Asynchronous programming with async and await

Article • 05/19/2022 • 15 minutes to read • 23 contributors

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 many 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 are 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 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 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 task 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 the bacon to be ready to flip, or the toast to pop.

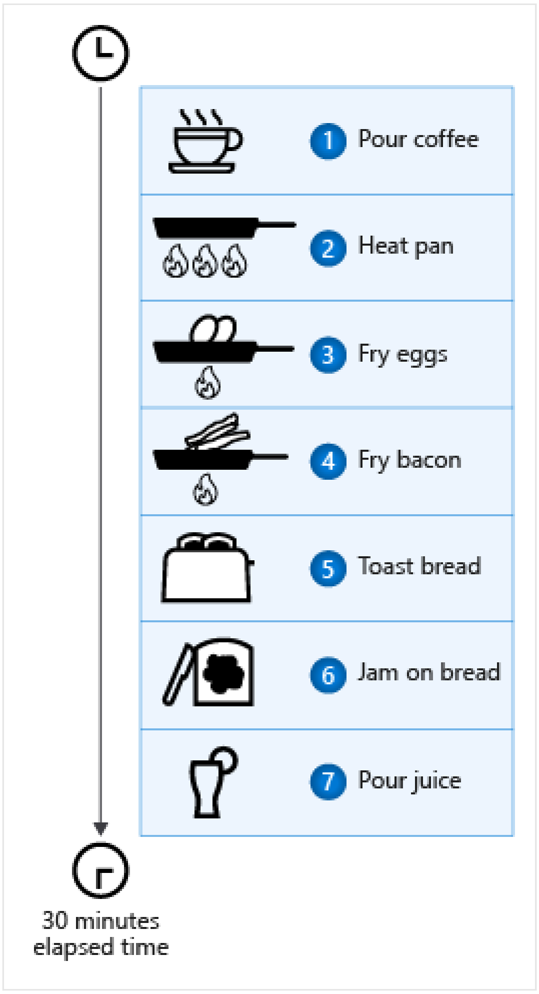
Now, consider those same instructions written as C# statements:

C#

Copy

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

namespace AsyncBreakfast
{
    class Program
    {
        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!");
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");

        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");

        private static Toast ToastBread(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the
toaster");
            }
            Console.WriteLine("Start toasting...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

        private static Bacon FryBacon(int slices)
        {
            Console.WriteLine($"putting {slices} slices of bacon in the
pan");
            Console.WriteLine("cooking first side of bacon...");
            Task.Delay(3000).Wait();
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("flipping a slice of bacon");
            }
            Console.WriteLine("cooking the second side of bacon...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put bacon on plate");

            return new Bacon();
        }

        private static Egg FryEggs(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            Task.Delay(3000).Wait();
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
```

```
                return new Coffee();
            }
        }
    }
```



The synchronously prepared breakfast took roughly 30 minutes because the total is the sum of each task.

> ⓘ **Note**
>
> The `Coffee`, `Egg`, `Bacon`, `Toast`, and `Juice` classes are empty. They are simply marker classes for the purpose of demonstration, contain no properties, and serve no other purpose.

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 been 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 its 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 FryEggsAsync(2);
    Console.WriteLine("eggs are ready");

    Bacon bacon = await FryBaconAsync(3);
    Console.WriteLine("bacon is ready");

    Toast toast = await ToastBreadAsync(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

ⓘ **Important**

The total elapsed time is roughly the same as the initial synchronous version. The code has yet to take advantage of some of the key features of asynchronous programming.

Tip

The method bodies of the `FryEggsAsync`, `FryBaconAsync`, and `ToastBreadAsync` have all been updated to return `Task<Egg>`, `Task<Bacon>`, and `Task<Toast>` respectively. The methods are renamed from their original version to include the "Async" suffix. Their implementations are shown as part of the final version later in this article.

Note

The `Main` method returns `Task`, despite not having a `return` expression—this is by design. For more information, see Evaluation of a void-returning async function.

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 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 wait 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:

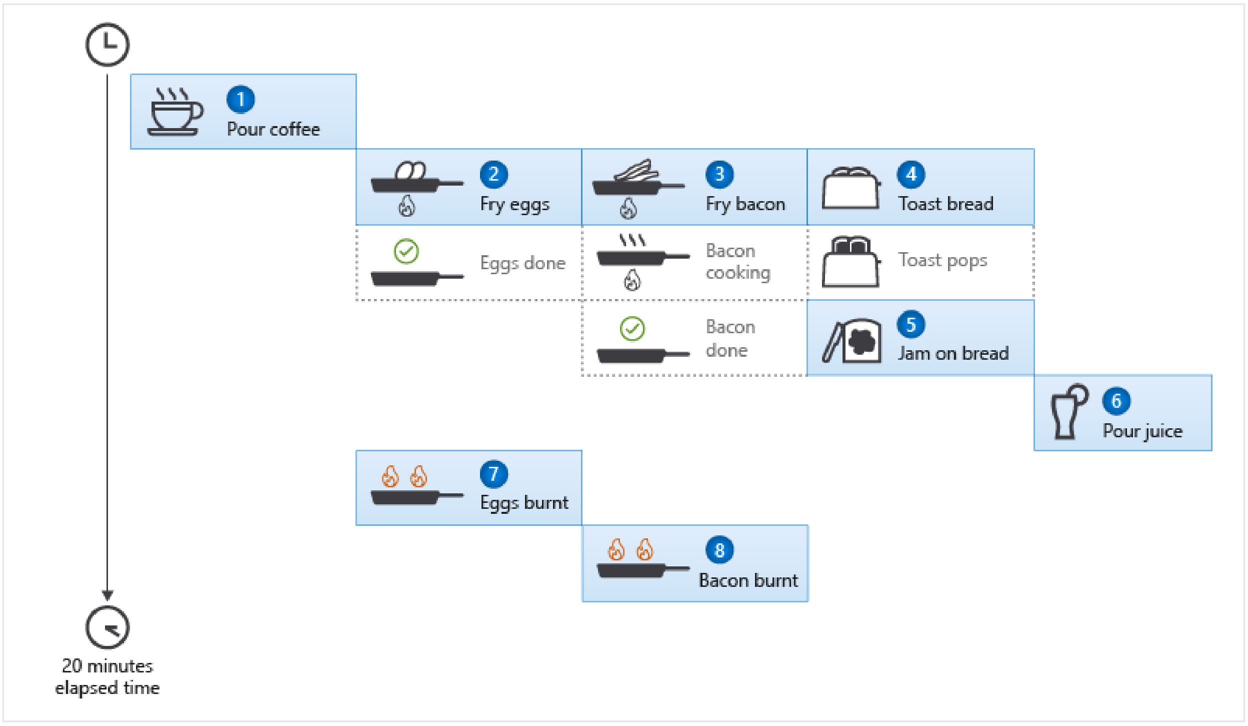
C# Copy

