

Asynchronous Programming in C#

Jeremy Clark
jeremybytes.com
github.com/jeremybytes

Schedule

- Class Hours 8:45 a.m. – 4:30 p.m.
- Breaks throughout the day
- Lunch 12:15 p.m. – 1:15 p.m.

Q&A throughout the day

All Times are Central Daylight Time

Materials

[https://github.com/jeremybytes/
async-workshop-2025](https://github.com/jeremybytes/async-workshop-2025)

Agenda

- Running Async Code
 - Task and await
 - Desktop and Web
- Where Continuations Run
 - TaskScheduler
 - ConfigureAwait
- Useful Properties & Methods
 - Continuation parameters
 - .IsFaulted
 - Exception Handling
 - Cancellation
- Writing Asynch Methods
 - Return or await
 - async void

Optional Topics (based on interest)

- Parallel Programming
 - Tasks
 - Channels
 - Parallel.ForEachAsync
- Progress Reporting
 - IProgress<T>
- Breaking Async
 - .Result
 - .Wait()
 - JoinableTaskFactory
 - Constructors

Topics (in no particular order)

- .NET Framework vs .NET 8/9
- TaskContinuationOptions
- CancellationToken.None
- ThrowIfCancellationRequested
- Action
- Lambda expressions
- Task.Run()
- Context vs. no context
- Task.Result
- AggregateException
- OperationCanceledException
- CancellationTokenSource
- async MVC Controllers
- .GetAwaiter().GetResult()

Running Asynchronous Code

Async Revolves Around Task

Use Task directly

- Extremely powerful
- Lots of options
- Very complex

await Task

- Looks like other code
- Limited functionality
- Fits the “95% scenario”

Task<T>

- Method Returns a Task
 - Task<T> GetDataAsync()
 - “T” = type for the result
- Task
 - Represents a concurrent operation
 - May or may not operate on a separate thread
 - Can be chained and combined

Desktop App Sample: Using Task

```
Task<List<Person>> peopleTask = reader.GetPeopleAsync();
peopleTask.ContinueWith(
    task =>
    {
        List<Person> people = task.Result;
        foreach (var person in people)
            PersonListBox.Items.Add(person);
    },
    TaskScheduler.FromCurrentSynchronizationContext());
```

Web App Sample: Using Task

```
Task<List<Person>> peopleTask = reader.GetPeopleAsync();
Task<ViewResult> result = peopleTask.ContinueWith(
    task =>
    {
        List<Person> people = task.Result;
        return View("Index", people);
    });
return result;
```

async & await

- Syntactic Wrapper Around Task
 - “await” pauses the current method until Task is complete.
 - Looks like a blocking operation
 - Does not block current thread
- “async” modifier
 - Tells the compiler to treat “await” as noted above
 - The compiled method is turned into a state machine

Desktop App Sample: Using await

```
List<Person> people = await reader.GetPeopleAsync();  
foreach (var person in people)  
{  
    PersonListBox.Items.Add(person);  
}
```

Web App Sample: Using await

```
List<Person> people = await reader.GetPeopleAsync();  
return View("Index", people);
```



Return Value with await

Whenever you “await” a Task in a method, the return value is wrapped in a Task.

No 'await'

```
public Person GetPerson(int id)
{
    List<Person> people = People.GetPeopleNoAsync();
    Person selectedPerson = people.Single(p => p.Id == id);
    return selectedPerson; // Person
}
```

Return types match

With 'await'

```
public async Task<Person> GetPersonAsync(int id)
{
    List<Person> people = await GetPeopleAsync();
    Person selectedPerson = people.Single(p => p.Id == id);
    return selectedPerson; // Person
}
```

Return types do not match

With Task

```
public Task<Person> GetPersonAsync(int id)
{
    Task<List<Person>> peopleTask = GetPeopleAsync();
    Task<Person> result = peopleTask.ContinueWith(task =>
    {
        List<Person> people = task.Result;
        Person selectedPerson = people.Single(p => p.Id == id);
        return selectedPerson; // Person
    });
    return result; // Task<Person>
}
```

Return types match


Return Value with await

- Whenever you “await” a Task in a method, the return value is wrapped in a Task.
 - This makes more sense if we think of “await” as wrapping and hiding a Task.
 - We do not need to deal with the complexity of the continuation task directly, but the Task is still there.

