Moroutines

Moroutines is a C# library written for advanced coroutine work in Unity.

Unity provides the ability to work with coroutines by default, but this approach has drawbacks. With the help of this library, we tried to get around these shortcomings. offering you our API for working with coroutines. You can use both the built-in approach for working with coroutines and our library at the same time.

Why Moroutines?

Unity already has a `Coroutine` class for working with coroutines. In our library, coroutines are called moroutines (Moroutine - more than coroutine) and are represented by the `Moroutine` class. This allows you to easily use both pure coroutines and more advanced moroutines.

What are the benefits?

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

- Coroutines B and C cannot wait (yield) for coroutine A at the same time.
- Coroutines do not store information about their state (reseted, running, stopped, completed or destroyed)
- Coroutines cannot be paused/reset and then started while continuing its execution.
- There is no way to add a delay (in seconds or frames) before starting an existing coroutine.
- No ability to wait (yield) pause or play coroutine.
- Coroutines do not have state change events.
- Coroutines do not know how to return a result.
- Coroutines cannot be grouped for easy control.
- There is no way to wait (yield) the execution of several coroutines
- A coroutine does not store information about its owner object.
- No way to create an unowned coroutine.
- Coroutines don't have names, they harder to filter and harder to debug.
- Game objects do not display information about the coroutines that are associated with them.
- And other shortcomings.

When creating Moroutines, we took into account these shortcomings and provided you with a convenient API for working with coroutines.

Import library

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

Namespace connection

To work with moroutines, you need to include the `Redcode.Moroutines` namespace. This space contains all the data types that we have created to work with moroutines.

```
using Redcode.Moroutines;
```

After that, you can use the `Moroutine` class from this library to work with moroutines.

Create a moroutine

To create a moroutine, declare a method that returns an enumerator:

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

In the example above, a method is declared in which the text "Tick!" to the Unity console. To create a moroutine use the `Moroutine.Create` method

```
using Redcode.Moroutines;

public class Test : MonoBehavior
{
    private void Start() => Moroutine.Create(TickEnumerator()); // Create a morutina.

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

The `Moroutine.Create` method creates a moroutine but does not start it.

Run moroutine execution

You can start a moroutine by calling the `Run` method:

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

Or like this:

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

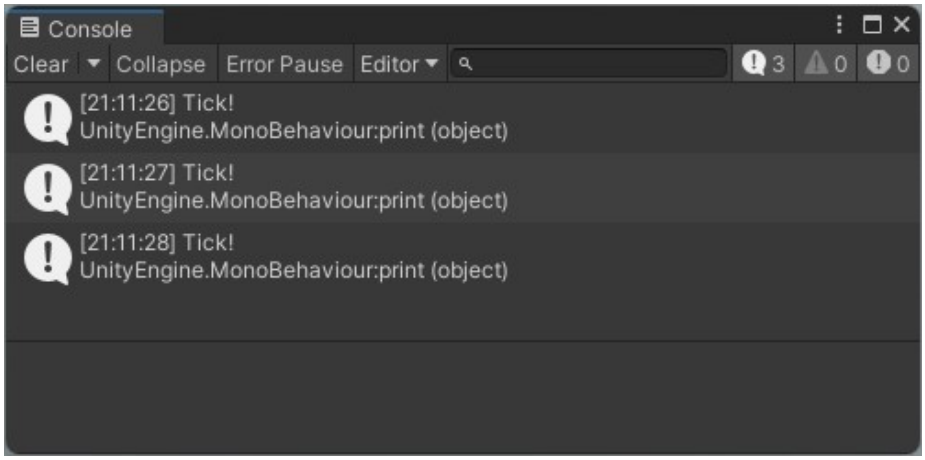
Or even shorter, using the `Moroutine.Run` static method.

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

> The `Moroutine.Run` method returns the running moroutine, so you can store it in a variable.

```
using Redcode.Moroutines;

public class Test : MonoBehavior
{
    private void Start() => Moroutine.Run(TickEnumerator()); // Create and run morutina.

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

If you start the game with this script, then the messages "Tick!" will appear in the console every second.



Stop moroutine

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

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

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

Continuation of moroutine

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

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

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

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

Moroutine completing

The method (`TickEnumerator()`) that was passed to the moroutine has an infinite loop inside. For this reason, such a moroutine will never end. Otherwise, the moroutine will end sooner or later. For example:

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

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

The `DelayEnumerator(float delay)` method is final. This method generates some `IEnumerator` object representing the execution of this method. `IEnumerator` objects cannot be restarted in C#. For this reason, when a moroutine that was passed an `IEnumerator` object finishes executing, it is automatically marked as destroyed, which means you can't run it again. If your code does not contain references to the destroyed moroutine, the garbage collector will automatically delete it after a while.

