SerializedTypeRestriction

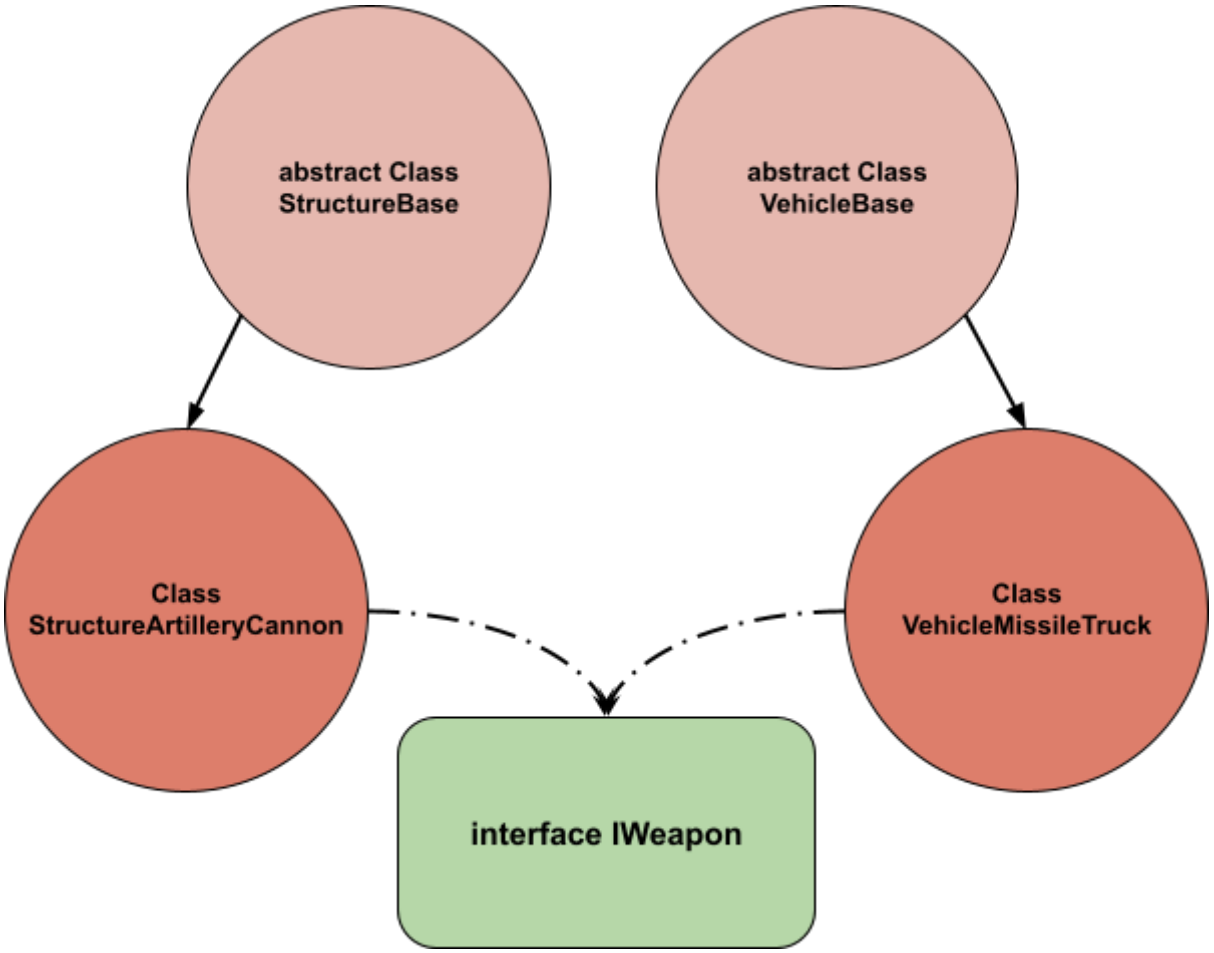
C# for UnityEngine attribute - User Guide

What is the “SerializedTypeRestriction” Attribute

SerializedTypeRestriction is a C# attribute for UnityEngine **fields serialized by reference**. It is used to **restrict** and **help select** what objects you can assign to a given field (variable), **depending on** the **base class** or **interface** required.

This way, you can easily create **flexible** and **robust code** based on **SOLID principles**.

