The Unity CoreLibrary

A collection of useful classes and extension methods for every Unity project

Cameron Reuschel

Version 0.0.1

Contents

1 Introduction

My name is Cameron Reuschel, a student of Computer Science and Games Engineering at the university of Würzburg, Germany. All of this started during my first year of Games Engineering, were we had to develop a small game in teams of three, with no previous expertise whatsoever.

Unlike most of my peers, my more than two years of Computer Science as well as experience tutoring for the Java programming course gave me plenty of knowledge about programming in general. During the course of the one-year project, I gained even more knowledge through my work as a software developer for Java Enterprise technology and my undying love for functional programming.

As far as Unity was concerned, I quickly grew uneasy with the way I had to write code in order to make things happen the way I wanted them to happen. Things like `GameObject.SendMessage()` and the non-existence of `transform.GetChildren()` made my software engineer heart bleed a little. It seemed like Unity abused the statically typed, compiled and enterprise-ready C# language so that anyone could write an unmaintainable mess of code.

Another issue is the way Unity abuses C#'s *yield return* construct for some sort of *same-thread-parallelism*. Don't get me wrong, I love coroutines. But the fact that C# does not support `async / await`, `ref` parameters or anonymous declarations of IEnumerators made me doubt my sanity.

Many of the classes and methods provided with the CoreLibrary started out as little helpers I wrote for myself in that first-year game project, and my two teammates quickly learned to appreciate all the little quality-of-life improvements.

When we started the CaveLands project, some of my colleagues quickly stumbled across the same shortcomings that I encountered some time ago. This led to me creating the `CoreLibrary`, an independent project containing all these little utilities in a *well-tested*, *well-documented* and polished form for everyone to use.

After a semester of internal usage in the CaveLands project, other Games Engineering students showed interest in using the `CoreLibrary` as well. So we decided to open source the project under the MIT license for everyone to use.

Fell free to add any issue, forgotten edge cases or feature wishes as an Issue on GitHub.
I'll also gladly accept any pull requests.

Enjoy.

2 Showcase

Coming soon.

3 CoreLibrary Basics

3.1 Requirements

The CoreLibrary comes with no dependencies or requirements other than *Unity 2018.2* or higher.

Everything should work with *C#4 (.NET 3.5)* and further versions out of the box. However, all following examples assume you use *C#6 (.NET 4.x)* or higher.

You can work with any editor or IDE, however I *highly recommend using either Visual Studio with JetBrains ReSharper or the JetBrains Rider IDE* in order to make full use of the supplied annotations and keep code quality high.

(I am not affiliated with Microsoft or JetBrains in any way)

3.2 Usage

Paste the `CoreLibrary` folder from the GitHub repo into your `Scripts` folder or download from the Asset Store. Then add the line

```
using CoreLibrary;
```

to the top of your file. This will usually happen automatically when you extend `BaseBehaviour` instead of `MonoBehaviour`. For now everything is included in this single namespace, in order to ease the usage of the included extension methods.

3.3 General Notes

For flexibility reasons, nothing in the CoreLibrary ever returns a `List<T>` or `HashSet<T>`. Instead, `IEnumerable<T>` and `IEnumerator` are used. Everything is *lazy*, meaning that no sequence operation (with the exception of `ForEach`) calculates anything before the value itself is required. This makes extensions such as `.Collect` as well as `.AndAlso` feel like natural extensions to the LINQ framework.

*An `IEnumerable<T>` can only be safely traversed **once**.*

When you want to safely traverse an `IEnumerable<T>` multiple times, you have to call `.ToList()` on it.

3.4 BaseBehaviour

Every component in Unity extends the class `UnityEngine.MonoBehaviour`. However, if you want all features from the CoreLibrary you should extend `CoreLibrary.Base.BaseBehaviour` instead, which itself extends `MonoBehaviour`. `BaseBehaviour` lets you use all generic utilities such as the mathematically correct modulus `Mod(x,y)` as well as shortcuts to `SetPerceivable`, `AssignComponent` and `AssignIfAbsent` described later.

3.5 Utility extensions

`SetPerceivable(bool state)` redirects to the extension method on `GameObject` and enables you to make an object *unperceivable*, meaning that all `Collider`s and `Renderer`s are inactive, but the object itself isn't. This allows audio tracks and coroutines played on the object to *end properly*. However, lifecycle methods such as `Update()`, `FixedUpdate()` etc. are still called, causing a potential performance loss when overusing.

`Transform.GetChildren()` is something urgently missing from Unity. Without it, you have to use something along the lines of:

```
var children = new List<Transform>();
for (var i = 0; i < t.childCount(); ++i) children.Add(t.GetChild(i))
```

In order to efficiently chain further LINQ queries (such as `.Find`, `.Where` etc.) the method returns an `IEnumerable<T>`, which is only **traversable once**.

3.6 Vector extensions

Imagine you have an object and you want to always have it at the same Y position as some other object. For some reason, you also want it's Z position to double every 2 seconds. Let's look at this in regular Unity:

```
private IEnumerator Start()
{
    while (true)
    {
        yield return new WaitForSeconds(2);
        this.transform.position = new Vector3(
            this.transform.position.x,
            this.transform.position.y,
            this.transform.position.z * 2);
    }
}

private Transform _other;
private void Update()
```

```
{
    // can't just set position.y since unity at design here
    this.transform.position = new Vector3(
        this.transform.position.x,
        _other.position.y,
        this.transform.position.z);
}
```

In order to get rid of this repetitive hassle once for all, we provide extension methods for both `Vector2` and `Vector3` to either set or transform any individual coordinate without modifying the original vector:

```
private IEnumerator Start()
{
    while (true)
    {
        yield return new WaitForSeconds(2);
        this.transform.position = this.transform.position.WithZ(z => z
            * 2);
    }
}

private Transform _other;
private void Update()
{
    // can't just set position.y since unity at design here
    this.transform.position =
        this.transform.position.WithY(_other.transform.position.y);
}
```

This makes working with vectors in an immutable (= **safer**) manner a lot more comfortable. Note that methods such as `v.WithXY(w.x, w.y)` are *not* provided, as that would be equal to `w.WithZ(v.z)`. In cases where two coordinates are not from the same source, keeping the `With`? calls separate causes more understandable code.

3.7 LINQ extensions

LINQ is a lovely framework that brings the *mapreduce* paradigm to C# in an efficient and SQL-ish way. With it you can replace almost any

```
for (int i=0;i<n;++i)
```

loop once and for all, which increases the expressibility of your code while making it a lot less error-prone. Everything in LINQ is internally implemented as simple Coroutines with `yield`. You can probably write it yourself.

