**Injector**

[What is Injector? 2](#_Toc483123775)

[What can it do? 2](#_Toc483123776)

[How does it work? 3](#_Toc483123777)

[How can I use it? 3](#_Toc483123778)

[How to setup class for Injector? 4](#_Toc483123779)

[What attributes can I use? 4](#_Toc483123780)

[Nested object fields 7](#_Toc483123781)

[What if I do something wrong? 8](#_Toc483123782)

[What if I want to check for errors manually? 8](#_Toc483123783)

[How can I create my own attributes? 8](#_Toc483123784)

****

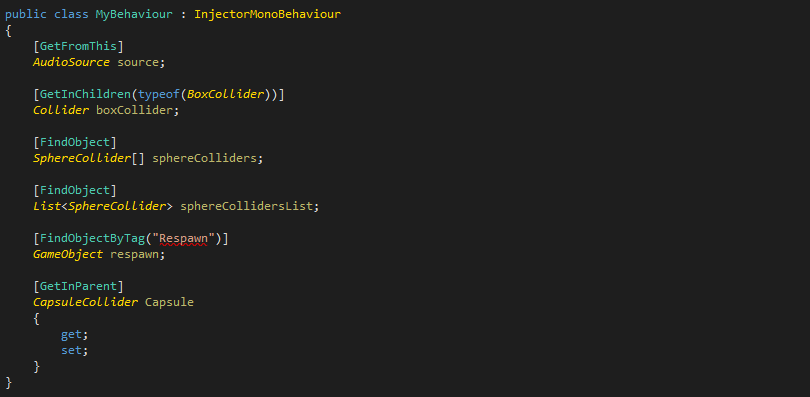
# What is Injector?

Injector is lightweight framework created to simplify development by removing monotonous tasks of initializing fields and resolving trivial dependencies from development process.

# What can it do?

How many times while development you were writing something like:

It’s not clean and messy solution of the problem. Injector allows you to clean up your code and forget about this initialization nightmare!

Look at the same code but with Injector:

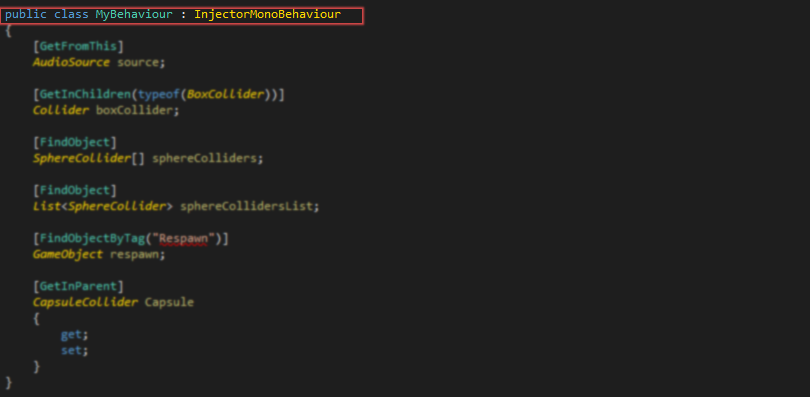
As you can see it is much cleaner and more descriptive on its own. All it is made behind the scene with high performance avoiding any garbage everywhere it is possible!

# How does it work?

It inspects your type and caches result of inspection, after that it fills all required fields and properties for you.

# How can I use it?

There are two ways to use it:

Derive from one of provided classes (InjectorMonoBehaviour, InjectorNetworkBehaviour):

Please note that in that case the Awake method of the base class is made virtual, so if you want to use it, you should override it calling base.Awake() there.

The second way is to ask Resolver to resolve all dependencies: 

Note that it can be called at the any moment, not only in Awake function.

# How to setup class for Injector?

The only thing you need to do – is set attributes on fields and properties you want to be filled by Injector.