Task.Result

.Result

- Should only be used inside a continuation.
- If ".Result" is used outside of a continuation, then the operation will block (and possibly deadlock).
- If ".Result" is accessed on a faulted task, it will raise an `AggregateException`.



`.GetAwaiter().GetResult()`

`.GetAwaiter().GetResult()`

- It is sometimes used because it returns an `Exception` (not an `AggregateException`).
- Blocking effects are the same as with `.Result`.

`.GetAwaiter().GetResult()`

DANGER

`.GetAwaiter().GetResult()` was designed for internal use only.

Remarks

This method is intended for compiler use rather than use directly in code.

<https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.getawaiter>



Task.Result

Advice

Avoid using .Result or
.GetAwaiter().GetResult()
to break asynchrony.

Where Continuations Run

Default Task Behavior

- By default, a Task continuation does **not** run on the current context (thread).
- This means if you need to access resources from the current context (thread), you cannot do it by default.

Note: "Context" and "thread" are not technically equivalent. There are some async operations that do not use thread resources. But for most situations, we can think of these as interchangeable.

Default Task Behavior

Runs in Main Context

Runs somewhere else

```
Task<List<Person>> peopleTask = reader.GetPeopleAsync();
```

```
peopleTask.ContinueWith(
```

```
task =>
```

Runs somewhere else

```
{  
    List<Person> people = task.Result;  
    foreach (var person in people)  
        PersonListBox.Items.Add(person);  
},
```

```
);
```

Task Scheduler

- `TaskScheduler.FromCurrentSynchronizationContext` will return to the prior context.
- For web applications, this means going back to the request thread.
- This may be needed for WebForms or applications that require Session or similar information.

Task Continuation in Main Context

Runs in Main Context

Runs somewhere else

```
Task<List<Person>> peopleTask = reader.GetPeopleAsync();
```

```
peopleTask.ContinueWith(
```

task => **Runs in Main Context**

```
{  
    List<Person> people = task.Result;  
    foreach (var person in people)  
        PersonListBox.Items.Add(person);  
},
```

```
TaskScheduler.FromCurrentSynchronizationContext());
```

Default await Behavior

- By default, code after "await" *does* run on the current context (thread).
- This means that you can safely access resources from that context (thread) – such as UI elements (desktop/mobile) or Session information (web).

Default await Behavior

Runs in Main Context

```
ClearListBox();
```

Runs somewhere else

```
List<Person> people = await reader.GetPeopleAsync();
```

```
foreach (var person in people)
{
    PersonListBox.Items.Add(person);
}
```

Runs in Main Context

ConfigureAwait

Code running after "await" returns to the current context.

This is fine for most situations, but using `ConfigureAwait(false)` can optimize performance.

Note: ASP.NET Core does not have a synchronization context. So `ConfigureAwait` has no effect.

ConfigureAwait

ConfigureAwait determines whether processing needs to go back to the current context (thread) after awaiting an operation.

ConfigureAwait

- `ConfigureAwait(true)` returns to the current context.
 - This is the default
- `ConfigureAwait(false)` uses whatever context is readily available.

`ConfigureAwait(false)` is preferred for optimization purposes.

await with ConfigureAwait(false)

Runs in Main Context

```
ClearListBox();
```

```
List<Person> people = await
```

Runs somewhere else

Runs somewhere else

```
reader.GetPeopleAsync()
```

```
.ConfigureAwait(false);
```

```
foreach (var person in people)
```

```
{
```

```
    PersonListBox.Items.Add(person);
```

```
}
```

Runs somewhere else

Importance of Asynchronous Code

Reminder:

Web servers have a limited number of threads to handle incoming requests.

Getting off of these threads (with async code) frees them up to take additional requests.

ConfigureAwait

General Guideline:

- `ConfigureAwait(false)` for library code
- `ConfigureAwait(true)` for UI code

Exception:

- ASP.NET Core applications do **not** have a current context, so this setting will be ignored.
- .NET Framework ASP.NET applications **do** have a context. You may need to go back to the prior context if you need Session or similar information.

