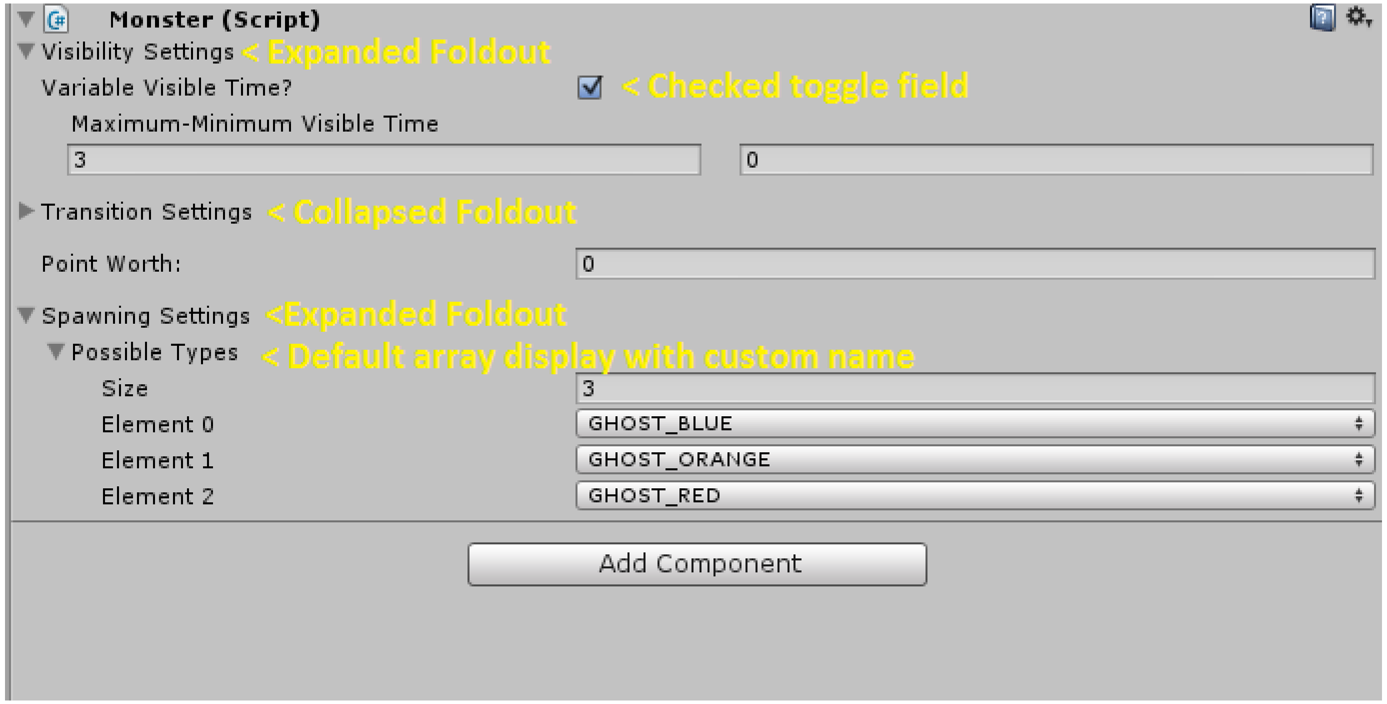
Section Four: Custom Editor

In this section of the unit you will explore how to create a custom editor for a class. You will be utilizing a class named Monster, which looks like the following:

```
using UnityEngine;
using System.Collections;
public enum MonsterType
{
    GHOST_BLUE,
    GHOST_ORANGE,
    GHOST_RED,
    LENGTH
}
public class Monster : MonoBehaviour {

    float initialHeight;

    #region Visible Time
    //Can this monster be visible for different lengths of time?
    private bool variableVisibleTime = false;
    public bool VariableVisibleTime
    {
        get { return variableVisibleTime; }
        set { variableVisibleTime = value; }
    }

    //How long the monster is visibsle for (max)
    private float visibleTimeMax = 3;
    public float VisibleTimeMax
    {
        get { return visibleTimeMax; }
        set { visibleTimeMax = value; }
    }

    //How long the monster is visible for (min) Only applies if Variable Visible Time is true
    private float visibleTimeMin;
    public float VisibleTimeMin
    {
        get { return visibleTimeMin; }
        set { visibleTimeMin = value; }
    }
    #endregion

    #region Transition Time
    private bool variableTransitionTime = false;
    public bool VariableTransitionTime
    {
        get { return variableTransitionTime; }
        set { variableTransitionTime = value; }
    }

    private float transitionTimeMax = 2;
    public float TransitionTimeMax
    {
        get { return transitionTimeMax; }
        set { transitionTimeMax = value; }
    }

    private float transitionTimeMin;
    public float TransitionTimeMin
    {
        get { return transitionTimeMin; }
        set { transitionTimeMin = value; }
    }
    #endregion


    private int pointWorth;
    public int PointWorth
    {
        get { return pointWorth; }
        set { pointWorth = value; }
    }

    //The types of monsters this monster can be
    public MonsterType[] possibleTypes;
    //The type of monster this monster currently is
    MonsterType myType;

    public bool disappearing;
    public bool visible;
}
```

Given this class, you will create a custom inspector that allows the user to edit the variables of the class. Although Unity provides a default inspector, there may be time when you wish to display information in a different way. Using a custom inspector allows you to do exactly that. Below is what the final inspector will look like:



The inspector is broken up into multiple parts. The first part is labeled Visibility Settings. This is a foldout, which can be expanded to display the options or collapsed to hide them.

When open, the user will be presented with a choice: Is there a variable visible time (a minimum and maximum time) or not (a set visible time).

If there is a variable visible time, there will be two float fields for the user to input the minimum and maximum time. If there is not a variable visible time, then only one float field will appear for the user to input the set visible time.

The next section is a repeat of the transition time, but for the transition setting instead.

Next is the point worth of the monster, this is a single float field.

Below that is where you can see the spawn settings foldout. For this section, you will use the default array display to allow the user to set the types of spawns.

STEP ONE: SETUP

The first thing you need to do is create a script in the editor folder. The script should be named the same name of the class you will be creating the inspector for, with "Editor" tacked on. For example, the class we are creating the custom editor for is Monster. Therefore, the editor script will be named MonsterEditor.

The script should use the UnityEditor, and also extend Editor.

```
using UnityEngine;
using System.Collections;
using UnityEditor;

public class MonsterEditor : Editor {


}
```

The next thing you need to do is specify that the script will be an editor script for the monster class. To do that, you use the attribute CustomEditor(type Type).

```
using UnityEngine;
using System.Collections;
using UnityEditor;
```

```
[CustomEditor(typeof(Monster))]
public class MonsterEditor : Editor {


}
```

Displaying items in the inspector is similar to when you were displaying items in the window. You use the EditorGUI and EditorGUILayout classes to display items in the inspector. Also similar to when creating the window, you need to get a reference to the object you are editing. However, instead of an object you will want to get a reference to the script you are going to be editing. In this case, the script will be on a Monster script (since that is what you are editing in the inspector).

To start, you will declare a few variables. You need a variable to store the script that you will be editing, as well as some variables to determine if the visibility and transition for the Monster will be a range or a set number.

```
[CustomEditor(typeof(Monster))]
public class MonsterEditor : Editor {

    Monster monsterScript;
    bool visiblerange = false;
    bool transitionRange = false;


}
```

Next, you need to get the reference to the Monster script that you will be editing. With an editor script such as this, there is something called a target. This is what the target of the actions are going to be. This is a built-in variable that you can access within the script. You can use the target to get a reference to the script, and then edit it from there. Target is an EditorObject by default, and needs to be casted to the Monster type.

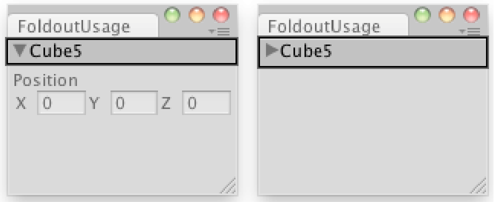
```
    void Awake()
    {
        monsterScript = (Monster)target;
    }
```

Now with the setup complete, you can begin to code the custom inspector!

STEP TWO: POPULATING THE GUI

When you are working with windows, you edit the GUI in the OnGUI function. For editing inspector windows, you use a similar function: OnInspectorGUI.

Here you will add a few more types of editor items to your arsenal:

- EditorGUILayout.Foldout(bool foldout, string label)
    - This will display a foldout menu, such as an array menu.
    - 

- EditorGUI.indentLevel
    - This will increase (or decrease) the indent level of the GUI items, so you can position some items further indented than others
- EditorGUILayout.PrefixLabel(string label)
    - This will display a label before the next field
- EditorGUILayout.Space
    - This works the same way as EditorGUILayout.Separator
- EditorGUILayout.PropertyField(SerializedProperty property, bool includeChildren)
    - This is used to make an array show up in a custom inspector the same way it would show up in the default inspector. You will look at making custom array displays in the next section.

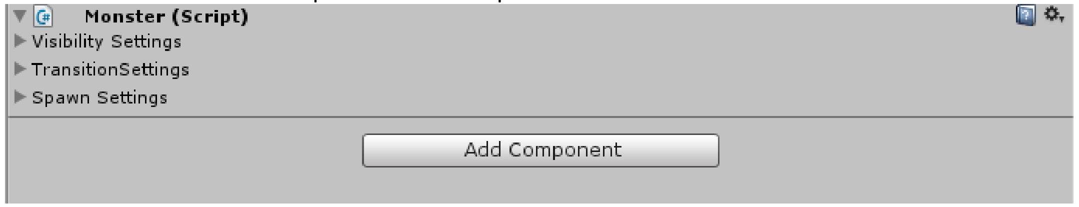
FOLDOUT DISPLAY

A foldout is a useful item to hide settings. Sometimes if there are a significant amount of settings, you can use a foldout to control what the user sees as a certain time. When working with a foldout, you need to keep track of weather the foldout is expanded (true) or collapsed (false). Then you run an if statement – if the foldout is visible, you render new items. If it is not visible, you do nothing.

To start with you can start by collecting if the foldouts are expanded or collapsed. The value of each foldout will be stored in the variables already created. Similar to how you worked with the EditorGUILayout for the window, you need to assign the value back into the variable.

```
public override void OnInspectorGUI()
{
    //Display a foldout, with the default value stored in visibleRange and the label "Visibility Settings"
    //As the foldout is manipulated, save changes into visibleRange
    visibleRange = EditorGUILayout.Foldout(visibleRange, "Visibility Settings");

    transitionRange = EditorGUILayout.Foldout(transitionRange, "TransitionSettings");

    spawnTypes = EditorGUILayout.Foldout(spawnTypes, "Spawn Settings");
}
```

CHECKPOINT!

1. Save your script
2. Assign the Monster script to the main camera.
3. Click the main camera to view the inspector of the script.
    a. 
    Monster (Script)
    Visibility Settings
    TransitionSettings
    Spawn Settings

4. If you click on the arrow next to any of the three settings, nothing should happen.
    a. This is because we have not programmed in any new items, we only programmed the display of a foldout option!