However, in the method declaration, you can replace `IEnumerator` with `IEnumerable`, in which case the moroutine can be restarted so that it starts execution again from the very beginning. It is for this reason that such moroutine are not automatically marked as destroyed.

```
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!");
}
```

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, the moroutine uses an `IEnumerable` object and therefore will not be destroyed automatically, however, using the `SetAudoDestroy` method, we indicated that after completion it should be destroyed. 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 it's run over and over again.

Manual destruction of moroutine

You can destroy a moroutine using the `Destroy` method:

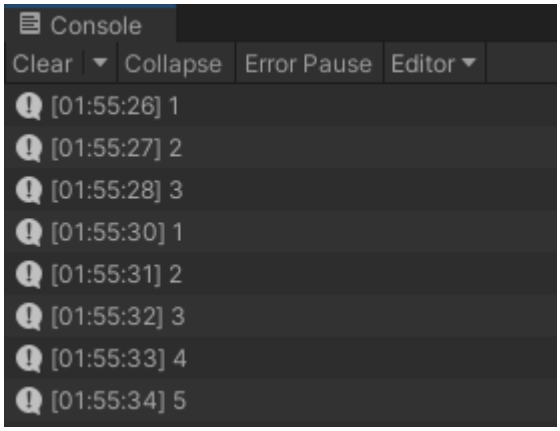
```
var mor = Moroutine.Run(TickEnumerator());
yield return new WaitForSeconds(3.5f)
mor.Destroy(); // Stop and destroy the moroutine.
```

> IMPORTANT! If a moroutine is no longer used in your game, then it must be destroyed (whether automatically or manually using the `Destroy` method), otherwise the memory will not be freed. Also, don't forget to "lose" all references to moroutine after destruction.
>
> Use the `IEnumerator` methods if there is no need to re-execute the moroutine. This will save you the unnecessary auto-destruct setting. Otherwise, you must remember to destroy the morutina.

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); // Wait 3.5 seconds..

    mor.Reset(); // Stop and reset moroutine (return to initial state).
    mor.Run(); // Restart.
}

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 moroutine they belong to, this allows you to chain multiple method calls together and shorten your code.

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

Or even shorter:

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

After executing a moroutine, you can also call the `Rerun` method on it to start it again, however, instead, use the `Run` method, it has a `rerunIfCompleted` parameter, which is `true` by default and is responsible for automatically restarting

the moroutine if it's already completed.

Moroutine state

You can check the status of a moroutine with the following properties:

- `IsReseted` - has the moroutine been reset to its initial state?
- `IsRunning` - is the moroutine running?
- `IsStopped` - is the moroutine stopped?
- `IsCompleted` - is the moroutine completed?
- `IsDestroyed` - is the moroutine destroyed?
- `CurrentState` - returns an enumeration that represents one of the above states.

The first four return a boolean value representing the corresponding state. Example:

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

Subscribe 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 whenever needed. The subscriber method must match the following signature:

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

The `moroutine` parameter will be set to 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 - subscription to reset.
- OnRunning - launch subscription.
- OnStopped - subscription to a stop.
- OnCompleted - subscription for completion.
- OnDestroyed - subscription for destruction.

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

Almost all moroutine methods return the 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 (yield) a certain state of the moroutine, then 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 for the run.
- `WaitForReset` - returns an object to wait for zeroing.
- `WaitForDestroy` - returns an object to wait for destruction.

For example like this:

