Table of Contents

Namespace SODD.AI

Classes

CollisionDetector

CollisionDetector2D

State

StateMachine

Enums

CollisionDetectionStrategy

Enum CollisionDetectionStrategy

Namespace: SODD.AI

```
[Flags]
public enum CollisionDetectionStrategy
```

Fields

`Colliders = 1`

`None = 0`

`TriggerColliders = 2`

Class CollisionDetector

Namespace: SODD.AI

```
public class CollisionDetector : MonoBehaviour
```

Inheritance

object ← CollisionDetector

Class CollisionDetector2D

Namespace: SODD.AI

```
public sealed class CollisionDetector2D : MonoBehaviour
```

Inheritance

object ← CollisionDetector2D

Class State

Namespace: SODD.AI

```
public sealed class State : MonoBehaviour
```

Inheritance

object ← State

Methods

Enter()

```
public void Enter()
```

Exit()

```
public void Exit()
```

Class StateMachine

Namespace: SODD.AI

```
public class StateMachine : MonoBehaviour
```

Inheritance

object ← StateMachine

Properties

CurrentState

```
public State CurrentState { get; set; }
```

Property Value

State

Methods

AddCurrentStateFirstToQueue()

```
public void AddCurrentStateFirstToQueue()
```

AddCurrentStateLastToQueue()

```
public void AddCurrentStateLastToQueue()
```

AddFirstToQueue(State)

```
public void AddFirstToQueue(State state)
```

Parameters

`state` State

AddLastToQueue(State)

```
public void AddLastToQueue(State state)
```

Parameters

`state` State

ConsumeFirstFromQueue()

```
public void ConsumeFirstFromQueue()
```

ConsumeLastFromQueue()

```
public void ConsumeLastFromQueue()
```

Namespace SODD.Attributes

Classes

CollapsibleAttribute
: Makes a property collapsible in the Unity Inspector.

DisableIfAttribute
: Disables the property or field in the Unity Inspector based on the result of a Comparison between the value of the named field or property and a specified value.

DisableIfMatchAttribute
: Disables the property or field in the Unity Inspector if the value of the specified field or property matches any of the given values.

DisabledAttribute
: Disables the property or field, making it visible in the Unity Inspector but not editable.

HideIfAttribute
: Hides the property or field in the Unity Inspector if the specified Comparison between the value of the named field or property and a given value evaluates to true.

HideIfMatchAttribute
: Hides the property or field in the Unity Inspector if the value of the specified field or property matches any of the given values.

OnValueChangedAttribute
: Calls the specified method when the associated field's value is changed in the Unity Editor.

ShowIfAttribute
: Shows the property or field in the Unity Inspector if the specified Comparison between the value of the named field or property and a given value evaluates to true. The property or field is hidden otherwise.

ShowIfMatchAttribute
: Shows the property or field in the Unity Inspector if the value of the specified field or property matches any of the given values, and hides it otherwise.

Class CollapsibleAttribute

Namespace: SODD.Attributes

Makes a property collapsible in the Unity Inspector.

```
[AttributeUsage(AttributeTargets.Property|AttributeTargets.Field)]
public class CollapsibleAttribute : PropertyAttribute
```

Inheritance

object ← CollapsibleAttribute

Examples

```
public class MyComponent : MonoBehaviour
{
    [Collapsible("Click to Expand/Collapse")]
    public int myCollapsibleInt;
}
```

Remarks

This attribute can be applied to any serializable field to enable it to be collapsed or expanded in the Inspector with an optional custom label.

Constructors

CollapsibleAttribute(string)

```
public CollapsibleAttribute(string label = "")
```

Parameters

`label` string

Properties

Label

```
public string Label { get; }
```

Property Value

string

Class DisableIfAttribute

Namespace: SODD.Attributes

Disables the property or field in the Unity Inspector based on the result of a Comparison between the value of the named field or property and a specified value.

```
[AttributeUsage(AttributeTargets.Property|AttributeTargets.Field)]
public class DisableIfAttribute : PropertyAttribute
```

Inheritance

object ← DisableIfAttribute

Examples

The following example demonstrates the use of the DisableIfAttribute to disable a property based on the runtime value of another property:

```
public class ExampleBehavior : MonoBehaviour
{
    public bool toggle = true;

    [DisableIf("toggle", false)]
    public float disabledIfToggleIsFalse;
}
```

In this example, the 'disabledIfToggleIsFalse' float property becomes non-editable in the Inspector if the 'toggle' boolean property is set to false.

Remarks

This attribute allows dynamic control over the editability of properties and fields in the Unity Inspector, enhancing the interactivity based on the current state of the object's properties or fields.

Constructors

DisableIfAttribute(string, object, Comparison)

```
public DisableIfAttribute(string name, object value, Comparison comparison
= Comparison.Equals)
```

Parameters

`name` string

`value` object

`comparison` Comparison

Fields

Comparison

```
public readonly Comparison Comparison
```

Field Value

Comparison

Name

```
public readonly string Name
```

Field Value

string

Value

```
public readonly object Value
```

Field Value

[object](#)

Class DisableIfMatchAttribute

Namespace: SODD.Attributes

Disables the property or field in the Unity Inspector if the value of the specified field or property matches any of the given values.

```
[AttributeUsage(AttributeTargets.Property|AttributeTargets.Field)]
public class DisableIfMatchAttribute : PropertyAttribute
```

Inheritance

object ← DisableIfMatchAttribute

Examples

The following example demonstrates how to use the DisableIfMatchAttribute to disable a property when another property's value matches any specified value:

```
public class ExampleBehaviour : MonoBehaviour
{
    public string state;

    [DisableIfMatch("state", "initial", "loading")]
    public float progress;
}
```

In this example, the 'progress' float property becomes non-editable in the Inspector if the 'state' string property is either "initial" or "loading".

Remarks

This attribute is useful for dynamically controlling the editability of properties and fields based on the state of other properties or fields in the same object, particularly when multiple conditions could trigger a disable state.

Constructors

DisableIfMatchAttribute(string, params object[])

```
public DisableIfMatchAttribute(string name, params object[] values)
```

Parameters

`name` string

`values` object[]

Fields

Name

```
public readonly string Name
```

Field Value

string

Values

```
public readonly object[] Values
```

Field Value

object[]

Class DisabledAttribute

Namespace: SODD.Attributes

Disables the property or field, making it visible in the Unity Inspector but not editable.

```
[AttributeUsage(AttributeTargets.Property|AttributeTargets.Field)]
public class DisabledAttribute : PropertyAttribute
```

Inheritance

object ← DisabledAttribute

Examples

Here is how you can apply the DisabledAttribute to a property in your class:

```
public class ExampleClass : MonoBehaviour
{
    [Disabled]
    public float ReadOnlyValue;

    void Update()
    {
        ReadOnlyValue = CalculateSomeValue();
    }

    private float CalculateSomeValue()
    {
        return Time.deltaTime * 100;
    }
}
```

In this example, `ReadOnlyValue` is displayed in the Unity Inspector but cannot be modified by users, ensuring that only calculated values are shown.

Remarks

This attribute is useful for displaying properties or fields in the Inspector without allowing changes at runtime. This can be particularly helpful for debugging or for showing calculated values.

Class HideIfAttribute

Namespace: SODD.Attributes

Hides the property or field in the Unity Inspector if the specified Comparison between the value of the named field or property and a given value evaluates to true.

```
[AttributeUsage(AttributeTargets.Property|AttributeTargets.Field)]
public class HideIfAttribute : PropertyAttribute
```

Inheritance

object ← HideIfAttribute

Examples

Here is how you can apply the HideIfAttribute:

```
public class MyComponent : MonoBehaviour
{
    public bool showAdvancedOptions;

    [HideIf("showAdvancedOptions", false)]
    public float advancedSetting;
}
```

In this example, 'advancedSetting' will be hidden in the Inspector if 'showAdvancedOptions' is false.

Remarks

Use this attribute to conditionally hide properties or fields from the Unity Inspector based on the state of other properties or fields within the same object. This can be particularly useful for cleaning up the inspector by hiding irrelevant options based on the current configuration or state.

Constructors

HideIfAttribute(string, object, Comparison)

```
public HideIfAttribute(string name, object value, Comparison comparison
= Comparison.Equals)
```

Parameters

`name` string

`value` object

`comparison` Comparison

Fields

Comparison

```
public readonly Comparison Comparison
```

Field Value

Comparison

Name

```
public readonly string Name
```

Field Value

string

Value

```
public readonly object Value
```

Field Value

[object]

Class HideIfMatchAttribute

Namespace: SODD.Attributes

Hides the property or field in the Unity Inspector if the value of the specified field or property matches any of the given values.

```
[AttributeUsage(AttributeTargets.Property|AttributeTargets.Field)]
public class HideIfMatchAttribute : PropertyAttribute
```

Inheritance

object ← HideIfMatchAttribute

Examples

The following example shows how to use the HideIfMatchAttribute to hide a property when another property's value matches any of the specified conditions:

```
public class MyComponent : MonoBehaviour
{
    public string currentState;

    [HideIfMatch("currentState", "init", "start")]
    public float sensitiveSetting;
}
```

In this example, the 'sensitiveSetting' float property will be hidden in the Inspector if the 'currentState' string property is either "init" or "start".

Remarks

This attribute can be used to conditionally hide properties or fields from the Unity Inspector based on the state of other properties or fields within the same object. It is particularly useful for simplifying user interfaces by hiding irrelevant options dependent on other configurations.

Constructors

HideIfMatchAttribute(string, params object[])

```
public HideIfMatchAttribute(string name, params object[] values)
```

Parameters

`name` string

`values` object[]

Fields

Name

```
public readonly string Name
```

Field Value

string

Values

```
public readonly object[] Values
```

Field Value

object[]

Class OnValueChangedAttribute

Namespace: SODD.Attributes

Calls the specified method when the associated field's value is changed in the Unity Editor.

```
[AttributeUsage(AttributeTargets.Property|AttributeTargets.Field)]
public class OnValueChangedAttribute : PropertyAttribute
```

Inheritance

object ← OnValueChangedAttribute

Examples

```
using UnityEngine;
using SODD.Attributes;

public class Example : MonoBehaviour
{
    [OnValueChanged("OnHealthChanged")]
    public int health;

    private void OnHealthChanged()
    {
        Debug.Log("Health value changed to: " + health);
    }
}
```

This example demonstrates the attribute applied to a health field, where any change in the editor will automatically call the OnHealthChanged method to handle related updates or logic.

Remarks

This attribute is designed to automatically invoke a method when a serialized field's value is updated. It is primarily used within the SODD Framework to log changes of scriptable variable values in the console when their debug option is enabled.

The method specified by the attribute should have no parameters and exist within the same class as the field it is associated with.

Constructors

OnValueChangedAttribute(string)

```
public OnValueChangedAttribute(string methodName)
```

Parameters

`methodName` string

Fields

MethodName

```
public readonly string MethodName
```

Field Value

string

Class ShowIfAttribute

Namespace: SODD.Attributes

Shows the property or field in the Unity Inspector if the specified Comparison between the value of the named field or property and a given value evaluates to true. The property or field is hidden otherwise.

```
[AttributeUsage(AttributeTargets.Property|AttributeTargets.Field)]
public class ShowIfAttribute : PropertyAttribute
```

Inheritance

object ← ShowIfAttribute

Examples

Here is an example of how to use the ShowIfAttribute:

```
public class MyComponent : MonoBehaviour
{
    public bool toggleFeature;

    [ShowIf("toggleFeature", true)]
    public float featureSpecificSetting;
}
```

In this example, 'featureSpecificSetting' will be visible in the Inspector only if 'toggleFeature' is true.

Remarks

This attribute enables dynamic control over the visibility of properties and fields in the Unity Inspector, depending on the state of other properties or fields within the same object. It is particularly useful for creating more intuitive and manageable interfaces by displaying only relevant options based on conditional logic.

Constructors

ShowIfAttribute(string, object, Comparison)

```
public ShowIfAttribute(string name, object value, Comparison comparison
= Comparison.Equals)
```

Parameters

`name` string

`value` object

`comparison` Comparison

Fields

Comparison

```
public readonly Comparison Comparison
```

Field Value

Comparison

Name

```
public readonly string Name
```

Field Value

string

Value

```
public readonly object Value
```

Field Value

object

Class ShowIfMatchAttribute

Namespace: SODD.Attributes

Shows the property or field in the Unity Inspector if the value of the specified field or property matches any of the given values, and hides it otherwise.

```
[AttributeUsage(AttributeTargets.Property|AttributeTargets.Field)]
public class ShowIfMatchAttribute : PropertyAttribute
```

Inheritance

object ← ShowIfMatchAttribute

Examples

The following example shows how to use the ShowIfMatchAttribute to conditionally show a property:

```
public class GameplaySettings : MonoBehaviour
{
    public string difficultyLevel;

    [ShowIfMatch("difficultyLevel", "Hard", "Extreme")]
    public float damageMultiplier;
}
```

In this example, the 'damageMultiplier' property will only be visible in the Inspector when 'difficultyLevel' is set to "Hard" or "Extreme".

Remarks

This attribute can be used to conditionally show or hide properties or fields from the Unity Inspector based on the state of other properties or fields within the same object. It is particularly useful for managing complex user interfaces, allowing the display of relevant options only when appropriate.

Constructors

ShowIfMatchAttribute(string, params object[])

```
public ShowIfMatchAttribute(string name, params object[] values)
```

Parameters

`name` string

`values` object[]

Fields

Name

```
public readonly string Name
```

Field Value

string

Values

```
public readonly object[] Values
```

Field Value

object[]

Namespace SODD.Collections

Classes

AudioClipCollection
: Represents a scriptable collection that stores AudioClip objects.

Collection<T>
: Represents an abstract collection of items of type T.

ComponentCollection
: Represents a scriptable collection that stores Unity Components.

GameObjectCollection
: Represents a collection that stores GameObjects.

ObjectCollection
: Represents a collection that stores Objects.

ScriptableObjectCollection
: Represents a collection that stores ScriptableObjects.

Class AudioClipCollection

Namespace: SODD.Collections

Represents a scriptable collection that stores AudioClip objects.

```
public sealed class AudioClipCollection : Collection<AudioClip>, IList<AudioClip>,
ICollection<AudioClip>, IEnumerable<AudioClip>, IEnumerable
```

Inheritance

object ← Collection<AudioClip> ← AudioClipCollection

Implements

IList<AudioClip>, ICollection<AudioClip>, IEnumerable<AudioClip>, IEnumerable

Inherited Members

Collection<AudioClip>.OnItemAdded , Collection<AudioClip>.OnItemRemoved , Collection<AudioClip>.GetEnumerator() , Collection<AudioClip>.Add(AudioClip) , Collection<AudioClip>.Clear() , Collection<AudioClip>.Contains(AudioClip) , Collection<AudioClip>.CopyTo(AudioClip[], int) , Collection<AudioClip>.Remove(AudioClip) , Collection<AudioClip>.Count , Collection<AudioClip>.IsReadOnly , Collection<AudioClip>.IndexOf(AudioClip) , Collection<AudioClip>.Insert(int, AudioClip) , Collection<AudioClip>.RemoveAt(int) , Collection<AudioClip>.this[int] , Collection<AudioClip>.GetRandom()

Extension Methods

EnumerableExtensions.ForEach<T>(IEnumerable<T>, Action<T>) , EnumerableExtensions.IsEmpty<T>(IEnumerable<T>) , EnumerableExtensions.TryFind<T>(IEnumerable<T>, Func<T, bool>, out T) , EnumerableExtensions.WrapAround<T>(IEnumerable<T>)

Examples

Example of using an AudioClipCollection to play audio at a specific location triggered by an event:

```
using UnityEngine;
using SODD.Collections;
using SODD.Events;

public class Example : MonoBehaviour
```

```
{
    public AudioClipCollection shotsAudioPool; // Assign this through the
Unity Editor.
    public Vector3Event onShotFired; // Assign this through the Unity Editor.

    private void OnEnable()
    {
        onShotFired.AddListener(OnShotFired);
    }

    private void OnDisable()
    {
        onShotFired.RemoveListener(OnShotFired);
    }

    private void OnShotFired(Vector3 position)
    {
        var audioClip = shotsAudioPool.GetRandom();
        AudioSource.PlayClipAtPoint(audioClip, position);
    }
}
```

This script demonstrates how an AudioClipCollection can be used in conjunction with a Vector3Event to play a random audio clip from a collection at the position specified by the event.

Remarks

This collection provides a concrete implementation of the abstract Collection<T> class, specifically for managing audio clips within a game. It facilitates the organized storage and access of sound assets, such as music tracks, sound effects, and other audio samples, which can be manipulated and referenced dynamically throughout the game's lifecycle.

Class Collection<T>

Namespace: SODD.Collections

Represents an abstract collection of items of type T.

```
public abstract class Collection<T> : ScriptableObject, IList<T>, ICollection<T>,
IEnumerable<T>, IEnumerable
```

Type Parameters

T

The type of elements in the collection.

Inheritance

object ← Collection<T>

Implements

IList<T>, ICollection<T>, IEnumerable<T>, IEnumerable

Derived

AudioClipCollection, ComponentCollection, GameObjectCollection, ObjectCollection, ScriptableObjectCollection

Extension Methods

EnumerableExtensions.ForEach<T>(IEnumerable<T>, Action<T>) , EnumerableExtensions.IsEmpty<T>(IEnumerable<T>) , EnumerableExtensions.TryFind<T>(IEnumerable<T>, Func<T, bool>, out T) , EnumerableExtensions.WrapAround<T>(IEnumerable<T>)

Examples

Example of a GameObject collection implementation:

```
[CreateAssetMenu(menuName = "My Collections/GameObject Collection", fileName =
nameof(GameObjectCollection)]
public class GameObjectCollection : Collection<GameObject> {}
```

Remarks

This abstract class provides the foundation for creating scriptable collection implementations. It enables the encapsulation and serialization a list of items, allowing for easy manipulation and reference within the Unity Editor and across different game components. Such collections are beneficial for managing groups of related items, such as inventory items, enemies, or waypoints, in a centralized and organized manner.

