Heavy Cav Studios Core Package

Overview

This package contains the core library used by all of the other packages by Heavy Cav Studios.

Core Utilities Overview

This document provides an overview of the core classes and utilities used in `HeavyCavStudios.Core`. The classes are organized by their functionality and purpose, making it easier to navigate through different aspects of the codebase.

Contents

Collections

Coroutines

Editor

Extensions

Factories

Patterns

Reflection

Serialization

Collections

OrderedDictionary<TKey, TValue>

The `OrderedDictionary<TKey, TValue>` is a custom implementation of a dictionary that maintains the order of the elements as they are added. It is part of the `HeavyCavStudios.Core.Collections` namespace and combines the benefits of a dictionary with an ordered collection of keys.

Namespace

`HeavyCavStudios.Core.Collections`

Class Definition

`OrderedDictionary<TKey, TValue>` implements `IEnumerable<KeyValuePair<TKey, TValue>>`

This class allows you to store key-value pairs while maintaining the order in which keys are added, unlike the standard `Dictionary<TKey, TValue>` that does not preserve order.

Fields

- `m_Dictionary`: A private field that stores the key-value pairs (`Dictionary<TKey, TValue>`).
- `m_Keys`: A private list that stores the keys (`List<TKey>`), preserving the order in which they were added.

Properties

- `this[TKey key]`: Indexer that gets or sets the value associated with the specified key. Setting the value will add the key-value pair if it doesn't already exist.
- `Count`: Gets the number of elements contained in the dictionary (`int`).
- `Values`: Returns a list of all values in the dictionary in the order of their keys (`List<TValue>`).

Methods

- `Add(TKey key, TValue value)`: Adds a key-value pair to the dictionary.

  - **Parameters**:
    - `TKey key`: The key to add.
    - `TValue value`: The value to associate with the key.
  - If the key already exists, an exception will be thrown.

- `Remove(TKey key)`: Removes the key-value pair with the specified key.

  - **Parameters**:
    - `TKey key`: The key of the element to remove.
  - **Returns**: `bool` indicating whether the element was successfully removed.

- `Clear()`: Clears all the elements in the dictionary.

- `GetAtOrDefault(int index)`: Retrieves the value at the specified index or returns the default value if the index is out of range.

  - **Parameters**:
    - `int index`: The index of the value to retrieve.
  - **Returns**: `TValue` corresponding to the given index or `default(TValue)` if the index is out of range.

- `TryGetValue(TKey key, out TValue value)`: Attempts to get the value associated with the specified key.

  - **Parameters**:
    - `TKey key`: The key of the value to retrieve.
    - `out TValue value`: When this method returns, contains the value associated with the specified key, if it exists.
  - **Returns**: `bool` indicating whether the key was found.

- `GetEnumerator()`: Returns an enumerator that iterates through the ordered key-value pairs.

  - **Returns**: `IEnumerator<KeyValuePair<TKey, TValue>>`

- `IEnumerator IEnumerable.GetEnumerator()`: Explicit interface implementation to allow iteration.

Usage Example

```csharp
using HeavyCavStudios.Core.Collections;

class Program
{
    static void Main()
    {
        OrderedDictionary<string, int> orderedDict = new OrderedDictionary<string, int>();

        orderedDict.Add("one", 1);
        orderedDict.Add("two", 2);
        orderedDict.Add("three", 3);

        // Access a value by key
        int value = orderedDict["two"]; // value = 2

        // Enumerate through the ordered dictionary
        foreach (var kvp in orderedDict)
        {
            Console.WriteLine($"Key: {kvp.Key}, Value: {kvp.Value}");
        }

        // Get value by index
        int valueAtIndex = orderedDict.GetAtOrDefault(1); // valueAtIndex = 2

        // Remove a key-value pair
        orderedDict.Remove("two");

        // Clear the dictionary
        orderedDict.Clear();
    }
}
```

Key Features

- **Maintains Insertion Order**: Unlike standard dictionaries, `OrderedDictionary<TKey, TValue>` keeps track of the order in which keys are added.
- **Enumerator Support**: Supports iteration through all key-value pairs in insertion order.
- **Indexed Access**: Provides access to values by key as well as access by index, offering additional flexibility.