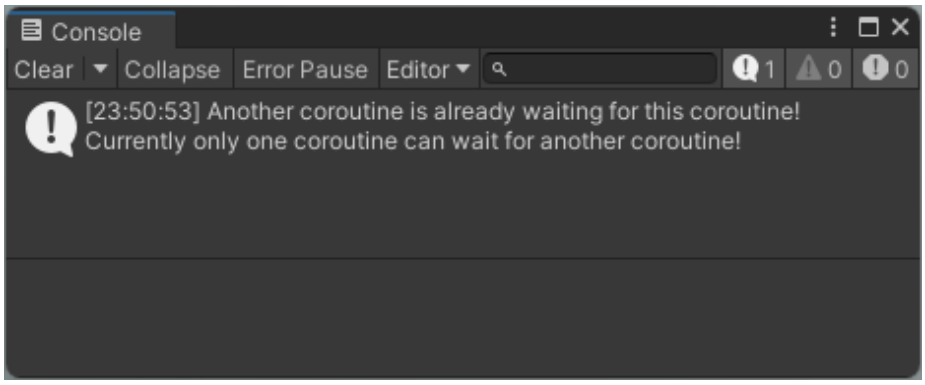
```
var mor = Moroutine.Run(CountEnumerable());

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

Or like this:

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

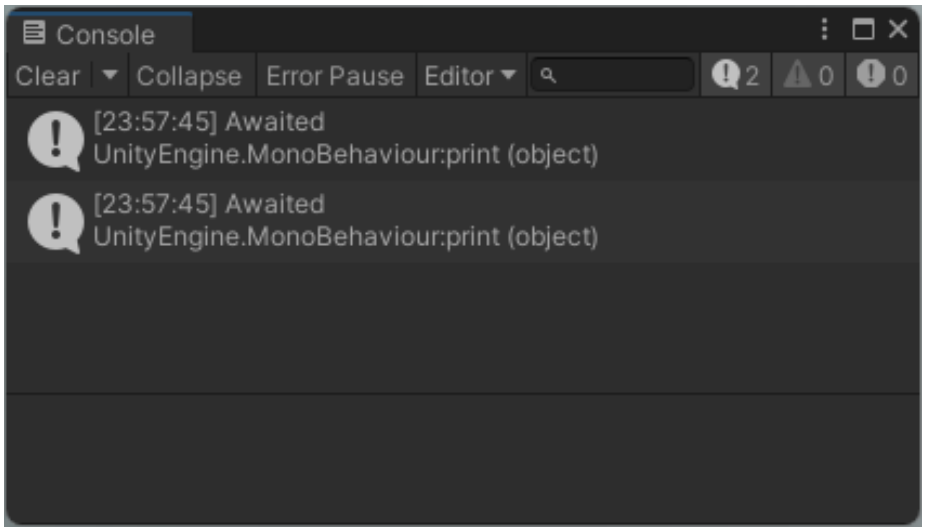
In the built-in coroutine engine, you can't have coroutines B and C waiting for coroutine A to complete at the same time, otherwise you'll get a warning in the console window:

```
private void Start()
{
    var coroutine = StartCoroutine(SomeEnumerator()); // coroutine A, simulates a certain pr
    StartCoroutine(WaitEnumerator(coroutine)); // coroutine B, waiting for coroutine A, even
    StartCoroutine(WaitEnumerator(coroutine)); // coroutine C, waiting for coroutine A, erro
}

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

private IEnumerator WaitEnumerator(Coroutine coroutine)
{
    yield return coroutine; // waiting for the received coroutine
    print("Awaited");
}
```



As you can see, this is true, but with moroutines there is no such problem, you can create as many moroutines as you like, which will wait for any other moroutines!

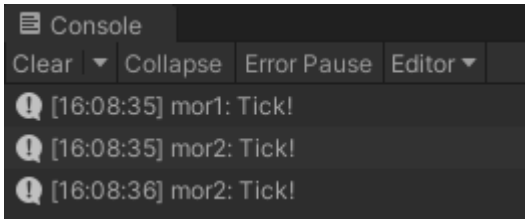
```
private void Start()
{
    var mor = Moroutine.Run(SomeEnumerable()); // Morutina A
    Moroutine.Run(WaitEnumerable(mor)); // Moroutine B, waiting for moroutine A, everything
    Moroutine.Run(WaitEnumerable(mor)); // Moroutine C, waiting for moroutine A, everything
}

private IEnumerable SomeEnumerable()
{
    yield return new WaitForSeconds(3f); // simulate some execution process..
}

private IEnumerable WaitEnumerable(Moroutine moroutine)
{
    yield return moroutine.WaitForComplete(); // wait for the received moroutine
    print("Awaited");
}
```



Working with multiple moroutines

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

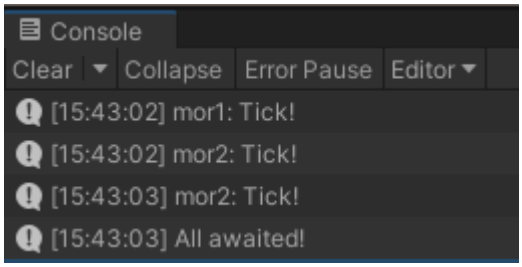
```
private void Start()
{
    List<Moroutine> mors = Moroutine.Run(TickEnumerable("mor1", 1), TickEnumerable("mor2", 2
}

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 mors = Moroutine.Run(TickEnumerable("mor1", 1), TickEnumerable("mor2", 2));
    yield return new WaitForAll(mors);

    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!");
    }
}
```



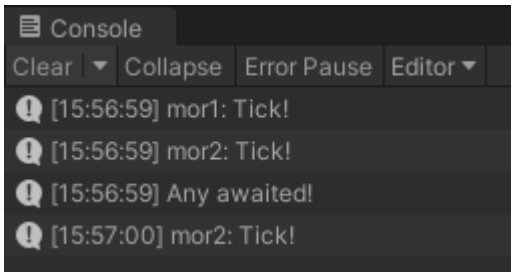
The `WaitForAll` class constructor has the following overloads:

```
WaitForAll(params Moroutine[]);
WaitForAll(IEnumerator[]);
WaitForAll(IEnumerable<IEnumerator>);
```

Waiting for at least one of several moroutines to complete Using the `WaitForAny` class, you can wait for at least one moroutine from the specified list to be executed.

