Creating Event Responders

One of the best ways for us to ensure our systems are exposed in and easy and accessible way to the designers is through Event Responders.

Event Responders are classes that can be added at-will to objects in the game through the Events GUI. All you need to do to create an Event Responder is create a single script that determines the Responder's behaviour and what types of events it can successfully respond to. Then, the Events GUI will automatically discover your script and make it available for use.

Example Event Responder

Once you know how an Event Responder is structured, the rest is easy.

```
[Serializable]
public class PlaySoundEventResponder : EventResponder
{
 [SerializeField]
 [Tooltip("The selected sound to play.")]
 private AudioClip soundToPlay;
 [SerializeField]
 [Tooltip("Sets the volume of the soundclip.")]
 private float volume;

 public override string Name
 {
  get { return "Play A Sound"; }
 }

 public override void HandleEvent(object sender, GenericEventArguments args)
 {
  //Play the sound
 }

 public override System.Type GetHandledType()
 {
  return typeof(GenericEventArguments);
 }
}
```

Let's go through this step by step:

```
[Serializable]
public class PlaySoundEventResponder : EventResponder
```

- Any Event Responder must subclass `EventResponder`- obvious, but there you go.
- Like a `MonoBehaviour`, each Event Responder must be defined in its own file where the name of the file matches the class name.
- Each EventResponder subclass should be marked `[Serializable]` to make sure Unity knows it can be safely saved/loaded in the scene.

```
[SerializeField]
[Tooltip("The selected sound to play.")]
private AudioClip soundToPlay;
```

- You can define any number of properties for your Responders, and just like a `MonoBehaviour`, they will be saved away inside the scene after being set through the Events GUI. However, any field to save must be marked with `[SerializeField]`.
- Also like a `MonoBehaviour`, your Responder properties can be anything- from `ints` and `floats` to `GameObjects` and `AudioClips`! Go crazy!

```
public override string Name
{
 get { return "Play A Sound"; }
}
```

- Every EventResponder needs to have a Name property- this is what will be displayed to the designers in the Events GUI, so make it something user-friendly and indicative of the Responder's behaviour!

```
public override System.Type GetHandledType()
{
 return typeof(GenericEventArguments);
}
```

- You need to also specify the specific type of event arguments that your Responder may accept- if you want to handle any event at all (anything from a collision to a switch event), specify `GenericEventArguments`. However, if you want to accept only a certain kind of event, e.g. a collision event, return the type of event arguments used by that event, e.g. `CollisonEventArguments`.

```
public override void HandleEvent(object sender, GenericEventArguments args)
{
 //Play the sound
}
```

- `HandleEvent` is where your actual Responder logic must go. Whatever it is your Responder is meant to do, do it here!
- Since `EventResponder` is not a generic class, `HandleEvent` can only accept `GenericEventArguments` objects. However, you can also use the convenience function `CastArguments<T>` defined in `EventResponder` for quick conversions and type checking- if the type is incorrect, an `ArgumentException` will be thrown with a descriptive message.

Limitations / Design Considerations

- Think carefully about the type of events you want to handle. The more generic you are the better, but that also limits the amount of information you can retrieve from events. This is something that can be improved through more functionality in the Events GUI, but for now it needs to be considered for each Responder.
- Remember that your Responder may not always be called when you expect it to be. Depending on its usage, it could be called on Awake(), before the game state is fully setup, or in some other kind of edge game state. You may need to account for this with some extra error-checking in your event handling.