Question Answering Assignment

1. What is the purpose of async/await keywords ?

The purpose of the `async` and `await` keywords in C# is to simplify asynchronous programming by allowing developers to write asynchronous code that looks and behaves like synchronous code, improving application responsiveness and resource utilization.

Detailed Explanation:

- **Async Keyword**
  The `async` keyword is used to mark a method, lambda expression, or anonymous method as asynchronous. An `async` method typically returns a `Task` or `Task<TResult>` (except in event handlers which may return void). Marking a method as `async` enables the use of the `await` keyword inside its body.

- **Await Keyword**
  The `await` keyword is used inside an async method to asynchronously wait for a task to complete without blocking the calling thread. When the program reaches an `await`, it pauses the execution of the current method and returns control to the caller, allowing other work to proceed. Once the awaited task finishes, execution resumes at the point after the `await`.

Purpose and Benefits:

- Non-blocking execution: Avoids blocking the main thread, especially critical in UI and web applications to keep them responsive during long-running operations like I/O or network calls.

- Simplified asynchronous code: Async/await enables writing asynchronous code in a linear, easy-to-read manner, eliminating the complexity of callbacks or manual thread management.

- Efficient resource usage: Frees up threads during waits, enabling better scalability in server applications.

- Improved responsiveness: UI applications remain responsive to user input while awaiting tasks such as file reads, web requests, or database queries.
- Easier error handling: Exceptions within async methods propagate naturally as in synchronous methods, making try-catch straightforward.

How Async/Await Work Together

1. An async method starts running synchronously on the calling thread.
2. When it hits an await on a task that is not finished, it returns control to the caller.
3. The calling thread proceeds with other work.
4. When the awaited task completes, the async method resumes execution after the await.

Example

```
public async Task<string> FetchDataAsync(string url)
{
    using HttpClient client = new HttpClient();
    string result = await client.GetStringAsync(url);  //
asynchronously waits here without blocking
    return result;
}
```

In this example, GetStringAsync executes asynchronously, allowing the calling thread (e.g., UI thread) to remain responsive.

async and await make asynchronous programming more accessible and maintainable, improving performance and user experience across many real-world scenarios such as network calls, file I/O, and parallel processing