ConfigureAwait

More Options in .NET 8 and .NET 9:

- `ConfigureAwait(ConfigureAwaitOptions)`
 - Lets you fine tune behavior
 - Generally for more complex situations

- Check Stephen Cleary's article for details:

<https://blog.stephencleary.com/2023/11/configureawait-in-net-8.html>



Lab

Lab – Recommended Practices and Continuations

[https://github.com/jeremybytes/
async-workshop-2025](https://github.com/jeremybytes/async-workshop-2025)



Useful Properties and Methods

.ContinueWith() Parameters

- Action<Task>
 - A delegate to run when the task is complete.
- TaskScheduler
 - TaskSchedule.FromCurrentSynchronizationContext will return to the prior context (e.g. to run the continuation on the UI thread).

.ContinueWith() Parameters

- CancellationToken
 - A canceled token prevents the continuation running.
 - CancellationToken.None can be used as a placeholder.
- TaskContinuationOptions
 - OnlyOn... and NotOn... values set conditions on whether the continuation will run.

Task Properties (.NET 8 / 9)

- Task Properties

- IsFaulted
- IsCanceled
- IsCompleted*
- IsCompletedSuccessfully

IsCompletedSuccessfully

- .NET 8 / 9
- .NET Standard 2.1
- NOT .NET Standard 2.0
- NOT .NET Framework

**Note: Means “no longer running”
not “completed successfully”*

Task Properties (.NET Framework)

- Task Properties

- IsFaulted
- IsCanceled
- IsCompleted*
- **Status**

**Note: Means “no longer running” not “completed successfully”*

- TaskStatus

- **Canceled**
- Created
- **Faulted**
- **RanToCompletion**
- Running
- WaitingForActivation
- WaitingForChildrenToComplete
- WaitingToRun

async void

- async void
- Only for true "fire and forget"
- Disadvantages
 - Cannot tell when (or if) the operation completes
 - Cannot tell whether the operation was successful
 - Cannot see exceptions that occur
- Reminder: Exceptions stay on their own thread unless we go looking for them. Using "await" with a Task is one way to show them.

async void

Advice

Avoid writing
async void method.

- Exception: Event handlers must return void. Be sure proper error handling is in place inside the event handler (or in a global exception handler)

Exception Handling

- `AggregateException`
 - Tree structure of exceptions
- `Flatten()`
 - Flattens the tree structure to a single level of `InnerExceptions`

Cancellation

- CancellationToken is ReadOnly when created directly
 - new CancellationToken(true)
 - new CancellationToken(false)
- CancellationTokenSource
 - IDisposable → "using" or call "Dispose"
 - cts.Token → CancellationToken
 - cts.Cancel() → Sets "IsCancellationRequested" to true

Cancellation

- `ThrowIfCancellationRequested`
 - Sets Task Status property
 - Sets `IsCompleted`, `IsCanceled`, etc. properties
 - Throws `OperationCanceledException` (needed for "await")

Web Cancellation

- MVC Controller Actions, Controller APIs and Minimal APIs can have a CancellationToken parameter.
- If the request is canceled (for example, by clicking the “Stop” button in a browser), the token is put into a cancellation requested state.

Web Cancellation

- The cancellation token can be used in the Action method or API.
- Reminder: Awaiting a canceled task throws an `OperationCanceledException`, so your Action/API code should be prepared to handle that.

Cancellation and Continuations

- ContinueWith CancellationToken parameter
 - When "IsCancellationRequested" is true, the continuation **will not run**.
 - An option is to use "CancellationToken.None" as a dummy token.

Cancellation and Continuations

- If canceled, continuation will not run

```
personTask.ContinueWith(task =>
{
    if (task.IsFaulted) ...
    if (task.IsCanceled) ...
    if (task.IsCompletedSuccessfully) ...
}, tokenSource.Token);
```


Cancellation and Continuations

Advice

Be careful when passing a
cancellation token to a
continuation as a parameter.



