Moroutines

Moroutines - More Than Coroutines, is a C# library written for Unity to handle coroutines.

Unity provides the ability to work with coroutines by default, but this approach has disadvantages. With this library we tried to work around those drawbacks by offering you your own coroutine API. Nevertheless, our library uses Unity-coroutines to organize pseudo-parallel execution of your functions. You can use both the built-in approach for working with coroutines and our library at the same time.

Why moroutine?

Unity already has a `Coroutine` class to work with coroutines and we had several ways to implement our library. We decided to take the path of least resistance and for that very reason our library calls coroutines differently - moroutines. This allows you to easily use both the coroutines built into Unity using the `Coroutine` class, and the more advanced coroutines from our library using the `Moroutine` class.

What are the benefits?

The built-in approach to working with coroutines has several disadvantages:

- One coroutine can only be waited on by another one coroutine, otherwise (if there are more than one waiting), Unity will report you an error in the console during game execution.
- There is no way to find out about the state of the coroutine by the `Coroutine` class object (zeroed, running, suspended, completed or destroyed).
- There is no way to pause or restart a coroutine on a `Coroutine` class object.
- There is no way to create a coroutine with a delayed start.
- There is no possibility of waiting for a pause or resuming a coroutine.
- There is no way to subscribe to coroutine state change events.
- There is no way to get the last result of the coroutine.
- It is not possible to run several coroutines with one method.
- There is no way to wait for the completion of the execution of several coroutines at once.
- There is no way to wait for the completion of the execution of at least one of several coroutines at once.
- There is no way to get all coroutines of a particular game object.
- And others...

Our library excludes the disadvantages listed above. You can easily control the coroutine with just a couple of lines of code, determine its state, react to events, and so on.

Import Library

You can import our library using the Asset Store or by downloading the Unity-package from here.

> Do not extract the contents of the `Plugins` folder to another location, this will make some `internal` classes available to you, which could lead to errors in the future.

Connecting namespaces

To work with coroutines, you need to connect the `Redcode.Moroutines` namespace. This space contains all the data types we have created to work with advanced coroutines.

```
using Redcode.Moroutines;
```

You can then use the `Moroutine` class from this library to work with coroutines.

Creating an advanced coroutine

To create a coroutine you need an `IEnumerator` object. As you probably know, the easiest way to create such an object is to use the `yield` instruction inside the method that returns the `IEnumerator`.

```
private IEnumerator TickEnumerator()
{
    while (true)
    {
        return new WaitForSeconds(1f);
        print("Tick!");
    }
}
```

The example above declares a method that outputs the text "Tick!" to the Unity console infinitely with a second delay. To run it in the built-in Unity way you would use the `StartCoroutine` method, but this method would return you a `UnityEngine.Coroutine` object, which provides no information about the state of the coroutine. Instead, you should use the `Moroutine.Create` method.

```
Moroutine.Create(TickEnumerator());
```

In this case, the static method `Moroutine.Create` will return you moroutine with many methods and properties to work with it. In general, a script with the above code examples will look like this.

```
using System.Collections;
using UnityEngine;
using Redcode.Moroutines;

public class Test : MonoBehaviour
```

```
{
    private void Start() => Moroutine.Create(TickEnumerator());

    private IEnumerator TickEnumerator()
    {
        while (true)
        {
            return new WaitForSeconds(1f);
            print("Tick!");
        }
    }
}
```

But if you try to run this code, nothing happens. This is because the `Moroutine.Create` method creates a moroutine and returns it, but does not start its execution process.

Run moroutine

You can run it by calling the `Run` method as in the example below.

```
var mor = Moroutine.Create(TickEnumerator());
mor.Run();
```

The example above can be shortened using a chain of method calls.

```
Moroutine.Create(TickEnumerator()).Run();
```

This example can also be shortened using the static method `Moroutine.Run`.

```
Moroutine.Run(TickEnumerator());
```