*E.G.: you can create a soldier system where each soldier unit can be assigned a different weapon, where you don’t depend on a single, monolithic WeaponBase base class, but instead using the flexible and powerful programming tool of **interfaces**.*

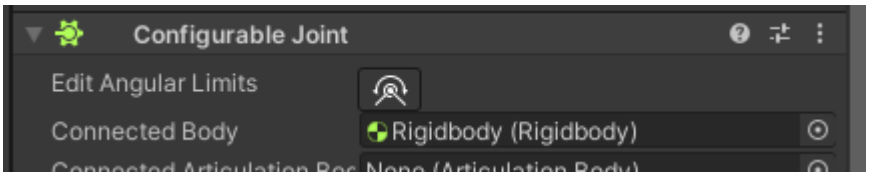


This way, you can make independent base classes while retaining the power of accessing them by a common specialization lower down the inheritance hierarchy tree.

SerializedTypeRestriction Attribute requirements

The **SerializedTypeRestriction** attribute can be applied to members which:

- Are **fields** (variables) **serialized** by **reference**.



*The "ConnectedBody" field of a joint is a **serialized reference field** of type UnityEngine.Rigidbody*

- Their **type** is **UnityEngine.Object**.

```
[SerializeField]
[SerializedTypeRestriction(typeof(IWeapon))]
private UnityEngine.Object _mainWeapon = null;
```

*UnityEngine's UI needs to be able to handle the reference serialization, thus **backing fields** must be of type **UnityEngine.Object***

- Alternatively, a **generic list** of type **UnityEngine.Object** can be used.
  The attribute will be applied to **each** item **independently**.

```
[Tooltip("Tooltip text")]
[SerializeField]
[SerializedTypeRestriction(typeof(ISupport))]
private List<UnityEngine.Object> _supportSources = null;
private IList<ISupport> _supportSourcesAccessor = null;
public IList<ISupport> supportSources
{ get {
        if (this._supportSourcesAccessor == null && this._s
        { this._supportSourcesAccessor = new PHATASS.Utils.

        return this._supportSourcesAccessor;
    }
}
```

*We can also use **Lists**, which offer a powerful way to apply the **SerializedTypeRestriction** attribute to a **vector** of items*

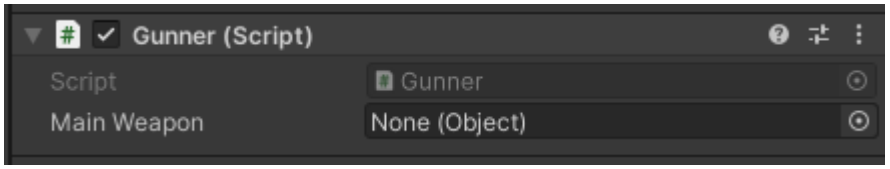
The **SerializedTypeRestriction** attribute takes a **single parameter**: the **type requirement**. This usually will be an **interface** or **abstract class**. Objects assigned will be required to **match** this type, by **implementing** or **inheriting** given interface or class.

```
[SerializedTypeRestriction(typeof(IWeapon))]
```

*When declaring the attribute, we give it the **IWeapon interface** so we can assign any object that **implements IWeapon***

UI on SerializedTypeRestriction fields

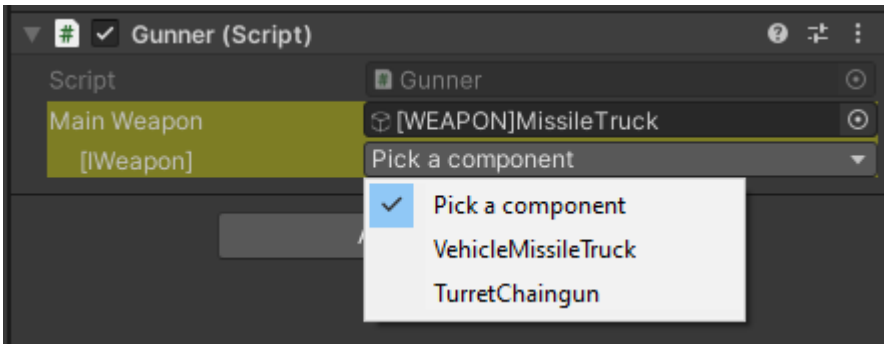
When properly **applied to** a **serialized** reference **field**, UnityEngine's UI should initially display it as a usual **reference field**.