# What attributes can I use?

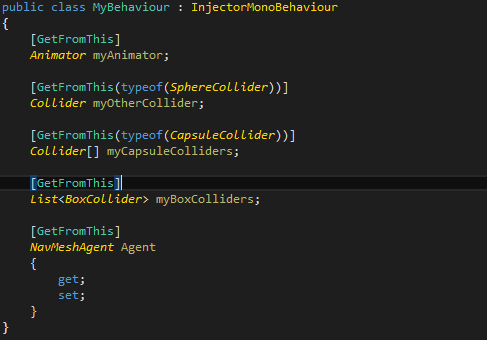
For now there are 8 supported attributes. Also it is open for extension so you can make your own attributes. The way you can create your own attributes will be described further.

Lets look at all awailable attributes:

GetFromThisAttribute(*Type* exactType)

Attribute used to get component/components from this game object.

*Type* exactType- exact component subclass you want to get. If not provided will be deducted from field itself.

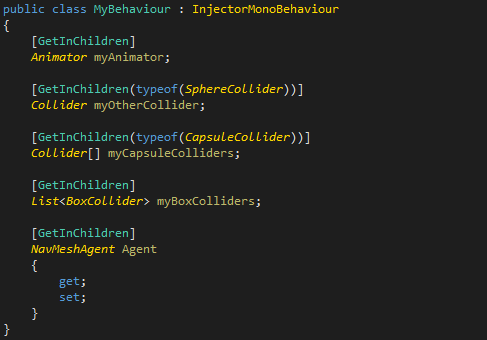
****

GetInChildrenAttribute(bool includeInactive, *Type* exactType)

Attribute used to get component/components from this game object and its child objects.

bool includeInactive *–* should inactive objects be included?

*Type* exactType - exact component subclass you want to get. If not provided will be deducted from field itself.

****

FindObjectByTagAttribute(string tag)

Attribute used to find GameObject by tag.

string tag – desired tag of game object.

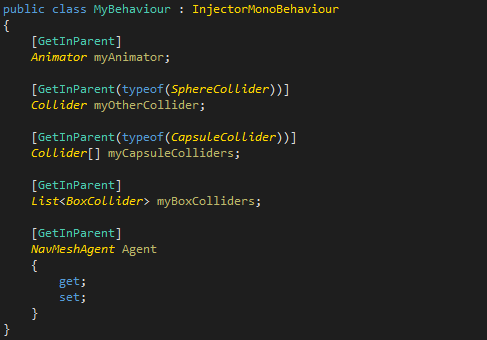
****

GetInParentAttribute(bool includeInactive, *Type* exactType)

Attribute used to get component/components from this game object and its parent objects.

bool includeInactive *–* should inactive objects be included?

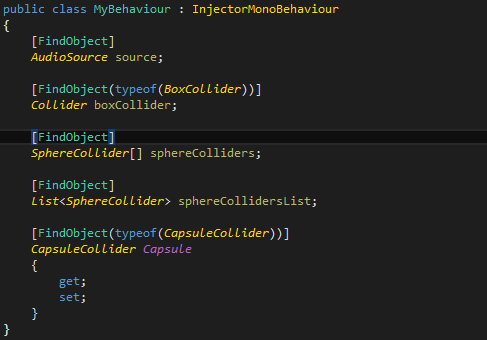
*Type* exactType - exact component subclass you want to get. If not provided will be deducted from field itself.

****

FindObjectAttribute(*Type* exactType)

Attribute used to find objects in the scene.

*Type* exactType - exact component subclass you want to get. If not provided will be deducted from field itself.

****

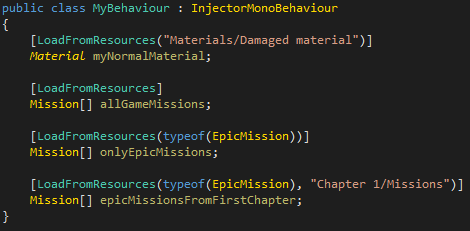
LoadFromResouresAttribute(*Type* exactType, *string* loadPath)

LoadFromResouresAttribute(*string* loadPath)

Attribute used to load things from resources folder.

*string* loadPath – folder you want to load from, or exact asset path (in case of single object loading)

*Type* exactType - exact component subclass you want to load. If not provided will be deducted from field itself.