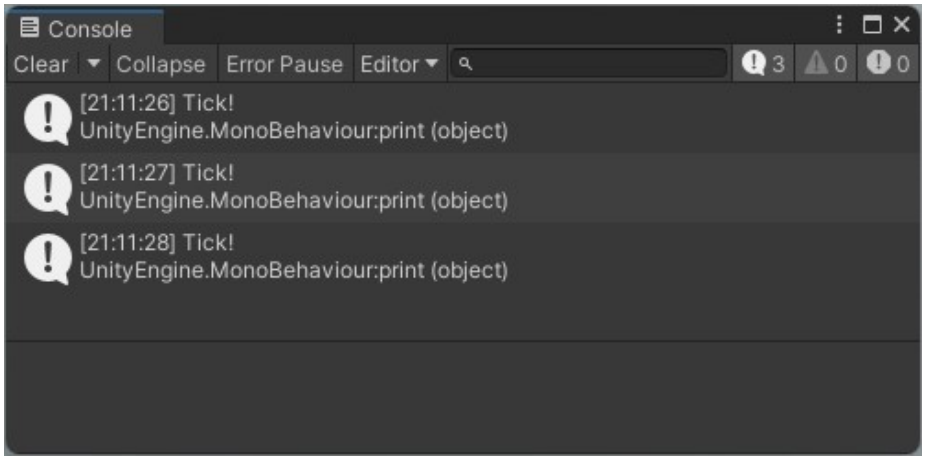
Use the method `Moroutine.Run` if you need to create a coroutine and run it immediately. The complete code example looks like this.

```
using System.Collections;
using UnityEngine;
using Redcode.Moroutines;

public class Test : MonoBehaviour
{
    private void Start() => Moroutine.Run(TickEnumerator());

    private IEnumerator TickEnumerator()
    {
        while (true)
        {
            return new WaitForSeconds(1f);
            print("Tick!");
        }
```

```
    }
}
```

> The `Moroutine.Run` method also returns a moroutine object, so you can use it not only to run, but also for further manipulation.

If you run the game with this script, you will get "Tick!" messages in the console every second.



Stop the moroutine.

To stop the moroutine, use the `Stop` method on the moroutine object.

```
var mor = Moroutine.Run(TickEnumerator()); // Run

return new WaitForSeconds(1f); // Wait 1 second
mor.Stop(); // Stop
```

Continue moroutine

If you want to continue moroutine after stopping, call the Run method on it again.

```
var mor = Moroutine.Run(TickEnumerator()); // Run

return new WaitForSeconds(1f); // Wait 1 second
mor.Stop(); // Stop

yield return new WaitForSeconds(3f); // Wait 3 seconds.
mor.Run(); // Continue
```

Moroutine completion

The method (`TickEnumerator()`) that was passed to the morutina has an infinite loop inside. For this reason, such a morutina will never end. However, if you pass a method that has a termination condition, then the morutina will terminate sooner or later. For example:

```
private void Start() => Moroutine.Run(DelayEnumerator(1f));

private IEnumerator DelayEnumerator(float delay)
{
    yield return new WaitForSeconds(delay);
    print("Completed!");
}
```

In this case, the `DelayEnumerator(float delay)` method is final. Please note that this method generates an `IEnumerator` object, which means that this object will not implement the `Reset` method, which means that such an object cannot be reset to initial state. For this reason, when a moroutine that was passed an `IEnumerator` object finishes executing, it is automatically destroyed (which makes sense), which means you can't run it again.

However, you can replace `IEnumerator` with `IEnumerable` in a method declaration. `IEnumerable` can generate `IEnumerator` objects, which can be used as an alternative to the `Reset` method.

```
private void Start() => Moroutine.Run(DelayEnumerable(1f));

private IEnumerable DelayEnumerable(float delay) // Note that the method now returns an IEn
{
    yield return new WaitForSeconds(delay);
    print("Completed!");
}
```

In this case, the moroutine can be restarted so that it starts execution from the beginning. It is for this reason that such moroutines are not automatically destroyed.

Auto-destruct settings

You can control the auto-destruction of a moroutine using the `SetAutoDestroy` method or the `AutoDestroy` property:

```
private void Start() => Moroutine.Run(DelayEnumerable(1f)).SetAutoDestroy(true); // <-- aut

private IEnumerable DelayEnumerable(float delay)
{
    yield return new WaitForSeconds(delay);
```

```
    print("Completed!");
}
```

In the example above, the moroutine is not automatically destroyed by default, however, with the `SetAudoDestroy` method, we specified that it should be destroyed after completion. Similarly, you can override the auto-destruction of a moroutine created with the `IEnumerator` object, but this doesn't make much sense, because once completed, such a moroutine simply won't do anything, even if you try to run it again and again.