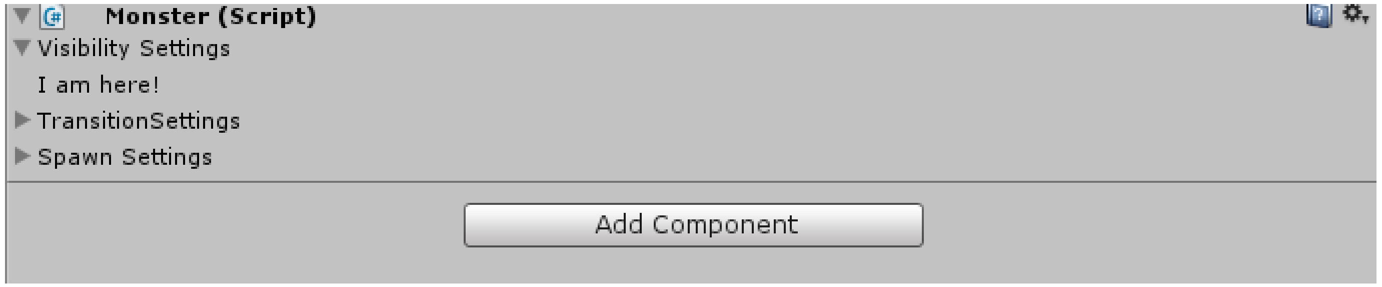
VISIBILITY SETTINGS

Next you can display something inside of each foldout. For now, you can just display a label so you can see how the foldout functions.

Below where you create the foldout, create an if statement. You want to check if the bool value visibleRange is true (this represents an expanded foldout menu). If it is true, display a simple label.

```
    public override void OnInspectorGUI()
    {
        //Display a foldout, with the default value stored in visibleRange and the label "Visibility Settings"
        //As the foldout is manipulated, save changes into visibleRange
        visibleRange = EditorGUILayout.Foldout(visibleRange, "Visibility Settings");
        if (visibleRange)
        {
            EditorGUILayout.LabelField("I am here!");
        }

        transitionRange = EditorGUILayout.Foldout(transitionRange, "TransitionSettings");

        spawnTypes = EditorGUILayout.Foldout(spawnTypes, "Spawn Settings");
    }
```

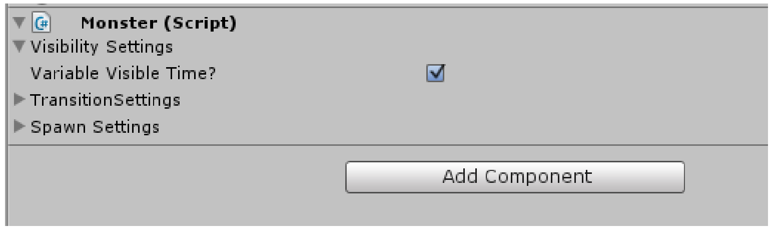
If you save the script and refresh the object (by clicking off and back onto the main camera) you should now see the following when the foldout is expanded:



Now you can begin added actual controls.

First, you will want to present the toggle button. However, instead of the toggle button affecting a local variable in the editor script you are going to want to affect the actual Monster script. Since you got a reference to the script at the beginning , you can access the VariableVisibleTime like any other variable.

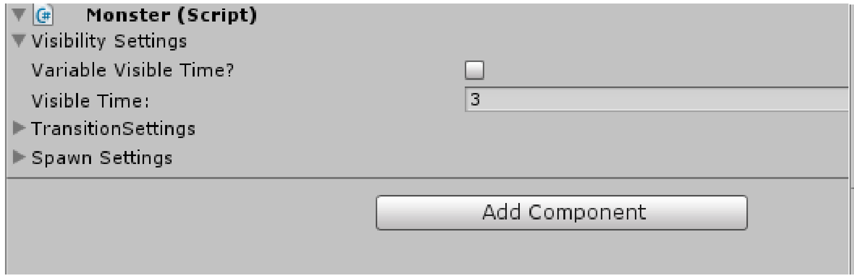
```
        if (visibleRange)
        {
            //The variable VariableVisibleTime on the monsterScript should equal the setting the user picked for the toggle
            //The toggle should display the variable VariableVisibleTime
            monsterScript.VariableVisibleTime = EditorGUILayout.Toggle("Variable Visible Time?",
                                                    monsterScript.VariableVisibleTime);
        }
```

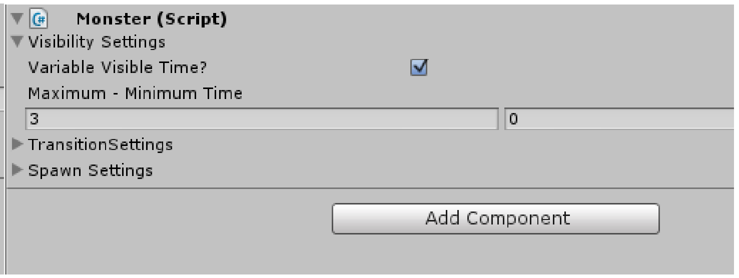


Monster (Script)
Visibility Settings
Variable Visible Time? ✓
TransitionSettings
Spawn Settings

Next you need to determine what to display. If the variable visible time is true, it means that you need to account for a minimum and maximum value. If it is false, you only need to show one value (which will be stored in the Max setting for the monster).

```
        if (visibleRange)
        {
            //The variable VariableVisibleTime on the monsterScript should equal the setting the user picked for the toggle
            //The toggle should display the variable VariableVisibleTime
            monsterScript.VariableVisibleTime = EditorGUILayout.Toggle("Variable Visible Time?",
                                                    monsterScript.VariableVisibleTime);

            if (monsterScript.VariableVisibleTime)
            {
                EditorGUILayout.LabelField("Maximum - Minimum Time");
                EditorGUILayout.BeginHorizontal();
                //Since I manually created the label above, I am using an overload of the Float Field so no default label
                //is displayed immediately before it.
                monsterScript.VisibleTimeMax = EditorGUILayout.FloatField(monsterScript.VisibleTimeMax);
                monsterScript.VisibleTimeMin = EditorGUILayout.FloatField(monsterScript.VisibleTimeMin);
                EditorGUILayout.EndHorizontal();
            }
            else
            {
                EditorGUILayout.BeginHorizontal();
                EditorGUILayout.PrefixLabel("Visible Time:");
                //Again, I manually created a prefix label to display so I am overloading the float field so no
                //default label is shown
                monsterScript.VisibleTimeMax = EditorGUILayout.FloatField(monsterScript.VisibleTimeMax);
                EditorGUILayout.EndHorizontal();
            }
        }
```


Monster (Script)
Visibility Settings
Variable Visible Time? ☐
Visible Time: 3
TransitionSettings
Spawn Settings

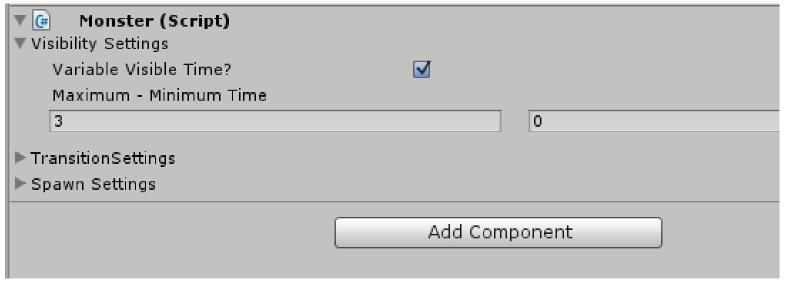

Monster (Script)
Visibility Settings
Variable Visible Time? ✓
Maximum - Minimum Time
3 0
TransitionSettings
Spawn Settings

There are just a couple of more edits to make. First, it would make more sense if everything inside of the foldout was indented so that you can see the settings belong to the foldout. This can easily be done with EditorGUI.indentLevel

```
        if (visibleRange)
        {
            //Increase the indent level by one
            EditorGUI.indentLevel++;
            <code omitted>

            if (monsterScript.VariableVisibleTime)
            {
                <code omitted>
            }
            else
            {
                <code omitted>
            }

            //Reduce the indent level by one
            EditorGUI.indentLevel--;
        }
```

Lastly, add a space after the visibility settings before the next foldout appears.

```
        if (visibleRange)
        {
           <code omitted>
        }

        EditorGUILayout.Space();
```


Monster (Script)
Visibility Settings
Variable Visible Time? ✓
Maximum - Minimum Time
3 0
TransitionSettings
Spawn Settings

TRANSITION SETTINGS

The transition settings will be the exact same as the visibility settings, but for the appropriate variables.

```
transitionRange = EditorGUILayout.Foldout(transitionRange, "Transition Settings");
if (transitionRange)
{
    //Increase the indent level by one
    EditorGUI.indentLevel++;

    //The variable VariableTransitionTime on the monsterScript should equal the setting the user picked for the
    // toggle
    //The toggle should display the variable VariableTransitionTime
    monsterScript.VariableTransitionTime = EditorGUILayout.Toggle("Variable Transition Time?",
                                            monsterScript.VariableTransitionTime);

    if (monsterScript.VariableTransitionTime)
    {
        EditorGUILayout.LabelField("Maximum - Minimum Time");
        EditorGUILayout.BeginHorizontal();
        //Since I manually created the label above, I am using an overload of the Float Field so no default label
        //is displayed immediately before it.
        monsterScript.TransitionTimeMax = EditorGUILayout.FloatField(monsterScript.TransitionTimeMax);
        monsterScript.TransitionTimeMin = EditorGUILayout.FloatField(monsterScript.TransitionTimeMin);
        EditorGUILayout.EndHorizontal();
    }
    else
    {
        EditorGUILayout.BeginHorizontal();
        EditorGUILayout.PrefixLabel("Transition Time:");
        //Again, I manually created a prefix label to display so I am overloading the float field so no
        //default label is shown
        monsterScript.TransitionTimeMax = EditorGUILayout.FloatField(monsterScript.TransitionTimeMax);
        EditorGUILayout.EndHorizontal();
    }

    //Reduce the indent level by one
    EditorGUI.indentLevel--;
}
```

```
        EditorGUILayout.Space();
```

SPAWN SETTINGS

The spawn settings will be handled differently. Because the spawn settings are actually an array of an enum they cannot be displayed in the traditional sense. There are many different ways to display arrays in a custom editor, or a custom window. For now, you will explore the most basic way: displaying an array in the default way.

When working with an object and a custom editor, an important thing to remember is that the object is Serialized. Since the object is Serialized, the properties inside of the object are also serialized. Most serialized properties are easy to pull from the script, such as the properties you have been working with. However, with arrays and lists you must access them directly.

The first thing you have to do is update the serializedObject (or the object you are editing). This will sync the properties in the object with the editor script. Then, you can edit the serializedProperties (in this case, the array). After editing the properties, you have to re-sync and apply the changes.