The collection supports adding and removing items and provides events for tracking these modifications. This functionality facilitates reactive programming patterns, where changes to the collection can trigger corresponding actions or updates within the game.

To create a custom collection, inherit from this class and specify the item type `T`, which can be any type supported by Unity (e.g., GameObject, int, string).

Fields

OnItemAdded

Event triggered when an item is added to the collection.

```
public GenericEvent<T> OnItemAdded
```

Field Value

GenericEvent<T>

OnItemRemoved

Event triggered when an item is removed from the collection.

```
public GenericEvent<T> OnItemRemoved
```

Field Value

GenericEvent<T>

Properties

Count

Gets the number of elements contained in the collection.

```
public int Count { get; }
```

Property Value

int

IsReadOnly

Gets a value indicating whether the collection is read-only.

```
public bool IsReadOnly { get; }
```

Property Value

bool

this[int]

Gets or sets the element at the specified index.

```
public T this[int index] { get; set; }
```

Parameters

index int

The zero-based index of the element to get or set.

Property Value

T

The element at the specified index.

Methods

Add(T)

Adds an item to the collection, triggering the OnItemAdded event.

```
public void Add(T item)
```

Parameters

`item` T

The item to add to the collection.

Clear()

Removes all items from the collection, triggering the OnItemRemoved event for each item.

```
public void Clear()
```

Contains(T)

Determines whether the collection contains a specific value.

```
public bool Contains(T item)
```

Parameters

`item` T

The object to locate in the collection.

Returns

bool

true if item is found in the collection; otherwise, false.

CopyTo(T[], int)

Copies the elements of the collection to an Array, starting at a particular Array index.

```
public void CopyTo(T[] array, int arrayIndex)
```

Parameters

`array` T[]

The one-dimensional Array that is the destination of the elements copied from the collection.

`arrayIndex` int

The zero-based index in array at which copying begins.

GetEnumerator()

Returns an enumerator that iterates through the collection.

```
public IEnumerator<T> GetEnumerator()
```

Returns

IEnumerator<T>

An enumerator for the collection of items.

GetRandom()

Returns a random item from the collection.

```
public T GetRandom()
```

Returns

T

A randomly selected item from the collection.

IndexOf(T)

Retrieves the index of a specific item in the collection.

```
public int IndexOf(T item)
```

Parameters

`item` T

The item to locate in the collection.

Returns

int

The index of the item if found in the collection; otherwise, -1.

Insert(int, T)

Inserts an item at the specified index in the collection.

```
public void Insert(int index, T item)
```

Parameters

`index` int

The zero-based index at which item should be inserted.

`item` T

The item to insert.

Remove(T)

Removes the first occurrence of a specific object from the collection, triggering the OnItemRemoved event.

```
public bool Remove(T item)
```

Parameters

`item` T

The item to remove from the collection.

Returns

bool

true if item was successfully removed from the collection; otherwise, false.

RemoveAt(int)

Removes the item at the specified index from the collection.

```
public void RemoveAt(int index)
```

Parameters

`index` int

The zero-based index of the item to remove.

Class ComponentCollection

Namespace: SODD.Collections

Represents a scriptable collection that stores Unity Components.

```
public sealed class ComponentCollection : Collection<Component>, IList<Component>,
ICollection<Component>, IEnumerable<Component>, IEnumerable
```

Inheritance

object ← Collection<Component> ← ComponentCollection

Implements

IList<Component>, ICollection<Component>, IEnumerable<Component>, IEnumerable

Inherited Members

Collection<Component>.OnItemAdded , Collection<Component>.OnItemRemoved , Collection<Component>.GetEnumerator() , Collection<Component>.Add(Component) , Collection<Component>.Clear() , Collection<Component>.Contains(Component) , Collection<Component>.CopyTo(Component[], int) , Collection<Component>.Remove(Component) , Collection<Component>.Count , Collection<Component>.IsReadOnly , Collection<Component>.IndexOf(Component) , Collection<Component>.Insert(int, Component) , Collection<Component>.RemoveAt(int) , Collection<Component>.this[int] , Collection<Component>.GetRandom()

Extension Methods

EnumerableExtensions.ForEach<T>(IEnumerable<T>, Action<T>) , EnumerableExtensions.IsEmpty<T>(IEnumerable<T>) , EnumerableExtensions.TryFind<T>(IEnumerable<T>, Func<T, bool>, out T) , EnumerableExtensions.WrapAround<T>(IEnumerable<T>)

Examples

Example of using ComponentCollection to manage explosives in a game:

```
public class Detonator : MonoBehaviour
{
    public ComponentCollection placedExplosives; // Assign this via the
Unity Editor.
    public VoidEvent onDetonateExplosives;
```

```
    private void OnEnable()
    {
        onDetonateExplosives.AddListener(DetonateExplosives);
    }

    private void OnDisable()
    {
        onDetonateExplosives.RemoveListener(DetonateExplosives);
    }

    private void DetonateExplosives(Void o)
    {
        foreach (var component in placedExplosives)
        {
            var explosive = (Explosive) component;
            explosive.Detonate();
        }
        placedExplosives.Clear();
    }
}
```

This script shows how explosives can dynamically add themselves to the 'placedExplosives' collection when instantiated and how a detonator can trigger their detonation without needing to directly gather or reference individual explosives. This allows the detonator to operate independently of the explosives' management.

Remarks

This collection provides a concrete implementation of the abstract Collection<T> class, specifically designed for managing Unity Components. It enables the structured and dynamic management of components within a game, facilitating operations like adding, removing, or iterating over components without requiring direct management by other objects. An essential use of this collection is in scenarios where components need to be managed independently of the objects that use them, enhancing modularity and reducing coupling in game architecture.

Class GameObjectCollection

Namespace: SODD.Collections

Represents a collection that stores GameObjects.

```
public sealed class GameObjectCollection : Collection<GameObject>,
IList<GameObject>, ICollection<GameObject>, IEnumerable<GameObject>, IEnumerable
```

Inheritance

object ← Collection<GameObject> ← GameObjectCollection

Implements

IList<GameObject>, ICollection<GameObject>, IEnumerable<GameObject>, IEnumerable

Inherited Members

Collection<GameObject>.OnItemAdded , Collection<GameObject>.OnItemRemoved , Collection<GameObject>.GetEnumerator() , Collection<GameObject>.Add(GameObject) , Collection<GameObject>.Clear() , Collection<GameObject>.Contains(GameObject) , Collection<GameObject>.CopyTo(GameObject[], int) , Collection<GameObject>.Remove(GameObject) , Collection<GameObject>.Count , Collection<GameObject>.IsReadOnly , Collection<GameObject>.IndexOf(GameObject) , Collection<GameObject>.Insert(int, GameObject) , Collection<GameObject>.RemoveAt(int) , Collection<GameObject>.this[int] , Collection<GameObject>.GetRandom()

Extension Methods

EnumerableExtensions.ForEach<T>(IEnumerable<T>, Action<T>) , EnumerableExtensions.IsEmpty<T>(IEnumerable<T>) , EnumerableExtensions.TryFind<T>(IEnumerable<T>, Func<T, bool>, out T) , EnumerableExtensions.WrapAround<T>(IEnumerable<T>)

See Also

Collection<T>

Class ObjectCollection

Namespace: SODD.Collections

Represents a collection that stores Objects.

```
public sealed class ObjectCollection : Collection<Object>, IList<Object>,
ICollection<Object>, IEnumerable<Object>, IEnumerable
```

Inheritance

object ← Collection<Object> ← ObjectCollection

Implements

IList<Object>, ICollection<Object>, IEnumerable<Object>, IEnumerable

Inherited Members

Collection<Object>.OnItemAdded , Collection<Object>.OnItemRemoved , Collection<Object>.GetEnumerator() , Collection<Object>.Add(Object) , Collection<Object>.Clear() , Collection<Object>.Contains(Object) , Collection<Object>.CopyTo(Object[], int) , Collection<Object>.Remove(Object) , Collection<Object>.Count , Collection<Object>.IsReadOnly , Collection<Object>.IndexOf(Object) , Collection<Object>.Insert(int, Object) , Collection<Object>.RemoveAt(int) , Collection<Object>.this[int] , Collection<Object>.GetRandom()

Extension Methods

EnumerableExtensions.ForEach<T>(IEnumerable<T>, Action<T>) , EnumerableExtensions.IsEmpty<T>(IEnumerable<T>) , EnumerableExtensions.TryFind<T>(IEnumerable<T>, Func<T, bool>, out T) , EnumerableExtensions.WrapAround<T>(IEnumerable<T>)

See Also

Collection<T>

Class ScriptableObjectCollection

Namespace: SODD.Collections

Represents a collection that stores ScriptableObjects.

```
public sealed class ScriptableObjectCollection : Collection<ScriptableObject>,
IList<ScriptableObject>, ICollection<ScriptableObject>,
IEnumerable<ScriptableObject>, IEnumerable
```

Inheritance

object ← Collection<ScriptableObject> ← ScriptableObjectCollection

Implements

IList<ScriptableObject>, ICollection<ScriptableObject>, IEnumerable<ScriptableObject>, IEnumerable

Inherited Members

Collection<ScriptableObject>.OnItemAdded ,
Collection<ScriptableObject>.OnItemRemoved ,
Collection<ScriptableObject>.GetEnumerator() ,
Collection<ScriptableObject>.Add(ScriptableObject) ,
Collection<ScriptableObject>.Clear() ,
Collection<ScriptableObject>.Contains(ScriptableObject) ,
Collection<ScriptableObject>.CopyTo(ScriptableObject[], int) ,
Collection<ScriptableObject>.Remove(ScriptableObject) ,
Collection<ScriptableObject>.Count , Collection<ScriptableObject>.IsReadOnly ,
Collection<ScriptableObject>.IndexOf(ScriptableObject) ,
Collection<ScriptableObject>.Insert(int, ScriptableObject) ,
Collection<ScriptableObject>.RemoveAt(int) , Collection<ScriptableObject>.this[int] ,
Collection<ScriptableObject>.GetRandom()

Extension Methods

EnumerableExtensions.ForEach<T>(IEnumerable<T>, Action<T>) ,
EnumerableExtensions.IsEmpty<T>(IEnumerable<T>) ,
EnumerableExtensions.TryFind<T>(IEnumerable<T>, Func<T, bool>, out T) ,
EnumerableExtensions.WrapAround<T>(IEnumerable<T>)

See Also

Collection<T>

Namespace SODD.Core

Classes

AudioMixerExtensions
: Extends AudioMixer type providing utility methods.

ComparisonExtensions
: Provides extension methods for the Comparison enum.

CoroutineBuilder
: Provides a flexible builder for constructing and managing custom coroutines.

EnumerableExtensions
: Extends IEnumerable<T> type providing utility methods.

GameObjectExtensions
: Extends GameObject type providing utility methods.

LifecycleEvents

Logger
: Provides utility functions for logging in the Unity Editor.

NavMeshAgentExtensions
: Extends NavMeshAgent type providing utility methods.

PassiveScriptableObject

PassiveScriptableObjectLoader

PersistentScriptableObject

PrimitiveTypeExtensions
: Extends primitive types providing utility methods.

TransformExtensions
: Extends Transform type providing utility methods.

Vector2Extensions
: Extends Vector2 type providing utility methods.

Vector3Extensions
: Extends Vector3 type providing utility methods.

Structs

Void

Represents a type that signifies the absence of any value.

Enums

Comparison

Defines comparison operations between values.

Scope

Specifies the scope of search for components in a GameObject.

ToStringStrategy

Class AudioMixerExtensions

Namespace: SODD.Core

Extends AudioMixer type providing utility methods.

```
public static class AudioMixerExtensions
```

Inheritance

object ← AudioMixerExtensions

Methods

GetVolume(AudioMixer, string, out float)

Gets the volume level of a specific parameter in the AudioMixer.

```
public static bool GetVolume(this AudioMixer mixer, string name, out
float percentage)
```

Parameters

`mixer` AudioMixer

The AudioMixer instance.

`name` string

The name of the parameter whose volume to retrieve.

`percentage` float

The output variable that will contain the volume level as a percentage (0 to 100).

Returns

bool

True if the parameter exists and its volume level was successfully retrieved; otherwise, false if the parameter doesn't exist or an error occurred during retrieval.

Remarks

This method retrieves the volume level of the specified parameter in the AudioMixer and converts it from the logarithmic scale (decibels) to a linear scale represented as a percentage.

SetVolume(AudioMixer, string, float)

Sets the volume level of a specific parameter in the AudioMixer.

```
public static void SetVolume(this AudioMixer mixer, string name, float percentage)
```

Parameters

`mixer` AudioMixer

The AudioMixer instance.

`name` string

The name of the parameter whose volume to set.

`percentage` float

The volume level to set as a percentage (0 to 100).

Remarks

This method sets the volume level of the specified parameter in the AudioMixer by converting the input percentage from a linear scale to the logarithmic scale (decibels). If the input percentage is less than 0, it will be clamped to 0. If the input percentage is greater than 100, it will be clamped to 100.

Enum Comparison

Namespace: SODD.Core

Defines comparison operations between values.

```
public enum Comparison
```

Extension Methods
ComparisonExtensions.Evaluate(Comparison, object, object)

Fields

`BiggerEqualsThan = 3`

`BiggerThan = 2`

`Equals = 0`

`LessEqualsThan = 5`

`LessThan = 4`

`NotEquals = 1`

Class ComparisonExtensions

Namespace: SODD.Core

Provides extension methods for the Comparison enum.

```
public static class ComparisonExtensions
```

Inheritance

object ← ComparisonExtensions

Methods

Evaluate(Comparison, object, object)

Evaluates the comparison between two objects based on the specified Comparison type.

```
public static bool Evaluate(this Comparison comparison, object value1,
object value2)
```

Parameters

`comparison` Comparison

The type of comparison to perform.

`value1` object

The first object to compare.

`value2` object

The second object to compare.

Returns

bool

A boolean value indicating the result of the comparison.

Examples

This example shows how to use the Evaluate method:

```
Comparison comparisonType = Comparison.LessThan;
int value1 = 5;
int value2 = 10;
bool result = comparisonType.Evaluate(value1, value2);
// result will be true because 5 is less than 10
```

Exceptions

ArgumentException

Thrown when the objects are not comparable or the comparison type is not applicable.

Class CoroutineBuilder

Namespace: SODD.Core

Provides a flexible builder for constructing and managing custom coroutines.

```
public class CoroutineBuilder : MonoBehaviour, ICloneable
```

Inheritance

object ← CoroutineBuilder

Implements

ICloneable

Remarks

The CoroutineBuilder class allows developers to dynamically construct coroutines by chaining various types of execution steps, such as invoking actions, waiting for seconds, or repeating steps. This class simplifies complex coroutine creation and management, making it easy to customize behavior at runtime without writing repetitive coroutine functions.

It supports conditional waits, loops, and method invocation, providing a powerful tool for creating sophisticated asynchronous behaviors.

Properties

IsRunning

Gets a value indicating whether the coroutine is currently running.

```
public bool IsRunning { get; }
```

Property Value

bool

Methods

Append(params CoroutineBuilder[])

Appends the specified coroutines to the current CoroutineBuilder instance.

```
public CoroutineBuilder Append(params CoroutineBuilder[] coroutines)
```

Parameters

`coroutines` CoroutineBuilder[]

The coroutines to append.

Returns

CoroutineBuilder

The current CoroutineBuilder instance.

Exceptions

Exception

Thrown when one of the coroutines is currently running.

Cancel()

Cancels the running coroutine.

```
public void Cancel()
```

CancelOnDisable(bool)

Configures whether the CoroutineBuilder instance should be canceled when it gets disabled.

```
public CoroutineBuilder CancelOnDisable(bool condition = true)
```

Parameters

`condition` bool

True if the instance should be canceled; otherwise, false.

Returns

CoroutineBuilder

The current CoroutineBuilder instance.

Clone()

Creates a shallow copy of the current CoroutineBuilder instance.

```
public object Clone()
```

Returns

object

A shallow copy of the current CoroutineBuilder instance.

DestroyOnFinish(bool)

Configures whether the CoroutineBuilder instance should be destroyed when the coroutine finishes.

```
public CoroutineBuilder DestroyOnFinish(bool condition = true)
```

Parameters

`condition` bool

True if the instance should be destroyed; otherwise, false.

Returns

CoroutineBuilder

The current CoroutineBuilder instance.

ForTimes(int)

Adds an execution step to repeat the previous steps a specified number of times during the coroutine execution.

```
public CoroutineBuilder ForTimes(int times)
```

Parameters

`times` int

The number of times to repeat the previous steps.

Returns

CoroutineBuilder

The current CoroutineBuilder instance.

Invoke(Action)

Adds an execution step to invoke the specified action during the coroutine execution.

```
public CoroutineBuilder Invoke(Action action)
```

Parameters

`action` Action

The action to be invoked.

Returns

CoroutineBuilder

The current CoroutineBuilder instance.

Run()

Runs the configured coroutine.

```
public void Run()
```

WaitForEndOfFrame()

Adds an execution step to wait for the end of the current frame during the coroutine execution.

```
public CoroutineBuilder WaitForEndOfFrame()
```

Returns

CoroutineBuilder

The current CoroutineBuilder instance.

WaitForFixedUpdate()

Adds an execution step to wait for the next fixed update during the coroutine execution.

```
public CoroutineBuilder WaitForFixedUpdate()
```

Returns

CoroutineBuilder

The current CoroutineBuilder instance.

WaitForSeconds(float)

Adds an execution step to wait for a specified number of seconds during the coroutine execution.

```
public CoroutineBuilder WaitForSeconds(float seconds)
```

Parameters

`seconds` float

The number of seconds to wait.

Returns

CoroutineBuilder

The current CoroutineBuilder instance.

WaitUntil(Func<bool>)

Adds an execution step to wait until the specified predicate is true during the coroutine execution.

```
public CoroutineBuilder WaitUntil(Func<bool> predicate)
```

Parameters

`predicate` Func<bool>

The predicate function to evaluate.

Returns

CoroutineBuilder

The current CoroutineBuilder instance.

WaitWhile(Func<bool>)

Adds an execution step to wait while the specified predicate is false during the coroutine execution.

```
public CoroutineBuilder WaitWhile(Func<bool> predicate)
```

Parameters

`predicate` Func<bool>

The predicate function to evaluate.