As great as LINQ is, some heavy use cases are still missing.

`Enumerable.ForEach(Action<T> action)` replaces the need to save the result of a lovely chain of LINQ calls into a variable only to write a clunky `foreach` loop. It's not for everyone, but I think it can make the code more overviewable.

`Enumerable.Collect(Func<T, TRes> mapping)` is equal to `Enumerable.Select(mapping).Where(v => v != null)`. It applies the `mapping` function to every element in the Enumerable, yielding a new Enumerable with the results. It also filters out every `null` value. Useful for `Select`ing with a mapping that may fail, such as `v.As<Whatever>()`.

`Enumerable.CollectMany(Func<T, IEnumerable<TRes>> mapping)` is a version of collect that maps each value in the Enumerable to a new Enumerable and then flattens the nested Enumerables into one flat Enumerable, in order. It also filters out nulls. Calls LINQ's `SelectMany` in the background.

`Enumerable.AndAlso(IEnumerable other)` merges together two LINQ "streams" of the *same type* and returns a new `IEnumerable`.

`Enumerable.Shuffle(System.Random random = null)` does exactly what it says it does: It returns a shuffled version of the enumerable using the Fisher-Yates algorithm. **It does not modify the original Enumerable**. You may pass your own instance of `System.Random` if you want to achieve a deterministic behavior (in Unity? HAHAHAHA). Otherwise the algorithm uses it's own. *This randomness is not cryptographically secure so don't even think about it*.

3.8 Singleton

It sometimes happens that an object should only be present *once per scene*. Such an object is called a `Singleton`. Often, this single object holds some importance and is often referenced from other code. An example of this is the `Player` component. Usually getting this code involves a very expensive call to `FindObjectOfType<T>()` in every referencing component, which can be devastating for scene load times if many objects call this. A simpler solution would be to store a reference to the one instance of the component in the *static context* of the class itself.

```
public class Player : MonoBehaviour
{
    public static Player Instance { get; private set; }

```

```
    private void Awake()
    {
        if (Player.Instance != null)
            throw new Exception("Already a player component in the
                scene!");
        Player.Instance = this;

        /* ... */
    }
}
```

Typing this out every time is tedious and also leads to easy errors (e.g. when someone accidentally writes != instead of ==). For this case, we provide a class:

```
abstract class CoreLibrary.Base.Singleton<T> : BaseBehaviour where T :
    Singleton<T>
```

The `where`-clause guarantees that the parameter `T` is the type of the implementing class itself. This is necessary to ensure safe typing for the `public static T Instance` property provided. Extending the class itself is enough for full singleton functionality:

```
public class Player : Singleton<Player> { /* ... */ }

// somewhere else
Player.Instance.Hp -= 5;
```

Some hints:

- There is no need to save the reference to `Player.Instance` during some call to `Start` or `Awake`, as the call itself is efficient enough.
- However, if typing out `Player.Instance` every time annoys you, you may define a getter somewhere in the class like `private Player pl => Player.Instance;`

3.9 LazySingleton

While `Singleton` throws an exception when it can't find an object, `LazySingleton` instantiates a new empty `GameObject` with only the requested component attached to it. This is especially useful for objects that are required in every scene but are easily forgotten. It otherwise behaves exactly like `Singleton`.

4 Coroutine Composition Utilities

4.1 Coroutine Basics

Consider the following example why coroutines can be useful: You want to have an object that may be launched only once at some point, making it fly for exactly a specified number of seconds:

```
// without coroutines

[RequireComponent(typeof(Rigidbody))]
public class BottleRocket : MonoBehaviour
{
    public float FlightTime;

    private bool _launched = false;
    private float _launchTime = 0;

    private Rigidbody _rb;

    private void Start()
    {
        _rb = GetComponent<Rigidbody>();
    }

    public void Launch()
    {
        if (_launched) return; // can't relaunch
        _launched = true;
        _launchTime = Time.time;
    }

    private void FixedUpdate()
    {
        var endTime = _launchTime + FlightTime;
        if (_launched && Time.time < endTime)
        {
            _rb.AddForce(Vector3.up * 1000); // FLY!
        }
    }
}
```

Basically, coroutines in the context of games are functions that execute “asynchronously” over multiple frames once started. In the above example, the launch code in `FixedUpdate()` only runs under special conditions: Only when the component has been launched (`_launched == true`) and less than `FlightTime` seconds have passed since launch. All the state required for even this very simple case is difficult to overview and therefore error-prone. And even worse, `endTime` is calculated every `FixedUpdate` over the lifetime of the object! Ouch.

```
// with coroutines

[RequireComponent(typeof(Rigidbody))]
public class BottleRocket : MonoBehaviour
{
    public float FlightTime;

    private bool _launchedOnce = false;
    private Rigidbody _rb;

    private void Start()
    {
        _rb = GetComponent<Rigidbody>();
    }

    public void Launch()
    {
        StartCoroutine(LaunchRoutine());
    }

    private IEnumerator LaunchRoutine()
    {
        if (_launchedOnce) yield break; // can't relaunch!
        _launchedOnce = true;

        // launch time stays local
        var _launchTime = Time.time;

        while (_launchTime + FlightTime < Time.time)
        {
            _rb.AddForce(Vector3.up * 1000); // FLY!
            yield return new WaitForFixedUpdate();
        }
    }

    private void FixedUpdate()
```

```
    {
        /* unnecessary */
    }
}
```

Having a method return `IEnumerator` enables the use of `yield` statements. A `yield return` lazily adds a value to the returned IEnumerator. In the context of game development with Unity, it usually returns a `YieldInstruction`: either `null` for *wait for next frame*, `new WaitForFixedUpdate()` to pause until the next fixedUpdate or `new WaitForSeconds(float)` to pause executions for a number of seconds before continuing the code, among others. Execution of a coroutine method is paused once it reaches a `yield return` and continued once the next value is requested.

It is important to note that just calling `LaunchRoutine` returns an `IEnumerator`, a list of values that are not yet computed. Before calling `enumerator.MoveNext()`, no code is executed. Unity handles these `.MoveNext()` calls internally and uses the returned `YieldInstruction`s to pause execution as specified by the returned value. For this to work, the coroutine has to be passed to a call of `StartCoroutine`. In the CoreLibrary, we provide the `coroutine.Start()` function, which is just a shorter way for writing `StartCoroutine` with less braces.

TL;DR: Never forget to call `StartCoroutine()` or `.Start()`!

Because forgetting the `StartCoroutine` call is such a common mistake that can't be caught by the type system, it is a *good design practice* to make the coroutine implementation private and provide a public method (here `Launch()`), that just takes care of starting the coroutine.

