Lab 01 - Recommended Practices and Continuations

The "Starter" folder contains the code files for this lab.

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

This solution is a console application that processes data from a text file. The "DataLoader" class creates a list of strings from the contents of the file ("data.txt"). These strings are passed to the "DataParser" class. The DataParser is responsible for parsing the strings into Person objects (which are then displayed to the console). Errors are logged to a file using the "FileLogger" class.

Run the application by pressing F5 (in either Visual Studio Code or Visual Studio 2019). This will build the application and give us some output that we can ignore for now.

In File Explorer, open the output folder: *[working_directory]/DataProcessor/bin/Debug/netcore3.1/*. Open the "data.txt" file. This contains a number of comma-separated value (CSV) records along with some invalid records. Run the application by double-clicking "DataProcessor.exe". The output will be as follows:

```
Successfully processed 9 records
John Koenig
Dylan Hunt
Turanga Leela
John Crichton
Dave Lister
Laura Roslin
John Sheridan
Dante Montana
Isaac Gampu
```

Now open the "log.txt" file in the same folder. It contains the errors and bad records.

```
==================
2020-10-31T17:32:42: Wrong number of fields in record - INVALID RECORD FORMAT
2020-10-31T17:32:42: Cannot parse Start Date field -
20,Check,Date,0/2//,9,{1} {0}
2020-10-31T17:32:42: Wrong number of fields in record - BAD RECORD
2020-10-31T17:32:42: Cannot parse Rating field -
21,Check,Rating,2014/05/03,a,
```

Lab Goals

Our goal is to change the logging function to an asynchronous operation. Our logger can then access files asynchronously. In addition, we will allow the async operations to propagate through the application.

Current Classes

DataProcessor.Library/ILogger
This is the interface for logging:

```
public interface ILogger
{
    void LogMessage(string message, string data);
}
```

DataProcessor.Library/FileLogger
Implements the ILogger interface to log messages to a file. Here is the "LogMessage" implementation:

```
public void LogMessage(string message, string data)
{
    using (var writer = new StreamWriter(logPath, true))
    {
        writer.WriteLine($"{DateTime.Now:s}: {message} - {data}");
    }
}
```

DataProcessor.Library/DataParser
Parses the data file and uses the "ILogger" interface to log any errors. The logger is passed in through the constructor.

```
public DataParser(ILogger logger)
{
    this.logger = logger ?? new NullLogger();
```

```
}
```
Note: if a logger is not passed in, then a "NullLogger" is used. This is a logger that does nothing.

The logger is used in the "ParseData" method of the "DataParser" class. We'll look at this method a bit more closely when making changes.

DataProcessor/Program

The program class has the entry point for the console application. Here is the "ProcessData" method:

```
static IReadOnlyCollection<Person> ProcessData()
{
    var loader = new DataLoader();
    IReadOnlyCollection<string> data = loader.LoadData();

    var logger = new FileLogger();
    var parser = new DataParser(logger);
    var records = parser.ParseData(data);
    return records;
}
```
This method loads the data from the text file, then creates the logger and data parser classes. It then calls "ParseData" and returns the records that come back.

Additional Project

Note: The "DataProcessor.Library.Tests" project is not used for this lab. It will be used in the next one (which is a continuation).

Hints

In ILogger.cs:

- Change the "LogMessage" interface to return "Task" instead of "void".

In FileLogger.cs:

- Change "LogMessage" to an asynchronous method to match the interface.
- The "StreamWriter" class has an asynchronous "WriteLineAsync" method that can be used.

In NullLogger.cs:

- Update the class to satisfy the updated "ILogger" interface.
- Do not return "null" from the method. There is a static property on Task that can be used instead.

In DataParser.cs:

- Await the "logger.LogMessage" calls in the "ParseData" method.
- Let the async bubble up through this method.

In Program.cs:

- Update the "ProcessData" method as needed. (That's all of the hints you get for this one).

Additional Hints:

- Since we're dealing with library code, think about where to use ".ConfigureAwait(false)" in the code.

If you need more assistance, step-by-step instructions are included below. Otherwise, **STOP READING NOW**

Updating to Asynchronous Methods: Step-By-Step

1. Update the ILogger interface to use an asynchronous method.

In the "ILogger.cs" file, change the LogMessage method so that it returns "Task" instead of "void".

Note: You will need to add a using statement for "System.Threading.Tasks". As a shortcut, click on "Task", and press "Ctrl+." in Visual Studio. This will offer to add the using statement for you.

```
using System.Threading.Tasks;

namespace DataProcessor.Library
{
    public interface ILogger
```

```
    {
        Task LogMessage(string message, string data);
    }
}
```

2. Update the FileLogger to match the interface.

In the "FileLogger.cs" file, change the return type of the LogMessage method from "void" to "Task". (You'll have to add a "using" statement here as well.)