Returns

CoroutineBuilder

The current CoroutineBuilder instance.

While(Func<bool>)

Adds an execution step to wait while the specified predicate is true during the coroutine execution.

```
public CoroutineBuilder While(Func<bool> predicate)
```

Parameters

`predicate` Func<bool>

The predicate function to evaluate.

Returns

CoroutineBuilder

The current CoroutineBuilder instance.

Class EnumerableExtensions

Namespace: SODD.Core

Extends IEnumerable<T> type providing utility methods.

```
public static class EnumerableExtensions
```

Inheritance

object ← EnumerableExtensions

Methods

ForEach<T>(IEnumerable<T>, Action<T>)

Performs the specified action on each element of the IEnumerable<T>.

```
public static void ForEach<T>(this IEnumerable<T> source, Action<T> action)
```

Parameters

`source` IEnumerable<T>

The IEnumerable<T> to iterate over.

`action` Action<T>

The action to perform on each element of the IEnumerable<T>.

Type Parameters

`T`

The type of elements in the IEnumerable<T>.

Remarks

This method iterates over each element in the source IEnumerable<T> and performs the specified action on each element using the provided delegate. If the action is null, this

method does nothing and returns immediately.

IsEmpty<T>(IEnumerable<T>)

```
public static bool IsEmpty<T>(this IEnumerable<T> source)
```

Parameters

`source` IEnumerable<T>

Returns

bool

Type Parameters

`T`

TryFind<T>(IEnumerable<T>, Func<T, bool>, out T)

Tries to find an element in the IEnumerable<T> that matches the specified predicate.

```
public static bool TryFind<T>(this IEnumerable<T> source, Func<T, bool> predicate,
out T output)
```

Parameters

`source` IEnumerable<T>

The IEnumerable<T> to search.

`predicate` Func<T, bool>

The predicate function used to find the element.

`output` T

When this method returns, contains the first element that matches the predicate, if found; otherwise, the default value of type T.

Returns

bool

True if an element matching the predicate is found and stored in the `output` parameter; otherwise, false if no such element is found.

Type Parameters

`T`

The type of elements in the IEnumerable<T>.

Remarks

This method searches the source IEnumerable<T> for an element that matches the specified predicate function. If a matching element is found, it is stored in the `output` parameter and the method returns true. If no matching element is found, the `output` parameter is set to the default value of type T and the method returns false.

WrapAround<T>(IEnumerable<T>)

Creates an infinite loop by wrapping the source IEnumerable<T> around itself.

```
public static IEnumerable<T> WrapAround<T>(this IEnumerable<T> source)
```

Parameters

`source` IEnumerable<T>

The IEnumerable<T> to wrap around.

Returns

IEnumerable<T>

An IEnumerable<T> that repeatedly returns elements from the source IEnumerable<T> in an infinite loop.

Type Parameters

T

The type of elements in the IEnumerable<T>.

Remarks

This method creates an infinite loop by repeatedly returning elements from the source IEnumerable<T>. When the end of the source sequence is reached, it wraps around and starts again from the beginning, effectively creating an infinite sequence.

Class GameObjectExtensions

Namespace: SODD.Core

Extends GameObject type providing utility methods.

```
public static class GameObjectExtensions
```

Inheritance

object ← GameObjectExtensions

Methods

Broadcast<T>(GameObject, Action<T>, Scope)

Executes the provided Action<T> on all components of type `T` found on the specified `gameObject` within the defined `scope`.

```
public static void Broadcast<T>(this GameObject gameObject, Action<T> action, Scope
scope = Scope.GameObject)
```

Parameters

`gameObject` GameObject

The GameObject from which the components are to be retrieved and the action is to be executed on each.

`action` Action<T>

The action to execute on each retrieved component. If no components are found, the action is not executed.

`scope` Scope

The Scope defining where to search for the components. The default is GameObject.

Type Parameters

`T`

The type of the components to retrieve and act upon.

Examples

The following example shows the use of `Broadcast` to disable all Collider components within and including children of a GameObject:

```
public class ExampleUsage : MonoBehaviour
{
    void Start()
    {
        gameObject.Broadcast<Collider>(collider => collider.enabled =
false, Scope.Children);
    }
}
```

Remarks

This method facilitates the application of a single action to multiple components of the same type distributed throughout the specified scope. It is particularly useful for scenarios where a uniform operation needs to be applied to an array of components, such as enabling, disabling, or resetting component states.

Coroutine(GameObject, bool, bool)

Creates and attaches a CoroutineBuilder to this GameObject.

```
public static CoroutineBuilder Coroutine(this GameObject gameObject, bool
destroyOnFinish = true, bool cancelOnDisable = true)
```

Parameters

`gameObject` GameObject

The GameObject on which to create the CoroutineBuilder.

`destroyOnFinish` bool

If set to true, the CoroutineBuilder will be automatically destroyed when the coroutine execution completes.

`cancelOnDisable` bool

If set to true, the coroutine execution will be automatically canceled if the GameObject is disabled.

Returns

CoroutineBuilder

A CoroutineBuilder attached to this GameObject.

Examples

```
// Create a coroutine that waits 2 seconds, then logs a message to the console
gameObject.Coroutine()
    .WaitForSeconds(2)
    .Invoke(() => Debug.Log("2 seconds have passed"))
    .Run();
```

Remarks

This method dynamically adds a new CoroutineBuilder component to the specified GameObject. It is ideal for situations where the behavior of the GameObject needs to be extended with custom asynchronous routines that can be defined on-the-fly using method chaining. The CoroutineBuilder supports various types of execution steps like delays, loops, and condition-based continuations, making it versatile for complex asynchronous behaviors.

DirectionFrom(GameObject, GameObject)

Calculates the direction vector from the position of the specified GameObject to the position of the GameObject.

```
public static Vector3 DirectionFrom(this GameObject gameObject, GameObject other)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the direction vector to.

`other` GameObject

The source GameObject whose position is the source of the direction vector.

Returns

Vector3

A Vector3 representing the direction vector from the position of the specified GameObject to the position of the GameObject.

Exceptions

ArgumentNullException

Thrown when the specified GameObject is null.

DirectionFrom(GameObject, Transform)

Calculates the direction vector from the specified Transform to the position of the GameObject.

```
public static Vector3 DirectionFrom(this GameObject gameObject, Transform transform)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the direction vector to.

`transform` Transform

The Transform whose position is the source of the direction vector.

Returns

Vector3

A Vector3 representing the direction vector from the specified Transform to the position of the GameObject.

Exceptions

ArgumentNullException

Thrown when the specified Transform is null.

DirectionFrom(GameObject, Vector3)

Calculates the direction vector from the specified point to the position of the GameObject.

```
public static Vector3 DirectionFrom(this GameObject gameObject, Vector3 point)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the direction vector to.

`point` Vector3

The source point from which the direction vector is calculated.

Returns

Vector3

A Vector3 representing the direction vector from the specified point to the position of the GameObject.

DirectionTo(GameObject, GameObject)

Calculates the direction vector from the position of the GameObject to the position of the specified GameObject.

```
public static Vector3 DirectionTo(this GameObject gameObject, GameObject other)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the direction vector from.

`other` GameObject

The target GameObject whose position is the target of the direction vector.

Returns

Vector3

A Vector3 representing the direction vector from the position of the GameObject to the position of the specified GameObject.

Exceptions

ArgumentNullException

Thrown when the specified GameObject is null.

DirectionTo(GameObject, Transform)

Calculates the direction vector from the position of the GameObject to the position of the specified Transform.

```
public static Vector3 DirectionTo(this GameObject gameObject, Transform transform)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the direction vector from.

`transform` Transform

The Transform whose position is the target of the direction vector.

Returns

Vector3

A Vector3 representing the direction vector from the position of the GameObject to the position of the specified Transform.

Exceptions

ArgumentNullException

Thrown when the specified Transform is null.

DirectionTo(GameObject, Vector3)

Calculates the direction vector from the position of the GameObject to the specified point.

```
public static Vector3 DirectionTo(this GameObject gameObject, Vector3 point)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the direction vector from.

`point` Vector3

The target point to which the direction vector is calculated.

Returns

Vector3

A Vector3 representing the direction vector from the position of the GameObject to the specified point.

DistanceTo(GameObject, GameObject)

Calculates the distance between the position of the GameObject and the position of the specified GameObject.

```
public static float DistanceTo(this GameObject gameObject, GameObject other)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the distance to.

`other` GameObject

The target GameObject whose position is the target of the distance calculation.

Returns

float

The distance between the position of the GameObject and the position of the specified GameObject.

Exceptions

ArgumentNullException

Thrown when the specified GameObject is null.

DistanceTo(GameObject, Transform)

Calculates the distance between the position of the GameObject and the position of the specified Transform.

```
public static float DistanceTo(this GameObject gameObject, Transform transform)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the distance to.

`transform` Transform

The Transform whose position is the target of the distance calculation.

Returns

float

The distance between the position of the GameObject and the position of the specified Transform.

Exceptions

ArgumentNullException

Thrown when the specified Transform is null.

DistanceTo(GameObject, Vector3)

Calculates the distance between the position of the GameObject and the specified point.

```
public static float DistanceTo(this GameObject gameObject, Vector3 point)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the distance to.

`point` Vector3

The target point whose distance is calculated from the GameObject's position.

Returns

float

The distance between the position of the GameObject and the specified point.

GetComponent<T>(GameObject, Scope)

Retrieves a component of type `T` from the specified GameObject according to the provided `scope`.

```
public static T GetComponent<T>(this GameObject gameObject, Scope scope
= Scope.GameObject)
```

Parameters

`gameObject` GameObject

The GameObject from which the component is to be retrieved.

`scope` Scope

The scope of the search for the component, which can be limited to the GameObject, its children, its parent, or both its parent and children. The default value is GameObject.

Returns

T

The component of type `T` found within the specified `scope` or `null` if no such component exists.

Type Parameters

`T`

The type of the component to retrieve.

Examples

```
// Retrieve a Rigidbody component only from the GameObject itself
Rigidbody rb = gameObject.GetComponent<Rigidbody>();

// Retrieve a Rigidbody component from the children
Rigidbody rbChild = gameObject.GetComponent<Rigidbody>(Scope.Children);

// Retrieve a Rigidbody component from the parent or children
Rigidbody rbFamily = gameObject.GetComponent<Rigidbody>(Scope.ParentsAndChildren);
```

```
This example demonstrates how to retrieve a Rigidbody component from a GameObject
with different scopes of search.
```

Remarks

This method allows for a flexible component search by specifying a `scope` for the search area:

- `GameObject`: Searches only on the given GameObject.
- `Children`: Searches all children of the GameObject, not including itself.
- `Parents`: Searches the parent objects of the GameObject, moving up the hierarchy.
- `ParentsAndChildren`: First searches up the parent chain, and if no component is found, searches down through the children.

Exceptions

ArgumentOutOfRangeException

Thrown if the `scope` is not one of the enumerated values.

GetComponents<T>(GameObject, Scope)

Retrieves all components of type `T` from the specified GameObject according to the provided `scope`.

```
public static T[] GetComponents<T>(this GameObject gameObject, Scope scope
= Scope.GameObject)
```

Parameters

`gameObject` GameObject

The GameObject from which the components are to be retrieved.

`scope` Scope

The scope of the search for the components, which can be limited to the GameObject, its children, its parent, or both its parent and children. The default is GameObject.

Returns

T[]

An array of components of type `T` found within the specified `scope`.

Type Parameters

`T`

The type of the components to retrieve.

Examples

```
// Retrieve all Rigidbody components only from the GameObject itself
Rigidbody[] rigidbodies = gameObject.GetComponents<Rigidbody>();

// Retrieve all Rigidbody components from the children
Rigidbody[] childRigidbodies = gameObject.GetComponents<Rigidbody>(Scope.Children);

// Retrieve all unique Rigidbody components from both the parent and children
Rigidbody[] familyRigidbodies = gameObject.GetComponents<Rigidbody>
(Scope.ParentsAndChildren);
```

This example demonstrates how to retrieve Rigidbody components from a GameObject with different scopes of search.

Remarks

This method expands the capabilities of the standard GameObject.GetComponent<T> method by allowing the search for components to be scoped more broadly than just the current GameObject. Depending on the `scope` parameter, components can be retrieved from:

- The GameObject itself (GameObject).
- All direct and indirect children of the GameObject (Children).
- The parent and all higher ancestors of the GameObject (Parents).
- Both the parent hierarchy and the children hierarchy (ParentsAndChildren), ensuring a unique set of components by removing duplicates.

This method is particularly useful for complex GameObject hierarchies where a single GameObject may interact with multiple related components spread across different levels of the hierarchy.

Exceptions

ArgumentOutOfRangeException

Thrown if the `scope` is not one of the enumerated values.

IsInLayerMask(GameObject, LayerMask)

Determines whether the GameObject is part of the specified LayerMask.

```
public static bool IsInLayerMask(this GameObject gameObject, LayerMask layerMask)
```

Parameters

`gameObject` GameObject

The GameObject to test.

`layerMask` LayerMask

The LayerMask to check against.

Returns

bool

`true` if the gameObject's layer is included in the layerMask; otherwise, `false`.

Examples

This example demonstrates how to check if a GameObject named "Player" is in a LayerMask defined for enemies:

```
GameObject player = GameObject.Find("Player");
LayerMask enemyLayerMask = LayerMask.GetMask("Enemy");

bool isPlayerInEnemyLayer = player.IsInLayerMask(enemyLayerMask);

if (isPlayerInEnemyLayer)
{
    Debug.Log("Player is in the Enemy Layer.");
}
else
{
    Debug.Log("Player is not in the Enemy Layer.");
}
```

Remarks

This method checks if the layer to which the gameObject belongs is included in the specified layerMask. It uses bitwise operations to compare the gameObject's layer with the

layerMask.

Send<T>(GameObject, Action<T>, Scope)

Executes the provided Action<T> on the first `T` component found on the specified `gameObject` within the defined `scope`.

```
public static void Send<T>(this GameObject gameObject, Action<T> action, Scope scope
= Scope.GameObject)
```

Parameters

`gameObject` GameObject

The GameObject from which the component is to be retrieved and the action is to be executed.

`action` Action<T>

The action to execute on the retrieved component. If the component is not found, the action is not executed.

`scope` Scope

The Scope defining where to search for the component. The default scope is GameObject.

Type Parameters

`T`

The type of the component to retrieve and act upon.

Examples

The following example shows the use of `Send` to apply damage to a damageable component upon projectile impact:

```
public class Projectile : MonoBehaviour
{
    public float damage;
    public LayerMask targetLayers;
    public Event<Vector3> onProjectileImpact;
```

```
    private void OnCollisionEnter(Collision other)
    {
        if (other.gameObject.IsInLayerMask(targetLayers))
        {
            other.gameObject.Send<IDamageable>(damageable =>
damageable.ReceiveDamage(damage));
            onProjectileImpact.Invoke(transform.position);
            Destroy(gameObject);
        }
    }
}
```

This example shows how the `Send` method is used within the `Projectile` class to find and interact with an `IDamageable` component on a collision object, applying damage and triggering an event upon impact.

Remarks

This method offers a robust and more efficient alternative to Unity's `SendMessage`, which relies on string method names and lacks type safety. By directly invoking a delegate on the component, `Send` avoids the overhead and errors associated with string-based method invocation.

This method attempts to find a component of type `T` on the `gameObject` according to the specified `scope`. If the component is found, the provided `action` is executed with the component as its argument. If no such component is found, or if `action` is null, no action is executed.

This method is useful for applying operations to components when the exact presence of the component is not guaranteed, or when the operation should only be applied conditionally based on the presence of the component.

TryGetComponent<T>(GameObject, out T, Scope)

W Tries to retrieve a component of type `T` from the specified `gameObject` according to the provided `scope`, and outputs the result.

```
public static bool TryGetComponent<T>(this GameObject gameObject, out T component,
Scope scope = Scope.GameObject)
```

Parameters

`gameObject` GameObject

The GameObject from which to retrieve the component.

`component` T

When this method returns, contains the component of type `T` if found, otherwise null. This parameter is passed uninitialized.

`scope` Scope

The Scope within which to search for the component. The default is GameObject.

Returns

bool

true if a component of type `T` is found; otherwise, false.

Type Parameters

`T`

The type of the component to retrieve.

Examples

The following example demonstrates how to use the TryGetComponent<T>(GameObject, out T, Scope) method to attempt to retrieve a component of type `MeshRenderer` from a game object.

```
MeshRenderer renderer;

if (gameObject.TryGetComponent<MeshRenderer>(out renderer))
{
    Console.WriteLine("Component found!");
}
else
{
    Console.WriteLine("Component not found.");
}
```

Remarks

This method extends GameObject and utilizes GameObject.GetComponent<T> to attempt to retrieve a component of the specified type T. If the component exists within the given scope on the GameObject, it is assigned to the output parameter 'component', and the method returns true. If no such component is found, the method returns false.

VectorFrom(GameObject, GameObject)

Calculates the direction vector from the position of the specified GameObject to the position of the GameObject.

```
public static Vector3 VectorFrom(this GameObject gameObject, GameObject other)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the direction vector to.

`other` GameObject

The source GameObject whose position is the source of the direction vector.

Returns

Vector3

A Vector3 representing the direction vector from the position of the specified GameObject to the position of the GameObject.

Remarks

This method calculates the direction vector from the position of the specified GameObject to the position of the GameObject. It uses the position of the specified GameObject as the starting point and the GameObject's current position as the target. The resulting direction vector points from the position of the specified GameObject towards the position of the GameObject.

Exceptions

ArgumentNullException

Thrown when the specified GameObject is null.

VectorFrom(GameObject, Transform)

Calculates the direction vector from the specified Transform to the position of the GameObject.

```
public static Vector3 VectorFrom(this GameObject gameObject, Transform transform)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the direction vector to.

`transform` Transform

The Transform whose position is the source of the direction vector.

Returns

Vector3

A Vector3 representing the direction vector from the specified Transform to the position of the GameObject.

Remarks

This method calculates the direction vector from the specified Transform to the position of the GameObject. It uses the Transform's position as the starting point and the GameObject's current position as the target. The resulting direction vector points from the specified Transform towards the position of the GameObject.

Exceptions

ArgumentNullException

Thrown when the specified Transform is null.

VectorFrom(GameObject, Vector3)

Calculates the direction vector from the specified point to the position of the GameObject.

