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 coroutines?

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 of working with coroutines has several disadvantages:

- Only one coroutine can wait for another one coroutine, otherwise (if there is more than one waiting), Unity will tell you an error in the console at runtime.
- There is no way to know the state of a coroutine (running, paused or finished) from the `Coroutine` class object.
- There is no way to pause or restart a coroutine by an object of the class `Coroutine`.
- No possibility to create a coroutine with a delayed start.
- No ability to pause or resume a coroutine.
- No possibility to subscribe to coroutine state change events.
- No possibility to get the last result of the coroutine.
- 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 cloning the repository from here to your computer. If you decide to clone a repository, all you have to is copy the `Plugins` folder into the `Assets` folder of your project. After that, all the classes you need to work with coroutines will be available to you.

> 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 an advanced coroutine object 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 coroutine and returns it, but does not start its execution process.

Run advanced coroutine

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());
```

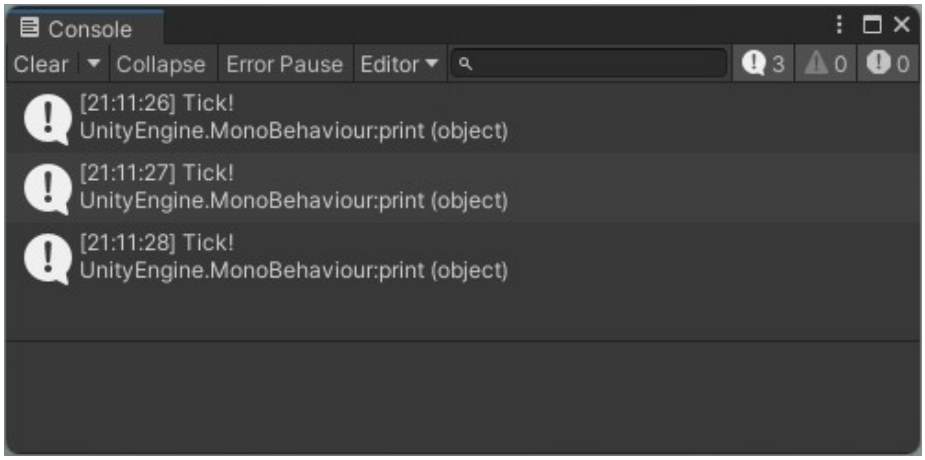
Use the method `Moroutine.Run` if you need to create a coroutine and run it immediately. The complete code example looks like this.

```
using System.Collections;
using UnityEngine;
using 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 morutina, use the `Stop` method on the morutina 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
```

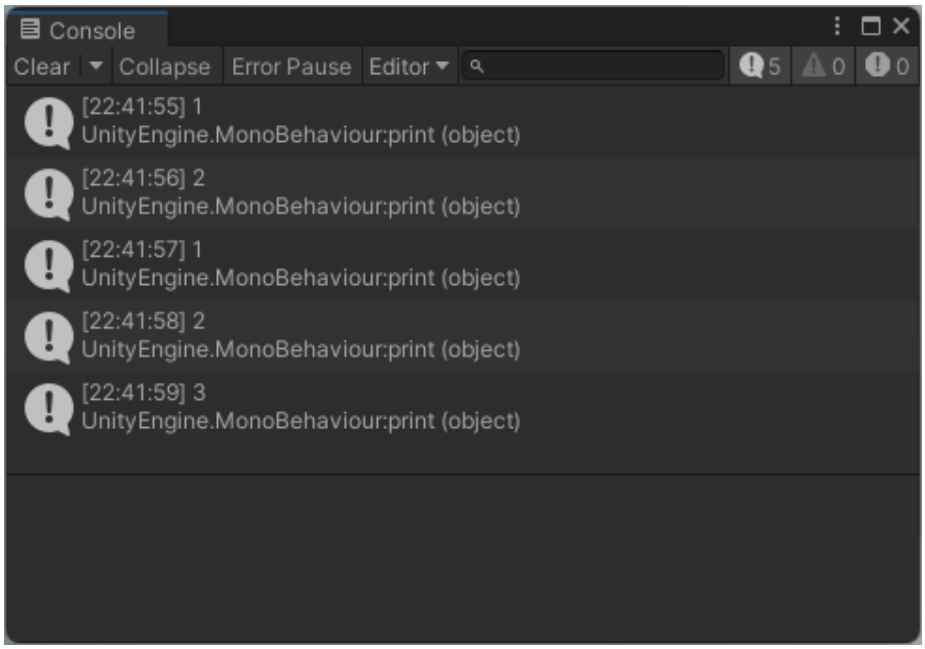
Restart moroutine

You can restart the morutine (start it from the beginning) using the method `Reset`. But before you use it, note that the method that returns an IEnumer-

ator and which uses the `yield` operator in its body generates an `IEnumerator` object which implements the `Current` property and the `MoveNext` method, but does not implement the `Reset` method. For this reason morutines that execute such methods will simply continue executing when trying to restart them. To make restarting a morutina possible, you must use IEnumerable instead of IEnumerator in the definition of the return value of the morutina method.

> We recommend using IEnumerable instead of IEnumerator wherever you write morutin methods, this will avoid obscure errors in the future.

