Michał Zegan

Asynchronous Programming Guide for .NET

Thanks!

A big shout-out to the great squad that made this guide happen! Special thanks to Bartosz Fijałkowski from the .NET job family[1] and Edyta Krukowska from the EB and Communications team. Without you, this project might have been stuck in the „Idea Limbo," never making it to the realm of a published eBook.

And let's extend major kudos to Bartosz Borkowski, and the other fantastic folks from the .NET Job Family. Your insights and feedback were invaluable, and your positive energy kept the project atmosphere as pleasant as a leisurely stroll in the park.

A huge thank you to the amazing and creative Małgorzata Barska. In one move, you changed the shape of this e-book and gave it a bright and professional look.

Last but not least, a major kudos for the C# Discord community! Without you and your vital discussions, I might still be trying to figure out where to start with asynchronous programming. :)

1 Job families are our internal units gathering people around specific technology. They help our members with personal growth, support certifications, conferences, knowledge sharing, workshops or even non-tech integration meetings. Job families ensure that every single person can develop his abilities within selected career path and direction he/she wants to follow. Wider description can be found at the end of this document.

GFT

Contents

1. Introduction

This guide covers asynchronous programming in .NET, focusing on the `async/await` construct of the C# programming language. It starts by describing basic concepts and goes on to show how async programming is to be employed in .NET world, including best practices and caveats. It also describes some details needed to understand and properly use the async mechanism.

It is expected the reader is familiar with the C# language, including object oriented programming and exception handling.

2. Processes and threads

Let's define some concepts related to multithreading, which will be useful later in the guide. People familiar with multithreading concepts in .NET can safely skip this section.

Processes

A single running program is called a *process*. One program might run multiple times at once, creating multiple processes. Processes are isolated from each other, mainly by having their own address space (meaning their own view of memory), their own table of open files and other resources.

Threads

A *thread* is an operating system resource contained in processes. Process starts with a single thread called the main thread, which is responsible for initially running the program, and which might create any number of additional threads later if needed. It's threads, not processes, which actually execute program code. One thread executes only a single piece of code at any given time sequentially. It is possible and pretty common that multiple threads simultaneously execute the same piece of code. Multiple threads are needed if the intention is to execute some parts of the program independently from the main code. For example, if a GUI application needs to fetch network data and decode video frames in addition to displaying the UI, it could use one thread for each of these three operations, so that they can execute independently. Threads of a process share it's resources including view of memory, so access to common variables by multiple threads requires extra care and usage of thread synchronization techniques. Each thread has it’s own stack - storing local variables and function parameter values.

Task Scheduler

On a single processor system, only one process/thread can execute at any given time. For that reason, there exists an operating system component called *scheduler*, which divides processor time and allocates time slices to each thread (of each process) in turn. Each thread has a chance to execute for it’s allocated time slice, then when it’s time slot is up, thread is forcefully paused (preempted) and it’s state, like processor registers, is saved , then another thread takes it’s place and gets resumed at the point it was interrupted (this process is called a context switch). After there are no more threads left, the whole process repeats again indefinitely. That gives the illusion of multiple threads/processes executing at the same time from the human perspective. In addition, executing code itself is completely unaware of context switches happening and is written as if it was never being interrupted. We can say threads/processes execute concurrently. On a multiprocessor system, there are multiple processors (each physical core or thread on a hyper threading system counts as one processor) and so there might be as many threads executing at exactly the same time as there are processors. Such threads execute in parallel. Because there are usually more operating system threads than processors, a scheduler is still in use. Programs have limited control over the operating system scheduler, like being able to force an immediate context switch, or to set a thread’s priority.

Thread Pool

Threads are pretty heavy resources to create and manage, and put pressure on the operating system. However, there might be a need to execute large number of possibly independent tasks concurrently or in parallel. Instead of creating a dedicated thread for each task separately, especially for short running tasks, it would be useful to have a way to reuse existing threads to better manage resources. A solution to the problem in .NET is called the *Thread Pool*.

The thread pool creates and manages a small set of threads (called worker threads). There exists the work queue user can send work items to, and worker threads constantly monitor the queue, repeatedly picking up and executing these items, which are often completely unrelated to each other. If there are less work items to execute than worker threads, threads without work to do remain idle, so that they can be reused for work submitted later, but if there are more work items than threads, incoming work remains in the queue until some thread completes the currently running operation and becomes free to pick something else. Based on settings and current throughput, thread pool manages (creates or destroys) worker threads as needed to accommodate the load. There is one thread pool per process in .NET, and it is optimized for executing relatively small, short running tasks.

This guide does not go into details about threads and concurrent/parallel programming. Asynchronous programming concept itself does not relate to threads, however in .NET threading support (mostly the thread pool) is being used under the hood to increase throughput.

3. Sync vs async

Let's describe what asynchronous operations actually are, mostly by comparing them to synchronous operations, which are also described. Note for people not familiar with newer C# features, whenever possible, C# 9 top level statements will be used for brevity.

The following code is an example of a synchronous operation:

```
using System;

DoWork();
Console.WriteLine(„Synchronous operation completed”);

static void DoWork()
{
    // do something interesting
}
```

This code does nothing, but you can imagine that the `DoWork` method is actually computing some values, performing IO or even entering an infinite loop. When top level code calls `DoWork`, it actually makes the processor switch from main code to the method. Top level code can continue only after the method actually returns back to it. In an abstract sense, we can say that top level code, by calling the `DoWork` method, actually begins a synchronous operation, which means it has to wait for `DoWork` to fully complete before printing the completion message.

On the other hand, asynchronous operations are operations which don't have to fully execute before the code calling them continues. In fact, from the perspective of the caller, starting such an operation returns to the calling code almost immediately, and the operation itself is running in the background independently. That allows code which originally started the operation to perform other work without waiting for it's completion.

Async operations usually include some way to notify the caller about their completion. Often it's some callback mechanism making the operation call some user defined function after it completes or encounters an error, allowing the program to react. Such a function is often passed as an argument when starting the operation.

3.1. Sync vs async for computational tasks

Operations performed by programs might be more CPU or I/O bound. Because CPU bound operations always occupy the processor, they are synchronous in nature. The only way you can make such an operation execute asynchronously is to run it on a different thread than the one starting the operation, for example on the .NET's thread pool, so that the calling thread can continue it's work without waiting for the computation to fully complete first. This, however, means another thread is tied up to execute the task. It is usually not beneficial to run computations asynchronously, as they are relatively fast themselves and executing them synchronously often yields better performance.

3.2. Sync vs async I/O

Sometimes there is a need to run CPU bound operations asynchronously for performance reasons, however where asynchronous programming model shows it's true potential is I/O bound operations, and this is the area where it's mostly being used. In .NET, most I/O related methods have both synchronous and asynchronous flavors, and in some cases (especially networking) they are exclusively async.

Characteristics of I/O bound tasks is a bit different than of CPU bound tasks, because in this case cpu is not involved besides requesting the operation and processing it's result, most work is done by devices other than CPU, like storage devices or network hardware, and operations tent to be orders of magnitude slower, especially when waiting for network data.

Below code is an example of synchronous IO operation, namely reading a file.

```
using System;
using System.IO;

// This program synchronously reads a text file and prints it.
// It purposefully doesn’t use high level I/O operations
// like File.ReadAllText or similar.

using var f = File.OpenText(„test.txt”);

var buffer = new char[4096];
while (true)
{
    int charsRead = f.Read(buffer); // 1

    if (charsRead == 0)
    {
        break;
    }

    Console.Write(new string(buffer, 0, charsRead));
}

Console.WriteLine();
```

Let’s ignore most of the program and focus on the marked line. This line reads some text from file to the buffer passed in as parameter in a synchronous manner. The operation might complete quickly if data is already present, for example in internal buffer of the stream, or cached by filesystem. However, if data is not present, an actual I/O operation to the storage device is performed and invoking the `Read()` method blocks, waiting for I/O to complete.

Sync / Blocking I/O

Blocking in this context means that a thread is not being scheduled during the I/O operation and is put on hold until it completes. Blocking times might not necessarily be so high for storage, especially SSD storage, but they might be really long in case of networking operations. As an example, in case of network issues or other side of a tcp connection not sending data at all, the data receiver would block possibly indefinitely.

