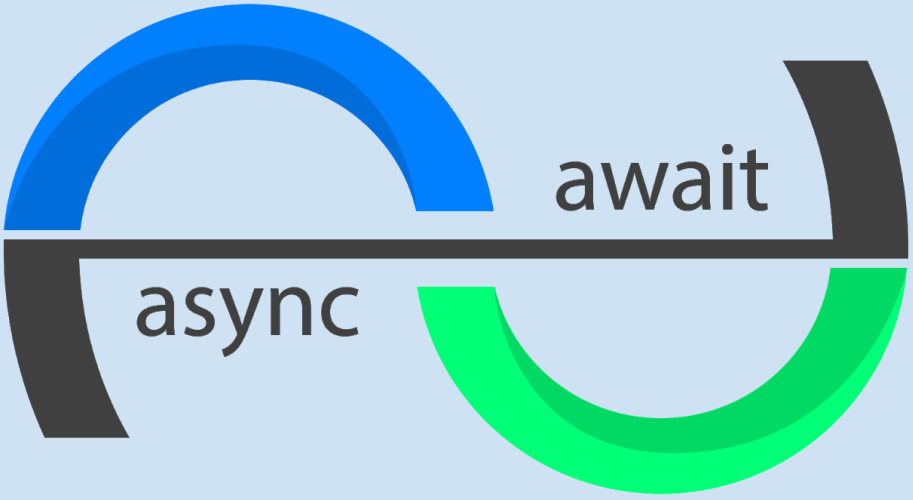

Await Extensions Documentation

What is async/await?

Async/await is a syntax sugar, which simplify working with asynchronous code. When you need wait while some **Task** (or object that implement **GetAwaiter**) completes, you just write **await** keyword before it and all code below this keyword runs (in case with Unity - on main thread) after the task is completed. Method which contains **await** keyword need mark as **async**.

Async/await mechanism was integrated in C# 5.0 and .NET 4.5.
For details about how use async/await, please visit microsoft site.

Unity and async/await

Unity officially support C# 5.0 and .NET 4.5 start from 2018 version (really it support C# 6.0 and .NET 4.6 too. Also you need switch project scripting runtime version to .NET 4.6 in “Player Settings”). It means that you can easy use this futures. Look at example below, it demonstrates how you can wait for 1 realtime second in **Awake** method with async/await:

```
private async void Awake()
{
    await Task.Delay(1000);
    Log("Awaited");
}
```

Log(“Awaited”); will be called from main thread after waiting 1000 milliseconds.

You can use all C# 6.0 and .NET 4.6 features in Unity, but out of box Unity does not support awaiting for different Unity types, for example this code will not work:

```
private async void Awake()
{
    await new WaitForSeconds(1f);
    Log("Awaited");
}
```

It is because Unity does not know how await WaitForSeconds object. If we “tell” Unity “how” - it can. To “tell” Unity “how” await WaitForSeconds object we need create **GetAwaiter** extension method for WaitForSeconds class, which return some object which know how await WaitForSeconds object and which used by await keyword.

If you do not know “awaitable/awaiter” pattern, please visit microsoft site.

All time-dependent operations (WaitForFixedUpdate, WaitUntil, awaiting coroutines, awaiting WWW request, and other) can be easily extended with method described above and then used with **await** keyword.

What “Await Extensions” exactly do?

It extends many Unity classes for work with **await** keyword. Look at example below, it shows all variants of using await with “Await Extensions” asset:

```
private async void Awake()
{
    // Await task.
    await Task.Delay(1000);

    // Await Unity Wait classes.
    await new WaitForEndOfFrame();
    await new WaitForFixedUpdate();
    await new WaitForSeconds(1f);
    await new WaitForSecondsRealtime(1f);
    await new WaitUntil(() => { return true; });
    await new WaitWhile(() => { return false; });

    // Await IEnumerator. The Unity coroutine engine will be used,
    // this means that Wait classes will be processed correctly.
    await EnumerateSomething();

    // Await downloading.
    WWW www = await new WWW("numbagamestudio.com/unity/bundles/mybundle");

    // Await loading asset from asset bundle.
    GameObject myPrefab = (GameObject)(await www.assetBundle.LoadAssetAsync("MyAsset")).asset;

    // Await asynchronous scene loading.
    await SceneManager.LoadSceneAsync(0);

    Log("The end of Awake method");
}

private IEnumerator EnumerateSomething()
{
    yield return new WaitForSeconds(1f);
    Log("This line will shown in console after previous line completely awaited 1 second.");
}
```