```
public static Vector3 VectorFrom(this GameObject gameObject, Vector3 point)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the direction vector to.

`point` Vector3

The source point from which the direction vector is calculated.

Returns

Vector3

A Vector3 representing the direction vector from the specified point to the position of the GameObject.

Remarks

This method calculates the direction vector from the specified point to the position of the GameObject. It uses the specified point as the starting point and the GameObject's current position as the target. The resulting direction vector points from the specified point towards the position of the GameObject.

VectorTo(GameObject, GameObject)

Calculates the direction vector from the position of the GameObject to the position of the specified GameObject.

```
public static Vector3 VectorTo(this GameObject gameObject, GameObject other)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the direction vector from.

`other` GameObject

The target GameObject whose position is the target of the direction vector.

Returns

Vector3

A Vector3 representing the direction vector from the position of the GameObject to the position of the specified GameObject.

Remarks

This method calculates the direction vector from the position of the GameObject to the position of the specified GameObject. It uses the GameObject's current position as the starting point and the position of the specified GameObject as the target. The resulting direction vector points from the GameObject towards the position of the specified GameObject.

Exceptions

ArgumentNullException

Thrown when the specified GameObject is null.

VectorTo(GameObject, Transform)

Calculates the direction vector from the GameObject's position to the position of the specified Transform.

```
public static Vector3 VectorTo(this GameObject gameObject, Transform transform)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the vector from.

`transform` Transform

The Transform whose position is the target of the direction vector.

Returns

Vector3

A Vector3 representing the direction vector from the GameObject's position to the position of the specified Transform.

Remarks

This method calculates the direction vector from the GameObject's position to the position of the specified Transform. It uses the GameObject's current position as the starting point and the position of the specified Transform as the target. The resulting direction vector points from the GameObject towards the position of the specified Transform.

Exceptions

ArgumentNullException

Thrown when the specified Transform is null.

VectorTo(GameObject, Vector3)

Calculates the direction vector from the GameObject's position to the specified point.

```
public static Vector3 VectorTo(this GameObject gameObject, Vector3 point)
```

Parameters

`gameObject` GameObject

The GameObject to calculate the vector from.

`point` Vector3

The target point to which the direction vector is calculated.

Returns

Vector3

A Vector3 representing the direction vector from the GameObject's position to the specified point.

Remarks

This method calculates the direction vector from the GameObject's position to the specified point. It uses the GameObject's current position as the starting point and the specified point as the target. The resulting direction vector points from the GameObject towards the specified point.

Class LifecycleEvents

Namespace: SODD.Core

```
public class LifecycleEvents : MonoBehaviour
```

Inheritance

object ← LifecycleEvents

Class Logger

Namespace: SODD.Core

Provides utility functions for logging in the Unity Editor.

```
public static class Logger
```

Inheritance

object ← Logger

Methods

LogAsset(Object, string)

Logs a message associated with a specific Unity asset in the console, including a clickable link to the asset in the editor.

```
public static void LogAsset(Object asset, string message)
```

Parameters

`asset` Object

The asset related to the message.

`message` string

The message to log.

Examples

Example of logging an asset-related message:

```
// Assuming 'exampleAsset' is a Unity asset, such as a ScriptableObject
Logger.LogAsset(exampleAsset, "This is a test message for logging purposes.");
```

This will output a log message in the Unity console with a clickable link to 'exampleAsset'.

Remarks

This method enhances debugging and logging in the Unity Editor by providing a direct link to the asset involved, making it easier to identify and access assets directly from the console log. Only available in the Unity Editor.

Class NavMeshAgentExtensions

Namespace: SODD.Core

Extends NavMeshAgent type providing utility methods.

```
public static class NavMeshAgentExtensions
```

Inheritance

object ← NavMeshAgentExtensions

Methods

CanReachDestination(NavMeshAgent)

Checks if the NavMeshAgent can reach its destination.

```
public static bool CanReachDestination(this NavMeshAgent agent)
```

Parameters

`agent` NavMeshAgent

The NavMeshAgent instance to check.

Returns

bool

True if the agent can reach its destination without any obstacles or path errors; otherwise, false if the agent's path is incomplete or blocked.

Remarks

This method checks the path status of the NavMeshAgent to determine if it can reach its destination. If the path is complete and there are no obstacles or path errors, the agent can reach its destination.

HasReachedDestination(NavMeshAgent)

Checks if the NavMeshAgent has reached its destination.

```
public static bool HasReachedDestination(this NavMeshAgent agent)
```

Parameters

`agent` NavMeshAgent

The NavMeshAgent instance to check.

Returns

bool

True if the agent has reached its destination or is very close to it; otherwise, false if the agent is still moving towards the destination.

Remarks

This method checks if the NavMeshAgent has reached its destination or is very close to it. It takes into account any pending path calculations, stopping distance, and current velocity.

Class PassiveScriptableObject

Namespace: SODD.Core

```
public abstract class PassiveScriptableObject : ScriptableObject
```

Inheritance

object ← PassiveScriptableObject

Derived

InputActionHandler<T>, ControlSchemeHandler, InputActionIconProvider, InputIconRepository, VariableRepository

Fields

reference

```
public bool reference
```

Field Value

bool

Class PassiveScriptableObjectLoader

Namespace: SODD.Core

```
public sealed class PassiveScriptableObjectLoader : MonoBehaviour
```

Inheritance

object ← PassiveScriptableObjectLoader

Fields

passiveScriptableObjects

```
public List<PassiveScriptableObject> passiveScriptableObjects
```

Field Value

List<PassiveScriptableObject>

Class PersistentScriptableObject

Namespace: SODD.Core

```
public abstract class PersistentScriptableObject : ScriptableObject
```

Inheritance

object ← PersistentScriptableObject

Derived

Variable<T>

Fields

persist

```
public bool persist
```

Field Value

bool

Class PrimitiveTypeExtensions

Namespace: SODD.Core

Extends primitive types providing utility methods.

```
public static class PrimitiveTypeExtensions
```

Inheritance

object ← PrimitiveTypeExtensions

Methods

ToBool(int)

Converts an integer value to a boolean representation.

```
public static bool ToBool(this int value)
```

Parameters

`value` int

The integer value to convert.

Returns

bool

True if the input integer value is non-zero; otherwise, false if the input integer value is 0.

Remarks

This method converts an integer value to its corresponding boolean representation: true if the input integer value is non-zero, and false if the input integer value is 0. It is a convenient way to interpret integer values as boolean conditions in certain scenarios.

ToInt(bool)

Converts a boolean value to an integer representation.

```
public static int ToInt(this bool value)
```

Parameters

`value` bool

The boolean value to convert.

Returns

int

The integer representation of the boolean value: 1 if the input value is true, and 0 if the input value is false.

Remarks

This method converts a boolean value to its corresponding integer representation: 1 for true and 0 for false. It is a convenient way to obtain an integer value when dealing with boolean conditions in certain scenarios.

Enum Scope

Namespace: SODD.Core

Specifies the scope of search for components in a GameObject.

```
public enum Scope
```

Fields

`Children = 1`

Search for the component on all children of the provided GameObject.

`GameObject = 0`

Search for the component only on the provided GameObject.

`Parents = 2`

Search for the component on the parent of the provided GameObject.

`ParentsAndChildren = 3`

Search for the component on the parent and all children of the provided GameObject.

Enum ToStringStrategy

Namespace: SODD.Core

```
public enum ToStringStrategy
```

Fields

`Capitalized = 0`

`Lowercase = 1`

`Uppercase = 2`

Class TransformExtensions

Namespace: SODD.Core

Extends Transform type providing utility methods.

```
public static class TransformExtensions
```

Inheritance

object ← TransformExtensions

Methods

DirectionFrom(Transform, GameObject)

Calculates the direction vector from the specified GameObject's position to the current transform's position.

```
public static Vector3 DirectionFrom(this Transform transform, GameObject gameObject)
```

Parameters

`transform` Transform

The current transform.

`gameObject` GameObject

The GameObject.

Returns

Vector3

The direction vector from the specified GameObject's position to the current transform's position.

DirectionFrom(Transform, Transform)

Calculates the direction vector from the specified transform's position to the current transform's position.

```
public static Vector3 DirectionFrom(this Transform transform, Transform other)
```

Parameters

`transform` Transform

The current transform.

`other` Transform

The other transform.

Returns

Vector3

The direction vector from the specified transform's position to the current transform's position.

DirectionFrom(Transform, Vector3)

Calculates the direction vector from the specified point to the current transform's position.

```
public static Vector3 DirectionFrom(this Transform transform, Vector3 point)
```

Parameters

`transform` Transform

The current transform.

`point` Vector3

The starting point.

Returns

Vector3

The direction vector from the specified point to the current transform's position.

DirectionTo(Transform, GameObject)

Calculates the direction vector from the current transform's position to a GameObject's position.

```
public static Vector3 DirectionTo(this Transform transform, GameObject gameObject)
```

Parameters

`transform` Transform

The current transform.

`gameObject` GameObject

The GameObject.

Returns

Vector3

The direction vector from the current transform's position to the GameObject's position.

DirectionTo(Transform, Transform)

Calculates the direction vector from the current transform's position to another transform's position.

```
public static Vector3 DirectionTo(this Transform transform, Transform other)
```

Parameters

`transform` Transform

The current transform.

`other` Transform

The other transform.

Returns

Vector3

The direction vector from the current transform's position to the other transform's position.

DirectionTo(Transform, Vector3)

Calculates the direction vector from the current transform's position to the specified point.

```
public static Vector3 DirectionTo(this Transform transform, Vector3 point)
```

Parameters

`transform` Transform

The current transform.

`point` Vector3

The target point.

Returns

Vector3

The direction vector from the current transform's position to the target point.

DistanceTo(Transform, GameObject)

Calculates the distance between the current transform's position and a GameObject's position.

```
public static float DistanceTo(this Transform transform, GameObject gameObject)
```

Parameters

`transform` Transform

The current transform.

`gameObject` GameObject

The GameObject.

Returns

float

The distance between the current transform's position and the GameObject's position.

DistanceTo(Transform, Transform)

Calculates the distance between the current transform's position and another transform's position.

```
public static float DistanceTo(this Transform transform, Transform other)
```

Parameters

`transform` Transform

The current transform.

`other` Transform

The other transform.

Returns

float

The distance between the current transform's position and the other transform's position.

DistanceTo(Transform, Vector3)

Calculates the distance between the current transform's position and the specified point.

```
public static float DistanceTo(this Transform transform, Vector3 point)
```

Parameters

`transform` Transform

The current transform.

`point` Vector3

The target point.

Returns

float

The distance between the current transform's position and the target point.

VectorFrom(Transform, GameObject)

Calculates the vector from the specified GameObject's position to the current transform's position.

```
public static Vector3 VectorFrom(this Transform transform, GameObject gameObject)
```

Parameters

`transform` Transform

The current transform.

`gameObject` GameObject

The GameObject.

Returns

Vector3

The vector from the specified GameObject's position to the current transform's position.

VectorFrom(Transform, Transform)

Calculates the vector from the specified transform's position to the current transform's position.

```
public static Vector3 VectorFrom(this Transform transform, Transform other)
```

Parameters

`transform` Transform

The current transform.

`other` Transform

The other transform.

Returns

Vector3

The vector from the specified transform's position to the current transform's position.

VectorFrom(Transform, Vector3)

Calculates the vector from the specified point to the current transform's position.

```
public static Vector3 VectorFrom(this Transform transform, Vector3 point)
```

Parameters

`transform` Transform

The current transform.

`point` Vector3

The starting point.

Returns

Vector3

The vector from the specified point to the current transform's position.

VectorTo(Transform, GameObject)

Calculates the vector from the current transform's position to a GameObject's position.

```
public static Vector3 VectorTo(this Transform transform, GameObject gameObject)
```

Parameters

`transform` Transform

The current transform.

`gameObject` GameObject

The GameObject.

Returns

Vector3

The vector from the current transform's position to the GameObject's position.

VectorTo(Transform, Transform)

Calculates the vector from the current transform's position to another transform's position.

```
public static Vector3 VectorTo(this Transform transform, Transform other)
```

Parameters

`transform` Transform

The current transform.

`other` Transform

The other transform.

Returns

Vector3

The vector from the current transform's position to the other transform's position.

VectorTo(Transform, Vector3)

Calculates the vector from the current transform's position to the specified point.

```
public static Vector3 VectorTo(this Transform transform, Vector3 point)
```

Parameters

`transform` Transform

The current transform.

`point` Vector3

The target point.

Returns

Vector3

The vector from the current transform's position to the target point.

Class Vector2Extensions

Namespace: SODD.Core

Extends Vector2 type providing utility methods.

```
public static class Vector2Extensions
```

Inheritance

object ← Vector2Extensions

Methods

DirectionFrom(Vector2, Vector2)

Calculates the direction vector from the specified point to the current Vector2.

```
public static Vector2 DirectionFrom(this Vector2 vector2, Vector2 point)
```

Parameters

`vector2` Vector2

The current Vector2.

`point` Vector2

The starting point.

Returns

Vector2

The direction vector from the specified point to the current Vector2.

DirectionTo(Vector2, Vector2)

Calculates the direction vector from the current Vector2 to the specified point.

```
public static Vector2 DirectionTo(this Vector2 vector2, Vector2 point)
```

Parameters

`vector2` Vector2

The current Vector2.

`point` Vector2

The target point.

Returns

Vector2

The direction vector from the current Vector2 to the target point.

DistanceTo(Vector2, Vector2)

Calculates the distance between the current Vector2 and the specified point.

```
public static float DistanceTo(this Vector2 vector2, Vector2 point)
```

Parameters

`vector2` Vector2

The current Vector2.

`point` Vector2

The target point.

Returns

float

The distance between the current Vector2 and the target point.

Rotate(Vector2, float, Vector2)

Rotates the current Vector2 by the specified angle around the specified axis.

```
public static Vector2 Rotate(this Vector2 vector, float angle, Vector2 axis)
```

Parameters

`vector` Vector2

The current Vector2.

`angle` float

The angle to rotate by.

`axis` Vector2

The axis to rotate around.

Returns

Vector2

The rotated Vector2.

VectorFrom(Vector2, Vector2)

Calculates the vector from the specified point to the current Vector2.

```
public static Vector2 VectorFrom(this Vector2 vector2, Vector2 point)
```

Parameters

`vector2` Vector2

The current Vector2.

`point` Vector2

The starting point.

Returns

Vector2

The vector from the specified point to the current Vector2.

VectorTo(Vector2, Vector2)

Calculates the vector from the current Vector2 to the specified point.

```
public static Vector2 VectorTo(this Vector2 vector2, Vector2 point)
```

Parameters

`vector2` Vector2

The current Vector2.

`point` Vector2

The target point.

Returns

Vector2

The vector from the current Vector2 to the target point.

Class Vector3Extensions

Namespace: SODD.Core

Extends Vector3 type providing utility methods.

```
public static class Vector3Extensions
```

Inheritance

object ← Vector3Extensions

Methods

DirectionFrom(Vector3, Vector3)

Calculates the direction vector from the specified point to the current Vector3.

```
public static Vector3 DirectionFrom(this Vector3 vector3, Vector3 point)
```

Parameters

`vector3` Vector3

The current Vector3.

`point` Vector3

The starting point.

Returns

Vector3

The direction vector from the specified point to the current Vector3.

DirectionTo(Vector3, Vector3)

Calculates the direction vector from the current Vector3 to the specified point.

```
public static Vector3 DirectionTo(this Vector3 vector3, Vector3 point)
```

Parameters

`vector3` Vector3

The current Vector3.

`point` Vector3

The target point.

Returns

Vector3

The direction vector from the current Vector3 to the target point.

DistanceTo(Vector3, Vector3)

Calculates the distance between the current Vector3 and the specified point.

```
public static float DistanceTo(this Vector3 vector3, Vector3 point)
```

Parameters

`vector3` Vector3

The current Vector3.

`point` Vector3

The target point.

Returns

float

The distance between the current Vector3 and the target point.

Rotate(Vector3, float, Vector3)

Rotates the current Vector3 by the specified angle around the specified axis.

```
public static Vector3 Rotate(this Vector3 vector, float angle, Vector3 axis)
```

Parameters

`vector` Vector3

The current Vector3.

`angle` float

The angle to rotate by.

`axis` Vector3

The axis to rotate around.

Returns

Vector3

The rotated Vector3.

VectorFrom(Vector3, Vector3)

Calculates the vector from the specified point to the current Vector3.

```
public static Vector3 VectorFrom(this Vector3 vector3, Vector3 point)
```

Parameters

`vector3` Vector3

The current Vector3.

`point` Vector3

The starting point.

Returns

Vector3

The vector from the specified point to the current Vector3.

VectorTo(Vector3, Vector3)

Calculates the vector from the current Vector3 to the specified point.

```
public static Vector3 VectorTo(this Vector3 vector3, Vector3 point)
```

Parameters

`vector3` Vector3

The current Vector3.

`point` Vector3

The target point.

Returns

Vector3

The vector from the current Vector3 to the target point.

Struct Void

Namespace: SODD.Core

Represents a type that signifies the absence of any value.

```
[Serializable]
public struct Void
```

Inherited Members
ValueType.Equals(object) , ValueType.GetHashCode() , ValueType.ToString()

Remarks

The Void type is used as a placeholder type parameter in events where no other data is needed. This allows for the use of generic event handling mechanisms without needing to specify a meaningful type. It is primarily used in the VoidEvent to represent an event that is triggered without any accompanying data.