```
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!");
    }
}
```



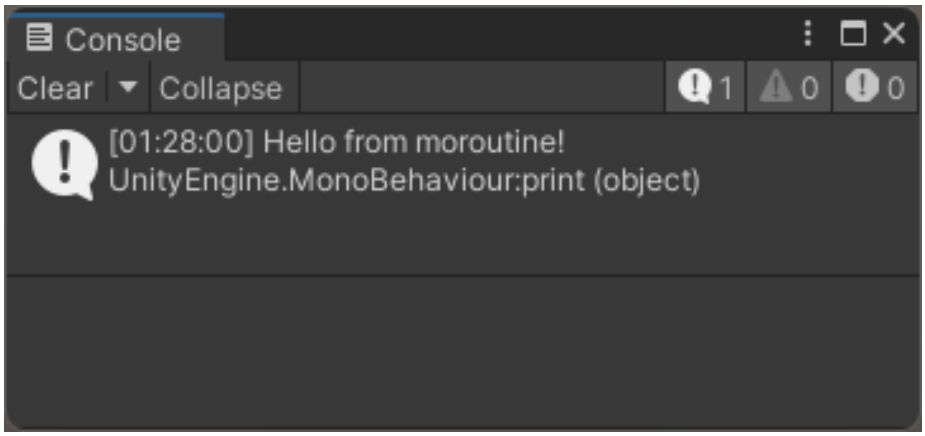
The `WaitForAny` class constructor has the following overloads:

```
WaitForAny(params Moroutine[]);
WaitForAny(IEnumerator[]);
WaitForAny(IEnumerable<IEnumerator>);
```

Moroutine result

Moroutines can store the result of execution. There is a `LastResult` property for this. This property stores the object returned by the last executed `yield return` statement.

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

    print(mor.LastResult); // output 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 the morout
}
```



Sometimes it can be very convenient!

Orphaned moroutines

So far, we have been learning how to create orphan moroutines. An ownerless moroutine is a moroutine that is not attached to any game object. The execution of such a moroutine can only be interrupted using the `Stop`, `Reset` or `Destroy` methods. Orphaned moroutine continue to exist and execute even if you load a new scene in the game, so you need to be more attentive to them. Under the hood, an unowned moroutine is launched on a game object located in the DontDestroyOnLoad scene.

Moroutines and their owners

You can associate a moroutine with any game object, that is, make it the owner of the moroutine. This means that the execution of the moroutine will only be possible if the owner object is active, otherwise the moroutine will be stopped and you will not be able to restart it or continue until the owner object becomes active. An attempt to run a moroutine on an inactive owner object will throw an exception. If the owner object is active again, then you can continue the execution of the moroutine using the `Run` method.

To specify the owner of a moroutine, 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
```

Instead of an owner object, you can pass any of its components. The result will be the same.

```
var mor = Moroutine.Run(this, CountEnumerable()); // this is a link to the current componen
```

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()); // start moroutine

yield return new WaitForSeconds(1f); // wait a second
mor.SetOwner(otherGameObject); // set a different owner

yield return new WaitForSeconds(1f); // wait a second
mor.MakeUnowned(); // made ownerless
```

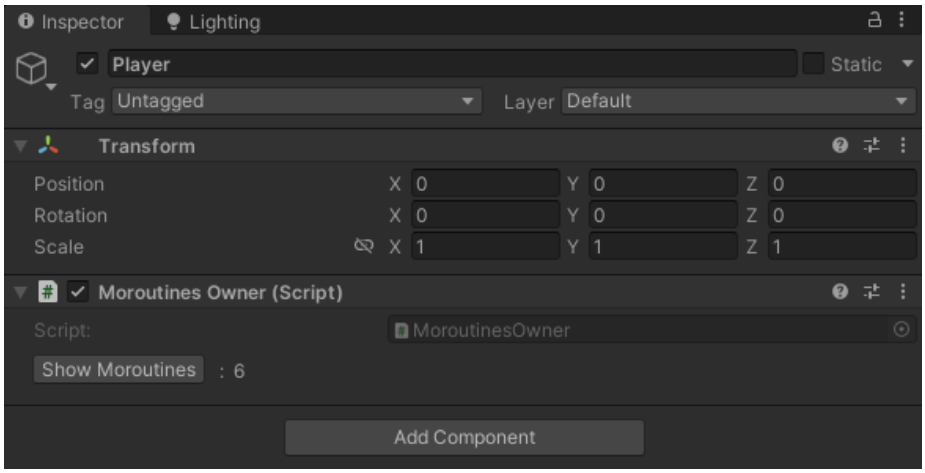
Use `this` instead of `gameObject` as it is shorter.