Manual destruction of morutina

You can destroy a morutina by calling its `Destroy` method:

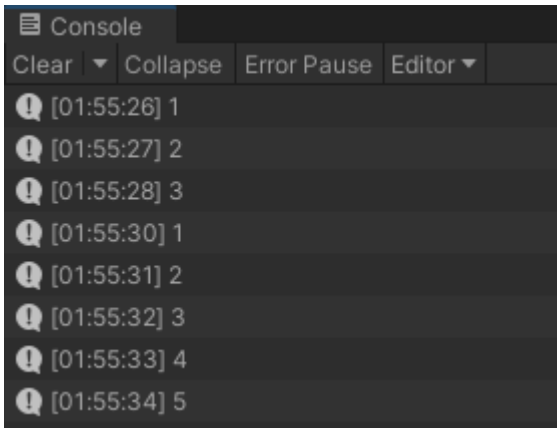
```
varmor = Moroutine.Run(TickEnumerator());
yield return new WaitForSeconds(3.5f)
mor.Destroy(); // Stop and destroy the morutina.
```

> If a morutina is no longer used in your game, then it must be destroyed, otherwise the memory will not be freed.

Restart moroutine

You can restart the moroutine (start its execution from the very beginning), to do this, use the `Reset` method.

```
private IEnumerator Start()
{
    var mor = Moroutine.Run(TimerEnumerable());
    yield return new WaitForSeconds(3.5f);    //    3.5     ..

    mor.Reset();    //                    (                 ).
    mor.Run();      //           .
}

private IEnumerable TimerEnumerable()
{
    var seconds = 0;

    while (true)
    {
        yield return new WaitForSeconds(1f);
        print(++seconds);
    }
}
```



Note that calling the `Reset` method resets the state of the moroutine and stops it. This means that you yourself must take care of its further launch. The `Run`, `Stop` and `Reset` methods return the morutina they belong to, this allows you to chain multiple method calls together and shorten your code.

```
mor.Reset().Run();
```

This code can also be shortened by using the `Rerun` method, which calls the `Reset` and `Run` methods in sequence.

```
mor.Rerun();
```

You can also call the `Reset` or `Rerun` methods on it to use it again after the moroutine has run, but this is likely to be unnecessary in this case. Instead, just use the `Run` method, it has the `rerunIfCompleted` parameter, which you can use if you want to replay the moroutine after completion. By default, this parameter is set to `true`.

Moroutine status

You can check the status of the moroutine using the following properties:

- `IsReseted` - is the moroutine reset?
- `IsRunning` - is the moroutine running?
- `IsStopped` - is the moroutine stopped?
- `IsCompleted` - is the moroutine completed?
- `IsDestroyed` - is the moroutine destroyed?
- `CurrentState` - returns an enumeration which represents one of the above states.

The first four return a Boolean value that represents the corresponding state. Example:

```
var mor = Moroutine.Run(CountEnumerable());
print(mor.IsRunning);
```

Subscription events and methods

Moroutines have the following events:

- `Reseted` - fires when the moroutine is reset to its initial state.
- `Running` - fires immediately after calling the `Run` method.
- `Stopped` - fires only when the moroutine has stopped.
- `Completed` - fires when the moroutine has finished.
- `Destroyed` - triggered when a moroutine is destroyed.

You can subscribe to any of these events when needed. The subscript method must match the following signature:

```
void EventHandler(moroutine moroutine);
```

The `moroutine` parameter will be substituted with the moroutine that caused the event.

```
var mor = Coroutine.Run(CountEnumerable());
mor.Completed += mor => print("Completed");
```

You can also quickly subscribe to the desired event using the following methods:

- OnReseted - to subscribe to zeroing.
- OnRunning - subscription for startup.
- OnStopped - subscription for stopping.
- OnCompleted - subscription for termination.
- OnDestroyed - subscription for destruction.

```
var mor = Moroutine.Run(CountEnumerable());
mor.OnCompleted(c => print("Completed"));
```

All of these methods return a moroutine on which they are called, so you can form long chains of calls like this:

```
Moroutine.Create(CountEnumerable()).OnCompleted(c => print("Completed")).Run();
```

Waiting for moroutine.

If you need to wait for a certain moroutine state, use the following methods:

- `WaitForComplete` - Returns an object to wait for completion.
- `WaitForStop` - returns an object to wait for a stop.
- `WaitForRun` - returns an object to wait to start.
- `WaitForReset` - returns an object to wait for a reset.
- `WaitForDestroy` - returns an object to wait for destruction.
-

Call the above methods to wait for the desired state, for example:

```
var mor = Moroutine.Run(CountEnumerable());
```