Fields

Instance

Provides a singleton instance of the Void type, used to represent an empty value.

```
public static readonly Void Instance
```

Field Value

Void

Namespace SODD.Data

Classes

SerializableDictionary<TK, TV>

Structs

Range<T>

Struct Range<T>

Namespace: SODD.Data

```
[Serializable]
public struct Range<T> where T : IComparable<T>
```

Type Parameters

T

Inherited Members
ValueType.Equals(object) , ValueType.GetHashCode() , ValueType.ToString()

Constructors

Range(T, T)

```
public Range(T min, T max)
```

Parameters

min T

max T

Fields

max

```
public T max
```

Field Value

T

min

```
public T min
```

Field Value

T

Methods

IsInRange(T)

```
public bool IsInRange(T value)
```

Parameters

value T

Returns

bool

Class SerializableDictionary<TK, TV>

Namespace: SODD.Data

```
[Serializable]
public class SerializableDictionary<TK, TV> : Dictionary<TK, TV>, IDictionary<TK,
TV>, ICollection<KeyValuePair<TK, TV>>, IReadOnlyDictionary<TK, TV>,
IReadOnlyCollection<KeyValuePair<TK, TV>>, IEnumerable<KeyValuePair<TK, TV>>,
IDictionary, ICollection, IEnumerable, IDeserializationCallback, ISerializable,
ISerializationCallbackReceiver
```

Type Parameters

TK

TV

Inheritance

object ← Dictionary<TK, TV> ← SerializableDictionary<TK, TV>

Implements

IDictionary<TK, TV>, ICollection<KeyValuePair<TK, TV>>, IReadOnlyDictionary<TK, TV>, IReadOnlyCollection<KeyValuePair<TK, TV>>, IEnumerable<KeyValuePair<TK, TV>>, IDictionary, ICollection, IEnumerable, IDeserializationCallback, ISerializable, ISerializationCallbackReceiver

Inherited Members

Dictionary<TK, TV>.Add(TK, TV) , Dictionary<TK, TV>.Clear() , Dictionary<TK, TV>.ContainsKey(TK) , Dictionary<TK, TV>.ContainsValue(TV) , Dictionary<TK, TV>.EnsureCapacity(int) , Dictionary<TK, TV>.GetEnumerator() , Dictionary<TK, TV>.OnDeserialization(object) , Dictionary<TK, TV>.Remove(TK) , Dictionary<TK, TV>.Remove(TK, out TV) , Dictionary<TK, TV>.TrimExcess() , Dictionary<TK, TV>.TrimExcess(int) , Dictionary<TK, TV>.TryAdd(TK, TV) , Dictionary<TK, TV>.TryGetValue(TK, out TV) , Dictionary<TK, TV>.Comparer , Dictionary<TK, TV>.Count , Dictionary<TK, TV>.this[TK] , Dictionary<TK, TV>.Keys , Dictionary<TK, TV>.Values

Extension Methods

EnumerableExtensions.ForEach<T>(IEnumerable<T>, Action<T>) , EnumerableExtensions.IsEmpty<T>(IEnumerable<T>) ,

EnumerableExtensions.TryFind<T>(IEnumerable<T>, Func<T, bool>, out T) ,
EnumerableExtensions.WrapAround<T>(IEnumerable<T>)

Methods

OnAfterDeserialize()

```
public void OnAfterDeserialize()
```

OnBeforeSerialize()

```
public void OnBeforeSerialize()
```

Namespace SODD.Events

Classes

BoolEvent
: Represents a scriptable event that carries a boolean payload.

Event<T>
: Provides a generic base class for scriptable event implementations, enabling events to be defined with specific data types as payloads.

FloatEvent
: Represents a scriptable event that carries a float payload.

GameObjectEvent
: Represents a scriptable event that carries a GameObject payload.

GenericEvent<T>
: Represents a generic event that can be listened to and invoked.

IntEvent
: Represents a scriptable event that carries an integer payload.

StringEvent
: Represents a scriptable event that carries a string payload.

Vector2Event
: Represents a scriptable event that carries a Vector2 payload.

Vector3Event
: Represents a scriptable event that carries a Vector3 payload.

VoidEvent
: Represents a scriptable event that carries no payload.

Interfaces

IEvent<T>
: Defines a generic interface for events that can have listeners added or removed, and be invoked with a payload of type `T`.

IListenableEvent<T>
: Defines a generic interface for events with payloads of type `T` that can have listeners added or removed.

Class BoolEvent

Namespace: SODD.Events

Represents a scriptable event that carries a boolean payload.

```
public sealed class BoolEvent : Event<bool>, IEvent<bool>, IListenableEvent<bool>
```

Inheritance

object ← Event<bool> ← BoolEvent

Implements

IEvent<bool>, IListenableEvent<bool>

Inherited Members

Event<bool>.AddListener(Action<bool>) , Event<bool>.RemoveListener(Action<bool>) , Event<bool>.Invoke(bool)

Examples

Below is an example demonstrating how to use a BoolEvent in a script to handle player actions like crouching:

```
public class BoolEventExample : MonoBehaviour
{
    public BoolEvent onCrouch; // Assign the event through the Unity Editor.

    private void OnEnable()
    {
        onCrouch.AddListener(OnCrouch);
    }

    private void OnDisable()
    {
        onCrouch.RemoveListener(OnCrouch);
    }

    private void OnCrouch(bool isCrouching)
    {
        if (isCrouching)
        {
            // Set movement speed to crouching speed
```

```
        }
        else
        {
            // Set movement speed to normal speed
        }
    }
}
```

This example illustrates how the `onCrouch` event can be triggered by various game mechanisms such as player input or specific gameplay triggers like a tutorial session. Importantly, the script is indifferent regarding who triggers the event, focusing solely on responding to the event itself.

Remarks

This class extends the generic Event<T> class, specifying `bool` as the type parameter. It is used to create events that need to communicate a boolean value, such as toggling a state, confirming a condition, or triggering binary decisions.

Use cases might include signaling game state changes, user interactions that have two states (like on/off switches), or as a response to certain player actions in a game.

This class can be instantiated as a ScriptableObject asset directly from the Unity Editor.

Class Event<T>

Namespace: SODD.Events

Provides a generic base class for scriptable event implementations, enabling events to be defined with specific data types as payloads.

```
public abstract class Event<T> : ScriptableObject, IEvent<T>, IListenableEvent<T>
```

Type Parameters

T

The type of the event payload.

Inheritance

object ← Event<T>

Implements

IEvent<T>, IListenableEvent<T>

Derived

BoolEvent, FloatEvent, GameObjectEvent, IntEvent, StringEvent, Vector2Event, Vector3Event, VoidEvent

Examples

Definition of a string event implementation:

```
[CreateAssetMenu(menuName = "My Events/String Event", fileName
= nameof(StringEvent))]
public class StringEvent : Event<string> {}
```

Remarks

This abstract class serves as the base of the ScriptableObject-based event system provided by the SODD Framework. It allows developers to create custom events that are reusable, and loosely coupled, enhancing the modularity and flexibility of game architecture.

It implements the IEvent<T> interface, which includes methods to add or remove listeners, and to invoke the event. This class can be extended to define events with any valid Unity

type as a payload, such as `int`, `string`, `GameObject`, etc.

The methods AddListener(Action<T>), RemoveListener(Action<T>), and Invoke(T) provided by this class should suffice for basic event handling scenarios. Developers can override these methods in derived classes to tailor event behavior to specific needs.

To create custom scriptable events, inherit from this class and specify the payload type `T`.

Fields

GenericEvent

```
protected readonly GenericEvent<T> GenericEvent
```

Field Value

GenericEvent<T>

Methods

AddListener(Action<T>)

Subscribes a listener to the event, allowing it to be notified when the event is invoked.

```
public void AddListener(Action<T> listener)
```

Parameters

`listener` Action<T>

The listener to add. It is a method that matches the signature of the Action<T> delegate, accepting a single parameter of type `T`.

Invoke(T)

Triggers the event, notifying all subscribed listeners and passing the specified payload to them.

```
public void Invoke(T payload)
```

Parameters

`payload` T

The payload to pass to the listeners.

RemoveListener(Action<T>)

Unsubscribes a previously added listener from the event, preventing it from being notified when the event is invoked.

```
public void RemoveListener(Action<T> listener)
```

Parameters

`listener` Action<T>

The listener to remove. It must be the same instance that was previously added with AddListener(Action<T>).

Class FloatEvent

Namespace: SODD.Events

Represents a scriptable event that carries a float payload.

```
public sealed class FloatEvent : Event<float>, IEvent<float>,
IListenableEvent<float>
```

Inheritance

object ← Event<float> ← FloatEvent

Implements

IEvent<float>, IListenableEvent<float>

Inherited Members

Event<float>.AddListener(Action<float>) , Event<float>.RemoveListener(Action<float>) , Event<float>.Invoke(float)

Examples

Below is an example demonstrating how to define and use a FloatEvent to manage game completion percentage:

```
public class GamePercentageManager : MonoBehaviour
{
    public FloatEvent onCompletionIncreased; // Assign this through the
Unity Editor.

    private float _completionPercentage = 0f;

    private void OnEnable()
    {
        onCompletionIncreased.AddListener(OnCompletionIncreased);
    }

    private void OnDisable()
    {
        onCompletionIncreased.RemoveListener(OnCompletionIncreased);
    }

    private void OnCompletionIncreased(float percentage)
```

```
    {
        _completionPercentage += percentage;
        Debug.Log("Completion percentage increased to: " + _completionPercentage
+ "%");
    }
}
```

This example shows how `onCompletionIncreased` can be used to incrementally update the game's completion percentage as players progress through levels or achieve specific milestones. The event system ensures that the game logic related to tracking completion is decoupled from the actions that trigger progress.

Remarks

This class extends the generic Event<T> class, specifying `float` as the type parameter. It is commonly used to handle numerical data that requires precision beyond integers, such as percentages, time durations, distances, or health values.

Typical use cases include adjusting player speed, modifying game timers, changing health bars, or any scenario where a change in a floating-point value needs to be communicated across different game components.

This class can be created as a ScriptableObject asset directly from the Unity Editor.

Class GameObjectEvent

Namespace: SODD.Events

Represents a scriptable event that carries a GameObject payload.

```
public sealed class GameObjectEvent : Event<GameObject>, IEvent<GameObject>,
IListenableEvent<GameObject>
```

Inheritance

object ← Event<GameObject> ← GameObjectEvent

Implements

IEvent<GameObject>, IListenableEvent<GameObject>

Inherited Members

Event<GameObject>.AddListener(Action<GameObject>) ,
Event<GameObject>.RemoveListener(Action<GameObject>) ,
Event<GameObject>.Invoke(GameObject)

Examples

Below is an example demonstrating how to define and use a GameObjectEvent to manage an inventory system in a game:

```
public class InventoryManager : MonoBehaviour
{
    public GameObjectEvent onAddInventoryItem; // Assign this through the
Unity Editor.
    public GameObjectEvent onRemoveInventoryItem; // Assign this through the
Unity Editor.

    private List<GameObject> _inventory = new();

    private void OnEnable()
    {
        onAddInventoryItem.AddListener(OnAddInventoryItem);
        onRemoveInventoryItem.AddListener(OnRemoveInventoryItem);
    }

    private void OnDisable()
    {
```

```
        onAddInventoryItem.RemoveListener(OnAddInventoryItem);
        onRemoveInventoryItem.RemoveListener(OnRemoveInventoryItem);
    }

    private void OnAddInventoryItem(GameObject item)
    {
        _inventory.Add(item);
    }

    private void OnRemoveInventoryItem(GameObject item)
    {
        _inventory.Remove(item);
    }
}
```

This example illustrates how the `onAddInventoryItem` and `onRemoveInventoryItem` events can be used to manage the items in the player's inventory, enabling the InventoryManager to react to changes without directly interacting with the game objects that trigger these changes.

Remarks

This class extends the generic Event<T> class, specifying `GameObject` as the type parameter. It is designed to handle events that involve GameObjects, such as spawning, transformations, or inventory management.

Typical use cases include adding or removing items from an inventory, activating or deactivating game objects, or any scenario where GameObjects are manipulated dynamically during gameplay.

This class can be created as a ScriptableObject asset directly from the Unity Editor.

Class GenericEvent<T>

Namespace: SODD.Events

Represents a generic event that can be listened to and invoked.

```
public sealed class GenericEvent<T> : IEvent<T>, IListenableEvent<T>
```

Type Parameters

`T`

The type of the event payload.

Inheritance
object ← GenericEvent<T>

Implements
IEvent<T>, IListenableEvent<T>

Methods

AddListener(Action<T>)

Adds a listener to the event.

```
public void AddListener(Action<T> listener)
```

Parameters

`listener` Action<T>

The listener to add.

Remarks

This method adds a listener to the event. The listener will be triggered whenever the event is invoked.

Invoke(T)

Invokes the event and notifies all registered listeners with the specified payload.

```
public void Invoke(T payload)
```

Parameters

`payload` T

The payload to pass to the listeners.

RemoveListener(Action<T>)

Removes the specified listener from the event.

```
public void RemoveListener(Action<T> listener)
```

Parameters

`listener` Action<T>

The listener to be removed.

Interface IEvent<T>

Namespace: SODD.Events

Defines a generic interface for events that can have listeners added or removed, and be invoked with a payload of type `T`.

```
public interface IEvent<T> : IListenableEvent<T>
```

Type Parameters

`T`

The type of the event payload. This type parameter specifies the data type that listeners will receive when the event is invoked.

Inherited Members
IListenableEvent<T>.AddListener(Action<T>) ,
IListenableEvent<T>.RemoveListener(Action<T>)

Examples

Example of defining a new event and subscribing a listener:

```
// Define a new event with a string payload
public class MyStringEvent : IEvent<string> {
    private event Action<string> listeners;

    public void AddListener(Action<string> listener) => listeners += listener;

    public void RemoveListener(Action<string> listener) => listeners -= listener;

    public void Invoke(string payload) => listeners?.Invoke(payload);
}

// Create an instance of the event
MyStringEvent myEvent = new MyStringEvent();

// Define a listener method
void MyEventListener(string message) {
    Debug.Log(message);
}
```

```
// Subscribe the listener to the event
myEvent.AddListener(MyEventListener);

// Invoke the event
myEvent.Invoke("Hello, World!");
```

This example will print "Hello, World!" to the console.

Remarks

The IEvent<T> interface serves as the foundation of the loosely coupled event system based on ScriptableObjects provided by the SODD Framework, from which all scriptable events implement.

Listeners can subscribe to the event using AddListener(Action<T>) and unsubscribe using RemoveListener(Action<T>). The event can be triggered using the Invoke(T) method, which notifies all subscribed listeners, passing the specified payload to them.

Usage of this interface promotes a pattern where components can react to specific game events, enhancing modularity and flexibility of the game's architecture.

Methods

Invoke(T)

Triggers the event, notifying all subscribed listeners and passing the specified payload to them.

```
void Invoke(T payload)
```

Parameters

`payload` T

The payload to pass to the listeners.

Interface IListenableEvent<T>

Namespace: SODD.Events

Defines a generic interface for events with payloads of type `T` that can have listeners added or removed.

```
public interface IListenableEvent<out T>
```

Type Parameters

`T`

The type of the event payload. This type parameter specifies the data type that listeners will receive when the event is invoked.

Examples

Implementing IListenableEvent<T> for a custom event system:

```
public class HealthChangedEvent : IListenableEvent<int>
{
    private event Action<int> _onHealthChanged;

    public void AddListener(Action<int> listener)
    {
        _onHealthChanged += listener;
    }

    public void RemoveListener(Action<int> listener)
    {
        _onHealthChanged -= listener;
    }

    private void Invoke(int newHealth)
    {
        _onHealthChanged?.Invoke(newHealth);
    }
}
```

In this example, `HealthChangedEvent` uses an internal event delegate to manage subscribing and unsubscribing of listeners, and provides a private method `Invoke` to invoke all

subscribed listeners with the current health value.

Remarks

The IListenableEvent<T> interface serves as the foundation for scriptable events and scriptable variables.

Listeners can subscribe to the event using AddListener(Action<T>) and unsubscribe using RemoveListener(Action<T>). When the event is internally triggered, it notifies all subscribed listeners, passing the specified payload to them.

Methods

AddListener(Action<T>)

Subscribes a listener to the event, allowing it to be notified when the event is invoked.

```
void AddListener(Action<out T> listener)
```

Parameters

`listener` Action<T>

The listener to add. It is a method that matches the signature of the Action<T> delegate, accepting a single parameter of type `T`.

RemoveListener(Action<T>)

Unsubscribes a previously added listener from the event, preventing it from being notified when the event is invoked.

```
void RemoveListener(Action<out T> listener)
```

Parameters

`listener` Action<T>

The listener to remove. It must be the same instance that was previously added with AddListener(Action<T>).

Class IntEvent

Namespace: SODD.Events

Represents a scriptable event that carries an integer payload.

```
public sealed class IntEvent : Event<int>, IEvent<int>, IListenableEvent<int>
```

Inheritance

object ← Event<int> ← IntEvent

Implements

IEvent<int>, IListenableEvent<int>

Inherited Members

Event<int>.AddListener(Action<int>) , Event<int>.RemoveListener(Action<int>) , Event<int>.Invoke(int)

Examples

Below is an example demonstrating how to define and use an IntEvent to track changes in a game score:

```
public class ScoreManager : MonoBehaviour
{
    public IntEvent onUpdateScore; // Assign this through the Unity Editor.

    private int _score;

    private void OnEnable()
    {
        onScoreChanged.AddListener(OnUpdateScore);
    }

    private void OnDisable()
    {
        onScoreChanged.RemoveListener(OnUpdateScore);
    }

    private void OnUpdateScore(int increment)
    {
        _score += increment;
```

```
    }
}
```

This example shows a score manager that tracks and updates the player's score based on increments received through an event. Multiple components in the game, such as enemies when slain or coins when collected, can trigger this event to notify the need to increase the score without directly referencing the score manager.

Remarks

This class extends the generic Event<T> class, specifying `int` as the type parameter. It is commonly used to communicate numeric values, such as scores, health points, or other quantifiable game mechanics.

Typical use cases include signaling changes in player scores, enemy health, or counting game items. This class enables events to be triggered with integer data, facilitating interaction between different components of the game without tight coupling.

This class can be instantiated as a ScriptableObject asset directly from the Unity Editor.

