Lab 05 - Working with Exceptions in Parallel Loops

Objectives

By default, most parallel loops short-cicuit (i.e., stop processing) when an exception is encoutered. In addition, the exception may be incomplete. In this lab, we will look at several parallel loops with 2 goals. (1) Get complete exception information from a short-circuiting loop, and (2) prevent the loop from short-circuiting and collecting all exceptions so that we can log or reprocess the data.

Application Overview

The "Starter" folder contains the code files for this lab. If you get stuck along the way, you can look at the "Completed" folder for the finished lab code. In addition, this lab contains and "Intermediate" folder that has some of the "in-between" code. The walkthrough will reference the "Intermediate" folder where applicable.

> Both .NET 8 and .NET 9 versions of this code are available. If you are using .NET 8, be aware that instructions that reference "dotnet9" or "net9.0" need to be changed to "dotnet8" and "net8.0", respectively.

Visual Studio 2022: Open the "ParallelExceptions.sln" solution.
Visual Studio Code: Open the "Starter" folder in VS Code.

There are multiple console applications to explore different loop types.

- **ExceptionLibrary** This is a shared project with code to output exceptions to the console and also to throw exceptions at random intervals.
- **ForEachAsyncExceptions** This is a console application that uses `Parallel.ForEachAsync` for parallel code.
- **WhenAllExceptions** This is a console application that uses `Task.WhenAll` to determine that a set of tasks has completed.
- **ChannelExceptions** This is a console application that uses a `Channel` for communication between a parallel producer and a sequential consumer.

We'll look at some of the details below.

Visual Studio 2022:
Build the application (by using F6 or "Build Solution" from the Build menu). Because these projects throw multiple exceptions, I prefer to run them from a command line (more information below).

Visual Studio Code:
Build the application. One way to do this is from the command line.

- Open a command prompt (PowerShell, Terminal, or cmd.exe) and navigate to the project folder: *[working_directory]\Lab05\dotnet9\Starter*
- Build the application using "dotnet build" This will build all of the projects in the solution.

```
PS C:..\Lab05\dotnet9\Starter> dotnet build
```

Running the Application

- Open a command prompt (PowerShell, Terminal, or cmd.exe) and navigate to the project folder for "ForEachAsyncExceptions".

> In Windows, and easy way to do this is to open the File Explorer, and navigate to the folder that you want, and then right-click and empty location and choose "Open PowerShell" or "Open Terminal" (if you have Windows Terminal installed).

Sample path for the project folder:
`C:\AsyncWorkshop\Labs\Lab05\dotnet9\Starter\ForEachAsyncExceptions\`

Note: be sure to use the appropriate path for the version of .NET you use.

```
PS C:\AsyncWorkshop\Labs\Lab05\dotnet9\Starter\ForEachAsyncExceptions>
```

- From the output folder, type `dotnet run` to run the application. (Most terminal applications have tab completion, so you can type the first few characters and press "Tab" multiple times until you find the correct one.)

```
PS [omitted-path]\Starter\ForEachAsyncException> dotnet run
```

- The application will output something similar to the following:

```
[some items omitted]
Processing item: 16
Processing item: 17
Processing item: 18
Processing item: 20
Processing item: 21
Processing item: 24
Processing item: 29
Processing item: 23
Processing item: 22
Processing item: 25
```

```
Processing item: 26
Processing item: 27
Processing item: 28
======================
Exception: Error in RunProcess loop: #16
======================
Total Processed: 26
Total Exceptions: 1
Done
```

All 3 applications attempt to process 100 records. In this example, only 26 records were processed before the loop short-circuited. A total of 1 exception is reported (although, as we'll see, there are more that are not reported).

Lab Goals

There are 2 goals for this lab:

- Report all of the exceptions from a short-circuiting loop.
- Update the loops so that they do not short-circuit. Process all 100 items and report all of the exceptions.

Current Classes & Methods

ExceptionLibrary/MightThrow.cs:

```
public static class MightThrow
{
    private static Random Randomizer { get; } = new();

    public static void Throw(int frequency, string message)
    {
        if (Randomizer.Next() % frequency == 0)
        {
            throw new Exception(message);
        }
    }
}
```

The `Throw` method allows us to throw exceptions at the frequency of our choice. For example, if we use a frequency of 5, then the method will throw an exception approximately 1 in 5 calls. This is indeterminate due to a random number, but the frequency works out on average.

ExceptionLibrary/ExceptionReporter.cs

This class has methods that will print exceptions to the console. In addition to standard exceptions, there is a method to print `AggregateException` with all of its inner exceptions.