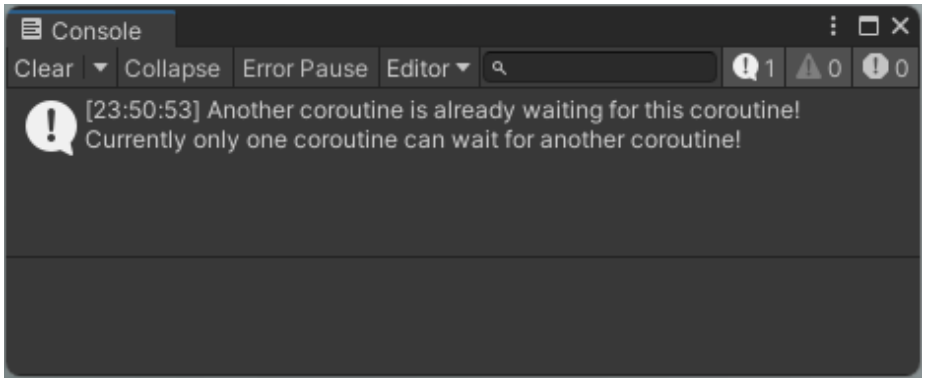
```
yield return mor.WaitForComplete(); // wait for moroutine to complete
print("Awaited"); // print text after moroutine is complete
```

The above example can be shortened to this:

```
yield return Moroutine.Run(CountEnumerable()).WaitForComplete();
print("Awaited");
```

In the built-in coroutine engine you were limited in the number of coroutines waiting, meaning one coroutine could only wait for one other coroutine, for example this code would report a second coroutine waiting error:

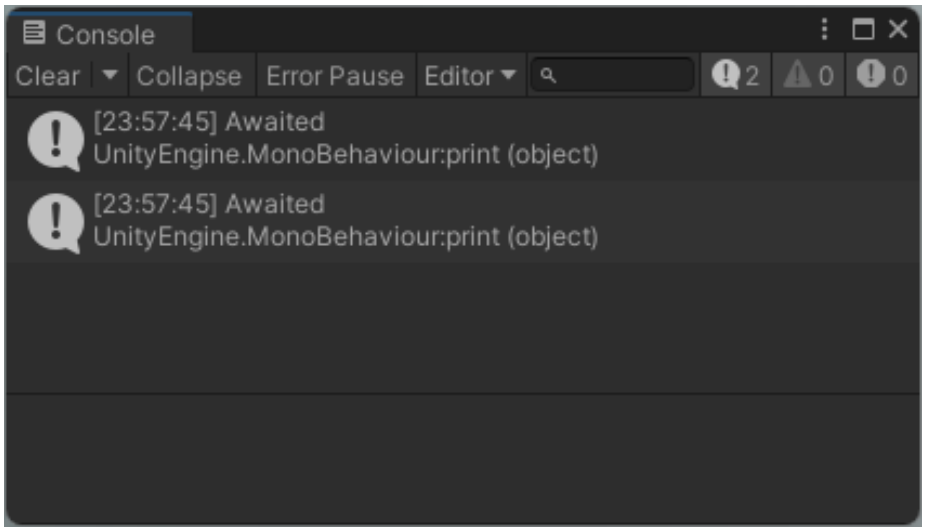
```
private void Start()
{
    var coroutine = StartCoroutine(SomeEnumerator()); // the first coroutine, imitates some
    StartCoroutine(WaitEnumerator(coroutine)); // second coroutine, waiting for the first o
    StartCoroutine(WaitEnumerator(coroutine)); // third coroutine, waiting for the first on
}

private IEnumerator SomeEnumerator()
{
    return new WaitForSeconds(3f); // simulate some execution process
}

private IEnumerator WaitEnumerator(coroutine coroutine)
{
    yield return coroutine;
    print("Awaited");
}
```



As you can see, this is indeed true, however there is no such problem with morutins, you can create as many morutins as you want, which will expect any other morutins!

```
private void Start()
{
```

```
    var mor = Moroutine.Run(SomeEnumerable());
    Moroutine.Run(WaitEnumerable(mor));
    Moroutine.Run(WaitEnumerable(mor));
}

private IEnumerable SomeEnumerable()
{
    return new WaitForSeconds(3f);
}

private IEnumerable WaitEnumerable(moroutine moroutine)
{
    yield return moroutine.WaitForComplete();
    print("Awaited");
}
```

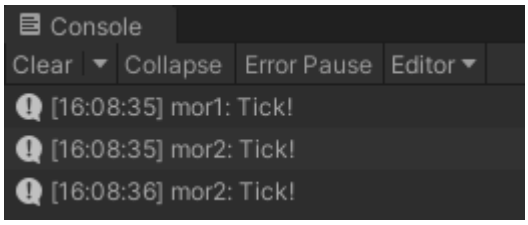


Working with multiple moroutines

You can create multiple moroutines at once using the `Create` and `Run` methods.

```
private void Start()
{
     List<Moroutine> mors = Moroutine.Run(TickEnumerable("mor1", 1), TickEnumerable("mor2",
}

private IEnumerable TickEnumerable(string prefix, int count)
{
     for (int i = 0; i < count; i++)
```