During blocking, especially long blocking, a thread is literally wasted. To perform other operations independently when blocking is involved, you would need to create a thread per operation. Because threads are expensive resources, this is not an ideal solution, especially if large number of in flight operations are expected. Ideally there would be a way to support thousands of clients/connections/requests with a small number of concurrently running threads, which requires an alternative to blocking. Of course servers/web applications are not the only kind of programs affected, and networking is not the only area where throughput/resource utilization issues might be important.

Async I/O

A response to the problem is asynchronous I/O. Contrary to CPU-bound operations, I/O is asynchronous in nature. When performing I/O operations, CPU might need to do actual work to send the request to some kind of device, however from that point it's not involved anymore and can continue other possibly unrelated work. When the device completes the operation, it raises a so called interrupt, which makes the processor stop what it's doing and switch to special code called an interrupt handler (managed by the operating system kernel/device driver), which reacts to it, for example marking the operation as completed and causing related work to be queued, then resumes the interrupted code. Applications can take advantage of this model by employing asynchronous I/O support provided by the operating system.

An important thing to remember looking at the above description is that in case of asynchronous I/O, there is no thread dedicated to perform the operation after it's started. Neither thread which started it, nor any other thread is directly responsible for the operation until it completes.

4. Task-based asynchronous pattern in .NET

.NET has always supported asynchronous programming, but the way it was implemented evolved over the years. The newest async programming pattern is called *Task-based asynchronous Pattern*, shortly named *TAP*. It relies on the *Task Parallel Library*, which is a library used to write asynchronous and parallel applications.

Asynchronous methods in .NET can be recognized by appearance of the *Async* suffix in their names, for example `ReadAsync`, and by the fact they return a task object representing the async operation like `System.Threading.Tasks.Task` or `System.Threading.Tasks.Task<T>`, or recently introduced `System.Threading.Tasks.ValueTask` or `System.Threading.Tasks.ValueTask<T>`. You can use the returned task to monitor the operation, including to register functions which are called on operation completion or failure. Such notification functions are called continuations.

4.1. Async/Await feature in C#

Asynchronous programming by way of manually manipulating tasks and registering continuations is difficult and error prone, so C# language introduced a feature which can be used to easily write asynchronous methods. This feature is called *async/await*. When using this feature, you can write async methods almost in the same way synchronous methods would be written, and compiler and runtime libraries do most of the hard work.

Below is an example of async code analogous to the synchronous I/O example above. For demonstration purposes, most of the program is wrapped in a separate method.

```
using System;
using System.IO;
using System.Threading.Tasks;

var chars = await PrintFileAsync(„test.txt”);
Console.WriteLine($”Number of characters in file: {chars}”);

static async Task<int> PrintFileAsync(string name) // 1
{
  // This program asynchronously reads a text file and prints it, then
  // returns number of characters in that file.
  // It purposefully doesn’t use high level I/O operations
  // like File.ReadAllText or similar.

  using var f = new StreamReader(new FileStream(name,
    FileMode.Open, FileAccess.Read,
    FileShare.Read, 4096, true)); // 2

  var buffer = new char[4096];
  int totalChars = 0;
  while (true)
  {
    int charsRead = await f.ReadAsync(buffer); // 3

    if (charsRead == 0)
    {
      break;
    }

    Console.Write(new string(buffer, 0, charsRead)); // 4
    totalChars += charsRead;
  }

  Console.WriteLine(); // 4

  return totalChars; // 5
}
```

Here are the important points related to the marked lines in PrintFileAsync method:

1. The asynchronous method declaration is identified by the `async` modifier. The method is meant to return the number of read characters, but it is declared to return `Task<int>` instead of int. Declarations and their meaning are further described in section 4.1.1 below.

2. The line opening a file for reading is more complex than the synchronous example, as it directly uses respective `FileStream` and `StreamReader` constructors. However it's necessary, because for asynchronous I/O on files to work correctly, we have to first open the file in async mode, which requires setting the `useAsync` parameter of `FileStream` constructor to `true`.

3. The `ReadAsync` call does the actual work of reading data from file. Because it starts an asynchronous operation, it returns a `Task<int>` instance immediately, which needs to be monitored for completion while the operation is in flight. Notice the appearance of a new `await` keyword directly before the call. The `await` keyword makes the method wait for the operation to fully complete before continuing, and then returns the operation result, being an `int` value in this case (but could be an I/O exception being thrown as well). `await` does not cause blocking and instead suspends the method until operation completes, further explanation below.

4. It's important to remember that console I/O in .NET is synchronous only, so things like repeatedly reading commands from console really do require a fully dedicated thread, which is often the main thread. You should not use async methods on console streams.

5. Even though the method is declared to return `Task<int>`, it's written as if it returned int directly. The return value automatically becomes the result of method's task.

As you can see, the top level code can also use the `await` operator, without needing to be specially marked.

The code above not only shows asynchronous methods being used, it also shows asynchronous methods being declared. Asynchronicity, when implemented right, is viral. In most cases a method starting an async operation can't itself meaningfully continue until it completes, but you don't want to lose the advantages of async by blocking. The goal is to free the thread so that it can be reused instead. Any method calling async methods should usually become async itself, and the same extends to it's callers, and callers of the callers, etc, for the same reason.

Starting async operations and waiting for their completion are separate concerns, even if usually they are combined. You can write the `await` separately from operation start, including in a different method, if you still hold the returned task object. It gives ability to do things such as being able to start multiple operations at once, then to await all of them, so they run concurrently and don't depend on each other, or to start a long running operation and store it's task for later monitoring.

4.1.1. Asynchronous method declarations

Let's start by showing how asynchronous methods can be declared when using the async/await feature. Using this feature is an implementation detail of the method and is not visible outside, so when consuming a library, async methods can only be identified by their signature.

You can declare an async method in C# by using the `async` modifier. It can take any parameters except parameters passed by reference (using `in`, `ref` and `out` keywords) and things like `Span<T>`. However, async method can return only the following:

- `System.Threading.Tasks.Task,`
- `System.Threading.Tasks.Task<T>,`
- `void.`

Async methods can actually return other types like `ValueTask` and user defined types made to behave like tasks, but this is an advanced use case.

An asynchronous method cannot return arbitrary types like `int` or `bool`. It has to return a type representing the in-flight operation, like a `Task`. Generally, methods which would normally return `void` become `Task` returning methods, and methods which would normally return some `T` type become `Task<T>` returning.

The below table contains some example synchronous method declarations and their async counterparts, focusing on the return types.

Sync declaration	**Async declaration**
`void DoWork()`	`async Task DoWorkAsync()`
`int DoWork()`	`async Task<int> DoWorkAsync()`
`IEnumerable<SomeFancyType> DoWork()`	`async Task<IEnumerable<SomeFancyType>> DoWorkAsync()`

All the above methods return `Task` or `Task<T>` in async variant. What about the async methods returning void? Such methods shouldn't generally be written except in case of creating asynchronous event handlers. This case will further be described in section 4.1.4 below.

It is also possible to create asynchronous lambdas, which is useful when passing task returning delegates. For example:

```
Func<Task> f = async () => await ...;
```

The `async` modifier is still required and comes before lambda parameter list.

4.1.2. About the awaits

Now, it's time to describe more precisely how async methods are to be written, and the `await` keyword operation.

Generally, as already seen in the example above, async methods look almost like their ordinary counterparts except the presence of `await` operator and the fact they return tasks as results. You can use most constructs of the C# language with some exceptions, but they are not that important at the moment. If an async method returns a value or throws an exception which is not caught inside of the method, it automatically becomes it's task completion result, which can be retrieved, for example, by awaiting that task somewhere.

Await operator

The most important feature specific to async methods is the `await` operator. In fact, the async modifier just makes the `await` work, and if a method is marked async and does not have any `await` in it's body, compiler issues a warning *CS1998*. Under the hood, the compiler rewrites async methods in a way that makes them suspendable. The `await` operator suspends the method until the awaited operation completes, but without blocking (explanation below). The method resumes execution when the task completes. The compiler arranges task resumption by registering continuations on task object, but the actual mechanics is normally hidden from you.

So let's take the above asynchronous file reading example again and describe how it executes in presence of awaits, focusing only on the `PrintFileAsync()` method. Here is how the execution proceeds, step by step:

1. When `PrintFileAsync()` is called, it first executes synchronously like ordinary methods. The text file is opened and `ReadAsync()` is first called, returning the `Task<int>` representing the ongoing operation.

2. `await` is applied on the returned `task` instance. Because the `task` is not completed, the `await` suspends the method, arranging it to be resumed later after operation completion. Suspending the method means it actually returns at this point, even though it didn't yet complete. This is also the moment when the caller (top level code in this case) receives a `Task<int>` from the method. From this point on the method is asynchronous.

