KennethDevelops'

DevUtils

v1.1.0

Table of Contents

Introduction

Welcome to DevUtils, a Unity asset package that offers a range of versatile and efficient tools designed to simplify and speed up your development workflow in Unity. This package aims to provide reusable solutions for common game development challenges and architectural needs, allowing you to focus more on creating your game logic and less on managing the underlying structure.

With DevUtils, you get a collection of robust utilities including:

1. `Singleton`: A generic implementation of the singleton pattern, allowing you to effortlessly instantiate and manage singleton objects in your game.

2. `MonoSingleton`: A singleton pattern built specifically for MonoBehaviour classes in Unity. It takes care of creating an instance on demand, managing the singleton lifecycle, and providing global access to the instance.

3. `OrderedList<T>`: A handy utility class for maintaining a list of elements with a specific order or priority. This is useful in scenarios like event management, task scheduling, and any situation where the order of elements matters.

4. `OrderedElement<T>`: A class encapsulating an element within an ordered collection and associates it with an ordering value. It's used in conjunction with `OrderedList<T>` to organize and manage ordered data.

And much more to come! The DevUtils package is an evolving suite of tools, with new utilities and features planned for future releases. It's not just a toolset, it's a growing ecosystem aimed at making your game development in Unity easier, faster, and more enjoyable.

Whether you're a seasoned developer looking for ways to streamline your process, or a beginner seeking to understand and apply best practices, DevUtils has something to offer you.

Getting Started

To get started with DevUtils, simply download the package from the Unity Asset Store and import it into your project. To use any of the classes in DevUtils, you will need to import the namespace `KennethDevelops.DevUtils.Model` at the top of your script.

Once imported, the utilities are ready for use and can be accessed from your scripts. Check out the documentation for each utility for more detailed information, usage instructions, and examples.

Stay tuned for updates, and happy game developing with DevUtils!

Singleton<T>

`Singleton<T>` is an abstract class used to implement the Singleton pattern. Singleton ensures that a class has only one instance during the application's lifecycle, providing a global point of access to this instance.

Remarks

The Singleton pattern is a popular design pattern with many use cases. It is beneficial when exactly one object is needed to coordinate actions across the system. For example, an object that represents the preferences of a user or system settings would be best implemented as a Singleton.

The `Singleton<T>` class has an `Instance` property that provides access to the single instance of the class and an `Init` method, which should be overridden to provide custom initialization logic.

Methods

protected abstract void `Init()`;

The `Init` method is an abstract method, intended to be overridden by the concrete Singleton classes. The method is automatically called once during the creation of the Singleton instance.

This method is where one-time setup code should be placed. It's useful for any initializations that need to happen once and stay the same throughout the lifetime of the instance.

Properties

public static T `Instance`

Returns the single instance of the `Singleton<T>`. If the instance has not been created yet, it will instantiate a new one and call the `Init` method before returning it. The `Instance` property is static, meaning it can be accessed without creating an object of the Singleton class.

Example

Here's an example of how to use the `Singleton<T>` class to create a `GameManager` Singleton that initializes the game state:

```
public class GameManager : Singleton<GameManager>
{
    private int score;

    protected override void Init()
    {
        score = 0;
        // More game state setup here...
    }

    public void IncreaseScore(int amount)
    {
        score += amount;
    }
}
```

In this example, `GameManager.Instance` can be accessed from any other part of the game to manage the game's state.

MonoSingleton<T>

The `MonoSingleton<T>` class is a MonoBehaviour abstract class. It is a generic singleton implementation for MonoBehaviour objects in Unity. It's designed to allow easy singleton-like access to a MonoBehaviour.

Summary

A Singleton MonoBehaviour (MonoSingleton) provides a globally accessible instance that is created automatically and managed by Unity's lifecycle. This singleton implementation ensures that only one instance of a type exists and allows for easy access to that instance. Furthermore, it ensures that the singleton instance persists across scenes unless specified otherwise.

Properties

protected virtual bool `DoNotDestroyOnLoad`

Override this property to set whether this singleton should be destroyed when a new scene is loaded.

public static bool `ApplicationQuitting`

A flag set to true when the application is quitting. Used to avoid creating new instances.

public static T `Instance`

Gets the instance of the singleton. Creates a new GameObject and adds the singleton instance as a component if it wasn't previously set.

Methods

protected abstract void `Init()`

This method is called when the Singleton instance is created. Override it to add initialization logic.

protected bool `SetInstanceIfNull(T instance)`

Sets the Instance if it was null. Returns whether the Instance was set. The method also calls `Init()` and if `DoNotDestroyOnLoad` is true, prevents the gameObject from being destroyed when loading a new scene.

protected void `SetInstance(T instance)`

Sets the Instance regardless if it was previously set or not. Similar to `SetInstanceIfNull(T instance)`, but does not check if Instance was already set. The method also calls `Init()` and if `DoNotDestroyOnLoad` is true, prevents the gameObject from being destroyed when loading a new scene.

protected virtual void `OnApplicationQuit()`

Sets `ApplicationQuitting` to true when the application is quitting.