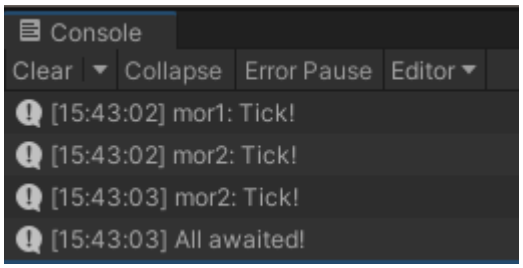
```
    {
        yield return new WaitForSeconds(1f);
        print($"{prefix}: Tick!");
    }
}
```



In this case, the method will return a list of created moroutines.

Waiting for multiple moroutines to complete You can also wait for multiple moroutines at once using the `WaitForAll` class object.

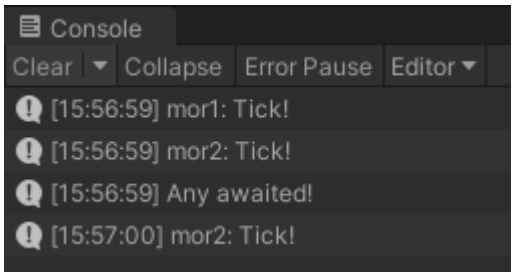
```
private IEnumerator Start()
{
    var tickMor1 = Moroutine.Run(TickEnumerable("mor1", 1));
    var tickMor2 = Moroutine.Run(TickEnumerable("mor2", 2));

    yield return new WaitForAll(tickMor1, tickMor2);
    print("All awaited!");
}

private IEnumerable TickEnumerable(string prefix, int count)
{
    for (int i = 0; i < count; i++)
    {
        yield return new WaitForSeconds(1f);
        print($"{prefix}: Tick!");
    }
}
```



You can also pass `params Moroutine[]`, `IEnumerator[]` or `IEnumerable<IEnumerator>` to the `WaitForAll` method to wait.

Wait for at least one of several moroutines to complete In addition to the `WaitForAll` class, there is also `WaitForAny`. With it, you can wait for the execution of at least one of the specified moroutines.

```
private IEnumerator Start()
{
    var tickMor1 = Moroutine.Run(TickEnumerable("mor1", 1));
    var tickMor2 = Moroutine.Run(TickEnumerable("mor2", 2));

    yield return new WaitForAny(tickMor1, tickMor2);
    print("Any awaited!");
}

private IEnumerable TickEnumerable(string prefix, int count)
{
    for (int i = 0; i < count; i++)
    {
        yield return new WaitForSeconds(1f);
        print($"{prefix}: Tick!");
    }
}
```



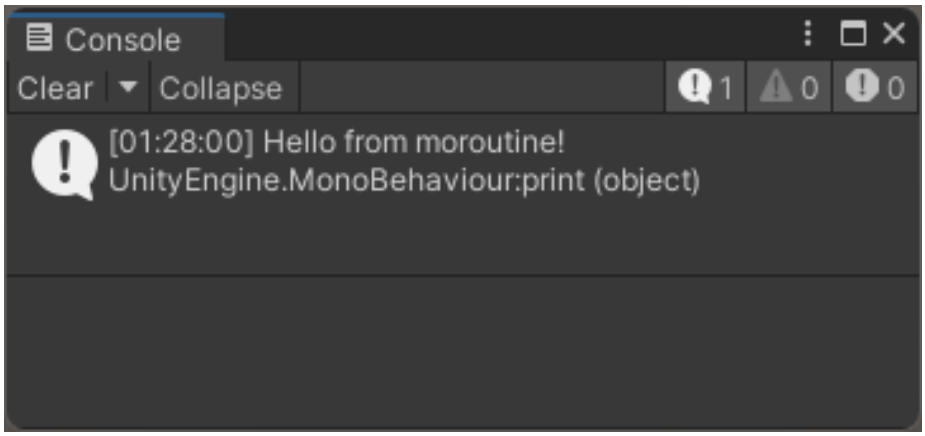
You can also pass `IList<Moroutine>`, `IEnumerator[]` or `IEnumerable<IEnumerator>` to the `WaitForAny` method to wait.

Result of moroutine

You can also easily get the last object (which was returned by the `yeild return` statement) via the `LastResult` property of the morutina.

```
private IEnumerator Start()
{
    var mor = Moroutine.Run(_owner, GenerateSomeResultEnumerable());
    yield return mor.WaitForComplete(); // wait for moroutine.

    print(mor.LastResult); // print its last result.
}
```

```
private IEnumerable GenerateSomeResultEnumerable()
{
    yield return new WaitForSeconds(3f); // simulate some process.
    yield return "Hello from moroutine!"; // and this will be the last result of moroutine.
}
```



Sometimes this comes in very handy!

