Blogs

Adding to Unity's Built-In Classes Using Extension Methods

by Josh Sutphin on 10/07/13 03:27:00 pm Expert Blogger Featured Post

The following blog post, unless otherwise noted, was written by a member of Gamasutra's community.
The thoughts and opinions expressed are those of the writer and not Gamasutra or its parent company.

(This article was originally published at third-helix.com.)

Have you ever found yourself wishing a built-in Unity class had some functionality that isn't there? C# extension methods are the answer!

In this article, I'll teach you how to use extension methods to add functionality to existing classes, no matter if they're built-in Unity types, types defined in a third-party plugin, or even types defined in an Asset Store package which you *could* edit but you're (rightly) worried about later package updates stomping your "patch".

Seemingly obvious API omissions can be frustrating, but extension methods let you "fix" just about

Latest Blogs

Press Releases

any API to your liking.

When Extension Methods Are Useful

Imagine you need a way to set the layer of a GameObject and all its children, but there's no GameObject.SetLayerRecursively() available. You could embed a loop in your code:

```
// Do some work.

// Set the layer of this GameObject and all its children.
gameObject.layer = someLayer;
foreach(Transform t in transform)
    t.gameObject.layer = someLayer;

// Do some more work.
```

This will work fine, although it's not the cleanest thing in the world, and you'll need to copy these lines of code around to every place where you want to do this operation. It would be better to encapsulate that code in a function, and make that function available to everyone via a "helper" class:

```
public class GameObjectHelper
{
    public static void SetLayerRecursively(GameObject gameObject, int layer)
    {
        gameObject.layer = someLayer;
        foreach(Transform t in transform)
            t.gameObject.layer = someLayer;
    }
}
```

```
public class Test : MonoBehaviour
{
    void Start()
    {
        GameObjectHelper.SetLayerRecursively(gameObject, someLayer);
    }
}
```

This works fine too, but it can be easy to forget to use the helper class to invoke a function that feels like it belongs in GameObject itself. Wouldn't it be nicer if you could just do this?

- In response to his grandfather's death,...
- "Chief Puzzle Officer" Now Available...
- RETRO ASSAULT: V'Lorn Moon Challenge App - Arcade...
- 5 Million Download Celebration!
- RPG Fairy Elements for iPhone, iPad & iPod Touch...

View All RSS


```
gameObject.SetLayerRecursively(someLayer);
```

As it turns out, C# lets you make this happen!

Declaring An Extension Method

Extension methods are declared just like regular functions, except that a) they must be static, b) they must be declared inside a non-generic, non-nested, static class, and c) they include a special syntax for their first argument, which is a reference to the calling object:

```
public static class GameObjectExtensions
{
   public static void SetLayerRecursively(this GameObject gameObject, int layer)
   {
      // Implementation goes here
   }
}
```

Though declared as static, this is invoked as if it were an instance method:

```
myGameObject.SetLayerRecursively(someLayer);
```

Note that when we call the method we actually omit the first argument and skip straight to the second. Take another look at the method declaration above. See how the first argument is declared using the "this" keyword? That's what tells the compiler to *infer* that argument as the calling object; in this case, myGameObject.

That's actually all there is to it. Extension methods are easy!

For what it's worth, I like to organize my extension methods into classes named ClassNameExtensions. So I have GameObjectExtensions, TransformExtensions, ColorExtensions, and so on. There's nothing that says you have to do this; I just like the organization. You could pack them all together into a single Extensions class if you prefer, or any other kind of organization you want. Just remember that extensions methods must be declared inside a non-generic, non-nested, static class; the compiler will complain otherwise.

Limitations of Extension Methods

Extension methods can *add to* an existing class, but they cannot *override* existing methods. For example, if you declared this:

```
public static class GameObjectExtensions
{
   public static Component GetComponent(this GameObject gameObject, Type type)
   {
      Debug.LogError("LOL, you got trolled");
   }
}
```

…the compiler would basically ignore you. The rule for multiple method declarations using the same signature is: *instance methods always take precedence over extension methods.*

To extend a type, you'll need make sure it's in scope with the "using" directive. If you're extending any built-in Unity type, you're covered by virtue of the "using UnityEngine" that's a standard entry in most Unity scripts. If you're extending an editor type, you'll need to add "using UnityEditor", just like you would if you were calling that type. If you're extending a type from a third-party plugin, you may need to import a namespace; check the plugin's documentation (or source code, if you have it) for details.