Class StringEvent

Namespace: SODD.Events

Represents a scriptable event that carries a string payload.

```
public sealed class StringEvent : Event<string>, IEvent<string>,
IListenableEvent<string>
```

Inheritance

object ← Event<string> ← StringEvent

Implements

IEvent<string>, IListenableEvent<string>

Inherited Members

Event<string>.AddListener(Action<string>) ,
Event<string>.RemoveListener(Action<string>) , Event<string>.Invoke(string)

Examples

Below is an example demonstrating how to define and use a StringEvent for displaying messages in a game's UI:

```
public class UIManager : MonoBehaviour
{
    public StringEvent onDisplayMessage; // Assign this through the Unity Editor.
    public UnityEngine.UI.Text messageText; // Assign this UI text element in the
Unity Editor.

    private void OnEnable()
    {
        onDisplayMessage.AddListener(DisplayMessage);
    }

    private void OnDisable()
    {
        onDisplayMessage.RemoveListener(DisplayMessage);
    }

    private void DisplayMessage(string message)
    {
```

```
            messageText.text = message;
        }
    }
```

This example shows how the `onDisplayMessage` event can be used to update a UI text element whenever a new message needs to be displayed. This approach decouples the UI update logic from the rest of the game logic, enhancing modularity and maintainability.

Remarks

This class extends the generic Event<T> class, specifying `string` as the type parameter. It is particularly useful for handling textual data within the game, such as user input, notifications, or dynamic text updates.

Typical use cases include sending messages between systems, updating UI text elements, or logging debug information.

This class can be created as a ScriptableObject asset directly from the Unity Editor.

Class Vector2Event

Namespace: SODD.Events

Represents a scriptable event that carries a Vector2 payload.

```
public sealed class Vector2Event : Event<Vector2>, IEvent<Vector2>,
IListenableEvent<Vector2>
```

Inheritance

object ← Event<Vector2> ← Vector2Event

Implements

IEvent<Vector2>, IListenableEvent<Vector2>

Inherited Members

Event<Vector2>.AddListener(Action<Vector2>) ,
Event<Vector2>.RemoveListener(Action<Vector2>) , Event<Vector2>.Invoke(Vector2)

Examples

Below is an example demonstrating how to use a Vector2Event for capturing and handling 2D player input in a game:

```
using UnityEngine;
using SODD.Events;

public class PlayerController : MonoBehaviour
{
    public Vector2Event onPlayerMove; // Assign this through the Unity Editor.

    private Vector2 _direction;

    private void OnEnable()
    {
        onPlayerMove.AddListener(OnPlayerMove);
    }

    private void OnDisable()
    {
        onPlayerMove.RemoveListener(OnPlayerMove);
    }
```

```
    private void FixedUpdate()
    {
        // Move player based on _direction
    }

    private void OnPlayerMove(Vector2 direction)
    {
        _direction = direction;
    }
}
```

This example shows how the `onPlayerMove` event can be used to read and respond to 2D movement inputs from the player. This method decouples input gathering from movement logic, allowing for more modular and easily maintainable code.

Remarks

This class extends the generic Event<T> class, specifying `Vector2` as the type parameter. It is particularly useful for scenarios involving 2D vectors, such as player movement, touch inputs, or any other two-dimensional data transmission.

Typical use cases include capturing player directional inputs, tracking touch gestures, or sending coordinates for game elements to react to.

This class can be created as a ScriptableObject asset directly from the Unity Editor.

Class Vector3Event

Namespace: SODD.Events

Represents a scriptable event that carries a Vector3 payload.

```
public sealed class Vector3Event : Event<Vector3>, IEvent<Vector3>,
IListenableEvent<Vector3>
```

Inheritance

object ← Event<Vector3> ← Vector3Event

Implements

IEvent<Vector3>, IListenableEvent<Vector3>

Inherited Members

Event<Vector3>.AddListener(Action<Vector3>) ,
Event<Vector3>.RemoveListener(Action<Vector3>) , Event<Vector3>.Invoke(Vector3)

Examples

Below is an example demonstrating how to define and use a Vector3Event to manage particle spawning at specific 3D coordinates:

```
public class ParticleSpawner : MonoBehaviour
{
    public Vector3Event onSpawnParticles; // Assign this through the Unity Editor.
    public ParticleSystem particleSystem; // Assign your Particle System through the
Unity Editor.

    private void OnEnable()
    {
        onSpawnParticles.AddListener(SpawnParticles);
    }

    private void OnDisable()
    {
        onSpawnParticles.RemoveListener(SpawnParticles);
    }

    private void SpawnParticles(Vector3 position)
    {
```

```
        particleSystem.transform.position = position;
        particleSystem.Emit(10); // Emit 10 particles at the given location
    }
}
```

This example shows how the `onSpawnParticles` event can be used to spawn particles at specific 3D coordinates received through the event, allowing for dynamic effects that are decoupled from the logic that triggers these effects.

Remarks

This class extends the generic Event<T> class, specifying `Vector3` as the type parameter. It is ideal for scenarios involving three-dimensional vectors, such as spatial coordinates for moving objects, spawning items, or effects in 3D space.

Typical use cases include determining spawn locations for characters or objects, tracking movement paths, or coordinating complex particle effects in response to game actions.

This class can be created as a ScriptableObject asset directly from the Unity Editor.

Class VoidEvent

Namespace: SODD.Events

Represents a scriptable event that carries no payload.

```
public sealed class VoidEvent : Event<Void>, IEvent<Void>, IListenableEvent<Void>
```

Inheritance

object ← Event<Void> ← VoidEvent

Implements

IEvent<Void>, IListenableEvent<Void>

Inherited Members

Event<Void>.AddListener(Action<Void>) , Event<Void>.RemoveListener(Action<Void>) , Event<Void>.Invoke(Void)

Examples

Below is an example demonstrating how to define and use a VoidEvent to trigger a level reset in a game:

```
public class LevelManager : MonoBehaviour
{
    public VoidEvent onLevelReset; // Assign this through the Unity Editor.

    private void OnEnable()
    {
        onLevelReset.AddListener(ResetLevel);
    }

    private void OnDisable()
    {
        onLevelReset.RemoveListener(ResetLevel);
    }

    private void ResetLevel()
    {
        // Add logic here to reset the level
    }
}
```

This example shows how the `onLevelReset` event can be used to initiate a level reset without needing to pass any specific data. This ensures that the action to reset can be easily triggered from multiple parts of the game without any dependencies on the event sender.

Remarks

This class extends the generic Event<T> class, specifying `Void` as the type parameter. It is ideal for situations where the occurrence of the event is important, but no data needs to be communicated.

This class can be created as a ScriptableObject asset directly from the Unity Editor.

Methods

Invoke()

```
public void Invoke()
```

Namespace SODD.Input

Classes

ControlSchemeHandler

InputActionIconProvider

Class ControlSchemeHandler

Namespace: SODD.Input

```
public class ControlSchemeHandler : PassiveScriptableObject
```

Inheritance

object ← PassiveScriptableObject ← ControlSchemeHandler

Inherited Members

PassiveScriptableObject.reference

Class InputActionIconProvider

Namespace: SODD.Input

```
public class InputActionIconProvider : PassiveScriptableObject
```

Inheritance

object ← PassiveScriptableObject ← InputActionIconProvider

Inherited Members

PassiveScriptableObject.reference

Namespace SODD.Input.ActionHandlers

Classes

BoolInputActionHandler
: A ScriptableObject that handles input actions passing boolean data.

FloatInputActionHandler
: A ScriptableObject that handles input actions passing float data.

InputActionHandler<T>
: Provides a base class for handling input actions and converting them into scriptable events.

Vector2InputActionHandler
: A ScriptableObject that handles input actions producing Vector2 values.

Vector3InputActionHandler
: Handles input actions that provide Vector3 data.

VoidInputActionHandler
: A ScriptableObject that handles input actions that do not pass data.

Class BoolInputActionHandler

Namespace: SODD.Input.ActionHandlers

A ScriptableObject that handles input actions passing boolean data.

```
public sealed class BoolInputActionHandler : InputActionHandler<bool>
```

Inheritance

object ← PassiveScriptableObject ← InputActionHandler<bool> ← BoolInputActionHandler

Inherited Members

PassiveScriptableObject.reference

Remarks

The `BoolInputActionHandler` is designed to process boolean input actions, providing a structured and reusable way to react to button press or toggle events within Unity's Input System. This concrete implementation of InputActionHandler<T> offers specific overrides for the action started, performed, and canceled events, translating input action data to boolean values.

Methods

OnActionCanceled(CallbackContext)

Called when the input action is canceled.

```
protected override void OnActionCanceled(InputAction.CallbackContext context)
```

Parameters

`context` CallbackContext

Context of the input action callback.

OnActionPerformed(CallbackContext)

Called when the input action is performed.

```
protected override void OnActionPerformed(InputAction.CallbackContext context)
```

Parameters

`context` CallbackContext

Context of the input action callback.

OnActionStarted(CallbackContext)

Called when the input action starts.

```
protected override void OnActionStarted(InputAction.CallbackContext context)
```

Parameters

`context` CallbackContext

Context of the input action callback.

Class FloatInputActionHandler

Namespace: SODD.Input.ActionHandlers

A ScriptableObject that handles input actions passing float data.

```
public sealed class FloatInputActionHandler : InputActionHandler<float>
```

Inheritance

object ← PassiveScriptableObject ← InputActionHandler<float> ← FloatInputActionHandler

Inherited Members

PassiveScriptableObject.reference

Remarks

The `FloatInputActionHandler` specializes in processing input actions that output float data, such as analog stick movement, trigger pressure, or any other input mechanism that generates a continuous range of values rather than discrete on/off signals.

This handler is ideal for scenarios where the precise value of an input, within a defined range, influences the game's behavior. Examples include adjusting the speed of a character's movement based on analog stick deflection or modulating audio volume.

Class InputActionHandler<T>

Namespace: SODD.Input.ActionHandlers

Provides a base class for handling input actions and converting them into scriptable events.

```
public abstract class InputActionHandler<T> : PassiveScriptableObject where T
: struct
```

Type Parameters

T

The data type of the input value, defined by the input action being handled.

Inheritance

object ← PassiveScriptableObject ← InputActionHandler<T>

Derived

BoolInputActionHandler, FloatInputActionHandler, Vector2InputActionHandler, Vector3InputActionHandler, VoidInputActionHandler

Inherited Members

PassiveScriptableObject.reference

Examples

Below is an example of how to create a concrete implementation of `InputActionHandler<T>` for handling Vector2 input actions, commonly used for 2D movement:

```
[CreateAssetMenu(menuName = "InputHandlers/Vector2InputActionHandler")]
public class Vector2InputActionHandler : InputActionHandler<Vector2>
{
    // Override methods if needed for custom behavior upon input events.
}
```

In the Unity Editor, create an instance of this ScriptableObject and assign the input action reference along with any event subscribers to handle the input action events.

Remarks

This abstract class serves as the foundation for all input action handler implementations within the SODD Framework. It allows for the decoupling of input handling logic from game objects and components by translating the events provided by Unity's Input System into the SODD Framework's scriptable event system.

This class listens to three phases of an input action—started, performed, and canceled—and invokes the corresponding scriptable events.

To create custom input action handlers inherit from this class and specifying the appropriate type for `T` based on your input action's expected output. Then, assign input action references and connect event listeners to handle specific input events.

Fields

inputActionReference

Reference to the InputAction to be handled.

```
protected InputActionReference inputActionReference
```

Field Value

InputActionReference

onActionCanceled

Event triggered when the input action is canceled.

```
protected Event<T> onActionCanceled
```

Field Value

Event<T>

onActionPerformed

Event triggered when the input action is performed.

```
protected Event<T> onActionPerformed
```

Field Value

Event<T>

onActionStarted

Event triggered when the input action starts.

```
protected Event<T> onActionStarted
```

Field Value

Event<T>

targetVariable

```
protected Variable<T> targetVariable
```

Field Value

Variable<T>

Methods

OnActionCanceled(CallbackContext)

Called when the input action is canceled.

```
protected virtual void OnActionCanceled(InputAction.CallbackContext context)
```

Parameters

`context` CallbackContext

Context of the input action callback.

OnActionPerformed(CallbackContext)

Called when the input action is performed.

```
protected virtual void OnActionPerformed(InputAction.CallbackContext context)
```

Parameters

`context` CallbackContext

Context of the input action callback.

OnActionStarted(CallbackContext)

Called when the input action starts.

```
protected virtual void OnActionStarted(InputAction.CallbackContext context)
```

Parameters

`context` CallbackContext

Context of the input action callback.

See Also

Event<T>

Class Vector2InputActionHandler

Namespace: SODD.Input.ActionHandlers

A ScriptableObject that handles input actions producing Vector2 values.

```
public sealed class Vector2InputActionHandler : InputActionHandler<Vector2>
```

Inheritance

object ← PassiveScriptableObject ← InputActionHandler<Vector2> ← Vector2InputActionHandler

Inherited Members

PassiveScriptableObject.reference

Remarks

The `Vector2InputActionHandler` is optimized for input actions where the output is a two-dimensional vector, such as directional inputs from a joystick or touchpad. Typical use cases include controlling character movement, camera panning, or any scenario requiring directional input.

Class Vector3InputActionHandler

Namespace: SODD.Input.ActionHandlers

Handles input actions that provide Vector3 data.

```
public sealed class Vector3InputActionHandler : InputActionHandler<Vector3>
```

Inheritance

object ← PassiveScriptableObject ← InputActionHandler<Vector3> ← Vector3InputActionHandler

Inherited Members

PassiveScriptableObject.reference

Remarks

This class extends InputActionHandler<T> to specifically handle input actions that output Vector3 data, such as 3D spatial movements or directional inputs. It is ideal for applications requiring precise control over 3D space, such as camera controls, character movement, or object manipulation. Subscribers can listen to the events raised by this handler to react to the start, performance, or cancellation of the input action with Vector3 data.

Class VoidInputActionHandler

Namespace: SODD.Input.ActionHandlers

A ScriptableObject that handles input actions that do not pass data.

```
public sealed class VoidInputActionHandler : InputActionHandler<Void>
```

Inheritance

object ← PassiveScriptableObject ← InputActionHandler<Void> ← VoidInputActionHandler

Inherited Members

PassiveScriptableObject.reference

Remarks

The `VoidInputActionHandler` is designed for input actions where the action itself is significant, but the action's data is not. This is common in scenarios such as button presses where the timing and occurrence of the action are important, but the action does not carry additional data (e.g., a jump or confirm button press).

This handler triggers events for the started, performed, and canceled phases of an input action, using a Void type as payload to signify the absence of specific data. This approach maintains consistency with the `InputActionHandler` system while accommodating actions that do not require data.

Methods

OnActionCanceled(CallbackContext)

Called when the input action is canceled.

```
protected override void OnActionCanceled(InputAction.CallbackContext context)
```

Parameters

`context` CallbackContext

Context of the input action callback.

OnActionPerformed(CallbackContext)

Called when the input action is performed.

```
protected override void OnActionPerformed(InputAction.CallbackContext context)
```

Parameters

`context` CallbackContext

Context of the input action callback.

OnActionStarted(CallbackContext)

Called when the input action starts.

```
protected override void OnActionStarted(InputAction.CallbackContext context)
```

Parameters

`context` CallbackContext

Context of the input action callback.

Namespace SODD.Listeners

Classes

BoolEventListener
: Represents an event listener for boolean events.

EventListener<T>
: Represents an abstract component that listens for events of a specific type `T` and triggers UnityEvents in response.

FloatEventListener
: Represents an event listener for float events.

GameObjectEventListener
: Represents an event listener for GameObject events.

IntEventListener
: Represents an event listener for integer events.

StringEventListener
: Represents an event listener for string events.

Vector2EventListener
: Represents an event listener for Vector2 events.

Vector3EventListener
: Represents an event listener for Vector3 events.

VoidEventListener
: Represents an event listener for void events.

Interfaces

IEventListener<T>
: Represents a listener for an event of type T.

Class BoolEventListener

Namespace: SODD.Listeners

Represents an event listener for boolean events.

```
public class BoolEventListener : EventListener<bool>, IEventListener<bool>
```

Inheritance

object ← EventListener<bool> ← BoolEventListener

Implements

IEventListener<bool>

Inherited Members

EventListener<bool>.targetEvent , EventListener<bool>.onEventInvoked , EventListener<bool>.StartListening(IEvent<bool>) , EventListener<bool>.StopListening(IEvent<bool>) , EventListener<bool>.OnEventInvoked(bool)

Class EventListener<T>

Namespace: SODD.Listeners

Represents an abstract component that listens for events of a specific type `T` and triggers UnityEvents in response.

```
public abstract class EventListener<T> : MonoBehaviour, IEventListener<T>
```

Type Parameters

`T`

The type of the event payload.

Inheritance

object ← EventListener<T>

Implements

IEventListener<T>

Derived

BoolEventListener, FloatEventListener, GameObjectEventListener, IntEventListener, StringEventListener, Vector2EventListener, Vector3EventListener, VoidEventListener

Examples

A StringEvent Listener implementation would look like this:

```
public class StringEventListener : EventListener<string> {}
```

Remarks

This abstract class provides a foundation for creating scriptable event listeners. It implements the IEventListener<T> interface and inherits the MonoBehaviour class, simplifying the process of subscribing to and unsubscribing from events based on the MonoBehaviour's lifecycle, ensuring that listeners are only active when the MonoBehaviour is enabled.

To create a custom event listener, inherit from this class and specify the payload type `T`, it can be any valid Unity type (e.g., int, string, GameObject, etc.). In the Unity Editor, attach

your new custom event listener component to a GameObject, assign the specific scriptable event to listen to and configure the response through the exposed UnityEvent onEvent Invoked.

Fields

onEventInvoked

```
protected UnityEvent<T> onEventInvoked
```

Field Value

UnityEvent<T>

targetEvent

```
protected Event<T> targetEvent
```

Field Value

Event<T>

Methods

OnEventInvoked(T)

Called when the event is invoked, triggering the assigned UnityEvent.

```
public void OnEventInvoked(T payload)
```

Parameters

`payload` T

The payload passed to the event.

