AwaitExtensions

AwaitExtensions is a library for the Unity game engine that allows you to apply async/await patter to multiple Unity classes.

What is `async/await`?

Async/`await` is a syntactic sugar that simplifies working with asynchronous code. When you need to wait for a `Task` object (or an object implementing `GetAwaiter`) to complete execution, you simply write the `await` keyword before it, and all code following that instruction will run after the task completes. The method containing the `await` keyword must be marked as `async`.

The `async/await` mechanism has been integrated in C# 5.0 and .NET 4.5. For details on how to use `async/await`, visit the Microsoft Web site.

Unity and `async/await` Unity officially supports C# 5.0 and .NET 4.5 starting in 2018. This means you can use these instructions without any problems. The example below demonstrates how you can wait 1 second of real time in the Awake method using `async/await`:

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

The `Log("Awaited")` method will be called from the main thread after waiting 1000 milliseconds.

You can use almost all features of C# in Unity, but by default Unity does not support waiting for its own types (WaitForSeconds and others), for example this code will not work:

```
private async void Awake()
{
  await new WaitForSeconds(1f); // error, await does not understand how to expect WaitForSec
  Log("Awaited");
}
```

This is because `await` does not know how to wait for the `WaitForSeconds` object. If we ”explain” to it how to wait, it can. To ”explain” to `await` how the WaitForSeconds object should be handled, we need to create a `GetAwaiter` extension method for the `WaitForSeconds` class which returns some object that knows how to expect `WaitForSeconds` and which is used by the `await` keyword.

All time-dependent operations (WaitForFixedUpdate, WaitUntil, wait for coroutine, wait for WWW request and others) can be easily extended with the above method and then used with the `await` keyword.

What exactly does Await Extensions do?

It extends many Unity classes to work with the `await` keyword. Take a look at the example below, it shows all the uses of the `await` instruction with the Await Extensions asset:

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

  // Wait for standard Unity Wait objects.
  await new WaitForEndOfFrame();
  Await new WaitForFixedUpdate();
  await new WaitForSeconds(1f);
  await new WaitForSecondsRealtime(1f);
  await new WaitUntil(() => true);
  await new WaitWhile(() => false);

  // We wait for the IEnumerator object to terminate. Unity's coroutine engine is used unde
  await EnumerateSomething();

  // waiting for something to load from the network.
  await UnityWebRequest.Get("google.com").SendWebRequest();

  // Waiting for the scene to load.
  await SceneManager.LoadSceneAsync(0);

  Debug.Log("End of Awake method");
}

private IEnumerator EnumerateSomething()
{
  yield return new WaitForSeconds(1f);
  Debug.Log("This line will be displayed in the console after waiting for the previous line
}
```

Behind the scenes Await Extension uses the Unity coroutine engine to wait. This means that you can use all time-dependent operations, and they will be handled correctly.

Threads.

What if you want some part of the method to run in the background thread and the rest in the main thread? You can do this with two special classes: `WaitForBackgroundThread` and `WaitForUpdate`. You just have to wait for one of these objects so that the code that follows starts executing in the corresponding thread. Look at the code below:

```
private async void Awake()
{
  Debug.Log( ``Hello from the main thread!)

  // The code following this line will execute in the background thread.
  // Don't use Unity APIs (GameObject, Components, etc.) in the background thread, it's forb
  Await new WaitForBackgroundThread();

  // This instruction will be executed in the background thread.
  int result = HeavyCalculations();

  // The code located after this line will be executed in the main thread.
  Await new WaitForUpdate();

  // This instruction is executed in the main thread.
  Debug.Log(result);
}

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

  return result;
}
```

The code after waiting `WaitForBackgroundThread` is executed in the background thread. The code after waiting `WaitForUpdate` runs in main thread after Update loop (called by Unity itself). Waiting for `WaitForUpdate` in the main thread just skips one frame.

Asynchronous methods and exceptions

In an asynchronous method, exceptions are caught by the task in which they occurred. This means that if you call an asynchronous method from a synchronous method - you have to manually wait for it (in which case the main thread will be blocked for the waiting time), otherwise the exceptions will remain unhandled. Look at the code below:

```
private void Awake()
{
  // In this case, an exception raised inside the task will be caught by the task itself (th
  CreateTask();

  // In this case, we explicitly wait for the completion of the task returned by the `Create
  // however, the main thread will freeze while the task is waiting.
  CreateTask.Wait();
}

private Task CreateTask() => Task.Run(() => throw new Exception());
```

Here, in the first case we create a task and forget about it (exceptions will not be generated in the main thread), in the second case we wait for it manually (the main thread will hang, but exceptions will be generated in the main thread).

However, there is another way to catch exceptions which is to use `async void` method which is called from synchronous method and calls asynchronous method with waiting. In this case, the exception will be re-generated in `UnitySynchronizationContext` in its `Update` method, which is called from the main Unity thread, which means that Unity will catch this exception, or you can catch it manually using the `try/catch` instruction. Look at the code below:

```
// CreateTaskAndCatchErrors is called from the main thread.
private async void Awake() => CreateTaskAndCatchErrors();

// Start a task using CreateTask and wait for it.
private async void CreateTaskAndCatchErrors() => await CreateTask();

// Generate an exception in the task.
private Task CreateTask() => Task.Run(() => throw new Exception());
```

Now exceptions will be caught by Unity. But we always need to create an extra `async void` method, which is not good. Fortunately, you don't need to create an `async void`, just use the `CatchErrors` extension method for an object of type `Task` that replay the error in the main thread if it occurs.

```
// CreateTaskAndCatchErrors is called from the main thread. Exceptions will be caught by the
Private async void Awake() => CreateTaskAndCatchErrors().CatchErrors();

// Generate an exception in the task.
private Task CreateTask() => Task.Run(() => throw new Exception());
```

Remember: never leave exceptions unprocessed unless you want a big headache in the future.

Conversion between `IEnumerator`, `Task` and `Coroutine`

You can convert any `Task` object to an `IEnumerator` object. This is useful for backward compatibility reasons. Simply call the `AsEnumerator` extension method for the task. The `IEnumerator` object will be enumerated until the task is complete. Example:

```
private IEnumerator Start()
{
  // Convert Task to IEnumerator.
  var enumerator = Task.Delay(1000).AsEnumerator();

  // wait for the task using the IEnumerator object
  while (enumerator.MoveNext())
    return null;

  Debug.Log("Task waiting is complete!");
}
```

The example above could be made more compact:

```
private IEnumerator Start()
{
  // Convert Task to IEnumerator and wait for it to finish using Unity coroutine engine.
  Return StartCoroutine(Task.Delay(1000).AsEnumerator());

  Debug.Log("Task Waiting is complete!");
}
```

This example can also be made even shorter by using the `AsCoroutine` method, which converts the task to a coroutine:

```
private IEnumerator Start()
{
  // Convert Task to IEnumerator and wait for it to finish using Unity's coroutine engine.
  Return Task.Delay(1000).AsCoroutine();

  Debug.Log("Task waiting is complete!");
}
```

Conclusion

You can now use asynchronous methods side by side with coroutines and `IEnumerator` objects, and executing code in the background thread has become noticeably easier.

You can expect:

- Any `YieldInstuction` object (`WaitForEndOfFrame`, `WaitForFixedUpdate`, `WaitForSeconds`, `Coroutine`, `AsyncOperation` and other derived classes);
- Any `CustomYieldInstruction` object (`WaitForSecondsRealtime`, `WaitUntil`, `WaitWhile` and others);
- `UnityWebRequestAsyncOperation` object with the result;
- `AssetBundleRequest` with the result;
- `IEnumerator`;
- `WaitForUpdate` - to execute code in the main thread;
- `WaitForBackgroundThread` - to execute code in a secondary thread.

After waiting for any of the above object types (except `WaitForBackgroundThread`), the code below is always executed in the main thread.

You can also convert the task to `IEnumerator` or `Coroutine` for backward compatibility.