Lab

Lab – Adding Async to an Existing Application

[https://github.com/jeremybytes/
async-workshop-2025](https://github.com/jeremybytes/async-workshop-2025)

Writing Asynchronous Methods

Writing Asynchronous Methods

- Directly return a Task
- Sample:

```
public Task<Person> GetPersonAsync(int id)
{
    Task<Person> personTask = Task.Run(() => GetPerson(id));
    return personTask;
}
```

Writing Asynchronous Methods

- If you "await" something in your method, then the return value is automatically wrapped in a Task.
- Sample:

```
public async Task<Person> GetPersonAsync(int id)
{
    Person person = await Task.Run(() => GetPerson(id));
    return person;
}
```



await or Not?

ADVICE

Prefer async/await
over directly returning Task

David Fowler's Async Guide

<https://github.com/davidfowl/AspNetCoreDiagnosticScenarios/blob/master/AsyncGuidance.md#prefer-asyncawait-over-directly-returning-task>

await and IDisposable

- It is important to await Tasks with IDisposable elements.
- If a Task is not awaited, an object may be disposed before the Task has a chance to complete.

await and IDisposable

- Sample (bad)

```
public Task<IReadOnlyCollection<Person>> GetPeople()  
{  
    using SqlDataReader reader = new(sqlFileName);  
    return reader.GetPeople();  
}
```

- The reader is disposed when this method exits (which may be before the reader.GetPeople task is complete).

await and IDisposable

- Sample (good)

```
public async Task<IReadOnlyCollection<Person>> GetPeople()  
{  
    using SQLReader reader = new(sqlFileName);  
    return await reader.GetPeople();  
}
```

- `await` pauses the operation, so the reader object will not be disposed until after the `reader.GetPeople` task is complete.

async void

- async void
- Only for true "fire and forget"
- Disadvantages
 - Cannot tell when (or if) the operation completes
 - Cannot tell whether the operation was successful
 - Cannot see exceptions that occur
- Reminder: Exceptions stay on their own thread unless we go looking for them. Using "await" with a Task is one way to show them.

async void and ASP.NET Core

“The use of async void in ASP.NET Core applications is **ALWAYS** bad.

Avoid it, never do it... Async void methods will crash the process if an exception is thrown.”

David Fowler's Async Guide

<https://github.com/davidfowl/AspNetCoreDiagnosticScenarios/blob/master/AsyncGuidance.md#async-void>

async void

Advice

Avoid writing
async void method.

- Exception: Event handlers must return void. Be sure proper error handling is in place inside the event handler (or in a global exception handler)

ConfigureAwait

General Guideline:

- `ConfigureAwait(false)` for library code
- `ConfigureAwait(true)` for UI code

Exception:

- ASP.NET Core applications do **not** have a current context, so this setting will be ignored.
- .NET Framework ASP.NET applications **do** have a context. You may need to go back to the prior context if you need Session or similar information.

Parallel Programming

Sequential Programming

- Multiple "await"s run in sequence (one at a time)
- Sample: multiple service calls

```
await CallService1Async();  
await CallService2Async();  
await CallService3Async();
```

CallService2Async will not run until after CallService1 Async is complete.
CallService3Async will not run until after CallService2Async is complete.

Parallel with Task

- Multiple Tasks can run in parallel (at the same time)
- Ex: multiple service calls