Ownerless moroutines.

So far, you and I have been learning how to create orphan moroutines. A orphaned moroutine is a moroutine that is not attached to any game object. Such a moroutine cannot be interrupted except with the `Stop`, `Reset` or `Destroy` methods.

Moroutines and their owners

You can associate a moroutine with any game object, that is, make that game object the owner of the moroutine. This means that execution of a moroutine will only be possible if the host object is active, otherwise the moroutine will be stopped and you cannot restart it or continue until the host object becomes active. Attempting to start moruthin on an inactive host object will generate an exception. If the host object is active again, you can continue executing the moroutine using the `Run` method.

To specify a moroutine's owner, specify it as the first parameter in the `Moroutine.Create` or `Moroutine.Run` methods.

```
var mor = Moroutine.Run(gameObject, CountEnumerable()); // gameObject is the host of the mo
```

You can also pass in any of its components instead of the owner of the moroutine. The result will be the same.

```
var mor = Moroutine.Run(this, CountEnumerable()); // this - is a reference to the current c
```

You can also use the `SetOwner` and `MakeUnowned` methods to set a different owner or make a moroutine unowned.

```
var mor = Moroutine.Run(gameObject, CountEnumerable());
mor.SetOwner(otherGameObject); // set a different owner.
mor.MakeUnowned(); // make the moroutine unowned.
```

> Use this keyword instead gameObject, it is more shortly. You can also use `mor.SetOwner((GameObject)null)` to make a moroutine ownerless.

If you need to get the owner of the moroutine, you can use the `Owner` property of the moroutine object.

```
var mor = moroutine.Run(gameObject, CountEnumerable());
print(mor.Owner.name);
```

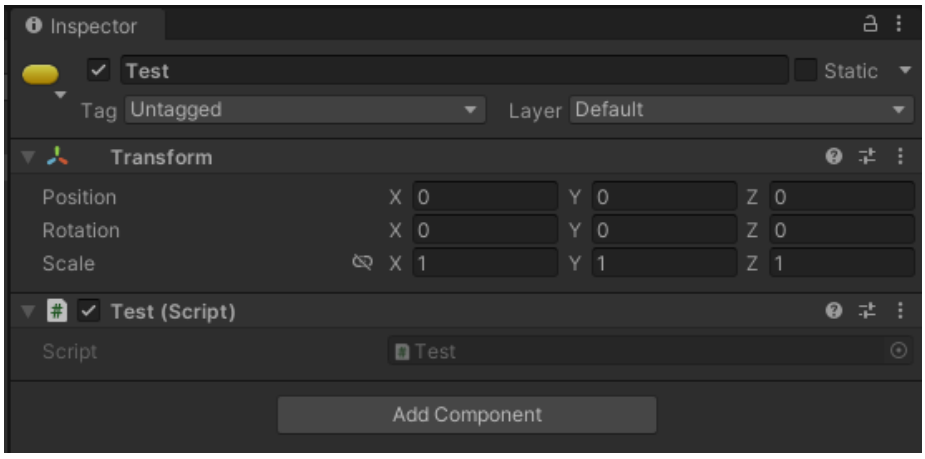
> `Owner` is a reference to the `Owner` component of the owner of the moroutine. An unowned moroutine has `Owner` equal to `null`.

`MoroutinesExecuter` object

Before your game starts, a `MoroutinesExecuter` object will be created in the scene, which will be isolated and hidden in the `DontDestroyOnLoad` scene so you won't notice it. You also won't be able to access this class from code. This object is the owner of all unowned moroutines.

The `Owner` component

Any moroutine can be assigned an owner when it is created. The owner is a normal game object. At the moment of assigning the owner of the moroutine, the `Owner` component is added to it, which will track the deactivation of this game object and, accordingly, stop the execution of the moroutine.



Many moroutines can be assigned to one owner. The `Owner` component will exist as long as it has at least one non-destroyed moroutine.

Don't try to affect the `Owner` component. It doesn't make any sense.