```
private IEnumerator Start()
{
    var mor = Moroutine.Run(CountEnumerable()); // Start moroutine

    return new WaitForSeconds(2.5f); // wait 2.5 seconds
    mor.Reset(); // Reset moroutine
    mor.Run(); // restart the moruthine
}

private IEnumerable CountEnumerable() // Note that the method returns IEnumerable
{
    for (int i = 1; i <= 3; i++)
    {
        return new WaitForSeconds(1f);
        print(i);
    }
}
```



Note that calling the Reset method resets the state of the moroutine and stops it. This means it's up to you to take care of its further startup. The `Run`, `Stop` and `Reset` methods return the moroutine they belong to, this allows you to bind several method calls to each other and shorten the code.

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

You can also call the `Reset` method 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 coroutine after completion. By default, this parameter is set to `true`.

Moroutine status

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

- `IsReseted` - whether the morutina is zeroed.
- `IsRunning` - whether the moruthine is running.
- `IsStopped` - whether the moruthine is stopped.
- `IsCompleted` - whether the morutina is complete.
- `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` - is triggered when the moroutine resets to its initial state.
- `Running` - is triggered immediately after the `Run` method is called.
- `Stopped` - only triggers when the moruthine is stopped (but not terminated).
- Completed‘ - Triggers when the coroutine is finished.

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 coroutine 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.

```
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 morutina 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.