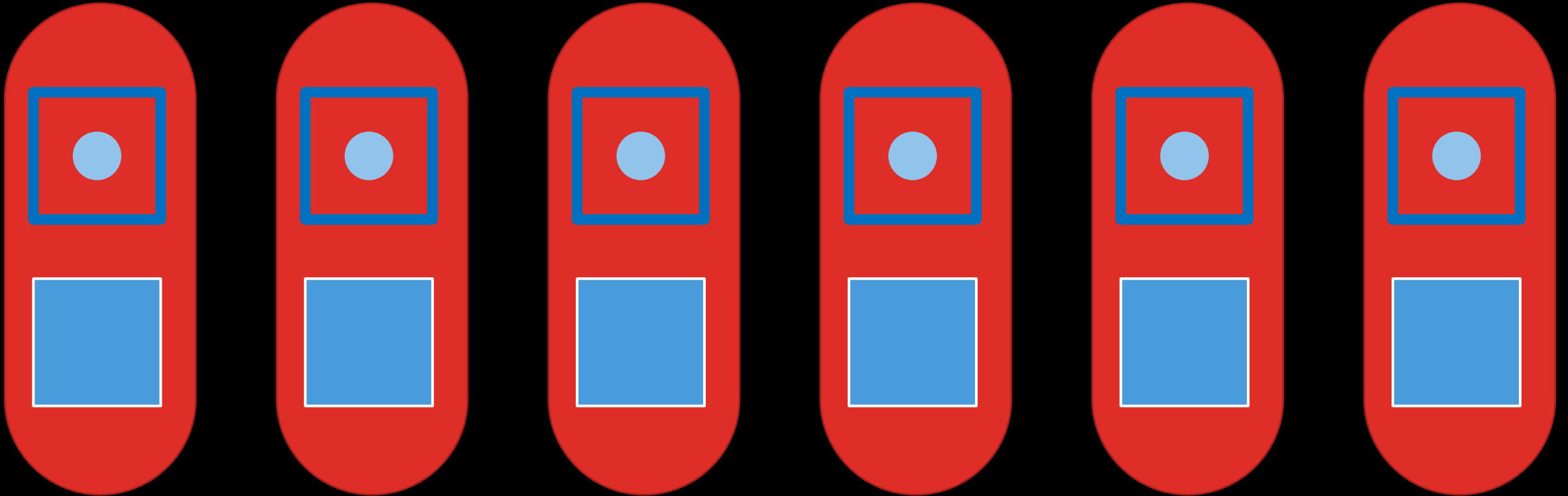
```
CallService1Async().ContinueWith(...);
```

```
CallService2Async().ContinueWith(...);
```

```
CallService3Async().ContinueWith(...);
```

CallService1Async, CallService2Async, and CallService3Async all run at the same time.

Get Data / Use Data

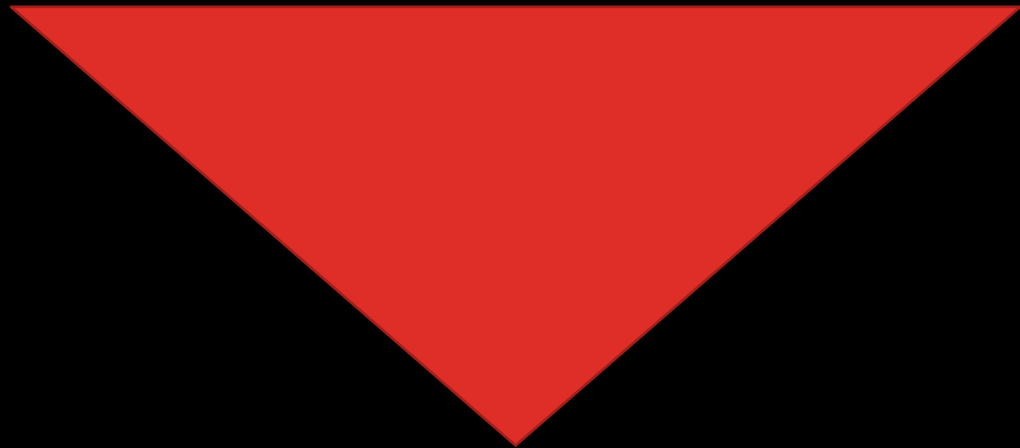
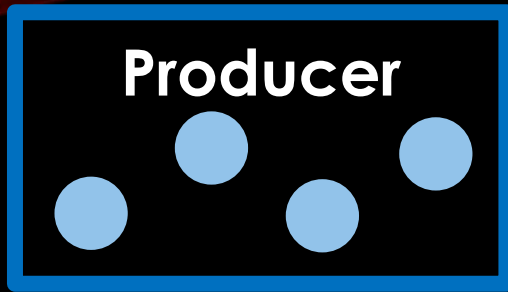
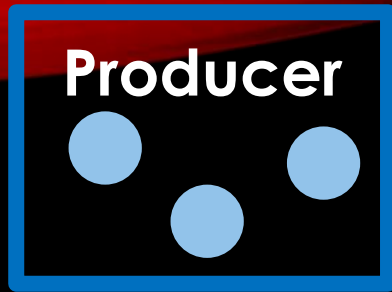


Parallel with Task

- `await Task.WhenAll()` can be used to determine when all tasks are complete
- Sample:

```
var taskList = new List<Task>();  
taskList.Add(task1);  
taskList.Add(task2);  
taskList.Add(task3);  
await Task.WhenAll(taskList);
```

Producer / Consumer



Consumer

What are Channels?