Behind the scenes await extensions use Unity coroutine engine for awaiting. It means that you can use all unity-time-dependent operations and they will correctly processed;

Threads

What about if you want run your code on background thread, and then run on main thread again? You can do this with two special classes: **WaitForBackgroundThread** and **WaitForUpdate**. You just need await one of them and code below run in the corresponding thread. Look at code below:

```
private async void Awake()
{
    Log("Hi from main thread");

    // Code below this line runs in background thread.
    // Do not use any Unity API in background threads, it is not allowed.
    await new WaitForBackgroundThread();

    // This will run on background thread.
    int result = HeavyCalculations();

    // Code below this line runs on main thread again.
    await new WaitForUpdate();

    // This will run on main thread.
    Log(result);
}

private int HeavyCalculations()
{
    int result = 0;
    for (int i = 0; i < 1024; i++) result += i;

    return result;
}
```

Code after awaiting **WaitForBackgroundThread** runs in background thread.
Code after awaiting **WaitForUpdate** runs in main thread after “update cycle”. Awaiting **WaitForUpdate** in main thread just skip one frame.

Asynchronous methods and exceptions

In asynchronous method exceptions catched by task on which it happens. It means that if you call asynchronous method from synchronous - you need manually wait it (that freeze main thread), otherwise exceptions remains unhandled. Look at code below:

```
private void Awake()
{
    // In this case exception catched by task (not throwed in main thread).
    CreateTask();

    // In this case we freeze main thread and exceptions will thrown.
    CreateTask().Wait();
}

private Task CreateTask()
{
    // Exception will captured by task.
    return Task.Run(() => { throw new Exception(); });
}
```

Here in first case we create task and forget about her (exceptions will not thrown in main thread), in second case we manually wait it (main thread will freeze, but exceptions will thrown in main thread).

There is another way to catching exceptions - use **async void** method, which called from synchronous method and call asynchronous method with awaiting it. In this case exception rethrown in **UnitySynchronizationContext** in its **Update** method which called from Unity main thread, it means that Unity will catch this exceptions or you can catch it manually with try-catch instruction. Look at code below:

```
private void Awake()
{
    // CreateTaskAndCatchErrors called from synchronous code.
    CreateTaskAndCatchErrors();
}

private async void CreateTaskAndCatchErrors()
{
    // CreateTask is asynchronous method.
    await CreateTask();
}

private Task CreateTask()
{
    // Exception will captured by task.
    return Task.Run(() => { throw new Exception(); });
}
```

Now exceptions will be catched by Unity. But we need always create additional **async void** method, what is not good. Fortunately you do not need create **async void** method, just use “Await Extensions” task extension method **CatchErrors** which rethrow error in main thread if its occurs.

```
private void Awake()
{
    // Exception will rethrowed.
    CreateTask().CatchErrors();
}

private Task CreateTask()
{
    // Exception will captured by task.
    return Task.Run(() => { throw new Exception(); });
}
```

Remember: Never leave exceptions not handled if you do not want a big headache.

Summing up

Now you do not need to use coroutines directly.
Mark the method as **async** and use **await** instructions.

You can await:

- Any YieldInstuction object (WaitForEndOfFrame, WaitForFixedUpdate, WaitForSeconds, Coroutine, AsyncOperation and other derived classes);
- Any CustomYieldInstruction object (WaitForSecondsRealtime, WaitUntil, WaitWhile and other);
- WWW object with result;
- AssetBundleRequest with result;
- IEnumerator;
- WaitForUpdate object;
- WaitForBackgroundThread object.

After any awaiting of one of enumerated above classes (except for WaitForBackgroundThread) continuation always runs in main thread.

Contacts

For all questions you can write me a message on e-mail: numbagamestudio@gmail.com;