**Asynchronous Programming C#**

# [When should I use Lazy<T>?](https://stackoverflow.com/questions/6847721/when-should-i-use-lazyt)

You typically use it when you want to instantiate something the first time its actually used. This delays the cost of creating it till if/when it's needed instead of always incurring the cost.

Usually this is preferable when the object may or may not be used and the cost of constructing it is non-trivial.

## Introduction

The System.Lazy<T> class is a very nice .NET Framework class that supports any object lazy initialization. In this tip, I explain with an example how we can use System.Lazy<T> class for lazy object initialization as well as implement singleton design pattern with the help of System.Lazy<T> class.

## Background

I am waiting for lazy loading feature in .NET Framework. I thought a popular framework like .NET Framework should support that type of feature. Fortunately, it comes with .NET Framework version 4 or later. In this post, I explain in detail with an appropriate example.

## Using the Code

At first, I create a Product entity class that has:

* + - **2 property members**
      * Id
      * Title
    - **and 2**static **method members**
      * GetProductListLazy
      * GetProductList

public class Product

{

public long Id { get; set; }

public string Title { get; set; }

public Product()

{

Thread.Sleep(5000);

}

public static IList<Lazy<Product>> GetProductListLazy()

{

var product1 = new Lazy<Product>(() =>

new Product { Id = 1, Title = "Product 1" });

var product2 = new Lazy<Product>(() =>

new Product { Id = 2, Title = "Product 2" });

var product3 = new Lazy<Product>(() =>

new Product { Id = 3, Title = "Product 3" });

return new List<Lazy<Product>>

{product1, product2, product3};

}

public static IList<Product> GetProductList()

{

var product1 = new Product

{ Id = 1, Title = "Product 1" };

var product2 = new Product

{ Id = 2, Title = "Product 2" };

var product3 = new Product

{ Id = 3, Title = "Product 3" };

return new List<Product>

(new[]{product1, product2, product3 });

}

}*//end Product class*

GetProductListLazy returns a List of Product object with Lazy Initialization support with the help of System.Lazy<T> class.

GetProductList returns a List of Product object without any lazy initialization support.

In my Product class default constructor, I hold code execution for 5 seconds using Thread.Sleepmethod

public Product()

{

Thread.Sleep(5000);*//5000 millisecond = 5 second*

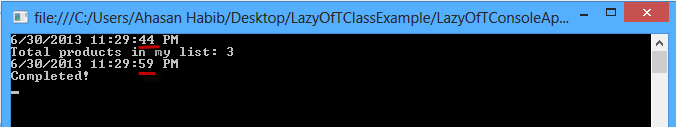
}

Now, I want to print the number of products containing the list in the console. I just call Product.GetProductList static method.

IList<Product> list = Product.GetProductList();

Console.WriteLine("Total products in my list: " + list.Count);

To print result (number of items in the list), it takes 15 seconds.



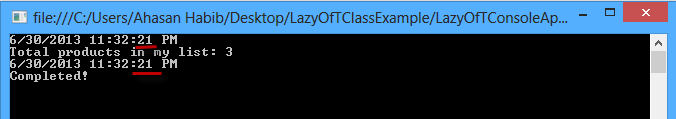
Why, because it first creates 3 instances of Product class, then adds to the list and returns. Per object creation takes 5 seconds. I explain before I use Thread.Sleep for achieving that.

On the other hand, if I called Product.GetProductListLazy method...

IList<Lazy<Product>> list = Product.GetLazyProductList();

Console.WriteLine("Total products in my list: " + list.Count);

...then it prints result (number of items in the list) within a millisecond.



Wow! I got my desired result!

If I iterate that list object, then we can understand in run-time when we actually access any of that object's member, that time object is created.

<https://www.codeproject.com/Articles/612801/Lazy-Object-Initialization-with-System-Lazy-of-T-c>

# How to Convert Your Old Sequential Code in to async

There are plenty of ansyc samples over the internet, and most of them are different and do not satisfy your requirements. Actually, async pattern depends on its creator, and can be implemented in various ways. It is important to understand the async pattern in order to use it. Only in this way, you can stop searching for the exact sample you need, and start coding your own async code. More than a year ago, I wrote a simple blog post about async pattern ([part 1](http://bhrnjica.net/2011/03/27/async-i-await-visenitno-programiranje-bez-niti-1-dio/) and [part 2](http://bhrnjica.net/2011/04/08/async-i-await-visenitno-programiranje-bez-niti-ii-dio/)) (Bosnian language), and also wrote how to call [Entity Framework](http://bhrnjica.net/2011/10/13/asynchronous-call-of-entity-framework-in-c-5-0/) with async pattern as well.

Today, I am going to show you how old synchronous code block converts into asynchronous. I think it is interesting because async pattern can improve your existing applications in various ways. First of all, async pattern can increase the responsiveness of an application, performance, etc.

