User Manual

Contents

1. Introduction

Thank you for purchasing Soda. A lot of thought, discussion and testing went into the development of this product. May it improve everything you achieve with Unity.

Integrating Soda into your project will profoundly change the way you develop within Unity. For this reason, this manual goes beyond describing package features and how-tos. When you use Soda, you have to be aware of **how** to use it in which situation. This even includes cases where using Soda is not a good idea at all. You have to be aware of the reasoning behind how Soda is supposed to be used. To get a good understanding of this reasoning, it is recommended that you read through this introduction in its entirety.

1.1. What is Dependency Injection?

"Dependency Injection" is a term that is very connected to what Soda is.

When researching it, you might come across some very sophisticated material that is often very in-depth, and it's hard to see the use or how it's connected to development within Unity. Here is a very simple explanation of what Dependency Injection is.

Let's take a very simple, standard Unity component: The `Light` component. When you add one to your GameObject, Unity's inspector window will display a list of its properties, allowing you to customize the behavior of the component. For example, you can set a light color. This seems very simple and intuitive, so we don't think about this a lot. However, as obligatory as this is, it gives us a very substantial feature. If we want a red light and a blue light, we don't have to write a `RedLight` component and a `BlueLight` component. We just have a `Light` component and we can set its color.

Dependency Injection takes this one step further. Think of a simple example in a game: a button that opens a door. The button needs to know which door in the scene it has to open, as there might be multiple doors. So instead of a color property, the button has a property where a door object is a valid value. A **reference** to the door is somehow given to the button.

In the Unity Editor, this is done by dragging a GameObject into a property shown by the inspector window. Thus, when you have properly worked with the Unity Editor before, you have used Dependency Injection already, perhaps without knowing.

This is Dependency Injection, and it's great – imagine showing the button which door to open without this feature. However, what Unity offers us in terms of Dependency Injection is limited in some cases. **These cases are where Soda comes in.**

1.2. What are ScriptableObjects?

Soda is a library that mostly consists of ScriptableObject classes. To understand how to use Soda, it's important to understand the basics of ScriptableObjects.

One of the most important classes in the UnityEngine library is `UnityEngine.Object`. This class is the base class for GameObjects and components and contains the code needed to have objects serialized as objects that can be referenced. For example, a door in your scene is stored in the scene file with a unique identifier. When you drag and drop the door into a button in the same scene, that button will be stored in the scene file with the additional information which door to open. Unity uses that identifier for that.

A third class that inherits from `UnityEngine.Object` is `ScriptableObject`. Instances of ScriptableObject classes also are part of your project, like GameObject or components. But, among other things, they can be created within your asset folder. This means that you have an asset, like a texture or a sound, that can have any kind of meaning. And just like the other assets, they have unique identifiers, so you can assign them to properties in the inspector.

ScriptableObjects are used for many things in Unity projects. From things like inventory systems and combat skills to very technical uses, where ScriptableObjects are used to reduce data redundancy, they have proven to be the solution to many problems people had in Unity - until they learned about ScriptableObjects.

1.3. What is Soda?

Soda, short for **S**criptable**O**bject **D**ependency **A**rchitecture, is inspired by Ryan Hipple's talk at the Unite Austin 2017. Feel free to watch it for a good introduction. However, this manual will cover many of the same aspects and more.

As proposed by Ryan Hipple in his talk, Soda implements multiple ScriptableObject classes that each represent very basic programming concepts. When you create a Soda ScriptableObject, it will not have any semantic meaning in itself – as opposed to ScriptableObjects that represent inventory items or combat skills. You create a Soda ScriptableObject instead of creating a piece of code; a variable or a class. How that object is used within your project defines its meaning.

You don't use Soda to achieve a specific goal, you use Soda to implement your solution in a cleaner, more robust and more modular way.

The following chapters of this manual list the classes Soda offers you. Each type of Soda ScriptableObject can help you implement a high-quality solution for your next task.

Soda comes with full source code to allow you to study its mechanics and to improve it or to adjust it to your project. If you improve anything or have a feature request, feel free to check the section 8. Support for contact options.

Important: When referencing Soda types in your code, you have to add this import statement:

```
using ThirteenPixels.Soda;
```

2. SodaEvents

SodaEvent is a more or less standard event class with some added functionality for debugging. SodaEvents are used to **monitor changes** in all Soda ScriptableObject classes. Use them instead of checking your ScriptableObjects' values in every frame.

Add SodaEvent objects offer two methods:

- `AddResponse(response)`
- `RemoveResponse(response)`

By using these methods, you can add and remove methods to a collection or responses that will be invoked when the event is raised.

The response parameter is either of type `Action` or `Action<T>` depending on the owner of the SodaEvent.

The following is an example for using SodaEvents.

```
[SerializeField]
private GlobalInt number;

private void OnEnable()
{
    number.onChange.AddResponse(UpdateNumber);
}

private void OnDisable()
{
    number.onChange.RemoveResponse(UpdateNumber);
}

private void UpdateNumber(int newValue)
{
    Debug.Log("New Value: " + newValue);
}
```

This code uses an `GlobalInt` (see 3.1. GlobalVariables), a ScriptableObject that represents an integer number. In the `OnEnable` method, this example component adds a response to that `GlobalInt`, and in `OnDisable`, this response is being removed again.

It's recommended to just write a private method and pass its name to `AddResponse` and `RemoveResponse`. Lambda expressions would technically work, but they would mean redundancy and when using them, you'd risk having two differing actions in the two method calls.

Note that there's no inherent mechanic in SodaEvents that would automatically clear their reponse lists. If one of your classes adds a response to a SodaEvent, it has to remove that response again on its own. It is recommended to use `OnEnable` and `OnDisable` for this, as shown in the example.

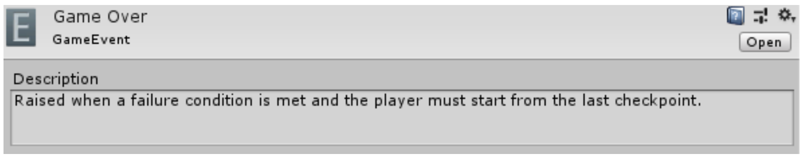
3. Soda ScriptableObject classes

This chapter contains introductions and explanations for Soda's core classes – ScriptableObject classes that allow you to fully embrace Unity's way of Dependency Injection.

Each family of Soda ScriptableObjects serves a different kind of purpose. Understanding when to use which, and even when to not use any of them at all, is important in order to make Soda a capable ally when developing with Unity.

