

# Asynchronous and Parallel Programming in C#

June 24-25, 2025 | Virtual Training

**Jeremy Clark**  
Developer Educator  
[jeremybytes.com](http://jeremybytes.com)

Level: Intermediate

# Schedule

- Days Tuesday & Wednesday
- Class Hours 9:00 a.m. – 5:00 p.m.
- Breaks throughout the day
- Lunch 12:30 p.m. – 1:30 p.m.

Q&A throughout the day

All Times are Central Daylight Time

# Materials

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

# Agenda Part 1

- Running Asynchronous Code
  - Task and await
  - Desktop and Web
- Where Continuations Run
  - TaskScheduler
  - ConfigureAwait
- Useful Properties & Methods
  - Continuation parameters
  - .IsFaulted
  - Exception Handling
  - Cancellation
- Writing Asynchronous Methods
  - Return or await
  - async void
- Task vs. ValueTask
- Progress Reporting
- IEnumerable<T>
  - yield return
  - Task.WhenEach
  - Cancellation

# Agenda Part 2

- Nerd Break: IL
  - Compiling Task vs. await
- Breaking Async
  - .Result
  - .Wait()
  - JoinableTaskFactory
  - Constructors
- Parallel Programming
  - Tasks
  - Channels
  - Parallel.ForEachAsync
- Parallel Exception Handling
- Parallel Considerations
  - Thread-safe Operations
  - Degrees of Parallelism



# Topics (in no particular order)

- .NET Framework vs .NET 8/9
- TaskContinuationOptions
- CancellationToken.None
- ThrowIfCancellationRequested
- Action
- Lambda expressions
- Task.Run()
- Context vs. no context
- BlockingCollection
- Task.Result
- AggregateException
- OperationCanceledException
- CancellationTokenSource
- Linked Token Source
- async MVC Controllers
- IProgress<T> vs. Progress<T>
- .GetAwaiter().GetResult()
- SemaphoreSlim

# 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 01 – Recommended Practices and Continuations

[https://github.com/jeremybytes/  
async-virtual-seminar-2025](https://github.com/jeremybytes/async-virtual-seminar-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.

# Linked Token Source

- Token sources can be linked together.

```
var parentCts = new CancellationTokenSource();  
var childCts = CancellationTokenSource  
    .CreateLinkedTokenSource(parentCts.Token);
```

- If the parent token is canceled, the child token is also canceled.
- The child token can be canceled independently.



Lab

## Lab 02 – Adding Async to an Existing Application

[https://github.com/jeremybytes/  
async-virtual-seminar-2025](https://github.com/jeremybytes/async-virtual-seminar-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.

# Task vs. ValueTask



# Task vs. ValueTask

## Advice

Use “Task” unless you have  
specific performance  
or memory issues.

# ValueTask

- ValueTask is a struct (no Task allocation unless it is required).
- Can be useful for methods that have both async and non-async paths (where the non-async path is used more frequently).

# ValueTask

```
public async ValueTask<Person> GetPersonAsync(int id)
{
    if (!cacheValid)
    {
        cachedPerson = await reader.GetPersonAsync(id);
        return cachedPerson;
    }
    else
    {
        return cachedPerson;
    }
}
```

**Async path**

**Non-async path**

# ValueTask Restrictions

- DO NOT await multiple times.

```
ValueTask<Person> personTask = GetPersonAsync(3);  
Person selectedPerson = await personTask;  
Person personCopy = await personTask;
```

- The ValueTask may already be recycled on the second “await”.

# ValueTask Restrictions

- DO NOT await concurrently.

```
ValueTask<Person> personTask = GetPersonAsync(3);  
Task.Run(async() => await personTask);  
Task.Run(async() => await personTask);
```

- This has the effect of awaiting the ValueTask multiple times.



# ValueTask Restrictions

- DO NOT use `GetAwaiter().GetResult()`.

```
ValueTask<Person> personTask = GetPersonAsync(3);  
Person selectedPerson =  
    personTask.GetAwaiter().GetResult();
```

This should go without saying, but it is specifically called out by Stephen Toub  
<https://devblogs.microsoft.com/dotnet/understanding-the-whys-whats-and-whens-of-valuetask/>



# Task vs. ValueTask

## Advice

Use “Task” unless you have  
specific performance  
or memory issues.