3. When read completes, the method is scheduled to resume on some, possibly different thread. In most cases except GUI applications, it will be a thread pool thread.

4. The `await` returns the number of characters in the buffer, or if the operation threw an exception, this is the moment when it appears.

5. If there was no exception, the loop continues normally until end of file is reached. Each time, when performing an await, the method is suspended again freeing the thread for reuse, then is being resumed after read completes, but not necessarily on the same thread.

6. After loop ends, the method completes with the total number of characters in the file. The method's return value becomes it's task completion result and can be picked up by top level code.

There are few important points to consider when reading the above description:

1. An asynchronous method marked with `async` should itself be composed of calls to other asynchronous methods, which likely themselves call other async methods… The reverse of the "async is viral" point above. The exception would be if awaiting previously stored tasks, but this is more rare.

2. An asynchronous method is not necessarily 100% asynchronous, it runs synchronously when called until reaching first await. It's actually not quite accurate, because if an operation completed before reaching it's await, the method will immediately continue without being unnecessarily suspended first. In addition, after the method is resumed later (for example on a thread pool thread), it executes code on that thread continuously until next await or return/throw, whichever comes first, which occupies the worker thread for the duration of that fragment of code. It means it's important to make the first, synchronous part of the method as short as possible, so that the operation turns asynchronous quickly. Code executing between awaits also shouldn't occupy it's thread for too long, unless necessary.

3. An asynchronous method, depending on the case, might actually complete synchronously. The method might return or throw without ever awaiting anything first. This still means that the method records the result in a task same as in asynchronous completion case, so there is no difference in handling it by the caller. This is useful, for example, if some normally asynchronously retrieved result has been cached, or there was an error starting the operation which is immediately visible and can be thrown.

The above two points need one more illustration:

```
using System.Threading.Tasks;

await DoWorkAsync(true);
await DoWorkAsync(false);

static async Task<bool> DoWorkAsync(bool something)
{
  if (something)
  {
    return true;
  }

  await Task.Delay(500);
  return false;
}
```

The `DoWorkAsync()` method does the following:

- If given `true` value as parameter, it synchronously returns `true`, the method never needs to suspend and caller immediately sees the method completing.

- If `false` is given as a parameter, the method executes asynchronously, in this case completing after roughly 500ms.

The top level code uses `await` here in both cases and it's not needed to account for the synchronous completion case. This is an implementation detail of the method which might change without notice and without breaking the application.

4.1.3. Exception handling in asynchronous methods

Asynchronous methods, like synchronous ones, can throw and catch exceptions. For the most part, exceptions work exactly the way one would expect. However, what happens if an exception is not caught in the async method?

In synchronous code, unhandled exception is propagated back to the method's caller in search for exception handlers, and to caller of the caller if not found there, etc…, and if there are no more methods left in the thread, the application terminates. This is called exception propagation. However, asynchronous methods execute in a different way. Different parts of them might execute at different times on different threads independently of the real caller. But the caller is still usually interested in the fact the async operation has failed. In fact, an exception is another type of method's result.

When an asynchronous method throws an exception which is not caught in method's code, exception is ultimately being set as the method's task result. That makes the task be marked as faulted with that exception object. If the caller (or possibly some other method) awaits it, then the exception is immediately propagated by rethrowing it in the awaiting method as soon as it resumes. Of course, because a method using `await` is itself async, if it doesn't handle the exception, everything repeats again until it is caught. `await` is not the only way of handling tasks, and this description might not literally apply to other cases, but the principle is always the same. For example, if an async method is called by synchronous code for any reason (remember sync code cannot use `await` and will most likely wait in a blocking way), exception will be rethrown and further propagated in a standard synchronous way starting from the point of waiting.

A method marked `async` never throws any uncaught exceptions directly, even exceptions thrown when still executing synchronously. All exceptions are wrapped inside of a task object.

What happens if an exception is thrown in async method, but not handled at all? There are two ways an exception coming out of async method can become unhandled.

Case 1: Handling properly awaited tasks

First case is similar to one of synchronous exceptions, where an exception, after being thrown, propagates through method's callers until it reaches the bottom of thread's call stack. In case of asynchronous methods, propagation goes through the chain of awaits (as described above), then if an exception is never handled, at some point it reaches the first asynchronous method in the call chain. That method is often the program's `Main` method/top level code. If it also doesn't handle the exception, the application terminates.

Let's take an example:

```
using System;
using System.Threading.Tasks;

await Method1();

static async Task Method1()
{
  await Method2();
}

static async Task Method2()
{
  await Method3();
}

static async Task Method3()
{
  await Task.Delay(10); // just to introduce an await...
  Method4();
}

static void Method4()
{
  throw new NotImplementedException();
}
```

Note this example includes synchronous methods in the mix. It's not recommended to call async code from sync code if not necessary, but doing the reverse is normal and impossible to avoid.

The `Method4` throws `NotImplementedException`. Here's how it propagates:

1. Because `Method4` does not handle the exception, it propagates to `Method3` in a standard synchronous way. Because it's also not handled there, it becomes the result of `Method3`'s task.
2. Because `Method3` has completed, `Method2` which awaits it gets resumed. As task completed with exception, that exception is rethrown in `Method2`.
3. Because `Method2` does not handle the exception, it becomes the result of method's task, which in turn triggers `Method1`.
4. Exception propagation continues through `Method1` and top level code, because both do not catch the exception.
5. Because the exception has propagated outside of top level code, application terminates and the exception is displayed to the user.

Frameworks, like *WPF* or *ASP.NET Core* usually run below normal user code, and they are responsible for handling exceptions thrown in code they invoke before they reach the runtime, but the specifics are framework dependent and are not described here.

Case 2: Handling NOT properly awaited tasks

The other case of unhandled exceptions is when an asynchronous call is not properly awaited or otherwise handled. This might happen when someone literally forgets to await or when someone starts a fire and forget task and thinks awaiting it is not necessary. Here is a small example of forgetting to await:

```
using System;
using System.Threading.Tasks;

DoWorkAsync(); // should await here

await Task.Delay(-1); //App never dies.

static async Task DoWorkAsync()
{
  await Task.Delay(5);
  throw new NotImplementedException();
}
```

The first, obvious problem is that the `DoWorkAsync()` method is called but not awaited, so top level code will continue without waiting for it's completion. An additional infinite delay is introduced after the method call so that the application never terminates and `DoWorkAsync()` has a chance to complete.

The second problem with this code is that, even though `DoWorkAsync()` throws an exception, it is not being handled. It's just stored in the method's resulting task, but the task is thrown away instead of being awaited and exception gets completely ignored. The task becomes eligible for garbage collection.

The `UnobservedTaskException` event

.NET has a mechanism to allow handling errors even in presence of such bugs.

If any task completes with an exception and is not handled at all, then when it's being garbage collected, Common Language Runtime raises the `System.Threading.Tasks.TaskScheduler.UnobservedTaskException` event. You may register an event handler to be notified about unseen task exceptions, and might react appropriately. However remember about the following:

- If the event handles some exception, it should mark it as observed, otherwise the exception, depending on .NET version and configuration, might cause application termination. Specifically, in .NET core and never, it will be ignored.
- This mechanism is to be used only as a last resort. This event almost always indicates an application bug which needs to be fixed by properly awaiting or even by catching exceptions before they reach this stage, even just to log them.
- This mechanism is not triggered in many other cases, like task which is stored and not awaited. It also doesn't get triggered if something actually does read the task's stored exception and then ignores it further, as it will then count as exception being handled.

This is the same example as above, but with `UnobservedTaskException` event handling implemented. Raising it requires a garbage collection to happen, so it's being triggered manually for demonstration purposes.

```
using System;
using System.Threading.Tasks;

// Event handler for unobserved tasks.
TaskScheduler.UnobservedTaskException += HandleSkippedError;

DoWorkAsync(); // should await here

await Task.Delay(500);
GC.Collect();
await Task.Delay(-1); //App never dies.

static async Task DoWorkAsync()
{
  await Task.Delay(5);
  throw new NotImplementedException();
}

static void HandleSkippedError(object sender,
UnobservedTaskExceptionEventArgs e)
{
  Console.WriteLine(e.Exception);
  e.SetObserved();
}
```

The above program will display an `AggregateException` containing the `NotImplementedException` thrown by `DoWorkAsync()`, even though we forgot to await.