```csharp
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");

Task<Bacon> baconTask = FryBaconAsync(3);
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");

Task<Toast> toastTask = ToastBreadAsync(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

```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Task<Bacon> baconTask = FryBaconAsync(3);
Task<Toast> toastTask = ToastBreadAsync(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 asynchronously prepared breakfast took roughly 20 minutes, this time savings is because some tasks ran concurrently.

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 to 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
static 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);

    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!");
}
```

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.

Asynchronous exceptions

Up to this point, you've implicitly assumed that all these tasks complete successfully. Asynchronous methods throw exceptions, just like their synchronous counterparts. Asynchronous support for exceptions and error handling strives for the same goals as asynchronous support in general: You should write code that reads like a series of synchronous statements. Tasks throw exceptions when they can't complete successfully. The client code can catch those exceptions when a started task is `awaited`. For example, let's assume that the toaster catches fire while making the toast. You can simulate that by modifying the `ToastBreadAsync` method to match the following code:

C# Copy

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire");
    await Task.Delay(1000);
    Console.WriteLine("Remove toast from toaster");
    return new Toast();
}
```

> **Note**
>
> You'll get a warning when you compile the preceding code regarding unreachable code. That's intentional, because once the toaster catches fire, operations won't proceed normally.

Run the application after making these changes, and you'll output similar to the following text:

Console

```
Pouring coffee
Coffee is ready
Warming the egg pan...
putting 3 slices of bacon in the pan
Cooking first side of bacon...
Putting a slice of bread in the toaster
Putting a slice of bread in the toaster
Start toasting...
Fire! Toast is ruined!
Flipping a slice of bacon
Flipping a slice of bacon
Flipping a slice of bacon
Cooking the second side of bacon...
Cracking 2 eggs
Cooking the eggs ...
Put bacon on plate
Put eggs on plate
Eggs are ready
Bacon is ready
Unhandled exception. System.InvalidOperationException: The toaster is on
fire
   at AsyncBreakfast.Program.ToastBreadAsync(Int32 slices) in
Program.cs:line 65
   at AsyncBreakfast.Program.MakeToastWithButterAndJamAsync(Int32 number) in
Program.cs:line 36
   at AsyncBreakfast.Program.Main(String[] args) in Program.cs:line 24
   at AsyncBreakfast.Program.<Main>(String[] args)
```

You'll notice quite a few tasks are completed between when the toaster catches fire and the exception is observed. When a task that runs asynchronously throws an exception, that Task is ***faulted***. The Task object holds the exception thrown in the Task.Exception property. Faulted tasks throw an exception when they're awaited.

There are two important mechanisms to understand: how an exception is stored in a faulted task, and how an exception is unpackaged and rethrown when code awaits a faulted task.

When code running asynchronously throws an exception, that exception is stored in the `Task`. The Task.Exception property is an System.AggregateException because more than one exception may be thrown during asynchronous work. Any exception thrown is added to the AggregateException.InnerExceptions collection. If that `Exception` property is null, a new `AggregateException` is created and the thrown exception is the first item in the collection.

The most common scenario for a faulted task is that the `Exception` property contains exactly one exception. When code `awaits` a faulted task, the first exception in the AggregateException.InnerExceptions collection is rethrown. That's why the output from this example shows an `InvalidOperationException` instead of an `AggregateException`. Extracting the first inner exception makes working with asynchronous methods as similar as possible to working with their synchronous counterparts. You can examine the `Exception` property in your code when your scenario may generate multiple exceptions.

Before going on, comment out these two lines in your `ToastBreadAsync` method. You don't want to start another fire:

C#

Copy

