Threads, Tasks, Synchronous and Asynchronous Programming

3

What is `async` and `await`?

The `async` keyword is used to mark methods as asynchronous, indicating that they can be paused and resumed. The `await` keyword is used to asynchronously wait for a task to complete without blocking the thread. Together, `async` and `await` make asynchronous programming more readable and maintainable by avoiding callback hell and making asynchronous code look similar to synchronous code.

Deep Dive into `async` and `await`

- **Asynchronous Programming**: Asynchronous programming is crucial for applications that perform time-consuming operations, such as I/O-bound tasks (e.g., web requests, file operations) or CPU-bound tasks, where the main thread can continue executing other code while waiting for a task to complete.
- **How `async` and `await` Work**: The `async` keyword marks a method as asynchronous, meaning it can use the `await` keyword to pause its execution until a `Task` is complete. The `await` keyword makes the asynchronous method non-blocking, allowing the current thread to continue executing other code.

Example: Using `async` and `await` in C#

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
class Program
{
        static async Task Main()
        {
        string url = "https://jsonplaceholder.typicode.com/posts/1";
        string result = await FetchDataAsync(url);
        Console.WriteLine(result);
        }
        static async Task<string> FetchDataAsync(string url)
        {
        using (HttpClient client = new HttpClient())
        {
        string response = await client.GetStringAsync(url);
        return response;
        }
        }
}
```

In this example, `FetchDataAsync` is marked as `async`, and `await` is used to asynchronously get the response from a URL. This means that while the `HttpClient` is waiting for a response, the program can continue executing other code, enhancing the overall efficiency.

Avoiding Deadlocks

One of the challenges of using `async` and `await` is avoiding deadlocks, especially in applications with a single-threaded context, such as GUI applications.

Example: Avoiding Deadlocks

```
using System;

using System.Threading.Tasks;

class Program

{

        static async Task Main()

        {

        await Task.Delay(1000);

        Console.WriteLine("Task completed without deadlock");

        }

}
```

In this example, `await` allows the task to complete without blocking the main thread, thus avoiding a deadlock.

Error Handling in Asynchronous Code

Handling errors in asynchronous code is similar to synchronous code. You can use `try-catch` blocks around `await` calls to handle exceptions.

Example: Error Handling with `async` and `await`

```
using System;

using System.Net.Http;

using System.Threading.Tasks;

class Program

{         static async Task Main()

          {

          try

          {

          string url = "https://jsonplaceholder.typicode.com/invalid-url";

          string result = await FetchDataAsync(url);

          Console.WriteLine(result);

          }

          catch (HttpRequestException ex)

          {

          Console.WriteLine($"Error fetching data: {ex.Message}");

          }

          }
```

```
    static async Task<string> FetchDataAsync(string url)
    {
    using (HttpClient client = new HttpClient())
    {
        return await client.GetStringAsync(url);
    }
    }
}
```

In this example, if the `HttpClient` encounters an invalid URL, the `HttpRequestException` is caught, and an error message is printed.

Tasks to Practice with `async` and `await`

1. **Task**: Create an asynchronous method that fetches data from a URL and prints the first 50 characters of the response.
    - **Solution**:

```
using System;

using System.Net.Http;

using System.Threading.Tasks;

class Program

{       static async Task Main()

        {       string url = "https://jsonplaceholder.typicode.com/posts/1";

        await FetchAndPrintDataAsync(url);

        }

        static async Task FetchAndPrintDataAsync(string url)

        {

        using (HttpClient client = new HttpClient())

        {       string response = await client.GetStringAsync(url);

        Console.WriteLine(response.Substring(0, 50));       }       }

}
```

Task: Create an asynchronous method that waits for 3 seconds and then prints "Waited for 3 seconds".

- **Solution**:

```
using System;
using System.Threading.Tasks;
class Program
{
        static async Task Main()
        {
        await WaitAndPrintAsync();
        }
        static async Task WaitAndPrintAsync()
        {
        await Task.Delay(3000);
        Console.WriteLine("Waited for 3 seconds");
        }
}
```

Task: Create an asynchronous method that throws an exception after a delay of 1 second. Catch the exception and print an error message.

- **Solution**:

```
using System;
using System.Threading.Tasks;
class Program
{       static async Task Main()
        {
        try
        {       await ThrowExceptionAsync();            }
        catch (Exception ex)
        {
        Console.WriteLine($"Caught exception: {ex.Message}");           }       }
        static async Task ThrowExceptionAsync()
        {       await Task.Delay(1000);
        throw new InvalidOperationException("An error occurred asynchronously");       }
}
```

- **Threads**: Basic unit of execution, managed by the OS. Useful for CPU-intensive operations but can be resource-heavy.
- **Tasks**: High-level abstraction for managing work asynchronously, using thread-pool threads. More lightweight compared to manually managing threads.
- **Synchronous vs. Asynchronous**: Synchronous operations block the program until they complete. Asynchronous operations allow other parts of the program to continue running, improving efficiency.
- `async`/`await`: Make asynchronous code easier to write and understand. Used for non-blocking operations like I/O.