Asynchronous programming with async and await

03/18/2019 • 10 minutes to read • 👤👤👤👤👤 +11

In this article

The Task asynchronous programming model (TAP) provides an abstraction over asynchronous code. You write code as a sequence of statements, just like always. You can read that code as though each statement completes before the next begins. The compiler performs a number of transformations because some of those statements may start work and return a [Task](#) that represents the ongoing work.

That's the goal of this syntax: enable code that reads like a sequence of statements, but executes in a much more complicated order based on external resource allocation and when tasks complete. It's analogous to how people give instructions for processes that include asynchronous tasks. Throughout this article, you'll use an example of instructions for making a breakfast to see how the `async` and `await` keywords make it easier to reason about code that includes a series of asynchronous instructions. You'd write the instructions something like the following list to explain how to make a breakfast:

1. Pour a cup of coffee.
2. Heat up a pan, then fry two eggs.
3. Fry three slices of bacon.
4. Toast two pieces of bread.
5. Add butter and jam to the toast.
6. Pour a glass of orange juice.

If you have experience with cooking, you'd execute those instructions **asynchronously**. You'd start warming the pan for eggs, then start the bacon. You'd put the bread in the toaster, then start the eggs. At each step of the process, you'd start a task, then turn your attention to tasks that are ready for your attention.

Cooking breakfast is a good example of asynchronous work that isn't parallel. One person (or thread) can handle all these tasks. Continuing the breakfast analogy, one person can make breakfast asynchronously by starting the next task before the first completes. The cooking progresses whether or not someone is watching it. As soon as you start warming the pan for the eggs, you can begin frying the bacon. Once the bacon starts, you can put the bread into the toaster.

For a parallel algorithm, you'd need multiple cooks (or threads). One would make the eggs, one the bacon, and so on. Each one would be focused on just that one task. Each cook (or thread) would be blocked synchronously waiting for bacon to be ready to flip, or the toast to pop.

Now, consider those same instructions written as C# statements:

C#   Copy

```csharp
static void Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    Egg eggs = FryEggs(2);
    Console.WriteLine("eggs are ready");
    Bacon bacon = FryBacon(3);
    Console.WriteLine("bacon is ready");
    Toast toast = ToastBread(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");

    Console.WriteLine("Breakfast is ready!");
}
```

Computers don't interpret those instructions the same way people do. The computer will block on each statement until the work is complete before moving on to the next statement. That creates an unsatisfying breakfast. The later tasks wouldn't be started until the earlier tasks had completed. It would take much longer to create the breakfast, and some items would have gotten cold before being served.

If you want the computer to execute the above instructions asynchronously, you must write asynchronous code.

These concerns are important for the programs you write today. When you write client programs, you want the UI to be responsive to user input. Your application shouldn't make a phone appear frozen while it's downloading data from the web. When you write server programs, you don't want threads blocked. Those threads could be serving other requests. Using synchronous code when asynchronous alternatives exist hurts your ability to scale out less expensively. You pay for those blocked threads.

Successful modern applications require asynchronous code. Without language support, writing asynchronous code required callbacks, completion events, or other means that obscured the original intent of the code. The advantage of the synchronous code is that it's easy to understand. The step-by-step actions make it easy to scan and understand. Traditional asynchronous models forced you to focus on the asynchronous nature of the code, not on the fundamental actions of the code.

Don't block, await instead

The preceding code demonstrates a bad practice: constructing synchronous code to perform asynchronous operations. As written, this code blocks the thread executing it from doing any other work. It won't be interrupted while any of the tasks are in progress. It would be as though you stared at the toaster after putting the bread in. You'd ignore anyone talking to you until the toast popped.

Let's start by updating this code so that the thread doesn't block while tasks are running. The await keyword provides a non-blocking way to start a task, then continue execution when that task completes. A simple asynchronous version of the make a breakfast code would look like the following snippet:

C#	⧉ Copy

```csharp
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    Egg eggs = await FryEggs(2);
    Console.WriteLine("eggs are ready");
    Bacon bacon = await FryBacon(3);
    Console.WriteLine("bacon is ready");
    Toast toast = await ToastBread(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
```

```
        Console.WriteLine("Breakfast is ready!");
}
```

This code doesn't block while the eggs or the bacon are cooking. This code won't start any other tasks though. You'd still put the toast in the toaster and stare at it until it pops. But at least, you'd respond to anyone that wanted your attention. In a restaurant where multiple orders are placed, the cook could start another breakfast while the first is cooking.

Now, the thread working on the breakfast isn't blocked while awaiting any started task that hasn't yet finished. For some applications, this change is all that's needed. A GUI application still responds to the user with just this change. However, for this scenario, you want more. You don't want each of the component tasks to be executed sequentially. It's better to start each of the component tasks before awaiting the previous task's completion.

Start tasks concurrently

In many scenarios, you want to start several independent tasks immediately. Then, as each task finishes, you can continue other work that's ready. In the breakfast analogy, that's how you get breakfast done more quickly. You also get everything done close to the same time. You'll get a hot breakfast.

The System.Threading.Tasks.Task and related types are classes you can use to reason about tasks that are in progress. That enables you to write code that more closely resembles the way you'd actually create breakfast. You'd start cooking the eggs, bacon, and toast at the same time. As each requires action, you'd turn your attention to that task, take care of the next action, then await for something else that requires your attention.

You start a task and hold on to the Task object that represents the work. You'll `await` each task before working with its result.