There's two general things of importance before we start:

First, each Soda ScriptableObject has a description field.



Raised when a failure condition is met and the player must start from the last checkpoint.

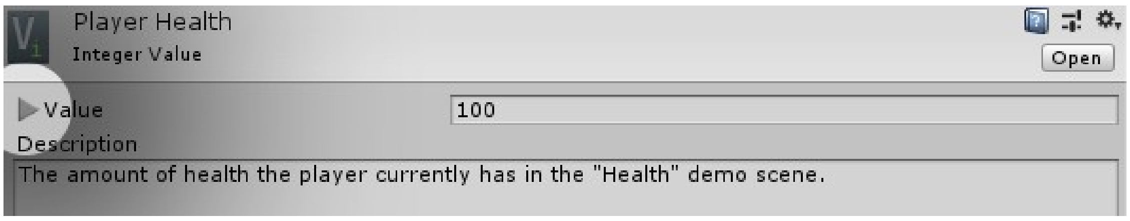
Use it to describe some semantics for your object so all team members understand what the object is about. While GameObjects can often be described well through their name and their context, this isn't as easy with ScriptableObjects, so description texts can be very helpful.

Second, Soda introduces a new concept to ScriptableObjects.

You probably noticed how Unity thankfully doesn't save any changes made to a scene after exiting playmode. After all, you want the game to reset to its initial state no matter what happened during your test run. ScriptableObjects, as they're Assets and thus not part of a scene, don't have this mechanic. Just like changes to material assets persist, even when they're applied during play mode.

Soda's ScriptableObjects work in ways that are incompatible with this fact. Your game will very often change the state of Soda ScriptableObjects, but the original values must be restored after play mode is exited.

To ensure this, some fields in the inspector feature a small, gray "play" icon:



Value: 100

The amount of health the player currently has in the "Health" demo scene.

This icon turns blue while in playmode, indicating that changes to the property's value will **not** persist after exiting play mode.

3.1. GlobalVariables

The most basic category of ScriptableObjects provided by Soda is **GlobalVariables**.

A GlobalVariable object represents a single **variable**. The type of the GlobalVariable determines the type of the value it represents.

As their name suggests, GlobalVariable objects act like **global variables** (or, as they're known in C#, **static variables**), but come with almost none of their disadvantages for code quality. They are not actually in the global scope, as other objects cannot find them by themselves (as compared to static fields and methods). However, using drag and drop in the Unity Editor, any object in your project can use them freely, without being involved with any other object that uses them.

Accordingly, **consider using a GlobalVariable object whenever you are considering a static variable for the purpose of other objects accessing it**.

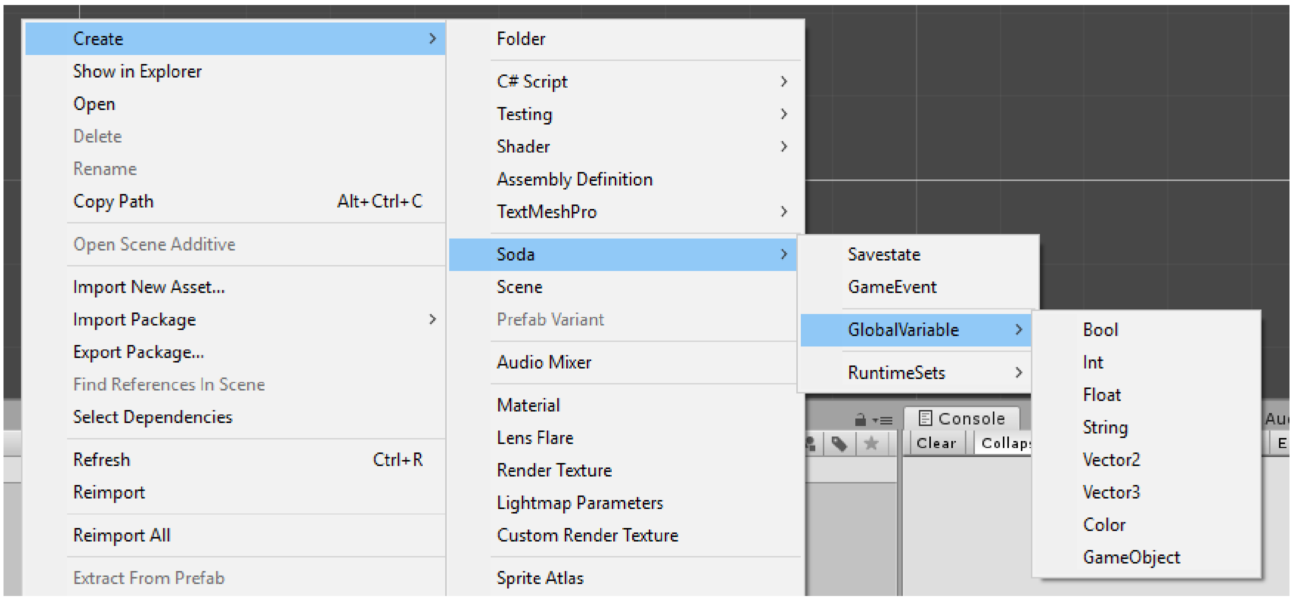
For example, the player character's health component might have a static variable that stores the player's health points. A health display script can access this variable and display its value. However, this means semantically binding the health display script to the player, as it is statically linked to that one variable in the player health script. If you want to display health points for a second player or a boss monster, you'd have to write another script that reads another static variable's value.

To avoid this, create a GlobalVariable object to represent that value instead of the static variable.

3.1.1 Creating a GlobalVariable

You can add your own GlobalVariable type (see 5.1. Creating a new GlobalVariable type), but Soda is shipped with these GlobalVariable types: `bool`, `int`, `float`, `string`, `Vector2`, `Vector3`, `Color` and `GameObject`.

You create them in your Project View via the create menu, under "Soda/GlobalVariable":



GlobalVariable classes are named `GlobalT`, with `T` being the type of the variable.

Let's assume that in this example, health points are implemented as an `int`. This would mean we're using a `GlobalInt` object to store the player's health points.

Once you created a `GlobalInt`, select it to see its inspector. You'll note that you can set an initial value to store and write a description text that briefly describes the semantic meaning of the represented value.