You can also use `mor.SetOwner((GameObject)null)` to make a moroutine orphan.

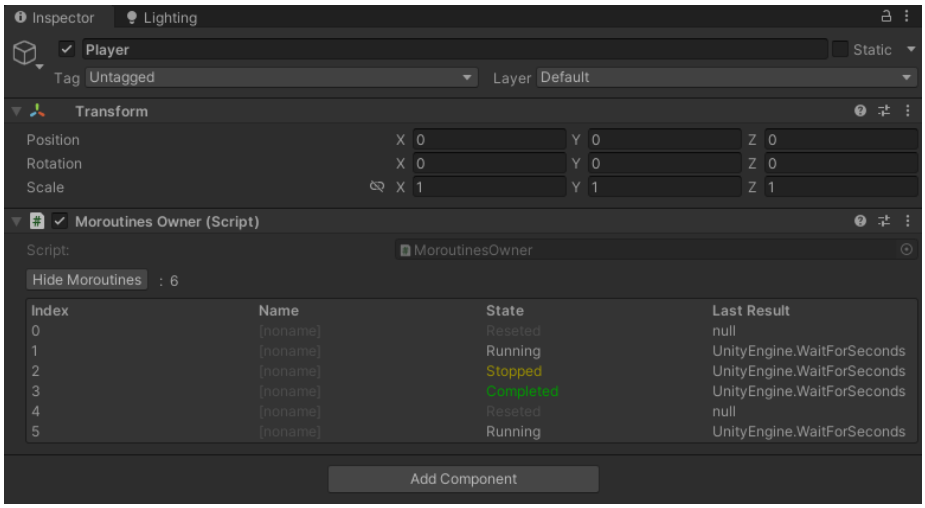
When `SetOwner` is called during the execution of a moroutine, its owner will be silently changed to the specified one.

`MoroutinesOwner` component

Whenever a game object is specified as the owner of a moroutine, a `MoroutinesOwner` component is automatically added to it. This component always keeps an up-to-date list of moroutines it owns, and is also responsible for deactivating them when the game object is disabled or deleted.



Opposite the "Show Moroutines" button, the number of actual moroutines owned by this game object is indicated. Clicking on this button will display detailed information about all current moroutines (their indexes in the list, names, status and last result):



If you need to get the owner of a moroutine, use its `Owner` property.

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

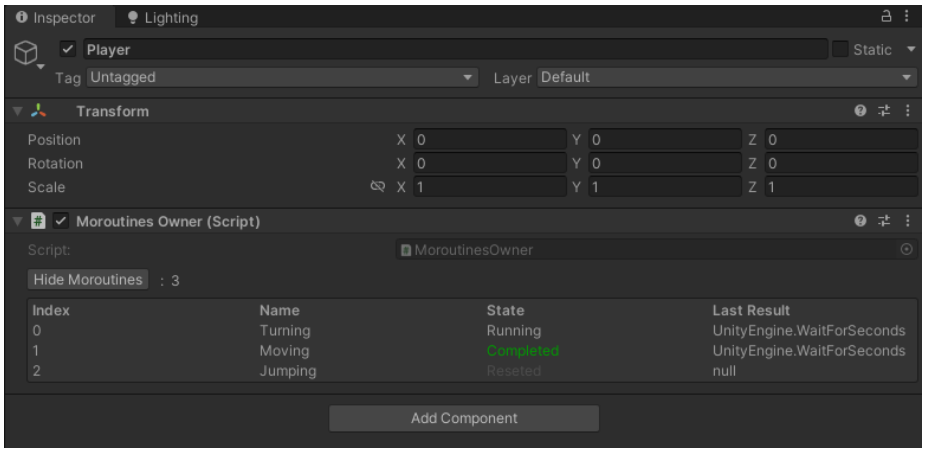
`Owner` is a reference to the `MoroutinesOwner` component of the owner of the moroutine. An ownerless moroutine has `Owner` equal to `null`. If you need to know if a moroutine is owned or unowned, use the boolean property `IsOwned`.

> The `MoroutinesOwner` component will exist as long as it has at least one actual moroutine. Destroyed moroutine is not relevant.