<https://www.codeproject.com/Articles/479481/Howplustoplusconvertplusyourplusoldplussequentialp>

The sample application calculates how many prime numbers exist in range. You need to enter number, press run button. The program starts counting. The progress bars inform the user status of counting.

Let's see the implementation of runBtn\_Click event:

private void btnRun\_Click(object sender, EventArgs e)

{

if(!int.TryParse(textBox1.Text,out m\_number))

m\_number= 1000000;

textBox1.Text = m\_number.ToString();

progressBar1.Maximum = m\_number;

progressBar1.Minimum = 0;

progressBar1.Value = 0;

*//call start calculation*

startCounting();

}

At the beginning of the btnRun\_Click, we prepare progressBar, and also convert text from textbox into inttype. At the end of the function, startCounting method is called. Here is the source code of the method:

private void startCounting()

{

int counter=0;

for(int i=2; i< m\_number; i++)

{

bool retVal= IsPrime(i);

progressBar1.Value++;

if (retVal)

{

counter++;

label3.Text = "Result is: " + counter.ToString();

}

}

}

The method is very simple. It iterates from 2 to specified number by calling helper method IsPrime (see the source code of the blog post) to check if a certain number is prime. Then the method increased the counter variable and tried to update label about current count value. If you run the sample and press run button, you can see that the application is not responsive on user input, and represents classic sequential, synchronous single thread application.

Now I am going to show how this implementation can be converted into async with minimum code changes. We are going to change only startCounting method, other code will remain the same. Explanation is divided in only 3 steps, which is enough to convert our code into full async pattern.

* Put async keyword right after public modifier of startCounting method.

**Explanation**: Every method which implements async patter needs to be decorated with async.

* Put sequential implementation in to Task action, and wait.

**Explanation**:With this, you define a Task object which will run the code without blocking the main thread. The simplest implementation is the following:

private async void startCalculation()

{

var task = new Task(() =&gt;

{

int counter = 0;

for (int i = 2; i < m\_number; i++)

{

bool retVal = IsPrime(i);

this.Invoke((Action)(()=>

{

progressBar1.Value++;

if (retVal)

{

counter++;

label3.Text = "Result is: " + counter.ToString();

}

}));

}

});

task.Start();

await task;

}

Now if you run the sample, you have fully asynchronous implementation. Main thread is free and can receive user input. So this is the simplest way how to convert your sync code in to async. This is the case when Task object creates another thread in order to execute the code. That’s why we called this.Invoke method in order to set controls properties progressBar.Value and label3.Text. Everything else remain the same as in the previous implementation.

* Create global variable of type CancellationTokenSourceand call Cancel method from Cancel Event handler.

**Explanation:** In this way, we can cancel counting at any time. With this case, we need to implement extra code in our previous implementation like the following:

private async void RunProces()

{

if (m\_IsRunning)

return;

m\_IsRunning = true;

int counter=0;

if (m\_ct != null)

{

m\_ct.Dispose();

m\_ct = null;

}

m\_ct = new CancellationTokenSource();

var task = new Task(() =>

{

for (int i = 0; i < m\_number; i++)

{

bool retVal = IsPrime(i);

this.Invoke((Action)(()=>

{

progressBar1.Value++;

if (retVal)

{

counter++;

label3.Text = "Result is: " + counter.ToString();

}

}));

if (m\_ct.IsCancellationRequested)

{

m\_IsRunning = false;

return;

}

}

}, m\_ct.Token);

task.Start();

await task;

}

With the last code implementation, you have full of async patter in your old application.

In this blog post, I have tried to explain as simply as possible the way in which you can convert your old sync code into new async programming pattern.

# [When correctly use Task.Run and when just async-await](https://stackoverflow.com/questions/18013523/when-correctly-use-task-run-and-when-just-async-await)

Note the [guidelines for performing work on a UI thread](http://blog.stephencleary.com/2013/04/ui-guidelines-for-async.html), collected on my blog:

* Don't block the UI thread for more than 50ms at a time.
* You can schedule ~100 continuations on the UI thread per second; 1000 is too much.

There are two techniques you should use:

**1) Use ConfigureAwait(false) when you can.**

E.g., await MyAsync().ConfigureAwait(false); instead of await MyAsync();.

ConfigureAwait(false) tells the await that you do not need to resume on the current context (in this case, "on the current context" means "on the UI thread"). However, for the rest of that asyncmethod (after the ConfigureAwait), you cannot do anything that assumes you're in the current context (e.g., update UI elements).

For more information, see my MSDN article [Best Practices in Asynchronous Programming](http://msdn.microsoft.com/en-us/magazine/jj991977.aspx).

**2) Use Task.Run to call CPU-bound methods.**

You should use Task.Run, but not within any code you want to be reusable (i.e., library code). So you use Task.Run to *call* the method, not as part of the *implementation* of the method.

So purely CPU-bound work would look like this:

// Documentation: This method is CPU-bound.

void DoWork();

Which you would call using Task.Run:

await Task.Run(() => DoWork());

Methods that are a *mixture* of CPU-bound and I/O-bound should have an Async signature with documentation pointing out their CPU-bound nature:

// Documentation: This method is CPU-bound.