Program.Main() (all 3 console applications)

Each console application has a `Main` method with the following code:

```
static async Task Main(string[] args)
{
    try
    {
        await RunProcess(100);
    }
    catch (Exception ex)
    {
        TotalExceptions++;
        ExceptionReporter.ShowException(ex);
    }

    Console.WriteLine($"Total Processed: {TotalProcessed}");
    Console.WriteLine($"Total Exceptions: {TotalExceptions}");
    Console.WriteLine("Done");
}
```

This gives us the basic framework for the applications. The `RunProcess` method in each application has the specifics for the parallel loop type.

Program.RunProcess (all 3 console applications)

The `RunProcess` parameter specifies the number of items to process. We won't go through the details of each application; the parallel loops are very similar to the code in Lab 04. The main difference is that these loops are not doing any actual work (such as an API call).

Hints

Showing All Exceptions

ForEachAsyncExceptions & WhenAllExceptions

- The reason we only get 1 exception is because we use `await` on multiple tasks (either the `ForEachAsync` loop or `Task.WhenAll`). When we `await` a task, only 1 of the inner exceptions is shown.
- Using a continuation gives you access to the full `AggregateException`. With the raw `AggregateException` you can look at all of the inner exceptions.

ChannelExceptions

- The `Produce` method uses a ForEachAsync loop. Rather than duplicating the code from the `ForEachAsyncExceptions` project, just skip this project and concentrate on the next goal.

Prevent Short-Circuiting

All Projects

- Consider using a try/catch block inside the loop to prevent the loop itself from throwing an exception.

ChannelExceptions

- Consider creating a new channel to hold exceptions. A separate error processor can output the exceptions to the console.

If you want more assistance, step-by-step instructions are included below. Otherwise, if you'd like to challenge yourself, **STOP READING NOW**

Showing All Exceptions: Step-By-Step

ForEachAsyncExceptions

> Note: the finished code for this section is part of the "Intermediate" solution.

1. Open the "ForEachAsyncExceptions/Program.cs" file.
2. Look at the `RunProcess` method:

```
private static async Task RunProcess(int numberOfItems)
{
    await Parallel.ForEachAsync(
        Enumerable.Range(1, numberOfItems),
        new ParallelOptions() { MaxDegreeOfParallelism = 10 },
        async (i, _) =>
        {
            Console.WriteLine($"Processing item: {i}");
            await Task.Delay(10); // simulate async task
            MightThrow.Throw(frequency, $"Error in RunProcess loop: #{i}");
            TotalProcessed++;
        });
}
```

- `Enumerable.Range(1, numberOfItems)` creates an IEnumerable that contains the values 1 through 100 (in this case).
- `MightThrow` throws an exception based on the frequency (approximately 1 in 5 times in this case).
- `TotalProcessed` is used to keep track of how many records are successfully processed. If an exception is thrown, then this value is not incremented.

2. Run the application an look at the results.

```
Processing item: 6
Processing item: 4
Processing item: 3
Processing item: 7
Processing item: 1
Processing item: 5
Processing item: 8
Processing item: 2
Processing item: 9
Processing item: 10
Processing item: 11
Processing item: 12
Processing item: 13
Processing item: 14
Processing item: 15
Processing item: 16
Processing item: 17
Processing item: 18
======================
Exception: Error in RunProcess loop: #2
======================
Total Processed: 15
Total Exceptions: 1
Done
```

> Since we use random values, this output will change each time the application is run.

Note that in this run there were 15 items successfully processed, and 1 Exception reported. The problem is that there were 18 items "Processing" which means 2 items are missing.

3. `ForEachAsync` returns a Task. Set up a continuation on that task.

```
await Parallel.ForEachAsync(
    Enumerable.Range(1, numberOfItems),
    new ParallelOptions() { MaxDegreeOfParallelism = 10 },
    async (i, _) =>
    {
        Console.WriteLine($"Processing item: {i}");
        await Task.Delay(10); // simulate async task
        MightThrow.Throw(frequency, $"Error in RunProcess loop: #{i}");
        TotalProcessed++;
    })
    .ContinueWith(task =>
    {
```

```
    });
```

4. Use `TaskContinuationOptions` to only run the continuation when the task is faulted.

```
    .ContinueWith(task =>
    {

    }, TaskContinuationOptions.OnlyOnFaulted);
```