Because of how Task Parallel Library works, the `UnobservedTaskException` event doesn't receive the exception directly, instead it gets an `AggregateException` instance containing it. This is mainly to support advanced TPL features related to parallel programming like attached child tasks, which aren't really relevant here.

4.1.4. `async void` methods

Most asynchronous methods are declared to return some task type, like `Task` or `Task<T>`. However, it is possible to also declare a method as `async void`:

```
public async void DoWorkAsync()
{
    // do something
}
```

Contrary to methods returning a task, `async void` methods don't return any object which could be used to track their completion. That means they are inherently fire and forget. Caller is not able to monitor their execution in any way, including awaiting them or registering continuations. Although fire and forget methods are sometimes useful and have their place, returning a `Task` (even in such cases) gives us more flexibility and control in the way method's completion is handled. As an example, even though caller does not care about method's result, it might still care about exceptions it throws and might try to log them, even if the called method itself didn't. Also, `async void` methods can accidentally be used in contexts where no asynchronous methods are expected to be called without introducing any compiler or runtime errors, because they look exactly like non asynchronous `void` returning methods (remember `async` modifier is method's implementation detail and is not visible from other projects). This is a source of pretty hard to detect bugs. As an example, you could pass such a method by way of a delegate to anything expecting a delegate of type `Action` or similar, and the calling code in question may not work correctly when the passed method is in fact asynchronous. For these reasons, `async Task` methods should be preferred over `async void` methods. ` in almost all cases.

There is one more reason why `async void` methods should be avoided, related to exception handling. Because there is no object representing an async operation, exceptions thrown from such methods can't be stored in a task. Instead, they are rethrown in the current synchronization context – the concept of a synchronization context will be explained in section 4.2 below. In practice it means that in case of a GUI application, the exception is rethrown in the GUI framework code and causes framework defined action related to unhandled exceptions. This means user code including the method's caller is unable to process it at all, even if that's desired. If there is no synchronization context (which usually means this is not a GUI application), the exception is rethrown on the thread pool and the application unconditionally terminates. Note mechanisms like the `UnobservedTaskException` event do not work in this case and cannot be used to log exceptions from failed `async void` operations.

The above description raises the question about such methods being useful at all. They should be used in one and only one case, which is asynchronous event handlers, mostly in GUI applications. Events are inherently fire and forget, they are usually raised by some framework code directly. Each event might have multiple event handlers registered. However, due to how events work, it's not possible to create and reliably use events returning anything else than void. For that reason, if you need to handle an event and call asynchronous operations in the event handler, then you have to declare the event handler method as async void.

```
private async void OnEvent(object? sender, EventArgs e)
{
  // do stuff
  await Task.Delay(500);
}
```

4.1.5. Asynchronous `Main` method

As a recap, when not using the top level statements feature and writing the `Main` method directly, it can normally take one of the following forms:

- `static void Main()`
- `static void Main(string[] args)`
- `static int Main()`
- `static int Main(string[] args)`

All these forms of `Main` are synchronous methods. As with any other methods, `Main` can also be made asynchronous, and each kind of `Main` shown above has it's asynchronous counterpart. These are:

- `async Task Main()`
- `async Task<int> Main()`
- `async Task Main(string[] args)`
- `async Task<int> Main(string[] args)`

They are analogous to the synchronous forms, except that they return a `Task` or `Task<int>` and you can use `async/await` feature in code. Here is an example:

```
using System;
using System.Threading.Tasks;
```

```
class Program
{
  static async Task Main()
  {
    await Task.Delay(500);
    Console.WriteLine(„Program completed successfully”);
  }
}
```

The compiler correctly handles the `Main`'s returned task. The program does not terminate until the method fully completes. In addition, exceptions propagated out of the `Main` method correctly terminate the application, same as in case of synchronous `Main`.

Remember that top level code is always turned into an invisible `Main` method. If it never awaits, it's turned into a synchronous `Main`, and if it has at least one `await` line, it's turned into it's async counterpart.

4.1.6. Asynchronous iterators

To write enumerators easily, you can use iterator methods.

Iterator methods return objects implementing `IEnumerable<T>` and `IEnumerator<T>` interfaces, but these interfaces are inherently synchronous. What happens if there was a need to access a slower data source like a database, which would often be accessed asynchronously? Let's see an example of an iterator method which returns numbers from 0 to 100. To simulate a slow data source, the method sleeps 100 milliseconds between returning each number.

```
static IEnumerable<int> Items()
{
  for (int i = 0; i <= 100; i++)
  {
    Thread.Sleep(100);
    yield return i;
  }
}
```

And here is a simple `foreach` example to use it.

```
foreach (var i in Items())
{
  Console.WriteLine(i);
}
```

The `foreach` repeatedly calls the enumerator's `MoveNext()` method, which generates further items. However, each call blocks for 100 milliseconds. Blocking is not really desired in async heavy code.

Previously, one workaround could be to first buffer the items in an asynchronous way into a collection then iterate through them synchronously, but it's not always desirable, especially if data comes in streaming fashion and there is lots of data to process. Enumerable data sources can potentially be of infinite size. For that reason C# 8 introduced asynchronous iterator methods. They are asynchronous methods returning `IAsyncEnumerable<T>` or `IAsyncEnumerator<T>`, which are interfaces used for asynchronous enumeration. In async iterator methods, you can use both `await` and `yield` keywords. Here is how the above example could be rewritten using this feature to eliminate blocking.

```
static async IAsyncEnumerable<int> Items()
{
  for (int i = 0; i <= 100; i++)
  {
    await Task.Delay(100);
    yield return i;
  }
}
```

However, to use `IAsyncEnumerable<T>` without writing code by hand, a special form of `foreach` loop is needed:

```
await foreach (var i in Items())
{
  Console.WriteLine(i);
}
```

This is equivalent to the above, except the `foreach` is preceded by an `await` keyword. This makes the loop itself asynchronous. Internally, instead of calling `MoveNext()` method of `IEnumerator<T>`, it calls and awaits `MoveNextAsync()` method of `IAsyncEnumerator<T>`. That makes it possible for the iterator methods, or other implementations of `IAsyncEnumerator<T>` interface to employ asynchronous operations instead of having to block when operating on slow resources.

4.1.7. Asynchronously disposable objects

All objects which directly or indirectly hold unmanaged resources, or otherwise need to contain some cleanup logic, implement `IDisposable` interface and it's `Dispose()` method. However, in some cases, disposing an object, like closing database connections, might require performing I/O. In order not to have to block on `Dispose()` calls, C# 8 introduced an `IAsyncDisposable` interface and it's `DisposeAsync()` method, which can be implemented by objects whose disposal requires performing I/O operations which might benefit from being executed asynchronously. As an example, closing database result sets or connections might require communication with database to perform an orderly resource release, and writing a file might require flushing stream buffers to disk before closing the stream.

Below is a short example of writing a file asynchronously, which shows asynchronous disposal. Note the `Stream` and `TextWriter` classes implement `IAsyncDisposable`.

```
using System;
using System.IO;

// This program asynchronously writes few lines to a text file.

await using var f = new StreamWriter(new FileStream(„test.txt”, // 1
  FileMode.OpenOrCreate, FileAccess.Write,
  FileShare.Read, 4096, true));

await f.WriteLineAsync(„line1”);
await f.WriteLineAsync(„line2”);
```

To dispose an object asynchronously, you need to use a special form of `using` block or declaration called `await using`, whose example is shown above. If the `await` is omitted, the synchronous `Dispose()` method will be called. It's possible for a single object to implement both synchronous and asynchronous dispose patterns or only one of them, depending on the expected usage. For that reason you should make sure to choose the correct form of `using`, as the compiler will not issue a warning in case of selecting the wrong one if both `Dispose` flavors are implemented.

4.2. Async in different application models

.NET can be used in different kinds of applications, including console, web, desktop or mobile applications. Different application models have different constraints related to threading, due to different way they work. The previous examples were all console applications which just called some asynchronous code. Console applications have no special constraints and use the thread pool for running async methods. Let’s describe threading requirements for other application models:

- In case of web applications (written in ASP.NET Core), they generally receive and process HTTP requests and send responses, and these requests/responses are independent from each other, at least where framework is concerned. HTTP protocol is stateless in nature. One application has to be able to serve potentially thousands of requests per second for different clients. For that reason, these requests execute independently from each other on the thread pool. That means there are no additional constraints in comparison to plain console applications, and asynchronous code works exactly the same way. It’s not true for the older ASP.NET applications, but that case is not being described here.
- GUI applications, like *WinForms*, *WPF*, *UWP* or *MAUI* work a bit differently. User interface is stateful and interface elements are shared resources. UI operations are executed on a single thread called the *UI Thread*, thread pool can only be used for tasks not requiring access to the user interface. .NET makes it easy to write asynchronous code which needs to interact with user interface, so asynchronous methods get resumed on the UI thread by default.
- In case of browser applications (like *Blazor*), the concept of asynchronous programming is built on top of the javascript’s support for async, not on threading constructs, and it’s not described here. However, it’s basically similar to the GUI case above, except there is no support for other threads at all.