3.1.2. Changing a GlobalVariable's value

Next, change your player health class to use the GlobalVariable to store the health points value.

```
[SerializeField]
private GlobalInt healthPoints;
```

You change the GlobalVariable's value with the `value` property.

```
public void ApplyDamage(int amount)
{
    healthPoints.value -= amount;
}

public void Kill()
{
    healthPoints.value = 0;
}
```

Once you implemented this, feel free to test the GlobalVariable by calling one of these methods. Note the following:

1. You can see any changes to the value immediately in the GlobalVariable's inspector when selected. This means that you can test your component without having to create an additional component that reacts to the changes it makes.
2. You can change the value by hand during play mode for debugging purposes.
3. When you exit play mode, the value will return to its initial value.

This concludes the introduction to GlobalVariables: How to create them and how to change their value. For properly **reading** a GlobalVariable's value, continue to the next chapter.

3.1.3. Reading a GlobalVariable's value

You can easily read a GlobalVariable's current value by reading its `value` property.

However, for many cases, this is a bad choice. If you think of a health bar displaying the player's health points, it's not a good idea to have it check the "Player HP" object's value every frame in case it has updated. Instead, use the `onChange` SodaEvent (see 2. SodaEvents) to assign a response to changes to the value.

It is recommended to **always** use the `onChange` event for monitoring value changes. Use the `value` property only when you want to check the value in situations where it didn't necessarily just change. For example, imagine the player's current score being saved in a `GlobalInt`. Your game over code could use the `value` property to check that object's value when the game over happens.

3.1.4. GlobalGameObject

A special GlobalVariable type is the `GlobalGameObject`. As the name suggests, it doesn't store a numeric value or a string, but a reference to a GameObject. This GlobalVariable type can be used to register one specific GameObject so other objects can see it and work with it – much like a pseudo-singleton that is often seen in Unity projects.

A GameObject can easily register itself to a `GlobalGameObject` with the **GlobalGameObjectRegister** component. Simply drop it onto the GameObject and assign the `GlobalGameObject` you want to register the GameObject to.

This alone isn't worth a lot, as you usually don't work with a GameObject, but its components. Using a `GlobalGameObject` means that you'd have to use `GetComponent` every time you want to work with the referenced GameOject's components. However, there's a better solution for that as well: The `GlobalGameObjectWithComponentCacheBase` class. You can create your own subclass for it and specify one or more components in it. These components are required to be on the GameObject in order to be accepted as a valid value for the `GlobalGameObject`. The component(s) will then be cached in the `GlobalGameObject` for direct use.

Here's a code example of such a class, representing a GameObject with a `Light` component:

```
[CreateAssetMenu(menuName = "Soda/GlobalVariable/GameObject/Global Light")]
public class GlobalLight : GlobalGameObjectWithComponentCacheBase<Light>
{
    protected override bool TryCreateComponentCache(GameObject gameObject,
                                                    out Light componentCache)
    {
        componentCache = gameObject.GetComponent<Light>();
        return componentCache != null;
    }
}
```

The `TryCreateComponentCache` method must return `true` only when the passed GameObject's components match your requirements – in this case, this means that the GameObject has a `Light` component. In addition, a reference to that component is set to the `componentCache` parameter. That reference will be stored as long as this particular GameObject stays represented by this `GlobalGameObject`.

The cached component(s) is/are available though the `componentCache` property:

```
[SerializeField]
private GlobalLight targetLight;

public void MakeLightDarkAndGreen()
{
    targetLight.componentCache.intensity = 0.1f;
    targetLight.componentCache.color = Color.green;
}
```

Note that there are are no `ScopedGameObject` variants with component caching support.

`GlobalGameObject`s with component caching are supposed to be used whenever the client object has to know semantics about the object it's working with. For example, the example code above knows that it references a GameObject that has a `light` component. This couples this object and the target object much more closely together, so it should only be done when there's good reasons for that.

If you just want to monitor or change one generic value (like the player character's HP, use a regular GlobalVariable object instead. If you want to fire a generic event (like the player character's death) without teaching the object causing event about all the consequences the event has, prese refer to 3.2. GameEvents.

3.1.5. Debugging GlobalVariables

GlobalVariables come with a special inspector that displays a list of all objects that subscribed to its `onChange` event. With this list, monitor or select and object that monitors this GlobalVariable's value for changes.

See GlobalVariables in action!

Check the Soda Demo folder and open the "Health" demo.

Open the "Dancer" demo for an example of a GlobalGameObject with Component Caching.

3.2. GameEvents

A **GameEvent** is another of Soda's ScriptableObject classes. A GameEvent that you create in your assets represents a single event – a thing that can be triggered, and that has reactions that happen when the event is triggered.

For example, the death of the player character can be considered an event. It happens when its health points reach zero, and when that happens, a "Game Over" screen appears.

3.2.1. About event systems

The idea behind **any** event system is that it is not a clean solution to have the piece of code that manages the player's health points to know about the "Game Over" screen in order to trigger it. A fullscreen message and a character's health points just aren't semantically related. So an event system introduces a layer between what triggers an event and what happens as a result.

When the player character's health points reach zero, the health code triggers the "Player Death" event and doesn't care what happens next. The event checks its list of listeners, objects that announced interest in the event happening beforehand, and informs them that the event is now occuring. One of the objects in the list is the "Game Over" screen, which makes itself visible as a reaction to the occuring event. The fact that the event was triggered by some health code in another part of the program is unknown to this object. As a result, **what triggers an event and who reacts to that event are seperate things that don't know of each other**.

3.2.2. Creating GameEvents

Soda offers GameEvents as **injectable events** that need no additional code to set up. You can pass a GameEvent to any object in your game and make it trigger the event at any time. An easy way is a GameEvent field, and a call to the `Raise` method at the right moment:

```
[SerializeField]
private GameEvent deathEvent;

public void Kill()
{
    deathEvent.Raise();
}
```

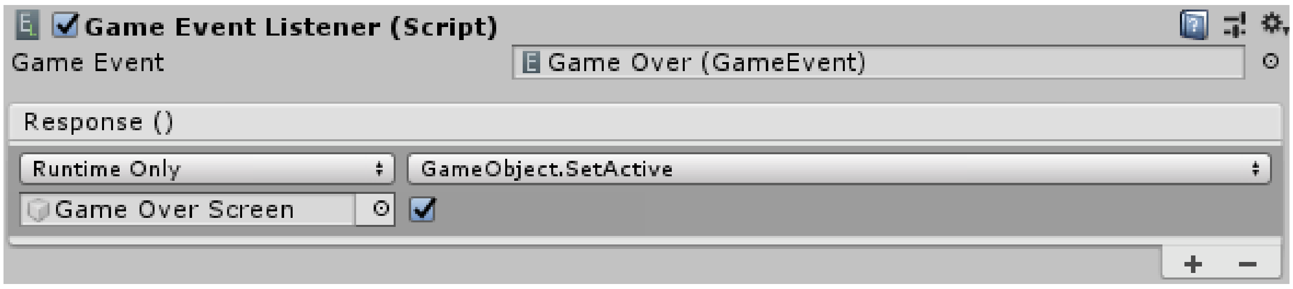
However, please note that in most cases, a UnityEvent offers a much more modular way to trigger events (see 6.1. Combination with UnityEvents).

