AsyncAwait

C#

Oh God, Why?

AsyncAwait

Async keyword

```
public async Task DoSomethingAsync()
{
  // In the Real World, we would actually do something...
  // For this example, we're just going to (asynchronously) wait 100ms.
  await Task.Delay(100);
}
```

Return value

```
public async Task<int> CalculateAnswer()
{
  await Task.Delay(100); // (Probably should be longer...)

  // Return a type of "int", not "Task<int>"
  return 42;
}
```

MathAnal

```
internal sealed class Type1
{
}

private static async Task<Type1> Method1Async()
{
    /* Асинхронная операция, возвращающая объект Type1 */
}

private static async Task<string> MyMethodAsync(int argument)
{
    var local = argument;

    var result1 = await Method1Async();

    return "Done";
}
```

MathAnal

```
private static async Task<string>
        MyMethodAsync(int argument)
{
    var local = argument;
    var result1 =
        await Method1Async();
    return "Done";
}
```

```
[DebuggerStepThrough, AsyncStateMachine(typeof(StateMachine))]
private static Task<string> MyMethodAsyncGenerated(int argument)
{
    // Создание экземпляра конечного автомата и его инициализация
    var stateMachine = new StateMachine()
    {
        // Создание построителя, возвращающего Task<string>.
        // Конечный автомат обращается к построителю для назначения
        // завершения задания или выдачи исключения.
        m_builder = AsyncTaskMethodBuilder<string>.Create(),
        m_state = 1,
        // инициализация местонахождения
        m_argument = argument // Копирование аргументов в поля конечного автомата
    };
    // Начало выполнения конечного автомата.
    stateMachine.m_builder.Start(ref stateMachine);
    return stateMachine.m_builder.Task; // Возвращение задания конечного автомата
}
```

MathAnal

```
private static async Task<string>
        MyMethodAsync(int argument)
{
    var local = argument;
    var result1 =
        await Method1Async();
    return "Done";
}
```

```
// Структура конечного автомата
[CompilerGenerated, StructLayout(LayoutKind.Auto)]
private struct StateMachine : IAsyncStateMachine
{
    // Поля для построителя конечного автомата (Task) и его местонахождения
    public AsyncTaskMethodBuilder<string> m_builder;

    public int m_state;

    // Аргумент и локальные переменные становятся полями:
    public int m_argument, m_local;
    public Type1 m_resultType1;

    // Одно поле на каждый тип Awaiter.
    private TaskAwaiter<Type1> m_awaiterType1;

    // Сам конечный автомат
    void IAsyncStateMachine.MoveNext()
    {
        string result = null; // Результат Task
        // Вставленный компилятором блок try гарантирует
        // завершение задания конечного автомата
        try...

        // Исключения нет: задание конечного автомата завершается с результатом
        m_builder.SetResult(result);
    }
}
```

MathAnal

```
try
{
    if (m_state == 1)
    {
        // Если метод конечного автомата выполняется впервые
        m_local = m_argument; // Выполнить начало исходного метода
    }

    TaskAwaiter<Type1> awaiterType1;
    switch (m_state)
    {
        case 1:
            // Начало исполнения кода
            // вызвать Method1Async и получить его объект ожидания
            awaiterType1 = Method1Async().GetAwaiter();
            if (!awaiterType1.IsCompleted)
            {
                m_state = 0; // 'Method1Async' завершается асинхронно
                m_awaiterType1 = awaiterType1; // Сохранить объект
                // ожидания до возвращения
                // Приказать объекту ожидания вызвать MoveNext
                // после завершения операции
                m_builder.AwaitUnsafeOnCompleted(ref awaiterType1, ref this);
                // Предыдущая строка вызывает метод OnCompleted
                // объекта awaiterType1, что приводит к вызову
                // ContinueWith(t => MoveNext()) для Task.
                // При завершении Task ContinueWith вызывает MoveNext
                return; // Поток возвращает
            } // управление вызывающей стороне

            // 'Method1Async' завершается синхронно.
            break;
        case 0: // 'Method1Async' завершается асинхронно
            awaiterType1 = m_awaiterType1; // Восстановление последнего
            break; // объекта ожидания
    }

    // После await сохраняем результат
    m_resultType1 = awaiterType1.GetResult();

    result = "Done"; // То, что в конечном итоге должна вернуть асинхронная функция.
}
catch (Exception exception)
{
    // Необработанное исключение: задание конечного автомата
    // завершается с исключением.
    m_builder.SetException(exception);
    return;
}
```

MathAnal

```
private static async Task<string>
        MyMethodAsync(int argument)
{
    var local = argument;
    var result1 =
        await Method1Async();
    return "Done";
}
```

```
if (m_state == 1)
{
    // Если метод конечного автомата выполняется впервые
    m_local = m_argument; // Выполнить начало исходного метода
}
```