By default, for applications like web or console applications, no specific threading requirements apply. This means an `await` might resume a method on any free thread. Such a thread will usually belong to the thread pool, but there are situations where it’s not the case. Examples of executing out of thread pool are rare, but might occur depending on various circumstances, mostly related to performance optimizations done by the runtime. A single async method usually executes multiple awaits throughout it’s lifetime, and each of them is allowed to resume on a different thread. You should not write code which assumes a specific thread of execution. Multiple asynchronous methods running at the same time might run concurrently on different threads, and if they share data, applying concurrent programming/multithreading techniques, like inter-thread synchronization, might be necessary. However, a single async method, even when moving between threads, will always execute on single thread at a time.

The above described model does not fit all possible use cases, so .NET has some mechanisms to allow influencing execution of tasks and async methods. The main example where free threading model does not work well is GUI applications, let's explain that case in more detail.

GUI threading model

In GUI frameworks, UI elements are represented as shared, mutable objects. Due to challenges related to concurrent programming, UI elements are usually bound to a thread which created them, and only that single thread can interact with them. In addition, only that thread receives and processes events related to these elements, including input related events. This thread is usually called an *UI thread*. There can be more than one UI thread per application, but each one manages it's own part of the user interface. Often, the program's main thread becomes the UI thread.

UI threads are responsible for receiving, dispatching and processing input and other events related to their bound UI elements. They run an event loop, which polls operating system queues for events and runs event handlers. They can also execute other user defined work, which makes them similar in function to thread pool's worker threads.

When an async method runs on the UI thread, for example it's an asynchronous event handler, the `await` operator will by default arrange the method to always be resumed on the UI thread. That makes it possible to easily write asynchronous code which manipulates the user interface. This is done in .NET using a mechanism called *synchronization context*.

Synchronization context

Synchronization context is a generic mechanism used by .NET to support alternative threading models. Each thread can separately set it's own synchronization context, and when that happens, the `await` operator, if run on that thread, will use it's associated sync context to resume the method instead of using the thread pool. In case of GUI applications, each UI thread has a synchronization context which schedules work back to the same thread. Writing or directly using synchronization context objects is an advanced topic which is out of scope for this guide.

Default `await` behaviour

By default, the behavior of `await` depends on the context where it's being used. For example, in a GUI application, async methods started on UI thread, like event handlers, will be scheduled back on that thread after resumption, but methods originally running on a non ui thread, like on thread pool, for example methods scheduled using `Task.Run()` or other similar mechanisms, will not detect the synchronization context and will use the default threading model.

This example is the fragment of a simple forms application. It has a form with control for displaying text, and when it opens, it's supposed to display the current external IP address of this computer. This example uses the Load event to react on form being opened and the *ipify.org* API to retrieve external IP.

```
private async void OnLoad(object? sender, EventArgs e)
{
    using var client = new HttpClient();
    var ip = await client.GetStringAsync(„http://api.ipify.org/”);
    IPText.Text = ip;
}
```

Automatic marshalling of all async methods back to UI thread has one drawback. Because the same thread is responsible both for acting on UI events and executing other user work, there is a risk of putting too much pressure on that thread, for example by starting too many or too intensive async operations. That will in turn make the UI less responsive, or sometimes could cause it to freeze completely.

Using `ConfigureAwait` method

A solution to this problem is to offload everything that doesn't need access to the UI to the thread pool. When an async method awaits an operation, it is possible to disable the default logic and cause the method to be unconditionally resumed on a thread pool. That's being done by using the `ConfigureAwait()` method on the returned task. It works as follows:

- `ConfigureAwait(true)` is the same as not using the `ConfigureAwait` at all and means that a method needs to be resumed in the same context where `await` happened. Although redundant, this is useful to explicitly show developer's intentions not to ignore the context.

- `ConfigureAwait(false)` means that the method will be resumed on thread pool, ignoring current context.

The below example is the same as above, but it tries to use `ConfigureAwait(false)` during retrieval of external IP address.

```
private async void OnLoad(object? sender, EventArgs e)
{
    using var client = new HttpClient();
    var ip = await client.GetStringAsync(„http://api.ipify.org/")
    .ConfigureAwait(false);
    IPText.Text = ip;
}
```

This code will cause an `InvalidOperationException` to be thrown in debug mode.
This is because, even though the HTTP operation is started on UI thread, the usage of `ConfigureAwait(false)` makes `await` operator to resume the method on the thread pool instead. Accessing the UI from another thread is an error.

If a method leaves the UI thread, it cannot return back to it without calling some API to explicitly schedule work on UI thread. That means if a method calls `ConfigureAwait(false)` at least once, it moves to a thread pool and will stay there until it completes. If you would like to create some complex operation that needs to update the UI, but also is intensive enough to offload most of it to the thread pool (for example networking), you need to factor it into multiple methods, where one method is a helper and executes the actual complex operation on thread pool, and the other one, being an actual method to call, doesn't use `ConfigureAwait(false)`, but instead calls and awaits the helper method, then performs all necessary UI work. Other more complex variations of this pattern are of course possible.

Let's see the below example which is again retrieving external IP address, but this time `HttpClient.GetAsString` is being reimplemented as an excuse to have some non UI operation which could be offloaded to a thread pool.

```
private async void OnLoad(object? sender, EventArgs e)
{
    // We are now on UI thread.
    using var client = new HttpClient();
    var ip = await GetIpAsync(client);
    // We are still on UI thread, even though GetIpAsync
    // completed on thread pool.
    IPText.Text = ip;
}

private async Task<string> GetIpAsync(HttpClient client)
{
    // We are on UI thread.
    var response = await client.GetAsync(„http://api.ipify.org”)
    .ConfigureAwait(false);
    // And now we are on thread pool and will stay there.
    response.EnsureSuccessStatusCode();
    return await response.Content.ReadAsStringAsync()
    .ConfigureAwait(false);
    // Completes on thread pool.
}
```

Notice that the `ConfigureAwait(false)` in `GetIpAsync` is repeated twice. This is a good practice even though a method which uses it once will never be rescheduled back to UI thread without manual intervention.

An important point to consider is that `ConfigureAwait` directly affects only the single await in a single method. It affects where that specific method will be resumed after the specific operation concludes, but does not affect any other async operations, including methods this method calls. In the above example, `GetIpAsync`, because of `ConfigureAwait(false)` used when awaiting `HttpClient` calls, will complete on the thread pool, but after it completes, the `OnLoad` event handler which awaits it's result will still resume on UI thread, because it never used `ConfigureAwait(false)` when awaiting the `GetIpAsync` call.

4.3. Operation cancellation

Asynchronous operations, same as synchronous operations, are not cancellable by default. Task objects can be used to monitor ongoing operations, but they don't have a way to cancel them. Cancellation in .NET is cooperative and relies on the operation itself supporting cancellation. The cancellation mechanism is independent of async and can be used for all kinds of possibly long running or intensive operations.

Generally, a cancellable operation, including asynchronous cancellable operation, indicates cancellation support by it's method having an additional (usually last) parameter of type

`CancellationToken`, often marked as optional. Such a method is written to check the cancellation token status, and to terminate if the token was cancelled. When such a method calls other cancellable operations, it should pass the same cancellation token as argument down to these operations. If it calls non cancellable operations, it's recommended that these operations either don't take much time to complete, or that there exists some other way to abort them which can be employed when cancellation is requested.