Task DoWorkAsync();

Which you would also call using Task.Run (since it is partially CPU-bound):

await Task.Run(() => DoWorkAsync());

# <https://stackoverflow.com/questions/18013523/when-correctly-use-task-run-and-when-just-async-await>

[**Do you have to put Task.Run in a method to make it async?**](https://stackoverflow.com/questions/17119075/do-you-have-to-put-task-run-in-a-method-to-make-it-async)

First, let's clear up some terminology: "asynchronous" (async) means that it may yield control back to the calling thread before it starts. In an async method, those "yield" points are await expressions.

This is very different than the term "asynchronous", as (mis)used by the MSDN documentation for years to mean "executes on a background thread".

To futher confuse the issue, async is very different than "awaitable"; there are some async methods whose return types are not awaitable, and many methods returning awaitable types that are not async.

Enough about what they *aren't*; here's what they *are*:

* The async keyword allows an asynchronous method (that is, it allows await expressions). async methods may return Task, Task<T>, or (if you must) void.
* Any type that follows a certain pattern can be awaitable. The most common awaitable types are Task and Task<T>.

So, if we reformulate your question to "how can I run an operation *on a background thread* in a way that it's awaitable", the answer is to use Task.Run:

private Task<int> DoWorkAsync() // No async because the method does not need await

{

return Task.Run(() =>

{

return 1 + 2;

});

}

(But this pattern is a poor approach; see below).

But if your question is "how do I create an async method that can yield back to its caller instead of blocking", the answer is to declare the method async and use await for its "yielding" points:

private async Task<int> GetWebPageHtmlSizeAsync()

{

var client = new HttpClient();

var html = await client.GetAsync("http://www.example.com/");

return html.Length;

}

So, the basic pattern of things is to have async code depend on "awaitables" in its awaitexpressions. These "awaitables" can be other async methods or just regular methods returning awaitables. Regular methods returning Task/Task<T> *can* use Task.Run to execute code on a background thread, or (more commonly) they can use TaskCompletionSource<T> or one of its shortcuts (TaskFactory.FromAsync, Task.FromResult, etc). I **don't** recommend wrapping an entire method in Task.Run; synchronous methods should have synchronous signatures, and it should be left up to the consumer whether it should be wrapped in a Task.Run:

private int DoWork()

{

return 1 + 2;

}

private void MoreSynchronousProcessing()

{

// Execute it directly (synchronously), since we are also a synchronous method.

var result = DoWork();

...

}

private async Task DoVariousThingsFromTheUIThreadAsync()

{

// I have a bunch of async work to do, and I am executed on the UI thread.

var result = await Task.Run(() => DoWork());

...

}

I have an [async/await intro](http://blog.stephencleary.com/2012/02/async-and-await.html) on my blog; at the end are some good followup resources. The MSDN docs for async are unusually good, too.

# Simple Async Await Example for Asynchronous Programming

[11 Replies](https://stephenhaunts.com/2014/10/10/simple-async-await-example-for-asynchronous-programming/#comments)

In this article I want to give a simple example for how to use the async await keywords in C# to create asynchronous background tasks.

Writing multi-threaded or asynchronous code has traditionally always been very hard to get right but is something that is needed to help keep our applications responsive and to avoid performance bottlenecks. C# 5 introduced a simplified model for doing asynchronous programming with the introduction of 2 new keywords, async and await.

If you specify that a method is an asynchronous method by using an async modifier, you enable the following two capabilities.

* The marked async method can use the await keyword to designate suspension points. The await operator tells the compiler that the async method can’t continue past that point until the awaited asynchronous process is complete. In the meantime, control returns to the caller of the async method. The suspension of an async method at an await expression doesn’t constitute an exit from the method, and finally blocks don’t run.
* The marked async method can itself be awaited by methods that call it.

An async method typically contains one or more occurrences of an await operator, but the absence of await expressions doesn’t cause a compiler error. If an async method doesn’t use an await operator to mark a suspension point, the method executes as a synchronous method does, despite the async modifier. The compiler issues a warning for such methods.

Asynchrony is essential for activities that are potentially blocking, such as when your application accesses the web or a file system. Access to a web resource, for example, is sometimes slow or delayed. If such an activity is blocked within a synchronous process, the entire application must wait. In an asynchronous process, the application can continue with other work that doesn’t depend on the web resource until the potentially blocking task finishes.