5. In the continuation, increment the `TotalExceptions` field and use the `ExceptionReporter` to output the exception to the console.

```
    .ContinueWith(task =>
    {
        TotalExceptions++;
        ExceptionReporter.ShowException(task.Exception!);
    }, TaskContinuationOptions.OnlyOnFaulted);
```

6. Build and run the application.

```
[some output omitted]
Processing item: 11
Processing item: 16
Processing item: 13
Processing item: 14
Processing item: 15
Processing item: 12
Processing item: 17
Processing item: 18
Processing item: 19
======================
Aggregate Exception: Count 4
Inner Exception: Error in RunProcess loop: #2
Inner Exception: Error in RunProcess loop: #12
Inner Exception: Error in RunProcess loop: #19
Inner Exception: Error in RunProcess loop: #15
======================
Total Processed: 15
Total Exceptions: 1
Done
```

This shows that 15 records were processed (and still reports only 1 exception). But this code gets the `AggregateException` and shows all of the inner exceptions instead of just 1. So we have 4 exceptions reported. That is a total of 19 items which matches this particular application run.

Here is the completed method:

```
private static async Task RunProcess(int numberOfItems)
{
    await Parallel.ForEachAsync(
        Enumerable.Range(1, numberOfItems),
        new ParallelOptions() { MaxDegreeOfParallelism = 10 },
        async (i, _) =>
        {
            Console.WriteLine($"Processing item: {i}");
            await Task.Delay(10); // simulate async task
            MightThrow.Throw(frequency, $"Error in RunProcess loop: #{i}");
            TotalProcessed++;
        })
        .ContinueWith(task =>
        {
            TotalExceptions++;
            ExceptionReporter.ShowException(task.Exception!);
        }, TaskContinuationOptions.OnlyOnFaulted);
}
```

7. Look at the try/catch block of the `Main` method.

```
try
{
    await RunProcess(100);
}
catch (Exception ex)
{
    TotalExceptions++;
    ExceptionReporter.ShowException(ex);
}
```

The `catch` block is no longer used. The `RunProcess` method does not throw an exception with the current state of the code.

8. Update the `catch` block of the main method with a message that we should never see.

```
try
{
```

```
    await RunProcess(100);
}
catch (Exception)
{
    Console.WriteLine("You shouldn't be here.");
}
```

You can also remove the `ex` since the exception is not used in the `catch` block.

9. Build and run the application to make sure that this catch block is not used. You should get similar output to the last application run.

In a later section, we will eliminate the short-circuiting of the loop and capture the inner exceptions separately as they occur.

Showing All Exceptions: Step-By-Step

WhenAllExceptions

> Note: the finished code for this section is part of the "Intermediate" solution.

1. Open the "WhenAllExceptions/Program.cs" file.
2. Review the `RunProcess` method.

```
private static async Task RunProcess(int numberOfItems)
{
    List<Task> allTasks = new();

    foreach (var i in Enumerable.Range(1, numberOfItems))
    {
        Task currentTask = Task.Run(async () =>
        {
            Console.WriteLine($"Processing item: {i}");
            await Task.Delay(10); // simulate async task
            MightThrow.Throw(frequency, $"Error in RunProcess loop: #{i}");
            TotalProcessed++;
        });
        allTasks.Add(currentTask);
    }

    await Task.WhenAll(allTasks);
}
```

This method uses a `foreach` loop to generate 100 new Tasks. Each task has the same code as in the `ForEachAsync` loop above. Each task is added to the `allTasks` collection. Finally, `Task.WhenAll` waits for all of

the tasks to complete.

3. Build and run the application.

```
[some output omitted]
Processing item: 35
Processing item: 20
Processing item: 97
Processing item: 98
Processing item: 54
Processing item: 85
======================
Exception: Error in RunProcess loop: #1
======================
Total Processed: 75
Total Exceptions: 1
Done
```

In this case, all 100 items are processed (meaning, the loop does not short-circuit), but we do not see all of the exceptions. Similar to with the `ForEachAsync` loop, we can use a continuation to access the `AggregateException`.