Now imagine multiple of these method-implementation pairs in a single component: Without a lot of discipline this quickly becomes mess. Especially when you are having a lot of `private IEnumerator ...` methods that just wait a few seconds before executing code. For reasons like this we provide a number of *coroutine control and composition structures*. These enable you to replace almost all `private IEnumerator ...` methods with *anonymous functions (lambdas)*. Consider the above example with the Core Library:

```
// with CoreLibrary features

using static CoreLibrary.Coroutines;

[RequireComponent(typeof(Rigidbody))]
public class BottleRocket : BaseBehaviour
{
    public float FlightTime;

    private bool _launchedOnce = false;
    private Rigidbody _rb;
```

```

    private void Start()
    {
        AssignComponent(out _rb);
    }

    public void Launch()
    {
        if (_launchedOnce) return; // can't relaunch!
        _launchedOnce = true;

        // launch time stays local
        var _launchTime = Time.time;

        RepeatWhile(
            () => _launchTime + FlightTime < Time.time,
            () => _rb.AddForce(Vector3.up * 100),
            fixedUpdate: true
        ).Start();
    }

    private void FixedUpdate()
    {
        /* unnecessary */
    }
}
```

The `using static CoreLibrary.Coroutines;` lets you omit the `Coroutines.` prefix in front of every coroutine-related function like the `RepeatWhile` seen above. *All further examples assume you have this line somewhere at the top of your source file.*

The notation `() => doSomething()` is called an *anonymous function* or *lambda*. It is a function object taking no arguments (hence the `()`) and when called executes the code on the right of the `=>`. The core library uses lambdas almost exclusively as **code blocks to execute later**. In the case `RepeatWhile` one must pass two lambdas: the first returns a boolean and is evaluated again before every loop, breaking the loop if it returns false. The second lambda can return whatever: It holds the body of loop.

An alternative to writing `() => doSomething()` is `() => {return doSomething();}`. By using curly braces after the arrow you can use more than one statement in a lambda, using them to pass more complex blocks of code to our custom control structures.

The `fixedUpdate: true` is a named parameter. In the case above we could just write `true`, but prefixing boolean parameters with the argument name makes the code more readable. Every control structure that executes code every frame has an optional `fixedUpdate` parameter as last argument which defaults to false.

The rest of this page explains our coroutine utilities by first presenting some regular Unity code followed by what you would replace it with.

Hint: No coroutine-related function in the CoreLibrary actually *starts* the coroutine. They are for composing new IEnumerables which may be later started by calling `enumerable.Start()`.

4.2 StartCoroutine() vs. .Start()

When using `StartCoroutone(myRoutine)`, the execution of the coroutine is *registered to the game object that started it*. This means that when the game object is destroyed or *set inactive*, the coroutine ends. This might be the desired behaviour when you want to implement an infinite coroutine.

In contrast, calling `myRoutine.Start()` registers the execution of the coroutine to a *global CoroutineRunner*. Coroutines started this way only stop when the scene ends.

4.3 DelayForFrames

Imagine you have a button. Every time the button is pressed, a sound should play after exactly `n` frames. If you press the button multiple times before the first sound plays all sounds should play. Building this without coroutine would require a custom queue that keeps track of frame count and similar.

```
// -- before
public void PlaySound(uint n)
{
    StartCoroutine(DelayPlayingSound(n));
}

private IEnumerator DelayPlayingSound(uint n)
{
    for (var i = 0; i < n; ++i) yield return null;
    sound.Play();
}
```

Now, for every delayed action you'll have to write another method-coroutine-pair, making your code a mess. Consider instead:

```
// -- after
public void PlaySound(uint n) => DelayForFrames(n, () => sound.Play()).
    Start();
```

C# 6.0 allows you to shorten single-line function bodies into a `declaration => expression;` for readability. This is completely optional, but shorter. This notation is especially useful for providing a lot of simple methods without polluting the class with way too many lines holding only curly braces.

4.4 RepeatForFrames

Sometimes instead of waiting a number of frames you want to repeat an action for a number of continuous frames. This is mostly useful for a small number of frames, like executing an action three times with a frame between each. To be honest I can't think of a useful example, so here is an abstract one:

```
// -- before
public void Foo()
{
    StartCoroutine(FooRoutine());
}

private IEnumerator FooRoutine()
{
    DoStuff();
    yield return null;
    DoStuff();
    yield return null;
    DoStuff();
}
```

```
// -- after
public void Foo() => RepeatForFrames(3, () => DoStuff());
```

4.5 WaitForSeconds

A very common case is to wait a certain time before executing some action. If fact it is so common that Project Synergy had at least 6 single `private IEnumerator ...` methods just to wait before doing something.

Imagine you just defeated a boss. You want some stuff to happen like sounds or light effects. Then, after exactly 5 seconds a door should open.

```
// -- before
public void InitiateDoorOpening()
```

```
{
    /* Do other stuff like play sound */
    StartCoroutine(OpenDoorRoutine());
}

private IEnumerator OpenDoorRoutine() {
    yield return new WaitForSeconds(5);
    door.Open();
}
```

```
// -- after
public void InitiateDoorOpening()
{
    /* Do other stuff like play sound */
    WaitForSeconds(5, () => door.Open()).Start();
}
```

4.6 RepeatForSeconds

Another very common case is to repeat an action every frame for a number of seconds. The example for this case is the example at the beginning of this page. No need to copy-paste.

4.7 RepeatEverySeconds

The method `RepeatEverySeconds(float interval, CodeBlock action, int? repetitions = null)` differs from the other repeat methods. It exists solely for the use case of repeating a block of code (`action`) every `interval` seconds, either until the coroutine is stopped or until the code has been executed `repetitions` times.

Imagine a simple component that does nothing but cause a light to blink every `BlinkInterval` seconds. But only for `MaxBlinks` times **or** until it is `.Stop()`ed. For example when coding a christmas tree, or whatever. I don't care.

```
// -- before
public float BlinkInterval;
public int MaxBlinks;

private bool _stopped = false;
public void Stop() => _stopped = true;

```

```
private void Blink() { ... }

private float _lastBlinkTime;
private int _doneBlinks;

private void Update()
{
    if (_stopped || _doneBlinks >= MaxBlinks) return;
    _doneBlinks += 1;

    if (Time.time - _lastBlinkTime > BlinkInterval)
    {
        _lastBlinkTime = Time.time;
        Blink();
    }
}
```

```
// -- after
public float BlinkInterval;
public int MaxBlinks;

private bool _stopped = false;
public void Stop() => _stopped = true;

private void Blink() { ... }

private IEnumerator Start() =>
    RepeatEverySeconds(BlinkInterval, () => Blink(), MaxBlinks)
    .YieldWhile(() => !_stopped);
```

The only **disadvantage** of using `RepeatEverySeconds` is that both the `interval` and the number of `repetitions` **cannot be changed** afterwards.