# 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 03 – Progress Reporting

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



# `IAsyncEnumerable<T>`

# IAsyncEnumerable<T>

- Similar to IEnumerable<T>, but it is asynchronous.
- `foreach` can be used to iterate over IEnumerable<T>
- `await foreach` can be used to iterate over IAsyncEnumerable<T>
- Available in .NET 8 & .NET 9 (not .NET Framework)

# IEnumerable<T> Sample

```
public IEnumerator<int> GetEnumerator()  
{  
    var value = (current: 0, next: 1);  
    while (true)  
    {  
        value = NextFibonacci(value);  
        yield return value.current;  
    }  
}
```

# Using IEnumerable<T>

```
FibonacciSequence fibs = new();  
foreach (int fib in fibs)  
{  
    Console.WriteLine(fib);  
}
```

# IAsyncEnumerable<T> Sample

```
public async IAsyncEnumerator<int> GetAsyncEnumerator(...)  
{  
    var value = (current: 0, next: 1);  
    while (true)  
    {  
        value = await NextFibonacciAsync(value);  
        yield return value.current;  
    }  
}
```

# Using IEnumerable<T>

```
AsyncFibonacciSequence asyncFibs = new();  
await foreach (int fib in fibs)  
{  
    Console.WriteLine(fib);  
}
```

# Task.WhenEach

\*\*\*New in .NET 9\*\*\*

- Task.WhenEach() takes a collection of tasks and returns an `IAsyncEnumerable<Task>`
- `await foreach` will run the body of the loop as each Task completes.



# IAsyncEnumerable<T> Cancellation

- A method that returns IAsyncEnumerable<T> must attribute a cancellation token parameter with [EnumeratorCancellation]

```
public async IAsyncEnumerable<Person> GetPeopleAsyncEnumerable(  
    IProgress<int> progress,  
    [EnumeratorCancellation] CancellationToken cancellationToken){ }
```

# Nerd Break: IL

# Intermediate Language & ILDasm

- IL (Intermediate Language)
  - C# (and other .NET languages) compile to an assembly-like intermediate language.
  - The .NET Runtime turns the IL into native machine code.

Note: Another option is Native AOT (Ahead-of-Time) Compilation. This compiles to native machine code (and not IL).