Lets take a look at a simple example.

using System;

using System.Threading.Tasks;

namespace AsyncAwaitExample

{

class Program

{

static void Main()

{

var demo = new AsyncAwaitDemo();

demo.DoStuff();

while (true)

{

Console.WriteLine("Doing Stuff on the Main Thread...................");

}

}

}

public class AsyncAwaitDemo

{

public async Task DoStuff()

{

await Task.Run(() =>

{

LongRunningOperation();

});

}

private static async Task<string> LongRunningOperation()

{

int counter;

for (counter = 0; counter < 50000; counter++)

{

Console.WriteLine(counter);

}

return "Counter = " + counter;

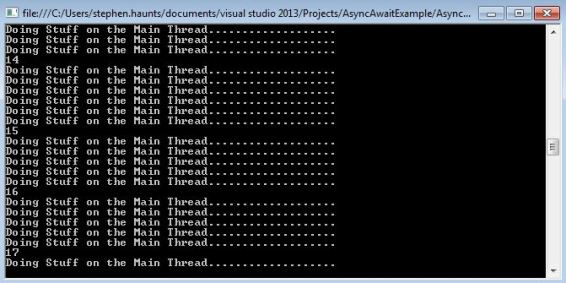
}

}

}

This program contains a class called **AsyncAwaitDemo**. In this class there is a public method called **DoStuff()** which launches a long running operation called **LongRunningOperation()**. This method is marked with the async keyword. For this simple example, the **LongRunningOperation()** method just counts to 50000 and prints the counter to the console.

Back in our **Main()** method, after executing the **DoStuff()** method we then go into an infinite while loop that prints to the console. When you run the program you see something like the following.

[](https://stephenhaunts.files.wordpress.com/2014/10/async_example.jpg)

*Here the code is running asynchronously.*

What you see here is that the infinite while loop is executing, but the long running task we kick off is also running in the background and counting up to 50000.

Lets modify the code slightly by removing the Task.Run, so that the method looks like the following :

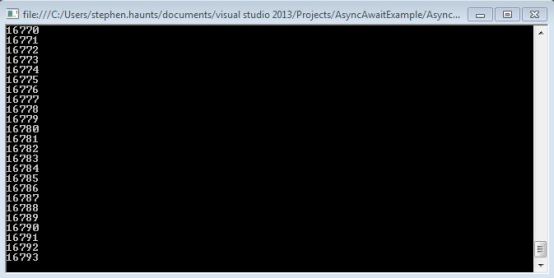
public async Task DoStuff()

{

LongRunningOperation();

}

If we then run the program we get the following.

[](https://stephenhaunts.files.wordpress.com/2014/10/async_example2.jpg)

*Here the code is running synchronously.*

Because we have removed the line with the **await** keyword, the **LongRunningMethod()** now runs synchronously, so the counter will first count up to 50000 and then the code will go into the while loop.

## Avoid Async Void

There are three possible return types you can use for async methods:

* Task
* Task<T>
* void

The main return types for async methods are just Task and Task<T>. When you are converting from synchronous to asynchronous code, any method returning a type becomes an async method returning Task<T>, and any method returning void becomes an async method returning Task.

Async methods that return void have a specific purpose and that is to make asynchronous error handlers possible, but async void methods have different error handling semantics. When an exception is thrown in an async Task or async Task<T> method, that exception is captured and placed directly on the Task object. With an async void method, there is no Task object involved, so any exceptions thrown out in an async void method will be raised directly on the SynchronizationContext that was active when the async void method started. For more information I recommend reading the following [MSDN article](https://msdn.microsoft.com/en-us/magazine/jj991977.aspx) that has some good examples.

## Wrapping Up

In this simple example we used the **Task.Run()** method to incorporate the await keyword into the code, but .NET framework also provides many async compatible API’s that you can also use with async await. You can recognize these members by the “Async” suffix that’s attached to the member name and a return type of **Task** or **Task**. For example, the System.IO.Stream class contains methods such as **CopyToAsync**, **ReadAsync**, and **WriteAsync** alongside the synchronous methods**CopyTo**, **Read**, and **Write**.

This is a nice and simple example that I hope help you to understand the usage of async / await. It is a flexible and powerful feature of C# 5 and above that should make it much easier for you to write responsive applications.

<https://stephenhaunts.com/2014/10/10/simple-async-await-example-for-asynchronous-programming/>

# Async/Await - Best Practices in Asynchronous Programming

By [Stephen Cleary](https://msdn.microsoft.com/en-us/magazine/mt149362?author=stephen+cleary) | March 2013

These days there’s a wealth of information about the new async and await support in the Microsoft .NET Framework 4.5. This article is intended as a “second step” in learning asynchronous programming; I assume that you’ve read at least one introductory article about it. This article presents nothing new, as the same advice can be found online in sources such as Stack Overflow, MSDN forums and the async/await FAQ. This article just highlights a few best practices that can get lost in the avalanche of available documentation.

The best practices in this article are more what you’d call “guidelines” than actual rules. There are exceptions to each of these guidelines. I’ll explain the reasoning behind each guideline so that it’s clear when it does and does not apply. The guidelines are summarized in **Figure 1**; I’ll discuss each in the following sections.