4.8 WaitUntil

It sometimes happens that you want to *schedule* some code for execution once a certain condition becomes true. For example, you are building an RPG and the player just defeated a boss, but you only want him/her to progress once he/she reaches a certain level. Then the door to the next area opens.

```
// -- before
public class NextAreaDoor : MonoBehaviour
```

```
{
    public BossEnemy Boss;
    public int MinLevel;
    private bool _opened = false;
    private void Update()
    {
        if (!_opened
            && Boss.IsDefeated
            && Player.Instance.Level >= MinLevel)
        {
            GetComponent<Door>().Open();
            _opened = false;
        }
    }
}
```

In this code, we don't even use a coroutine. Some door component keeps checking every frame whether it can finally open. It's a whole file just dedicated to this single piece of logic. You can't put it into the boss logic because the player might not have the required level after defeating him. Or can you?

```
// -- after
public class FirstAreaBoss : BossEnemy
{
    public Door NextAreaDoor;
    public int MinLevel;

    public float Hp { get; private set; }

    /* some other fields and state and methods and stuff */

    public void TakeDamage(float amount)
    {
        Hp -= CalcActualDamage(amount);
        if (Hp > 0)
        {
            /* animations, sound... */
        }
        else
        {
            /* death animations, sound... */
            WaitUntil(
                () => Player.Instance.Level >= MinLevel,
                () => Door.Open()
```

```
            ).Start();
        }
    }
}
```

Now the repeated level checks are abstracted into an anonymous coroutine that runs in parallel until it is ready. One whole file less.

4.9 RepeatWhile

Sometimes you want to repeat something until a certain condition is met. Consider a heat tracking missile that follows a target until it hits it.

```
// -- before

public Transform Target;

public float SpeedInMPS = 100;

public void LockAndFollow()
{
    StartCoroutine(FollowRoutine());
}

private IEnumerator FollowRoutine()
{
    while (Vector3.Distance(Target.position, transform.position) < 5)
    {
        var goalDir = Target.position - transform.position;
        var dir = Vector3.Slerp(transform.forward, goalDir, .2f);
        transform.forward = dir;
        transform.position +=
            transform.forward.normalized * SpeedInMPS * Time.deltaTime;
        yield return null;
    }
    Explode();
}
```

```
// -- after

public Transform Target;
```

```

public float SpeedInMPS = 100;

public void LockAndFollow()
{
    RepeatWhile(() => {
        return Vector3.Distance(Target.position, transform.position) <
            5;
    }, () => {
        var goalDir = Target.position - transform.position;
        var dir = Vector3.Slerp(transform.forward, goalDir, .2f);
        transform.forward = dir;
        transform.position +=
            transform.forward.normalized * SpeedInMPS * Time.deltaTime;
    }).Afterwards(() => Explode());
}
```

`RepeatWhile` does *not* execute the passed action when the condition is already **`false`**.

4.10 Do

For completeness, the CoreLibrary provides `IEnumerable Do(YieldAction action)`. A `YieldAction` is a function which takes no arguments and returns something, whatever it is. `Do` turns this function into a coroutine: It executes the function and `yield` **`return`**s the result of the function. The use of `Do` can mostly be avoided by using the right design.

`Do` is a constructor for a singleton coroutine: One that only generates a single result.

4.11 .Afterwards and .YieldWhile

The above example also shows the usage of `.Afterwards(code)`, which can be called on an existing IEnumerator (= not started coroutine). The passed code is executed after the coroutine it is called on runs out, regardless of how it ends. **Even if an exception is thrown!** `.Afterwards` is often used for some cleanup code. The resulting IEnumerator must still be passed to `StartCoroutine()` or `.Start()`. The passed code takes the form of a lambda without arguments.

Additionally we provide a `.YieldWhile(()=> condition)` extension method to interrupt existing IEnumerators early. The resulting IEnumerator checks the condition before every evaluation and if **`false`** stops the whole coroutine. It can be useful when used in combination with `WaitForSeconds`, where the delayed action is only executed when the passed condition still holds true.

Imagine you have some function `IEnumerator OpenDoor()` that rotates some door until it is open. There is also a function `IEnumerator CloseDoor()` that does the opposite: Rotate the door until it is closed. Of course we want to be able to interrupt the opening and start closing the door at any point.

```
// -- before

var isOpening = false;
var isClosing = false;

public void Open() => InterruptibleOpenRoutine();

private void InterruptibleOpenRoutine()
{
    isClosing = false;
    isOpening = true;
    var handler = StartCoroutine(OpenDoor());
    while (isOpening) yield return null;
    StopCoroutine(handler);
    isOpening = false;
}

public void Close() => InterruptibleCloseRoutine();

private void InterruptibleCloseRoutine()
{
    isOpening = false;
    isClosing = true;
    var handler = StartCoroutine(CloseDoor());
    while (isClosing) yield return null;
    StopCoroutine(handler);
    isClosing = false;
}

public void HoldTheDoor()
{
    isOpening = false;
    isClosing = false;
}
```

This implementation is already pretty damn smart: It uses coroutines that start another coroutine and keep track of the state in parallel, stopping and cleaning up afterwards. But be honest, *would you have written the code this way?* Still, it can be better:

```
// -- after

private bool _isOpening = false;
private bool _isClosing = false;

public void Open()
{
    _isClosing = false;
    _isOpening = true;
    OpenDoor()
        .YieldWhile(() => _isOpening)
        .Afterwards(() => _isOpening = false).Start();
}

public void Close()
{
    _isOpening = false;
    _isClosing = true;
    CloseDoor()
        .YieldWhile(() => _isClosing)
        .Afterwards(() => _isClosing = false).Start();
}

public void HoldTheDoor()
{
    _isOpening = false;
    _isClosing = false;
}
```

Everything happens exactly as the code tells you. In my opinion everyone should see this code and instantly understand what is happening. Open the door as long as it is opening and afterwards set the flag to false, just in case. No weird coroutine handling tricks. It's really hard to make errors.