Moroutine names

You may have noticed that the `MoroutinesOwner` component has "[noname]" in the "Names" column of the list of moroutines. This means that you have not given a special name to these moroutines. Most of the time your moroutines will remain unnamed, but sometimes for debugging purposes it is convenient to give one of them a name. To do this, use the `SetName` method or the `Name` property of the moroutine:

```
Moroutine.Run(_owner, TurningEnumerator()).SetName("Turning");
Moroutine.Run(_owner, MovingEnumerator(2f)).Name = "Moving";
Moroutine.Create(_owner, JumpingEnumerator()).SetName("Jumping");
```



Get all the owner's moroutines

Given a `MoroutinesOwner` bean, you can get a list of its actual moroutines using the `Moroutines` property:

```
var mor = Moroutine.Run(_owner, TurningEnumerator());
Moroutine.Run(_owner, MovingEnumerator(2f));
Moroutine.Create(_owner, JumpingEnumerator());

var moroutines = mor.Owner.Moroutines; // get a readonly list of moroutine
```

In the example above, 3 moroutines are created with the same owner. Using the `Owner` property, we got the owner, and then all of its moroutines. The `Moroutines` property returns a new `ReadOnlyCollection` instance, so you can't change the original.

Another way to get all the moroutines of a game object is to use the `GetMoroutines()` extension method. Don't forget to include the `Redcode.Moroutines.Extensions` namespace:

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

Moroutine.Run(this, TickEnumerable(1), TickEnumerable(2)); // started 2 moroutines on the cu
var mors = gameObject.GetMoroutines(); // got all moroutines of the current game object
```

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

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

`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 ownerless moroutines.

Getting all orphaned moroutines

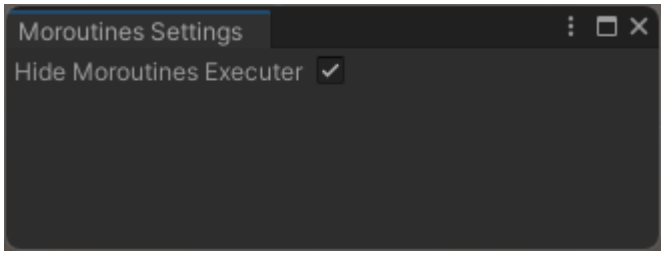
Use the `Moroutine.GetUnownedMoroutines` static method to get unowned moroutines. You can also use a state mask.

```
var mors = Moroutine.GetUnownedMoroutines(Moroutine.State.Running); // get all orphan runnin
```

Moroutines Settings

The settings window is located in the menu `Window` > `Moroutines` > `Settings` and consists of the following items:

- Hide Moroutines Executer - do I need to hide the owner of all orphaned moroutines in the editor or not?



Grouping moroutines with `MoroutinesGroup`

Sometimes it is convenient to group moroutines into one object and manage them all through it. To do this, use the `MoroutinesGroup` class.

```
var mor1 = Moroutine.Create(TickEnumerator(1));
var mor2 = Moroutine.Create(TickEnumerator(3));

var group = new MoroutinesGroup(mor1, mor2);
group.Run();
```

You can also pass a list of moroutines to the constructor:

```
var mors = Moroutine.Create(TickEnumerator(1), TickEnumerator(3));

var group = new MoroutinesGroup(mors);
group.Run();
```

Moreover, you can include the `Redcode.Moroutines.Extensions` namespace and use the `ToMoroutinesGroup()` method on a collection of moroutines to create a group from it:

```
var group = Moroutine.Create(TickEnumerator(1), TickEnumerator(3)).ToMoroutinesGroup();
group.Run();
```

Group management Use the `Run`, `Stop`, `Reset`, `Rerun` and `Destroy` methods to control the whole group. Calls to these methods essentially just call these methods on each moroutine of the group:

```
var group = Moroutine.Create(TickEnumerator(1), TickEnumerator(3)).ToMoroutinesGroup();

group.Run();
yield return new WaitForSeconds(1f);

group.rerun();
```

Changing the composition of the group The list of moroutines that make up the group is represented by the `Moroutines` property. This is a regular list, you can work with it just like any other list:

```
group.Moroutines.Add(mor7);
group.Moroutines.RemoteAt(4);
```

Set the owner for all moroutines in the group To set the owner of all moroutines in a group, use the `SetOwner` or `MakeUnowned` methods. You can also use the `Owner` property to get the owner of all moroutines, however if there is at least one moroutine whose owner is different, then this property will return `null`.

```
var group = Moroutine.Run(TickEnumerator(1), TickEnumerator(3)).ToMoroutinesGroup();
group.SetOwner(_owner);

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

State of all moroutines You can use any of the following properties to determine if all moroutines in a group match this state:

- `IsReseted` - all moroutines are reset.
- `IsRunning` - all moroutines are running.
- `IsStopped` - all moroutines are stopped.
- `IsCompleted` - all moroutines are completed.
- `IsDestroyed` - all moroutines are marked as destroyed.
- `IsOwned` - all moroutines have an owner.

Destruction of all moroutines and setting auto-destruction Use the `AutoDestroy` property or the `SetAutoDestroy` method to read/set the autodestroy behavior for the entire group. If you want to fundamentally destroy all moroutines, use the `Destroy` method.