First, create the foldout the same way you did before:

```
        spawnTypes = EditorGUILayout.Foldout(spawnTypes, "Spawn Settings");
        if (spawnTypes)
        {

        }
```

Next, sync the serialized object with the script;

```
        spawnTypes = EditorGUILayout.Foldout(spawnTypes, "Spawn Settings");
        if (spawnTypes)
        {
            serializedObject.Update();
        }
```

Now, display the array as a default inspector array (aka the way the default inspector would show an array). PropertyFiend is what you use to show an array. However, you cannot directly access the array like you did with your previous variables – meaning you cannot do monsterScript.possibleTypes. Instead, you have to search for the serialized property (remember – all properties are serialized on a serializedObject, but arrays have to be directly pulled into the script, unlike the other properties)

```
EditorGUILayout.PropertyField(serializedObject.FindProperty("possibleTypes"), true);
```

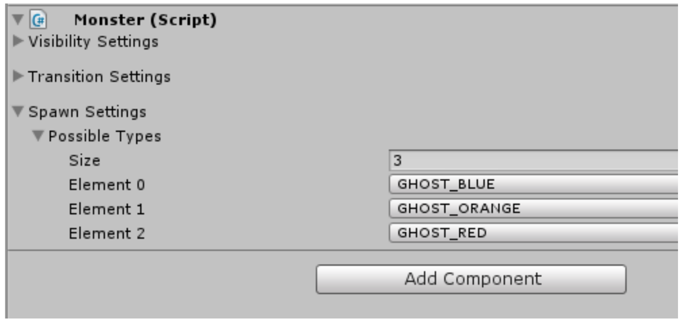
By passing true into the PropertyField function, you are telling unity to include the children of the array (aka each element in the array).

Lastly, you want to apply all of the changes

```
        spawnTypes = EditorGUILayout.Foldout(spawnTypes, "Spawn Settings");
        if (spawnTypes)
        {
            //Sync the object with the script
            serializedObject.Update();

            //Show the array field
            EditorGUILayout.PropertyField(serializedObject.FindProperty("possibleTypes"), true);
        }

        //Apply all changes to the object
        serializedObject.ApplyModifiedProperties();
```

Pretty it up by increasing the indent where appropriate:

```
        spawnTypes = EditorGUILayout.Foldout(spawnTypes, "Spawn Settings");
        if (spawnTypes)
        {
            EditorGUI.indentLevel++;

            //Sync the object with the script
            serializedObject.Update();

            //Show the array field
            EditorGUILayout.PropertyField(serializedObject.FindProperty("possibleTypes"), true);

            EditorGUI.indentLevel--;
        }

        EditorGUILayout.Space();
```



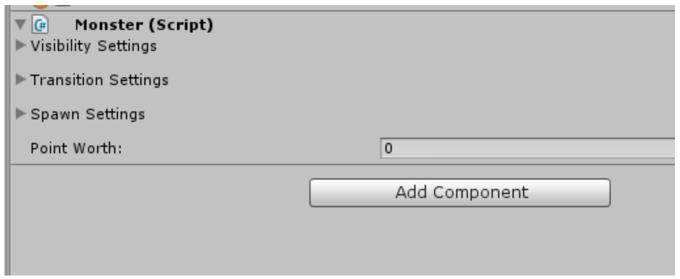
▼ Monster (Script)
▶ Visibility Settings
▶ Transition Settings
▼ Spawn Settings
  ▼ Possible Types

Size	3
Element 0	GHOST_BLUE
Element 1	GHOST_ORANGE
Element 2	GHOST_RED

POINT VALUE

The last field to include is the point value that the monster is worth. You can include this anywhere in the script.

```
        EditorGUILayout.Space();

        //Display the point value
        monsterScript.PointWorth = EditorGUILayout.IntField("Point Worth: ", monsterScript.PointWorth);

        //Apply all changes to the object
        serializedObject.ApplyModifiedProperties();
```



Point Worth: 0

WRAP-UP

You now have the tools to:

1. Create custom inspectors for built-in unity types

COMPLETE CODE

```
using UnityEngine;
using System.Collections;
using UnityEditor;

[CustomEditor(typeof(Monster))]
public class MonsterEditor : Editor {

    Monster monsterScript;
    bool visibleRange = false;
    bool transitionRange = false;
    bool spawnTypes = false;

    void Awake()
    {
        monsterScript = (Monster)target;
    }

    public override void OnInspectorGUI()
    {
        //Display a foldout, with the default value stored in visibleRange and the label "Visibility Settings"
        //As the foldout is manipulated, save changes into visibleRange
        visibleRange = EditorGUILayout.Foldout(visibleRange, "Visibility Settings");
        if (visibleRange)
        {
            //Increase the indent level by one
            EditorGUI.indentLevel++;

            //The variable VariableVisibleTime on the monsterScript should equal the setting the user picked for the toggle
            //The toggle should display the variable VariableVisibleTime
            monsterScript.VariableVisibleTime = EditorGUILayout.Toggle("Variable Visible Time?",
                                                monsterScript.VariableVisibleTime);

            if (monsterScript.VariableVisibleTime)
            {
                EditorGUILayout.LabelField("Maximum - Minimum Time");
                EditorGUILayout.BeginHorizontal();
                //Since I manually created the label above, I am using an overload of the Float Field so no default label
                //is displayed immediately before it.
                monsterScript.VisibleTimeMax = EditorGUILayout.FloatField(monsterScript.VisibleTimeMax);
                monsterScript.VisibleTimeMin = EditorGUILayout.FloatField(monsterScript.VisibleTimeMin);
                EditorGUILayout.EndHorizontal();
            }
            else
            {
                EditorGUILayout.BeginHorizontal();
                EditorGUILayout.PrefixLabel("Visible Time:");
                //Again, I manually created a prefix label to display so I am overloading the float field so no
                //default label is shown
```

```
                monsterScript.VisibleTimeMax = EditorGUILayout.FloatField(monsterScript.VisibleTimeMax);
                EditorGUILayout.EndHorizontal();
            }

            //Reduce the indent level by one
            EditorGUI.indentLevel--;
        }

        EditorGUILayout.Space();

        transitionRange = EditorGUILayout.Foldout(transitionRange, "Transition Settings");
        if (transitionRange)
        {
            //Increase the indent level by one
            EditorGUI.indentLevel++;

            //The variable VariableTransitionTime on the monsterScript should equal the setting the user picked for the toggle
            //The toggle should display the variable VariableTransitionTime
            monsterScript.VariableTransitionTime = EditorGUILayout.Toggle("Variable Transition Time?",
                                                    monsterScript.VariableTransitionTime);

            if (monsterScript.VariableTransitionTime)
            {
                EditorGUILayout.LabelField("Maximum - Minimum Time");
                EditorGUILayout.BeginHorizontal();
                //Since I manually created the label above, I am using an overload of the Float Field so no default label
                //is displayed immediately before it.
                monsterScript.TransitionTimeMax = EditorGUILayout.FloatField(monsterScript.TransitionTimeMax);
                monsterScript.TransitionTimeMin = EditorGUILayout.FloatField(monsterScript.TransitionTimeMin);
                EditorGUILayout.EndHorizontal();
            }
            else
            {
                EditorGUILayout.BeginHorizontal();
                EditorGUILayout.PrefixLabel("Transition Time:");
                //Again, I manually created a prefix label to display so I am overloading the float field so no
                //default label is shown
                monsterScript.TransitionTimeMax = EditorGUILayout.FloatField(monsterScript.TransitionTimeMax);
                EditorGUILayout.EndHorizontal();
            }

            //Reduce the indent level by one
            EditorGUI.indentLevel--;
        }

        EditorGUILayout.Space();


        spawnTypes = EditorGUILayout.Foldout(spawnTypes, "Spawn Settings");
        if (spawnTypes)
        {

            EditorGUI.indentLevel++;

            //Sync the object with the script
            serializedObject.Update();

            //Show the array field
            EditorGUILayout.PropertyField(serializedObject.FindProperty("possibleTypes"), true);

            EditorGUI.indentLevel--;
        }

        EditorGUILayout.Space();

        //Display the point value
        monsterScript.PointWorth = EditorGUILayout.IntField("Point Worth: ", monsterScript.PointWorth);

        //Apply all changes to the object
        serializedObject.ApplyModifiedProperties();
    }

}
```

Section Five: Custom Classes

Armed with the ability to create a custom inspector, things change when you want to start displaying custom data types and rearranging the way arrays are displayed. In this section, you will create a script that will allow a designer to change an items type: breakable, solid, damaging, healing or passable objects. Breakable objects will be worth a certain amount of points. Solid objects will have the option to move. Passable objects have no settings. Damaging objects can deal different types of damage. Healing objects can heal either the players health or the players shield. Healing objects can either be consumed when they are ran into, or consumed when interacted with.

Breakable	Points
Solid	Moving, Start point, End point
Damaging	Fire, Physical, Amount
Healing	Amount, Health, Shield, Contact, Interaction
Passable	

You will use a custom inspector to display the script called ObjectController. Then, you will use an editor drawer to display the different settings for each object.

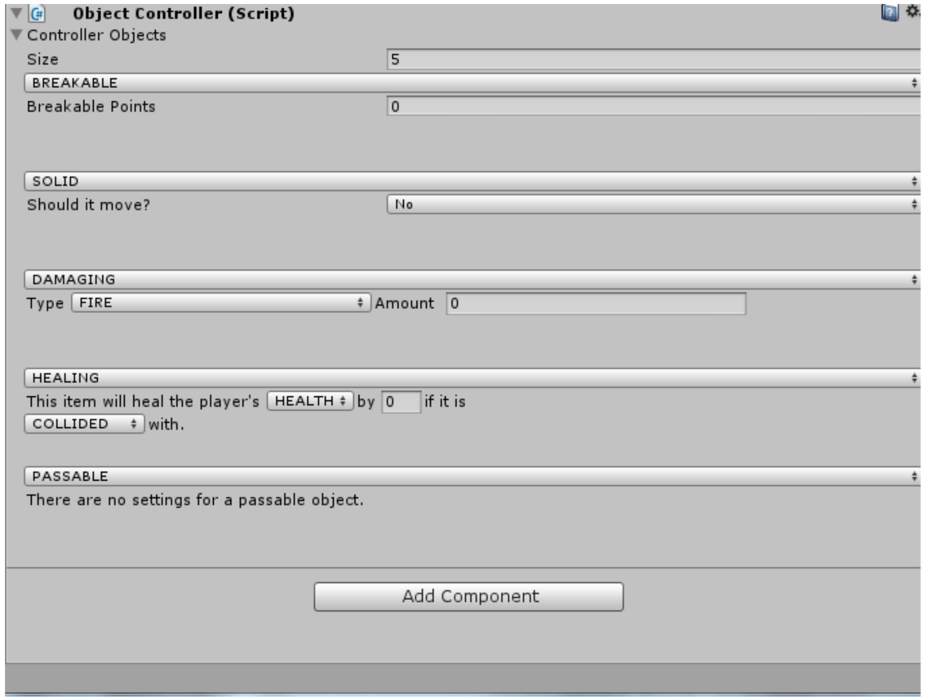
STEP ONE: SETUP

To start, we will create the empty scripts. You will need one script for the controller and the objects. You will also need an editor script for both the controller and the objects:

ObjectTypes.cs	Scripts>Common
ObjectTypesDrawer.cs	Editor
ObjectController.cs	Scripts>Common
ObjectControllerEditor.cs	Editor

The ObjectTypes will be a container that will store all of the variables for an object. This script is used solely to store information. The ObjectController will store all of the objects in the scene.