Bonus: If you can't keep track of multiple boolean variables, implement a simple state machine:

```
// outside of the behaviour
public class DoorOpenState
{
    public IsOpening { get; private set; } = false;
    public IsClosing { get; private set; } = false;
    public void Open()
    {
```

```
        IsOpening = true;
        IsClosing = false;
    }
    public void Close()
    {
        IsOpening = false;
        IsClosing = true;
    }
    public void Hold()
    {
        IsOpening = false;
        IsClosing = false;
    }
}

// later in behaviour
private DoorOpenState _doorState = new DoorOpenState();
private bool IsOpening => _openState.IsOpening();
private bool IsClosing => _openState.IsClosing();

public void Open()
{
    _doorState.Open();
    OpenDoor()
        .YieldWhile(() => _isOpening)
        .Afterwards(() => _doorState.Hold()).Start();
}

public void Close()
{
    _doorState.Close();
    CloseDoor()
        .YieldWhile(() => _isClosing)
        .Afterwards(() => _doorState.Hold()).Start();
}

public void HoldTheDoor() => _doorState.Hold();
```

`.YieldWhile` does *not* execute the passed action when the condition is already **`false`**.

4.12 .AndThen

`.Afterwards` lets you somehow combine multiple coroutines by calling

```
// don't do this
SomeRoutine().Afterwards(() => OtherRoutine().Start()).Start();
```

This might work for simple cases. However, it also has some downsides:

- The coroutine handler returned by the first `.Start()` has nothing to do with the execution of `OtherRoutine`.
- When you want to use additional `.Afterwards` or `.YieldWhile` calls on any routine it gets complicated quickly.
- It has `.Start()).Start()` - wtf?

That is why the CoreLibrary provides an extension method `IEnumerator.AndThen(otherRoutine)` for concatenating routines before starting them. With it, the above example is as simple as

```
// do this instead
SomeRoutine().AndThen(OtherRoutine()).Start();
```

If you still say "do you *really* need this?" consider this:

```
// don't do this
SomeRoutine().Afterwards(
    () => OtherRoutine().Afterwards(
        () => ThirdRoutine().Start()).Start()).Start();
```

```
// do this instead
SomeRoutine().AndThen(OtherRoutine()).AndThen(ThirdRoutine()).Start();
```

Note that the passed routines are not prefixed by a `() =>`. This is because IEnumerators are already *lazy* - there is no need to wrap them into a function object in order to execute them later as they are already only ever executed once they are started by `StartCoroutine` or `Do`.

Bonus: `.AndThen` can be used to delay a coroutine for some time. For example:

```
WaitUntil(() => Player.Level >= 10).AndThen( StartFirstClassQuest() ).Start();

WaitForSeconds(5).AndThen( OpenDoor() ).Start();

DelayForFrames(2).AndThen( SomeRoutineThatNeededToBeDelayed() ).Start();
```

This is the reason why the `afterwards` parameter is always optional.

4.13 DoBefore

Imagine a use case where you wanted to execute some code before using `RepeatWhile`, regardless of whether the passed condition is initially false or not. When you want to immediately start the constructed coroutine, that is no problem:

```
private bool _isRunning = false;

public void Run()
{
    _isRunning = true;
    RepeatWhile(() => _isRunning, () => DoSomething()).Start();
}
```

When you do not want to start the coroutine immediately, you can also write it like this:

```
private bool _isRunning = false;

public IEnumerator Run()
{
    _isRunning = true;
    yield return RepeatWhile(() => _isRunning, () => DoSomething());
}
```

But for the very rare case where you want the entire new coroutine in a single expression, the CoreLibrary provides `IEnumerator DoBefore(CodeBlock action, IEnumerator coroutine)`:

```
private bool _isRunning = false;

public void Run() => DoBefore(
    () => _isRunning = true,
    RepeatWhile(
        () => _isRunning,
        () => DoSomething()
    ));
```

4.14 .Flatten

For the rare cases when you want to **manually iterate through an `IEnumerator`**, the CoreLibrary provides `.Flatten()`.

In unity, when writing coroutines, it is common to `yield return anotherIEnumerator`. This other `IEnumerator` might yield other nested `IEnumerator`s as well. When the Unity runtime encounters a nested coroutine after `StartCoroutine` or the CoreLibrary's `.Start()`, it executes the whole nested coroutine first before proceeding with the current one.

Now, when we want to *manually inject code between each step of some `IEnumerator`*, we have to manually iterate through it like this:

```
IEnumerator myCoroutine = ...;
while (myCoroutine.MoveNext())
{
    if (_inactive) continue;
    if (_somethingHappened) yield break;
    yield return myCoroutine.Current;
}
```

In this artificial case, we want to not have the Unity runtime wait for whatever when `_inactive` and stop early if `_somethingHappened`. Patterns like this cause *unexpected behaviour* when encountering a nested `IEnumerator`. Consider the following code:

```
private IEnumerator SetupLaunch() { ... }
private IEnumerator Launch() { ... }
private IEnumerator PostLaunch() { ... }
private IEnumerator LaunchProcedure()
{
    yield return SetupLaunch();
    yield return Launch();
    yield return PostLaunch();
}
```

When setting `myCoroutine = LaunchProcedure();` in the first example, then the conditions are checked exactly *thrice*: once each after `SetupLaunch`, `Launch` and `PostLaunch` have completed. This is because when Unity is handed a complete IEnumerable through `yield return`, it executes the entire coroutine before continueing the above `while` loop. But what if we wanted to execute code **after every single** `yield return null` or other `YieldInstruction`, regardless of how `myCoroutine` is implemented?

For this use case, the CoreLibrary offers `.Flatten()` - for the Unity runtime, `.Flatten()` does absolutely nothing. However, the structure of *arbitrarily deeply nested* `IEnumerator`s is *flattened* into a single `IEnumerator` that is guaranteed to never `yield return` another `IEnumerator`, so that you can execute code after every single actual execution pause. This is for example used in the implementation of `.YieldWhile` to ensure that the end condition is checked as often as possible.

If you did not understand the use of `.Flatten()` yet, here's an example that breaks:

```
bool _running = true;

IEnumerator a()
{
  yield return b();
}

IEnumerator b()
{
  _running = false;
  yield return null;
  DoSomethingDangerous();
}

IEnumerator BrokenYieldWhile(IEnumerator wrapped, Condition cond)
{
    while (cond() && wrapped.MoveNext()) yield return wrapped.Current;
}

BrokenYieldWhile(a(), () => _running).Start();
```

In the above case the Unity runtime executes the whole `b()` coroutine before `BrokenYieldWhile` checks the condition and has a chance to interrupt. This means that, unintuitively, `DoSomethingDangerous` is actually called. And for this reason, we need `.Flatten()`.

5 Component Queries

5.1 Introduction

In regular Unity, one may retrieve a GameObject's components by using `gameObject.GetComponent<Renderer>()`. However, this is not only verbose but also not very flexible. For example, `GetComponent` only searches in the object itself, not in it's children or parents. For both use cases other, even more verbose methods, exist. However, these can't search more than one layer deep (I guess, it's not exactly well documented...).