Notes

- Attempting to access a key that doesn't exist using the indexer will throw an exception. Use `TryGetValue` to avoid exceptions when keys might be missing.
- When adding a key-value pair, if the key already exists, it will throw an exception. To update the value for an existing key, use the indexer (`orderedDict[key] = newValue`).

Coroutines

DebouncedWithValue

The `DebouncedWithValue<T>` class is part of the `HeavyCavStudios.Core.Coroutines` namespace and provides a mechanism to debounce function calls with a value parameter. This is useful when you want to delay the execution of a method until a certain period of inactivity has passed.

Namespace

`HeavyCavStudios.Core.Coroutines`

Class Definition

`DebouncedWithValue<T>`

Fields

- **`m_WaitInSeconds`**: The time in seconds to wait before executing the handler (`float`).
- **`m_Handler`**: The action to execute after the debounce period (`Action<T>`).
- **`m_Debounced`**: Stores the current running coroutine (`IEnumerator`).

Methods

- **`DebouncedWithValue(Action<T> handler, float waitTime)`**: Constructor that initializes the debounce handler and wait time.

  - **Parameters**:
    - `Action<T> handler`: The handler to call after the debounce period.
    - `float waitTime`: The amount of time to wait before invoking the handler.

- **`Debounce(T value, MonoBehaviour context)`**: Initiates or restarts the debounce process with the provided value.

  - **Parameters**:
    - `T value`: The value to pass to the handler when called.
    - `MonoBehaviour context`: The MonoBehaviour used to start and stop coroutines.

Usage Example

```
using HeavyCavStudios.Core.Coroutines;
using UnityEngine;

public class DebounceExample : MonoBehaviour
{
    void Start()
    {
        DebouncedWithValue<int> debouncer = new DebouncedWithValue<int>
(HandleValue, 2.0f);
        debouncer.Debounce(42, this);
    }

    void HandleValue(int value)
    {
        Debug.Log("Debounced value: " + value);
    }
}
```

Key Features

- **Delayed Execution**: Executes the provided handler only after the specified wait time has passed without interruption.
- **Restartable Debounce**: If called again before the time has elapsed, the timer resets.

Notes

- Requires a `MonoBehaviour` context to manage coroutines.

DebouncedWithoutValue

The `DebouncedWithoutValue` class is part of the `HeavyCavStudios.Core.Coroutines` namespace and provides a mechanism to debounce function calls without requiring a value parameter.

Class Definition

`DebouncedWithoutValue`

Fields

- **`m_WaitInSeconds`**: The time in seconds to wait before executing the handler (`float`).
- **`m_Handler`**: The action to execute after the debounce period (`Action`).
- **`m_Debounced`**: Stores the current running coroutine (`IEnumerator`).

Methods

- **`DebouncedWithoutValue(Action handler, float waitTime)`**: Constructor that initializes the debounce handler and wait time.

  - **Parameters**:
    - `Action handler`: The handler to call after the debounce period.
    - `float waitTime`: The amount of time to wait before invoking the handler.

- **`Debounce(MonoBehaviour context)`**: Initiates or restarts the debounce process.

  - **Parameters**:
    - `MonoBehaviour context`: The MonoBehaviour used to start and stop coroutines.

Usage Example

```
using HeavyCavStudios.Core.Coroutines;
using UnityEngine;

public class DebounceExampleWithoutValue : MonoBehaviour
{
    void Start()
    {
        DebouncedWithoutValue debouncer = new
DebouncedWithoutValue(HandleAction, 2.0f);
        debouncer.Debounce(this);
    }

    void HandleAction()
    {
        Debug.Log("Debounced action executed");
    }
}
```

Key Features

- **Delayed Execution**: Executes the provided handler only after the specified wait time has passed without interruption.
- **Restartable Debounce**: If called again before the time has elapsed, the timer resets.

Notes

- Requires a `MonoBehaviour` context to manage coroutines.

WaitForSecondsPausable

The `WaitForSecondsPausable` class is part of the `HeavyCavStudios.Core.Coroutines` namespace and allows for a wait time that can be paused based on a given condition.

Class Definition

`WaitForSecondsPausable : CustomYieldInstruction`

Fields

- **`m_WaitTime`**: The time in seconds to wait (`float`).
- **`m_IsPaused`**: A function that determines if the wait should be paused (`Func<bool>`).

Properties

- **`keepWaiting`**: Determines if the yield instruction should keep waiting.
  - **Returns**: `bool` indicating whether the waiting should continue.

