using Kore.Coroutines;

The Utils Package contains **Optimized coroutines** that can be called also outside MonoBehaviours and have a nicer syntax and faster execution, most of the functions are a replacement for Unity Coroutines however Kore still support CustomYieldInstruction (because it is used by many third party frameworks like DOTween). This document assumes you are already confident with regular Coroutines.

Kore.Coroutines allows to run coroutines generating the least possible amount of Garbage (It is really possible to have coroutines with **0 garbage** in your game), greatly helping to not add pressure on the Garbage Collector. Internally lightweight object pools are used.

List of content

Basic usage:

Advanced Usage:

Don't forget to import the namespace!

```
using Kore.Coroutines;
```

Start a new Coroutine:

```
Coroutine.Run( MyEnumerator() );
```

You can also Start coroutines to run on specified Updates:

```
Coroutine.Run( MyEnumerator(), Method.Update); //default
Coroutine.Run( MyEnumerator(), Method.FixedUpdate);
Coroutine.Run( MyEnumerator(), Method.LateUpdate);
```

Update: called after all Update methods have been called

FixedUpdate: called after all FixedUpdate methods have been called

LateUpdate: called during EndOfFrame

Nested Coroutine yield

Used to execute another coroutine and wait its completion

```
IEnumerator MyEnumerator(){
   Debug.Log("A");
   // Suspend MyEnumerator execution until AnotherEnumerator() ends
   yield return Coroutine.Nested( AnotherEnumerator());
   Debug.Log("D");
}

IEnumerator AnotherEnumerator(){
   Debug.Log("B");
   yield return null;  // wait next update
   Debug.Log("C");
}
```

OUTPUT (over 4 frames) = **A B C D**

State Change yield

Fluent and alternative way to implement a Finite State Machine using coroutines.

```
// Infinite loop of 2 alternating states
IEnumerator StateA(){
   Debug.Log("State A");
   yield return State.Change( StateB());
}

IEnumerator StateB(){
   Debug.Log("State B");
   yield return State.Change( StateA()));
}
```

OUTPUT = **A B A B A B A B A B.....**

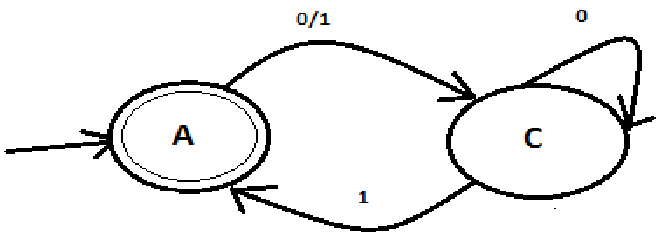
Wait For yield

This is used to wait a number of seconds based on unity Time.deltaTime/ Time.fixedDeltaTime (depending on which Method you are running the coroutine).

```
IEnumerator MyEnumerator(){
   Debug.Log("A");
   float seconds = 3;
   yield return Wait.For( seconds);
   Debug.Log("B");
}
```

Performance tips: State.Cache

Assume you want to implement the following state machine:




```
public class MyFiniteStateMachine
{
	public MyFiniteStateMachine()
	{
		Coroutine.Run( StateA() );
	}

	IEnumerator StateA()
	{
		while(true)
		{
			if(input == 1 || input == 0)
				yield return State.Change( StateC());

			yield return null;
		}
	}

	IEnumerator StateC()
	{
		while(true)
		{
			if(input == 1)
				yield return State.Change( StateA());

			if(input == 0)
				yield return State.Change( StateC());

			yield return null;
		}
	}
}
```

While I optimized the library to not generate Garbage if not strictly necessary (all yield instructions infact are recycled from a pool), I have no control over C# Runtime: each time you create a IEnumerator some garbage is generated.

The State machine syntax is very convenient for directing animations and have a very simple syntax, however if you want to use it in performance critical areas like Artificial Intelligence you have to rely on State.Cache(). That allows you to cache IEnumerators and save performance:

Compare it with the new StateMachine using State.Cache().

```
public class MyFiniteStateMachine
{
        IEnumerator A,C; // cached States
        StateCache cache; // this allows to call "Enter/Exit" methods when changin
                              // state.

        public MyFiniteStateMachine()
        {
                A = StateA();
                C = StateC();
                cache = State.Cache();

                Coroutine.Run( StateA() );
        }

        IEnumerator StateA()
        {
                while(true)
                {
                        //optionally takes a lambda: called once only when
                        //entering the state
                        yield return cache.Enter();

                        if(input == 1 || input == 0)
                                // optionally takes a lambda (called when quitting)
                                yield return cache.Change( StateC());
                }
        }
```

```
    IEnumerator StateC()
    {
        while(true)
        {
            //optionally takes a lambda: called once only when
            //entering the state
            yield return cache.Enter();

            if(input == 1)
                // optionally takes a lambda (called when quitting)
                yield return cache.Change( StateA());

            if(input == 0)
                // optionally takes a lambda (called when quitting)
                yield return cache.Change( StateC());
        }
    }
}
```

Of course also the lambdas can be stored into a Action object to save some garbage generation. The performance of this is comparable to a regular StateMachine pattern, but you also have the advantage of running it from inside a coroutine, which means you will have a easier life in running sequence of actions for each state or to wait input only in predetermined moments, this allows in most cases to use less states because some states (especially the transitional only states) can be embedded directly into the coroutine allowing it to be more compact.