Let's make these changes to the breakfast code. The first step is to store the tasks for operations when they start, rather than awaiting them:

```C#                                                          Copy
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");
Task<Egg> eggsTask = FryEggs(2);
Egg eggs = await eggsTask;
Console.WriteLine("eggs are ready");
Task<Bacon> baconTask = FryBacon(3);
```

```csharp
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");
Task<Toast> toastTask = ToastBread(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");
```

Next, you can move the `await` statements for the bacon and eggs to the end of the method, before serving breakfast:

C#   ⧉ Copy

```csharp
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");
Task<Egg> eggsTask = FryEggs(2);
Task<Bacon> baconTask = FryBacon(3);
Task<Toast> toastTask = ToastBread(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("oj is ready");

Egg eggs = await eggsTask;
Console.WriteLine("eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");

Console.WriteLine("Breakfast is ready!");
```

The preceding code works better. You start all the asynchronous tasks at once. You await each task only when you need the results. The preceding code may be similar to code in a web application that makes requests of different microservices, then combines the results into a single page. You'll make all the requests immediately, then `await` all those tasks and compose the web page.

Composition with tasks

You have everything ready for breakfast at the same time except the toast. Making the toast is the composition of an asynchronous operation (toasting the bread), and synchronous operations (adding the butter and the jam). Updating this code illustrates an important concept:

> ⓘ **Important**
>
> The composition of an asynchronous operation followed by synchronous work is an asynchronous operation. Stated another way, if any portion of an operation is asynchronous, the entire operation is asynchronous.

The preceding code showed you that you can use [Task](#) or [Task<TResult>](#) objects to hold running tasks. You `await` each task before using its result. The next step is to create methods that represent the combination of other work. Before serving breakfast, you want to await the task that represents toasting the bread before adding butter and jam. You can represent that work with the following code:

C#   📋 Copy

```csharp
async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);
    return toast;
}
```

The preceding method has the `async` modifier in its signature. That signals to the compiler that this method contains an `await` statement; it contains asynchronous operations. This method represents the task that toasts the bread, then adds butter and jam. This method returns a [Task<TResult>](#) that represents the composition of those three operations. The main block of code now becomes:

C#   📋 Copy

```csharp
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);
```

```csharp
        var eggs = await eggsTask;
        Console.WriteLine("eggs are ready");
        var bacon = await baconTask;
        Console.WriteLine("bacon is ready");
        var toast = await toastTask;
        Console.WriteLine("toast is ready");
        Juice oj = PourOJ();
        Console.WriteLine("oj is ready");

        Console.WriteLine("Breakfast is ready!");

        async Task<Toast> MakeToastWithButterAndJamAsync(int number)
        {
            var toast = await ToastBreadAsync(number);
            ApplyButter(toast);
            ApplyJam(toast);
            return toast;
        }
    }
```

The previous change illustrated an important technique for working with asynchronous code. You compose tasks by separating the operations into a new method that returns a task. You can choose when to await that task. You can start other tasks concurrently.

Await tasks efficiently

The series of `await` statements at the end of the preceding code can be improved by using methods of the `Task` class. One of those APIs is [WhenAll](#), which returns a [Task](#) that completes when all the tasks in its argument list have completed, as shown in the following code:

C#	Copy

```csharp
await Task.WhenAll(eggsTask, baconTask, toastTask);
Console.WriteLine("eggs are ready");
Console.WriteLine("bacon is ready");
Console.WriteLine("toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Another option is to use [WhenAny,](#) which returns a `Task<Task>` that completes when any of its arguments completes. You can await the returned task, knowing that it has already finished. The following code shows how you could use [WhenAny](#) to await the first task to

finish and then process its result. After processing the result from the completed task, you
remove that completed task from the list of tasks passed to `WhenAny`.

C#   Copy

```csharp
var allTasks = new List<Task>{eggsTask, baconTask, toastTask};
while (allTasks.Any())
{
    Task finished = await Task.WhenAny(allTasks);
    if (finished == eggsTask)
    {
        Console.WriteLine("eggs are ready");
    }
    else if (finished == baconTask)
    {
        Console.WriteLine("bacon is ready");
    }
    else if (finished == toastTask)
    {
        Console.WriteLine("toast is ready");
    }
    allTasks.Remove(finished);
}
Juice oj = PourOJ();
Console.WriteLine("oj is ready");
Console.WriteLine("Breakfast is ready!");
```

After all those changes, the final version of `Main` looks like the following code:

C#   Copy

```csharp
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var allTasks = new List<Task>{eggsTask, baconTask, toastTask};
    while (allTasks.Any())
    {
        Task finished = await Task.WhenAny(allTasks);
        if (finished == eggsTask)
        {
            Console.WriteLine("eggs are ready");
        }
        else if (finished == baconTask)
```

```csharp
        {
            Console.WriteLine("bacon is ready");
        }
        else if (finished == toastTask)
        {
            Console.WriteLine("toast is ready");
        }
        allTasks.Remove(finished);
    }
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");

    async Task<Toast> MakeToastWithButterAndJamAsync(int number)
    {
        var toast = await ToastBreadAsync(number);
        ApplyButter(toast);
        ApplyJam(toast);
        return toast;
    }
}
```

This final code is asynchronous. It more accurately reflects how a person would cook a breakfast. Compare the preceding code with the first code sample in this article. The core actions are still clear from reading the code. You can read this code the same way you'd read those instructions for making a breakfast at the beginning of this article. The language features for `async` and `await` provide the translation every person makes to follow those written instructions: start tasks as you can and don't block waiting for tasks to complete.

Is this page helpful?

👍 Yes   👎 No