Constructor

- **`WaitForSecondsPausable(float time, Func<bool> pauseCondition)`**: Initializes the pausable wait time.
  - **Parameters**:
    - `float time`: The time in seconds to wait.
    - `Func<bool> pauseCondition`: A function that returns `true` if the wait should be paused.

Usage Example

```
using HeavyCavStudios.Core.Coroutines;
using UnityEngine;

public class WaitExample : MonoBehaviour
{
    bool isPaused = false;

    IEnumerator Start()
    {
        yield return new WaitForSecondsPausable(5.0f, () => isPaused);
        Debug.Log("Wait completed");
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.P))
        {
            isPaused = !isPaused;
        }
    }
}
```

Key Features

- **Pausable Waiting**: Allows waiting that can be paused based on a user-defined condition.
- **Custom Yield Instruction**: Extends `CustomYieldInstruction` to integrate seamlessly with Unity's coroutine system.

Notes

- Useful for scenarios where the wait time needs to be paused and resumed, such as game state changes or pauses.

Editor Utilities

AssetUtilities

The `AssetUtilities` class is part of the `HeavyCavStudios.Core.Editor` namespace and provides utility methods for working with assets in the Unity Editor.

Namespace

`HeavyCavStudios.Core.Editor`

Class Definition

`AssetUtilities`

Methods

- **`LoadAllAssetsOfType<T>()`**: Loads all assets of the specified type from the Unity project.
  - **Type Parameter**:
    - `T`: The type of asset to load. Must be a `ScriptableObject`.
  - **Returns**: A `List<T>` containing all assets of type `T` found in the project.

Usage Example

```
#if UNITY_EDITOR
using HeavyCavStudios.Core.Editor;
using UnityEditor;
using UnityEngine;

public class AssetLoaderExample : MonoBehaviour
{
    void Start()
    {
        List<MyScriptableObject> assets =
AssetUtilities.LoadAllAssetsOfType<MyScriptableObject>();
        foreach (var asset in assets)
        {
            Debug.Log("Loaded asset: " + asset.name);
        }
    }
}
#endif
```

Key Features

- **Editor-Only**: This utility is only available in the Unity Editor and helps streamline loading assets during development.
- **Generic Asset Loading**: Provides a simple way to load all assets of a specific type (`ScriptableObject`).

Notes

- This utility uses `AssetDatabase` and is therefore only available in the Unity Editor (`#if UNITY_EDITOR` directive). Attempting to use this class outside of the editor will result in compilation errors.

Extensions

CollectionExtensions

The `CollectionExtensions` class is part of the `HeavyCavStudios.Core.Extensions` namespace and provides utility extension methods for working with collections such as lists.

Namespace

`HeavyCavStudios.Core.Extensions`

Class Definition

`CollectionExtensions`

Methods

- **`GetRandomElement<T>(this List<T> list)`**: Returns a random element from the list.
  - **Parameters**:
    - `List<T> list`: The list to select a random element from.
  - **Returns**: A random element of type `T` from the list, or `default` if the list is empty.

Usage Example

```csharp
using HeavyCavStudios.Core.Extensions;
using System.Collections.Generic;
using UnityEngine;

public class RandomElementExample : MonoBehaviour
{
    void Start()
    {
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
        int randomElement = numbers.GetRandomElement();
        Debug.Log("Random element: " + randomElement);
    }
}
```

Key Features

- **Random Selection**: Provides an easy way to get a random element from a list.

Notes

- Returns `default` if the list is empty.

GameObjectExtensions

The `GameObjectExtensions` class is part of the `HeavyCavStudios.Core.Extensions` namespace and provides utility extension methods for working with `GameObject` instances in Unity.

Namespace

`HeavyCavStudios.Core.Extensions`

Class Definition

`GameObjectExtensions`

Methods

- **`GetChildByName(this GameObject gameObject, string name, bool recursive = false)`**: Finds a child `GameObject` by its name.
  - **Parameters**:
    - `GameObject gameObject`: The parent game object to search in.
    - `string name`: The name of the child to find.
    - `bool recursive`: Whether to search recursively through all child objects.
  - **Returns**: The child `GameObject` with the specified name or `null` if not found.

- **`GetChildren(this GameObject gameObject, Func<GameObject, bool> selector = null, bool recursive = false)`**: Retrieves a list of child game objects that match a given condition.