The final inspector will look like the following:



Object Controller (Script)
Controller Objects
Size 5
BREAKABLE
Breakable Points 0
SOLID
Should it move? No
DAMAGING
Type FIRE Amount 0
HEALING
This item will heal the player's HEALTH by 0 if it is COLLIDED with.
PASSABLE
There are no settings for a passable object.

STEP TWO: OBJECTTYPES.CS

This script will be a container. It will store information about all the different types of objects. You could split this script up into five different scripts, one for each type of object. However, for the purpose of this demonstration a single script will do.

You will need to store various information about each object type, including what type the object is. Using enums to control the designers input, the script would look like this:

```
using UnityEngine;
using System.Collections;

public enum ObjectType
{
    BREAKABLE,
    SOLID,
    DAMAGING,
    HEALING,
    PASSABLE
}

public enum DamageTypes
{
    FIRE,
    PHYSICAL
}

public enum HealingTypes
{
    HEALTH,
    SHIELD
}

public enum HealingPickups
{
    CONTACT,
    INTERACTION
}

[System.Serializable]
public class ObjectTypes : MonoBehaviour {

    public ObjectType type;

    public float breakablePoints;

    public bool solidMoving;
    public Vector3 solidStart;
    public Vector3 solidEnd;

    public DamageTypes damageType;
    public float damageAmount;

    public HealingTypes healingType;
    public HealingPickups healingPickupType;
    public float healingAmount;

}
```

We use the System.Serialzable attribute to inform unity that we need it to serialize this class for custom editing. If a class is not serialized, then unity cannot get information about the script when the script isn't running.

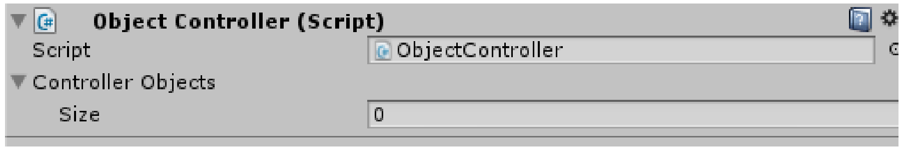
STEP THREE: OBJECTCONTROLLER.CS

This script will store all of the objects that the designer has created in an array. It is a very small script.

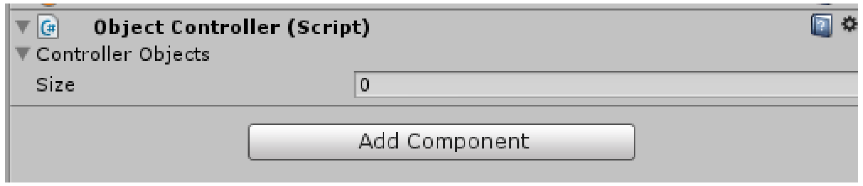
```
using UnityEngine;
using System.Collections;

public class ObjectController : MonoBehaviour {

    public ObjectTypes[] controllerObjects;
}
```

Note that we are going to have an array of ObjectTypes. If you were to have multiple scripts for each object, then you would want them all to share a superclass. This will allow you to have an array of SuperObejcts (since you can store children of a class inside an array of supers).

Next, you are going to tell Unity to display this class in a custom way, meaning you do not want unity to create the inspector for it automatically. Be default, the array of ObjectTypes will display like so:



For right now, all you are going to do is remove the option for the script, and actually display the array of objects:



SETUP

To start, follow the same procedure as the previous section for a custom editor.

```
using UnityEngine;
using System.Collections;
using UnityEditor;

[CustomEditor(typeof(ObjectController))]
public class ObjectControllerEditor : Editor {

    ObjectController controllerScript;

    void Awake()
    {
        controllerScript = (ObjectController)target;
    }

    public override void OnInspectorGUI()
    {
        serializedObject.Update();

        serializedObject.ApplyModifiedProperties();
    }
}
```

Remember: Because you are going to be working with an array you need to include the serializedObject.Update/serializedObject.ApplyModifiedProperties!

ACCESSING THE ARRAY

You are going to use a different method to display the array this time. Instead of letting unity display it for you, you will be manually displaying it.

To start, you need to get access to the array from the ObjectController script. Previously you used EditorGUILayout.PropertyField(serializedObject.FindProperty(string property)) to access the property. This time, instead of just accessing it for immediate editing, you want to store it. You store properties obtained this way in a SerializedProperty.

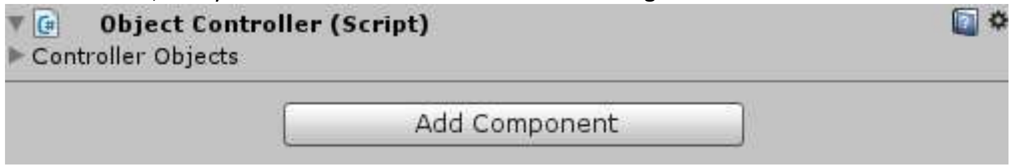
```
        SerializedProperty controller = serializedObject.FindProperty("controllerObjects");
```

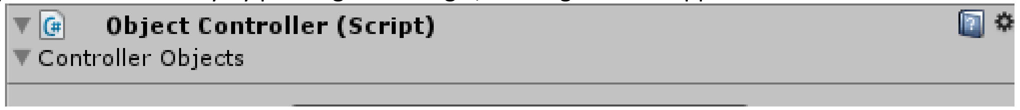
You now have access to the array in the script!

Next, you want to add a display for the property. This is as easy as using EditorGUILayout.PropertyField like you did before. Except this time, you do not include “true” for the children parameter and you will pass in the SerializedProperty instead of re-finding it.

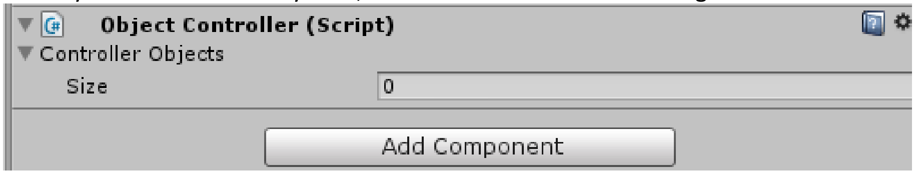
```
        EditorGUILayout.PropertyField(controller);
```

CHECKPOINT!

1. Apply the ObjectController script to the main camera of your scene.
2. Click the Main Camera, and your editor should look like the following:
   a. 

3. If you try to extend the array by pushing the triangle, nothing should happen!
   a. 

This is because when you created the property field this time, you did not pass in true for the children. This means that the children of the array (aka all items stored in the index) are not being shown!

4. Change your code to EditorGUILayout.PropertyField(controller, true);
5. Go back to unity and check the array now, it should look like the following
   a. 

6. Change your code back to the original (without the true parameter)

DISPLAYING THE ARRAY

Now that you have access to the array, you can display it.

In order to show a list, there are three things that you have to consider: the foldout (which you already created), its size, and its elements.

Showing the elements can be done by stepping through each element in the array and using another PropertyField to show it's default display:

```
        for (int i = 0; i < controller.arraySize; i++)
        {
            EditorGUILayout.PropertyField(controller.GetArrayElementAtIndex(i));
        }
```

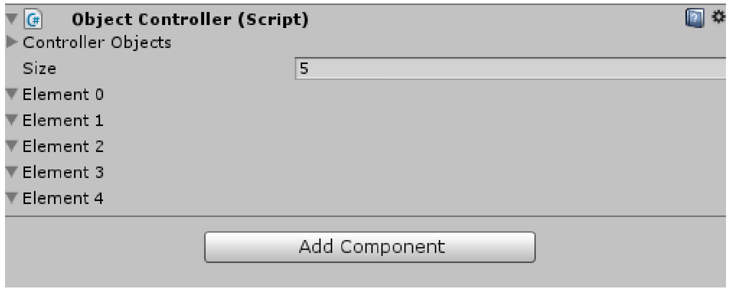
Testing it at this point will show nothing, as we have not added anything to this array at this point.

In order to add items to the array, you need access to the size of the array. Now, you may think that you can already access the size (controller.arraySize). However, that method does not return an editable version of the array size – it only returns an integer. Since you are working in a custom inspector, a simple integer is not enough. In order for the changes to reflect on the actual class, you need a SerializedProperty to communicate the change! In other words, if you use controller.arraySize the user will not be able to change the size of the array!

To get the SerializedProperty for the size of the array, you will use EditorGUILayout.PropertyField(controller.FindPropertyRelative("Array.size")). This will search only properties on the controller for a property called Array.size. With the property in hand, you can create a property field to display it.

```
        EditorGUILayout.PropertyField(controller);
        EditorGUILayout.PropertyField(controller.FindPropertyRelative("Array.size"));

        for (int i = 0; i < controller.arraySize; i++)
        {
            EditorGUILayout.PropertyField(controller.GetArrayElementAtIndex(i));
        }
```

Now the user can add/remove elements (although it won't show anything, since we have not yet added a way to display something with an ObjectType type).



The last thing you can do is allow the user to expand the array (show its elements) or hide the arrays elements.

This is a very easy property to get, as a Serialized array has a property named “isExpanded”.

Since you only want to display the elements of the array if the array is expanded, you can wrap the code used to display the array into an if statement.

If you wanted to get extra fancy you can even increase the indenting for the elements!

```
using UnityEngine;
using System.Collections;
using UnityEditor;

[CustomEditor(typeof(ObjectController))]
public class ObjectControllerEditor : Editor {

    ObjectController controllerScript;

    void Awake()
    {
        controllerScript = (ObjectController)target;
    }

    public override void OnInspectorGUI()
    {
        serializedObject.Update();

        SerializedProperty controller = serializedObject.FindProperty("controllerObjects");
```

```
            EditorGUILayout.PropertyField(controller);

            if (controller.isExpanded)
            {
                EditorGUILayout.PropertyField(controller.FindPropertyRelative("Array.size"));
                EditorGUI.indentLevel++;
                for (int i = 0; i < controller.arraySize; i++)
                {
                    EditorGUILayout.PropertyField(controller.GetArrayElementAtIndex(i));
                }
                EditorGUI.indentLevel--;
            }

            serializedObject.ApplyModifiedProperties();
        }
    }
```

STEP FOUR: DRAWER SETUP

The next thing you will do is create a custom editor for the ObjectType class. This will be similar to the custom editor created in the previous section. In this step, instead of creating an entire custom editor, you are going to create a drawer. A drawer is used to customize the look of a script, whereas an editor would be required if you were trying to adjust everything about the inspector.

The attribute for a Drawer is CustomPropertyDrawer.

```
[CustomPropertyDrawer(typeof(ObjectTypes))]
```

The class must also extend PropertyDrawer.

