JAC

Don't Use == null On Unity Objects

Nov 20, 2015

Don't use `if (obj != null)` on a `GameObject`, `Texture2D`, `Sprite`, or any other `UnityEngine.Object` without understanding when `obj == null` doesn't mean `obj` is acctually null.

Instead, always use the implicit bool cast (i.e. `if (gameObject)`). That way if you happen to use it wrong it will fail to compile instead of give you unexpected values at runtime.

Update: I'm not first to write about this, see this Unity blog post too

Destroyed != null

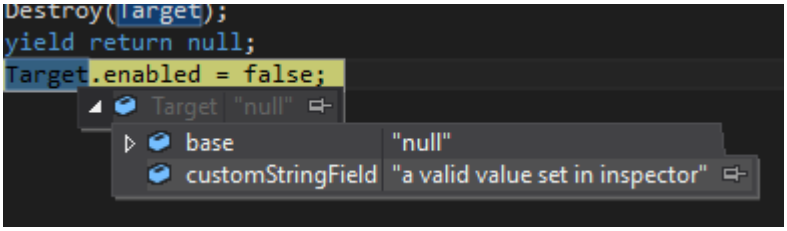
The problem is a reference to a destroyed object isn't a null reference, even though it pretends to be. Once you destroy a Unity object, even though it's no longer a valid **Unity Object**, it's still a valid **C# object**.

`UnityEngine.Object` overloads the equality operator so that `destroyedObject == null` returns true. If you're not aware of when that's an issue it can cause you problems.

Playing Dead

So, the weird thing is, that even after a custom `MonoBehavior` or `ScriptableObject` has been Destroy'd, any fields or methods you added to the C# class will still work and have valid values (!!!). Your object still works. Just so as long as you don't use any built in externed properties or methods, of course those won't work anymore.

Here's what a destroyed `MonoBehavior` looks like if you inspect it in the debugger.

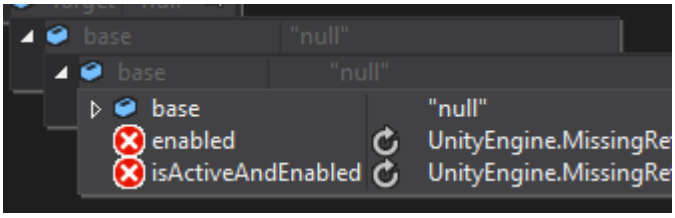


It even still has the field values I gave it! Also note that if you `ToString` a destroyed object, it returns "null". The normal string value of a null reference in C# is `""`, the empty string.

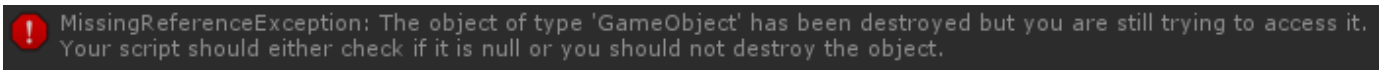
> *"Hello Object, are you dead?"*

> *"Yes."*

Of course accessing the actual `MonoBehavior` properties throw exceptions, as expected.



Also noticed that you don't get a `NullReferenceException` as you would if it was actually a null reference, instead you get a `MissingReferenceException`:



That's how they're able to provide this helpful but misleading hint: It's not a null reference, it's a reference to a gravestone. Ironically their suggestion to "check if it is null" is your first hint that it is in fact, not null.

Yet `deadObject == null` still returns true in this case, as they've taught you to expect.

That's because Unity overrides the `==` operator for `UnityEngine.Object` and all objects that extend it so that `== null` will be also be true if the relevant native engine object has been destroyed, in addition to if the reference is actually a null reference.

Why do they do this? Because Unity engine objects aren't pure C# objects.

What is a Unity Object?

Unity objects are really just thin C# wrappers for native C++ objects in the native side of the Unity engine that live outside of the managed, garbage collected C# runtime.

When you call `Destroy(obj)` on a Unity object the C++ engine destroys the native object and nulls out the C# object's internal pointer to it. However, only the garbage collector is allowed to kill managed C# objects, and only when all references to it go out of scope (to prevent use after free errors), so the husk of a C# handle object lives on until it gets garbage collected normally. Of course, any calls to Unity built-in methods or properties on those objects will then fail, because the C++ engine object they reference is gone.

So...

Using the overloaded `==` and thinking of destroyed objects as null usually works fine. Until you do something like pass a `MonoBehavior` or `GameObject` to a method like...

```
void Asset.IsNotNull(object obj, string message) {
    if (obj == null)
        Debug.LogErrorFormat("Assert Null: " + message);
}
```

As you might expect **now**, this will **not** assert for a destroyed `GameObject`, even though it's not a valid `GameObject` reference. Since C# is a static language where the operator implementation used is picked at compile time, the compiler uses the default equality operator implementation for `object` (that is, `System.Object`), not `UnityEngine.Object`'s more specialized "check the native pointer too" operator. At compile time, `object`'s basic implementation is the most specific implementation we can safely use for all possible `object` typed values in `obj`, and operators aren't runtime polymorphic in C#.

Also remember how my destroyed `MonoBehavior` still had a valid string reference? References to destroyed objects can cause memory leaks like this. We've had lingering references to dead Unity objects keeping textures, and other large assets from being unloaded from memory when they were otherwise unused, because there was a lingering reference to a destroyed custom `MonoBehavior` that still referenced it in one of our custom fields.

You never expect this behavior if you think of "destroyed reference" and "null reference" as the same thing.

Stepping Over Gravestones

When you regularly think, as Unity encourages, of a destroyed object as `null`, that kind of mistake is easy to make. Obviously I've made that mistake enough times to be bitter about it.

Just don't do it. Be aware that Destoyed isn't null. Think of `GameObject`s and other `UnityEngine.Object`s like containers, like `string`s, or `List`s. Remember that before you can do operations on those you have to do something like…

```
if (!string.IsNullOrEmpty(name)) \\...
// or...
if (list != null && list.Count > 0) \\...
```

Honestly, I wish the Unity interfaces didn't hide the fact that destroyed objects are different from null references from you. I wish the canonical way of expressing this was something more standard like:

```
if (gameObject != null && gameObject.IsValid) \\...
```

Which although verbose, makes the behavior clear, and seeing it makes the above gotchas more obvious. This is basically what happens internally in the implementation of these operators too. Unfortunately there's no public property indicating if an object is dead or alive, so we just have to keep aware.

In lieu of that, `UnityEngine.Object` does implement an implicit conversion to bool, which I always use over equality to null. The benifit of that (besides being shorter) is `if (obj)` won't

always use over equality to null. The benifit of that (besides being shorter) is `if (obj)` won't compile for regular `object` references, since most C# objects don't implement that implicit

conversion, stopping you from making the mistake with the asset, where you've cast to object somewhere.

So for the Asset, use something like:

```
void Asset.IsValid(UnityEngine.Object obj, string message) {
    if (!obj)
        Debug.LogErrorFormat("Assert Missing: " + message);
}
```

And remember to try to diligently null out your object references after you destroy them to prevent unexpected memory leaks, and if you do need to check a Unity object reference for validity before accessing built-in properties, use:

```
// not != null
if (customMonoBehavior) {
    customMonoBehavior.enabled = false;
    Destroy(customMonoBehavior);
    customMonoBehavior = null;
}
```

Stay safe.

JAC

Jovanni A. Cutigni