Now, you can create a GameEvent ScriptableObject asset, pretty much like you create a GlobalVariable object: In your create menu, under "Soda/GameEvent".

Once it's created, and you perhaps added a description that highlights the semantics of the event, drag it into the field of the component.

3.2.3. Reacting to GameEvents

To set up some GameObject to react to a GameEvent happening, add a **GameEventListener** component to that GameObject. The component consists of two properties:



Game Event Listener (Script)
Game Event: Game Over (GameEvent)
Response ()
Runtime Only | GameObject.SetActive
Game Over Screen | ✓

1. A "Game Event" property where you drag a GameEvent from your assets to.
2. A "Response" UnityEvent where you set up a response, just like with a UI.Button.

Whenever the assigned GameEvent is raised, the response you set up is invoked, as long as the GameEventListener component still exists and is enabled.

In addition to the GameEventListener component, a GameEvent offers you a UnityEvent when you select it in your assets. You can use this for reactions outside of the loaded scene, for example involving other Soda ScriptableObjects. Or in other words: You can trigger another event through an event.

3.2.4. Parameterized GameEvents

Soda 1.2.4 added parameterized GameEvents – GameEvents that are capable of passing a single value with each raise. This addition came quite late as there were concerns about their usefulness despite users asking for it rather often.

This section is going to explain the potential issues with them in addition on how to use them properly.

Potential Issues

Soda's core idea is that it should help you build cleaner architecture that allows you to write more modular, robust and testable code. The package as well as this manual have been carefully crafted to make it as easy as possible to avoid any pitfalls on your way. Soda is intentionally limited in some parts, in order to offer as few ways to build something less clean than possible.

Parameterized GameEvents come very intuitive to people, as GameEvents are used to convey momentary information ("something happened"), and that information should often contain a "how" in addition to a "what". For example, a "Game Over" event might contain a bit of data that specifies the reason for the Game Over – that the player died or the time ran out.

However, they come as a potential threat to Soda's core idea as it's possible to build problematic systems with them. The issue lies with the definition of semantics.

Consider this case: A GameEvent is raised whenever the player takes damage. Since it makes sense in the project, the GameEvent has an `int` parameter that represents an amount of health points related to that damage event. That `int` value may represent either the amount of damage taken, or it may represent the amount of health the player has left after taking the damage. One of these options might be obviously right to you, but it's not something inherently clear when you receive the event's `int` parameter value. So you have to define that the parameter of this GameEvent is representing the amounf of damage taken, and all your classes that are supposed to use the event have to adhere to that decision.

The issue is that there is no way to enforce this in your code, or even to support you in writing it correctly. When you write a C# method call in your code, your IDE will show you a popup that tells you the meaning of each parameter – or you can open the documentation to see it. What a parameter is representing is *defined at compile time*. However, with a `GameEventInt`, what the parameter means and how it is to be interpreted is unclear until the very moment you drag a specific GameEvent into your component. As a result, there is simply no way to test whether your code understands the meaning of the parameter value in the way that the event sender intended. This means that bugs in your code that come from a mismatch between how the sender of a GameEvent and its receiver interpret the parameter are easy to make and incredibly hard to track down. This is even worse considering that there is a potentially infinite amount of classes reacting to and raising any given GameEvent.

With all that being said, parameterized GameEvents offer unique advantages over the alternative way (using a GlobalGameObject or RuntimeSet) that will be explained later:

Response diversity

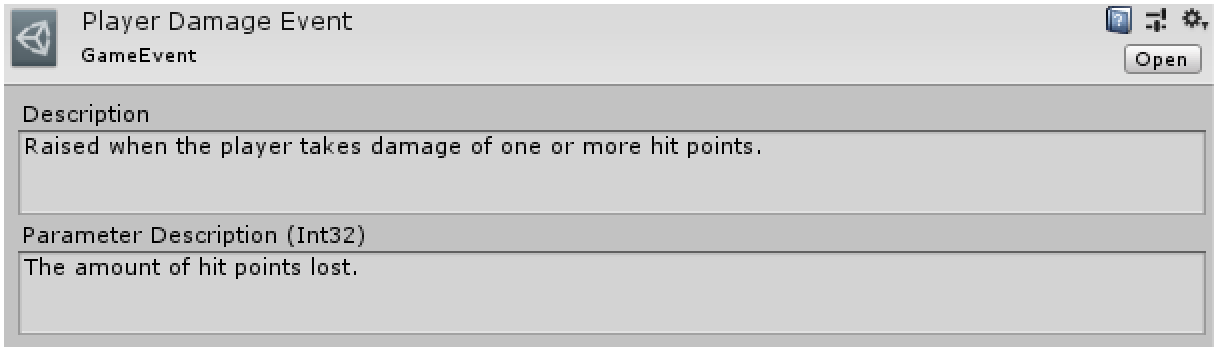
Response Diversity means that you can write any amount of unique classes that react properly to any GameEvent. When you use a RuntimeSet instead of a GameEvent to propagate an event to a group of targets, these targets have to be of the same type, and you call the same method on all of them. Even though it's possible to mitigate this with aggressive component-based design, it's way simpler to have a GameEvent triggering any response that any class adds to the list.

Anonymity

When using a GameEvent as a proxy, sender and receiver of a GameEvent can be written entirely independently. While this is basically the cause for the mentioned issue, it's also an advantage, allowing you to write independent and modular classes.

What you can do

Parameterized GameEvents offer an additional description field in which you can describe the meaning of the parameter:



Player Damage Event
GameEvent

Description
Raised when the player takes damage of one or more hit points.

Parameter Description (Int32)
The amount of hit points lost.

Although this is just a text field in the inspector, it can be used to stipulate the semantics of the parameter of this specific GameEvent. Objects that raise this GameEvent or respond to it should handle the parameter value accordingly.

How to use parameterized GameEvents

After this lengthy word of warning, here's how to use parameterized GameEvents.