****

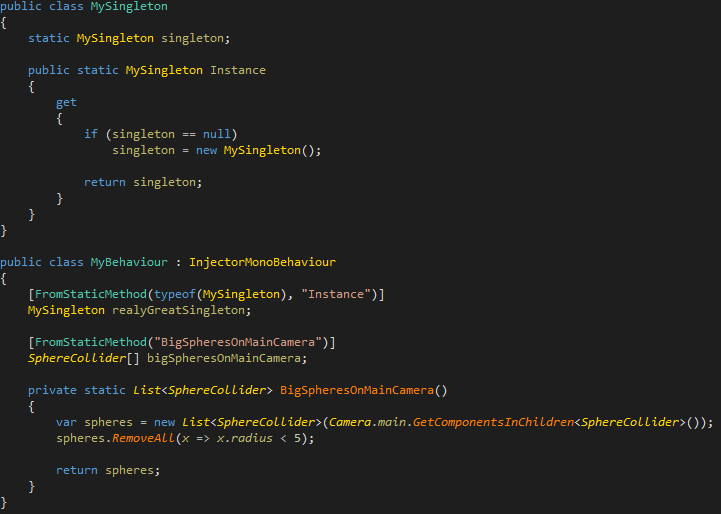
FromStaticMethodAttribute(*Type* type, *string* method)

FromStaticMethodAttribute (*string* method)

Attribute that will call static method or property with specified name on specified type

*Type* type – the type that implements method or property that should be called (if not specified, will attempt to get from the type where the field is declared)

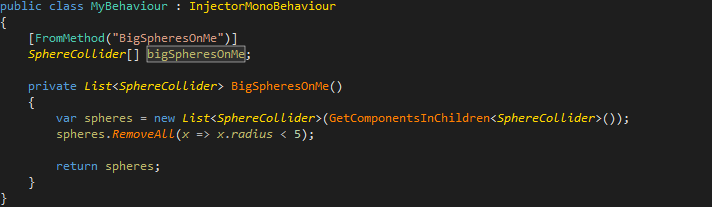
*string* method – the name of the method or property that should be called to resolve dependency

****

FromMethodAttribute (*string* method)

Attribute that will call method or property with specified name on specified type

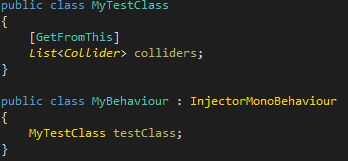
*string* method – the name of the method or property that should be called to resolve dependency

****

# Nested object fields

You may want to have class that contains another struct, or class, that can contain fields that should be injected.

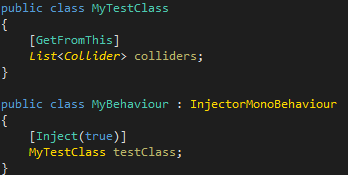
For instance:



For security and preformance reasons, it will not be resolved automaticaly.

To make it possible, you should use special attribute – InjectAttribute(*bool* fillNulls= false).

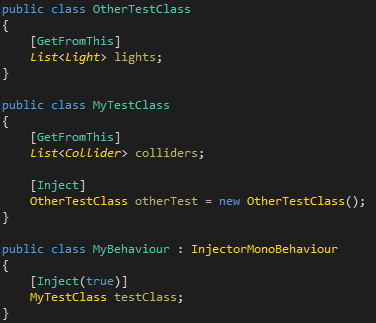
So the same code will look like:

****

Now your field will be injected with using Colliders on the GameObject where MyBehaviour instance is placed.

Fill nulls parameters means that injector should create the instance of the class and then inject it. If not, you’ll get an error.

Also you can make as many nesting levels as you wish. Let’s look at the example of two levels of nesting.

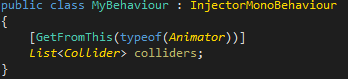


Please note that the otherTest field was initialized in constructor, so we are not passing paramter to create the instance for the field.

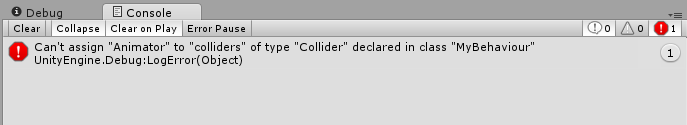
# What if I do something wrong?

