Using async and await in C#

Step 1: Understanding Asynchronous Programming

Asynchronous programming allows your application to perform tasks without waiting for others to complete, keeping the program responsive. You'll create a small application using the Visual Studio Code console application you created at the start of the course. Your application will run multiple asynchronous methods.

Remove any existing code in the Program.cs file of your console application and create all the code in each step in that file.

Step 2: Creating an Asynchronous Method

Create a simple asynchronous method that simulates a task taking time to complete, like downloading data.

Instructions:

1. In the Program.cs file, create a class called Program.
2. Inside the Program class, create a method named DownloadDataAsync.
3. Use the async keyword to make the method asynchronous.
4. Inside the method, use await Task.Delay to simulate a delay.
5. Print a message before and after the delay to show when the method starts and ends.

Step 3: Running Multiple Asynchronous Methods

Create a Main method and create a variable assigned to the async method to simulate the download delay.

Instructions:

1. Below the DownloadDataAsync method, create a Main method.
2. In the Main method, create an instance of the Program class.
3. Call the DownloadDataAsync method using await.
4. To check your answer, run the Visual Studio Code console application. If you receive an error when you run the code, go to the reading on the next page to compare your code to the correct answer.

Step 4: Running Multiple Asynchronous Methods

Run multiple asynchronous methods in parallel to see how they can execute simultaneously.

Instructions:

1. Below the DownloadDataAsync method, create a second method named DownloadDataAsync2.
2. Update the Main method to use Task.WhenAll to run DownloadDataAsync and DownloadDataAsync2 in parallel.
3. Observe how both methods run at the same time.
4. To check your answer, run the Visual Studio Code console application. If you receive an error when you run the code, go to the reading on the next page to compare your code to the correct answer.

Step 5: Handling Exceptions in Asynchronous Methods

Add error handling to asynchronous methods using a try-catch block.

Instructions:

1. Modify the DownloadDataAsync method to include a try-catch block.
2. Simulate an error by adding code that throws an exception.
3. Catch the exception and display an error message.
4. To check your answer, run the Visual Studio Code console application. If you receive an error when you run the code, go to the reading on the next page to compare your code to the correct answer.

Code:

```
namespace UsingAsyncAwait
{
    class Program
    {
        // Step 2: First asynchronous method
        public async Task DownloadDataAsync()
        {
            try
            {
                Console.WriteLine("DownloadDataAsync started ...");
                await Task.Delay(3000); // Simulate 3-second delay
                throw new Exception("Simulated download error.");
```

```
                Console.WriteLine("DownloadDataAsync completed.");
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error in DownloadDataAsync:
{ex.Message}");
            }
        }

        // Step 4: Second asynchronous method
        public async Task DownloadDataAsync2()
        {
            Console.WriteLine("DownloadDataAsync2 started...");
            await Task.Delay(2000); // Simulate 2-second delay
            Console.WriteLine("DownloadDataAsync2 completed");
        }

        // Step 3 + 4: Main method (entry point)
        static async Task Main(string[] args)
        {
            Program program = new();

            // Step 4: Run both async methods in parallel
            Task task1 = program.DownloadDataAsync();
            Task task2 = program.DownloadDataAsync2();

            await Task.WhenAll(task1, task2);
            Console.WriteLine("All downloads completed.");

        }
    }
}
```