Get all the owner's moroutines

You can get all non-destroyed moroutines of any owner. To do this, include the `Redcode.Moroutines.Extensions` namespace and use the `GetMoroutines` method on the game object.

```
// ...
using Redcode.Moroutines.Extensions;
// ...

private IEnumerator Start()
{
     Moroutine.Run(this, TickEnumerable(1), TickEnumerable(2));

     varmors = gameObject.GetMoroutines();
     yield return new WaitForAll(mors);

     print("All awaited!");
}

private IEnumerable TickEnumerable(int count)
{
     for (int i = 0; i < count; i++)
     {
         yield return new WaitForSeconds(1f);
         print("Tick!");
     }
}
```

You can also use a state mask to filter out moroutins.

```
var mors = gameObject.GetMoroutines(Moroutine.State.Stopped | Moroutine.State.Running);
```

Getting all unowned moroutines

Use the `Moroutine.GetUnownedMoroutines` static method to get unowned moroutines. You can also use a state mask.

```
var mors = Moroutine.GetUnownedMoroutines(Moroutine.State.Running);
```

Auxiliary class `Routines`

The `Routines` static class stores the most commonly used methods to organize the execution logic of moroutines. All methods generate and return an

`IEnumerable` object which can be used by substituting other methods. In particular, there are the following methods:

- `Delay` - adds a time delay before the execution of the moroutine.
- `FrameDelay` - adds a frame delay before the execution of the moroutine.
- Repeat `Repeat` - repeats the moroutine the specified number of times.
- Wait `Wait` - Wait for execution of objects `YieldInstruction` and `CustomYieldInstruction`.

Example with `Delay`:

```
private void Start() => Moroutine.Run(Routines.Delay(1f, CountEnumerable()));

private IEnumerable CountEnumerable()
{
    for (int i = 1; i <= 3; i++)
    {
        return new WaitForSeconds(1f);
        print(i);
    }
}
```

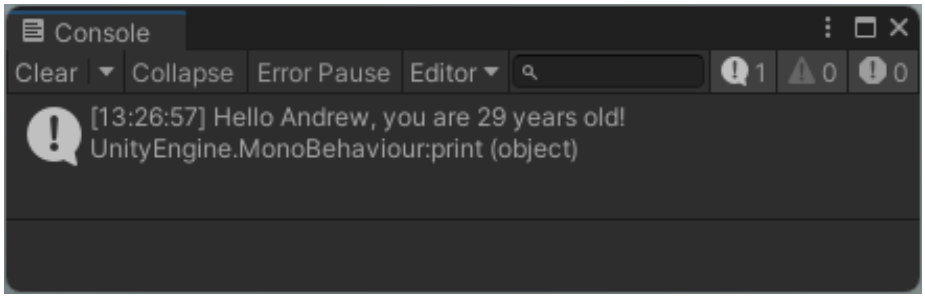
This example uses the `Delay` method, which adds a second delay before the `CountEnumerator` enumerator is executed, using the line `Routines.Delay(1f, CountEnumerable())`. As mentioned above, all methods of class `Routines` return an object `IEnumerable`, so to make the result of gluing methods `Delay` and `CountEnumerable` moroutine, you need to substitute it (the result) in method `Moroutine.Run`.

The `Delay` method also knows how to work with Action-methods, which essentially gives you the ability to quickly arrange a delayed execution of the method you need, for example:

```
private void Start() => Moroutine.Run(Routines.Delay(1f, () => print("Delayed print!")));
```

Or

```
private void Start() => Moroutine.Run(Routines.Delay(1f, () => welcome("Andrew", 29)))

private void Welcome(string name, int age) => print($"Hello {name}, you are {age} years old
```



As you can see this is very convenient and reduces code duplication.

> These methods can work with both `IEnumerable` and `IEnumerator` objects, but if you plan to restart your enumerators, you should use `IEnumerable` objects.

The `FrameDelay` method adds a frame delay before executing the enumerator. For example, if you want to wait for 1 game frame and then execute the enumerator code, it would look like this:

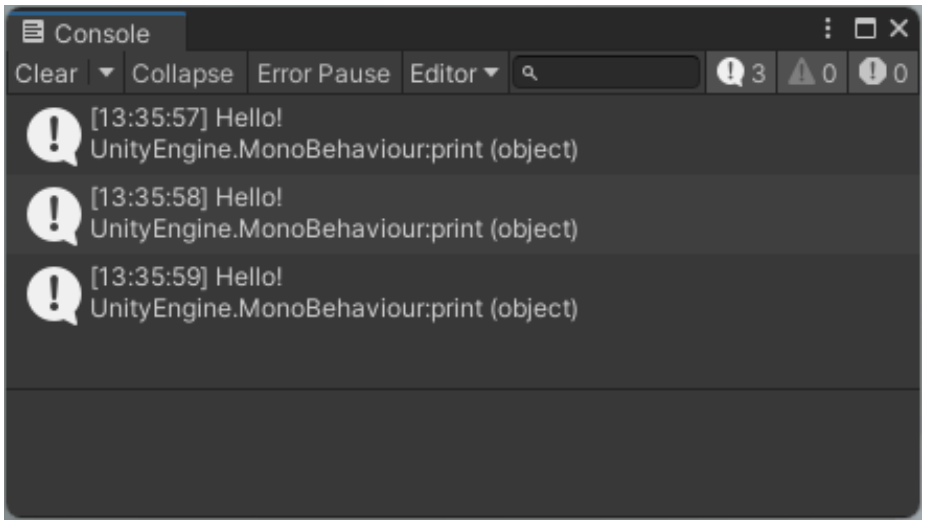
```
private void Start() => Moroutine.Run(Routines.FrameDelay(1, () => print("1 frame skipped!")
```