I've already mentioned that extension methods must be declared inside a non-generic, non-nested, static class, which means you can't just drop them in willy-nilly wherever you want. In practice, this "limitation" turns out to be a useful organizing device. You can certainly argue that this is virtually identical to the "helper class" example at the top of this article, and in terms of implementation effort that's probably true; the difference is that with extension methods you get a cleaner, more normalized calling syntax. This really comes down to personal preference.

Some Useful Extension Methods

For the rest of this article, I'll share some useful extension methods I've written over the past few months. Feel free to incorporate these into your own codebase and modify them at-will. (But don't just copy-paste them into a text file and try to sell it on the Asset Store; that would make you a horrible person.)

Set Layer Recursively

I use the excellent 2D Toolkit for UI, and I use a two-camera setup: one camera for the scene, one camera for the UI. In order to make the UI camera render only UI objects, all the UI objects need to be on a UI layer. When I create UI objects from script I need to set that layer, and often I'm creating objects with children (like a button with a child sprite and a child text label).

It'd be nice to have a way to set the layer for this entire hierarchy with a single function call (you'll

recognize this as the example at the top of this article). Here's what the call looks like:

```
myButton.gameObject.SetLayerRecursively(LayerMask.NameToLayer("UI"));
```

And here's that extension method:

```
// Set the layer of this GameObject and all of its children.
public static void SetLayerRecursively(this GameObject gameObject, int layer)
{
    gameObject.layer = layer;
    foreach(Transform t in gameObject.transform)
        t.gameObject.SetLayerRecursively(layer);
}
```

Set Visual/Collision Recursively

While we're doing things recursively, wouldn't it be nice if we could enable/disable just the renderers, or just the colliders, for an entire hierarchy? This can be useful for UI, but it can also be useful in the scene. Imagine a complex hierarchy that defines a really fancy animated force field, and a game mechanic whereby you can switch your avatar's polarity, allowing passage through force fields of a particular color. When you switch polarity, you'd loop through the matching force fields and disable their colliders, like this:

```
foreach(ForceField forceField in forceFields)
    forceField.gameObject.SetCollisionRecursively(false);
```

Here's that extension method:

```
public static void SetCollisionRecursively(this GameObject gameObject, bool tf)
{
    Collider[] colliders = gameObject.GetComponentsInChildren();
    foreach(Collider collider in colliders)
        collider.enabled = tf;
}
```

And the same principle applies for renderers:

```
public static void SetVisualRecursively(this GameObject gameObject, bool tf)
{
    Renderer[] renderers = gameObject.GetComponentsInChildren();
```

```
    foreach(Renderer renderer in renderers)
        renderer.enabled = tf;
}
```

Filter Child Components By Tag

It's easy to search for child components using GameObject.GetComponentsInChildren(), but what if you have a have hierarchy in which you have lots of instances of a type, and you only want those instances which have a particular tag? In my case, I had a compound object with numerous renderers, and I needed to drive a material color on a subset of those renderers tagged as "Shield".

This proved convenient:

```
m_renderers = gameObject.GetComponentsInChildrenWithTag("Shield");
```

Here's that extension method:

```
public static T[] GetComponentsInChildrenWithTag(this GameObject gameObject, string tag)
    where T: Component
{
    List results = new List();

    if(gameObject.CompareTag(tag))
        results.Add(gameObject.GetComponent());

    foreach(Transform t in gameObject.transform)
        results.AddRange(t.gameObject.GetComponentsInChildrenWithTag(tag));

    return results.ToArray();
}
```

I noted earlier that extension methods must be declared inside non-generic classes. That doesn't mean the extension method *itself* can't be generic, however! When we call this method we replace T with the type we're interested in — in the preceding call example, it was Renderer — and that type is used for each occurrence of T in the method implementation. The "where" keyword specifies that T must be of type Component or a type derived from Component (the compiler will throw an error otherwise).

See this MSDN article for more information about generics.

Get Components In Parents

It's all well and good to get components in children, but sometimes you need to search *up*. A common case I run into is when figuring out what to do with a collision result. I have a Player type, and its hierarchy is made up of several GameObjects which represent visuals, colliders, equipped items, and so on. Typically when I get a collision result on a player, the collider is bound to a child GameObject, so I can't just do GetComponent() and then player.TakeDamage() or whatever, because there's no Player component on the GameObject I actually hit. In this case I need to search *up* the hierarchy and find the Player to which this collider is parented; also the Player may not necessarily be this collider's immediate parent.