**Figure 1 Summary of Asynchronous Programming Guidelines**

|  |  |  |
| --- | --- | --- |
| Name | Description | Exceptions |
| Avoid async void | Prefer async Task methods over async void methods | Event handlers |
| Async all the way | Don’t mix blocking and async code | Console main method |
| Configure context | Use ConfigureAwait(false) when you can | Methods that require con­text |

## Avoid Async Void

There are three possible return types for async methods: Task, Task<T> and void, but the natural return types for async methods are just Task and Task<T>. When converting from synchronous to asynchronous code, any method returning a type T becomes an async method returning Task<T>, and any method returning void becomes an async method returning Task. The following code snippet illustrates a synchronous void-returning method and its asynchronous equivalent:

void MyMethod()

{

  // Do synchronous work.

  Thread.Sleep(1000);

}

async Task MyMethodAsync()

{

  // Do asynchronous work.

  await Task.Delay(1000);

}

Void-returning async methods have a specific purpose: to make asynchronous event handlers possible. It is possible to have an event handler that returns some actual type, but that doesn't work well with the language; invoking an event handler that returns a type is very awkward, and the notion of an event handler actually returning something doesn't make much sense. Event handlers naturally return void, so async methods return void so that you can have an asynchronous event handler. However, some semantics of an async void method are subtly different than the semantics of an async Task or async Task<T> method.

Async void methods have different error-handling semantics. When an exception is thrown out of an async Task or async Task<T> method, that exception is captured and placed on the Task object. With async void methods, there is no Task object, so any exceptions thrown out of an async void method will be raised directly on the SynchronizationContext that was active when the async void method started. **Figure 2** illustrates that exceptions thrown from async void methods can’t be caught naturally.

**Figure 2 Exceptions from an Async Void Method Can’t Be Caught with Catch**

private async void ThrowExceptionAsync()

{

  throw new InvalidOperationException();

}

public void AsyncVoidExceptions\_CannotBeCaughtByCatch()

{

  try

  {

    ThrowExceptionAsync();

  }

  catch (Exception)

  {

    // The exception is never caught here!

    throw;

  }

}

These exceptions can be observed using AppDomain.UnhandledException or a similar catch-all event for GUI/ASP.NET applications, but using those events for regular exception handling is a recipe for unmaintainability.

Async void methods have different composing semantics. Async methods returning Task or Task<T> can be easily composed using await, Task.WhenAny, Task.WhenAll and so on. Async methods returning void don’t provide an easy way to notify the calling code that they’ve completed. It’s easy to start several async void methods, but it’s not easy to determine when they’ve finished. Async void methods will notify their SynchronizationContext when they start and finish, but a custom SynchronizationContext is a complex solution for regular application code.

Async void methods are difficult to test. Because of the differences in error handling and composing, it’s difficult to write unit tests that call async void methods. The MSTest asynchronous testing support only works for async methods returning Task or Task<T>. It’s possible to install a SynchronizationContext that detects when all async void methods have completed and collects any exceptions, but it’s much easier to just make the async void methods return Task instead.

It’s clear that async void methods have several disadvantages compared to async Task methods, but they’re quite useful in one particular case: asynchronous event handlers. The differences in semantics make sense for asynchronous event handlers. They raise their exceptions directly on the SynchronizationContext, which is similar to how synchronous event handlers behave. Synchronous event handlers are usually private, so they can’t be composed or directly tested. An approach I like to take is to minimize the code in my asynchronous event handler—for example, have it await an async Task method that contains the actual logic. The following code illustrates this approach, using async void methods for event handlers without sacrificing testability:

private async void button1\_Click(object sender, EventArgs e)

{

  await Button1ClickAsync();

}

public async Task Button1ClickAsync()

{

  // Do asynchronous work.

  await Task.Delay(1000);

}

Async void methods can wreak havoc if the caller isn’t expecting them to be async. When the return type is Task, the caller knows it’s dealing with a future operation; when the return type is void, the caller might assume the method is complete by the time it returns. This problem can crop up in many unexpected ways. It’s usually wrong to provide an async implementation (or override) of a void-returning method on an interface (or base class). Some events also assume that their handlers are complete when they return. One subtle trap is passing an async lambda to a method taking an Action parameter; in this case, the async lambda returns void and inherits all the problems of async void methods. As a general rule, async lambdas should only be used if they’re converted to a delegate type that returns Task (for example, Func<Task>).

To summarize this first guideline, you should prefer async Task to async void. Async Task methods enable easier error-handling, composability and testability. The exception to this guideline is asynchronous event handlers, which must return void. This exception includes methods that are logically event handlers even if they’re not literally event handlers (for example, ICommand.Execute implementations).

## Async All the Way

Asynchronous code reminds me of the story of a fellow who mentioned that the world was suspended in space and was immediately challenged by an elderly lady claiming that the world rested on the back of a giant turtle. When the man enquired what the turtle was standing on, the lady replied, “You’re very clever, young man, but it’s turtles all the way down!” As you convert synchronous code to asynchronous code, you’ll find that it works best if asynchronous code calls and is called by other asynchronous code—all the way down (or “up,” if you prefer). Others have also noticed the spreading behavior of asynchronous programming and have called it “contagious” or compared it to a zombie virus. Whether turtles or zombies, it’s definitely true that asynchronous code tends to drive surrounding code to also be asynchronous. This behavior is inherent in all types of asynchronous programming, not just the new async/await keywords.