Use Case Examples

Suppose we are developing a game and we have a GameManager class that maintains the game's state and offers functions for game progress. This GameManager should be accessible from various points in our code and persist across different scenes.

```
public class GameManager : MonoSingleton<GameManager> {

    private int score;

    protected override void Init() {
        score = 0;
    }

    public void IncrementScore(int increment) {
        score += increment;
    }

    public int GetScore() {
        return score;
    }
}
```

In another class or scene, we can access the GameManager and its properties and methods as follows:

```
public class ScoreController : MonoBehaviour
```

```
{
    public void UpdateScore(int increment) {
        GameManager.Instance.IncrementScore(increment);
    }

    public void DisplayScore() {
        Debug.Log($"Current Score: {GameManager.Instance.GetScore()}");
    }
}
```

In this example, we are leveraging the singleton nature of our GameManager. No matter where or when we call `GameManager.Instance`, we're accessing the same GameManager instance throughout our game.

A different case can be when we want a specific `GameManager` in the initial scene with specific values. In another testing scene, we may not care about those initial values so it's fine to use the `GameManager` that gets generated with default values. We can ensure that the instance with the specific values only gets created once and no new instances are created in the test scene by using `SetInstanceIfNull(this)` in the `Awake()` method:

```
public class GameManager : MonoSingleton<GameManager> {

    private int score;

    protected override void Init() {
        score = 0;
    }

    private void Awake() {
        SetInstanceIfNull(this);
    }

    public void IncrementScore(int increment) {
        score += increment;
    }

    public int GetScore() {
        return score;
    }
}
```

Here, `SetInstanceIfNull(this)` ensures that this specific `GameManager` only becomes the singleton instance if no other instance has been set. This way, we can control the initialization of our `MonoSingleton` instances more granely.

OrderedList<T>

The OrderedList class represents a collection of elements, each associated with a priority or rank. This class allows for managing elements in a way that maintains their order based on assigned or inherent priorities. This could be particularly beneficial in scenarios where the order of processing or presentation of elements matters and is not necessarily determined by the order of insertion. The elements are arranged according to an order value, which can be explicitly set during addition or defaults to a pre-specified value. The order of elements can be manipulated as per the needs of the application, offering a flexible and efficient solution for managing ordered collections.

Class Signature

public class OrderedList<T> : IEnumerable<OrderedElement<T>>

Constructors

OrderedList()

Creates a new instance of the `OrderedList<T>` class.

public OrderedList()

OrderedList(int defaultOrder)

Creates a new instance of the `OrderedList<T>` class with a specified default order.

public OrderedList(int defaultOrder)

Parameters

- `defaultOrder`: The default order for the elements.

Properties

Count

Returns the number of elements in the list.

public int Count

Methods

Add(int order, T element)

Adds an element to the list with a specified order.

public OrderedList<T> Add(int order, T element)

Parameters

- `order`: The order of the element.
- `element`: The element to add to the list.

Returns

Returns the updated list.

Add(OrderedElement element)

Adds an ordered element to the list.

public OrderedList<T> Add(OrderedElement<T> element)

Parameters

- `element`: The ordered element to add to the list.

Returns

Returns the updated list.

Remove(OrderedElement element)

Removes an ordered element from the list.

public OrderedList<T> Remove(OrderedElement<T> element)

Parameters

- `element`: The ordered element to remove from the list.

Returns

Returns the updated list.

Remove(T element)

Removes an element from the list.

public OrderedList<T> Remove(T element)

Parameters

- `element`: The element to remove from the list.

Returns

Returns the updated list.

RemoveAt(int order)

Removes the element at a specific order from the list.

public OrderedList<T> RemoveAt(int order)

Parameters

- `order`: The order of the element to remove.

Returns

Returns the updated list.

RemoveAll(T element)

Removes all instances of a specific element from the list.

public OrderedList<T> RemoveAll(T element)

Parameters

- `element`: The element to remove.

Returns

Returns the updated list.

RemoveAll(Func<T, bool> condition)

Removes all elements that match the conditions defined by the specified predicate.

public OrderedList<T> RemoveAll(Func<T, bool> condition)

Parameters

- `condition`: The predicate delegate that defines the conditions of the elements to remove.

Returns

Returns the updated list.

GetOrderedElements()

Returns the list of ordered elements.

public List<OrderedElement<T>> GetOrderedElements()

Returns

Returns the list of ordered elements.

GetEnumerator()

Returns an enumerator that iterates through the list.

public IEnumerator GetEnumerator()

Returns

An enumerator that can be used to iterate through the list.

Operators

operator - (OrderedList list, OrderedElement element)

Removes an ordered element from the list.

public static OrderedList<T> operator - (OrderedList<T> list, OrderedElement<T> element)

Parameters

- `list`: The ordered list.
- `element`: The ordered element to remove.

Returns

Returns the updated list.

operator - (OrderedList list, T element)

Removes an element from the list.

public static OrderedList<T> operator - (OrderedList<T> list, T element)

Parameters

- `list`: The ordered list.
- `element`: The element to remove.