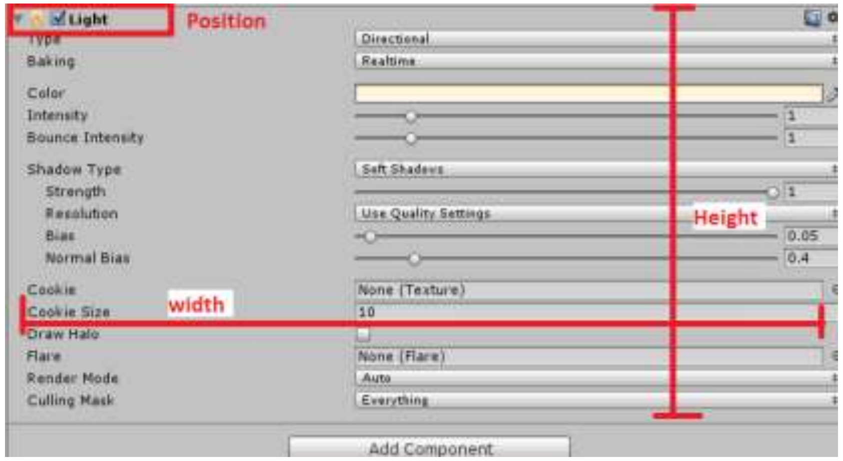
```
public class ObjectTypesDrawer : PropertyDrawer {
```

Start off by declaring the following variables:

```
    ObjectTypes thisObject;

    float extraHeight = 55f;

    int shouldSolidMove = 0;
```

Then, you will use the OnGUI function (not on InspectorGUI).

The inspector for a script has a position on the inspector panel. You need to make sure you are tracking this position so you can place items inside of it appropriately.



```
Type               Directional
Baking             Realtime
Color
Intensity          1
Bounce Intensity   1
Shadow Type        Soft Shadows
  Strength         1
  Resolution       Use Quality Settings
  Bias             0.05
  Normal Bias      0.4
Cookie             None (Texture)
Cookie Size        10
Draw Halo
Flare              None (Flare)
Render Mode        Auto
Culling Mask       Everything
```

In order to get the position, as well as some other information about the inspector and the script you are writing the inspector for, you will override the OnGUI function that you have been using up until now. You no longer want to use it in its default state, now you are going to want to play around in it.

```
public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)
```

Next, you need to inform unity that you are going to start displaying items that are Serialized.

```
EditorGUI.BeginProperty(position, label, property);
```

At the end of the function, you will tell unity that you are done displaying items.

```
EditorGUI.EndProperty();
```

Between BeginProperty and EndProperty is where you will program in the display of the items.

STEP FIVE: SERIALIZING

Before you can start editing the properties on the script, you need to get access to them. When creating a custom inspector you were able to tap into the script directly. However, with a drawer you have to extract the information. Remember: drawers are manually serialized, custom inspectors are automatically serialized (with the exception of arrays).

To manually extract a serialized property off of the ObjectType script, you use property.FindPropertyRelative (string property). You will then need to store this into a SerliazedProperty value.

```
        SerializedProperty objectType = property.FindPropertyRelative("type");

        SerializedProperty breakablePoints = property.FindPropertyRelative("breakablePoints");
        SerializedProperty solidMoving = property.FindPropertyRelative("solidMoving");
        SerializedProperty solidStart = property.FindPropertyRelative("solidStart");
        SerializedProperty solidEnd = property.FindPropertyRelative("solidEnd");

        SerializedProperty damageType = property.FindPropertyRelative("damageType");
        SerializedProperty damageAmount = property.FindPropertyRelative("damageAmount");

        SerializedProperty healingType = property.FindPropertyRelative("healingType");
        SerializedProperty healingPickupType = property.FindPropertyRelative("healingPickupType");
        SerializedProperty healingAmount = property.FindPropertyRelative("healingAmount");
```

STEP SIX: CHANGING THE OBJECT'S TYPE

Now you have a single instance of an ObjectType, with all of its values extracted and ready for editing.

The first thing you are going to want to display is what the object's current type is. Since you are not working with a normal inspector (you are working with a Drawer instead) you cannot use EditorGUILayout! Instead, you must use EditorGUI. This means you will have to manually assign the position, height, and width of EACH display.

The display for the object's type will be located at: position.x, position.y, position.width, 15f

This means that it will start at the x and y location given to us by the override, with the width given to us by the override, and a set height of 15f.

To set those numbers, you can store them in a Rect. A Rect is a class that stores a width, height, x, and y position.

```
Rect objectTypeDisplay = new Rect(position.x, position.y, position.width, 15f);
```

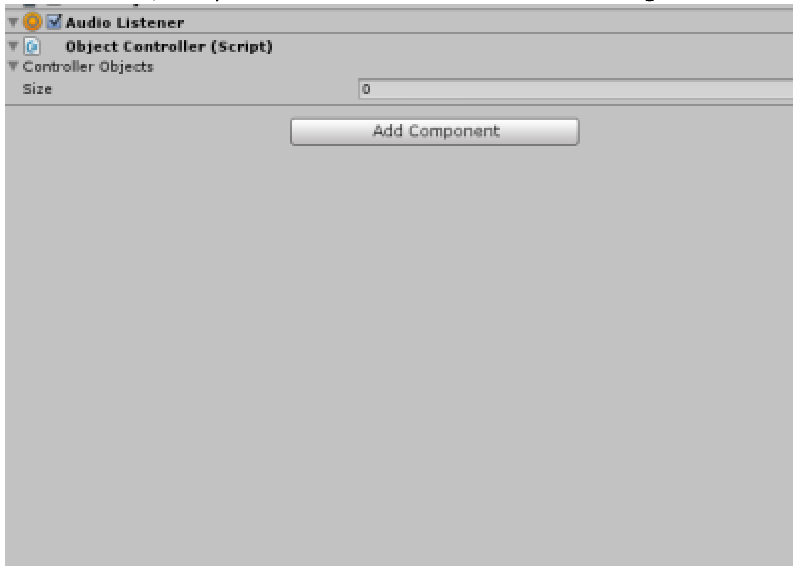
And then to display it on the editor you will use EditorGUI.PropertyField(Rect position, SerializedProperty property, GUIContent options).

Because you are using serialized properties, you must use the EditorGUI.PropertyField to display it. This will determine what type of interaction pattern to show, and show it.

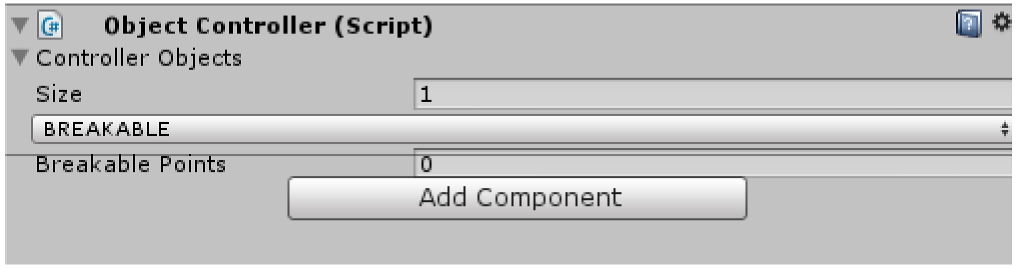
```
        EditorGUI.PropertyField(objectTypeDisplay, objectType, GUIContent.none);
```

CHECKPOINT!

1. Apply the ObjectController script to the main camera of your scene.
2. Click the Main Camera, and your editor should look like the following:

   a. 

3. Change Size to 1
4. Your editor should look like the following:

a. 
Object Controller (Script)
Controller Objects
Size 1
BREAKABLE
Breakable Points 0
Add Component

STEP SEVEN: ADDING MORE ROOM

As you can see, your code is drawing the properties exactly where you told them too. However, there isn't enough space!

Since you are manually adding objects, unity does not know to increase the space from where the class options end. You can force unity to increase this space.

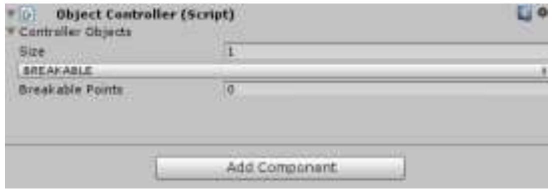
To start, create a new float variable called extraHeight, and assign it a default value of 55f.

Next, overwrite the unity method GetPropertyHeight (this gets the height of the propertyDrawer by default):

```
public override float GetPropertyHeight(SerializedProperty property, GUIContent label)
```

Inside the method, call the super but add your extra height value:

```
return base.GetPropertyHeight(property, label) + extraHeight;
```


Object Controller (Script)
Controller Objects
Size 1
BREAKABLE
Breakable Points 0
Add Component

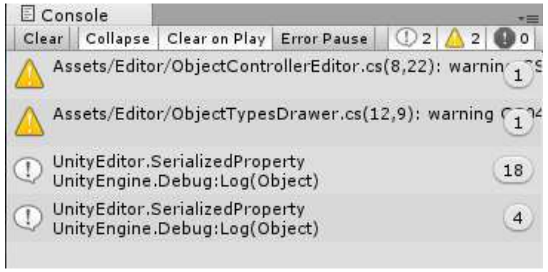
STEP EIGHT: UNDERLYING STRUCTURE

Now you need to start adding specific options for each type of object. However, before you do that you need to know if the user has actually changed the object type!

Unity automatically created the change when the user changes the property field you created. But the editor Drawer is storing that information as a SerializedProperty, NOT as the ObjectType like it is in the actual script.

Test this for yourself. After the PropertyField, add a debug statement that debugs out the objectType. On the script ObjectTypes (which is what we are editing) it will BREAKABLE, HEALING, DAMAING, ect. However, on the Drawer it is just a SerializedProperty.

```
        SerializedProperty healingType = property.FindPropertyRelative("healingType");
        SerializedProperty healingPickupType = property.FindPropertyRelative("healingPickupType");
        SerializedProperty healingAmount = property.FindPropertyRelative("healingAmount");

        EditorGUI.PropertyField(objectTypeDisplay, objectType, GUIContent.none);

        Debug.Log(objectType);
```



```
Assets/Editor/ObjectControllerEditor.cs(8,22): warnin     1
Assets/Editor/ObjectTypesDrawer.cs(12,9): warning C       1
UnityEditor.SerializedProperty                           18
UnityEngine.Debug:Log(Object)
UnityEditor.SerializedProperty                            4
UnityEngine.Debug:Log(Object)
```

Even though it is a SerializedProperty, it does contain some of the underlying information. You can access this underlying information through the serialized property. For example, the objectType variable on the ObjectTypes script is of the type ObjectType (an enum). Therefore, you can access the underlying enum information of the SerializedProperty by using the method enumValueIndex.

```
Debug.Log(objectType.enumValueIndex);
```

This will get you the index number (BREAKABLE = 0, SOLID = 1, DAMAING = 2, HEALING = 3, PASSABLE = 4 if you copied the enum from this section).

Now that you can capture the index, you can use a switch statement to perform different actions based on which state the enum is in. To make the code more readable, you can caste the enum from numeric values to enum values using the (ObjectType) caste:

```
switch ((ObjectType)objectType.enumValueIndex)
        {
            case ObjectType.BREAKABLE:
                break;
            case ObjectType.DAMAGING:
                break;
            case ObjectType.HEALING:
                break;
            case ObjectType.PASSABLE:
                break;
            case ObjectType.SOLID:
                break;
        }
```

CHECKPOINT!

It is important to remember what is happening when you are creating these editors.

The first thing you did (as a user) was add the ObjectController to the main camera. The object controller contains an array of ObjectType, so the editor displays the default array notation (aka the size). When you make the array larger than zero, it then shows the EditorDrawer for EACH ObjectType. Meaning if you have the array set to size 5, it will run the EditorDrawer five times, once for each ObjectType in the array.

Because you are using a single script for all FIVE types, you need to account for each type in the ObjectType script. If you were to have separate scripts for each object type, then you would not need the switch/case statement.