So now I do this:

```
Player player = collider.gameObject.GetComponentInParents();
```

Here's that extension method:

```
public static T GetComponentInParents(this GameObject gameObject)
    where T : Component
{
    for(Transform t = gameObject.transform; t != null; t = t.parent)
    {
        T result = t.GetComponent();
        if(result != null)
            return result;
    }

    return null;
}
```

Just like Unity has both GameObject.GetComponentInChildren (singular) and GameObject.GetComponentsInChildren (plural), I also created a version that gets *all* components in parents:

```
public static T[] GetComponentsInParents(this GameObject gameObject)
    where T: Component
{
    List results = new List();
    for(Transform t = gameObject.transform; t != null; t = t.parent)
    {
```

```
        T result = t.GetComponent();
        if(result != null)
            results.Add(result);
    }

    return results.ToArray();
}
```

Note: it would be trivial to create a GetComponentsInParentsWithTag, but I haven't run into a need for it yet. If you'd like to exercise your newfound knowledge of extension methods, this might be a good exercise.

Get An Object's Collision Mask

Here's one whose omission is particularly perplexing. You can get the layer a GameObject is on, which is useful for both rendering and collision purposes, but there's no easy way to figure out *the set of layers that GameObject can collide against*.

I ran into this when implementing weapons. Projectile-based weapons are simple: they get a Rigidbody and a Collider of some sort, and the collision system handles everything for me. But a rail gun-like weapon is a different story: there's no Rigidbody, just a script-invoked raytest. You can pass a collision mask — a bitfield — into a raytest, but what if you want the collision mask to be based on the weapon's layer? It'd be nice to set some weapons to "Team1" and others to "Team2", perhaps, and also to ensure your code doesn't break if you change the collision matrix in the project's Physics Settings.

Really, I wanted to just do this:

```
if(Physics.Raycast(startPosition, direction, out hitInfo, distance,
    weapon.gameObject.GetCollisionMask())
)
{
    // Handle a hit
}
```

That raycast will only hit objects which the calling weapon is allowed to collide with, based on its layer and the project's collision matrix.

Here's that extension method:

```
public static int GetCollisionMask(this GameObject gameObject, int layer = -1)
{
    if(layer == -1)
        layer = gameObject.layer;

    int mask = 0;
    for(int i = 0; i < 32; i++)
        mask |= (Physics.GetIgnoreLayerCollision(layer, i) ? 0 : 1) << i;

    return mask;
}
```

Note the optional "layer" argument. If omitted, it uses the layer of the calling GameObject, which is the most common/intuitive case (for me, at least). But you can specify a layer and it'll hand you the collision mask for that layer instead.

Easily Change A Color's Alpha

I often find myself wanting to modulate the alpha value of a color without changing the color itself, for blinking/pulsing effects. Because Color is a struct, and structs in C# are immutable, you can't simply assign color.a; you'll get a compiler error. Fortunately, extension methods can extend structs as well as classes:

```
public static Color WithAlpha(this Color color, float alpha)
{
    return new Color(color.r, color.g, color.b, alpha);
}
```

This method makes modulating a color's alpha clean and simple:

```
GUI.color = desiredColor.WithAlpha(currentAlpha);
```

Performance Considerations

There really aren't any. Extension methods are a compile-time device; once you reach runtime, they look and act just like any other method call. So it doesn't matter whether you write an extension method or an explicit helper class; as long as the guts of the method are identical, both implementations should perform identically. You could theoretically squeeze a vanishingly tiny bit of performance out of simply inlining all the code, avoiding the overhead of any function call at all, but in this day and age that's completely pointless to worry about unless you're calling the function millions of times.

Extension methods are mainly useful as a way of cleaning up and normalizing your code. They have little or no effect on the behavior of that code.

Conclusion

I hope this article has been clear and useful. Extension methods are one of my favorite features of C#, because I'm a bit obsessive about having clean, easy-to-read code and they can really help make that happen. Of course, there's more than one way to skin a cat, and I'm by no means suggesting that extension methods are the *right* or *only* way to do things. You can certainly argue that making an explicit helper class containing regular static functions — like GameObjectHelper.SetActiveRecursively() in the example at the top of this article — is just as good as an extension method-based implementation of the same; they're six of one, half-dozen of another. I prefer the extension method approach because it feels like a natural and intuitive API extension, rather than a "bolt-on", but that's strictly a matter of personal preference.