4. Add a continuation to the `Task.WhenAll` method.

```
await Task.WhenAll(allTasks)
    .ContinueWith(task =>
    {
    });
```

5. As we did above, use `TaskContinuationOptions` to only run this when faulted. Then add the code to increment `TotalExceptions` and to output the exception to the console.

```
await Task.WhenAll(allTasks)
    .ContinueWith(task =>
    {
        TotalExceptions++;
        ExceptionReporter.ShowException(task.Exception!);
    }, TaskContinuationOptions.OnlyOnFaulted);
```

6. Build and run the application.

```
[some items omitted]
Processing item: 94
Processing item: 95
Processing item: 96
Processing item: 97
Processing item: 21
Processing item: 22
Processing item: 23
======================
Aggregate Exception: Count 22
Inner Exception: Error in RunProcess loop: #32
[some items omitted]
Inner Exception: Error in RunProcess loop: #92
Inner Exception: Error in RunProcess loop: #39
Inner Exception: Error in RunProcess loop: #17
======================
Total Processed: 72
Total Exceptions: 1
Done
```

This is better. We now see all of the exceptions inside the `AggregateException` -- 22 of them. But we have a problem. There were 72 items processed and 22 exceptions. **That does not add up to 100**.

The issue here is the way that we update the fields. The ++ operator is not thread-safe, so we should not use it in parallel operations. But there is a special `Interlocked` type that we can use.

7. Look at the code inside the `foreach` loop.

```
Console.WriteLine($"Processing item: {i}");
await Task.Delay(10); // simulate async task
MightThrow.Throw(frequency, $"Error in RunProcess loop: #{i}");
TotalProcessed++;
```

8. Change the update of `TotalProcessed` to use the `Interlocked.Increment` method.

```
Console.WriteLine($"Processing item: {i}");
await Task.Delay(10); // simulate async task
MightThrow.Throw(frequency, $"Error in RunProcess loop: #{i}");
Interlocked.Increment(ref TotalProcessed);
```

This is a thread-safe process that will increment the `TotalProcessed` field. Notice that you must pass the field by reference (using the `ref` keyword).

9. Build and run the application.

```
[some items omitted]
Processing item: 90
Processing item: 91
Processing item: 92
Processing item: 93
Processing item: 64
Processing item: 65
Processing item: 81
Processing item: 97
Processing item: 98
Processing item: 99
Processing item: 100
======================
Aggregate Exception: Count 24
Inner Exception: Error in RunProcess loop: #92
[some items omitted]
Inner Exception: Error in RunProcess loop: #20
Inner Exception: Error in RunProcess loop: #6
Inner Exception: Error in RunProcess loop: #33
Inner Exception: Error in RunProcess loop: #100
======================
Total Processed: 76
Total Exceptions: 1
Done
```

Now we see a correct number of items processed (76 here) which matches up with the number of exceptions (24). All 100 records are represented.

10. Just to be safe, change the increment of the `TotalExceptions` field in the continuation to also use `Interlocked`.

```
await Task.WhenAll(allTasks)
    .ContinueWith(task =>
    {
        Interlocked.Increment(ref TotalExceptions);
        ExceptionReporter.ShowException(task.Exception!);
    }, TaskContinuationOptions.OnlyOnFaulted);
```

11. As a final step, update the `catch` block in the `Main` method since it is no longer used (just like in the previous application).

```
try
{
    await RunProcess(100);
```

```
}
catch (Exception)
{
    Console.WriteLine("You shouldn't be here.");
}
```

In a later section, we will capture the inner exceptions separately as they occur.

Prevent Short-Circuiting: Step-By-Step

ForEachAsyncExceptions

> Note: the finished code for this section is part of the "Completed" solution. This assumes that you completed the above steps for the "WhenAllExceptions" application. If not, you can start with the code in the "Intermediate" solution.

1. Open the "ForEachAsyncExceptions/Program.cs" file.

2. Build and run the application.

```
[some items omitted]
Processing item: 12
Processing item: 13
Processing item: 14
Processing item: 15
Processing item: 16
Processing item: 17
======================
Aggregate Exception: Count 3
Inner Exception: Error in RunProcess loop: #7
Inner Exception: Error in RunProcess loop: #1
Inner Exception: Error in RunProcess loop: #11
======================
Total Processed: 15
Total Exceptions: 1
Done
```

As a reminder, this loop short-circuits and shows the contents of the `AggregateException`.