Group events When you call group management methods (`Run`, `Stop` and others), the corresponding event is fired in the group. The following events are supported:

- `Reseted` - all moroutines of the group are reset.
- `Running` - all moroutines of the group are running.
- `Stopped` - all moroutines of the group are stopped.
- `Destroyed` - all moroutines of the group are destroyed.

You can quickly subscribe to these events using the `OnReseted`, `OnRunning`, `OnStopped` and `OnDestroyed` methods, respectively:

```
var group = Moroutine.Create(TickEnumerator(1), TickEnumerator(3)).ToMoroutinesGroup();
group.OnStopped(g => print("Stopped")).Run(); // subscribed to the stop event of the entire

yield return new WaitForSeconds(2f);

group.Stop(); // stop the group, the Stopped event will fire
```

Search for orphan moroutines in a group Use the `GetUnownedMoroutines` method (you can specify a search mask) to get a list of unowned moroutines in the group.

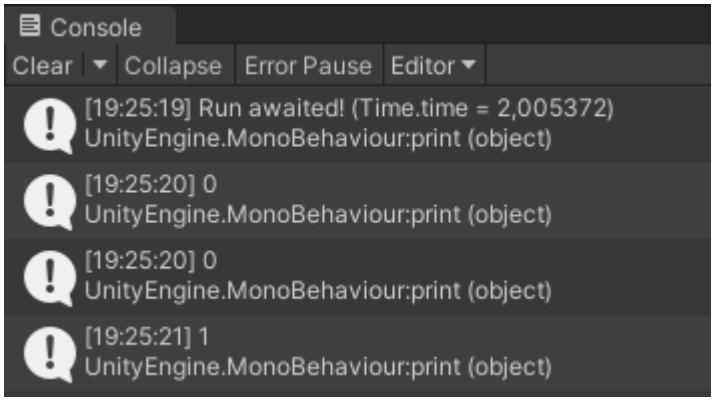
Waiting for group events You can wait for the group event you want. To do this, use one of the following methods:

- `WaitForComplete` - wait until all moroutines in the group are completed.
- `WaitForStop` - wait until all moroutines in the group are stopped.
- `WaitForRun` - wait until all moroutines in the group are running.
- `WaitForReset` - wait until all moroutines of the group are reset.
- `WaitForDestroy` - wait until all moroutines of the group are marked as destroyed.

In the example below, let's start a moroutine that will wait until the group starts executing:

```
private IEnumerator Start()
{
    var group = Moroutine.Create(TickEnumerator(1), TickEnumerator(3)).ToMoroutinesGroup();
    Moroutine.Run(WaitEnumerator(group)); // start moroutine, which will wait until the grou

    yield return new WaitForSeconds(2f); // wait 2 seconds
```

```
    group.Run(); // start the group
}

private IEnumerable TickEnumerator(int count)
{
    for (int i = 0; i < count; i++)
    {
        yield return new WaitForSeconds(1f);
        print(i);
    }
}

private IEnumerable WaitEnumerator(MoroutinesGroup group)
{
    yield return group.WaitForRun(); // wait until the group starts executing
    print($"Run awaited! (Time.time = {Time.time})");
}
```



As you can see, the `WaitEnumerator` moroutine waited until the group started executing.

Helper class `Routines`

The static class `Routines` stores the most commonly used methods for organizing the execution logic of moroutines. All methods generate and return an `IEnumerable` object that can be used by substituting into other methods. In particular, the following methods are available:

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

Example with `Delay`:

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

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

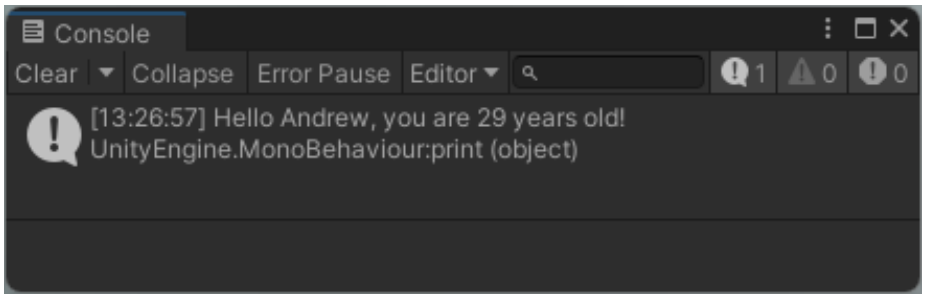
This example uses the `Delay` method, which adds a second delay before executing the `CountEnumerator` enumerator using the `Routines.Delay(1f, CountEnumerable())` line. As mentioned above, all methods of the `Routines` class return an `IEnumerable` object, therefore, in order to make a moroutine from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

The `Delay` method can also work with Action methods, which essentially gives you the ability to quickly organize the 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 shortens the code.