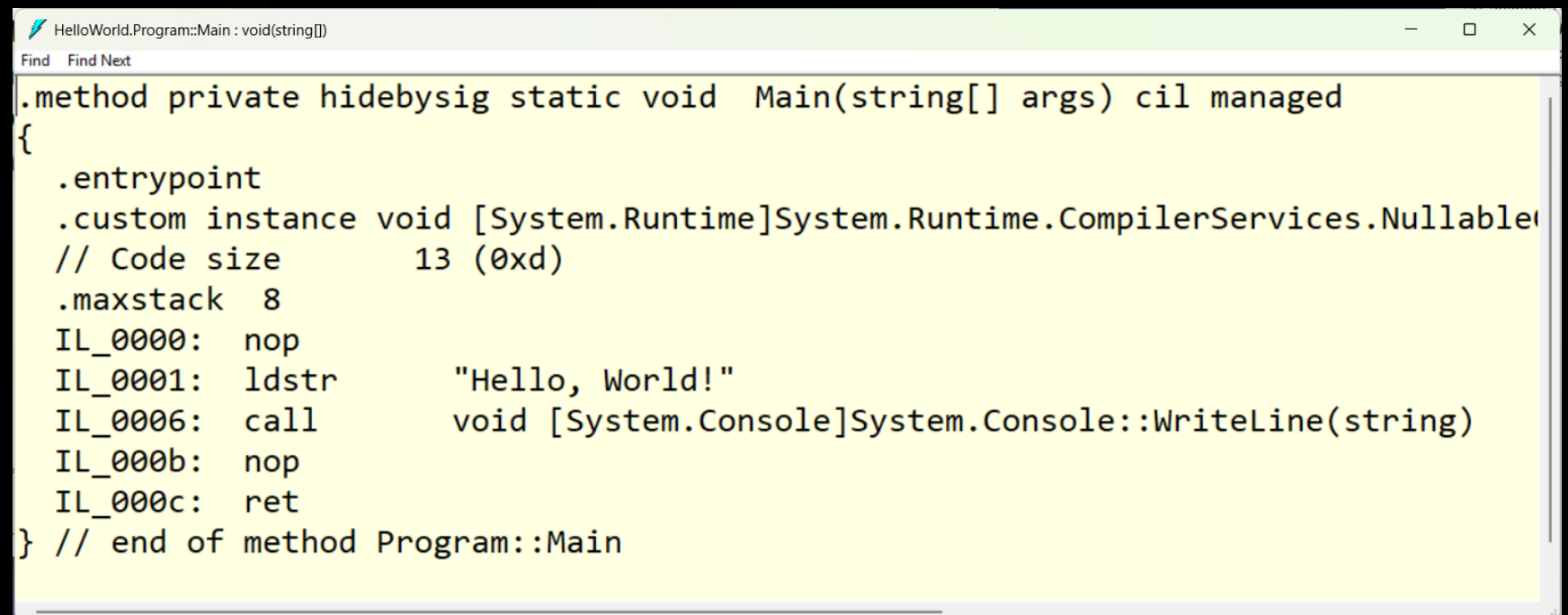
# Intermediate Language & ILDasm

- ILDasm (Intermediate Language Disassembler)
  - A .NET tool that shows the IL inside of an assembly.
  - Included with Visual Studio (and the .NET SDK).

<https://github.com/jeremybytes/ildasm-setup>

# Sample IL

```
static void Main(string[] args)
{
    Console.WriteLine("Hello, World!");
}
```



```
HelloWorld.Program::Main : void(string[])
Find Find Next

.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    .custom instance void [System.Runtime]System.Runtime.CompilerServices.NullableAttribute::SetMethod(ushort, ushort)
    // Code size 13 (0xd)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldstr "Hello, World!"
    IL_0006: call void [System.Console]System.Console::WriteLine(string)
    IL_000b: nop
    IL_000c: ret
} // end of method Program::Main
```

# Why? Curiosity!

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

```
using var enumerator =
    people.GetEnumerator();

while(enumerator.MoveNext())
{
    PersonListBox.Items.Add(
        enumerator.Current);
}
```

These blocks of code compile to the same IL

# Task vs. await

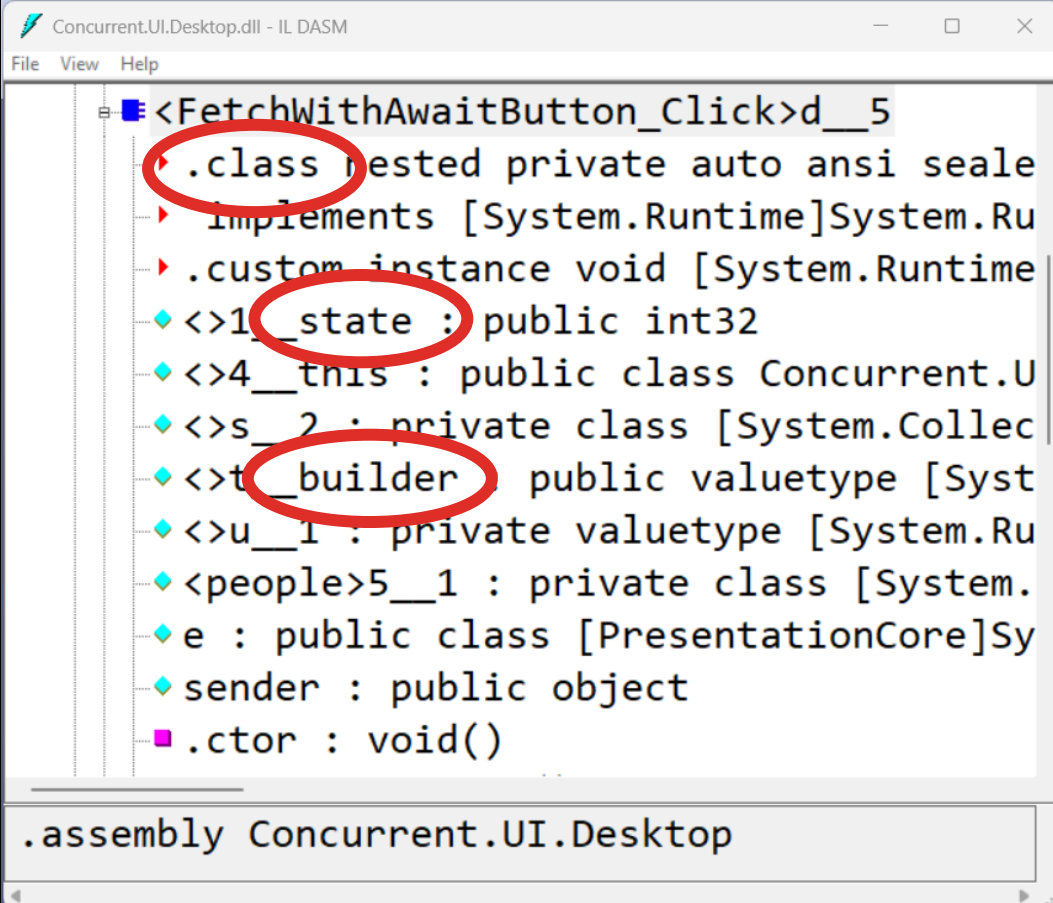
- Task compiles to a standard method

```
Concurrent.UI.MainWindow::FetchWithTaskButton_Click : void(object,class [PresentationCore]System.Windows.RoutedEventArgs)
Find Find Next
.locals init (class [System.Runtime]System.Threading.Tasks.Task`1<class [System.Col
        valuetype [System.Runtime]System.Threading.CancellationToken V_1)
IL_0000: nop
IL_0001: ldarg.0
IL_0002: call instance void Concurrent.UI.MainWindow::ClearListBox()
IL_0007: nop
IL_0008: ldarg.0
IL_0009: ldfld class [TaskAwait.Library]TaskAwait.Library.PersonReader Concur
IL_000e: ldloc.s V_1
IL_0010: initobj [System.Runtime]System.Threading.CancellationToken
IL_0016: ldloc.1
IL_0017: callvirt instance class [System.Runtime]System.Threading.Tasks.Task`1<c
IL_001c: stloc.0
IL_001d: ldloc.0
IL_001e: ldarg.0
IL_001f: ldftn instance void Concurrent.UI.MainWindow::PopulateListBox()
IL_0025: newobj instance void class [System.Runtime]System.Action`1<class [Sys
```