“Async all the way” means that you shouldn’t mix synchronous and asynchronous code without carefully considering the consequences. In particular, it’s usually a bad idea to block on async code by calling Task.Wait or Task.Result. This is an especially common problem for programmers who are “dipping their toes” into asynchronous programming, converting just a small part of their application and wrapping it in a synchronous API so the rest of the application is isolated from the changes. Unfortunately, they run into problems with deadlocks. After answering many async-related questions on the MSDN forums, Stack Overflow and e-mail, I can say this is by far the most-asked question by async newcomers once they learn the basics: “Why does my partially async code deadlock?”

**Figure 3** shows a simple example where one method blocks on the result of an async method. This code will work just fine in a console application but will deadlock when called from a GUI or ASP.NET context. This behavior can be confusing, especially considering that stepping through the debugger implies that it’s the await that never completes. The actual cause of the deadlock is further up the call stack when Task.Wait is called.

**Figure 3 A Common Deadlock Problem When Blocking on Async Code**

public static class DeadlockDemo

{

  private static async Task DelayAsync()

  {

    await Task.Delay(1000);

  }

  // This method causes a deadlock when called in a GUI or ASP.NET context.

  public static void Test()

  {

    // Start the delay.

    var delayTask = DelayAsync();

    // Wait for the delay to complete.

    delayTask.Wait();

  }

}

The root cause of this deadlock is due to the way await handles contexts. By default, when an incomplete Task is awaited, the current “context” is captured and used to resume the method when the Task completes. This “context” is the current SynchronizationContext unless it’s null, in which case it’s the current TaskScheduler. GUI and ASP.NET applications have a SynchronizationContext that permits only one chunk of code to run at a time. When the await completes, it attempts to execute the remainder of the async method within the captured context. But that context already has a thread in it, which is (synchronously) waiting for the async method to complete. They’re each waiting for the other, causing a deadlock.

Note that console applications don’t cause this deadlock. They have a thread pool SynchronizationContext instead of a one-chunk-at-a-time SynchronizationContext, so when the await completes, it schedules the remainder of the async method on a thread pool thread. The method is able to complete, which completes its returned task, and there’s no deadlock. This difference in behavior can be confusing when programmers write a test console program, observe the partially async code work as expected, and then move the same code into a GUI or ASP.NET application, where it deadlocks.

The best solution to this problem is to allow async code to grow naturally through the codebase. If you follow this solution, you’ll see async code expand to its entry point, usually an event handler or controller action. Console applications can’t follow this solution fully because the Main method can’t be async. If the Main method were async, it could return before it completed, causing the program to end. **Figure 4** demonstrates this exception to the guideline: The Main method for a console application is one of the few situations where code may block on an asynchronous method.

**Figure 4 The Main Method May Call Task.Wait or Task.Result**

class Program

{

  static void Main()

  {

    MainAsync().Wait();

  }

  static async Task MainAsync()

  {

    try

    {

      // Asynchronous implementation.

      await Task.Delay(1000);

    }

    catch (Exception ex)

    {

      // Handle exceptions.

    }

  }

}

Allowing async to grow through the codebase is the best solution, but this means there’s a lot of initial work for an application to see real benefit from async code. There are a few techniques for incrementally converting a large codebase to async code, but they’re outside the scope of this article. In some cases, using Task.Wait or Task.Result can help with a partial conversion, but you need to be aware of the deadlock problem as well as the error-handling problem. I’ll explain the error-handling problem now and show how to avoid the deadlock problem later in this article.

Every Task will store a list of exceptions. When you await a Task, the first exception is re-thrown, so you can catch the specific exception type (such as InvalidOperationException). However, when you synchronously block on a Task using Task.Wait or Task.Result, all of the exceptions are wrapped in an AggregateException and thrown. Refer again to **Figure 4**. The try/catch in MainAsync will catch a specific exception type, but if you put the try/catch in Main, then it will always catch an AggregateException. Error handling is much easier to deal with when you don’t have an AggregateException, so I put the “global” try/catch in MainAsync.

So far, I’ve shown two problems with blocking on async code: possible deadlocks and more-complicated error handling. There’s also a problem with using blocking code within an async method. Consider this simple example:

public static class NotFullyAsynchronousDemo

{

  // This method synchronously blocks a thread.

  public static async Task TestNotFullyAsync()

  {

    await Task.Yield();

    Thread.Sleep(5000);

  }

}

This method isn’t fully asynchronous. It will immediately yield, returning an incomplete task, but when it resumes it will synchronously block whatever thread is running. If this method is called from a GUI context, it will block the GUI thread; if it’s called from an ASP.NET request context, it will block the current ASP.NET request thread. Asynchronous code works best if it doesn’t synchronously block. **Figure 5** is a cheat sheet of async replacements for synchronous operations.