Because of these shortcomings the core library provides a number of special query methods, which will be presented in the rest of this page. All of these methods have an optional parameter `Search where`, which lets you decide the scope of the search. `Search` is an enum containing the values `InObjectOnly`, `InChildren`, `InParents` and `InWholeHierarchy`. The default is always `InObjectOnly`. `InChildren` does a depth-first search through all the children until an instance of the requested component is found. `InParents` linearly traverses all parents until the scene root or until the requested component is found. `InWholeHierarchy` searches the parents first for efficiency reasons, then the children. In all cases, the object itself is searched first.

The rest of this page explains all queries by first presenting how it is usually done followed by a proper use case of the corresponding CoreLibrary query.

5.2 `AssignComponent<T>` and `AssignIfAbsent<T>`

Usually, finding other relevant components on the same game object and saving them into instance fields is a tedious but necessary task, since `GetComponent` is an expensive call and should therefore be called as little as possible. For this, we extend the `GameObject` class with two extension methods:

- `void AssignComponent<T>(out T variable, Search where = Search.InObjectOnly )where T:Component`
- `bool AssignIfAbsent<T>(ref T variable, Search where = Search.InObjectOnly )where T:Component`

Both of these methods are also handily available in `BaseBehaviour`. Consider the following code:

```
// in regular Unity code
[RequireComponent(typeof(Animator))]
public class SampleComponent : MonoBehaviour
{
    // may be in some parent object as well
    private Rigidbody _rb;
```

```

    // in this object only
    private Animator _anim;

    private void Awake() {
        // some complicated code that might set _anim
    }

    private void Start() {
        // rigidbody may be in some parent object
        var currentTransform = this.transform;
        do {
            if (_rb == null)
                _rb = currentTransform.GetComponent<Rigidbody>();
            if (_rb != null) break;
            currentTransform = currentTransform.parent;
        } while (currentTransform != null);
        if (_rb == null)
            throw new Exception("No rigidbody for " + gameObject.name);

        if (_anim != null)
        { // if already set in Awake
            /* Play some special animation */
        } else _anim = GetComponent<Animator>();
    }
}
```

```
// semantically equivalent code with our CoreLibrary
[RequireComponent(typeof(Animator))]
public class SampleComponent : BaseBehaviour
{
    private Rigidbody _rb;
    private Animator _anim = null; // for use as  ref  param

    private void Awake() {
        // some complicated code that might set _anim
    }

    private void Start() {
        // type parameter T is automatically inferred
        // an error is thrown when nothing can be found
        AssignComponent(out _rb, Search.InParents);
```

```

        if (!AssignIfAbsent(ref _anim))
        { // if already set in Awake
            /* Play some special animation */
        }
    }
}
```

`AssignComponent` takes a reference to a variable marked as `out`. This guarantees that the variable is assigned somewhere inside the function. It also infers the type `T` for you, saving a lot of unnecessary verbosity.

`AssignIfAbsent` however takes a `ref` parameter. This does not guarantee that the variable will be set, however it requires the passed variable to be explicitly assigned before passing it to the function. This is necessary to check whether the variable already contains a value. For convenience, **`true`** is returned if the passed variable was **`null`** and has been assigned, and **`false`** if no assignment happened.

`AssignIfAbsent` is particularly useful to avoid potential multiple calls to `AssignComponent`, which repeats the search every time. You can use it for loading a component lazily or *on demand*: Consider some object with an `Animator` that is only needed in one specific way of interacting with it. There are hundreds of these objects in the scene and you are only going to interact with very few of them, and with some of them multiple times. Now, calling `AssignComponent` somewhere in `Start` is wasted computing time 90% of the time. Calling `AssignComponent` every time the interaction happen s may waste valuable resources on redundant searches. In this case `AssignIfAbsent` offers an efficient solution, as the cost of re-searching is only one very cheap **`null`**-check. The `bool` return value may be ignored in this case.

5.3 `Is<T>`, `As<T>` and `All<T>`

Imagine you are an awesome flying space cat shooting your EMP laser eyes at multiple invading alien spacecraft stupidly flying in a straight row. All their antigravity drives deactivate and they fall to their deaths, exploding as they hit the hard and dry ground of the Arizona desert. **"What?"** - what?

Shooting laser rays sounds exactly like a Raycast. Now, in Unity, a RaycastAll yields a `RaycastHit[]`. Each hit contains a reference to the hit `Rigidbody` as well as the object's `Transform`. So consider the following code:

```
RaycastHit[] hits = /* ... */;

// -- regular Unity
// Fails if components are distributed over a more
```

```
// complex object hierarchy such as in complex prefabs.
// Also fails on other edge cases...
var shipsHit = hits
    .Where(h => h.transform.GetComponent<Spaceship>() != null).ToList();
var hitAnimations = shipsHit
    .Select(s => s.transform.GetComponent<Animator>())
    .Where(a => a != null);
var engines = shipsHit
    .SelectMany(s => s.transform.GetComponents<AntigravityEngine>());
foreach (var anim in hitAnimations) anim.play("hit");
foreach (var engine in engines) engine.Explode();
foreach (var hit in shipsHit) hit.rigidbody.useGravity(true);
```

```
// -- with CoreLibrary code
var shipsHit = hits
    .Where(h => h.transform.Is<Spaceship>(Search.InWholeHierarchy));
shipsHit
    .Collect(s => s.transform.As<Animator>())
    .ForEach(anim => anim.play("hit"));
shipsHit
    .SelectMany(s => s.transform.All<AntigravityEngine>(Search.InWholeHierarchy))
    .ForEach(engine => engine.Explode());
shipsHit.ForEach(hit => hit.rigidbody.useGravity(true));
```

As you can see, `Is`, `As` and `All` make the code more concise but they as well as complex searches much easier! `.Collect` is a nice shortcut to lose the null checks. `.ForEach` depends on one's taste. I personally prefer it to an additional variable and a loop.

All three methods are available as extensions to both `GameObject` and `Transform` classes for convenience.

6 Scene Query System

6.1 Introduction

When you want to find for example all `Resource`s in a scene, calling `Object.FindObjectsOfType<T>()` is *really inefficient*. Sometimes however, you need to query all objects of a type in a scene.

A possible approach to this is to write some `class ResourceManager : Singleton<ResourceManager>`, which holds a list of all Resources. Now, at the beginning of each scene, `Object.FindObjectsOfType<Resource>` is called once to fill that list.