1. Create a class that extends `GameEventBase<T>`.

```
namespace ThirteenPixels.Soda
{
    using UnityEngine;
    using UnityEngine.Events;

    [CreateAssetMenu(menuName = "Soda/GameEvent/Int" order = 250)]
    public class GameEventInt : GameEventBase<int>
    {
        [System.Serializable]
        private class IntEvent : UnityEvent<int> { }
        [SerializeField]
        private IntEvent _onRaiseGlobally = default;
        protected override UnityEvent<int> onRaiseGlobally =>
_onRaiseGlobally;
    }
}
```

2. Create one or more instances of this class using the Assets/Create menu.

3. Reference and raise the GameEvent as usual, except with a parameter:

```
[SerializeField]
private GameEventInt damageEvent;

public void ApplyDamage(int amount)
{
    damageEvent.Raise(amount);
}
```

4. Respond to the event using the `onRaiseWithParameter` property:

```
[SerializeField]
private GameEventInt damageEvent;

private void OnEnable()
{
    damageEvent.onRaiseWithParameter.AddResponse(OnTakeDamage);
}

private void OnDisable()
{
    damageEvent.onRaiseWithParameter.RemoveResponse(OnTakeDamage);
}

private void OnTakeDamage(int amount)
{
    Debug.Log("I took " + amount + "damage!");
}
```

5. To respond to the event while ignoring the parameter, continue using the `onRaise` property. You can use this to write modular components that respond to any GameEvent type. As an example, the included GameEventListener component supports dragging parameterized GameEvents into it.

Alternatives

Whenever you are thinking about using a parameterized GameEvent, please consider one of the following alternatives. As mentioned, parameterized GameEvents have their unique advantages, but if you need neither response diversity not anonymity, the following ideas might be the safer and cleaner way for the issue at hand.

Alternative A

In cases where the value in question has a reason to permanently exist, create a GlobalVariable and store the value in it. As an example: You have a "Game Over" GameEvent, and you'd like the object(s) responding to it to know the final score of the game. Instead of passing it along with the "Game Over" GameEvent, simply have a `GlobalInt` that stores the score. The objects that process the final score can react to the GameEvent by reading the `GlobalInt`'s `value` property.

Alternative B

In situations where the value you'd like to transmit is more temporary in nature, alternative A isn't very elegant. Imagine a situation where you have a component that displays messages in the game's UI. Any class should be able to send a string to this component to have it displayed. In this case, there is only one kind of response and the sender knows what kind of object they're sending their string to. Instead of using a `GameEventString`, try this:

1. Create a class deriving from `GlobalGameObjectWithComponentCacheBase` to create a GlobalVariable type that references the component that would have received the event (see 3.1.4. GlobalGameObject). In the mentioned example, this would be the component responsible to displaying the messages.

2. Create a GlobalVariable type of that type and make sure the component is registered to it at runtime (possibly through use of a GlobalGameObjectRegister component).

3. Reference the GlobalVariable object in your sender component and use the receiver component's public methods to achieve your goal.

To showcase this solution, here's some example code, implementing the message display example. This would be the component displaying the messages:

```
public class MessageDisplay : MonoBehaviour
{
    public void DisplayMessage(string message)
    {
        // Somehow append the message to the list of displayed messages
    }
}
```

With the GlobalVariable type creation wizard, create a `GlobalMessageDisplay` class.

Then, create an instance of this class and make sure a `MessageDisplay` is registered to it. You can use the `GlobalGameObjectRegister` component to do so.

Finally, use code like this to send a message to the registered component:

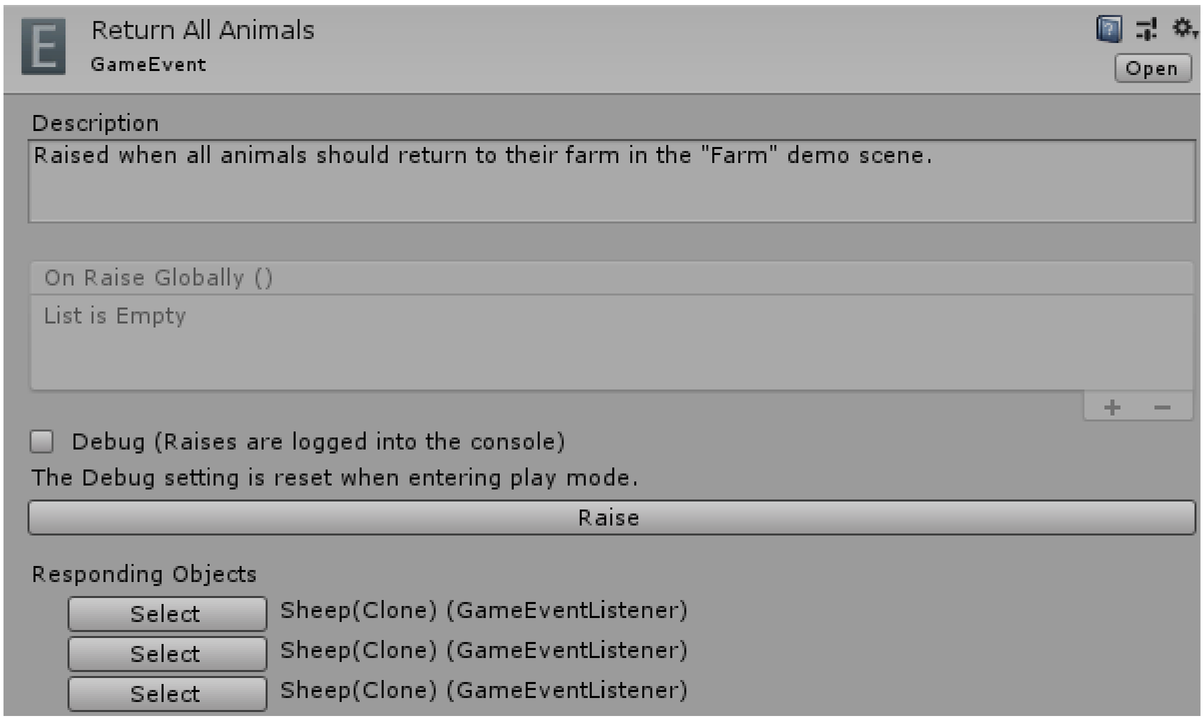
```
[SerializeField]
private GlobalMessageDisplay messageDisplay;

private void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        messageDisplay.value.DisplayMessage("Space was pressed!");
    }
}
```

To see an example of this solution, check the "Dancer" demo scene.