> These methods can work on both `IEnumerable` and `IEnumerator` objects, however, if you plan on restarting your enumerators, you must use `IEnumerable` objects.

The `FrameDelay` method adds a frame delay before executing the enumerator. For example, if you need to wait 1 game frame and then execute the enumerator

code, it would look like this:

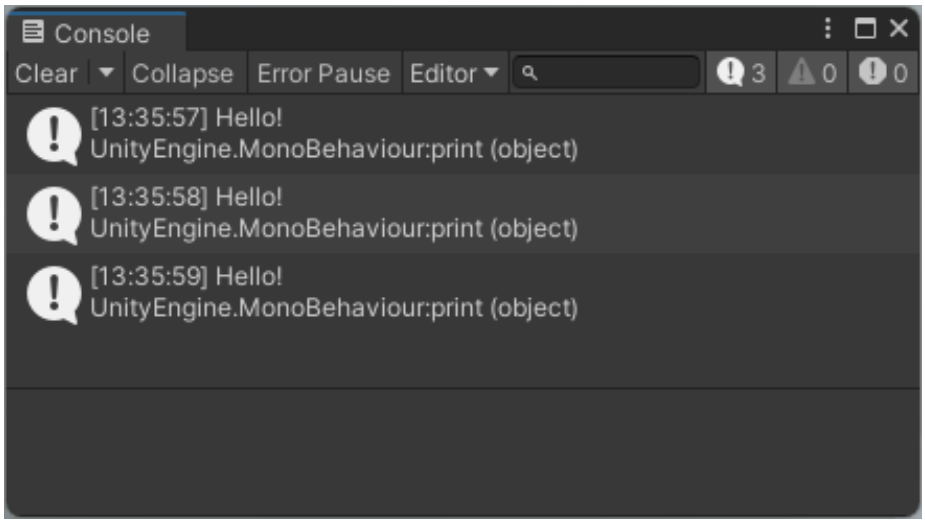
```
private void Start() => Moroutine.Run(Routines.FrameDelay(1, () => print("1 frame skipped!")
```

This method, as well as the `Delay` method, can work with Action methods.

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

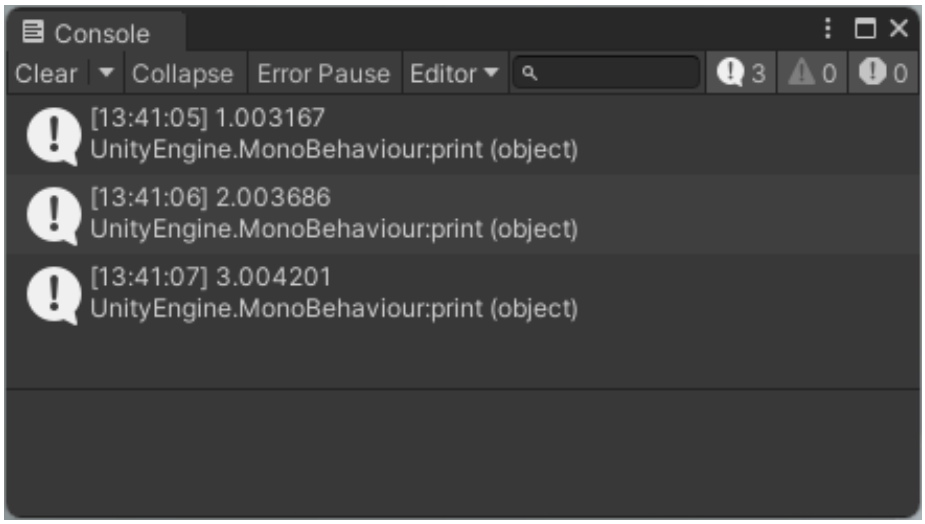
```
private void Start() => Moroutine.Run(Routines.Repeat(3, WaitAndPrintEnumerator()));

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

As a result, the text "Hello!" will be displayed in the console 3 times every second.



You can combine the `Delay`, `FrameDelay` and `Repeat` methods with each other, for example, if you need to execute a certain function 3 times with a delay of 1 second, then it will look like this:

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



Such nesting of methods in each other can be unlimited.

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

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

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

Extensions

The `Redcode.Moroutines.Extensions` namespace 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(); // Got a YieldInstruction object
var customYieldInstruction = yieldInstruction.AsCustomYieldInstruction(); // YieldInstructio
var moroutine = customYieldInstruction.AsMoroutine(); // CustomYieldInstruction converted to
```

You will most likely rarely need such a conversion, but there is a possibility.

That's all for now, you're now ready to use moroutines!