  - **Parameters**:
    - `GameObject gameObject`: The parent game object.
    - `Func<GameObject, bool> selector`: A predicate to filter the children (optional).
    - `bool recursive`: Whether to search recursively.
  - **Returns**: A list of child `GameObject` instances that match the given condition.

- **`GetChildByTag(this GameObject gameObject, string tag, bool recursive = false)`**: Finds a child `GameObject` by its tag.

  - **Parameters**:
    - `GameObject gameObject`: The parent game object to search in.
    - `string tag`: The tag to search for.
    - `bool recursive`: Whether to search recursively through all child objects.
  - **Returns**: The child `GameObject` with the specified tag or `null` if not found.

- **`GetComponentOfChild<T>(this GameObject gameObject, string name, bool recursive = false, bool throwIfChildNotFound = false)`**: Gets a component of a child `GameObject` by name.

  - **Parameters**:
    - `string name`: The name of the child to find.
    - `bool recursive`: Whether to search recursively.
    - `bool throwIfChildNotFound`: Whether to throw an exception if the child is not found.
  - **Returns**: The component of type `T` if found, or `default` if not found.

- **`GetComponentInHierarchy<T>(this GameObject gameObject, out GameObject componentOwner)`**: Searches for a component in the hierarchy, including the current `GameObject`, its children, and ancestors.

  - **Returns**: The component of type `T` if found, and sets `componentOwner` to the `GameObject` containing the component.

- **`GetComponentsInChildrenRecursive<T>(this GameObject gameObject)`**: Retrieves a list of components of type `T` in all child game objects recursively.

  - **Returns**: A list of components of type `T` found in all children.

- **`GetComponentInAncestors<T>(this GameObject gameObject, out GameObject componentOwner)`**: Finds a component of type `T` in the parent hierarchy.

  - **Returns**: The component of type `T` if found, and sets `componentOwner` to the `GameObject` containing the component.

- **`PerformActionOnHierarchyRecursive(this GameObject gameObject, Func<GameObject, bool> predicate, Action<GameObject> action)`**: Performs an action on the `GameObject` hierarchy based on a predicate.

  - **Parameters**:
    - `Func<GameObject, bool> predicate`: A condition to determine if the action should be performed.
    - `Action<GameObject> action`: The action to perform on the matching game objects.

- **`IsUIElement(this GameObject gameObject)`**: Checks if the `GameObject` is a UI element.

  - **Returns**: `true` if the `GameObject` has a `RectTransform` component, indicating it is a UI element.

Usage Example

```
using HeavyCavStudios.Core.Extensions;
using UnityEngine;

public class GameObjectExtensionsExample : MonoBehaviour
{
    void Start()
    {
        GameObject child = gameObject.GetChildByName("ChildObject");
        if (child != null)
        {
            Debug.Log("Found child: " + child.name);
        }

        List<GameObject> uiElements = gameObject.GetChildren(go =>
go.IsUIElement());
        Debug.Log("Number of UI elements: " + uiElements.Count);
    }
}
```

Key Features

- **Recursive Search**: Provides multiple ways to search and interact with children, parents, and components in the hierarchy.
- **Flexible Component Access**: Helps access components and perform actions in a more concise way.

Notes

- Recursive methods allow for deep searches through child objects, which may impact performance if used on large hierarchies.
- The `GetComponentInHierarchy` methods provide a convenient way to find components across multiple levels of the GameObject hierarchy.

Patterns

ICloneable

The `ICloneable<T>` interface is part of the `Core.Patterns.Cloning` namespace and provides a method for cloning objects of type `T`.

Namespace

`Core.Patterns.Cloning`

Interface Definition

`ICloneable<T>`

Methods

- **`Clone()`**: Creates a copy of the current instance.
  - **Returns**: A new instance of type `T` that is a clone of the current object.

Usage Example

```
using Core.Patterns.Cloning;

public class MyClass : ICloneable<MyClass>
{
    public int Value { get; set; }

    public MyClass Clone()
    {
        return new MyClass { Value = this.Value };
    }
}

class Program
{
    static void Main()
    {
        MyClass original = new MyClass { Value = 42 };
        MyClass clone = original.Clone();
        System.Console.WriteLine(clone.Value); // Output: 42
    }
}
```

Key Features

- **Generic Cloning**: Provides a standard way to implement cloning for custom types.