3. In the `RunProcess` method, remove the continuation on the `ForEachAsync` method call.

```
await Parallel.ForEachAsync(
    Enumerable.Range(1, numberOfItems),
    new ParallelOptions() { MaxDegreeOfParallelism = 10 },
    async (i, _) =>
    {
```

```
        Console.WriteLine($"Processing item: {i}");
        await Task.Delay(10); // simulate async task
        MightThrow.Throw(frequency, $"Error in RunProcess loop: #{i}");
        TotalProcessed++;
    });
```

4. Add a try/catch block to the body of the loop.

```
await Parallel.ForEachAsync(
    Enumerable.Range(1, numberOfItems),
    new ParallelOptions() { MaxDegreeOfParallelism = 10 },
    async (i, _) =>
    {
        try
        {
            Console.WriteLine($"Processing item: {i}");
            await Task.Delay(10); // simulate async task
            MightThrow.Throw(frequency, $"Error in RunProcess loop: #{i}");
            TotalProcessed++;
        }
        catch (Exception ex)
        {

        }
    });
```

5. Inside the catch block, increment the `TotalExceptions` field and show the exception.

```
catch (Exception ex)
{
    Interlocked.Increment(ref TotalExceptions);
    ExceptionReporter.ShowException(ex);
}
```

6. Build and run the application.

```
[some items omitted]
Processing item: 99
Processing item: 100
======================
Exception: Error in RunProcess loop: #97
======================
======================
Exception: Error in RunProcess loop: #93
```

```
======================
======================
Exception: Error in RunProcess loop: #96
======================
======================
Exception: Error in RunProcess loop: #95
======================
======================
Exception: Error in RunProcess loop: #94
======================
Total Processed: 79
Total Exceptions: 20
Done
```

Oops, the numbers do not add up to 100.

7. Use `Interlocked.Increment` for the `TotalProcessed` field.

```
try
{
    Console.WriteLine($"Processing item: {i}");
    await Task.Delay(10); // simulate async task
    MightThrow.Throw(frequency, $"Error in RunProcess loop: #{i}");
    Interlocked.Increment(ref TotalProcessed);
}
```

8. Build and run the application.

```
[some items omitted]
Processing item: 93
Processing item: 95
======================
Exception: Error in RunProcess loop: #81
======================
Processing item: 97
Processing item: 92
Processing item: 98
Processing item: 99
Processing item: 100
======================
Exception: Error in RunProcess loop: #91
======================
Total Processed: 87
Total Exceptions: 13
Done
```

Now the totals add up.

Also notice that the exceptions are reported as they happen. The exceptions are mixed in with the successful records. This is because we no longer rely on an `AggregateException` of a `Task`. Instead, we catch and handle the exceptions inside of the `Task`, so the `Task` is never in a faulted state.

Prevent Short-Circuiting: Step-By-Step

WhenAllExceptions

> Note: the finished code for this section is part of the "Completed" solution. This assumes that you completed the above steps for the "WhenAllExceptions" application. If not, you can start with the code in the "Intermediate" solution.

1. Open the "WhenAllExceptions/Program.cs" file.

2. Build and run the application.

```
[some items omitted]
Processing item: 64
Processing item: 50
Processing item: 51
Processing item: 97
Processing item: 98
Processing item: 69
Processing item: 70
======================
Aggregate Exception: Count 15
Inner Exception: Error in RunProcess loop: #97
Inner Exception: Error in RunProcess loop: #68
[some items omitted]
Inner Exception: Error in RunProcess loop: #88
Inner Exception: Error in RunProcess loop: #11
Inner Exception: Error in RunProcess loop: #57
Inner Exception: Error in RunProcess loop: #77
======================
Total Processed: 85
Total Exceptions: 1
Done
```

As a reminder, the loop completes, and we can get the details of the `AggregateException`. Now we'll change this to show the exceptions as they happen.

3. In the `RunProcess` method, remove the continuation from the `Task.WhenAll` method call.

```
await Task.WhenAll(allTasks);
```

4. Inside the `foreach` loop, add a try/catch block to the body of the task.

```
Task currentTask = Task.Run(async () =>
{
    try
    {
        Console.WriteLine($"Processing item: {i}");
        await Task.Delay(10); // simulate async task
        MightThrow.Throw(frequency, $"Error in RunProcess loop: #{i}");
        Interlocked.Increment(ref TotalProcessed);
    }
    catch (Exception ex)
    {

    }
});
```

5. In the `catch` block, add the code to increment the exception total and show the exception.

```
catch (Exception ex)
{
    Interlocked.Increment(ref TotalExceptions);
    ExceptionReporter.ShowException(ex);
}
```

6. Build and run the application.

```
======================
Exception: Error in RunProcess loop: #97
======================
======================
Exception: Error in RunProcess loop: #23
======================
======================
Exception: Error in RunProcess loop: #93
======================
Total Processed: 86
Total Exceptions: 14
Done
```

The application now processed the errors as they happen. Since throwing an exception is computationally "expensive", the exceptions in this code tend to sink to the bottom of the output. One reason this differs from the

`ForEachAsync` loop is that we do not have any limits set on degrees of parallelism, so the computer figures that out on its own.