STEP NINE: OPTIONS!

Now you can start adding options for the designers to alter.

BREAKABLE

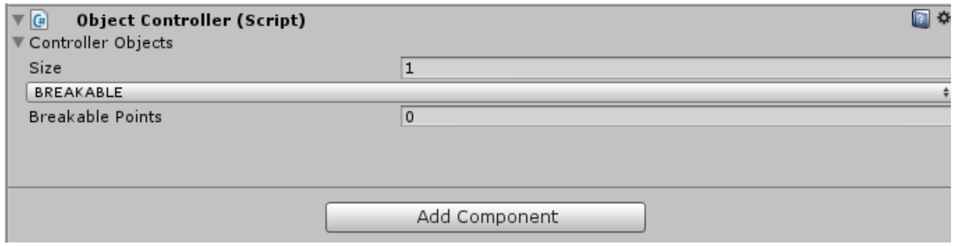
You can start with the breakable object. We know that a breakable object only had one option: How many points it is worth.

Again, to add a new option for the designer you need to use the EditorGUI.PropertyField. Unity will automatically decide what type of field it is and display the appropriate interaction pattern.

You also need to dictate where to place the option. Since you are placing the dropbox at position.x, position.y, position.width, 15f you are going to want to display this option at LEAST 15 pixels below it. It is typical to include a buffer as well to add a little space between options. Therefore, you could use position.x, position.y + 17f, position.width, 15f.

This will place it at the same location left/right, the same location down plus 17 (15 because of the height of the previous property plus 2 buffer), the same width as the inspector panel, and 15 height.

```
switch ((ObjectType)objectType.enumValueIndex)
        {
            case ObjectType.BREAKABLE:
                Rect breakableRect = new Rect(position.x, position.y+17, position.width, 15f);
                EditorGUI.PropertyField(breakableRect, breakablePoints);
                break;
            case ObjectType.DAMAGING:
                break;
            case ObjectType.HEALING:
                break;
            case ObjectType.PASSABLE:
                break;
            case ObjectType.SOLID:
                break;
        }
```

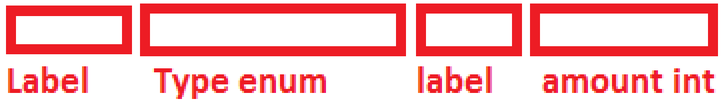


Controller Objects
Size 1
BREAKABLE
Breakable Points 0

DAMAGING

A damaging object has two different types: Fire, or Physical. It also has an amount option that accounts for how much damage will happen.

You will display these two controls next to each other, and manually include their label (so unity doesn't automatically create the label for you). It will look something like the image below:



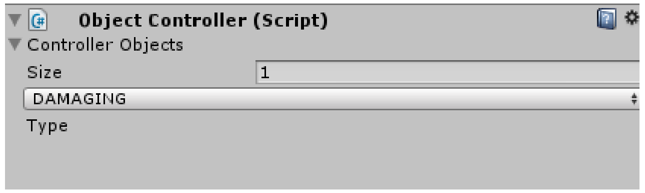
To start, declare a float to store the offset on the X. This way you don't have to manually keep track of far over your placement is.

Next, set up the Rect for the damage type label. This will be the label that says "Type". Since this is the first control going on the line, the offset should be 0, aka the position.x of the drawer.

You may have to play with the width of the field until you get it just "right" (meaning it displays the whole label and no more!). Once you find the "right" number, then you will add that to the offset for the next control.

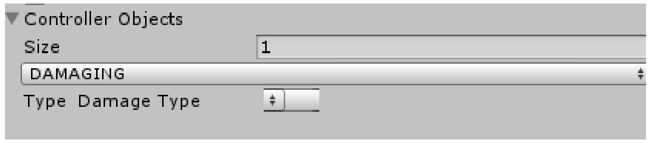
Lastly, you can display the label using EditorGUI.LabelField(Rect position, string label).

```
switch ((ObjectType)objectType.enumValueIndex)
{
    case ObjectType.BREAKABLE:
        Rect breakableRect = new Rect(position.x, position.y+17, position.width, 15f);
        EditorGUI.PropertyField(breakableRect, breakablePoints);
        break;
    case ObjectType.DAMAGING:
        float offset = position.x;

        Rect damageTypeLabelRect = new Rect(offset, position.y + 17, 35f, 17f);
        offset += 35;
        EditorGUI.LabelField(damageTypeLabelRect, "Type");
        break;
```


Object Controller (Script)
Controller Objects
Size 1
DAMAGING
Type

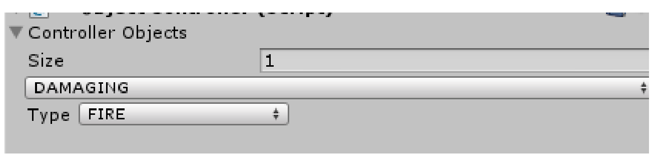
Next, you will add the enum that corresponds to the damage type (DamageTypes enum). You can display this enum the same way you are displaying the ObjectType enum. However, when you create the rect for it you need to make sure you are setting its x position to the proper offset. Also, since you are going to want this field to scale as the user changes the size of the inspector you do not want to hard-code in a number for the width. Instead, you will want to use a formula. You may have to play around with it to determine the proper size.

```
        case ObjectType.DAMAGING:
            float offset = position.x;

            Rect damageTypeLabelRect = new Rect(offset, position.y + 17, 35f, 17f);
            offset += 35;
            EditorGUI.LabelField(damageTypeLabelRect, "Type");

            Rect damageTypeRect = new Rect(offset, position.y + 17, position.width / 3, 17f);
            offset += position.width / 3;
            EditorGUI.PropertyField(damageTypeRect, damageType);
```



So far, you never told unity not to generate the label for this control. Therefore, the label it generated (Damage Type) is taking up some of the space that you dedicated to the control. In order to remove the auto-generated label, you will use GUIContent.none for the GUIContent parameter.

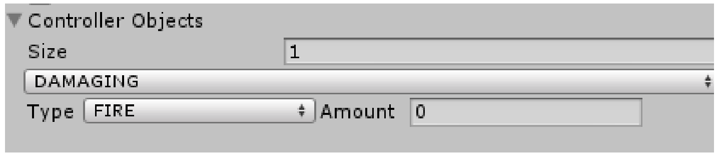
```
EditorGUI.PropertyField(damageTypeRect, damageType, GUIContent.none);
```



Now add the label and the field for the amount.

```
case ObjectType.DAMAGING:
    float offset = position.x;

    Rect damageTypeLabelRect = new Rect(offset, position.y + 17, 35f, 17f);
    offset += 35;
    EditorGUI.LabelField(damageTypeLabelRect, "Type");

    Rect damageTypeRect = new Rect(offset, position.y + 17, position.width / 3, 17f);
    offset += position.width / 3;
    EditorGUI.PropertyField(damageTypeRect, damageType, GUIContent.none);

    Rect damageAmountLabelRect = new Rect(offset, position.y + 17, 55f, 17f);
    offset += 55;
    EditorGUI.LabelField(damageAmountLabelRect, "Amount");

    Rect damageAmountRect = new Rect(offset, position.y + 17, position.width / 3, 17f);
    EditorGUI.PropertyField(damageAmountRect, damageAmount, GUIContent.none);
```

```
            break;
```



Controller Objects
Size 1
DAMAGING
Type FIRE Amount 0

SOLID

The next control you will work on is the solid control. The solid control is a little more complex, as it can have multiple states.

The first thing you need is to provide a dropdown for the user to decide if the solid object should move or not. This dropdown will also need a position.

```
            Rect shouldMove = new Rect(position.x, position.y + 17, position.width, 17f);
```

The dropdown will pull from the bool option named solidMoving, so TRUE or FALSE. However, it is not intuitive to the user to have TRUE/FALSE as dropdown options. Instead, using "Yes" or "No" would be more familiar. Therefore, you will need to change the options shown to the user.

In order to do this, you need to convert the bool value stored in the solidMoving Serialized Property from a bool to an int. (note this is using a conditional operator)

```
            int index = solidMoving.boolValue ? 0 : 1;
```

If you do not know what a conditional operator is, you can reference it here https://msdn.microsoft.com/en-us/library/ty67wk28.aspx

Next, create a string array that will correspond a string to the index. In this case, 0 = "Yes" and 1 = "No".

```
string[] options = new string[] { "Yes", "No" };
```

Next, you will display the options as a dropdown (known as a Popup to the Unity API). Since you are NOT displaying a SerializedField at this point (instead you will be displaying the custom index created), you will need to use the same methods as before to reassign the value.

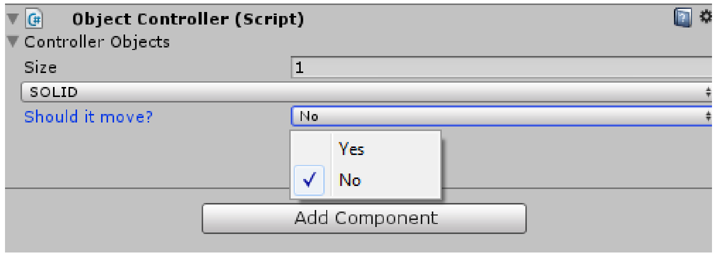
```
index = EditorGUI.Popup(shouldMove,"Should it move?", index, options);
```

The EditorGUI.Popup(Rect position, string Label, int index, string[] options) will create a dropdown box with the current selection being index. It will then show the option in the string array at the location of index.

For example, if we set index 0 then it will look inside of options[0] and display the string it finds.

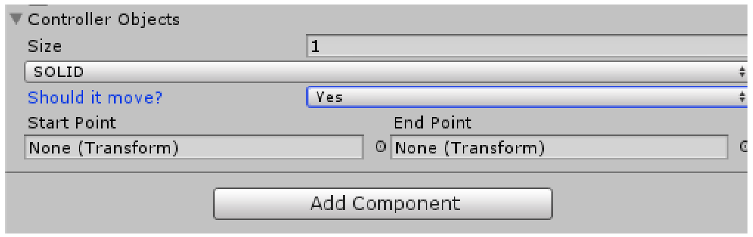
At this point this is only editing the local version of this option. You need to reconvert this from an int to a bool and then store it inside of the SerializedProperty that links to the script you are editing.

```
solidMoving.boolValue = (index > 0) ? false : true;
```