Like every coding practice, extension methods are just one tool in the toolbox. But I encourage you to give them a shot!

(Josh Sutphin is an indie game developer, former lead designer of Starhawk (PS3), and creator of the Ludum Dare-winning RTS/tower-defense hybrid Fail-Deadly. He blogs at third-helix.com and tweets nonsense at @invicticide.)

[View All Jobs]

Comments

Bryson Whiteman 7 Oct 2013 at 7:40 pm PST

Nice post! Helpful stuff.

Login to Reply or Like

Wendelin Reich 8 Oct 2013 at 1:26 am PST

Yes, this is a great write-up, thanks!

Login to Reply or Like

Charles Clark 8 Oct 2013 at 6:33 pm PST

That's some useful tips that I will definitely leverage soon, thanks man!

Login to Reply or Like

David Leon 9 Oct 2013 at 2:01 am PST

Great post! :)

Just a quick note.. You may find errors in 'gameObject.GetComponentsInChildren();' if you don't specify the Class type, like this: 'gameObject.GetComponentsInChildren();' .

Login to Reply or Like

David Leon 9 Oct 2013 at 2:30 am PST

I also had to change other methods. The most important:

```
public static T[] GetComponentsInChildrenWithTag(this GameObject gameObject, string tag)
where T : Component
{
List results = new List();

if(gameObject.CompareTag(tag))
results.Add(gameObject.GetComponent());

foreach(Transform t in gameObject.transform)
results.AddRange(t.gameObject.GetComponentsInChildrenWithTag(tag));

return results.ToArray();
}
```

Login to Reply or Like

Josh Sutphin 9 Oct 2013 at 12:15 pm PST

I don't see what you've changed about the method itself, but I did notice (thanks to your comments) that my use case examples for GetComponentInParents and GetComponentsInChildrenWithTag had omitted the portion of the call. They should've been called like so:

```
m_renderers = gameObject.GetComponentsInChildrenWithTag("Shield");
```

and:

Player player = collider.gameObject.GetComponentInParents();

I've updated the article accordingly.

Login to Reply or Like

Luis Guimaraes 9 Oct 2013 at 6:06 am PST

I stopped using them just because they won't show in the docs and it becomes confusing.

Login to Reply or Like

Jason Bentley 9 Oct 2013 at 7:24 am PST

Excellent information, thank you!

Do you have any suggestions for making coroutines work as extensions?

I'm attempting to do this:
public static void AnimateLocalScale(this Transform transform, AnimationCurve Curve, float Length)
{
// StartTheCoroutine
}

which should start this:
static IEnumerator AnimateLocalScaleOverTime(Transform ObjTransform, AnimationCurve Curve, float Length)

Not having any luck.

Login to Reply or Like

Jason Bentley 9 Oct 2013 at 7:56 am PST

I did make it work, but still interested to see how you would have solved

the issue.

Login to Reply or Like

Josh Sutphin 9 Oct 2013 at 12:46 pm PST

Good question! I just went through the exercise of setting this up myself, to see how it would work, and here's what I came up with:

First, you need a MonoBehaviour to start a coroutine from, and your extension class isn't a MonoBehaviour (nor can it be). So I created a TransformAnimator class that inherits from MonoBehaviour, and I implemented the coroutine function AnimateLocalScaleOverTime in that class.

My actual implementation is full of debug/test stuff, but the pseudocode gist of is (apologies for formatting):

....while(not doneAnimating)
........step animation
........yield return null;
....yield break; // Terminate the coroutine

Then back in the extension method, I need to get (or create, if necessary) an instance of TransformAnimator that can manage the coroutine instance:

....TransformAnimator ta = transform.GetComponent();
....if(!ta)
........ta = transform.gameObject.AddComponent();
....ta.StartCoroutine(ta.AnimateLocalScaleOverTime(params));

Note that I'm invoking both StartCoroutine *and* the coroutine function itself on a TransformAnimator instance.

My exercise does not remove the TransformAnimator component after the coroutine terminates, but it easily could do so.

BTW you had a neat idea, here. Using a coroutine for this purpose had never crossed my mind before, but this seems really useful. :)

Login to Reply or Like

Jason Bentley 10 Oct 2013 at 6:54 am PST

"Note that I'm invoking both StartCoroutine *and* the coroutine function itself on a TransformAnimator instance."