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

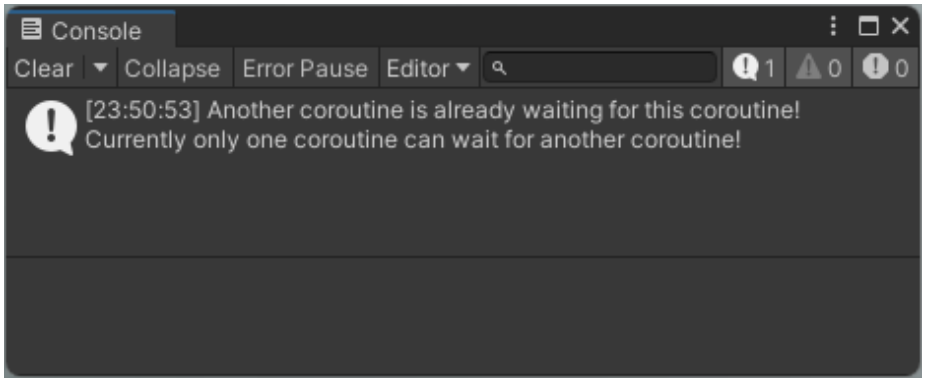
```
var mor = Moroutine.Run(CountEnumerable());

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:

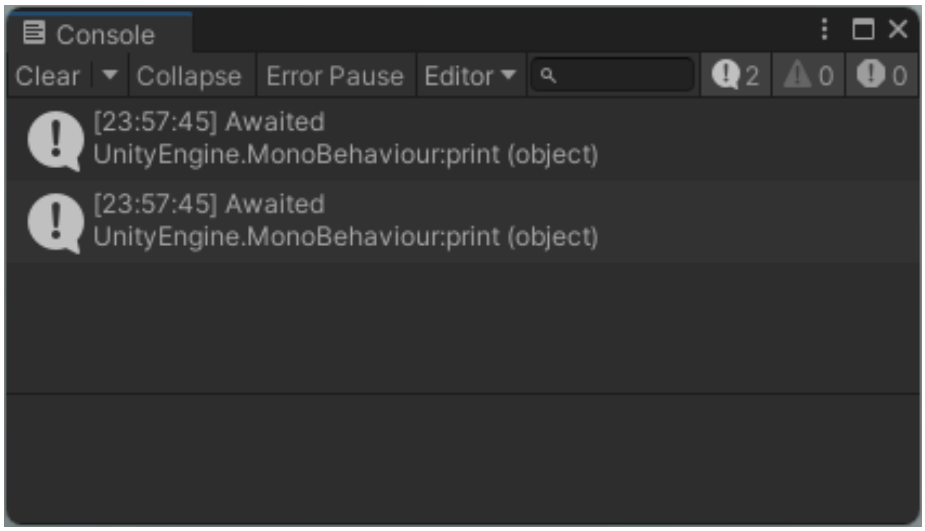
```
private void Start()
{
    var coroutine = StartCoroutine(SomeEnumerator()); // the first coroutine, imitates some
    StartCoroutine(WaitEnumerator(coroutine)); // second coroutine, waiting for the first or
    StartCoroutine(WaitEnumerator(coroutine)); // third coroutine, waiting for the first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

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");
}
```

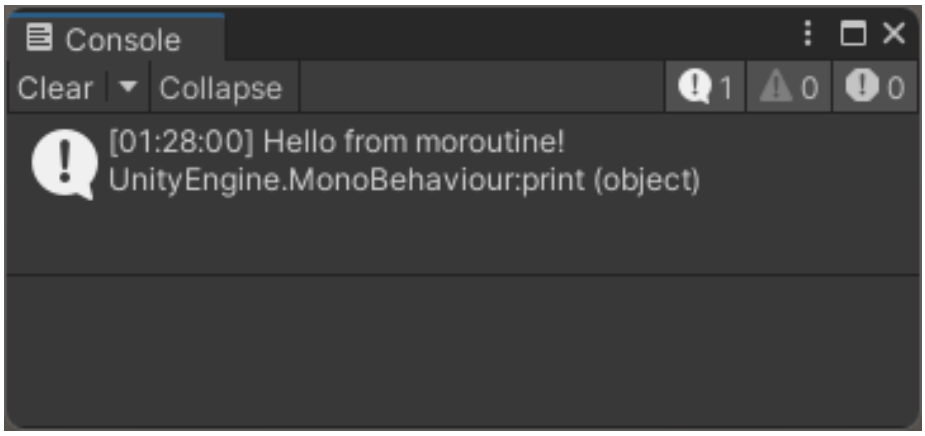


result of moroutine

You can also easily get the last object that was set in the `Current` property of the generated enumerator 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` or `Reset` methods.

moroutines and their owners

You can associate a moruthina with any game object, that is, make that game object the owner of the moruthina. This means that execution of a moruthine will only be possible if the host object is active, otherwise the moruthine 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 moruthin using the `Run` method.

To specify a moroutine host, 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 mor
```

> You can't change the moroutine's host after the moroutine has been created.

