured context.  
   Introduce async and await in CrashCommandExecute().

private async void CrashCommandExecute(object obj)

{

try

{

await JobThatThrows();

}

catch (Exception e)

{

Console.WriteLine(e);

}

}

1. Run the program once again and click the crash button.
2. Now, the exception is handled correctly. But what happens if the exception is thrown inside of the Task itself. Let’s explore!

private async Task JobThatThrows()

{

await Task.Run(() => throw new Exception("Big explosion"));

}

1. Run the program and click the button. You should still have a breakpoint within the catch.
2. It still works. The difference is that now the exception is thrown in the context of the ThreadPool thread inside the Task. It’s still propagated to the UI via the Task object returned by JobThatThrows(). To get a better understanding of this, store the Task object in a local variable in CrashCommandExecute():

private async void CrashCommandExecute(object obj)

{

Task t;

try

{

t = JobThatThrows();

await t;

}

catch (Exception e)

{

Console.WriteLine(e);

}

}

1. With the breakpoint in the catch still active, run the program and click the button. When the breakpoint is hit, mouse over the variable *t* and see its properties *Status* and *Exception*.
2. The *Exception* property of the Task object *t* is of type *AggregatedException*. This potentially contains a collection of exceptions if multiple exceptions were thrown on the Task object. Often you want to be able to distinguish between exception types. So, let’s see how that works. Change the exception type in JobThatThrows() to *IndexOutOfRangeException*, just so that it is easy to distinguish from the generic *Exception* type.

private async Task JobThatThrows()

{

await Task.Run(() => throw new IndexOutOfRangeException("Index out of range!"));

}

1. The claim made above is, that the exception type of the Task object is AggregatedException, so add a catch for that specific type in CrashCommandExecute():

private async void CrashCommandExecute(object obj)

{

Task t;

try

{

t = JobThatThrows();

await t;

}

catch (AggregateException ae)

{

Console.WriteLine(ae);

}

catch (Exception e)

{

Console.WriteLine(e);

}

}

1. Add a breakpoint in both catch statements. Run the program and click the crash button.
2. You might have expected to hit the breakpoint of the AggregatedException catch. But you didn’t. What happens is, that after the await statement, the exception is “unwrapped”, which gives you the IndexOutOfRangeException. If you have multiple exceptions on a Task object, this becomes a problem. Let’s explore that scenario. Make another async method that starts a Task which throws a NotSupportedException:  
     
   private async Task JobThatsNotSupported()

{

await Task.Run(() => throw new NotSupportedException("NotSupported"));

}

1. In CrashCommandExecute(), introduce 2 new task variables, *t2* and *all*. Then assign JobThatsNotSupported() to *t2* and combine *t* and *t2* into *all*. The method *WhenAll()* creates a new Task that finishes, once all of the Task object passed to it are finished. This is a common scenario when you want to start multiple asynchronous jobs at the same time but need all of them to finish before moving on.

private async void CrashCommandExecute(object obj)

{

Task t;

Task t2;

Task all;

try

{

t = JobThatThrows();

t2 = JobThatsNotSupported();

all = Task.WhenAll(t, t2);

await all;

}

catch (AggregateException e)

{

Console.WriteLine(e);

}

catch (Exception e)

{

Console.WriteLine(e);

}

}

1. So now a Task is awaited (*all*), which will contain 2 exceptions. Have a breakpoint in both catch statements, run the program, and click the button.
2. As you might have guessed, you only get the IndexOutOfRangeException in the generic catch, not the catch of the AggregatedException. When the breakpoint is hit in the catch, try to mouse over the *all* object and inspect its Exception property. Especially the InnerExceptions.
3. So, the information about both exceptions is there. But how do you retrieve it? You need to have a generic catch without the exception type specified. In this catch, you are able to inspect the AggregatedException of the *all* object.

private async void CrashCommandExecute(object obj)

{

Task t;

Task t2;

Task all = null;

try

{

t = JobThatThrows();

t2 = JobThatsNotSupported();

all = Task.WhenAll(t, t2);

await all;