StartListening(IEvent<T>)

Subscribes to the target event.

```
public void StartListening(IEvent<T> @event)
```

Parameters

`event` IEvent<T>

The event to start listening to.

StopListening(IEvent<T>)

Unsubscribes from the target event.

```
public void StopListening(IEvent<T> @event)
```

Parameters

`event` IEvent<T>

The event to stop listening to.

See Also

IEventListener<T>
Event<T>
IEvent<T>

Class FloatEventListener

Namespace: SODD.Listeners

Represents an event listener for float events.

```
public class FloatEventListener : EventListener<float>, IEventListener<float>
```

Inheritance

object ← EventListener<float> ← FloatEventListener

Implements

IEventListener<float>

Inherited Members

EventListener<float>.targetEvent , EventListener<float>.onEventInvoked , EventListener<float>.StartListening(IEvent<float>) , EventListener<float>.StopListening(IEvent<float>) , EventListener<float>.OnEventInvoked(float)

Class GameObjectEventListener

Namespace: SODD.Listeners

Represents an event listener for GameObject events.

```
public class GameObjectEventListener : EventListener<GameObject>,
IEventListener<GameObject>
```

Inheritance

object ← EventListener<GameObject> ← GameObjectEventListener

Implements

IEventListener<GameObject>

Inherited Members

EventListener<GameObject>.targetEvent , EventListener<GameObject>.onEventInvoked , EventListener<GameObject>.StartListening(IEvent<GameObject>) , EventListener<GameObject>.StopListening(IEvent<GameObject>) , EventListener<GameObject>.OnEventInvoked(GameObject)

Interface IEventListener<T>

Namespace: SODD.Listeners

Represents a listener for an event of type T.

```
public interface IEventListener<T>
```

Type Parameters

T

The type of the event payload.

Methods

OnEventInvoked(T)

Defines a method that is called when an event is invoked.

```
void OnEventInvoked(T payload)
```

Parameters

payload T

The payload passed to the event.

StartListening(IEvent<T>)

Starts listening to the specified event by adding the supplied event listener.

```
void StartListening(IEvent<T> @event)
```

Parameters

event IEvent<T>

The event to listen to.

Remarks

This method adds the OnEventInvoked(T) method as a listener to the specified event, which will be invoked whenever the event is triggered.

StopListening(IEvent<T>)

Stops listening to an event.

```
void StopListening(IEvent<T> @event)
```

Parameters

`event` IEvent<T>

The event to stop listening to.

Class IntEventListener

Namespace: SODD.Listeners

Represents an event listener for integer events.

```
public class IntEventListener : EventListener<int>, IEventListener<int>
```

Inheritance

object ← EventListener<int> ← IntEventListener

Implements

IEventListener<int>

Inherited Members

EventListener<int>.targetEvent , EventListener<int>.onEventInvoked , EventListener<int>.StartListening(IEvent<int>) , EventListener<int>.StopListening(IEvent<int>) , EventListener<int>.OnEventInvoked(int)

Class StringEventListener

Namespace: SODD.Listeners

Represents an event listener for string events.

```
public class StringEventListener : EventListener<string>, IEventListener<string>
```

Inheritance

object ← EventListener<string> ← StringEventListener

Implements

IEventListener<string>

Inherited Members

EventListener<string>.targetEvent , EventListener<string>.onEventInvoked , EventListener<string>.StartListening(IEvent<string>) , EventListener<string>.StopListening(IEvent<string>) , EventListener<string>.OnEventInvoked(string)

Class Vector2EventListener

Namespace: SODD.Listeners

Represents an event listener for Vector2 events.

```
public class Vector2EventListener : EventListener<Vector2>, IEventListener<Vector2>
```

Inheritance

object ← EventListener<Vector2> ← Vector2EventListener

Implements

IEventListener<Vector2>

Inherited Members

EventListener<Vector2>.targetEvent , EventListener<Vector2>.onEventInvoked , EventListener<Vector2>.StartListening(IEvent<Vector2>) , EventListener<Vector2>.StopListening(IEvent<Vector2>) , EventListener<Vector2>.OnEventInvoked(Vector2)

Class Vector3EventListener

Namespace: SODD.Listeners

Represents an event listener for Vector3 events.

```
public class Vector3EventListener : EventListener<Vector3>, IEventListener<Vector3>
```

Inheritance

object ← EventListener<Vector3> ← Vector3EventListener

Implements

IEventListener<Vector3>

Inherited Members

EventListener<Vector3>.targetEvent , EventListener<Vector3>.onEventInvoked , EventListener<Vector3>.StartListening(IEvent<Vector3>) , EventListener<Vector3>.StopListening(IEvent<Vector3>) , EventListener<Vector3>.OnEventInvoked(Vector3)

Class VoidEventListener

Namespace: SODD.Listeners

Represents an event listener for void events.

```
public class VoidEventListener : EventListener<Void>, IEventListener<Void>
```

Inheritance

object ← EventListener<Void> ← VoidEventListener

Implements

IEventListener<Void>

Inherited Members

EventListener<Void>.targetEvent , EventListener<Void>.onEventInvoked , EventListener<Void>.StartListening(IEvent<Void>) , EventListener<Void>.StopListening(IEvent<Void>) , EventListener<Void>.OnEventInvoked(Void)

Namespace SODD.Observers

Classes

BoolVariableObserver

Represents a component that observes changes in a BoolVariable's value and triggers UnityEvents in response.

FloatVariableObserver

Represents a component that observes changes in a FloatVariable's value and triggers UnityEvents in response.

IntVariableObserver

Represents a component that observes changes in a IntVariable's value and triggers UnityEvents in response.

StringVariableObserver

Represents a component that observes changes in a StringVariable's value and triggers UnityEvents in response.

VariableObserver<T>

Represents an abstract component that observes changes in a Variable<T> of a specific type T and triggers UnityEvents in response.

Vector2VariableObserver

Represents a component that observes changes in a Vector2Variable's value and triggers UnityEvents in response.

Vector3VariableObserver

Represents a component that observes changes in a Vector3Variable's value and triggers UnityEvents in response.

Class BoolVariableObserver

Namespace: SODD.Observers

Represents a component that observes changes in a BoolVariable's value and triggers UnityEvents in response.

```
public class BoolVariableObserver : VariableObserver<bool>
```

Inheritance

object ← VariableObserver<bool> ← BoolVariableObserver

Remarks

This class extends VariableObserver<T> by adding functionality to convert the boolean value to a string in different formats (lowercase, uppercase, or capitalized) and trigger an event when the string value changes. This feature can be useful for UI elements where boolean states need to be displayed textually.

Methods

OnVariableValueChanged(bool)

Called when the variable's value changes.

```
protected override void OnVariableValueChanged(bool value)
```

Parameters

`value` bool

The new value of the variable.

Class FloatVariableObserver

Namespace: SODD.Observers

Represents a component that observes changes in a FloatVariable's value and triggers UnityEvents in response.

```
public class FloatVariableObserver : VariableObserver<float>
```

Inheritance

object ← VariableObserver<float> ← FloatVariableObserver

Methods

OnVariableValueChanged(float)

Called when the variable's value changes.

```
protected override void OnVariableValueChanged(float value)
```

Parameters

`value` float

The new value of the variable.

Class IntVariableObserver

Namespace: SODD.Observers

Represents a component that observes changes in a IntVariable's value and triggers UnityEvents in response.

```
public class IntVariableObserver : VariableObserver<int>
```

Inheritance

object ← VariableObserver<int> ← IntVariableObserver

Fields

onValueChangedAsString

```
[CollapsibleAttribute]
public UnityEvent<string> onValueChangedAsString
```

Field Value

UnityEvent<string>

Methods

OnVariableValueChanged(int)

Called when the variable's value changes.

```
protected override void OnVariableValueChanged(int value)
```

Parameters

`value` int

The new value of the variable.

Class StringVariableObserver

Namespace: SODD.Observers

Represents a component that observes changes in a StringVariable's value and triggers UnityEvents in response.

```
public class StringVariableObserver : VariableObserver<string>
```

Inheritance

object ← VariableObserver<string> ← StringVariableObserver

Inherited Members

VariableObserver<string>.OnVariableValueChanged(string)

Remarks

This class inherits from the VariableObserver<T> where `T` is a string. It provides a ready-to-use observer component that can be attached to any GameObject to monitor changes in string variables. This observer does not implement additional logic; it utilizes the functionality provided by its base class to link string variable changes directly to Unity events.

Class VariableObserver<T>

Namespace: SODD.Observers

Represents an abstract component that observes changes in a Variable<T> of a specific type `T` and triggers UnityEvents in response.

```
public abstract class VariableObserver<T> : MonoBehaviour
```

Type Parameters

`T`

The type of the variable being observed.

Inheritance

object ← VariableObserver<T>

Derived

BoolVariableObserver, FloatVariableObserver, IntVariableObserver, StringVariableObserver, Vector2VariableObserver, Vector3VariableObserver

Examples

An implementation example for an IntVariable observer might look like this:

```
public class IntVariableObserver : VariableObserver<int>
{
}
```

Remarks

This abstract class provides a foundation for creating scriptable variable observers. It simplifies the process of subscribing to and unsubscribing from variable changes, ensuring that observers are only active when the MonoBehaviour is enabled.

To create a custom variable observer, inherit from this class and specify the variable type `T`, which can be any valid Unity type (e.g., int, string, GameObject, etc.). In the Unity Editor, attach your new custom variable observer component to a GameObject, assign the specific scriptable variable to observe, and configure the response through the exposed UnityEvent onValueChanged.

Methods

OnVariableValueChanged(T)

Called when the variable's value changes.

```
protected virtual void OnVariableValueChanged(T value)
```

Parameters

`value` T

The new value of the variable.

See Also

Variable<T>

Class Vector2VariableObserver

Namespace: SODD.Observers

Represents a component that observes changes in a Vector2Variable's value and triggers UnityEvents in response.

```
public class Vector2VariableObserver : VariableObserver<Vector2>
```

Inheritance

object ← VariableObserver<Vector2> ← Vector2VariableObserver

Fields

onValueChangedAsString

```
[CollapsibleAttribute]
public UnityEvent<string> onValueChangedAsString
```

Field Value

UnityEvent<string>

Methods

OnVariableValueChanged(Vector2)

Called when the variable's value changes.

```
protected override void OnVariableValueChanged(Vector2 value)
```

Parameters

`value` Vector2

The new value of the variable.

Class Vector3VariableObserver

Namespace: SODD.Observers

Represents a component that observes changes in a Vector3Variable's value and triggers UnityEvents in response.

```
public class Vector3VariableObserver : VariableObserver<Vector3>
```

Inheritance

object ← VariableObserver<Vector3> ← Vector3VariableObserver

Fields

onValueChangedAsString

```
[CollapsibleAttribute]
public UnityEvent<string> onValueChangedAsString
```

Field Value

UnityEvent<string>

Methods

OnVariableValueChanged(Vector3)

Called when the variable's value changes.

```
protected override void OnVariableValueChanged(Vector3 value)
```

Parameters

`value` Vector3

The new value of the variable.

Namespace SODD.Repositories

Classes

BinaryFileRepository<T>
: Provides a concrete implementation of FileRepository<T> that manages data through binary serialization.

FileRepository<T>
: Provides a base implementation of IRepository<T> for file-based data storage.

InputIconRepository

JsonFileRepository<T>
: Provides a concrete implementation of FileRepository<T> that manages data through JSON serialization.

VariableRepository
: Manages a repository of variables, allowing for their persistent storage and retrieval using binary serialization.

Interfaces

IRepository<T>
: Defines a generic repository interface for managing persistent data.

Class BinaryFileRepository<T>

Namespace: SODD.Repositories

Provides a concrete implementation of FileRepository<T> that manages data through binary serialization.

```
public class BinaryFileRepository<T> : FileRepository<T>, IRepository<T>
```

Type Parameters

T

The type of data managed by the repository.

Inheritance

object ← FileRepository<T> ← BinaryFileRepository<T>

Implements

IRepository<T>

Inherited Members

FileRepository<T>.filename , FileRepository<T>.Filename

Remarks

This class implements the saving, loading, and deleting of data using the BinaryFormatter, which serializes and deserializes an object, or an entire graph of connected objects, in binary format. It is suitable for storing data such as game states, configurations, or user data in a compact, binary format that is not human-readable, enhancing both security and storage efficiency.

Constructors

BinaryFileRepository()

Initializes a new instance of the BinaryFileRepository<T> class.

```
public BinaryFileRepository()
```

BinaryFileRepository(string)

Initializes a new instance of the BinaryFileRepository<T> class with the specified filename.

```
public BinaryFileRepository(string filename)
```

Parameters

`filename` string

The filename to use for storing the binary data. Cannot be null.

Methods

Delete()

Deletes the binary file where the data is stored.

```
public override void Delete()
```

Remarks

This method deletes the file specified by Filename from Application.persistentDataPath.

Exists()

Determines whether the repository exists.

```
public override bool Exists()
```

Returns

bool

`true` if the repository exists; otherwise, `false`.

Load()

Loads the entity from a binary file.

```
public override T Load()
```

Returns

T

The loaded entity of type `T`.

Remarks

This method deserializes the data from a binary file specified by Filename into an object of type `T`.

Save(T)

Saves the specified entity to a binary file.

```
public override void Save(T t)
```

Parameters

`t` T

The entity to save. The entity must be serializable.

Remarks

This method serializes the object of type `T` using BinaryFormatter and writes it to a file specified by Filename. The data path used is Application.persistentDataPath.

Class FileRepository<T>

Namespace: SODD.Repositories

Provides a base implementation of IRepository<T> for file-based data storage.

```
public abstract class FileRepository<T> : IRepository<T>
```

Type Parameters

T

The type of data managed by the repository.

Inheritance
object ← FileRepository<T>

Implements
IRepository<T>

Derived
BinaryFileRepository<T>, JsonFileRepository<T>

Remarks

This abstract class lays the foundation for creating file-based repositories that can save, load, and delete data of type `T`. Implementations require a filename for the underlying storage mechanism.

Constructors

FileRepository()

Initializes a new instance of the FileRepository<T> class without specifying a filename. The filename must be set before performing I/O operations.

```
protected FileRepository()
```

FileRepository(string)

Initializes a new instance of the FileRepository<T> class with the specified filename.

```
protected FileRepository(string filename)
```

Parameters

`filename` string

The filename to use for data storage. Cannot be null.

Fields

filename

The filename used for storing data.

```
protected string filename
```

Field Value

string

Properties

Filename

Gets or sets the filename used for data storage.

```
public virtual string Filename { get; set; }
```

Property Value

string

Methods

Delete()

Deletes the entity of type `T` from the repository.

```
public abstract void Delete()
```

Exists()

Determines whether the repository exists.

```
public abstract bool Exists()
```

Returns

bool

`true` if the repository exists; otherwise, `false`.

Load()

Loads the entity of type `T` from the repository.

```
public abstract T Load()
```

Returns

T

The loaded entity of type `T`.

Save(T)

Saves the specified entity of type `T` to the repository.

```
public abstract void Save(T t)
```

Parameters

`t` T

The entity to save.

Interface IRepository<T>

Namespace: SODD.Repositories

Defines a generic repository interface for managing persistent data.

```
public interface IRepository<T>
```

Type Parameters

`T`

The type of data the repository manages.

Remarks

This interface provides a framework for implementing CRUD (Create, Read, Update, Delete) operations on data of type `T`. Implementations should encapsulate the specifics of data storage, retrieval, and deletion.

Methods

Delete()

Deletes the entity of type `T` from the repository.

```
void Delete()
```

Exists()

Determines whether the repository exists.

```
bool Exists()
```

Returns

bool

`true` if the repository exists; otherwise, `false`.

Load()

Loads the entity of type `T` from the repository.

```
T Load()
```

Returns

T

The loaded entity of type `T`.

Save(T)

Saves the specified entity of type `T` to the repository.

```
void Save(T t)
```

Parameters

`t` T

The entity to save.

Class InputIconRepository

Namespace: SODD.Repositories

```
public class InputIconRepository : PassiveScriptableObject
```

Inheritance

object ← PassiveScriptableObject ← InputIconRepository

Inherited Members

PassiveScriptableObject.reference

Methods

GetIcon(string)

```
public Sprite GetIcon(string path)
```

Parameters

`path` string

Returns

Sprite

Class JsonFileRepository<T>

Namespace: SODD.Repositories

Provides a concrete implementation of FileRepository<T> that manages data through JSON serialization.

```
public class JsonFileRepository<T> : FileRepository<T>, IRepository<T>
```

Type Parameters

T

The type of data managed by the repository.

Inheritance

object ← FileRepository<T> ← JsonFileRepository<T>

Implements

IRepository<T>

Inherited Members

FileRepository<T>.filename , FileRepository<T>.Filename

Remarks

This class implements the saving, loading, and deleting of data using JSON serialization, leveraging JsonUtility for serialization and deserialization. It is ideal for storing data such as game settings, player progress, or other configuration data in a human-readable format, making it easier to edit manually if necessary.

Constructors

JsonFileRepository()

Initializes a new instance of the JsonFileRepository<T> class.

```
public JsonFileRepository()
```

JsonFileRepository(string)

Initializes a new instance of the JsonFileRepository<T> class with the specified filename.

```
public JsonFileRepository(string filename)
```

Parameters

`filename` string

The filename to use for storing the JSON data. Cannot be null.

Methods

Delete()

Deletes the JSON file where the data is stored.

```
public override void Delete()
```

Remarks

This method deletes the file specified by Filename from Application.persistentDataPath.

Exists()

Determines whether the repository exists.

```
public override bool Exists()
```

Returns

bool

`true` if the repository exists; otherwise, `false`.

Load()

Loads the entity from a JSON file.

```
public override T Load()
```

Returns

T

The loaded entity of type `T`.

Remarks

This method deserializes the data from a JSON file specified by Filename into an object of type `T` using JsonUtility.

Save(T)

Saves the specified entity to a JSON file.

```
public override void Save(T t)
```

Parameters

`t` T

The entity to save.

Remarks

This method serializes the object of type `T` using JsonUtility and writes it to a file specified by Filename. The data path used is Application.persistentDataPath.