If you have a similar case, but with (potentially) multiple receivers, RuntimeSets offer the same functionality.

3.2.5. Debugging GameEvents


Return All Animals
GameEvent

Description
Raised when all animals should return to their farm in the "Farm" demo scene.

On Raise Globally ()
List is Empty

Debug (Raises are logged into the console)
The Debug setting is reset when entering play mode.

Responding Objects
Sheep(Clone) (GameEventListener)
Sheep(Clone) (GameEventListener)
Sheep(Clone) (GameEventListener)

When you select a GameEvent in your assets, the inspector you see gives you some debugging options.

This starts with a **Raise button** that allows you to raise an event manually to see whether the objects reacting to the event work. With this, you don't need to implement any triggering objects to test the reacting objects.

Likewise, the **checkbox labelled "Debug"** allows you to test objects triggering the event without having to add objects reacting to it. As long as this checkbox is checked, whenever anything calls the `Raise` method of the event, a `Debug.Log` is triggered, so you know whether the event trigger works.

And finally, during play mode, the inspector shows you a list of all objects that are currently registered to respond to the event.

See GameEvents in action!

Check the Soda Demo folder and open the "Farm" demo.

3.3. RuntimeSets

A **RuntimeSet** is another ScriptableObject class. It is used to store references to a number of objects, usually with a shared trait. For example, you could store all enemies in a RuntimeSet and access them all for a specific purpose. As an example, imagine a stealth game. When the player character is spotted, all enemies are alerted, so they start looking for the player character in order to attack it.

RuntimeSets are an extremely powerful tool with options for customization and optimization. Two types of easy-to-use RuntimeSets are available by default:

- `RuntimeSetGameObject`
- `RuntimeSetTransform`

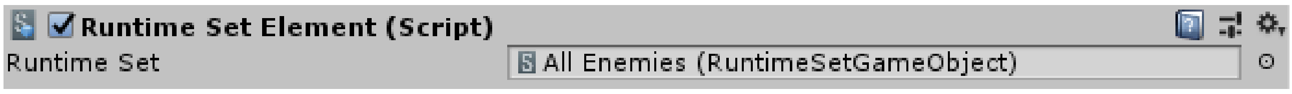
As the names suggest, the former stores references to GameObjects, while the other directly references the Transform components of the GameObjects added to the RuntimeSet. More RuntimeSet types can be created, which allows you to maintain RuntimeSets referencing very specific components directly – even multiple components at once. See 5.2. Creating a new RuntimeSet type for more about this.

You can create a new RuntimeSet object as you'd expect: In the create menu, under "Soda/RuntimeSet".

3.3.1. Adding objects to a RuntimeSet

A RuntimeSet is a passive object that just offers a way to add and remove objects to and from it. Other objects can then access the added objects. The RuntimeSet object does nothing by itself.

The simple way to add an object to a RuntimeSet is to add a **RuntimeSetElement** component to the object you want to add, and to reference the RuntimeSet you want to add it to in the component. Then, simply drag the RuntimeSet into the component's property:


Runtime Set Element (Script)
Runtime Set: All Enemies (RuntimeSetGameObject)

This component will automatically register the GameObject it is on into the referenced RuntimeSet as long as it is enabled. Disabling (that includes destroying) the RuntimeSetElement or its GameObject will automatically unregister it from the RuntimeSet.

However, you are not bound to the RuntimeSetElement component. Any component can add a GameObject to any RuntimeSetElement, which is useful if the defining trait that all objects in the RuntimeSet whould have is dynamic. For example, you can create an area and register all objects in that area to a specific RuntimeSet. As an example, think about a finish line, where all cars that crossed it (or have not crossed it yet!) are in a RuntimeSet. Once a race time limit is reached, all cars in the RuntimeSet could, perhaps, explode.

To manually add a GameObject to a RuntimeSet, use the `Add` method:

```
myRuntimeSet.Add(gameObject);
```

To manually remove the GameObjet, use the `Remove` method.

3.3.2. Working with objects in a RuntimeSet

RuntimeSets offer you a few options to access their data.

You can check whether an object is element of a RuntimeSet, by using the `Contains` method.

The readonly property `elementCount` returns the amount of objects in the RuntimeSet. You can use that to determine things like whether the last enemy has been defeated.

The `ForEach` method is similar to Linq's `ForEach`. You call it and pass an `Action<T>` with `T` being the element type of the RuntimeSet. The action will be invoked for every item in the RuntimeSet. An example:

```
[SerializedField]
private RuntimeSetGameObject allEnemies;
```

```
// Destroy all enemies
allEnemies.ForEach(enemy => Destroy(enemy));
```

You can also use this to acquire data:

```
[SerializedField]
private RuntimeSetTransform allThings;

// Check the amount of things that are positioned lower than y = 0
var thingsUnderground = 0;
allThings.ForEach(thing =>
{
  if (thing.position.y < 0)
  {
    thingsUnderground++;
  }
});
Debug.Log(thingsUnderground + " things are underground!");
```

If you prefer, you can use the `GetAllElements` method to get a readonly collection of all elements in the RuntimeSet.

```
var allThingsElements = allThings.GetAllElements();
foreach (var thingTransform in allThingsElements)
{
  thingTransform.position += Vector3.up;
}
```

3.3.3. RuntimeSet element count monitoring

RuntimeSets feature a SodaEvent that allows you to get informed about elements getting added or removed. The following example monitors the amount of elements in a RuntimeSet to check whether it's empty:

```
[SerializeField]
private RuntimeSetBase allEnemies;
private bool hadElementsBefore = false;

private void OnEnable()
{
    allEnemies.onElementCountChange.AddResponse(UpdateElementCount);
}

private void OnDisable()
{
    allEnemies.onElementCountChange.RemoveResponse(UpdateElementCount);
}

private void UpdateElementCount(int newAmountOfEnemies)
{
    if (newAmountOfEnemies > 0)
    {
        hadElementsBefore = true;
    }
    else if (hadElementsBefore)
    {
        Debug.Log("All enemies have been defeated!");
        hadElementsBefore = false;
    }
}
```

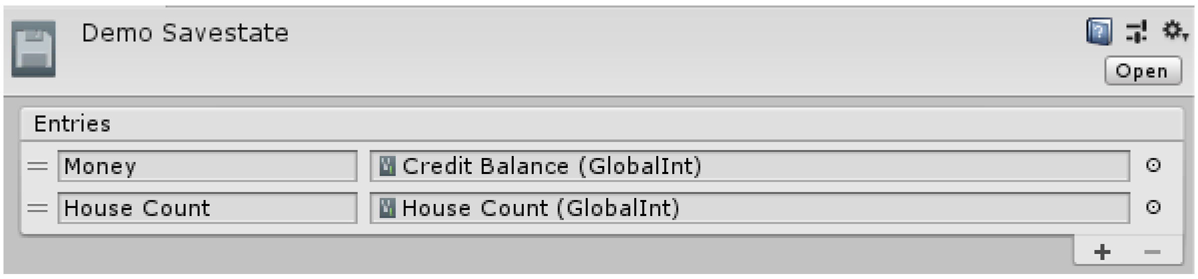
The `bool` field is used to ensure that the `Debug.Log` isn't triggered before the enemy objects registered themselves in the RuntimeSet.