}

catch

{

var allExceptions = all?.Exception?.InnerExceptions;

if (allExceptions != null)

{

foreach (var ex in allExceptions)

{

Console.WriteLine(ex);

}

}

}

}

1. Add a breakpoint in the for-loop of the catch. Run the program and click the button.
2. If you are only interested in specific exceptions and want to ignore other exceptions, you can use the *Handle()* method of the AggregatedException. Below, the catch has been re-written to only handle exceptions of type NotSupportedException.

catch

{

all?.Exception?.Handle(ex =>

{

if (ex is NotSupportedException)

{

Console.WriteLine(ex);

return true;

}

return false;

});

}

1. That’s it for error handling. There are other aspects to explore, but the ones you’ve been acquainted with in this exercise are the most important ones.

## Unit testing

This is the final part of the exercise. You should have a fairly good understanding of async/await by now, what it does and how to use it. Hence, this part of the exercise doesn’t contain any code snippets. It will tell you what to do, step by step, but how to do it is for you to find out.

This will only scratch the surface of async unit testing, since it will otherwise become an exercise on advanced mocking rather quickly (which is another exercise entirely).

1. It’s time to unit test the code. There isn’t much to test, but it will suffice. Add a .NET 4.7.2 Unit Test Project to the solution.
2. Add the nuget package *Moq* by Daniel Cazzulino to the project (right click the project and click “Manage Nuget Packages…”).
3. Rename the template unit test class to *WorkerTests*. You’ll be using this class to test Worker.DoWork().
4. In the test class, rename the test method, if there is one, or create a new one and call it DoWork\_RunToCompletion\_ReturnsWorkDone ()
5. Make sure the test method is *async Task*, not void
6. In your test method, create an instance of Worker. Call DoWork() and assert that the return value equals the string “Work Done!“.
7. Run the test. If it doesn’t pass, fix it.
8. Try to remove the await operator before running the test. Talk about what you expect to happen, before running the test.
9. Perhaps you’ve noticed that it takes a while to run the test. That’s no surprise, since the test awaits a Task that takes 5 seconds to complete. This can quickly become a problem, as you write more and more tests. So, whenever you have a sleep or a timeout of any kind, it’s a very good idea to be able to override it for unit testing purposes. There are a few ways to achieve this but the simplest one is to inject the timespan as a parameter in DoWork(). Go ahead and add that parameter to DoWork(). Pass it on to BackgroundWork() and use the parameter in Thread.Sleep() instead of the hardcoded 500ms. Remember to fix the call to DoWork() in MainWindowViewModel.StartStopCommandExecute() as well.
10. In your test, use a sleep interval of 1. Your test method now takes 10ms to execute rather than 5000ms.
11. It might be worthwhile to assert the status of the Task besides the return value. To do that, you’ll need to store the task in a local variable, so that you may assert on the *Status* property.
12. Write a test named DoWork\_IsCancelled\_ReturnsWorkCancelled. Assert that the return value is the string “Work cancelled!” when cancel is requested on the CancellationTokenSource.

## What’s next?

If you’ve made it this far, you are awesome! If you still have time to spare, consider reading some of the appendices or some of the links you skipped. Maybe you didn’t quite get the concept of SynchronizationContext and TaskScheduler. Try to store the SynchronizationContext and TaskScheduler a few places in the code to see what the value of them are during execution. This will give you an idea of how context switching works. You could also have a look at how async/await is used in the GPS source code. Or if you want to write an asynchronous application of your own, give that a go. You could, for instance, make an application that utilizes the asynchronous HTTP client of .NET, or read/write files asynchronously. The possibilities are endless!

# Appendix

## What is a Task?

A Task in C# is an object that wraps some work to be done and can be run asynchronously. A Task also provides information about the status of that work, e.g. is it running or finished, what was the result, etc. You typically use a Task to wrap work that is blocking, i.e. something that takes time and would block the UI until completed if it were called directly on the UI thread. This could be a HTTP call, downloading a file, doing a time-consuming calculation, communicating with hardware, etc.