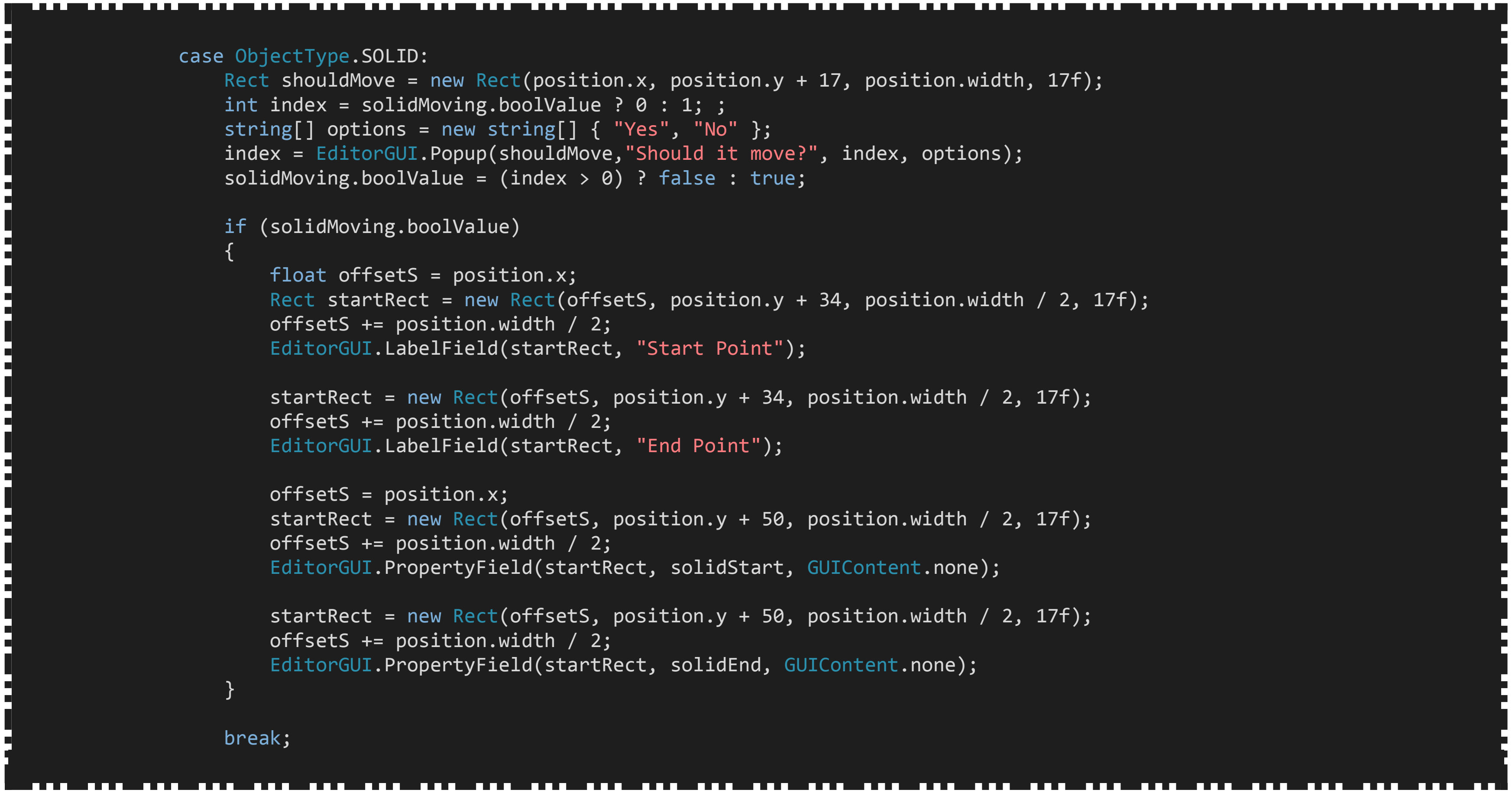

Object Controller (Script)
Controller Objects
Size 1
SOLID
Should it move? No

If the answer is No, then you are done. However, if the answer is yes then you need to allow the designer to specify the start and end points for the movement of the object.

Adding the additional options is done the same way the previous options you have done in this section.



```
Controller Objects
  Size                    1
  SOLID
  Should it move?         Yes
  Start Point                          End Point
  None (Transform)                     None (Transform)

                  Add Component
```



```csharp
case ObjectType.SOLID:
    Rect shouldMove = new Rect(position.x, position.y + 17, position.width, 17f);
    int index = solidMoving.boolValue ? 0 : 1; ;
    string[] options = new string[] { "Yes", "No" };
    index = EditorGUI.Popup(shouldMove,"Should it move?", index, options);
    solidMoving.boolValue = (index > 0) ? false : true;

    if (solidMoving.boolValue)
    {
        float offsetS = position.x;
        Rect startRect = new Rect(offsetS, position.y + 34, position.width / 2, 17f);
        offsetS += position.width / 2;
        EditorGUI.LabelField(startRect, "Start Point");

        startRect = new Rect(offsetS, position.y + 34, position.width / 2, 17f);
        offsetS += position.width / 2;
        EditorGUI.LabelField(startRect, "End Point");

        offsetS = position.x;
        startRect = new Rect(offsetS, position.y + 50, position.width / 2, 17f);
        offsetS += position.width / 2;
        EditorGUI.PropertyField(startRect, solidStart, GUIContent.none);

        startRect = new Rect(offsetS, position.y + 50, position.width / 2, 17f);
        offsetS += position.width / 2;
        EditorGUI.PropertyField(startRect, solidEnd, GUIContent.none);
    }

    break;
```

WRAP UP

You now have the tools to:

- Create a custom display for enums and custom classes
- Create property drawers for custom classes that can be used inside custom inspectors

COMPLETE CODE

ObjectTypes.cs

```
using UnityEngine;
using System.Collections;

public enum ObjectType
{
    BREAKABLE,
    SOLID,
    DAMAGING,
    HEALING,
    PASSABLE
}
public enum DamageTypes
{
    FIRE,
    PHYSICAL
}
public enum HealingTypes
{
    HEALTH,
    SHIELD
}
public enum HealingPickups
```

```
{
    COLLIDED,
    INTERACTED
}

[System.Serializable]
public class ObjectTypes{

    public ObjectType type;

    public float breakablePoints;

    public bool solidMoving;
    public Transform solidStart;
    public Transform solidEnd;

    public DamageTypes damageType;
    public float damageAmount;

    public HealingTypes healingType;
    public HealingPickups healingPickupType;
    public float healingAmount;

}
```

ObjectControllerEditor.cs

```
using UnityEngine;
using System.Collections;
using UnityEditor;

[CustomEditor(typeof(ObjectController))]
public class ObjectControllerEditor : Editor {

    //Store a reference to the script that is being edited
    ObjectController controllerScript;
    void Awake()
    {
        //Capture a reference to the script that is being edited
        controllerScript = (ObjectController)target;
    }

    public override void OnInspectorGUI()
    {
        //Update the serializedObject for array alterations
        serializedObject.Update();

        //Get an editable of the ObjectType[] controllerObjects on the script
        SerializedProperty controller = serializedObject.FindProperty("controllerObjects");

        //Display the foldout for the ObjectType array
        EditorGUILayout.PropertyField(controller);

        //Determine if the foldout is set to display the elements of the array
        if (controller.isExpanded)
        {
            //Get an editable copy of the array size for the user, and then display it
            EditorGUILayout.PropertyField(controller.FindPropertyRelative("Array.size"));

            //Increase the indent level
            EditorGUI.indentLevel++;

            //Go through each element of the array and display it
            for (int i = 0; i < controller.arraySize; i++)
            {
                EditorGUILayout.PropertyField(controller.GetArrayElementAtIndex(i));
            }

            //Decreate the indent level
            EditorGUI.indentLevel--;
        }

        //Apply any changes to the script
        serializedObject.ApplyModifiedProperties();
    }
```

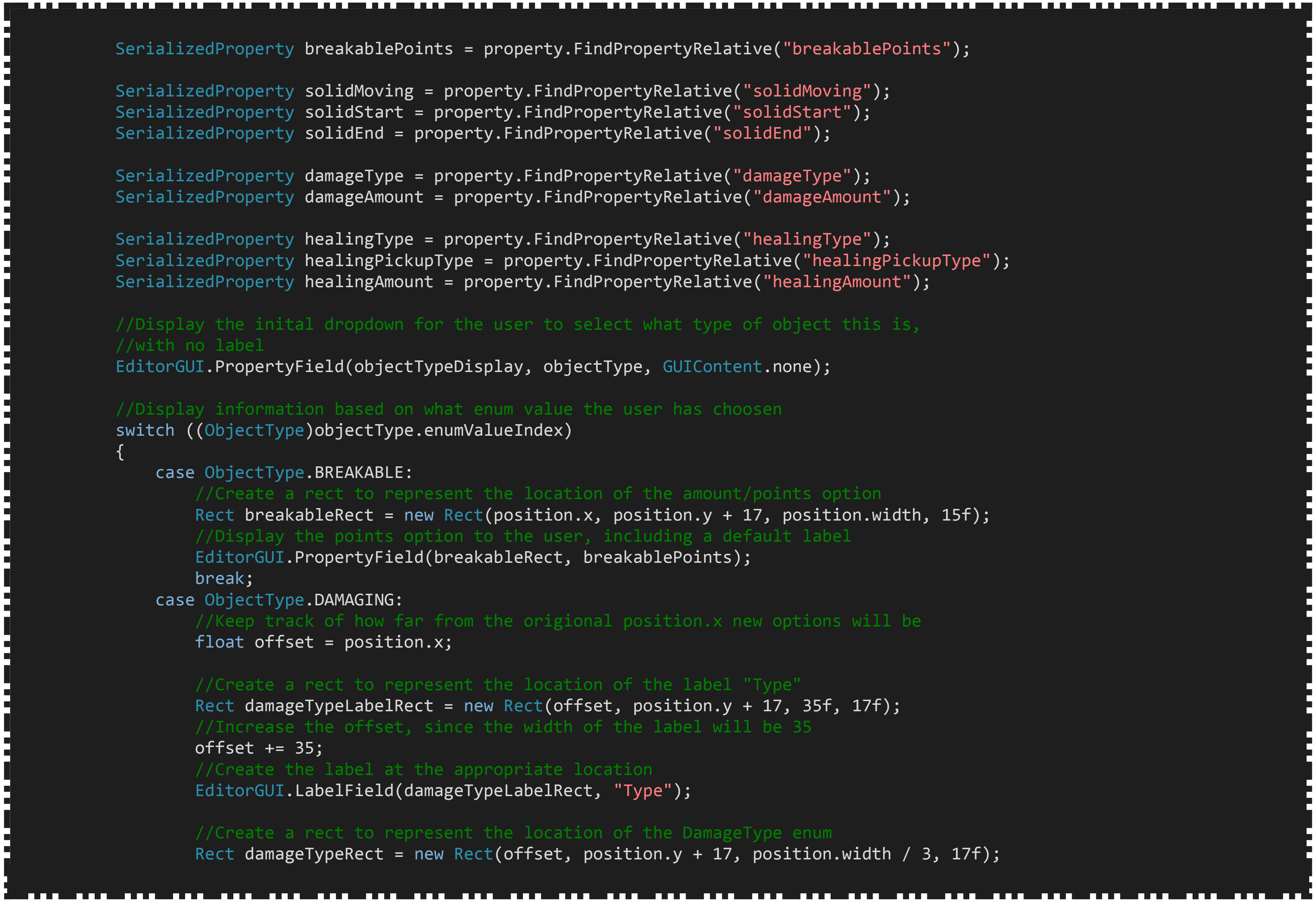
```
}
```

ObjectController.cs

```
using UnityEngine;
using System.Collections;

public class ObjectController : MonoBehaviour {

    public ObjectTypes[] controllerObjects;
}
```

