Guide to Variable Objects

<Something something...>

This guide is intended for those who are using Variable Objects for the first time. It is mainly focused on programmers, but anyone with basic coding know-how should be able to follow along.

It assumes that you have some experience with Unity; at the very least, you need to have completed a couple of entry-level Unity tutorials. As of this writing, good, beginner tutorials are the Interactive Tutorials (https://unity3d.com/learn/tutorials/s/interactive-tutorials) and the Roll-a-ball Tutorial (https://unity3d.com/learn/tutorials/s/roll-ball-tutorial).

This guide is divided into two parts. The first part focuses on the basics of using the package, while the second part steps through more specific, niche tasks. Once reading the first part, you should be able to use the Variable Object package for your projects. The second part is for those who want to do more with the system, or who need to do a specific task.

Table of Contents

Part I – The Basics

Why Bother With Variable Objects?

Before we dive into variable objects, it will be helpful to review the ways a Unity program normally stores data.

Using a Variable Inside the Component

The first and most simple way is to create a variable in a MonoBehaviour object. For example, we could have a player component that looks like this:

```
using UnityEngine;

public class Player : MonoBehaviour {

    public string name = "Fred";
    public int score = 0;
    public int hitpoints = 10;
}
```

This player component will track the player's score, their health, and their name. The player's points might go up when they collect treasure, while their hitpoints might decrease when they touch some spikes. This system is pretty straightforward, as the data is kept right with the player's GameObject. We could even create many players, and there would be no problems.

However, this system falls apart if the player's GameObject is destroyed, because then the Player component is also destroyed, and all of the data is lost. We might want to keep track of their score and their name but respawn the player. But, instead, their name is reset to "Fred" and they have 0 points! Even if we avoid deliberately destroying the player's GameObject, Unity will usually destroy the player's GameObject when it loads a new scene.

Furthermore, if we want to create UI that shows the player's score and health, we have to find some way of finding the player object and then tracking these variables. We could use Unity's `FindObjectsOfType` function, but this is slow and could lead to issues if we have many players. And what will we do if the player's GameObject isn't in the scene?

Using a Static Variable or Singleton

We don't have to keep this data inside the component. We can "solve" our problem by making our variables static, like this:

```
using UnityEngine;

public class Player : MonoBehaviour {

    static public string name = "Fred";
    static public int score = 0;
    public int hitpoints = 10;
}
```

This causes the name and score variables to be kept outside the Player component. It is the same across all instances of our Player component. Thus, if the player's GameObject gets destroyed, this data is not lost. The next Player component will have the old score and name values. We also don't need to find the Player component, because it's not part of the Player class. It's totally separate, so making our UI is simpler.

This works fine unless we decide we want another player. If we try to create a second player, they will have the same name and score. This is probably not what we want. We would have to create a list of scores and a list of names, and then each Player component would have to decide which name and score to use.

Static variables don't work with the Unity Editor, either. Static variables cannot be changed using Unity's Inspector window. Their defaults cannot be changed without editing the code, and they cannot be viewed or changed during runtime.

The singleton pattern is another solution. There are many Unity-specific implementation of the singleton on the internet, but singletons only solve the second problem. (A singleton component *can* be edited using Unity's Inspector.) They do not fix the first problem, and they bring a slew of other issues.

Whether using static variables or singletons, we cannot escape the fact that code controls where we store our values. Our components are rigid, and cannot be used for anything other than their intended purpose.

Using a Variable Object

A more flexible option is to use a variable object. By using a variable object, the data is kept outside of the Player component, much like we were using a static variable. This means that data which we want to keep (such as the name and score) will be preserved after the player's GameObject is destroyed.

Unlike static variables, however, Unity's Inspector can change a variable object's value on-the-fly. Also, unlike static variables, individual variable objects can be shuffled around using the Inspector. This means that one component may be easily reused.

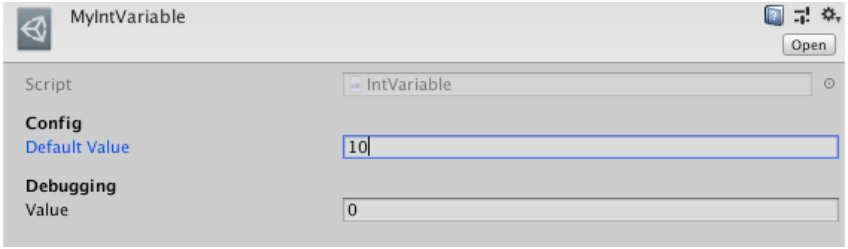
For example, we could use the same Player component on four player GameObjects. In a competitive mode, we could have track each player's score and name separately. Then, without changing our Player component, we could create a cooperative mode that had players sharing a score variable. This means our Player component has become more reusable without adding much of any code at all.