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

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

```
print(mor.Owner.name);
```

Object `MoroutinesOwner`

Actually **starting** (exactly launching, using the `Run` method) any moroutine will take place on the `MoroutinesOwner` object (that is, inside the `Run` method there is this code line `MoroutinesOwner.Instance.StartCoroutine(RunEnumerator())`), but it is well hidden from you and you do not have to try to find it or do anything with it. Just before your game starts, a `MoroutinesOwner` object will be created in the scene, which will be isolated in the `DontDestroyOnLoad` scene and hidden in it, so that you won't notice it.

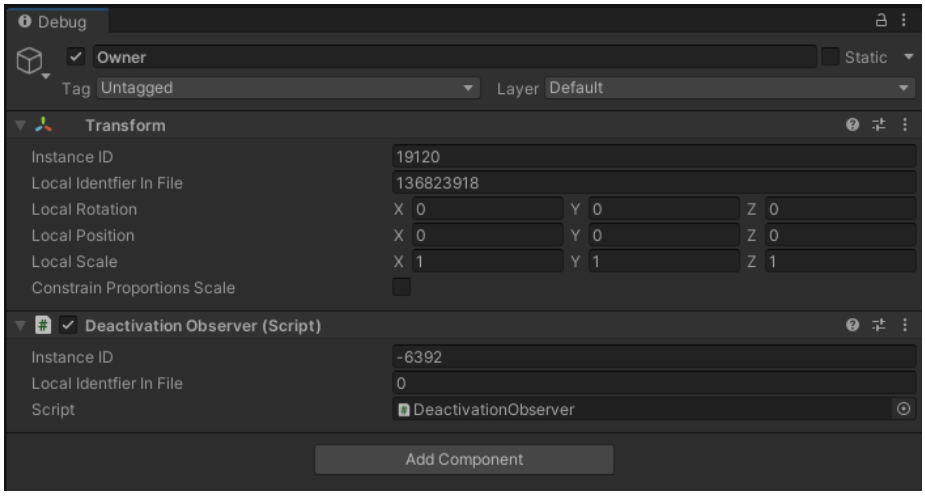
- All ownerless moroutines are owned and run on the `MoroutinesOwner` object.
- Moroutines with a host belong to their host objects and are launched on the `MoroutinesOwner` object.

  Do not try to influence the `MoroutinesOwner` object in any way.

How the deactivation of the host objects is tracked

To track the deactivation of moroutine hosts, the `DeactivationObserver` script is added to them (the hosts), which emits a `Deactivated` event, to which the moroutine associated with that host is subscribed in advance if the object is deactivated. The moruthina reacts to the deactivated event and calls the `Stop` method on itself, which causes the moruthina state to stop.

However, you will only be able to see this script on the object if you switch the inspector window to debug mode.