```csharp
Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

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 complete. 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`.

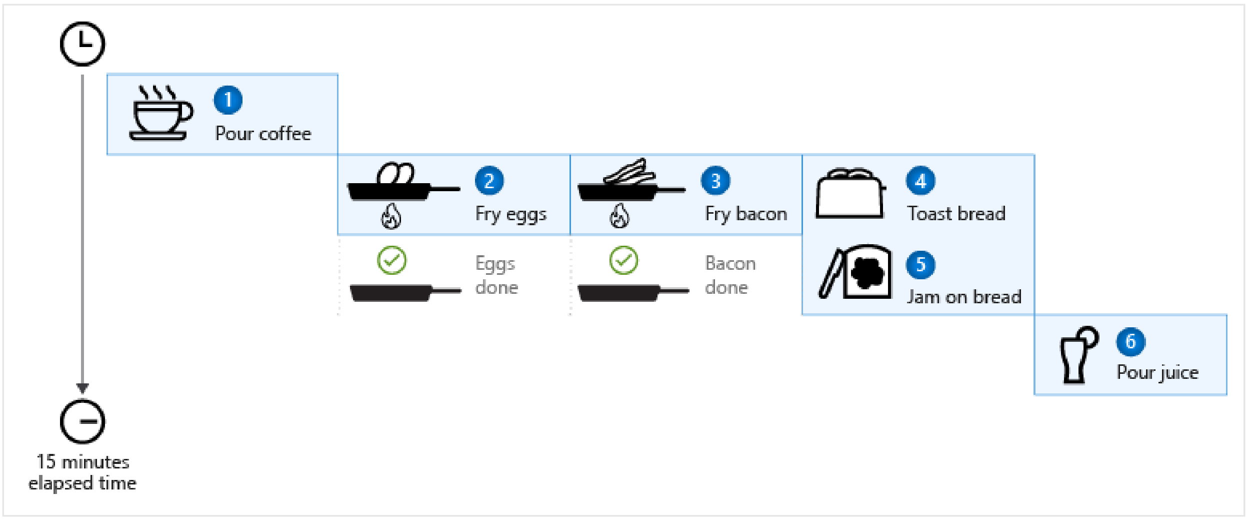
C# Copy

```csharp
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("Eggs are ready");
    }
    else if (finishedTask == baconTask)
    {
        Console.WriteLine("Bacon is ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("Toast is ready");
    }
    breakfastTasks.Remove(finishedTask);
}
```

After all those changes, the final version of the code looks like this:

C# Copy

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

namespace AsyncBreakfast
{
    class Program
    {
        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 breakfastTasks = new List<Task> { eggsTask, baconTask,
toastTask };
            while (breakfastTasks.Count > 0)
            {
                Task finishedTask = await Task.WhenAny(breakfastTasks);
                if (finishedTask == eggsTask)
                {
                    Console.WriteLine("eggs are ready");
                }
                else if (finishedTask == baconTask)
                {
                    Console.WriteLine("bacon is ready");
                }
                else if (finishedTask == toastTask)
                {
                    Console.WriteLine("toast is ready");
                }
                breakfastTasks.Remove(finishedTask);
            }

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

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

            return toast;
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");

        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");
        private static async Task<Toast> ToastBreadAsync(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the 
toaster");
            }
            Console.WriteLine("Start toasting...");
            await Task.Delay(3000);
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

        private static async Task<Bacon> FryBaconAsync(int slices)
        {
            Console.WriteLine($"putting {slices} slices of bacon in the 
pan");
            Console.WriteLine("cooking first side of bacon...");
            await Task.Delay(3000);
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("flipping a slice of bacon");
            }
            Console.WriteLine("cooking the second side of bacon...");
            await Task.Delay(3000);
            Console.WriteLine("Put bacon on plate");

            return new Bacon();
        }

        private static async Task<Egg> FryEggsAsync(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            await Task.Delay(3000);
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            await Task.Delay(3000);
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```



The final version of the asynchronously prepared breakfast took roughly 15 minutes because some tasks ran concurrently, and the code monitored multiple tasks at once and only took action when it was needed.

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.

Next steps

Explore real world scenarios for asynchronous programs