Welcome to the Unity Forums! Please take the time to read our Code of Conduct to familiarize yourself with the forum rules and how to pos constructively.

New Forum User Notice | Unite 2023 Registrations are now LIVE!

Search...

# When we need new instance of intrinsic YieldInstruction?

Watch Thread

**RamType0**

Joined: Sep 11, 2018
Posts: 67

When we need new instance of intrinsic YieldInstruction?(e.g. WaitForEndOfFrame,WaitForFixedUpdate, not including Coroutine or CustomYieldInstruction)

If we don't need new instance in any time(pooling one instance is always valid),
please replace newobj instruction for intrinsic YieldInstruction by ldsfld instruction for singleton instance in ILPostProcessor by default.

RamType0, **Nov 30, 2020** Report #1 Like Reply

---

**PraetorBlue**

Joined: Dec 13, 2012
Posts: 7,701

Pooling is always valid as far as I know. Just be careful with the ones that have fields e.g. WaitForSeconds.

PraetorBlue, **Nov 30, 2020** Report #2 Like Reply

RamType0 likes this.

---

**Bunny83**

Joined: Oct 18, 2010
Posts: 3,452

Pooling and caching are similar but generally different. A pool of objects is a collection of objects that can be used and when an instance is no longer needed it has to be returned to the pool. Caching is just about storing a single instance in a variable.

Note that the built-in YieldInstructions (WaitForSeconds, WaitForEndOfFrame, WaitForFixedUpdate) do not contain any "state" that is changed. So you can simply reuse them. However the newer "CustomYieldInstruction" and everything derived from it (like WaitForSecondsRealtime) can not be cached or pooled because they actually contain state that is changed. They actually represent a sub-coroutine as the CustomYieldInstruction class implements the IEnumerator interface.

As for automatically replacing those in a post processor, I don't think it's worth it. The only two direct cases would be WaitForEndOfFrame and WaitForFixedUpdate. WaitForSeconds can be cached, but only when used with a compile time constant. As soon as you use a field of the outer class the post processor can not really reason about the usage.

For the two that can be cached without any issues you could simply create a class like this:

Code (CSharp):

```
1. public static class WaitFor
2. {
3.     public static readonly EndOfFrame = new WaitForEndOfFrame();
4.     public static readonly FixedUpdate = new WaitForFixedUpdate();
5. }
```

So instead of

```
yield return new WaitForEndOfFrame();
```

you could simply use

```
yield return WaitFor.EndOfFrame;
```

Bunny83, **Nov 30, 2020** Report #3 Like Reply

PraetorBlue likes this.

---

**Bunny83**

Joined: Oct 18, 2010
Posts: 3,452

For anyone who's looking for a garbage free WaitForSeconds solution, I just did a few tests and came up with a hacky solution that seems to work

Spoiler: Explanation how I found the solution

First of all I tested if Unity's coroutine scheduler will use the stored delay value inside the WaitForSeconds instancne after it was passed through a yield return statement. It seems Unity only "extracts" the delay time when the instance is yielded. From that point on any changes to the private "m_Seconds" field inside the WaitForSeconds instance has no effect on the wait time.

First I simply used reflection to change the value inside the instance. So I used a single statically cached WaitForSeconds instance in several coroutines at the same time and changed the internal value of that instance through reflection. This worked just fine, however doesn't buy us anything because even we avoid creating a new WaitForSecond instance, setting a value type field through reflection requires the value we want to assign to be boxed. So we would still create garbage. Also reflection has a bit of overhead (even though it's not that much).

So my final solution is essentially an IL code "hack". I created an extremely simply class in C#, compiled it to an assembly, decompiled it with "ildasm" to il code and just inserted a

manual call to the WaitForSeconds constructor. I recompiled the assembly and the result is a 2kb assembly file that provides a simply way to wait for any amount of seconds with 0 garbage allocation.

Note that the constructor in .NET is just an instance method like any other method, but it's handled slightly different. When you create a new object instance with a particular constructor we actually use the "newobj" il instruction which expects a reference to the constructor you want to use. However when a constructor calls the base constructor, of course no new object should be created. This call to the base constructor is a normal instance method call. That's what I'm using to essentially overwrite the internal private field. I first tried to simply set the private field through IL code, however even on IL level it will cause a runtime exception because of the "private" visibility of the field. Since the WaitForSeconds constructor does nothing else than setting the private field (and calls the empty base constructor of YieldInstruction), it's essentially similar to a normal property setter.

So here are the results:

| Overview | Total | Self | Calls | GC Alloc | Time ms | Self ms |
|---|---|---|---|---|---|---|
| ▶ WaitForSecondsSingleton.Get() | 0.0% | 0.0% | 1 | 0 B | 0.00 | 0.00 |

Here's the compiled assembly and here's the IL code that this assembly is made of, just in case you don't trust me ^^. I stripped a lot of meta data noise from the assembly and it seems none of that is actually required. That cuts down the assembly size from 3k to 2k. There's still alot of padding which we may be able to remove, but Windows could be a bit picky when it comes to the page alignment.

I haven't tested it in any build yet, so feel free to run some tests

The usage is pretty straight forward. Place the assembly in your project and you should be able to use

Code (CSharp):

```
1.  yield return WaitForSecondsSingleton.Get(5f);
```

instead of

Code (CSharp):

```
1.  yield return new WaitForSeconds(5f);
```

Keep in mind that you **must not** cache the return value of the Get method as it returns the one single static instance every time. So other calls to Get will modify the internal wait time. Just use the static Get method whenever you want to wait for a certain amount of seconds without any garbage allocations.