- Similar to a concurrent queue.
 - Write items to the channel.
 - Read items from the channel in the same order they were added.
 - Items are removed as they are read.
- Concurrent means that you can safely write and read from multiple threads without worry of missed writes or duplicate reads.

Where to Get Channels

- Built in to .NET 8 & .NET 9
- Available from NuGet for .NET Framework
 - `System.Threading.Channels`

Parallel with Channels

- Overall Steps
 - Create a channel
 - Write to a channel
 - Read from a channel
 - Mark the channel “complete”

Creating a Channel

- `CreateBounded<T>`
 - Creates a channel of a specific size
 - If the channel is full, writers are blocked until space is available

```
var channel = Channel.CreateBounded<Person>(10);
```

Writing to a Channel

- `writer.WriteAsync()`
 - Writes an item to the channel

```
await writer.WriteAsync(person);
```


Reading from a Channel (.NET 8/9)

- `reader.ReadAllAsync()`
 - Returns an `IAsyncEnumerable<T>`

```
await foreach(var person in reader.ReadAllAsync())  
{  
    // use item here  
}
```

- If the channel is empty, the loop will pause until an item is available.
- If the channel is “complete”, the loop will exit.

Reading from a Channel (.NET Framework)

- `WaitToReadAsync` and `TryRead` can be used to retrieve items.

```
while (await reader.WaitToReadAsync())
{
    while (reader.TryRead(out Person person))
    {
        // use item here
    }
}
```

Marking a Channel “Complete”

- `writer.Complete()`
 - Indicates that no further items will be written
 - Writing to a “complete” channel throws an exception
 - Reading from a “complete” channel will continue normally until the channel is empty

Parallel.ForEachAsync

- Loops over items and runs them in parallel
- “await” can be used safely inside the loop
 - Note: this is not true for “Parallel.ForEach”
- The entire loop can be “await”ed (this means all iterations will be complete)
- Available in .NET 8 & .NET 9 (not .NET Framework)

Parallel.ForEachAsync

Waits for all iterations to finish

```
await Parallel.ForEachAsync(
```

ids, **← The items to iterate over**

```
new ParallelOptions { MaxDegreeOfParallelism = 10 },
```

```
    async (id, _) =>
```

```
    {
```

```
        var person = await reader.GetPersonAsync(id);
```

```
        DisplayPerson(person);
```

```
    });
```

The method to run in parallel

Comparing Loops

	Await	Task	Channel	ForEachAsync
Runs in Parallel	No	Yes	Yes	Yes
Continuation on Main Thread	Yes	Yes (optional)	Yes	No
Continuation in Parallel	No	Yes	No (optional)	Yes
Set Degrees of Parallelism	No	No	No	Yes

Considerations

- For desktop / mobile, running continuations on the main thread is required to access UI elements.
- If continuations run in parallel, then they need to be thread-aware or surrounded by a “lock”
- Being able to specify degrees of parallelism gives control over resource usage
 - Limiting CPU resources to allow the UI to update
 - Limiting number of concurrent network connections



Lab

Lab 04 – Parallel Practices

[https://github.com/jeremybytes/
async-workshop-2025](https://github.com/jeremybytes/async-workshop-2025)

Progress Reporting

Progress Reporting

- `IProgress<T>`
 - Good for a parameter of an async method.
- `Progress<T>`
 - Built-in type for creating an `IProgress<T>` object.

Reporting Progress from an Async Method

```
public async Task<List<Person>> GetPeopleAsync(IProgress<int> progress)
{
    for (int i = 0; i < ids.Count; i++)
    {
        // Other work here
        int percentComplete = CalculatePercentComplete();
        progress.Report(percentComplete);
    }
    return people;
}
```

Configuring Progress<T>

- Action<T> callback in constructor

```
Progress<int> progress = new(p => ProgressBar.Value = p);
```

- ProgressChanged event handler

```
Progress<int> progress = new();  
progress.ProgressChanged +=  
    (_, e) => ProgressBar.Value = e;
```

Note: If the callback and event handlers are configured, they all run.