Getting Started

At this point, you should already have the Variable Objects package installed in your project. If not, go read the “Installing” file, and come back here when you’re done.

A variable object, is an asset that gets stored in your Unity project, similar to a prefab or a material. However, a variable object simply stores a value, such as a color, a position, or a number. Much like variables in C#, variable object are strongly typed. That is, a variable object which holds GameObjects cannot store a Vector3 or a Transform.

We’ll start by creating an IntVariable. Within the Unity project with the package installed, starting with the dropdown menus at the top, do `Assets => Create => Variable => Int`, and Name your new IntVariable *MyIntVariable*. It should become a file in your project. Once you select MyIntVariable, you should see two fields: Default Value and Value (see picture). In general, you only want to change Default Value. For demonstration purposes, change Default Value to 10, and Value to 0.



If you enter Play Mode now, you would see that Value changed to 10. This is as expected; Default Value is what you’ll primarily be changing when setting up variable objects. The Value field is only there so that you may debug your game while it is running.

But our new variable is rather pointless without a script to make use of it! Create a new script and name it *AddPointsOnPress*. Then, using your IDE of choice, put this code in the new script:

```
using UnityEngine;
using ReachBeyond.VariableObjects;                    // (1)

public class AddPointsOnPress : MonoBehaviour {

     [SerializeField] private IntReference score;     // (2)
     [SerializeField] private string buttonName = "Fire1";

     // Every frame...
     void Update () {
          // ...check if the button is being pressed...
          if(Input.GetButtonDown(buttonName)) {
               // ...and gain a point if it is.
               score.Value++;                         // (3)
          }
     }
}
```

(We use “[SerializeField]” in examples because it is a more OOP-friendly way of allowing Unity to edit something. It is similar to making the variables public, but it keeps other components from having access to these variables.)

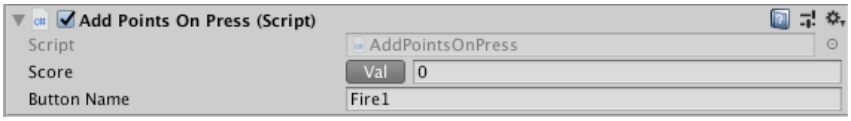
Line (1) allows us to use the Variable Object library. Line (2) declares a potential reference to an IntVariable, while (3) increases the actual score. Later, we will cover why we need to write

```
score.Value++;
```

instead of

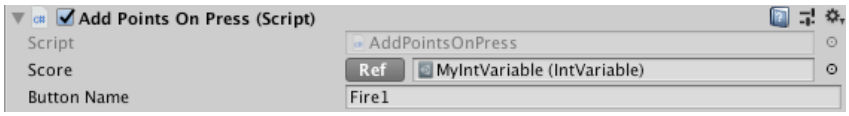
```
score++;
```

For now, let’s go back to the Unity Editor. Attach this script to an active GameObject in your scene, such as the scene’s camera object. You should see this:



If you enter Play Mode, you should see that the *Score* number will increase every time you hit the “Fire1” button. (*Left mouse button* or *Left Control*, by default.) However, this value is currently local to this instance of *AddPointsOnPress*. This is as intended; since we have not attached *Score* to any variable objects, the value is kept within our *AddPointsOnPress* component.

Now, let’s change *Score* so it uses the *MyIntVariable* we created a few moments ago. Leave Play Mode, and click the “Val” button. It should change into “Ref,” and the number field will say “None (Int Variable).” Drag-and-drop *MyIntVariable* into this new field. It should look like this when you are done:



Now, when you enter Play Mode and hit "Fire1," you will not see *Score* increase. However, if you select *MyIntVariable* within the Project Window, you will see that its *Value* does increase. Effectively, we've moved *AddPointsOnPress's Score* variable into an object that the rest of our project may access.