Here is an example of a simple console application which runs an async operation, using `Task.Delay(100)` in a loop to simulate some work. The loop is infinite and can only be cancelled by the passed cancellation token.

```
using System.Threading;
using System.Threading.Tasks;

using var cts = new CancellationTokenSource();
// This is intended to model a background operation which we
// don't await immediately.
// We await it after we try to stop it to get it's result.
// Othervise it runs forever.
var operation = ExecuteAsync(cts.Token);

// Cancel the operation, then await to retrieve result.
// The operation will throw when cancelled.
cts.Cancel();
await operation;

static async Task ExecuteAsync(CancellationToken ct = default)
{
  while (true)
  {
    // Our cancellation check.
    ct.ThrowIfCancellationRequested();
    // Fake delay, also make sure to pass token downstream.
    await Task.Delay(100, ct);
  }
}
```

Running this program should cause an `TaskCanceledException` to be displayed. This exception is the result of a cancelled `ExecuteAsync()` method. If the method was not awaited after issuing a cancel, there would be no proof of cancellation being successful, even though the method completed.

5. Asynchronous programming best practices

This section lists some best practices which should be followed when writing asynchronous applications in .NET, especially using async/await language features. These are in addition to best practices mentioned elsewhere in the guide.

Each section contains description of a best practice, it's explanation, good and bad practice examples where applicable, and possible exceptions to the rule.

5.1. Use `ConfigureAwait(false)` when awaiting async operations in libraries

All general purpose libraries which await inside of their own async methods should always use `ConfigureAwait(false)`, so that the thread pool is always used to execute async method's code.

5.1.1. Explanation

General purpose libraries can be used by applications of any kind and should behave correctly no matter of the application's type. However, when not using `ConfigureAwait()` in library's own async methods, these methods would be resumed on a thread pool or on UI thread depending on the application, so behavior of the library would unnecessarily depend on application specific factors. Remember that placing too much work on UI thread can make the GUI unresponsive, and not using `ConfigureAwait(false)` in a library would cause library's internal operations to be resumed on UI thread for all GUI applications. General purpose libraries usually don't need to interact with the UI or application specific framework.

5.1.2. Exceptions

The exception to the rule is libraries which are application model or framework specific, for example ASP.NET Core specific libraries or UI control libraries. Of course, general purpose libraries might contain application model specific parts themselves, and these also aren't bound by this rule if they interact with the model or integrate with specific application framework in use. However, you should follow this best practice in most library code.

5.2. Use `Task.Run()` to execute CPU bound tasks asynchronously

When trying to asynchronously execute a CPU bound operation, use `Task.Run()` or similar TPL method to schedule it on a thread pool.

5.2.1. Explanation

CPU bound operations are synchronous in nature and mostly occupy the thread they execute on. If you have a need to execute such operations asynchronously, the only way to do it is to schedule them onto a different thread. The recommended mechanism is the `Task.Run()` method, which schedules task to the thread pool, instead of creating a dedicated thread for the task. You can use a different method like `Task.Factory.StartNew`, however this is a lower level, more complex method, and it's usage is not recommended for most common scenarios.

Before deciding whether to run a CPU bound task asynchronously, you should to measure the performance of synchronous vs asynchronous execution of the specific operation in a specific context. Synchronous execution is generally easier to manage and doesn't require handling of tasks nor additional context switches, so it's recommended to use it and only use asynchronous execution if there is a performance benefit.

5.2.2. Example

Below is an example of a synchronous method for computing factorial values.

```
static BigInteger ComputeFactorial(int n)
{
  var result = BigInteger.One;

  for (int i = 2; i <= n; i++)
  {
    result *= i;
  }
  return result;
}
```

This method should be called synchronously in most cases, however for larger `n` values, in some contexts, asynchronous execution might be beneficial. If it's needed, it can be done like this:

```
await Task.Run(() => ComputeFactorial(40000));
```

`Task.Run()` returns a `Task` object, which can be awaited in async methods.

5.3. Avoid calling IO bound async methods using `Task.Run`

Asynchronous methods should be directly called without using API's like `Task.Run()` to schedule them. This is true no matter whether they are to be awaited or whether their execution is to be monitored separately.

5.3.1. Explanation

Sometimes it's necessary to execute an asynchronous method without directly awaiting it. For example, in a web application, you might create an API action which begins some long running operation. Then, instead of awaiting this operation, potentially causing the request to time out, you might store operation's task and provide another API action to monitor it's progress and detect completion.

Sometimes, you can be tempted to use `Task.Run()` in such cases like this to begin the operation, as the intention is to execute it asynchronously, but not to directly depend on it's completion:

```
OperationTask = Task.Run(async () => await SomeOperationAsync(...));
```

However, although such code works correctly, using this pattern is completely unnecessary. An asynchronous method itself returns a Task object which can be used to monitor it's progress. Also, even though async methods, when called, execute code synchronously until encountering the first `await`, the synchronous portion is usually very brief. Using `Task.Run()` would unnecessarily schedule that synchronous code to the thread pool, which would actually degrade performance as opposed to executing such a method directly. `Task.Run()` doesn't in any way affect later method execution, as further scheduling is driven by the `await` operator. The exception is applications with alternative threading models like GUI applications, where `Task.Run()` forces a method to run completely on the thread pool, even if normally it would prefer to be scheduled on an UI thread. In such case, using `Task.Run()` would be a bug.

5.3.2. Exceptions

An exception happens when the method's synchronous portion is actually computationally intensive to the point where, at least in specific contexts, scheduling it to run asynchronously gives a performance benefit.

An additional exception to the rule happens when `Task.Run()` would be used in a GUI application to force some external, badly written method which doesn't correctly use `ConfigureAwait(false)` to run on a thread pool. This is just a workaround, and if possible, the offending method should be fixed instead.

5.4. Prefer `Task.Run()` over `TaskFactory.StartNew()`

If there is a need to explicitly schedule operations, whenever possible, use `Task.Run()` to schedule tasks on the thread pool. Use `TaskFactory.StartNew()` only when actually necessary.

5.4.1. Explanation

This was mentioned above but deserves it's own section. `Task.Run()` is preferred over `TaskFactory.StartNew()` because it's easier to use.

The main differences between the two are:

- `Task.Run()` always schedules operations to the thread pool, by using the default task scheduler, no matter the context. `TaskFactory.StartNew()` allows to explicitly choose a scheduler, but defaults to `TaskScheduler.Current`, which means scheduling depends on context. This means it's relatively easy to introduce bugs related to using incorrect scheduler when the intention is to force usage of a thread pool. Both methods don't use a synchronization context by default, even if it's present, so none of these methods default to scheduling tasks on UI thread, for example.

- `Task.Run()` has explicit overloads accepting `Func<Task>` and `Func<Task<T>>` delegates returning a `Task` or `Task<T>` object, which works as you would expect. This makes executing asynchronous methods with `Task.Run()` easier if that's necessary. `TaskFactory.StartNew()`, when given an async delegate, will actually return a `Task<Task>` or `Task<Task<T>>` object. That's because it doesn't handle async methods specially. Instead, it treats every method the same, and at the low level, an async method's result is just it's task.

5.4.2. Example

This is a simple example of using `Task.Run()`:

```
await Task.Run(() => Console.WriteLine(„test”));
```

However, using `TaskFactory.StartNew()` to achieve the same effect is more complex because you have to pass the correct options and task scheduler. This is what `Task.Run()` does internally:

```
await Task.Factory.StartNew(() => Console.WriteLine(„test”),
CancellationToken.None, TaskCreationOptions.DenyChildAttach,
TaskScheduler.Default);
```

The case of asynchronous code being wrapped is the trickier one. For example, this is how to wrap an async method/lambda with `Task.Run`.

```
await Task.Run(async () => await Task.Delay(1000));
```

As you can see, it's straight forward, and even though the lambda has been wrapped by `Task.Run()`, the returned task will correctly handle that case and will complete after the async lambda completes. The difference between this and directly calling the lambda is just that the lambda itself returns a task on encountering first `await`, while `Task.Run` returns a task immediately and it also covers the moment when lambda didn't yet start.

However, that's not so easy with TaskFactory.StartNew(). Namely, this code is not only more complex, but also doesn't work as intended:

```
await Task.Factory.StartNew(async () => await Task.Delay(1000),
CancellationToken.None, TaskCreationOptions.DenyChildAttach,
TaskScheduler.Default);
```

In this case, the `TaskFactory.StartNew()` doesn't return a `Task` instance. Instead, it returns a `Task<Task>` instance.

The above happens because the lambda, even though it's an async lambda, is being treated like an ordinary method here. So `TaskFactory.StartNew` interprets this lambda as an ordinary function whose return type is `Task`, so it just returns a task object covering exclusively the synchronous portion of that lambda, which is the only part being directly scheduled by `StartNew()`. The synchronous fragment completes very quickly by returning the `Task` representing the started operation, which is then treated as normal method's return value and captured by task returned from `StartNew`.