Returns

Returns the updated list.

operator + (OrderedList list, OrderedElement element)

Adds an ordered element to the list.

public static OrderedList<T> operator + (OrderedList<T> list, OrderedElement<T> element)

Parameters

- `list`: The ordered list.
- `element`: The ordered element to add.

Returns

Returns the updated list.

operator + (OrderedList list, T element)

Adds an element to the list.

public static OrderedList<T> operator + (OrderedList<T> list, T element)

Parameters

- `list`: The ordered list.
- `element`: The element to add.

Returns

Returns the updated list.

Examples

```
OrderedList<int> priorityList = new OrderedList<int>();
priorityList = priorityList + 5;
priorityList = priorityList + 3;
priorityList = priorityList - 5;
```

The above example shows the instantiation of an ordered list of integers. The + operator is used to add elements 5 and 3, while the - operator removes element 5 from the list.

```
OrderedList<string> toDoList = new OrderedList<string>(50);
toDoList.Add(1, "Wake up");
toDoList.Add(2, "Eat breakfast");
toDoList.Add(3, "Work");
toDoList.Remove("Work");
```

This example illustrates the usage of a `OrderedList` to manage a list of tasks in a to-do list. In this case, the tasks have been added in a certain order, and later, the task "Work" is removed from the list.

OrderedElement<T>

The `OrderedElement<T>` class encapsulates an element within an ordered collection and associates it with an ordering value. The order of an element can be explicitly set during its creation, allowing the containing list or set to arrange its elements based on these assigned order values. This could be particularly beneficial in situations where individual items of a collection carry inherent priority or sequence value, impacting the way they are processed or presented. The type of the encapsulated element is generic, making this a versatile solution for ordered collections containing diverse types of objects.

This class is primarily used by the OrderedList but you could use it on your own.

Properties

`order`

This public field represents the order value of the element in the list.

public int order;

`element`

This public field represents the encapsulated element in the list.

public T element;

Method

`OrderedElement(order, element)`

This is the constructor for the `OrderedElement<T>` class. It creates a new instance of `OrderedElement<T>` with a specified order and element.

```
public OrderedElement(int order, T element) {
   this.order   = order;
   this.element = element;
}
```

Parameters

- `order` (int): The order of the element.
- `element` (T): The encapsulated element.

Use Cases

Suppose you are managing a task queue with tasks of varying priorities. Each task can be represented as an `OrderedElement<T>`, where `T` is the task object and `order` represents the priority. Tasks with higher priority would have a lower order value.

OrderedElement<Task> highPriorityTask = new OrderedElement<Task>(1, task1);
OrderedElement<Task> lowPriorityTask = new OrderedElement<Task>(10, task2);

This way, the `OrderedList<T>` can manage these tasks efficiently according to their priorities.

SerializableDictionary<TKey,TValue>

The `SerializableDictionary` class is a versatile tool that allows you to create your own serializable dictionaries within Unity, which doesn't natively support dictionary serialization. This class can be used to create serializable dictionaries of any key-value pair types. By extending this class and specifying your desired key-value types, Unity can both serialize the field of this custom type and display it in the editor.

This class inherits from `Dictionary<TKey, TValue>` and implements `ISerializationCallbackReceiver` interface to interact with Unity's serialization system.

public class SerializableDictionary<TKey, TValue> : Dictionary<TKey, TValue>, ISerializationCallbackReceiver

Properties

keys

This private field represents a list of dictionary keys.

private List<TKey> keys;

values

This private field represents a list of dictionary values.

private List<TValue> values;

Methods

SerializableDictionary()

The default constructor for the `SerializableDictionary<TKey, TValue>` class.

public SerializableDictionary()

SerializableDictionary(info, context)

The constructor for the `SerializableDictionary<TKey, TValue>` class that is used during the serialization process.

protected SerializableDictionary(SerializationInfo info, StreamingContext context)

OnBeforeSerialize()

This method is automatically called before Unity serializes the object. It takes the keys and values from the dictionary and adds them to the `keys` and `values` lists, preparing the object for serialization.

public void OnBeforeSerialize()

OnAfterDeserialize()

This method is automatically called after Unity has deserialized the object. It clears the dictionary and adds keys and values from the `keys` and `values` lists to the dictionary. If the number of keys doesn't match the number of values, an exception is thrown.

public void OnAfterDeserialize()

Use Cases

Imagine you're creating a game where you need to manage loot drops for various enemy types. Each enemy type can drop multiple items, each with its own drop chance. This can be represented with a `SerializableDictionary<EnemyType, List<LootDrop>>`, where `EnemyType` is an enum of different enemy types, and `LootDrop` is a class that represents an item and its drop chance.

[Serializable]

public class LootDropDictionary : SerializableDictionary<EnemyType, List<LootDrop>> { }

Now, this dictionary can be serialized by Unity, and you can set the loot drops for each enemy type directly from the Unity editor.

Another scenario could be tracking player's progress in a game with multiple levels. You could have a `SerializableDictionary<int, LevelData>` where the integer key represents the level number, and `LevelData` stores the player's best time, score, etc. for that level.

[Serializable]

public class LevelProgressDictionary : SerializableDictionary<int, LevelData> { }