*UnityEngine's UI lets you **drag-and-drop** any UnityEngine.Object, like **GameObjects**, **MonoBehaviours** or **ScriptableObjects***

However, when we try to **assign** an **object** to the **SerializedTypeRestriction field**, the displayed **UI changes depending** on **type** of the assigned object:

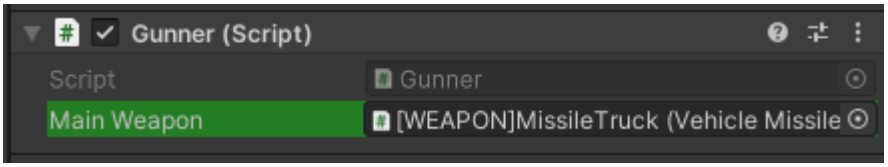
- For a **UnityEngine.GameObject**, a dropdown **picker** is displayed, listing **any components** in the GameObject that **match** the **type given** in the attribute**.**



*Here, **every component** of the GameObject that implements the interface **IWeapon** is displayed*

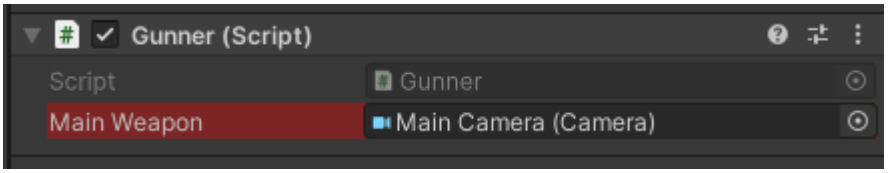
Once any such **option** is **picked**, the serialized **field** will be **updated** to reference that object.

- For any object that **matches** the given **type restriction**, a **green underlay** is displayed on the field to indicate the object is **valid**.



*A serialized object that appropriately **fits** given **type restriction** is **valid***

- **Non-GameObjects not matching** the **type restriction** are **not valid.** They are displayed with a **red underlay.**



*Camera does **not** implement **IWeapon**, and is **not** a **GameObject**, so it is **not valid***

The content of not valid fields will still be serialized so you can easily find any errors on your project. However, this means the getter property must filter these.

How to use the SerializedTypeRestriction in your own code

Copy-pasteable code samples available in the quick guide

As with any other attributes, **applying** the SerializedTypeRestriction attribute to a **field** is as simple as adding it **before** the **field** declaration, alongside any **required parameters**.

The **SerializedTypeRestriction** attribute takes a **single parameter:** a **System.Type** object, representing the type requirement we want to apply to this field.
This **field** also **needs** to be **serialized** so we also add a **SerializeField** attribute.

```
[SerializeField]
[SerializedTypeRestriction(typeof(IWeapon))]
private UnityEngine.Object _mainWeapon = null;
```

*We apply the **SerializedTypeRestriction** attribute with an **IWeapon** type constraint.*
This lets us use this field to serialize any object that implements IWeapon

Since this **backing field**'s type **must** be **UnityEngine.Object**, but we want to get objects of type IWeapon, we use a **getter property** to **access** and cast this field.

```
public IWeapon mainWeapon                          // Getter property
{ get {
    if (this._mainWeapon == null) { return null; }  // Manually handle null
    else { return this._mainWeapon as IWeapon; }    // Cast and return
}}
```

*We will **only access** our mainWeapon field through **this property**.*
If we want to make our weapon publicly accessible, we can make this property public.

> ⚠️ **cautionary warning** ⚠️
> *Due to the way UnityEngine handles wrapping of **null** instances of **UnityEngine.Object**,*
> *a **type-casted null instance** of UnityEngine.Object is **not recognized** as null.*
> *This means our **getter property** needs to specifically **handle null** values **before casting**.*

Lists and arrays

The **SerializedTypeRestriction** attribute can also be used to create array-like **object lists**.
We only need to apply the attribute to a **generic list** of type **UnityEngine.Object.**

```
[SerializeField]
[SerializedTypeRestriction(typeof(ISupport))]
private List<UnityEngine.Object> _supportSources = null; // Backing field. Don't access th
```

Each element** of this list will have the SerializedTypeRestriction UI applied **separately

Casting a **List<T>** isn't as simple as casting a single value, so we use a **wrapper** that will help us filter and cast each element of that list.

```
private IList<ISupport> _supportSourcesAccessor = null; // Wrapper cache. We store here our IList<ISupport> wrapper for easy access
public IList<ISupport> supportSources                   // Getter property. Access this to get a usable IList<ISupport>
{ get {
        if (this._supportSourcesAccessor == null && this._supportSources != null) //create accessor if unavailable
        { this._supportSourcesAccessor = new PHATASS.Utils.Types.Wrappers.UnityObjectListCastedAccessor<ISupport>(this._supportSources); }

        return this._supportSourcesAccessor;
    }
}
```

*This wrapper will **cast** the items of our **list** to the **desired** type **ISupport** on-the-fly.*

To facilitate this, this package includes an easy-to-use IList **wrapper**, with which we can **access** a List<UnityEngine.Object> as if it was an **IList<T>** of our **desired type**.