One solution that might come to mind to correctly await async methods wrapped by `TaskFactory.StartNew()` is to double await:

```
await (await Task.Factory.StartNew(async () => await Task.Delay(1000),
CancellationToken.None, TaskCreationOptions.DenyChildAttach,
TaskScheduler.Default));
```

The inner await awaits the `Task<Task>` which completes when the lambda returns a `Task` representing it's own execution, and the outer await grabs this task and awaits it, completing when the delay commences.

However, this is not really intuitive. For that reason, there exists an extension method called Unwrap() which works on `Task<Task>` and `Task<Task<T>>` types, which turns them into ordinary `Task` and `Task<T>` instances, which behave the same as those returned by `Task.Run()` and correctly handle the whole async lambda execution. Here is an example of it's usage, which is actually equivalent to `Task.Run()` example above:

```
await Task.Factory.StartNew(async () => await Task.Delay(1000),
CancellationToken.None, TaskCreationOptions.DenyChildAttach,
TaskScheduler.Default)
.Unwrap();
```

You have to remember about correctly using Unwrap when calling `TaskFactory.StartNew()`, otherwise the await will complete at unexpected times, usually too early, at the end of any async method's synchronous fragment, and this will happen without a warning. This might be a source for hard to detect bugs. This is one more reason why `TaskFactory.StartNew()` should be avoided whenever possible.

5.4.3. Exceptions

`TaskFactory.StartNew()` has it's uses, for example when actually caring about task schedulers or when it's necessary to use non standard task creation options. One such use will be described in section 5.5 below. However, especially when wrapping async methods with `StartNew()`, you have to pay attention to the caveats described above in order not to introduce subtle bugs.

5.5. Avoid usage of thread pool for long running operations

When executing long running operations, do not use a thread pool. Instead, create a dedicated thread for the operation.

If possible, it's recommended to make such operations themselves fully asynchronous, in which case this rule does not apply. This rule applies only if that's not possible or desirable for any reason, including performance. As an example of making long running operations asynchronous, instead of a thread processing items from a `ConcurrentQueue` or a `BlockingCollection`, it is better to use asynchronous aware `Channel` instances.

5.5.1. Explanation

Long running operations are operations which occupy a thread for extended periods of time. These could be really long non parallelizable computations, or operations which, for any reason, require extensive blocking, for example background handling of queues. This doesn't cover IO bound async operations, as these are written to free threads promptly anyway. This also doesn't cover parallel operations, because they are usually divided into many smaller, short running units.

It is generally not recommended to put long running tasks on the thread pool, as it's optimized for reasonably short running work items. Running too many such long operations might exhaust the thread pool and cause new work items not to be executed until a thread is freed. It might also cause creation of extensive number of additional threads to cope with the situation. However, threads aren't added immediately and are expensive resources, so it cannot be seen as a solution to the problem.

You should create dedicated threads for long running operations. One can do it directly or using `TaskFactory.StartNew()` with `TaskCreationOptions.LongRunning` flag.

5.5.2. Example

This is a method mimicking a long running operation which probably runs throughout application's lifetime or until explicitly cancelled:

```
static void Run(CancellationToken ct)
{
  while (!ct.IsCancellationRequested)
  {
    // Do something, potentially waiting on external events...
  }
}
```

This operation is long running as it actually runs until explicitly cancelled. For that reason, it's best to schedule it to a dedicated thread:

```
var thread = new Thread(Run);
thread.Start();
```

If the operation would finish by itself and return a result, you can't capture such a result this way and would need additional code + some thread synchronization for that.

Alternatively, you can use `TaskFactory.StartNew()`, which gives an advantage of having more control over the execution and operation result if any. You can give it a hint that the operation is long running, which might influence the task scheduler's decision. `TaskScheduler.Default`, when seeing this flag, will create a dedicated thread for the operation. Remember that other task schedulers might behave differently, so you have to pay attention to the method invocation. Here is an example:

```
await Task.Factory.StartNew(Run,
CancellationToken.None,
TaskCreationOptions.DenyChildAttach | TaskCreationOptions.LongRunning,
TaskScheduler.Default);
```

5.5.3. Exceptions

Not all cases of long running operations necessarily require dedicated threads. As an example, singleton background jobs which run for the lifetime of the application, if they are written as long running synchronous tasks, might be scheduled on the thread pool, as occupying one of a dozen threads starting from the very beginning might not make an impact for the application. So, you might decide to use thread pool if the number of such tasks is going to be sufficiently rare and doesn't change frequently. As always, it's best to measure performance of each solution to make an informed decision.

5.6. Avoid blocking in async methods

Asynchronous methods should not invoke operations which block the calling thread. This rule can also be generalized to say that no code running on any kind of worker thread, like thread pool or UI thread, should use blocking methods.

5.6.1. Explanation

Async operations run on some kind of generic worker threads like thread pool threads. Using blocking operations on such threads is wasteful, as the thread can't then be used to execute other work, decreasing application's throughput. In case of the thread pool, it might lead to pool exhaustion and the need to create more threads. In case of UI threads, it causes the UI to be less responsive, or in case of long blocking operations, to freeze completely.

The above description applies equally to most non asynchronous work running on a thread pool or UI thread, like event handlers or tasks submitted by `Task.Run()`.

General rule of thumb is that asynchronous methods should mostly call other asynchronous methods instead of their synchronous, blocking counterparts.

5.6.2. Example

Probably one of the more common examples of that problem is the situation where you need to make an operation wait for some time before continuing. Normally you do that by calling `Thread.Sleep()` method:

```
Thread.Sleep(1000); // Wait 1 second.
```

However, that's not a good practice inside of async aware code or generally on thread pool/ui thread, as this will block the current thread for 1 second. You should find an asynchronous way to achieve the same effect. In this case, you should use `Task.Delay()` method.

```
await Task.Delay(1000);
```

5.6.3. Exceptions

There are exceptions to the rule related to operations which block for short periods of time. For example, in the need of inter thread synchronization, it's reasonable to use mechanisms like locks, even in async methods, provided the locked sections of code run relatively fast (for example they only update data structures). That is despite the fact the `lock` keyword blocks a thread if it cannot acquire a lock at the time. When that's not the case and there is a risk of long blocking at the `lock` statement, or there is a need to await inside a lock (which you can't do because locks are bound to the identity of locking thread), use a `SemaphoreSlim`, which supports asynchronous operation and doesn't have thread affinity.

It is also acceptable to use synchronous I/O, usually file I/O, in cases where it's simpler to use and actual I/O is relatively infrequent, like reading and caching configuration, loading code etc.

There might be other programming patterns which require blocking, usually in relation to inter thread synchronization and parallel programming. However, parallel programming is out of scope for this guide.

5.7. Avoid async over sync

Avoid creating async APIs which just call their synchronous counterparts using mechanisms like `Task.Run()`.

5.7.1. Explanation

Async over sync is a term describing the situation where asynchronous API is created by internally calling synchronous (usually blocking) methods which do the actual work. Asynchronicity in this case is achieved by scheduling these calls onto different threads, often using `Task.Run`, and awaiting them. This often happens when trying to create an asynchronous wrapper for synchronous only

API, often because of being forced to work in async environment or just trying to add async api for something which doesn't originally have it.

The problem with this approach is that real asynchronous API should not be blocking any thread for any single operation. Making async methods which do their work by just calling synchronous methods using `Task.Run()` makes it potentially slower in comparison to calling the synchronous API directly, as in addition to a synchronous method having to be executed, it also needs to be scheduled on thread pool, and it's result communicated back to the awaiting method. That way an additional thread becomes wasted without gaining the benefits of async. Of course, running too many such operations might exhaust the thread pool.

5.7.2. Exceptions

CPU bound operations are always synchronous and, if performance considerations apply, it might be better to schedule them on a thread pool to make them asynchronous. In case of IO bound operations, there are cases where no asynchronous api exists, or such api can't be reasonably created in any way other than scheduling the synchronous counterpart on another thread, and such api is required in some context, like using an async heavy framework. Decisions about what to do should be based on the actual usage context and performance measurements. However, calling the api directly should be considered first, if no real async alternative can be found.

5.8. Avoid sync over async

Avoid blocking waits on asynchronous method's task.

5.8.1. Explanation

Sync over async is a practice where an asynchronous method is being called (usually in synchronous code), and then a blocking wait is performed on it's returned task. The motivation is usually that the fragment of code calling the operation cannot, without being rewritten, itself be made async, so it's not possible to await the task, yet the synchronous code actually depends on the operation result or on it's completion. Usually, you would use `Task.Wait()`, `Task.Result` or `Task.GetAwaiter().GetResult()` for this purpose.