Notes

- This interface allows for type-safe cloning, ensuring that the cloned object is of the same type as the original.

Event System

EventBus

The `EventBus` class is part of the `HeavyCavStudios.Core.Patterns.Events` namespace and implements a global event management system that allows for subscribing, unsubscribing, and raising events across the application. It extends from `AbstractSingleton<EventBus>` to ensure only one instance is used throughout the project.

Namespace

`HeavyCavStudios.Core.Patterns.Events`

Class Definition

`EventBus : AbstractSingleton<EventBus>`

Fields

- **`m_EventLookup`**: A dictionary that holds event types (`Type`) as keys and `EventWrapper` objects as values, used to manage event handlers.

Methods

- **`Subscribe<T>(string name, EventHandler<T> handler) where T : class, IEvent`**: Subscribes a handler to an event of type `T`.
  - **Parameters**:
    - `string name`: The name of the event.
    - `EventHandler<T> handler`: The event handler to subscribe.
- **`Unsubscribe<T>(string name, EventHandler<T> handler) where T : class, IEvent`**: Unsubscribes a handler from an event of type `T`.
  - **Parameters**:
    - `string name`: The name of the event.
    - `EventHandler<T> handler`: The event handler to unsubscribe.
  - **Throws**: `EventMissingException` if the event type was never subscribed.
- **`Raise<T>(object sender, T args, bool raise = true) where T : class, IEvent`**: Raises an event of type `T`.
  - **Parameters**:

        - `object sender`: The sender of the event.
        - `T args`: The arguments for the event.
        - `bool raise`: If `true`, throws an exception if no handlers are found.
    - **Throws**: `EventMissingException` if the event type was never subscribed and `raise` is set to `true`.

- **`ClearAll()`**: Clears all events from the event lookup.

- **`Initialize()`**: Protected method to initialize the singleton instance.

Usage Example

```csharp
using HeavyCavStudios.Core.Patterns.Events;

public class MyEvent : IEvent
{
    public string Message { get; set; }
}

public class EventBusExample
{
    public void Example()
    {
        EventBus.Instance.Subscribe<MyEvent>("TestEvent", OnMyEvent);
        EventBus.Instance.Raise(this, new MyEvent { Message = "Hello, World!" });
        EventBus.Instance.Unsubscribe<MyEvent>("TestEvent", OnMyEvent);
    }

    private void OnMyEvent(object sender, MyEvent e)
    {
        Console.WriteLine(e.Message);
    }
}
```

Key Features

- **Global Event Management**: Provides a centralized way to manage events, reducing coupling between classes.
- **Type-Safe Event Handling**: Enforces type safety using generics.

Notes

- Events must implement the `IEvent` interface.
- Uses a singleton pattern to ensure only one instance of `EventBus` is present throughout the application.

EventWrapper

The `EventWrapper` class is used internally by `EventBus` to manage event handlers.

Namespace

`HeavyCavStudios.Core.Patterns.Events`

Class Definition

`EventWrapper`

Fields

- **`m_Name`**: The name of the event (`string`).
- **`m_Type`**: The type of the event (`Type`).
- **`m_Handlers`**: A list of handlers (`List<object>`) associated with the event.

Methods

- **`AddHandler<T>(EventHandler<T> handler)`**: Adds a handler to the list of event handlers.

  - **Parameters**:
    - `EventHandler<T> handler`: The handler to add.
  - **Throws**: `TypeMismatchException` if the handler type does not match the event type.

- **`RemoveHandler<T>(EventHandler<T> handler)`**: Removes a handler from the list of event handlers.

  - **Parameters**:
    - `EventHandler<T> handler`: The handler to remove.
  - **Throws**: `TypeMismatchException` if the handler type does not match the event type.

- **`Invoke<T>(object sender, T args)`**: Invokes all handlers associated with the event.

  - **Parameters**:
    - `object sender`: The sender of the event.
    - `T args`: The event arguments.
  - **Throws**: `TypeMismatchException` if the argument type does not match the event type.

Notes

- Handles type validation to ensure the correct handler is called for each event.

EventMissingException

The `EventMissingException` class is a custom exception thrown when an event that is expected to exist is missing.

Namespace

`HeavyCavStudios.Core.Patterns.Events`

Class Definition

`EventMissingException : Exception`

Constructor

- **`EventMissingException(string msg)`**: Initializes a new instance of the `EventMissingException` class with a specified error message.