Class VariableRepository

Namespace: SODD.Repositories

Manages a repository of variables, allowing for their persistent storage and retrieval using binary serialization.

```
public class VariableRepository : PassiveScriptableObject
```

Inheritance

object ← PassiveScriptableObject ← VariableRepository

Inherited Members

PassiveScriptableObject.reference

Remarks

This ScriptableObject serves as a repository for various game variables, such as settings or player data, and leverages binary serialization for efficient, non-human-readable storage.

Methods

Delete()

Deletes the binary file where the variables are stored.

```
public void Delete()
```

Load()

Loads the state of variables from a binary file.

```
public void Load()
```

Save()

Saves the current state of variables to a binary file.

```
public void Save()
```

Namespace SODD.UI

Classes

OptionSelector

Class OptionSelector

Namespace: SODD.UI

```
public class OptionSelector : Selectable
```

Inheritance

object ← OptionSelector

Properties

Value

```
public int Value { get; set; }
```

Property Value

int

Methods

AddOption(string)

```
public void AddOption(string option)
```

Parameters

`option` string

AddOptions(List<string>)

```
public void AddOptions(List<string> optionsList)
```

Parameters

`optionsList` List<string>

OnMove(AxisEventData)

```
public override void OnMove(AxisEventData eventData)
```

Parameters

`eventData` AxisEventData

SelectNext()

```
public void SelectNext()
```

SelectPrevious()

```
public void SelectPrevious()
```

SetOptions(List<string>)

```
public void SetOptions(List<string> optionsList)
```

Parameters

`optionsList` List<string>

SetValueWithoutNotify(int)

```
public void SetValueWithoutNotify(int value)
```

Parameters

`value` int

Namespace SODD.Variables

Classes

BoolVariable
: A ScriptableObject representing a boolean variable.

FloatVariable
: A ScriptableObject representing a floating-point number variable.

GameObjectVariable
: A ScriptableObject representing a GameObject variable.

IntVariable
: A ScriptableObject representing an integer variable.

LayerMaskVariable
: A ScriptableObject representing a LayerMask variable.

StringVariable
: A ScriptableObject representing a string variable.

ValueReference<T>
: Represents a reference holder for values of type `T`.

Variable<T>
: Represents an abstract variable of type `T`.

Vector2Variable
: A ScriptableObject representing a Vector2 variable.

Vector3Variable
: A ScriptableObject representing a Vector3 variable.

Interfaces

IVariable
: Represents a generic variable interface for type-agnostic variable usage.

IVariable<T>
: Represents a generic variable interface with a specific type.

Class BoolVariable

Namespace: SODD.Variables

A ScriptableObject representing a boolean variable.

```
public class BoolVariable : Variable<bool>, IVariable, IVariable<bool>,
IListenableEvent<bool>
```

Inheritance

object ← PersistentScriptableObject ← Variable<bool> ← BoolVariable

Implements

IVariable, IVariable<bool>, IListenableEvent<bool>

Inherited Members

Variable<bool>.value , Variable<bool>.readOnly , Variable<bool>.OnValueChanged , Variable<bool>.Id , Variable<bool>.Value , Variable<bool>.HandleValueChange() , Variable<bool>.AddListener(Action<bool>) , Variable<bool>.RemoveListener(Action<bool>) , PersistentScriptableObject.persist

Remarks

The BoolVariable class is a specialized implementation of the Variable class for boolean (true/false) values. It facilitates the creation and management of a boolean value that can be shared across different components and scripts within a Unity project, allowing for a centralized approach to handling binary states.

This class can be used to create boolean variables directly in the Unity Editor. These variables are ideal for toggling states, managing conditions, or controlling binary features such as switches or flags in game mechanics.

The boolean value can be marked as read-only to prevent runtime modifications. Additionally, the class emits an event whenever the value changes, providing the capability to react dynamically to state changes.

Class FloatVariable

Namespace: SODD.Variables

A ScriptableObject representing a floating-point number variable.

```
public class FloatVariable : Variable<float>, IVariable, IVariable<float>,
IListenableEvent<float>
```

Inheritance

object ← PersistentScriptableObject ← Variable<float> ← FloatVariable

Implements

IVariable, IVariable<float>, IListenableEvent<float>

Inherited Members

Variable<float>.value , Variable<float>.readOnly , Variable<float>.OnValueChanged , Variable<float>.Id , Variable<float>.Value , Variable<float>.HandleValueChange() , Variable<float>.AddListener(Action<float>) , Variable<float>.RemoveListener(Action<float>) , PersistentScriptableObject.persist

Remarks

The FloatVariable class is a specialized implementation of the Variable class for float values. It is designed to hold a floating-point number that can be shared across different components and scripts in a Unity project, allowing for centralized management of float data.

This class can be used to create float variables directly in the Unity Editor. These variables can be utilized for a range of purposes such as tracking dynamic numerical values like health points, speed, percentages, or any other measurements that require floating-point precision.

The float value can be set as read-only to prevent changes at runtime, maintaining data integrity. The class also emits an event whenever the value changes, enabling other scripts to respond dynamically to these changes.

Class GameObjectVariable

Namespace: SODD.Variables

A ScriptableObject representing a GameObject variable.

```
public class GameObjectVariable : Variable<GameObject>, IVariable,
IVariable<GameObject>, IListenableEvent<GameObject>
```

Inheritance

object ← PersistentScriptableObject ← Variable<GameObject> ← GameObjectVariable

Implements

IVariable, IVariable<GameObject>, IListenableEvent<GameObject>

Inherited Members

Variable<GameObject>.value , Variable<GameObject>.readOnly , Variable<GameObject>.OnValueChanged , Variable<GameObject>.Id , Variable<GameObject>.Value , Variable<GameObject>.HandleValueChange() , Variable<GameObject>.AddListener(Action<GameObject>) , Variable<GameObject>.RemoveListener(Action<GameObject>) , PersistentScriptableObject.persist

Remarks

The GameObjectVariable class is a specialized implementation of the Variable class for GameObject values. It is designed to hold a GameObject value that can be shared across different components and scripts in a Unity project, allowing for centralized management of GameObject data.

This class can be used to create GameObject variables directly in the Unity Editor. These variables can be utilized for a range of purposes such as storing references to key GameObjects, controlling object behavior, or any other interactions that may need to be dynamically changed or referenced during gameplay.

The GameObject value can be set as read-only to prevent changes at runtime, ensuring the consistency of object interactions. The class also emits an event whenever the value changes, enabling other scripts to react dynamically to GameObject updates.

Interface IVariable

Namespace: SODD.Variables

Represents a generic variable interface for type-agnostic variable usage.

```
public interface IVariable
```

Remarks

This interface allows for handling variables whose type might not be known at compile time or when it is necessary to operate on variables in a type-agnostic manner. It can be used in scenarios where variables are managed dynamically, such as in systems that load or save settings where the types can vary.

Properties

Id

```
string Id { get; }
```

Property Value

string

Value

Gets or sets the value of the variable.

```
object Value { get; set; }
```

Property Value

object

The value of the variable, stored as an object.

Interface IVariable<T>

Namespace: SODD.Variables

Represents a generic variable interface with a specific type.

```
public interface IVariable<T>
```

Type Parameters

T

The type of the variable's value.

Remarks

This interface defines a generic variable with a specific type, allowing for the creation of type-safe variables. It's used to encapsulate data that can be exposed and manipulated within the Unity Editor and through scripts.

Properties

Id

```
string Id { get; }
```

Property Value

string

Value

Gets or sets the value of the variable.

```
T Value { get; set; }
```

Property Value

T

The value of the variable, of type T.

Class IntVariable

Namespace: SODD.Variables

A ScriptableObject representing an integer variable.

```
public class IntVariable : Variable<int>, IVariable, IVariable<int>,
IListenableEvent<int>
```

Inheritance

object ← PersistentScriptableObject ← Variable<int> ← IntVariable

Implements

IVariable, IVariable<int>, IListenableEvent<int>

Inherited Members

Variable<int>.value , Variable<int>.readOnly , Variable<int>.OnValueChanged , Variable<int>.Id , Variable<int>.Value , Variable<int>.HandleValueChange() , Variable<int>.AddListener(Action<int>) , Variable<int>.RemoveListener(Action<int>) , PersistentScriptableObject.persist

Remarks

The IntVariable class is a specialized implementation of the Variable class for integer values. It is designed to hold an integer value that can be shared across different components and scripts in a Unity project, allowing for centralized management of integer data.

This class can be used to create integer variables directly in the Unity Editor. These variables can be used for various purposes such as keeping track of scores, counts, or other numerical game-related data that require integer representation.

The integer value can be set as read-only to prevent changes at runtime, ensuring data integrity. The class also emits an event whenever the value changes, allowing other scripts to react to these changes.

Class LayerMaskVariable

Namespace: SODD.Variables

A ScriptableObject representing a LayerMask variable.

```
public class LayerMaskVariable : Variable<LayerMask>, IVariable,
IVariable<LayerMask>, IListenableEvent<LayerMask>
```

Inheritance

object ← PersistentScriptableObject ← Variable<LayerMask> ← LayerMaskVariable

Implements

IVariable, IVariable<LayerMask>, IListenableEvent<LayerMask>

Inherited Members

Variable<LayerMask>.value , Variable<LayerMask>.readOnly , Variable<LayerMask>.OnValueChanged , Variable<LayerMask>.Id , Variable<LayerMask>.Value , Variable<LayerMask>.HandleValueChange() , Variable<LayerMask>.AddListener(Action<LayerMask>) , Variable<LayerMask>.RemoveListener(Action<LayerMask>) , PersistentScriptableObject.persist

Remarks

The LayerMaskVariable class is a specialized implementation of the Variable class for LayerMask values. It is designed to hold a LayerMask value that can be shared across different components and scripts in a Unity project, allowing for centralized management of LayerMask data.

This class can be used to create LayerMask variables directly in the Unity Editor. These variables can be utilized for a range of purposes such as setting certain layers to interact or not interact with others, managing collision detection, or any other game rules that may need to be dynamically changed or referenced during gameplay.

The LayerMask value can be set as read-only to prevent changes at runtime, ensuring the consistency of layer rules. The class also emits an event whenever the value changes, enabling other scripts to react dynamically to LayerMask updates.

Class StringVariable

Namespace: SODD.Variables

A ScriptableObject representing a string variable.

```
public class StringVariable : Variable<string>, IVariable,
IVariable<string>, IListenableEvent<string>
```

Inheritance

object ← PersistentScriptableObject ← Variable<string> ← StringVariable

Implements

IVariable, IVariable<string>, IListenableEvent<string>

Inherited Members

Variable<string>.value , Variable<string>.readOnly , Variable<string>.OnValueChanged , Variable<string>.Id , Variable<string>.Value , Variable<string>.HandleValueChange() , Variable<string>.AddListener(Action<string>) , Variable<string>.RemoveListener(Action<string>) , PersistentScriptableObject.persist

Remarks

The StringVariable class is a specialized implementation of the Variable class for string values. It is designed to hold a string value that can be shared across different components and scripts in a Unity project, allowing for centralized management of string data.

This class can be used to create string variables directly in the Unity Editor. These variables can be utilized for a range of purposes such as storing text for UI elements, dialogue, descriptions, or any other textual content that may need to be dynamically changed or referenced during gameplay.

The string value can be set as read-only to prevent changes at runtime, ensuring the integrity of the textual content. The class also emits an event whenever the value changes, enabling other scripts to react dynamically to text updates.

Class ValueReference<T>

Namespace: SODD.Variables

Represents a reference holder for values of type `T`.

```
[Serializable]
public class ValueReference<T> : IVariable<T>, IListenableEvent<T>
```

Type Parameters

`T`

The type of the data held by this reference.

Inheritance

object ← ValueReference<T>

Implements

IVariable<T>, IListenableEvent<T>

Examples

Example of using ValueReference to switch between a local and shared value:

```
public class Player : MonoBehaviour
{
    public ValueReference<int> maxHealth;

    void Start()
    {
        Debug.Log("Max Health: " + maxHealth.Value);
    }
}
```

In this example, the `maxHealth` can be configured in the Unity Editor to use either a local field value or a reference to a shared Variable<int> asset. This allows for easy adjustments in gameplay mechanics without needing to touch the source code.

Remarks

This class provides a flexible mechanism to reference values either directly as serialized fields, or indirectly through a reference to a Variable<T> asset. This allows developers to switch between inner instance-specific values and shared variable values without modifying the code that uses said values.

Properties

Id

```
public string Id { get; }
```

Property Value

string

Value

Gets or sets the value of the reference, choosing between a local field and an external variable based on configuration.

```
public T Value { get; set; }
```

Property Value

T

Methods

AddListener(Action<T>)

Registers an action to be called when the referenced value changes.

```
public void AddListener(Action<T> listener)
```

Parameters

`listener` Action<T>

The action to invoke when the value changes.

RemoveListener(Action<T>)

Unregisters an action previously added to listen for value changes.

```
public void RemoveListener(Action<T> listener)
```

Parameters

`listener` Action<T>

The action to remove.

Class Variable<T>

Namespace: SODD.Variables

Represents an abstract variable of type `T`.

```
public abstract class Variable<T> : PersistentScriptableObject, IVariable,
IVariable<T>, IListenableEvent<T>
```

Type Parameters

`T`

The type of the variable.

Inheritance

object ← PersistentScriptableObject ← Variable<T>

Implements

IVariable, IVariable<T>, IListenableEvent<T>

Derived

BoolVariable, FloatVariable, GameObjectVariable, IntVariable, LayerMaskVariable, StringVariable, Vector2Variable, Vector3Variable

Inherited Members

PersistentScriptableObject.persist

Examples

Example of an IntVariable implementation:

```
[CreateAssetMenu(menuName = "My Variables/Int Variable", fileName
= nameof(StringVariable))]
public class IntVariable : Variable<int> {}
```

Remarks

This abstract class serves as the base for all scriptable variable implementations. It implements the IVariable<T> interface allowing the creation of data containers that can be easily manipulated within the Unity Editor and referenced across different scripts, making it

useful for a wide range of applications, such as game settings, runtime configurations, or shared data across different game components.

The value of the variable can be marked as read-only to prevent modification at runtime. Additionally, the class implements the IListenableEvent<T> interface, which allows listeners to be added and notified when the value of the variable changes.

To create custom scriptable variable, inherit from this class and specify the payload type `T`, it can be any valid Unity type (e.g., int, string, GameObject, etc.).

Fields

OnValueChanged

An event that is triggered whenever the value of the variable changes.

```
protected readonly IEvent<T> OnValueChanged
```

Field Value

IEvent<T>

readOnly

```
protected bool readOnly
```

Field Value

bool

value

```
[OnValueChangedAttribute]
protected T value
```

Field Value

T

Properties

Id

Gets the variable's unique identifier.

```
public string Id { get; }
```

Property Value

string

Value

Gets or sets the value of the variable.

```
public T Value { get; set; }
```

Property Value

T

The current value of the variable.

Methods

AddListener(Action<T>)

Registers an action to be called when the variable's value changes.

```
public void AddListener(Action<T> listener)
```

Parameters

`listener` Action<T>

The action to invoke when the value changes.

HandleValueChange()

Invokes the OnValueChanged event and can log the change if debugging is enabled.

```
protected void HandleValueChange()
```

RemoveListener(Action<T>)

Unregisters an action previously added to listen for value changes.

```
public void RemoveListener(Action<T> listener)
```

Parameters

`listener` Action<T>

The action to remove.

Class Vector2Variable

Namespace: SODD.Variables

A ScriptableObject representing a Vector2 variable.

```
public class Vector2Variable : Variable<Vector2>, IVariable,
IVariable<Vector2>, IListenableEvent<Vector2>
```

Inheritance

object ← PersistentScriptableObject ← Variable<Vector2> ← Vector2Variable

Implements

IVariable, IVariable<Vector2>, IListenableEvent<Vector2>

Inherited Members

Variable<Vector2>.value , Variable<Vector2>.readOnly , Variable<Vector2>.OnValueChanged , Variable<Vector2>.Id , Variable<Vector2>.Value , Variable<Vector2>.HandleValueChange() , Variable<Vector2>.AddListener(Action<Vector2>) , Variable<Vector2>.RemoveListener(Action<Vector2>) , PersistentScriptableObject.persist

Remarks

The Vector2Variable class is a specialized implementation of the Variable class for Vector2 values. It is designed to hold a two-dimensional vector that can be shared across different components and scripts in a Unity project, allowing for centralized management of Vector2 data.

This class can be used to create Vector2 variables directly in the Unity Editor. These variables can be utilized for various purposes, such as tracking positions, velocities, or other 2D vector-related data within your game or application.

The Vector2 value can be set as read-only to prevent changes at runtime, ensuring the integrity of the data. The class also emits an event whenever the value changes, enabling other scripts to react dynamically to Vector2 data updates.

Class Vector3Variable

Namespace: SODD.Variables

A ScriptableObject representing a Vector3 variable.

```
public class Vector3Variable : Variable<Vector3>, IVariable,
IVariable<Vector3>, IListenableEvent<Vector3>
```

Inheritance

object ← PersistentScriptableObject ← Variable<Vector3> ← Vector3Variable

Implements

IVariable, IVariable<Vector3>, IListenableEvent<Vector3>

Inherited Members

Variable<Vector3>.value , Variable<Vector3>.readOnly , Variable<Vector3>.OnValueChanged , Variable<Vector3>.Id , Variable<Vector3>.Value , Variable<Vector3>.HandleValueChange() , Variable<Vector3>.AddListener(Action<Vector3>) , Variable<Vector3>.RemoveListener(Action<Vector3>) , PersistentScriptableObject.persist

Remarks

The Vector3Variable class is a specialized implementation of the Variable class for Vector3 values. It is designed to hold a three-dimensional vector that can be shared across different components and scripts in a Unity project, allowing for centralized management of Vector3 data.

This class can be used to create Vector3 variables directly in the Unity Editor. These variables can be utilized for various purposes, such as tracking positions, velocities, directions, or other 3D vector-related data within your game or application.

The Vector3 value can be set as read-only to prevent changes at runtime, ensuring the integrity of the data. The class also emits an event whenever the value changes, enabling other scripts to react dynamically to Vector3 data updates.