It is important to distinguish between a *Thread* and a *Task*. A Thread represents an actual OS thread and gives you full control over it, i.e. you can Suspend() or Resume() a Thread, you can observe its state, you can control its stack size, apartment state etc.. You typically use a Thread for something that runs throughout the lifespan of the application and that you need full control over. The Core in GPS, for instance, runs in its own thread. In most cases however, a Task is the better choice.

A Task is an abstract concept. When you start a Task, it executes on what is called a ThreadPool thread. The runtime handles and manages a pool of threads and schedules the Task on one of those threads. Unlike a Thread created manually, you have no control over the ThreadPool managed by the runtime. Since Tasks run on the ThreadPool, they should not be used for long-running operations, as they can fill up the ThreadPool and block new work. Instead, Task provides a LongRunning option, which will tell the TaskScheduler to spin up a new Thread rather than running on the ThreadPool.

## Context, SynchronizationContext and TaskScheduler

The 3 concepts mentioned in the title are crucial to asynchronous programming. Let’s kick things off by having a look at *Context*. Every thread has an associated context, or a “current” context if you will. The context is the boundaries that the thread can operate within in terms of object access and possible execution paths. Think of the context as the threads’ view of the world and the state of that thread.

The *SynchronizationContext* provides a way to queue a unit of work to a context. Note that this unit of work is queued to a context rather than a specific thread. This allows a context to offload work to a different context. Another aspect of SynchronizationContext is that every thread has a “current” context. A thread’s context isn’t necessarily unique; its context instance may be shared with other threads. It’s possible for a thread to change its current context, but this is quite rare. The actual “context” of the SynchronizationContext varies from framework to framework. WPF applications use a *DispatcherSynchronizationContext*, which queues delegates to the UI thread’s Dispatcher. The context for DispatcherSynchronizationContext is a single UI thread. For Tasks running on a ThreadPool thread, the SynchronizationContext is the so-called *default* SynchronizationContext. By convention, if a thread’s current SynchronizationContext is null, then it implicitly has a default SynchronizationContext. The context of the default SynchronizationContext covers all ThreadPool threads. Thus, UI applications usually have two synchronization contexts: the UI SynchronizationContext covering the UI thread, and the default SynchronizationContext covering the ThreadPool threads.

Finally, there’s the *TaskScheduler*. Simply put, the TaskScheduler is responsible for scheduling the execution of work on a given SynchronizationContext. Hence, for WPF, the there is a TaskScheduler for the UI SynchronizationContext (DispatcherSynchronizationContext) and a TaskScheduler for scheduling work on the ThreadPool (default SynchronizationContext). The TaskScheduler handling the ThreadPool is the *default* TaskScheduler by convention.

The description above is simplified and skips a lot of details. If you want to know more, read the excellent articles linked below:

<https://docs.microsoft.com/en-us/archive/msdn-magazine/2011/february/msdn-magazine-parallel-computing-it-s-all-about-the-synchronizationcontext>

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskscheduler?view=net-5.0>

## Updating the UI in a Task

Let’s explore what happens, if you update a UI element from within a Task, i.e. not on the UI thread.

1. Go to MainWindow (*MainWindow.xaml*) and add the following line just below the Label definition (inside the Grid):

<Button Grid.Column="0" Grid.Row="1" x:Name="CrashButton" Content="Crash!" Click="CrashButton\_OnClick"/>

1. In the code-behind (*MainWindow.xaml.cs*), add the following button click handler:  
     
   private void CrashButton\_OnClick(object sender, RoutedEventArgs e)

{

Task.Run(() => CrashButton.Content = "Crash and burn");

}

1. Before running the program, let’s have a quick look at the code. A button has been added to the view (in xaml). It has the name “CrashButton” and a click event handler called CrashButton\_OnClick(). The handler is defined in code-behind. When called, the handler creates and starts a Task, which updates the button content. Note that Task.Run() is just shorthand for creating and starting a Task in one go. Now, run the program, click the button, and watch the explosion from a safe distance.
2. Remember to remove the code added above, before continuing the exercise!