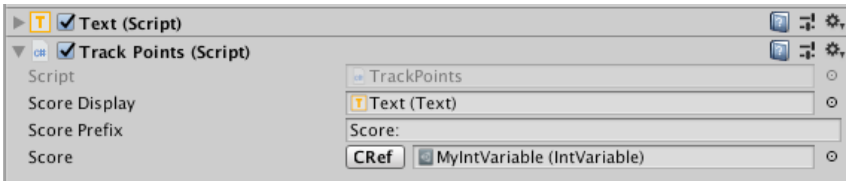
Let's take advantage of this! We'll create a script which can display an IntVariable to a GUI text element. We'll use this create a HUD that shows the player their score. Create a new script, name it *TrackPoints*, and put this in it:

```
using UnityEngine;
using UnityEngine.UI;
using ReachBeyond.VariableObjects;

public class TrackPoints : MonoBehaviour {

     [SerializeField] private Text scoreDisplay;
     [SerializeField] private string scorePrefix = "Score: ";

     [SerializeField] private IntConstReference score;    // (1)

     // Every GUI frame...
     void OnGUI() {
          // ...update the text display.
          scoreDisplay.text = ScorePrefix + score;        // (2)
     }
}
```

Line (1) creates a constant integer reference called *score*. Just like before, this can either refer to a local value, or it can refer to a variable object. (2) uses *score*'s value. Alternatively, we could have written

```
scoreDisplay.text = ScorePrefix + score.ConstValue;
```

Again, we'll go in more depth on this topic later. For now, we'll return to Unity so that we can set up the score display. In the Hierarchy Window, click `Create => UI => Text`. Attach our *TrackPoints* script to it. Set it up like this:



(Feel free to configure the Text component so that it is more readable. It will probably be hard to read in its default state.)

Finally, enter Play Mode. When you press "Fire1," you should see the score text update. If not, go back and make sure you got everything working. Otherwise… ***CELEBRATE!***

Constant Reference or Reference?

Observant readers will notice a difference between the references used in the two components we've created. The first one uses *IntReference*, while the second uses *IntConstReference*. These are almost exactly the same. They can both contain a localized value or point to a variable object. They both have *ConstValue* fields, which let your code access the value which the reference is pointing to.

However, as the name implies, *ConstValue* cannot be changed directly. In either *TrackPoints* or *AddPointsOnPress*, you would get an error if you wrote

```
score.ConstValue = 3;
```

since *ConstValue* is readonly. This is where *IntReference* is different: it has a *Value* field. Code may use *Value* to modify the value pointed to by the reference, much like we did in *AddPointsOnPress*. If you try to access *Value* in an *IntConstReference*, you will get an error.

Why bother with constant references when they can't do as much? It makes designing with variable objects easier. If you look at *TrackPoints* in the Unity Inspector, you'll see that its button says "CRef" or "CVal" (meaning "Constant Reference" or "Constant Value"). The button for *AddPointsOnPress*, on the other hand, says "Ref" or "Val," and is a different color.

This visual difference is intentional. If we give *TrackPoints* access to a variable object, we know that *TrackPoints* won't change the value stored by the variable object, because it uses a constant reference. But if we give *AddPointsOnPress* access to a variable object, we will be expecting it to modify the variable object. We can tell ahead of time what variable objects will be changed by each component.

This feature will become even more useful as new tools get added.

Implicit Conversion

You could write this in either script above:

```
int num1 = 5 + score;
```

Since the both *IntConstReference* and *IntReference* implicitly use *ConstValue* when they are referenced this way, Unity's compiler understands what you are trying to do. However, there could be edge-cases where you will need to explicitly use *ConstValue*.

So Many Types!

Up to this point, we have been working just with *IntVariable* and its reference types, *IntReference* and *IntConstReference*. As you've probably figured out, this system can work with many different datatypes, such as bools, floats, and GameObjects. (And you can expand the system to work with your own, custom datatypes!) The fundamentals that we've covered are the same across all datatypes. However, there is one caveat to point out…

Classes Are Special

For reasons previously mentioned, this code is not valid:

```
[SerializeField] private TransformConstReference targetTransform;

void Start() {
     targetTransform.ConstValue = transform;
}
```

However, the following code is perfectly valid:

```
[SerializeField] private TransformConstReference targetTransform;

void Start() {
     targetTransform.ConstValue.position = Vector3.zero;
}
```

Why is this? Take a moment to figure it out. (Hint: do an internet search for "mutable class C#")

When a constant reference points to a class, it only stops the class from being reassigned. That is, we can't change *ConstValue* itself. However, variables of the target object can still be changed. This is because most classes are "mutable," meaning that they have elements which may be modified.

Thus, if you have a mutable class (such as Transforms or GameObjects) pointed to by a constant reference, parts of it may be modified. However, immutable types (such as integers or floats) may not be changed this way.

Registraters

To be written

Part II – Specific Tasks

Adding Support For Custom Types

To be written