# Task vs. await

- await compiles to a state machine



The screenshot shows the IL DASM window for Concurrent.UI.Desktop.dll. The assembly name is Concurrent.UI.Desktop. The method being viewed is <FetchWithAwaitButton\_Click>d\_\_5. The IL code is as follows:

```
.class nested private auto ansi sealed  
    implements [System.Runtime]System.Runtime.CompilerServices.AsyncStateMachineAttribute  
    .custom instance void [System.Runtime]System.Runtime.CompilerServices.AsyncStateMachineAttribute::ctor(int32)  
    <>1__state : public int32  
    <>4__this : public class Concurrent.UI.Desktop.FetchWithAwaitButton_Click  
    <>s_2 : private class [System.Collections.Generic]System.Collections.Generic.List`1<System.Threading.Tasks.Task>  
    <>t_builder : public valuetype [System.Threading.Tasks]System.Threading.Tasks.TaskBuilder  
    <>u_1 : private valuetype [System.Runtime]System.Runtime.CompilerServices.AsyncTaskMethodBuilder`1<System.Threading.Tasks.Task>  
    <people>5__1 : private class [System.Threading.Tasks]System.Threading.Tasks.Task  
    e : public class [PresentationCore]System.Windows.Controls.Button  
    sender : public object  
    .ctor : void()
```