MathAnal

```
private static async Task<string>
        MyMethodAsync(int argument)
{
    var local = argument;
    var result1 =
        await Method1Async();
    return "Done";
}
```

```
TaskAwaiter<Type1> awaiterType1;
switch (m_state)
{
    case 1:
        // Начало исполнения кода
        // вызвать Method1Async и получить его объект ожидания
        awaiterType1 = Method1Async().GetAwaiter();
        if (!awaiterType1.IsCompleted)
        {
            m_state = 0; // 'Method1Async' завершается асинхронно
            m_awaiterType1 = awaiterType1; // Сохранить объект
            // ожидания до возвращения
            // Приказать объекту ожидания вызвать MoveNext
            // после завершения операции
            m_builder.AwaitUnsafeOnCompleted(ref awaiterType1, ref this);
            // Предыдущая строка вызывает метод OnCompleted
            // объекта awaiterType1, что приводит к вызову
            // ContinueWith(t => MoveNext()) для Task.
            // При завершении Task ContinueWith вызывает MoveNext
            return; // Поток возвращает
        } // управление вызывающей стороне

        // 'Method1Async' завершается синхронно.
        break;
    case 0: // 'Method1Async' завершается асинхронно
        awaiterType1 = m_awaiterType1; // Восстановление последнего
        break; // объекта ожидания
}
```

MathAnal

```
private static async Task<string>
        MyMethodAsync(int argument)
{
    var local = argument;
    var result1 =
        await Method1Async();
    return "Done";
}
```

```
// После await сохраняем результат
m_resultType1 = awaiterType1.GetResult();

// То, что в конечном итоге должна вернуть асинхронная функция.
result = "Done";
```

```
catch (Exception exception)
{
    // Необработанное исключение: задание конечного автомата
    // завершается с исключением.
    m_builder.SetException(exception);
    return;
}

// Исключения нет: задание конечного автомата завершается с результатом
m_builder.SetResult(result);
```

Context

- UI context.
- ASP.NET request context.
- thread pool context.

Context

```
// WinForms example (it works exactly the same for WPF).
private async void DownloadFileButton_Click(object sender, EventArgs e)
{
  // Since we asynchronously wait, the UI thread is not blocked by the file download.
  await DownloadFileAsync(fileNameTextBox.Text);

  // Since we resume on the UI context, we can directly access UI elements.
  resultTextBox.Text = "File downloaded!";
}

// ASP.NET example
protected async void MyButton_Click(object sender, EventArgs e)
{
  // Since we asynchronously wait, the ASP.NET thread is not blocked by the file download.
  // This allows the thread to handle other requests while we're waiting.
  await DownloadFileAsync(...);

  // Since we resume on the ASP.NET context, we can access the current request.
  // We may actually be on another *thread*, but we have the same ASP.NET request context.
  Response.Write("File downloaded!");
}
```

ConfigureAwait

```
private async Task DownloadFileAsync(string fileName)
{
  // Use HttpClient or whatever to download the file contents.
  var fileContents = await DownloadFileContentsAsync(fileName).ConfigureAwait(false);

  // Note that because of the ConfigureAwait(false), we are not on the original context here.
  // Instead, we're running on the thread pool.

  // Write the file contents out to a disk file.
  await WriteToDiskAsync(fileName, fileContents).ConfigureAwait(false);

  // The second call to ConfigureAwait(false) is not *required*, but it is Good Practice.
}

// WinForms example (it works exactly the same for WPF).
private async void DownloadFileButton_Click(object sender, EventArgs e)
{
  // Since we asynchronously wait, the UI thread is not blocked by the file download.
  await DownloadFileAsync(fileNameTextBox.Text);

  // Since we resume on the UI context, we can directly access UI elements.
  resultTextBox.Text = "File downloaded!";
}
```

ConfigureAwait

```
public async Task DoOperationsConcurrentlyAsync()
{
  Task[] tasks = new Task[3];
  tasks[0] = DoOperation0Async();
  tasks[1] = DoOperation1Async();
  tasks[2] = DoOperation2Async();

  // At this point, all three tasks are running at the same time.

  // Now, we await them all.
  await Task.WhenAll(tasks);
}
```

```
public async Task<int> GetFirstToRespondAsync()
{
  // Call two web services; take the first response.
  Task<int>[] tasks = new[] { WebService1Async(), WebService2Async() };

  // Await for the first one to respond.
  Task<int> firstTask = await Task.WhenAny(tasks);

  // Return the result.
  return await firstTask;
}
```

Dummy Deadlock

```
// My "library" method.
public static async Task<JObject> GetJsonAsync(Uri uri)
{
  using (var client = new HttpClient())
  {
    var jsonString = await client.GetStringAsync(uri);
    return JObject.Parse(jsonString);
  }
}

// My "top-level" method.
public class MyController : ApiController
{
  public string Get()
  {
    var jsonTask = GetJsonAsync(...);
    return jsonTask.Result.ToString();
  }
}
```

Dummy Deadlock

```
public static async Task<JObject> GetJsonAsync(Uri uri)
{
  using (var client = new HttpClient())
  {
    var jsonString = await client.GetStringAsync(uri).ConfigureAwait(false);
    return JObject.Parse(jsonString);
  }
}
```

Dummy Deadlock

```
public class MyController : ApiController
{
  public async Task<string> Get()
  {
    var json = await GetJsonAsync(...);
    return json.ToString();
  }
}
```

Dummy Deadlock

```
public static async Task<JObject> GetJsonAsync(Uri uri)
{
  using (var client = new HttpClient())
  {
    var jsonString = await client.GetStringAsync(uri).ConfigureAwait(false);
    return JObject.Parse(jsonString);
  }
}
```

```
public class MyController : ApiController
{
  public async Task<string> Get()
  {
    var json = await GetJsonAsync(...);
    return json.ToString();
  }
}
```