Reporting Types

The reporting type can be simple or complex.

- Single Value (simple)

```
Progress<int> progress = new();
```

- Custom Type (complex)

```
Progress<ProgressMessage> progress = new();
```

- Tuple (complex)

```
Progress<(int, string)> progress = new();
```

Tuple Sample

```
complexProgress.ProgressChanged += (_, e) =>
{
    (int progress, string message) = e;
    ProgressBar.Value = progress;
    ProgressText.Text = progress switch
    {
        0 => "",
        100 => "",
        _ => $"Current Item: {message}",
    };
};
```



Lab

Lab – Progress Reporting

[https://github.com/jeremybytes/
async-workshop-2025](https://github.com/jeremybytes/async-workshop-2025)

Breaking Async

Breaking Async

- Blocking / Potential Deadlock
 - .Result
 - .GetAwaiter().GetResult()
 - .Wait()
 - Task.WaitAll()
- Blocking / No Deadlock
 - JoinableTaskFactory

.Result

- Using .Result blocks until the result is populated.
- May result in a deadlock.

```
PersonReader reader = new();  
return reader.GetPeopleAsync().Result;
```

.GetAwaiter().GetResult()

- Using .GetAwaiter().GetResult() blocks until the result is populated.
- May result in a deadlock.

```
PersonReader reader = new();  
return reader.GetPeopleAsync().GetAwaiter().GetResult();
```

`.GetAwaiter().GetResult()`

DANGER

`.GetAwaiter().GetResult()` was designed for internal use only.

Remarks

This method is intended for compiler use rather than use directly in code.

<https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.getawaiter>



.Wait()

- Using .Wait() blocks until the async process is done.
- May result in a deadlock.

```
PersonReader reader = new();  
Task<List<Person>> peopleTask = reader.GetPeopleAsync();  
peopleTask.Wait();  
return peopleTask.Result;
```

Task.WaitAll()

- Using Task.WaitAll() blocks until the async process is done.
- May result in a deadlock.

```
PersonReader reader = new();  
Task<List<Person>> peopleTask = reader.GetPeopleAsync();  
Task.WaitAll([peopleTask]);  
return peopleTask.Result;
```

JoinableTaskFactory

- Using JoinableTaskFactory will block, but will not deadlock.
- Available from NuGet:
 - Microsoft.VisualStudio.Threading

JoinableTaskFactory Sample

```
JoinableTaskContext context = new();  
JoinableTaskFactory factory = new(context);  
  
List<Person> people = factory.Run(async () =>  
    await reader.GetPeopleAsync());  
return people;
```


.ConfigureAwait(false)

- .Result
 - .GetAwaiter().GetResult()
 - .Wait()
 - Task.WhenAll()
-
- All of the above processes block, but if the async code uses **.ConfigureAwait(false)**, they may not deadlock.



`.ConfigureAwait(false)`

JoinableTaskFactory **does not
deadlock** regardless of
whether the async code uses
`.ConfigureAwait(false)` or not.

Async Constructor?

- Constructors **cannot** be asynchronous.

```
public AsyncConstructorReader()  
{  
    PersonReader reader = new();  
  
    // NOT ALLOWED  
    people = await reader.GetPeopleAsync();  
}
```

Static Factory Method

- Static factory methods **can** be asynchronous.

```
private AsyncConstructorReader() { }

public static async Task<AsyncConstructorReader>
    CreateReaderAsync()
{
    AsyncConstructorReader myself = new();
    PersonReader taskLibraryReader = new();
    people = await taskLibraryReader.GetPeopleAsync();
    return myself;
}
```

Using a Static Factory Method

```
private async void AsyncConstructorButton_Click(  
    object sender, RoutedEventArgs e)  
{  
    ClearListBox();  
    IPersonReader reader =  
        await AsyncConstructorReader.CreateReaderAsync();  
    FetchData(reader);  
}
```

Wrap Up



Thank You!

Jeremy Clark

- jeremybytes.com
- jeremy@jeremybytes.com
- [@jeremybytes](https://twitter.com/jeremybytes)

<https://github.com/jeremybytes/async-workshop-2025>