First of all, don't do it ☺

But if you did, let's say something like this (which is not good as you can't assign animators to list of colliders):

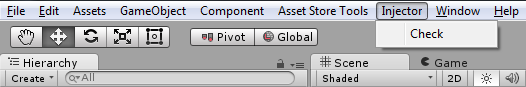


As soon as you let Unity rebuild your project, you’ll be notified by error in the console like you can see on screenshot:



All errors that can be found statically will be shown is such way to you.

# What if I want to check for errors manually?

No problem! You just can use check manually. Just use toolbar menu: **Injector →Check**

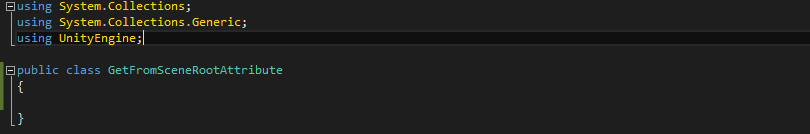
# How can I create my own attributes?

It’s easy! Just derive from:

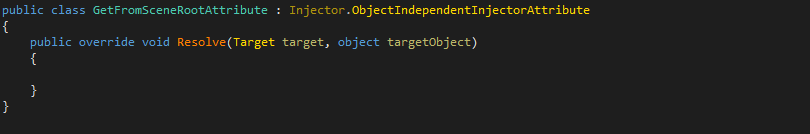
* ObjectIndependentInjectorAttribute – if you your resolve doesn’t depends on specific GameObject (FindObjectOfType, for example)
* GameObjectInjectorAttribute – if you your resolve depends on specific GameObject (GetComponent, for example)

Let’s try to create attribute that fill your field or property with root objects of the scene!

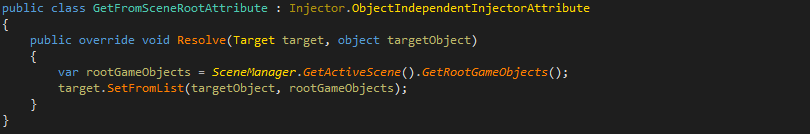
Define class with prefered name:



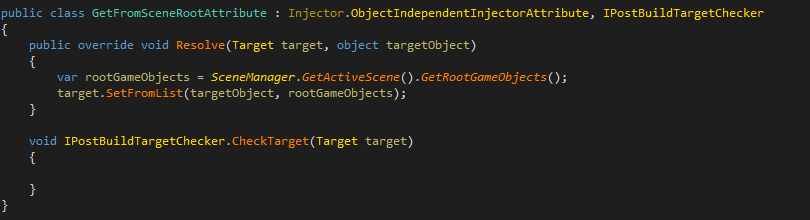
Inherit it from one of the base attribute classes (in our case from ObjectIndependentInjectorAttribute) and implement public override void Resolve(Target target, object targetObject) method:



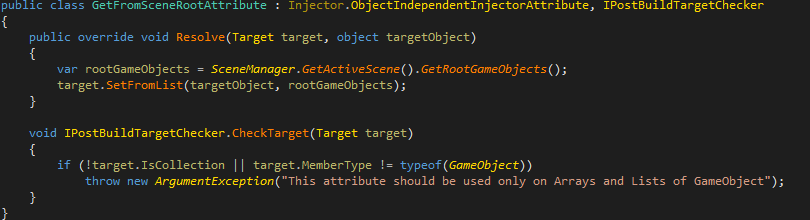
Implement method behaviour:



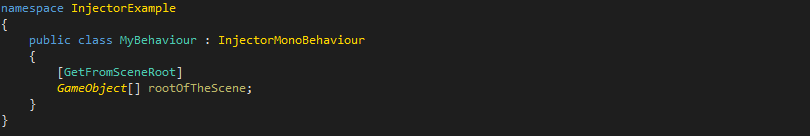
In short terms, it’s done! But for safety purposes we have to add some checks to our method. For instance it should be usied only for collections, because it makes no sence for single GameObject field. In order to check target we have to implement interface IPostBuildTargetChecker:



Now we have to check is it okay to be on that target:



It’s done! Now we can use it on any of our behaviours:



**Thanks for your attention!**