See RuntimeSets in action!

Check the Soda Demo folder and open the "Sheep Controller" demo.

3.4. Savestates

A **Savestate** is a ScriptableObject that represents a dictionary, mapping GlobalVariables to string keys. It can be used to serialize and deserialize (save and load) the values of the GlobalVariables. This supports files and Unity's PlayerPrefs, but can be extended to also support web protocols.



Money	Credit Balance (GlobalInt)
House Count	House Count (GlobalInt)

To use a Savestate, create one in the create menu, under "Soda/Savestate".

Then, add any GlobalVariables to the list if you want them as part of your savegame, and assign a key to each of them, as shown above. Then, use any code you want to save or load the values by referencing the Savestate:

```
[SerializedField]
private Savestate savestate;
```

...and calling `Save` or `Load`:

```
private void Start()
{
  savestate.Load();
}

private void SaveGame()
{
  savestate.Save();
}
```

The Savestate system allows you to save and load from potentially any source. From PlayerPrefs, over files in Json, Xml or any other format, to web resources. Because of that, you need to specify an `ISavestateReader` for loading values and an `ISavestateWriter` for saving them. By default, Soda comes with a class implementing both interfaces for Unity's PlayerPrefs (called `SavestateReaderWriterPlayerPrefs`) and an example implementation of a class for saving the Savestate into a Json file.

It's recommended to us a `[RuntimeInitializeOnLoadMethod]` to set a default reader and writer for Savevstates as soon as the game launches:

```
[RuntimeInitializeOnLoadMethod]
private static void InitializeSavestateSystem()
{
    var readerWriter = new SavestateReaderWriterPlayerPrefs();
    SavestateSettings.defaultReader = readerWriter;
    SavestateSettings.defaultWriter = readerWriter;
}
```

With this code somewhere in your project, code loading and saving Savestate data, like shown on the previous page, will work just out of the box.

See Savestates in action!

Check the Soda Demo folder and open the "Savestates" demo.

The scripts provided in the "Savestates" demo are recommended material for studying best practices for Savestates.

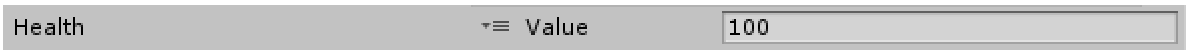
4. ScopedVariables

ScopedVariables are a feature easily overlooked, despite them being extremely powerful for clean code architecture when GlobalVariables are involved.

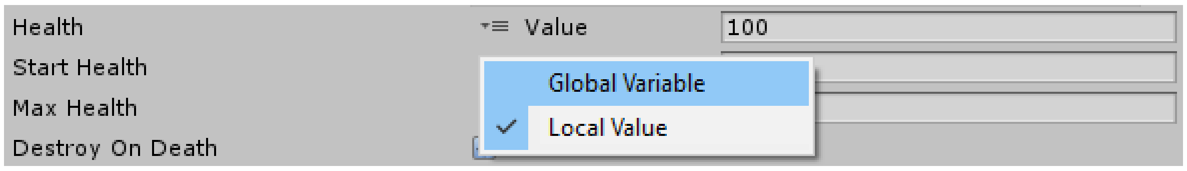
A ScopedVariable is an object used as a serialized field type, usually in a component. Each GlobalVariable type has a matching ScopedVariable type. The naming pattern is always the same. As an example, the `GlobalInt` type has a matching type `ScopedInt`.

```
[SerializeField]
private ScopedInt myNumber;
```

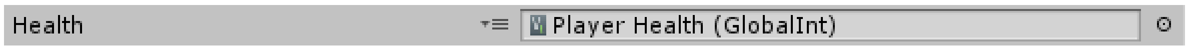
What a ScopedVariable does is rather simple. It allows you to pick between a **local value** of the ScopedVariable's type (in this case, `int`):


Health Value 100

...or, by clicking the little icon next to the value...


Health Value 100
Start Health
Max Health
Destroy On Death

...a field to drag and drop a matching GlobalVariable object into (a `GlobalInt` in this case):


Health Player Health (GlobalInt)

The consequences of this choice are rather massive, and a deep understanding of the yielded possibilities is very important.

A ScopedVariable field set to "local value" will act just like a regular field for the type in question. To stick with the `ScopedInt` example, this means an `int` field. If you make a prefab with a component that has a `ScopedInt` "health points" field set to "local value", each instance of the prefab will have its own health points value, as you'd expect.

As explained in the chapter 3.1. GlobalVariables, a GlobalVariable basically acts like a static field. So if we have a `GlobalInt` field, it's comparable to having a `static int` field.

In conclusion, ScopedVariables allow you to pick between a static and a local variable *on a per-object basis*. Imagine a health component that is used by both the player character and NPCs: The NPCs each have their own local HP `int` value, while the player character references a `GlobalInt` for that.

It is recommended to always use a ScopedVariable rather than a GlobalVariable *whenever you are not 100% certain that the field exists solely for write access to a GlobalVariable.*

5. Creating new Soda classes

5.1. Creating a new GlobalVariable type

The included types for GlobalVariables are useful, but you should not feel limited to them. Creating new GlobalVariable types allows you to reference anything that Unity can serialize. To make creation of new types both easy and safe, Soda comes with a wizard window that helps you create new GlobalVariable classes. You can access it under "Window/Soda/Create/GlobalVariable Type".

If you wanted, for example, a `GlobalQuaternion`, meaning a GlobalVariable representing a Quaternion, you would now enter "Quaternion" into the text field. By clicking "Generate", the wizard will create a new script file that contains the correct `GlobalQuaternion` class, a matching `ScopedQuaternion` class and some editor code. You don't have to further touch the class, it's instantly ready for use.