Change the body of the method to use "WriteLineAsync" instead of "WriteLine".

Since "WriteLineAsync" returns a Task, it is really tempting to directly return it. Here's what that looks like:

```
public Task LogMessage(string message, string data)
{
    using (var writer = new StreamWriter(logPath, true))
    {
        // THIS WON'T WORK AS EXPECTED
        return writer.WriteLineAsync($"{DateTime.Now:s}: {message} -
{data}");
    }
}
```

Caution

Even though this compiles, it will not work as expected. This is because of the "using" statement on the StreamWriter class. By using this code, the StreamWriter will get disposed before the "WriteLineAsync" method is called. This results in a runtime exception.

Rather than returning the task directly, await it. (Don't forget to add the "async" modifier to the method.)

```
public async Task LogMessage(string message, string data)
{
    using (var writer = new StreamWriter(logPath, true))
    {
        await writer.WriteLineAsync($"{DateTime.Now:s}: {message} - {data}");
    }
}
```

3. Update the NullLogger to match the interface.

Change the return type of the LogMessage method from "void" to "Task". (You'll need to add the relevant using statement. Since we've done this several times, I'll stop reminding you about it.)

```
public class NullLogger : ILogger
{
    public Task LogMessage(string message, string data)
    {
        // AVOID RETURNING NULL
        return null;
    }
}
```

Caution

It's tempting to return "null" from this method, particularly since the logger intentionally does nothing. But this is a bad practice can cause issues for callers.

We could create a new Task with a TaskFactory, but a better solution is to use the static "CompletedTask" property on the Task type.

```
public class NullLogger : ILogger
{
    public Task LogMessage(string message, string data)
    {
        return Task.CompletedTask;
    }
}
```

This lets the calling code know that there is a completed task (and one that completed successfully). This fits in well with callers.

Now that the loggers are all updated, it's time to move on to where the loggers are used.

4. Update the DataParser to await the async method of the logger.

In the "DataParser.cs" file, locate the ParseData method. This has a number of calls to "logger.LogMessage".

If we leave the calls the way they are, the code will still work. The "LogMessage" methods will be called on the logger, but we won't know if

they complete successfully. By awaiting the methods, we know when they are finished, and we can also deal with exceptions if there are problems.

Here's one of the calls (updated with await):

```
    if (fields.Length != 6)
    {
        await logger.LogMessage("Wrong number of fields in record", record);
        continue;
    }
```

Since we are using "await", we need to mark the method as "async" and change the return type to Task:

```
public async Task<IReadOnlyCollection<Person>> ParseData(IEnumerable<string>
data)
```

With the return type change, we are letting the asynchronous code bubble up through our application. This is exactly what we want.

Here is the completed method with all of the awaits filled in:

```
public async Task<IReadOnlyCollection<Person>> ParseData(IEnumerable<string>
data)
{
    var processedRecords = new List<Person>();
    foreach (var record in data)
    {
        var fields = record.Split(',');
        if (fields.Length != 6)
        {
            await logger.LogMessage("Wrong number of fields in record",
record);
            continue;
        }

        int id;
        if (!Int32.TryParse(fields[0], out id))
        {
            await logger.LogMessage("Cannot parse Id field", record);
            continue;
        }

        DateTime startDate;
        if (!DateTime.TryParse(fields[3], out startDate))
        {
            await logger.LogMessage("Cannot parse Start Date field", record);
            continue;
        }
```

```
        int rating;
        if (!Int32.TryParse(fields[4], out rating))
        {
            await logger.LogMessage("Cannot parse Rating field", record);
            continue;
        }

        var person = new Person()
        {
            Id = id,
            GivenName = fields[1],
            FamilyName = fields[2],
            StartDate = startDate,
            Rating = rating,
            FormatString = fields[5]
        };
        // Successfully parsed record
        processedRecords.Add(person);
    }
    return processedRecords;
}
```

5. Update the Program, starting with the ProcessData method.

In the "Program.cs" file, update the ProcessData method by awaiting the "ParseData" method that we just updated. (Don't forget about changing the return type.)

```
static async Task<IReadOnlyCollection<Person>> ProcessData()
{
    var loader = new DataLoader();
    IReadOnlyCollection<string> data = loader.LoadData();

    var logger = new FileLogger();
    var parser = new DataParser(logger);
    var records = await parser.ParseData(data);
    return records;
}
```

6. Update the Main method.

Add an await to the call to "ProcessData". You'll need to change the return type on "Main", and that's fine (async Main methods were added to C# a few language updates ago).

```
static async Task Main(string[] args)
{
    var records = await ProcessData();

    Console.WriteLine($"Successfully processed {records.Count()} records");
    foreach(var person in records)
    {
        Console.WriteLine(person);
    }
    Console.WriteLine("Press Enter to continue...");
    Console.ReadLine();
}
```

7. Build and run the application.

Note: when you build the application, there will be some warnings about the unit test project. Don't worry about that right now, we'll get to that in the next lab.