This method as well as the `Delay` method knows how to work with Action-methods.

The `Repeat` method repeats the specified enumeration a specified number of times. If you want infinite repetition of the enumerator execution, specify -1 as the `count` parameter of the `Repeat` method. Example:

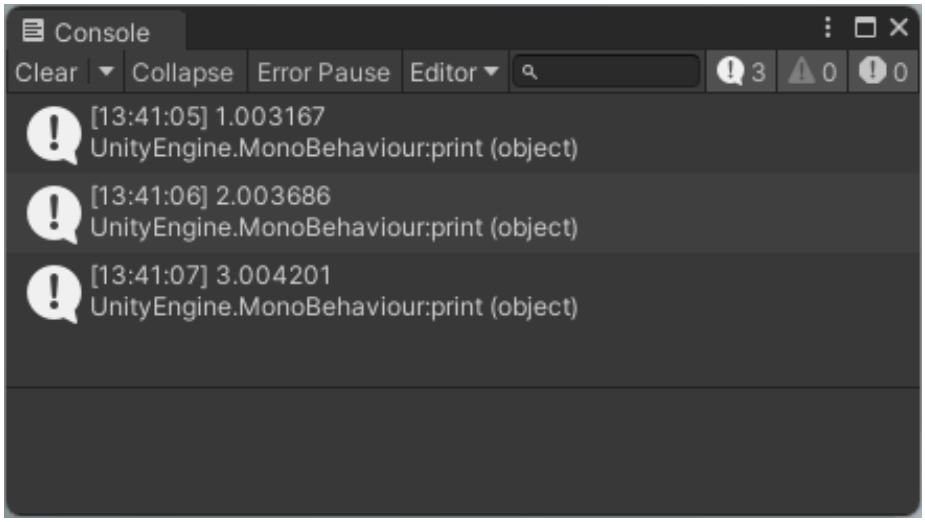
```
private void Start() => Moroutine.Run(Routines.Repeat(3, WaitAndPrintEnumerator());

private IEnumerable WaitAndPrintEnumerator()
{
    return new WaitForSeconds(1f);
    print("Hello!");
}
```

As a result, the text "Hello!" will be printed 3 times after every second.



You can combine `Delay`, `FrameDelay` and `Repeat` methods together, for example, if you want to execute some function 3 times with a 1 second delay, it will look like this

```
private void Start() => Moroutine.Run(Routines.Repeat(3, Routines.Delay(1f, () => print(Time
```



This nesting of methods into each other can be unlimited.

The `Wait` method allows you to quickly wrap a `YieldInstrution` or `CustomYieldInstruction` object into a `IEnumerable` that will simply wait for their execution. For example, if you want to wrap a `YieldInstruction`` object into a coroutine so that you can later monitor the execution status of the YieldInstruction“ through that coroutine, you can write code like this:

```
var moroutine = Moroutine.Run(Routines.Wait(instruction));
```

Where `instruction` is an object of class `YieldInstruction`.

Extensions

In addition to the main namespace, there is also the `Moroutines.Extensions` namespace, which contains extension methods for the `YieldInstruction` and `CustomYieldInstruction` classes. These methods allow you to quickly convert `Moroutine`, `YieldInstruction` and `CustomYieldInstruction` to each other. For example:

```
var delayMoroutine = Moroutine.Run(Routines.Delay(1f, () => print("Converting"))); // Create

var yieldInstruction = delayMoroutine.WaitForComplete(); // Received YieldInstruction objec
var customYieldInstruction = yieldInstruction.AsCustomYieldInstruction(); // Converted Yield
var moroutine = customYieldInstruction.AsMoroutine(); // CustomYieldInstruction was converte
```

You'll probably rarely need this conversion, but it's possible.

That's it! Now you're ready to use our coroutine engine, good luck!