However, if you want your new GlobalVariable type to support Savestate loading and saving, you'd need to override and implement the methods `LoadValue` and `SaveValue`.

5.2. Creating a new RuntimeSet type

Similar to GlobalVariables, RuntimeSets have a wizard for creating new types. You can create RuntimeSets for GameObjects with any kind of component combination. You can find the wizard under "Window/Soda/Create/RuntimeSet Type". Follow its instructions to create script files that are ready for use immediately after creation.

5.3. Creating new Savestate readers and writers

Soda comes with Savestate support for PlayerPrefs and rudimentary support for JSON files. It is not recommended to use the included JSON support class for production – it's intended as an example only.

To extend Savestates to support any file type you like, or even web APIs, make a new class that implements the interfaces `ISavestateReader` and `ISavestateWriter`. Initialize the Savestate system with your custom class (see 3.4. Savestates) to have it serialize and deserialize your Savestate data in any way you like.

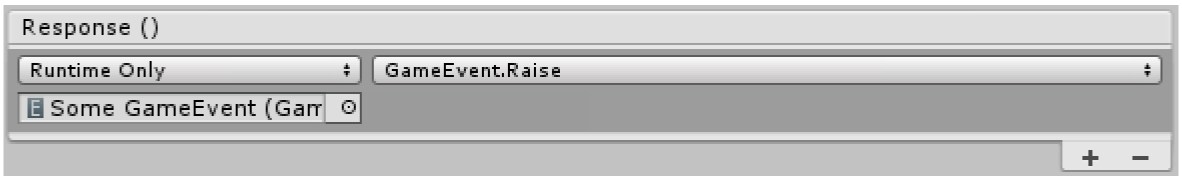
6. Best Practices

6.1. Combination with UnityEvents

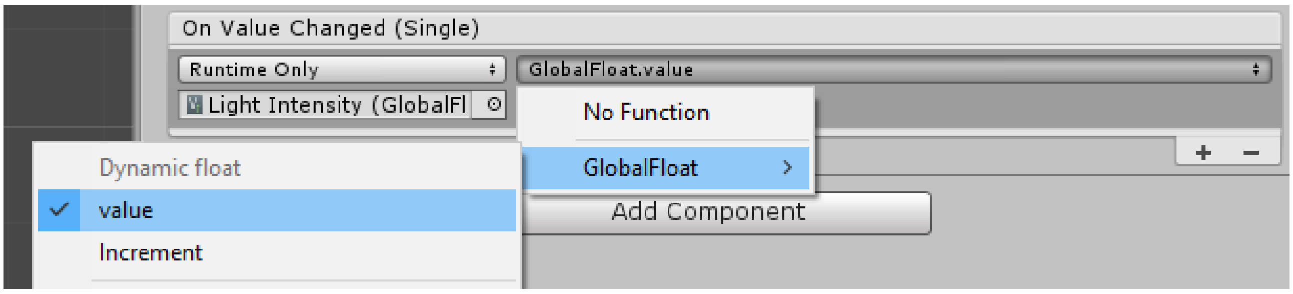
UnityEvents are a valuable ally when embracing Dependency Injection. They allow you to add an event to a component that is invoked whenever the component wants to; but the response to the event is entirely set in the Unity Editor. To bring up the classic example of the button that opens a door – you can give the button script a UnityEvent that is invoked when the button is used, drag the door into the button's inspector and pick the door's method that opens it. While this example can also be implemented by adding a serializable field referencing a door to the button script, using a UnityEvent instead allows us to assign any object, and even multiple objects, to respond to the button. By doing this, a `DoorButton` becomes a generic `Button` than can trigger **anything**.

Soda's ScriptableObjects are designed with UnityEvents in mind, and it's highly recommended to make use of them in order to create modular, reusable components.

For example, instead of creating a serializable field that references a `GameEvent`, create a serialized UnityEvent and drag the GameEvent into it, picking the `Raise` method:


Response ()
Runtime Only
GameEvent.Raise
Some GameEvent (Gam

Another example would be to assign a `GlobalFloat` to a regular UI Slider to set its value:


On Value Changed (Single)
Runtime Only
GlobalFloat.value
Light Intensity (GlobalFl

To think this further, imagine creating an entire options menu with globally accessible values. All the UI elements of the menu are setup by creating a properly chosen GlobalVariable and assigning it to the UnityEvent given by the UI Element.

When designing components, it is recommended to aim for a similar workflow result.

7. Not using Soda

As mentioned in 1. Introduction, it is important to know when to use Soda and when not to. More than once did people who got the hang of using Soda become obsessed with the concept, and they started using ScriptableObjects for every occasion that popped up. Despite Soda's goal being improving your project's architecture, the practice projects actually got messier than they would have been without Soda.

This section is meant to remind you that Soda is intended as the missing puzzle piece for Unity's dependency injection framework. In many cases, the best choice will be using standard Unity features. Since the core question is "how do objects communicate with each other?", consider these options:

- *Using a serialized field, you drag an object onto another.*
  This works for connections that are part of your scene design. Which door does the button open? Drag the door onto the button component.
- *The code responsible for spawning an object initializes its state.*
  This works for objects that are spawned after scene start. Which point on the map does an tower defence enemy want to run to? Consider dragging the target into the spawner component and have the spawner insitialize each spawned enemy.
- *Objects are introduced through a common context.*
  A common context could be the physics engine. Two colliders touching get to know each other through Unity's physics events. Another example would be a custom spatial hashmap in which an object can ask for nearby objects.
- *Use regular static data.*
  In section 3.1. GlobalVariables, it says "consider using a GlobalVariable object whenever you are considering a static variable for the purpose of other objects accessing it."
  This does not mean that you should stop using static fields and methods altogether. Some things are very well placed in a static context. If you can be reasonably sure to only have ever one of it, like an Input class or maybe your custom spatial hashmap, consider just using a static context.

If you go through these options or maybe even more, you often find a good and direct way to solve a problem. If none of the options work well in your context, consider using a Soda ScriptableObject.

Acquiring a good sense for when to use Soda - and when to use one of the many other available options instead - is very important for being able to create a robust and maintanable project architecture.

8. Support

If you have questions, issues, proposals or want to get to know other Soda developers, feel free to join the 13Pixels slack workspace: 13pixels.de/slack.

Alternatively, you can write a mail to assetstore@13pixels.de.

Have a productive time!