The .assembly Concurrent.UI.Desktop



# 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);  
}
```

# 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.



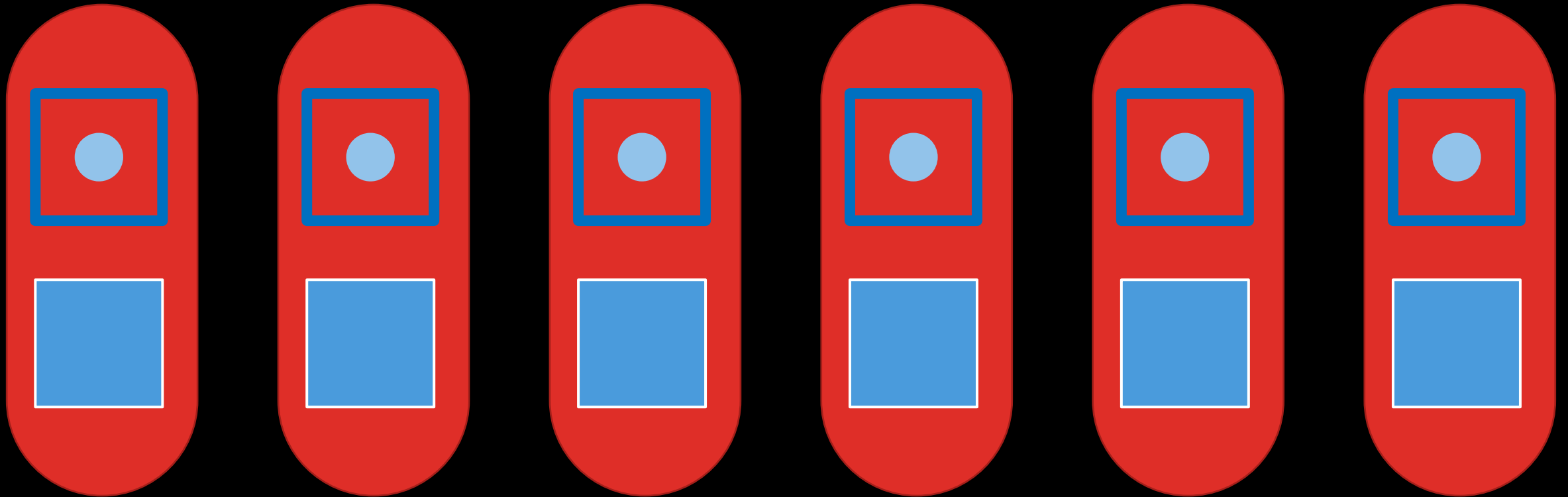
# Parallel with Task

- 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

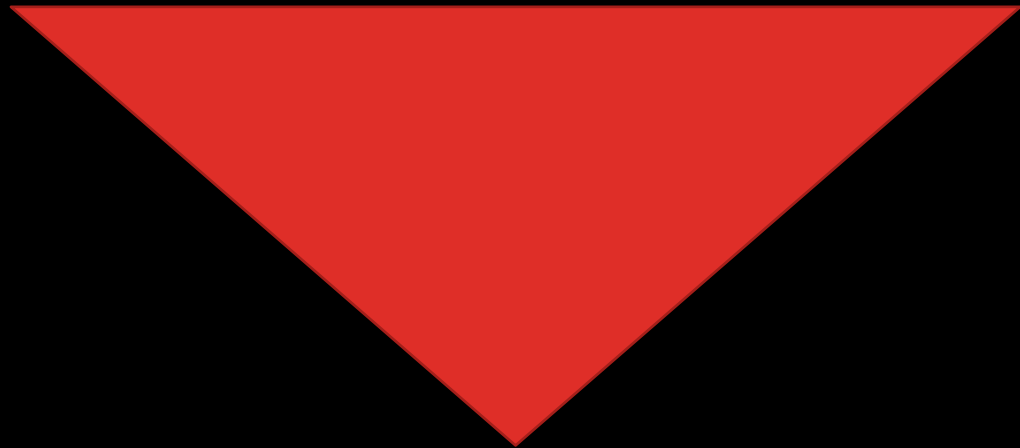
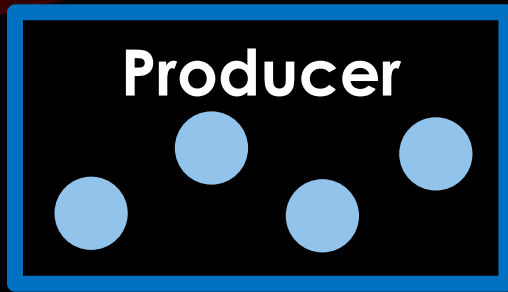
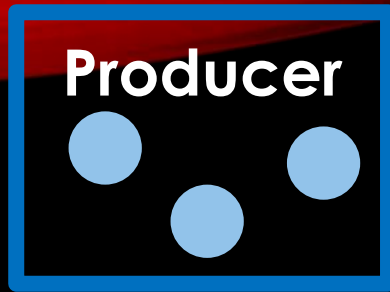


# Parallel with Task

- `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

	<b>Await</b>	<b>Task</b>	<b>Channel</b>	<b>ForEachAsync</b>
<b>Runs in Parallel</b>	No	Yes	Yes	Yes
<b>Continuation on Main Thread</b>	Yes	Yes (optional)	Yes	No
<b>Continuation in Parallel</b>	No	Yes	No (optional)	Yes
<b>Set Degrees of Parallelism</b>	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-virtual-seminar-2025](https://github.com/jeremybytes/async-virtual-seminar-2025)

# Parallel Exception Handling

# Parallel Exceptions

- Reminder, when you await a faulted Task, `AggregateException` is unwrapped and only 1 exception is thrown.
- In a `Parallel.ForEachAsync` loop, the loop short-circuit if an exception is thrown.

# Parallel Exceptions Issue #1

- Only 1 inner exception of the aggregate is thrown.