Prevent Short-Circuiting: Step-By-Step

ChannelExceptions

> Note: the finished code for this section is part of the "Intermediate" solution.

1. Open the "ChannelExceptions/Program.cs" file.

2. Build and run the application.

```
[some items omitted]
Processing item: 11
Processing item: 12
Processing item: 13
Processing item: 15
Processing item: 14
Processing item: 16
Processing item: 17
Processing item: 18
[some items omitted]
Consuming item: 6
Consuming item: 10
Consuming item: 17
Consuming item: 14
Consuming item: 15
Consuming item: 13
Consuming item: 12
Consuming item: 11
Consuming item: 18
Consuming item: 16
======================
Exception: Error in Producer: #1
======================
Total Produced: 16
Total Consumed: 16
Total Exceptions: 1
Done
```

This application has both a Producer and a Consumer. The Producer uses a parallel loop and may throw exceptions. From this output, we can see that the Producer short-circuits on error.

Before look at the Producer, let's do a quick review of how channels work in this application.

3. Look at the `RunProcess` method.

```
private static async Task RunProcess(int numberOfItems)
{
    var channel = Channel.CreateUnbounded<int>();
    Task consumer = Consumer(channel.Reader);
    Task producer = Producer(numberOfItems, channel.Writer);

    await producer;
    await consumer;
}
```

First, we create a channel. Next, we start the Consumer and pass it the channel reader. The Consumer reads from the channel and outputs a message to the console. Then we start the Producer and pass it the channel writer. The Producer creates items (based on the `numberOfItems` parameter) and writes them to the channel. Finally, we wait for both the Producer and Consumer to complete asynchronously.

3. Look at the `Producer` method.

```
private static async Task Producer(int numberOfItems,
    ChannelWriter<int> writer)
{
    try
    {
        await Parallel.ForEachAsync(
            Enumerable.Range(1, numberOfItems),
            new ParallelOptions() { MaxDegreeOfParallelism = 10 },
            async (i, _) =>
            {
                Console.WriteLine($"Processing item: {i}");
                await Task.Delay(10); // simulate async task
                MightThrow.Throw(frequency, $"Error in Producer: #{i}");
                TotalProduced++;
                await writer.WriteAsync(i);
            });
    }
    finally
    {
        writer.Complete();
    }
}
```

This uses a `Parallel.ForEachAsync` loop very similar to the `ForEachAsyncExceptions` example. One main difference is that at the end of the loop body, we write the item to the channel.

The loop is wrapped in a try/finally block to ensure that the channel writer is marked "Complete". This lets the Consumer know that there will be no more items, and it can stop waiting.

We could approach the problem in a similar way to the earlier example. But since we use a `Channel` here, let's explore using channels to set up an error pipeline.

Here's the general idea: when we get an error, instead of throwing an exception, we put it onto a dedicated error channel. Then we can have another process that reads the exceptions off of the channel.

4. Add a new `ChannelWriter<Exception>` parameter to the Producer method.

```
private static async Task Producer(int numberOfItems,
    ChannelWriter<int> writer, ChannelWriter<Exception> errorWriter)
```

5. Add a try/catch block inside the body of the loop.

```
await Parallel.ForEachAsync(
    Enumerable.Range(1, numberOfItems),
    new ParallelOptions() { MaxDegreeOfParallelism = 10 },
    async (i, _) =>
    {
        try
        {
            Console.WriteLine($"Processing item: {i}");
            await Task.Delay(10); // simulate async task
            MightThrow.Throw(frequency, $"Error in Producer: #{i}");
            TotalProduced++;
            await writer.WriteAsync(i);
        }
        catch (Exception ex)
        {

        }
    });
```

6. Inside the `catch` block, write the exception to the error channel.

```
catch (Exception ex)
{
    await errorWriter.WriteAsync(ex);
}
```

7. The last step is mark the error channel `Complete`. Do this in the finally block (where the original channel is marked complete).

```
finally
{
    writer.Complete();
    errorWriter.Complete();
}
```

8. Create a new method called `ProcessErrors` to read the exceptions from the error channel. Use a `ChannelReader<Exception>` as a parameter.

```
private static async Task ProcessErrors(
    ChannelReader<Exception> errorReader)
{

}
```

9. Inside the method, use the `ReadAllAsync` method on the channel reader to set up an asynchronous `foreach` loop. If you need a hint, look at the `Consumer` method in this same class.

```
private static async Task ProcessErrors(
    ChannelReader<Exception> errorReader)
{
    await foreach (var ex in errorReader.ReadAllAsync())
    {

    }
}
```

10. Inside the loop, increment the total exceptions field and show the exception.

```
private static async Task ProcessErrors(
    ChannelReader<Exception> errorReader)
{
    await foreach (var ex in errorReader.ReadAllAsync())
    {
        Interlocked.Increment(ref TotalExceptions);
        ExceptionReporter.ShowException(ex);
    }
}
```

Next, we need to create the error channel and hook everything together.