ObjectTypesDrawer.cs

```
using UnityEngine;
using System.Collections;
using UnityEditor;

[CustomPropertyDrawer(typeof(ObjectTypes))]
public class ObjectTypesDrawer : PropertyDrawer
{
    //Create extra spacing below the inspector for this object
    float extraHeight = 55f;

    public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)
    {
        //Create a rect that represents the enum dropdown position for the object type
        Rect objectTypeDisplay = new Rect(position.x, position.y, position.width, 15f);

        //Inform the editor that a new portion of the editor is going to be started
        EditorGUI.BeginProperty(position, label, property);

        //Gather editable references to all of the variables needed off the script that is being edited
        SerializedProperty objectType = property.FindPropertyRelative("type");
```



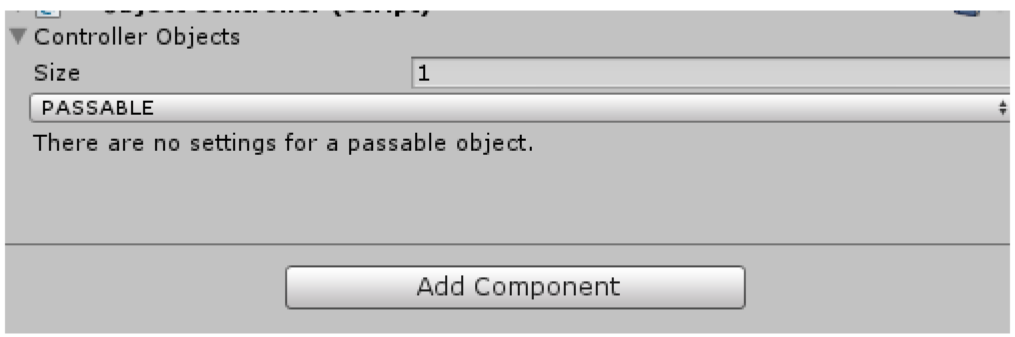
```
SerializedProperty breakablePoints = property.FindPropertyRelative("breakablePoints");

SerializedProperty solidMoving = property.FindPropertyRelative("solidMoving");
SerializedProperty solidStart = property.FindPropertyRelative("solidStart");
SerializedProperty solidEnd = property.FindPropertyRelative("solidEnd");

SerializedProperty damageType = property.FindPropertyRelative("damageType");
SerializedProperty damageAmount = property.FindPropertyRelative("damageAmount");

SerializedProperty healingType = property.FindPropertyRelative("healingType");
SerializedProperty healingPickupType = property.FindPropertyRelative("healingPickupType");
SerializedProperty healingAmount = property.FindPropertyRelative("healingAmount");

//Display the inital dropdown for the user to select what type of object this is,
//with no label
EditorGUI.PropertyField(objectTypeDisplay, objectType, GUIContent.none);

//Display information based on what enum value the user has choosen
switch ((ObjectType)objectType.enumValueIndex)
{
    case ObjectType.BREAKABLE:
        //Create a rect to represent the location of the amount/points option
        Rect breakableRect = new Rect(position.x, position.y + 17, position.width, 15f);
        //Display the points option to the user, including a default label
        EditorGUI.PropertyField(breakableRect, breakablePoints);
        break;
    case ObjectType.DAMAGING:
        //Keep track of how far from the origional position.x new options will be
        float offset = position.x;

        //Create a rect to represent the location of the label "Type"
        Rect damageTypeLabelRect = new Rect(offset, position.y + 17, 35f, 17f);
        //Increase the offset, since the width of the label will be 35
        offset += 35;
        //Create the label at the appropriate location
        EditorGUI.LabelField(damageTypeLabelRect, "Type");

        //Create a rect to represent the location of the DamageType enum
        Rect damageTypeRect = new Rect(offset, position.y + 17, position.width / 3, 17f);
        //Increase the offset
        offset += position.width / 3;
        //Create the dropdown for DamageType at the appropriate location
        //without a default label
        EditorGUI.PropertyField(damageTypeRect, damageType, GUIContent.none);

        //Create a rect to represent the location of the label "Amount"
        Rect damageAmountLabelRect = new Rect(offset, position.y + 17, 55f, 17f);
        //Increase the offset, since the width of the label will be 55
        offset += 55;
        //Create the label at the appropriate location
        EditorGUI.LabelField(damageAmountLabelRect, "Amount");

        //Create a rect to represent the location of damageAmount int value
        Rect damageAmountRect = new Rect(offset, position.y + 17, position.width / 3, 17f);
        //Create the input field for the damageAmount at the appropriate location
        //without a default label
        EditorGUI.PropertyField(damageAmountRect, damageAmount, GUIContent.none);

        break;
    case ObjectType.HEALING:
        break;
    case ObjectType.PASSABLE:
        break;
    case ObjectType.SOLID:
        //Create a rect to represent the location that the dropdown will appear
        Rect shouldMove = new Rect(position.x, position.y + 17, position.width, 17f);
        //Convert the solidMoving bool to an integer
        int index = solidMoving.boolValue ? 0 : 1; ;
        //Set up what the dropdown options will be
        string[] options = new string[] { "Yes", "No" };
        //Display a dropdown at the appropriate location, with a set label,
        //displaying the value of index as the corrisponding string
        index = EditorGUI.Popup(shouldMove, "Should it move?", index, options);
        //Convert the value of index to a bool and store it in the SerializedProperty
        solidMoving.boolValue = (index > 0) ? false : true;

        //If solidMoving is true
        if (solidMoving.boolValue)
```

```
                {
                    //Store the offset on the x axis
                    float offsetS = position.x;
                    //Create a rect to represent the location of the "Start Point" label
                    Rect startRect = new Rect(offsetS, position.y + 34, position.width / 2, 17f);
                    //Add to the offset the width of the label
                    offsetS += position.width / 2;
                    //Create the label
                    EditorGUI.LabelField(startRect, "Start Point");

                    //Edit a rect to represent the location of the "End Point" label
                    startRect = new Rect(offsetS, position.y + 34, position.width / 2, 17f);
                    //Add to the offset the width of the label
                    offsetS += position.width / 2;
                    //Create the label
                    EditorGUI.LabelField(startRect, "End Point");

                    //Reset the offset since the next options will be one line down
                    offsetS = position.x;
                    //Edit the rect to represent the location of the solidStart option
                    startRect = new Rect(offsetS, position.y + 50, position.width / 2, 17f);
                    //Add to the offset the width of the option
                    offsetS += position.width / 2;
                    //Display the option with no label
                    EditorGUI.PropertyField(startRect, solidStart, GUIContent.none);

                    //Edit the rect to represent the location of the solidEnd option
                    startRect = new Rect(offsetS, position.y + 50, position.width / 2, 17f);
                    //Display the option with no label
                    EditorGUI.PropertyField(startRect, solidEnd, GUIContent.none);
                }

                break;
        }

        //Inform the editor this is the end of the custom information
        EditorGUI.EndProperty();

    }
    //Gets the height of the property
    public override float GetPropertyHeight(SerializedProperty property, GUIContent label)
    {
        //Increases the height of the property by extraHeight
        return base.GetPropertyHeight(property, label) + extraHeight;
    }
}
```

On Your Own Part 1

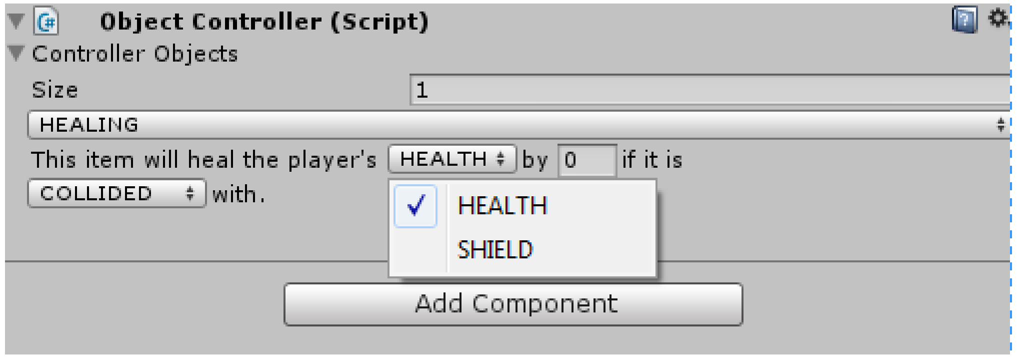
Finish the last two types of object in this lab: Passable and Healing. They should look like the following:

PASSABLE

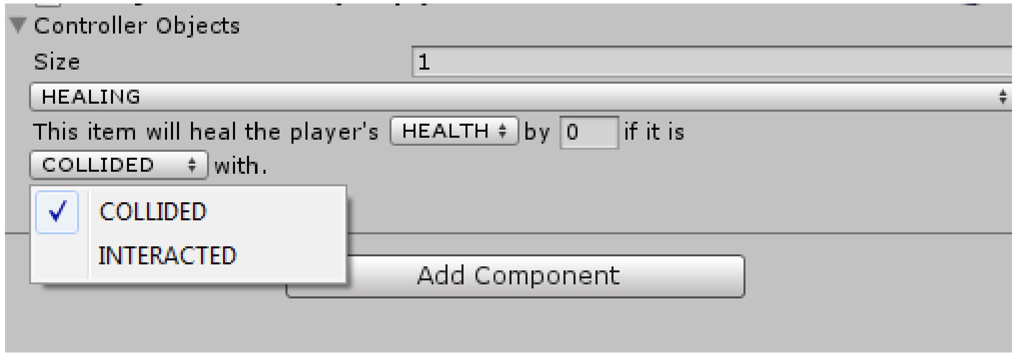


```
▼ Controller Objects
  Size                1
  PASSABLE
  There are no settings for a passable object.
```

HEALING



Controller Objects
Size 1
HEALING
This item will heal the player's HEALTH by 0 if it is
COLLIDED with.



Controller Objects
Size 1
HEALING
This item will heal the player's HEALTH by 0 if it is
COLLIDED with.

On Your Own Part 2

Create a tool that will allow a designer to add enemies and weapons to the game. The definition of an enemy and a weapon as a user object is displayed below. Since you are not implementing what enemies and weapons can do, the user object will not contain the actions section

Enemy	Name (text)	Weapon	Classification (2 handed, 1 handed, Off-hand)
	Faction (Aggressive, Neutral, Passive)		Type (Sword, Shield, Relic, Hammer, Scyth)
	Attack Type (Melee, Ranged)		Damage (0-100)

Attack Damage (Physical, Spell)	Upgradable (Yes, No)
Spell Types (Fire, Frost, Arcane)	
Weapon Type (2 Handed, Dual Wield)	
Health	
Mana	
Armour Type (Cloth, Leather, Mail, Plate)	

The following is the constraints:

ENEMIES:

If the Attack Damage is Physical, do not show:

- Spell type
- Mana
- Cloth Armour Type

If the weapon type is 2 Handed do not show:

- Mail Armour Type

Armour Type is NOT exclusive, an enemy can have multiple armour types

Weapon Type is NOT excluse, an enemy can have multiple weapon types

WEAPONS:

If a weapon is 2 handed, do not show;

- Shield Type
- Relic Type

If a weapon is 1 handed, do not show:

- Scyth
- Shield Type
- Relic Type

If a weapon is off-hand do not show:

- Sword
- Hammer
- Scyth

Hammers are not upgradable

Once you complete the tools, create a simple script to read the designer inputted values and display them to console as proof of completion.