```
try
{
    await Task.WhenAll(continuations);
}
catch (Exception ex)
{
    DisplayException(ex); // Shows 1 inner exception
}
```

# Show All Exceptions

- By adding a continuation, you can access the full `AggregateException` (and all of the inner exceptions)

```
await Task.WhenAll(continuations)
    .ContinueWith(task =>
    {
        // Task.Exception contains full AggregateException
        if (task.IsFaulted)
            DisplayException(task.Exception);
    });
```

## Parallel Exceptions Issue #2

- Loop short-circuits, i.e., stops processing.

```
try
{
    // Stops processing on exception
    await Parallel.ForEachAsync(...);
}
catch (Exception ex)
{
    DisplayException(ex);
}
```

# Log and Continue

- Move the try/catch block inside the parallel loop.

```
await Parallel.ForEachAsync(ids, async (id, _) => {  
    try  
    { // Possible exception thrown here }  
    catch (Exception ex)  
    {  
        // Each exception logged as it happens  
        DisplayException(ex);  
    }  
});
```

# ConfigureAwait

- ConfigureAwait has a **SuppressThrowing** parameter that will keep the await from throwing an exception even if the Task is faulted.
- This can be useful if you handle exceptions separately.
- .NET 8 & .NET 9 (not .NET Framework)



# SuppressThrowing Sample

```
await Task.WhenAll(continuations)  
    .ConfigureAwait(ConfigureAwaitOptions.SuppressThrowing);
```

- Check Stephen Cleary's article for details:

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

# Channel Exceptions

- Channels can be used for pipelines (meaning, sending items from one process to another).
- A separate error channel can be useful for retry and/or logging purposes.

# Writing to Exception Channel

```
await Parallel.ForEachAsync(ids, async (id, _) => {  
    try {  
        // Do stuff and write to success channel  
        await writer.WriteAsync(item);  
    }  
    catch (Exception ex) {  
        // Write to error channel  
        await errorWriter.WriteAsync(ex);  
    }  
});
```

# Reading from Exception Channel

```
private static async Task ProcessErrors(  
    ChannelReader<Exception> errorReader)  
{  
    await foreach (var ex in errorReader.ReadAllAsync())  
    {  
        // Retry and/or log errors  
        ExceptionReporter.LogException(ex);  
    }  
}
```

# Parallel Considerations



# Parallel Considerations

- Thread-safe collections
- Thread-safe operations
- Continuing on the main thread
- Limit degrees of parallelism

# Thread-safe Collections

- `List<T>` is not thread-safe. If you call `Add` from concurrent processes, some items may not be added.
- When writing to a collection from concurrent processes (such as parallel code), use a concurrent collection.
- `System.Collections.Concurrent`
- `BlockingCollection<T>` is a good place to start.

# Thread-safe Increment

- Increment (++) and decrement (--) operators are not thread-safe.
- The Interlocked class has a useful set of thread-safe methods.
  - Increment
  - Decrement
  - Add
  - Several others



# Interlocked.Increment Sample

```
await Parallel.ForEachAsync(items, async (i, _) => {  
    try {  
        // Process Item  
        Interlocked.Increment(ref TotalProcessed);  
    }  
    catch (Exception ex) {  
        Interlocked.Increment(ref TotalExceptions);  
        ExceptionReporter.ShowException(ex);  
    }  
});
```

# Comparing Loops

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

# Continuations on Main Thread

- For Windows Desktop and MAUI applications, the **Dispatcher** class can be used to run a delegate on the UI thread.

# Dispatcher Samples

- Windows Desktop - Dispatcher.Invoke(Action)

```
Dispatcher.Invoke(() => CreateUIElements(result, DigitsBox));
```

- MAUI - Dispatcher.Dispatch()

```
Dispatcher.Dispatch(() => CreateUIElements(result, DigitsBox));
```

# Max Degrees of Parallelism

- SemaphoreSlim can be used to limit the number of concurrent processes.
- Steps
  - Create SemaphoreSlim with initial count.
  - Call WaitAsync to wait for entry.
  - Call Release when done.

# Semaphore Slim Sample

```
using SemaphoreSlim semaphore = new(10);  
foreach (var imageData in rawData) {  
    await semaphore.WaitAsync();  
    var task = classifier.Predict(imageData);  
    var continuation = task.ContinueWith(t => {  
        CreateUIElements(t.Result, DigitsBox);  
        semaphore.Release();  
    });  
}
```



Lab

## Lab 05 – Parallel Exception Handling

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



# Wrap Up



Thank You!

Jeremy Clark

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

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