11. In the `RunProcess` method, create a new `errorChannel`.

```
private static async Task RunProcess(int numberOfItems)
{
    var channel = Channel.CreateUnbounded<int>();
    var errorChannel = Channel.CreateUnbounded<Exception>();

    Task consumer = Consumer(channel.Reader);
    Task producer = Producer(numberOfItems, channel.Writer);

    await producer;
    await consumer;
}
```

12. Update the `Producer` method call to add the writer for the error channel.

```
Task producer = Producer(numberOfItems, channel.Writer,
                         errorChannel.Writer);
```

13. After waiting for the Producer and Consumer to finish, await the new `ProcessErrors` method.

```
await producer;
await consumer;

await ProcessErrors(errorChannel.Reader);
```

Here's the completed `RunProcess` method:

```
private static async Task RunProcess(int numberOfItems)
{
    var channel = Channel.CreateUnbounded<int>();
    var errorChannel = Channel.CreateUnbounded<Exception>();

    Task consumer = Consumer(channel.Reader);
    Task producer = Producer(numberOfItems, channel.Writer,
                             errorChannel.Writer);

    await producer;
    await consumer;

    await ProcessErrors(errorChannel.Reader);
}
```

14. For safety, update any increment operators (++) to use `Interlocked.Increment`.

```
Interlocked.Increment(ref TotalProduced);
```

```
Interlocked.Increment(ref TotalConsumed);
```

15. Build and run the application.

```
[some items omitted]
Processing item: 82
Processing item: 83
Processing item: 85
Processing item: 86
Consuming item: 80
Consuming item: 79
Consuming item: 77
Consuming item: 73
Consuming item: 78
Consuming item: 71
Consuming item: 72
Consuming item: 74
[some items omitted]
Processing item: 98
Processing item: 99
Processing item: 100
Consuming item: 96
Consuming item: 99
Consuming item: 97
Consuming item: 91
======================
Exception: Error in Producer: #15
======================
======================
Exception: Error in Producer: #16
======================
======================
Exception: Error in Producer: #36
======================
[some items omitted]
======================
Exception: Error in Producer: #81
======================
======================
```

```
Exception: Error in Producer: #95
=====================
Total Produced: 87
Total Consumed: 87
Total Exceptions: 13
Done
```

Since we read the error channel after the Producer and Consumer are complete, the errors show up at the end. If we wanted, we could start the error processor earlier and have it run concurrently. It all depends on our needs, including the amount of data processed and the number of errors likely to occur.

One Last Thing

ChannelExceptions

When looking at channels, we saw what happens if we get an exception in the Producer. But what happens if errors can occur in the Consumer as well? We'll explore that a little bit by adding exceptions to the consumer and using the existing error channel.

> Note: the finished code for this section is part of the "Completed" solution. This assumes that you completed the above steps for the "WhenAllExceptions" application. If not, you can start with the code in the "Intermediate" solution.

1. Open the "WhenAllExceptions/Program.cs" file.

2. Look at the `Consumer` method.

```
private static async Task Consumer(ChannelReader<int> reader)
{
    await foreach (int i in reader.ReadAllAsync())
    {
        Console.WriteLine($"Consuming item: {i}");
        Interlocked.Increment(ref TotalConsumed);
    }
}
```

This code reads items off of the main channel, outputs and message and increments the total consumed field. Let's see what happens if an exception occurs here.

3. Add a potential exception using the `MightThrow` class.

```
private static async Task Consumer(ChannelReader<int> reader)
{
    await foreach (int i in reader.ReadAllAsync())
    {
```

```
        Console.WriteLine($"Consuming item: {i}");
        MightThrow.Throw(frequency, $"Error in Consumer: #{i}");
        Interlocked.Increment(ref TotalConsumed);
    }
}
```

4. Build and run the application.

```
[some items omitted]
Processing item: 100
Processing item: 92
Processing item: 93
Processing item: 94
Processing item: 97
Processing item: 98
Processing item: 96
Processing item: 99
======================
Exception: Error in Consumer: #9
======================
Total Produced: 81
Total Consumed: 5
Total Exceptions: 1
Done
```

Now our output looks similar to before adding the error channel. This is because we do not have any exception handling in the Consumer. When the Consumer throws an exception, the `foreach` loop short-circuits. The exception bubbles up until it hits our original exception block, and only 1 exception is shown.