This is the part that I was running into and I came up with, basically, exactly the same solution. :)

This Unite video got me thinking about Coroutines a lot recently and, honestly, I've *just* gone through and replaced a lot of Update code with Coroutines.

http://www.youtube.com/watch?v=ciDD6WI-Evk

Login to Reply or Like

Josh Sutphin 12 Oct 2013 at 1:37 pm PST

Coroutines are pretty rad. I do tend to use them sparingly, though, because they can very quickly get tricky to debug and profile.

Login to Reply or Like 1

Paulo Mattos 9 Oct 2013 at 9:47 am PST

Good info, thanks!

C# extensions methods are also great for API discovery. For instance, if you work on a project with hundreds of classes, good luck finding the desired utility method on your own. With extension methods, the IDE will find it for you during auto-complete ;-)

A minor correction: structs are not immutable, they are actually value types. For example, this works fine:

Color c;
c.a = 0.5f;

...but this won't work:

obj.GetColor().a = 0.5f;

Login to Reply or Like 1

Josh Sutphin 9 Oct 2013 at 12:49 pm PST

I must admit, this is a distinction I've never fully understood. I knew structs are value types in C#. What I've never fully grokked is why your first example is legal, while the second isn't. I know that that's *true*, I just don't really get *why* it's true.

Login to Reply or Like 1

Edgar Harris 9 Oct 2013 at 3:28 pm PST

Reading my explanation it probably isn't clear. The reason why the second isn't legal has to do with structs being value types. When you return a struct value a copy of the original struct is returned. So in the following code the method Foo() returns a copy of a MyStruct that it created and that copy is assigned to myVar.
MyStruct myVar = Foo();
However if I just call Foo() and then try to set a property on the returned struct that is an error. The reason is a copy the struct is returned on the stack. The property of that copied struct is set, and then the struct goes off the stack because it was never assigned to a variable, so the following code is a no-op.
Foo().MyProp = someValue;
Since that code is completely nonsensical the C# compiler complains.

Login to Reply or Like 1

Edgar Harris 9 Oct 2013 at 2:09 pm PST

Yeah, but if you look at the standard library you'll notice that most structs are immutable, and I would recommend keeping structs immutable as much as possible. Otherwise you end up with really weird idiosyncrasies. Consider the following example:

```
class Foo{ Bar MyProperty{get;set;}}
...
Foo f = new Foo();
f.MyProperty.SomeOtherProp = newValue;
```

If Bar is a class this code runs as expected. If Bar is a struct that code will not compile, because a copy of your Bar instance is returned on the stack, the property is invoked on the copied value, and then that copy is cleared from the stack. It is not immediately obvious to consumers of your class that this is the case. If you Bar is a struct and is immutable you would end up with something like this:

```
Foo f = new Foo();
f.MyProperty = f.MyProperty.SetSomeOtherPropValue(newValue);
```

In the immutable case the proper use of the Bar struct is very clear.

Login to Reply or Like 1

Paulo Mattos 9 Oct 2013 at 7:26 pm PST

Just a quick note: Miguel de Icaza has a simple proposal for fixing this at the C# compiler-level:

http://tirania.org/blog/archive/2012/Apr-11.html

Login to Reply or Like

Josh Sutphin 9 Oct 2013 at 8:18 pm PST

Yes, this makes sense. Thanks for the clarification!

Login to Reply or Like

Roger Miller 9 Oct 2013 at 10:45 am PST

Excellent, thanks mate!

Login to Reply or Like

Kristian LH Jespersen 10 Oct 2013 at 2:20 am PST

Hi just wanted to share a version of the SetCollisionRecursively & SetVisualsRecursively that uses generics:

```
public static void SetComponentRecursively<T>(this GameObject gameObject, bool tf)
where T : Component
{
T[] comps = gameObject.GetComponentsInChildren<T>();
foreach (T comp in comps)
{

try
{
System.Reflection.PropertyInfo pi = (typeof(T)).GetProperty("enabled");
if (null != pi && pi.CanWrite)
{
pi.SetValue(comp, tf, null);
}
else
{
Console.WriteLine("Property does not exist, or cannot write");
}
}
catch (NullReferenceException e)
{
Console.WriteLine("The property does not exist in Component." + e.Message);
}
```

}
}

EDIT: Added that was lost due to html formatting in original post

Login to Reply or Like

Josh Sutphin 12 Oct 2013 at 1:40 pm PST

My favorite thing about sharing knowledge is watching others run with it, and come up with things I hadn't thought of. Nicely done. :D

Although I think you missed the type specifier in your GetComponentsInChildren call. Shouldn't it have been:

T[] comps = gameObject.GetComponentsInChildren();

Login to Reply or Like

Kristian LH Jespersen 14 Oct 2013 at 12:44 am PST

Thanks for letting me know, its strange though, It seems that whenever I copy/paste the method it just leaves out the type specifier...
My original post actually had the type specifier, its just not being shown.