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);
    }
}
```

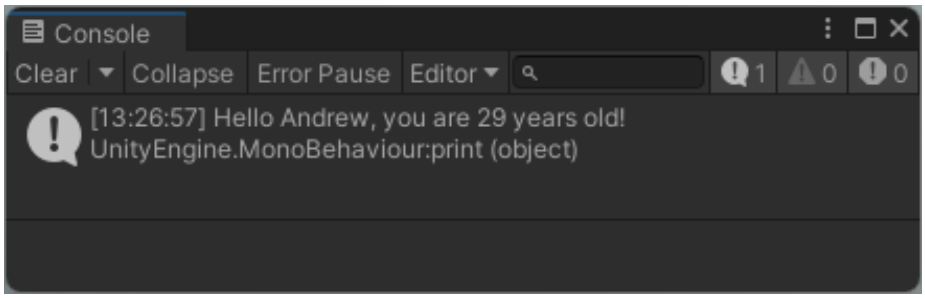
This example uses the `Delay` method, which adds a second delay before the `CountEnumerator` enumerator is executed, using the line `Routines.Delay(1f, CountEnumerable())`. As mentioned above, all methods of class `Routines` return an object `IEnumerable`, so to make the result of gluing methods `Delay` 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 (in some cases there are exceptions, not important cases at all), but if you plan to restart your enumerators, you should use `IEnumerable` objects. We recommend always using `IEnumerable` object generation instead of `IEnumerator`.

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:

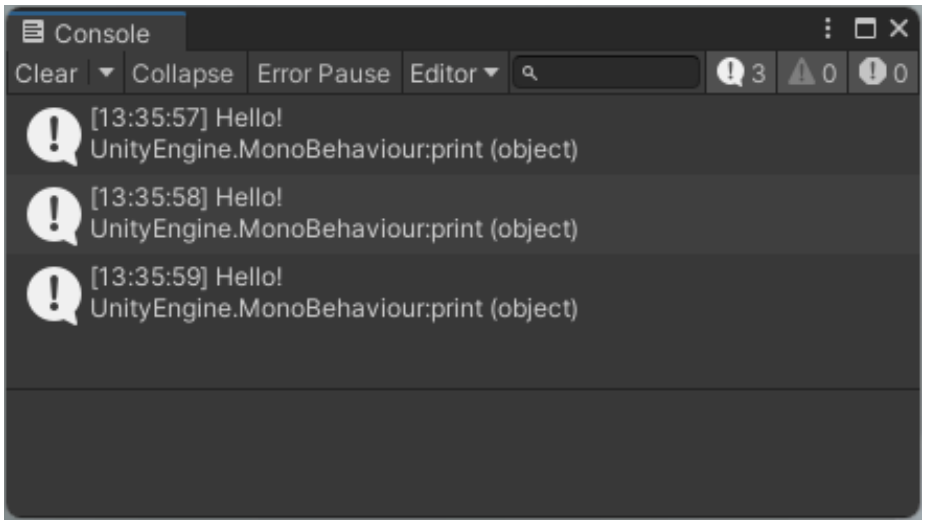
```
private void Start() => Moroutine.Run(Routines.FrameDelay(1, () => print("1 frame skipped!")
```

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:

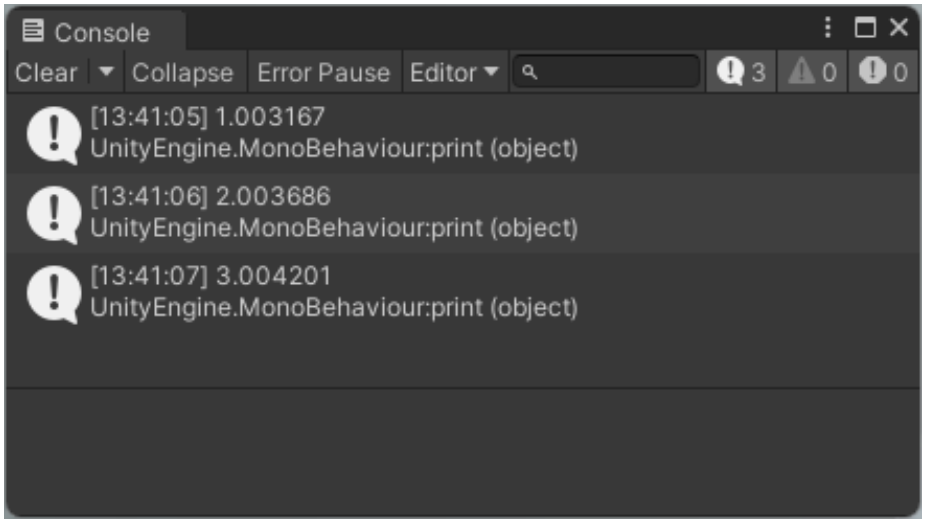
```
private void Start() => Moroutine.Run(Routines.Repeat(3, WaitAndPrintEnumerator());

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 object
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!