Use F5 to build and run the application. The output should be the same as before:

```
Successfully processed 9 records
John Koenig
Dylan Hunt
Turanga Leela
John Crichton
Dave Lister
Laura Roslin
John Sheridan
Dante Montana
Isaac Gampu
```

8. Check to make sure the logger is still logging to the "log.txt" file.

In File Explorer, open the output folder: *[working_directory]/DataProcessor/bin/Debug/netcore3.1/*.

Open the "log.txt" file in the same folder. It contains the errors and bad records. The logger appends to the file, so you should see recent time stamps at the end of the file.

```
==================
2020-10-31T17:32:42: Wrong number of fields in record - INVALID RECORD FORMAT
```

```
2020-10-31T17:32:42: Cannot parse Start Date field -
20,Check,Date,0/2//,9,{1} {0}
2020-10-31T17:32:42: Wrong number of fields in record - BAD RECORD
2020-10-31T17:32:42: Cannot parse Rating field -
21,Check,Rating,2014/05/03,a,
==================
2020-10-31T18:43:08: Wrong number of fields in record - INVALID RECORD FORMAT
2020-10-31T18:43:08: Cannot parse Start Date field -
20,Check,Date,0/2//,9,{1} {0}
2020-10-31T18:43:08: Wrong number of fields in record - BAD RECORD
2020-10-31T18:43:08: Cannot parse Rating field -
21,Check,Rating,2014/05/03,a,
```

Optimizing Using .ConfigureAwait() - Step-by-Step

As a reminder, when we await a task, the current context is saved. Then after the async code completes, we're returned to that context to continue running.

Our code works (as we can see), but it uses unneeded resources. To optimize our code, we should add "ConfigureAwait(false)" wherever we are awaiting inside the library.

We'll start at the bottom and work our way up.

1. FileLogger class / LogMessage method

Add ".ConfigureAwait(false)" to the "WriteLineAsync" method call.

```
public async Task LogMessage(string message, string data)
{
    using (var writer = new StreamWriter(logPath, true))
    {
        await writer.WriteLineAsync($"{DateTime.Now:s}: {message} - {data}")
            .ConfigureAwait(false);
    }
}
```

2. DataParser class / ParseData method

Add ".ConfigureAwait(false)" to all of the "LogMessage" method calls.

Note: Since the code lines can get pretty long, I often put the ".ConfigureAwait(false)" on the next line.

```
public async Task<IReadOnlyCollection<Person>> ParseData(IEnumerable<string>
data)
{
    var processedRecords = new List<Person>();
    foreach (var record in data)
    {
        var fields = record.Split(',');
        if (fields.Length != 6)
        {
            await logger.LogMessage("Wrong number of fields in record",
record)
                .ConfigureAwait(false);
            continue;
        }

        int id;
        if (!Int32.TryParse(fields[0], out id))
        {
            await logger.LogMessage("Cannot parse Id field", record)
                .ConfigureAwait(false);
            continue;
        }

        DateTime startDate;
        if (!DateTime.TryParse(fields[3], out startDate))
        {
            await logger.LogMessage("Cannot parse Start Date field", record)
                .ConfigureAwait(false);
            continue;
        }

        int rating;
        if (!Int32.TryParse(fields[4], out rating))
        {
            await logger.LogMessage("Cannot parse Rating field", record)
                .ConfigureAwait(false);
            continue;
        }

        var person = new Person()
        {
            Id = id,
            GivenName = fields[1],
            FamilyName = fields[2],
            StartDate = startDate,
            Rating = rating,
```

```
            FormatString = fields[5]
        };
        // Successfully parsed record
        processedRecords.Add(person);
    }
    return processedRecords;
}
```

3. Program class / ProcessData method

Since we are not worried about a specific context for our console application, we can add ".ConfigureAwait(false)" there as well.

```
static async Task<IReadOnlyCollection<Person>> ProcessData()
{
    var loader = new DataLoader();
    IReadOnlyCollection<string> data = loader.LoadData();

    var logger = new FileLogger();
    var parser = new DataParser(logger);
    var records = await parser.ParseData(data).ConfigureAwait(false);
    return records;
}
```

4. Program class / Main method

And the last stop is the Main method.

```
static async Task Main(string[] args)
{
    var records = await ProcessData().ConfigureAwait(false);

    Console.WriteLine($"Successfully processed {records.Count()} records");
    foreach(var person in records)
    {
        Console.WriteLine(person);
    }
    Console.WriteLine("Press Enter to continue...");
    Console.ReadLine();
}
```

Conclusion

So this has shown us how to add asynchronous methods to our application, allow them to propagate through, and also use ".ConfigureAwait(false)" to optimize our code.

End of Lab 01 - Recommended Practices and Continuations