While this solves the problem for resources, do you really want to create an additional component for *every single type you want to query*? Also, how do you manage new `Resource`s that are spawned at runtime and are not found during the initial query?

The CoreLibrary provides the `QueryableBaseBehaviour` and `Query` classes to solve these problems for every behaviour in a safe and robust way.

6.2 Using CoreLibrary Queries

When you want your behaviour to be queryable, all you have to do is extend `QueryableBaseBehaviour` instead of `BaseBehaviour`. The *only* difference to extending `BaseBehaviour` directly is that `QueryableBaseBehaviour` automatically adds a component of type `Queryable`. This component interacts with the `Query` singleton, ensuring that it's game object is properly represented.

In order to have a `Resource` be queryable, all you have to do is this:

```
class Resource : QueryableBaseBehaviour { ... }
```

So later you can easily query all resources:

```
// this loops through every object in the scene
// which has a subtype of  Resource  attached to it
foreach (var resource in Query.All<Resource>()) { ... }
```

6.3 .All and .AllActive

Additionally to the aforementioned `Query.All<T>()`, the CoreLibrary provides `Query.AllActive<T>()`, which (big surprise) does only retrieve the objects in the scene with components that extend `T` **with active game objects**.

6.4 .AllWith and .AllActiveWith

You might to query only the `Resource`s which also have an `Interactable` attached to it, assuming that `Interactable : QueryableBaseBehaviour` as well. For this use case, there's `Query.AllWith<T>(other, types)` and the corresponding `Query.AllActiveWith`.

```
// get all  Resource s which also have an  Interactable  component
var resources = Query.AllWith<Resource>(typeof(Interactable));
```

`.AllWith` takes at least one additional type bound, but can take as many as you want. The queried objects need to have at least one component of each of the specified types in order to be found.

7 Generic Pool

7.1 Introduction

Imagine you want to implement a gun that shoots bullets. These bullets are physical objects, so that they can ricochet and have a realistic hitbox detection. A naive implementation may look like this:

```
public class Gun : MonoBehaviour
{
    public Rigidbody BulletPrefab;

    public float Impulse;

    public void Shoot()
    {
        var newBullet = Instantiate(BulletPrefab);
        newBullet.transform.position = transform.position;
        newBullet.transform.forward = transform.forward;
        newBullet.AddForce(
            transform.forward.normalized * Impulse, ForceMode.Impulse);
    }
}

public class Bullet : MonoBehaviour
{
    // some code that calls Despawn()

    public void Despawn()
    {
        Object.Destroy(this);
    }
}
```

The problem with this implementation is that you need to spawn a new Bullet each time `Shoot()` is called a new `GameObject` is created by the engine. This can lead to FPS drops when firing many shots in rapid succession.

A common solution to this problem is spawning a number of bullets at the beginning of the scene. Then, every time `Shoot()` is called, a bullet is taken out of the **pool**. Bullets in the pool are **reusable**, meaning that once they stop moving or despawn they can be used again. No additional costly creations or destructions involved.

7.2 General Usage

With the CoreLibrary, our gun would look like this:

```
[RequireComponent(typeof(GenericPool))]
public class Gun : BaseBehaviour
{
    public Rigidbody BulletPrefab;

    public float Impulse;

    GenericPool _pool;

    private void Start()
    {
        AssignComponent(out _pool);
        _pool.Template = BulletPrefab;
        _pool.Capacity = 100;
        _pool.GrowRate = 0;
        // save settings and initialize
        _pool.Init();
    }

    public void Shoot()
    {
        var newBullet = _pool.RequestItem(transform.position);
        newBullet.transform.forward = transform.forward;
        newBullet.AddForce(
            transform.forward.normalized * Impulse, ForceMode.Impulse);
    }
}
```

```
Oh no, the code got larger! I thought that never happens!
```

– Yeah, you are right. This is because I did not write a custom, use-case specific pool for comparison.

In the example above, instead of configuring the pool in the Unity inspector we added a `RequireComponent(typeof(GenericPool))` to our `Gun` class. This automatically adds a `GenericPool` component, which we configured in the `Start()` method. Whenever we configure a pool in a script, we need to call the `pool.Init()` method. When you don't, then the pool initializes itself at the first call to `RequestItem()`, which might cause a small freeze.

When the pool does not find any item to reuse (which never happens in our example, why later), it's behaviour depends on the value of `GrowRate`. You see, in order to prevent glitches when there suddenly are no bullet lefts, the pool **grows** similar to the way a `List<T>` grows it's underlying array. When an item is requested but there are no free items to be found, the pool grows by a factor of `GrowRate`. Per default, `GrowRate` is set to `0.3`. There is a rough estimation. *If you know better, set it yourself.*

In the above case, since we never run out of bullets, we set `GrowRate = 0`. Now when we run out of bullets due to a bug, an `Error` is thrown to signal that something went wrong.

7.3 Usage from the Inspector

You do not have to write `[RequireComponent(typeof(GenericPool))]`. Maybe you want all your guns in the scene to share a single pool of bullets, just to be safe. For this use case, you should create an additional class for each type of **singleton pool** you need in the scene:

```
[RequireComponent(typeof(GenericPool))]
class BulletPool : Singleton<BulletPool>
{
    private GenericPool _pool;

    private void Start() => AssignComponent(out _pool);

    public static GameObject RequestBullet(Vector3 position)
    {
        return Instance._pool.RequestItem(position);
    }
}
```

The `Singleton<BulletPool>` ensures that there is exactly one component of this type in the whole scene. Then you place this pool on any game object in the scene, and request a new bullet like this *in any script*:

```
BulletPool.RequestBullet(transform.position);
```

Obviously, the pool isn't configured in the script. Instead, you configure the pool through the Unity inspector. Drag the bullet prefab into the `Template Object` field and set `Capacity` and `GrowRate` as required.

When configuring from the inspector, it is important to tick `Init On Scene Start`, so that initialization is done at scene start and not once the first item is requested, to prevent potential freezes.

7.4 Reusables

```
public class Bullet : Reusable
{
    // we want the bullets to lie around in the world
    // as physical objects until they need to be reused

    public override void ResetForReuse()
    {
        // this is called *before* reusing
        // you never have to call this yourself
    }

    public override void AfterReuse()
    {
        // this is called *after* a successful reuse
        // you never have to call this yourself
    }

    public override void ReuseRequested()
    {
        // this is called when there are no items left
        // when you call FreeForReuse() in this method,
        // the object is immediately reused.

        // you never have to call this yourself

        gameObject.SetActive(false);
        FreeForReuse();
    }
}
```

The `Bullet` class now extends `Reusable` and has to override three methods.