The type specifier is also left out in your comment. I guess there is some Gamasutra comments formatting that removes it

Edit: For future reference, to be able to write "<T>" you need to use html markup & l t ; T & g t ; (remove spaces)

Login to Reply or Like

Eris Koleszar 10 Oct 2013 at 9:35 am PST

Thanks for the article, I will use this for a long time! I have a question, though.

Is there any way to pass the object calling the function as a reference? I don't want to have to pass the variable as a parameter to edit it. E.g.

```
public static void X(this Transform t, float eulerX, ref Transform thisTransform)
{
thisTransform.rotation = Quaternion.Euler(new Vector3(eulerX,
thisTransform.rotation.eulerAngles.y, thisTransform.rotation.eulerAngles.z));
}
```

Login to Reply or Like

Jason Bentley 10 Oct 2013 at 10:13 am PST

Even though the first parameter is ignored when you call the function, it is accessible inside the function.

One of his examples:

```
public static void SetLayerRecursively(this GameObject gameObject, int
layer)
{
gameObject.layer = layer;
...
}
```

Login to Reply or Like

Eris Koleszar 10 Oct 2013 at 10:39 am PST

I used a bad example. What I'm really trying to alter is a Color (struct).

Using these three examples:

```
public static void BadFoo(this Color c)
{
c.b = 1.0f;
}
public static void Foo(this Color c, ref Color col)
{
col.b = 1.0f;
}
public static Color Bar(this Color c)
```

```
{
c.b = 1.0f;
return c;
}

myColor.BadFoo(); // does nothing
myColor.Foo(ref myColor); // sets blue value to 1.0f
myColor = myColor.Bar(); // sets blue value to 1.0f
```

Are the last two my only options and why would I use one over the other?

Login to Reply or Like

Alexander Jhin 10 Oct 2013 at 11:52 am PST

Unless you are passing a ref, a struct is pass by value. So, the variable c in BadFoo is referring to a stack COPY of the the original struct. That's what c.b = 1.0f ends up modifying.

Your Foo example is "better" than the Bar example, because it's more obvious what it does and doesn't require the extra assignment, which you might forget to do.

However, if you are familiar with structs (especially immutable structs), the Bar example looks more "natural" because that's what structs usually do: They create new copies of themselves when operated on, which must then be assigned.

Login to Reply or Like 2

Eris Koleszar 11 Oct 2013 at 8:10 am PST

Thanks! That helps.

Login to Reply or Like

Josh Sutphin 12 Oct 2013 at 1:42 pm PST

Alexander has the correct answer.

FWIW, coming from a C++ background, the fact that C# structs are value types still messes with my head to this day. :P

Login to Reply or Like

Jason Bentley 10 Oct 2013 at 12:03 pm PST

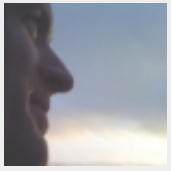
Paulo Mattos and Josh Sutphin talk about this just above here.

What I got from their discussion is that this is a problem when ever you create an Extension on a value type. The predefined value types and Structs (a custom value type) have the same issue: "c" in your first example is a copy of the value.

I like Alexander's suggestion.

Login to Reply or Like 1

Chris March 21 Jan 2014 at 9:55 am PST



Is it possible to add handler functions for the new Mecanim AnimationEvents via extensions? I'm guessing this is complicated by the need for another assembly to call it, but perhaps I got the namespace wrong. Mine is not recognized:

```
namespace UnityEngine
{
public static class GameObjectExtensions
{
public static void tag(this GameObject gameObject)
{

}
}
};
```

Login to Reply or Like

Richard Ellicott 13 Nov 2015 at 6:19 am PST

i do love this method

ANYONE know how to add to say the "Physics" library?

I want to add "CubeRadius" or something similar, when i just about change an existing feature but it's still very neutral to my project

Login to Reply or Like

Login to Comment