**Figure 5 The “Async Way” of Doing Things**

|  |  |  |
| --- | --- | --- |
| To Do This … | Instead of This … | Use This |
| Retrieve the result of a background task | Task.Wait or Task.Result | await |
| Wait for any task to complete | Task.WaitAny | await Task.WhenAny |
| Retrieve the results of multiple tasks | Task.WaitAll | await Task.WhenAll |
| Wait a period of time | Thread.Sleep | await Task.Delay |

To summarize this second guideline, you should avoid mixing async and blocking code. Mixed async and blocking code can cause deadlocks, more-complex error handling and unexpected blocking of context threads. The exception to this guideline is the Main method for console applications, or—if you’re an advanced user—managing a partially asynchronous codebase.

## Configure Context

Earlier in this article, I briefly explained how the “context” is captured by default when an incomplete Task is awaited, and that this captured context is used to resume the async method. The example in **Figure 3** shows how resuming on the context clashes with synchronous blocking to cause a deadlock. This context behavior can also cause another problem—one of performance. As asynchronous GUI applications grow larger, you might find many small parts of async methods all using the GUI thread as their context. This can cause sluggishness as responsiveness suffers from “thousands of paper cuts.”

To mitigate this, await the result of ConfigureAwait whenever you can. The following code snippet illustrates the default context behavior and the use of ConfigureAwait:

async Task MyMethodAsync()

{

  // Code here runs in the original context.

  await Task.Delay(1000);

  // Code here runs in the original context.

  await Task.Delay(1000).ConfigureAwait(

    continueOnCapturedContext: false);

  // Code here runs without the original

  // context (in this case, on the thread pool).

}

By using ConfigureAwait, you enable a small amount of parallelism: Some asynchronous code can run in parallel with the GUI thread instead of constantly badgering it with bits of work to do.

Aside from performance, ConfigureAwait has another important aspect: It can avoid deadlocks. Consider **Figure 3** again; if you add “ConfigureAwait(false)” to the line of code in DelayAsync, then the deadlock is avoided. This time, when the await completes, it attempts to execute the remainder of the async method within the thread pool context. The method is able to complete, which completes its returned task, and there’s no deadlock. This technique is particularly useful if you need to gradually convert an application from synchronous to asynchronous.

If you can use ConfigureAwait at some point within a method, then I recommend you use it for every await in that method after that point. Recall that the context is captured only if an incomplete Task is awaited; if the Task is already complete, then the context isn’t captured. Some tasks might complete faster than expected in different hardware and network situations, and you need to graciously handle a returned task that completes before it’s awaited. **Figure 6** shows a modified example.

**Figure 6 Handling a Returned Task that Completes Before It’s Awaited**

async Task MyMethodAsync()

{

  // Code here runs in the original context.

  await Task.FromResult(1);

  // Code here runs in the original context.

  await Task.FromResult(1).ConfigureAwait(continueOnCapturedContext: false);

  // Code here runs in the original context.

  var random = new Random();

  int delay = random.Next(2); // Delay is either 0 or 1

  await Task.Delay(delay).ConfigureAwait(continueOnCapturedContext: false);

  // Code here might or might not run in the original context.

  // The same is true when you await any Task

  // that might complete very quickly.

}

You should not use ConfigureAwait when you have code after the await in the method that needs the context. For GUI apps, this includes any code that manipulates GUI elements, writes data-bound properties or depends on a GUI-specific type such as Dispatcher/CoreDispatcher. For ASP.NET apps, this includes any code that uses HttpContext.Current or builds an ASP.NET response, including return statements in controller actions. **Figure 7**demonstrates one common pattern in GUI apps—having an async event handler disable its control at the beginning of the method, perform some awaits and then re-enable its control at the end of the handler; the event handler can’t give up its context because it needs to re-enable its control.

**Figure 7 Having an Async Event Handler Disable and Re-Enable Its Control**

private async void button1\_Click(object sender, EventArgs e)

{

  button1.Enabled = false;

  try

  {

    // Can't use ConfigureAwait here ...

    await Task.Delay(1000);

  }

  finally

  {

    // Because we need the context here.

    button1.Enabled = true;

  }

}

Each async method has its own context, so if one async method calls another async method, their contexts are independent. **Figure 8** shows a minor modification of **Figure 7**.

**Figure 8 Each Async Method Has Its Own Context**

private async Task HandleClickAsync()

{

  // Can use ConfigureAwait here.

  await Task.Delay(1000).ConfigureAwait(continueOnCapturedContext: false);

}

private async void button1\_Click(object sender, EventArgs e)

{

  button1.Enabled = false;

  try

  {

    // Can't use ConfigureAwait here.

    await HandleClickAsync();

  }

  finally

  {

    // We are back on the original context for this method.

    button1.Enabled = true;

  }

}

Context-free code is more reusable. Try to create a barrier in your code between the context-sensitive code and context-free code, and minimize the context-sensitive code. In **Figure 8**, I recommend putting all the core logic of the event handler within a testable and context-free async Task method, leaving only the minimal code in the context-sensitive event handler. Even if you’re writing an ASP.NET application, if you have a core library that’s potentially shared with desktop applications, consider using ConfigureAwait in the library code.