The problem with this approach is that using a blocking wait blocks the current thread. It's actually a specific kind of the "do not block worker threads" rule, especially that it's often the case that most code runs on worker/dispatcher threads. The specific difference between using `await` and this approach is that in this case, in addition to a thread (often taken from thread pool) briefly executing code of the called async method, another thread is wasted for waiting, and this other thread is not available for picking up work. In case of UI thread this rule is even more important, because of the fact the called async method is likely to schedule it's work on the same thread (when not using `ConfigureAwait(false)`), using a blocking wait will cause a deadlock. Namely, this is a situation where a blocking wait happening on UI thread depends on an operation which wants to execute code on the same UI thread, which is not available due to the wait. That way the operation can't continue and never completes. This in turn causes the wait never to return and the UI to freeze.

Often, you can actually refactor/rewrite the code for it to become async and to be able to safely await such methods, which will be shown in the example below. Of course, that might not always be possible.

5.8.2. Example

One of the cases where the sync over async antipattern might often be encountered is the case of constructors and properties.

Constructors and properties in C# do not support asynchronous execution. They are required to operate synchronously, however there are cases where you would like to use an asynchronous method inside. Example could be a property which lazily fetches data from database, or a constructor which needs to use asynchronous API to fill object fields. The most obvious and straightforward way to write such constructors and properties is to use a blocking wait.

This is an example `Configuration` class. It loads and caches the configuration from some data source like a database, and allows to asynchronously reload it.

```
using System.Threading.Tasks;

public class Configuration
{
  // Properties...

  // This constructor initializes the configuration, performing an initial
refresh.
  public Configuration()
  {
    RefreshAsync().GetAwaiter().GetResult();
  }

  public async Task RefreshAsync()
  {
    // Use a database to load the configuration...
  }
}
```

This stub class contains a method called `RefreshAsync()`, which is the main method used to perform configuration reload. Because the configuration needs to be eagerly loaded, constructor internally uses this method, however it issues a blocking wait, which is not an ideal solution.

There are ways to avoid such problems. In this case, a good approach would be to create a static factory method which in turn instantiates the object. Such a method could be asynchronous and perform a refresh without blocking. That means the constructor itself should be private.

```
using System.Threading.Tasks;

public class Configuration
{
  // Properties...

  private Configuration()
  {
    // Empty, most work is done by factory.
  }


  public static async Task<Configuration> CreateAsync()
  {
    var cfg = new Configuration();
    await cfg.RefreshAsync();
    return cfg;
  }

  public async Task RefreshAsync()
  {
    // Use a database to load the configuration...
  }
}
```

A similar approach could be used to work around the need to call async methods in properties. You could either cache their result eagerly or replace them with async getter/setter methods as appropriate.

5.8.3. Exceptions

There are cases where blocking waits are acceptable or even appropriate. For example, when calling asynchronous methods on dedicated threads, there is no other choice, although it's better to use their synchronous blocking counterparts in such cases, if any are available. Also, there are cases where rewrite/refactor to async is not a viable option, for example in some legacy code (remember async being viral. As always, however, it's a good practice to measure performance of the given solution and choose the best approach based on results.

5.9. Do not use the `async` keyword for methods which always return synchronously

Methods which never need to `await` should not use the async keyword, even if they return a task. They should be written like synchronous methods, returning results using `Task.FromResult()`, `Task.CompletedTask` and similar API directly.

5.9.1. Explanation

Sometimes it is necessary to write a synchronous method which behaves like asynchronous methods. For example, when implementing an interface which exposes only asynchronous methods, but where the implementation doesn't need to perform any actual asynchronous work (for example fixed/eagerly cached/precomputed data). In such a case, you are still required to write a method which complies to the async api signature.

In such cases, however, no `await` is required in the method's code. Writing such an async method without `await` actually issues a compiler warning. The correct approach is to skip the async keyword and to write the method by manually returning tasks with appropriate apis.

5.9.2. Example

This is an example interface and it's stub implementation.

```
using System.Collections.Generic;
using System.Threading.Tasks;

// This is an interface for getting values based on keys from a data
source.
public interface IKeyValueStore
{
  Task<string> GetAsync(string key);
  Task SetAsync(string key, string value);
}

// This is a stub class used for tests, for example.
public class TestStore : IKeyValueStore
{
  public Task<string> GetAsync(string key)
  {
    if (key == „ProductVersion”)
    {
      // Return a task manually.
      return Task.FromResult(„1.0.0”);
    }

    // Behave same as async methods, return a task with
    // wrapped exception instead of throwing directly.
    return Task.FromException<string>(new KeyNotFoundException());
  }

  public Task SetAsync(string key, string value) =>
  Task.CompletedTask;
}
```

Be careful about the behavior of exceptions thrown by such synchronous methods, directly or indirectly. Normally, asynchronous methods wrap all exceptions in the returned task object. That means it's good practice to use return `Task.FromException` instead of throw to surface exceptions from task returning synchronous methods, if possible, to minimize surprises.

5.10.Throw argument related exceptions directly instead of wrapping them in tasks

Arrange async methods so that exceptions related to argument validation are thrown directly, instead of being wrapped in a task object.

5.10.1. Explanation

Argument validation exceptions signify actual programmer errors and it's best for them to be thrown as soon as possible. In case of asynchronous methods, they would ideally be thrown directly from the method itself instead of being wrapped in some kind of task object.

The problem here is that all exceptions thrown from methods marked async are wrapped in their returned tasks. The solution to the problem is to split an async method in two parts, one marked async doing actual work, and another one without the async modifier, which performs argument validation then calls the other method and returns it's task. In fact, there is a feature called local functions in C# language, which was created specifically to address similar use cases.

5.10.2. Example

This is an example of an async method doing argument validation. The method is split in two parts where the first one is a method which is to be called by the user and which does argument validation, and the second part does the real work. The second method is represented as a local function. This makes it explicit that these two parts are actually related.

```
static Task MyDelay(int n)
{
  // Argument validation
  if (n <= 0)
  {
    throw new ArgumentException(nameof(n));
  }

  return MyDelayCore(n);

  static async Task MyDelayCore(int n)
  {
    // Do the real work, all exceptions here are wrapped.
    await Task.Delay(n);
  }
}
```

6. Further learning

This guide does not cover all the details about asynchronous programming and related features, like the TPL. Below is a non-exhaustive list of external resources, like documentation and blog posts, which you can use to further extend your knowledge:

- Microsoft's threading documentation,
- More information about asynchronous programming patterns, including the TAP,
- Description of implementing the `IAsyncDisposable` interface,
- Asynchronous programming best practices written by ASP.NET Core team,
- Stephen Toub's detailed blog article about history of async in .NET and inner-workings of async/await feature,
- `System.Threading.Tasks.Task` API reference for .NET 7.0.

.NET Poland Job Family

.NET Job Family is an internal unit at GFT Poland, making sure that its members have everything to follow the desired development path in their career. We help you decide, provide guidance and knowledge, support with trainings, conferences and certifications. You just pursuit your dream.

We are the experts, group of coworkers gathered around Microsoft technology stack. This means little less than a hundred of well skilled and fully experienced .NET specialists. More than 80% of us already achieved senior level and we can describe our tech leaders as versatile unique people with usually more than 10 years of professional experience.

In GFT we mostly work in FinTech, but we are not attached only to financial area. We are involved in various types of projects, like automotive or DevOps related. We are developers and consultants. Our goal is to understand the core of the business, discover and address the risks upfront, help to decide which road should be taken. Greenfield, legacy, redesign, move to cloud – that's what we do. Frontend, backend, full stack, whole system design? Sure, we can.

We always seek for something extra from our work, a little more than a bit of fun. Automation frameworks, chat bots, performance improvements, DevOps activities – we are not afraid of anything.

With such a group of unique people there is always area to gain new skills, knowledge and experience. We meet on a regular basis, keep every member up to date with all family and company related news and changes. We share technical knowledge, experiences from solved problems, introduce projects we are working on and allow other to learn from our mistakes. We do it live, online and in our offices. After work we continue to integrate on regular family events. We are not scared of flying axes, fast and furious karts or heavy bowling balls. We do whatever is necessary to give everyone feeling that we are taking care of each other and got our back.

Give us the challenge, we will find the right tools and overcome it. People capable of achieving that we already have!