Let's add some error handling in the Consumer.

5. Add a try/catch block inside the `foreach` loop.

```
await foreach (int i in reader.ReadAllAsync())
{
    try
    {
        Console.WriteLine($"Consuming item: {i}");
        MightThrow.Throw(frequency, $"Error in Consumer: #{i}");
        Interlocked.Increment(ref TotalConsumed);
    }
    catch (Exception ex)
    {

    }
}
```

6. As in the Producer, we'll add a new parameter for an error channel and write to it in the try block.

```
private static async Task Consumer(ChannelReader<int> reader,
    ChannelWriter<Exception> errorWriter)
{
    await foreach (int i in reader.ReadAllAsync())
    {
        try
        {
            Console.WriteLine($"Consuming item: {i}");
            MightThrow.Throw(frequency, $"Error in Consumer: #{i}");
            Interlocked.Increment(ref TotalConsumed);
        }
        catch (Exception ex)
        {
            await errorWriter.WriteAsync(ex);
        }
    }
}
```

7. In the `RunProcess` method, update the `Consumer` method call with the new parameter.

```
Task consumer = Consumer(channel.Reader,
                         errorChannel.Writer);
Task producer = Producer(numberOfItems, channel.Writer,
                         errorChannel.Writer);
```

Since we use the error channel in multiple methods, we need to think about where we mark the channel `Complete`. Right now, it is in the `Producer` method, but we should move that to a better location to make sure the error channel is not closed too soon.

8. In the `Producer` method, remove the call to mark the error channel `Complete`.

```
finally
{
    writer.Complete();
    //errorWriter.Complete();
}
```

9. In the `RunProcess` method, mark the error channel `Complete` after both the Producer and Consumer have completed.

```
await producer;
await consumer;
errorChannel.Writer.Complete();

await ProcessErrors(errorChannel.Reader);
```

Here is the completed `RunProcess` method:

```
private static async Task RunProcess(int numberOfItems)
{
    var channel = Channel.CreateUnbounded<int>();
    var errorChannel = Channel.CreateUnbounded<Exception>();

    Task consumer = Consumer(channel.Reader,
                             errorChannel.Writer);
    Task producer = Producer(numberOfItems, channel.Writer,
                             errorChannel.Writer);

    await producer;
    await consumer;
    errorChannel.Writer.Complete();

    await ProcessErrors(errorChannel.Reader);
}
```

10. Build and run the application.

```
[some items omitted]
Consuming item: 87
Processing item: 96
Processing item: 97
Processing item: 98
Consuming item: 85
Consuming item: 84
Consuming item: 81
Consuming item: 97
Consuming item: 93
Consuming item: 95
Consuming item: 94
Consuming item: 100
======================
Exception: Error in Producer: #7
======================
[some items omitted]
======================
```

```
Exception: Error in Consumer: #52
======================
======================
Exception: Error in Producer: #70
======================
======================
Exception: Error in Producer: #78
======================
[some items omitted]
======================
Exception: Error in Consumer: #100
======================
Total Produced: 75
Total Consumed: 58
Total Exceptions: 42
Done
```

In the output, we see both "Error in Producer" and "Error in Consumer" messages. In addition, the totals add up correctly. The total produced is 75 (with 25 producer errors). This means *at most* there would be 75 items consumed (if there were no errors). Since there were errors in the consumer, only 58 items were consumed (meaning 17 consumer errors). The total errors is 42. And 42 and 58 correctly add up to 100.

The way that we set up error pipelines depends on the needs of the application. In this case, we shared an error channel for both the producer and consumer. But we could set up separate error channels. We could set up a retry functionality on one of the channels. The combinations are endless.

Conclusion

In this lab, we have seen how to change the default exception handling in parallel loops. We updated parallel loops so that they do not short-circuit when an exception occurs, and we also recorded all of the exceptions that were thrown in the parallel processes. We can use these examples to help us decide the best way to handle errors in the parallel loops in our own applications.

 # Lab 05 - Working with Exceptions in Parallel Loops