- `ResetForReuse` is called by the pool every time it is about to reuse the item. This is analog to the `Reset` method in Unity, but for reusable items.
- `AfterReuse` is called by the pool just after it has successfully reused the item. It is analog to `Start`, in that it allows you to call initialization code after an object is "spawned".
- `ReuseRequested` is called by the pool every time no item is available anymore. The pool loops through all items it manages and requests a reuse from each until an item calls `FreeForReuse()` in this method.

You do not have to add code to any of the three methods. However, overriding them is still mandatory in order to force you to think about your `Reusable`'s behaviour and prevent potential bugs.

7.5 Optimizations

The goal of the CoreLibrary is to create generic, robust and well-document extensions to the Unity engine that can be used in any project. For this reason, the `GenericPool` component has some notable behaviours which make using it safer for everyone.

7.5.1 Lazy Initialization

During initialization, the pool allocates it's buffer and fills it with clones of the specified `Template`. *Changes to `Template` and `Capacity` do not matter once a pool is initialized*. Because of this reason, initialization is performed lazily per default: Either when `.Init()` is explicitly called or once the very first item is requested. This enables you to configure a pool in another script (as seen in the above example). It also gives you the possibility to improve scene loading times by moving initialization to a later point in time.

This decision comes at a cost: When the user configures a pool in the inspector and forgets to tick `Init on Scene Start`, a freeze might occur once the first item is requested later in the scene.

7.5.2 Slow Growth

When a pool runs out of available items, it's capacity increases by a factor of `GrowRate`. If we created all items at the moment of growth we would cause a potential freeze. And we do not want freezes. For this reason, the pool only grows by *one item per frame*. When the pool is still growing and another item is requested, it is instantly instantiated on demand.

8 Design Patterns

This section contains a collection of design patters for using CoreLibrary code nicely. It will expand over time as more patterns and best practices are discovered.

8.1 Bind coroutine lifespan to boolean variable

```
// don't do this

public class SomeComponent : BaseBehaviour
{
    private Coroutine _routine = null;

    public void DoSomethingUseful()
    {
        _routine = SomeCoroutine().Start();
    }

    public void StopDoingSomethingUseful()
    {
        if (_routine != null) StopCoroutine(_routine);
    }

    public bool IsRunning => ???; // how?!
}
```

```
// do this instead

public class SomeComponent : BaseBehaviour
{
    private bool _isRunning = false;

    public void DoSomethingUseful()
    {
        _isRunning = true;
        SomeCoroutine()
            .YieldWhile(() => _isRunning)
            .Afterwards(() => _isRunning = false)
            .Start();
    }

```

```
    public void StopDoingSomethingUseful() => _isRunning = false;

    public void IsRunning => _isRunning;
}
```

This pattern is used for ending coroutines early in a nice way. In regular Unity code, if you want to stop a running coroutine early, you have to save a reference to the returned `Coroutine` object. While that produces working code, it doesn't give you any way to check whether the coroutine is actually running right now. Now, if you want to implement `IsRunning`, you'd usually have to keep track of an additional boolean. So why not make that boolean the only thing managing the state of the coroutine?

8.1.1 Don'ts

You might consider encapsulating the coroutine entirely behind a boolean property, but that is a terrible idea. Not only does writing `IsRunning = true` shouldn't have side effects like starting a coroutine, but you'll also quickly run into questions like *"what happens if it is already running?"*. Trust me, I just did while trying to write example code. It's a mess. Keep it simple.

8.2 Use `foreach` and `.ForEach`

Everything related to sequences of somethings yields an `IEnumerable<T>`. This means that the values are calculated *lazily* or *on demand*. Calling `.ToList()` forces all these values to compute, whether you need them or not. For example, when using `.FirstOrDefault()` only the very first value is computed.

This is why *calls to `.ToList()` should be avoided* whenever possible.

8.2.1 Don't do this

```
List<Cat> allMyCats = allMyYellowCats.AndAlso(allMyBrownCats).ToList();
for (var i = 0; i < allMyCats.Count(); ++i)
{
    Debug.Log($"Cat number {i} is called {allMyCats[i].name}!");
}
```

This example is chosen explicitly so that we need the index `i`, which *could* mandate the use of a counting **`for`**. However, using a counting **`for`** here forces us to build an entirely new `List` only to use the indexing operator `[i]` on it.

Counting `for` loops were a great thing back in the 80's when you only worked with `array`s, or linearly aligned blocks of memory. But times have changed, and some data structures (such as `HashSet`) cannot be represented as a linear block of memory, making indexing a *very inflexible* way of accessing values in a collection of elements.

8.2.2 Instead do this

```
IEnumerable<Cat> allMyCats = allMyYellowCats.AndAlso(allMyBrownCats);
var catIndex = 0;
foreach (var cat in allMyCats)
{
    Debug.Log($"Cat number {catIndex++} is called {cat.name}!");
}
```

"But we have an index so shouldn't we use counting `for`?" – **NO.**

The cats may not even have a strict order depending on where you got the yellow and brown cats. We never explicitly sort them. What's more important: `.AndAlso` yields an `IEnumerable<Cat>`, which means that it is implemented using `yield return`. So we do not need any additional memory for `allMyCats`, as opposed to when we explicitly create a new list. As a consequence, index access takes `O(n)` runtime, as the values are obviously not linearly aligned in memory. Calling `.ToList` forces this linear alignment, but... why should we do it?

As an added bonus: `foreach` is much less likely to cause a bug than `for`. I once spent half a day chasing a simple bug in a `for`: I wrote `for (int i = 0; i < tabs; tabs++)`. Oops. In this case however, I wasn't iterating through a collection. Still, counting `for`s are dangerous and should *not ever be used for iterating through a sequence of elements* (unless you are writing code in C or Golang).

`.ForEach` is the CoreLibrary's way for having one less variable to find a name for. The above code could be rewritten as:

```
var catIndex = 0;
allMyYellowCats.AndAlso(allMyBrownCats).ForEach(cat =>
{
    Debug.Log($"Cat number {catIndex++} is called {cat.name}!");
});
```

In this case, using `foreach` is the more readable way. However, sometimes it is hard to give something a useful name, especially after a long chain of LINQ calls. Or you just want to keep it very short, e.g.

```

void printCat(Cat cat, int index) =>
```

```
    Debug.Log($"Cat number {index++} is called {cat.name}!");

/* ... */

var catIndex = 0;
allMyYellowCats.AndAlso(allMyBrownCats)
    .ForEach(c => printCat(c, catIndex++))
```

Which one you use is a matter of taste and code style. Generally, I would prefer using `foreach` unless I want to use a direct method reference or single expression. Just don't use **`for`**.