To summarize this third guideline, you should use Configure­Await when possible. Context-free code has better performance for GUI applications and is a useful technique for avoiding deadlocks when working with a partially async codebase. The exceptions to this guideline are methods that require the context.

## Know Your Tools

There’s a lot to learn about async and await, and it’s natural to get a little disoriented. **Figure 9** is a quick reference of solutions to common problems.

**Figure 9 Solutions to Common Async Problems**

|  |  |
| --- | --- |
| Problem | Solution |
| Create a task to execute code | Task.Run or TaskFactory.StartNew (not the Task constructor or Task.Start) |
| Create a task wrapper for an operation or event | TaskFactory.FromAsync or TaskCompletionSource<T> |
| Support cancellation | CancellationTokenSource and CancellationToken |
| Report progress | IProgress<T> and Progress<T> |
| Handle streams of data | TPL Dataflow or Reactive Extensions |
| Synchronize access to a shared resource | SemaphoreSlim |
| Asynchronously initialize a resource | AsyncLazy<T> |
| Async-ready producer/consumer structures | TPL Dataflow or AsyncCollection<T> |

The first problem is task creation. Obviously, an async method can create a task, and that’s the easiest option. If you need to run code on the thread pool, use Task.Run. If you want to create a task wrapper for an existing asynchronous operation or event, use TaskCompletionSource<T>. The next common problem is how to handle cancellation and progress reporting. The base class library (BCL) includes types specifically intended to solve these issues: CancellationTokenSource/CancellationToken and IProgress<T>/Progress<T>. Asynchronous code should use the Task-based Asynchronous Pattern, or TAP ([msdn.microsoft.com/library/hh873175](https://msdn.microsoft.com/library/hh873175)), which explains task creation, cancellation and progress reporting in detail.

Another problem that comes up is how to handle streams of asynchronous data. Tasks are great, but they can only return one object and only complete once. For asynchronous streams, you can use either TPL Dataflow or Reactive Extensions (Rx). TPL Dataflow creates a “mesh” that has an actor-like feel to it. Rx is more powerful and efficient but has a more difficult learning curve. Both TPL Dataflow and Rx have async-ready methods and work well with asynchronous code.

Just because your code is asynchronous doesn’t mean that it’s safe. Shared resources still need to be protected, and this is complicated by the fact that you can’t await from inside a lock. Here’s an example of async code that can corrupt shared state if it executes twice, even if it always runs on the same thread:

int value;

Task<int> GetNextValueAsync(int current);

async Task UpdateValueAsync()

{

  value = await GetNextValueAsync(value);

}

The problem is that the method reads the value and suspends itself at the await, and when the method resumes it assumes the value hasn’t changed. To solve this problem, the SemaphoreSlim class was augmented with the async-ready WaitAsync overloads. **Figure 10** demonstrates SemaphoreSlim.WaitAsync.

**Figure 10 SemaphoreSlim Permits Asynchronous Synchronization**

SemaphoreSlim mutex = new SemaphoreSlim(1);

int value;

Task<int> GetNextValueAsync(int current);

async Task UpdateValueAsync()

{

  await mutex.WaitAsync().ConfigureAwait(false);

  try

  {

    value = await GetNextValueAsync(value);

  }

  finally

  {

    mutex.Release();

  }

}

Asynchronous code is often used to initialize a resource that’s then cached and shared. There isn’t a built-in type for this, but Stephen Toub developed an AsyncLazy<T> that acts like a merge of Task<T> and Lazy<T>. The original type is described on his blog ([bit.ly/dEN178](http://bit.ly/dEN178)), and an updated version is available in my AsyncEx library ([nitoasyncex.codeplex.com](http://nitoasyncex.codeplex.com/)).

Finally, some async-ready data structures are sometimes needed. TPL Dataflow provides a BufferBlock<T> that acts like an async-ready producer/consumer queue. Alternatively, AsyncEx provides AsyncCollection<T>, which is an async version of BlockingCollection<T>.

I hope the guidelines and pointers in this article have been helpful. Async is a truly awesome language feature, and now is a great time to start using it!

**Stephen Cleary** is a husband, father and programmer living in northern Michigan. He has worked with multithreading and asynchronous programming for 16 years and has used async support in the Microsoft .NET Framework since the first CTP. His home page, including his blog, is at[*stephencleary.com*](http://stephencleary.com/).

Thanks to the following technical expert for reviewing this article: Stephen Toub  
Stephen Toub works on the Visual Studio team at Microsoft. He specializes in areas related to parallelism and asynchrony.

<https://msdn.microsoft.com/en-us/magazine/jj991977.aspx>