NOTE: This is of course still a hack that relies on the fact that Unity's coroutine scheduler will read the supplied delay time only once at the moment it's yielded. This behaviour is internal to Unity and could of course change in the future. So you get no guarantee or warranty

Bunny83, **Nov 30, 2020** Report #4 Like Reply

eisenpony likes this.

---

**PraetorBlue**

Joined: Dec 13, 2012
Posts: 7,701

Bunny83 said: ↑

*For anyone who's looking for a garbage free WaitForSeconds solution, I just did a few tests and came up with a hacky solution that seems to work*

Spoiler: Explanation how I found the solution

*First of all I tested if Unity's coroutine scheduler will use the stored delay value inside the WaitForSeconds instancne after it was passed through a yield return statement. It seems Unity only "extracts" the delay time when the instance is yielded. From that point on any changes to the private "m_Seconds" field inside the WaitForSeconds instance has no effect on the wait time.*

*First I simply used reflection to change the value inside the instance. So I used a single statically cached WaitForSeconds instance in several coroutines at the same time and changed the internal value of that instance through reflection. This worked just fine, however doesn't buy us anything because even we avoid creating a new WaitForSecond instance, setting a value type field through reflection requires the value we want to assign to be boxed. So we would still create garbage. Also reflection has a bit of overhead (even though it's not that much).*

*So my final solution is essentially an IL code "hack". I created an extremely simply class in C#, compiled it to an assembly, decompiled it with "ildasm" to il code and just inserted a manual call to the WaitForSeconds constructor. I recompiled the assembly and the result is a 2kb assembly file that provides a simply way to wait for any amount of seconds with 0 garbage allocation.*

*Note that the constructor in .NET is just an instance method like any other method, but it's handled slightly different. When you create a new object instance with a particular constructor we actually use the "newobj" il instruction which expects a reference to the constructor you want to use. However when a constructor calls the base constructor, of course no new object should be created. This call to the base constructor is a normal instance method call. That's what I'm using to essentially overwrite the internal private field. I first tried to simply set the private field through IL code, however even on IL level it will cause a runtime exception because of the "private" visibility of the field. Since the WaitForSeconds constructor does nothing else than setting the private field (and calls the empty base constructor of YieldInstruction), it's essentially similar to a normal property setter.*

*So here are the results:*

| Overview | Total | Self | Calls | GC Alloc | Time ms | Self ms |
|---|---|---|---|---|---|---|
| ▶ WaitForSecondsSingleton.Get() | 0.0% | 0.0% | 1 | 0 B | 0.00 | 0.00 |

*Here's the compiled assembly and here's the IL code that this assembly is made of, just in case you don't trust me ^^. I stripped a lot of meta data noise from the assembly and it seems none of that is actually required. That cuts down the assembly size from 3k to 2k. There's still alot of padding which we may be able to remove, but Windows could be a bit picky when it comes to the page alignment.*

*I haven't tested it in any build yet, so feel free to run some tests*

*The usage is pretty straight forward. Place the assembly in your project and you should be able to use*

*Code (CSharp):*

```
1.  yield return WaitForSecondsSingleton.Get(5f);
```

*instead of*

Code (CSharp):

```
1.  yield return new WaitForSeconds(5f);
```

*Keep in mind that you **must not** cache the return value of the Get method as it returns the one single static instance every time. So other calls to Get will modify the internal wait time. Just use the static Get method whenever you want to wait for a certain amount of seconds without any garbage allocations.*

*NOTE: This is of course still a hack that relies on the fact that Unity's coroutine scheduler will read the supplied delay time only once at the moment it's yielded. This behaviour is internal to Unity and could of course change in the future. So you get no guarantee or warranty* 😁

~~Is there any reason not to just do this?~~

Ok I started writing this and figured out why you can't (without reflection):

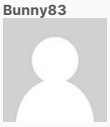
Code (CSharp):

```
1.  public static class WaitForSecondsSingleton {
2.    private static readonly WaitForSeconds _instance = new WaitForSeconds(0f);
3.    public static WaitForSeconds Get(float time = 0f) {
4.      _instance... // oh wait this field is not accessible or mutable without reflection
5.      return _instance;
6.    }
7.  }
```

PraetorBlue, **Nov 30, 2020** Report #5 Like Reply

Bunny83 likes this.

**Bunny83**



Joined: Oct 18, 2010
Posts: 3,452

PraetorBlue said: ↑

*oh wait this field is not accessible or mutable without reflection*

Yep, that's the main issue. The best solution would be when Unity provides us with a similar native solution so we could actually rely on this behaviour. At least this solution is probably the fastest zero GC solution you can get at the moment.

Bunny83, **Nov 30, 2020** Report #6 Like Reply

**AlMartson**

Write your reply...

Post Reply | Upload a File | More Options...




© 2023 Unity Technologies

Legal Privacy Policy Cookies Do Not Sell or Share My Personal Information

Your Privacy Choices (Cookie Settings)

"Unity", Unity logos, and other Unity trademarks are trademarks or registered trademarks of Unity Technologies or its affiliates in the U.S. and elsewhere (more info here). Other names or brands are trademarks of their respective owners.