Notes

- Used to indicate that an event was never subscribed before being raised or unsubscribed.

IEvent

The `IEvent` interface is a marker interface used to represent events in the `EventBus` system.

Namespace

`HeavyCavStudios.Core.Patterns.Events`

Interface Definition

`IEvent`

Notes

- Any class representing an event must implement the `IEvent` interface.

TypeMismatchException

The `TypeMismatchException` class is a custom exception thrown when there is a type mismatch during event handler operations.

Namespace

`HeavyCavStudios.Core.Patterns.Events`

Class Definition

`TypeMismatchException : Exception`

Constructor

- **`TypeMismatchException(string msg)`**: Initializes a new instance of the `TypeMismatchException` class with a specified error message.

Notes

- Used to indicate a mismatch between expected and actual types during event handling operations.

Factory Pattern

IFactory

The `IFactory<T>` interface is part of the `HeavyCavStudios.Core.Patterns.Factory` namespace and provides a standard way to implement a factory pattern for creating instances of type `T`.

Namespace

`HeavyCavStudios.Core.Patterns.Factory`

Interface Definition

`IFactory<out T>`

Methods

- **`Create()`**: Creates and returns a new instance of type `T`.
  - **Returns**: A new instance of type `T`.

Usage Example

```
using HeavyCavStudios.Core.Patterns.Factory;

public class MyClass
{
    public string Name { get; set; }
}

public class MyClassFactory : IFactory<MyClass>
{
    public MyClass Create()
    {
        return new MyClass { Name = "New Instance" };
    }
}
```

```
public class FactoryExample
{
    public void Example()
    {
        IFactory<MyClass> factory = new MyClassFactory();
        MyClass instance = factory.Create();
        Console.WriteLine(instance.Name); // Output: New Instance
    }
}
```

Key Features

- **Object Creation**: Provides a standard way to create objects, promoting the separation of object creation from usage.

Notes

- The `Create` method is designed to return a new instance, allowing for greater flexibility in object instantiation.

Object Pooling Pattern

ObjectPool

The `ObjectPool<T>` class is part of the `HeavyCavStudios.Core.Patterns.Pooling` namespace and provides a mechanism to manage the reuse of objects, minimizing the cost of creating and destroying them repeatedly.

Namespace

`HeavyCavStudios.Core.Patterns.Pooling`

Class Definition

`ObjectPool<T>`

Fields

- **`m_Pool`**: A stack that holds instances of objects (`Stack<T>`).
- **`m_Factory`**: A factory used to create new instances of objects (`IFactory<T>`).

Constructor

- **`ObjectPool(int initialSize, IFactory<T> factory)`**: Initializes the pool with a specified number of objects created by the provided factory.
  - **Parameters**:

    - `int initialSize`: The initial number of objects in the pool.
    - `IFactory<T> factory`: The factory used to create new instances.

Methods

- `GetObject()`: Retrieves an object from the pool. If the pool is empty, a new object is created using the factory.

  - **Returns**: An instance of type `T`.

- `ReturnObject(T obj)`: Returns an object to the pool for reuse.

  - **Parameters**:
    - `T obj`: The object to return to the pool.

Usage Example

```
using HeavyCavStudios.Core.Patterns.Factory;
using HeavyCavStudios.Core.Patterns.Pooling;

public class Bullet
{
    public void Fire() { /* Fire the bullet */ }
}

public class BulletFactory : IFactory<Bullet>
{
    public Bullet Create() => new Bullet();
}

public class GameManager
{
    ObjectPool<Bullet> bulletPool;

    public GameManager()
    {
        bulletPool = new ObjectPool<Bullet>(10, new BulletFactory());
    }

    public void Shoot()
    {
        Bullet bullet = bulletPool.GetObject();
        bullet.Fire();
        bulletPool.ReturnObject(bullet);
    }
}
```

Key Features

- **Object Reuse**: Reduces the overhead of creating and destroying objects, improving performance.
- **Integration with Factory Pattern**: Uses the `IFactory<T>` interface to create new instances when necessary.

Notes

- The pool size can dynamically increase as more objects are requested.

ObjectPoolManager

The `ObjectPoolManager` class is part of the `HeavyCavStudios.Core.Patterns.Pooling` namespace and acts as a centralized manager for multiple object pools, ensuring easy access and management of pools across the application.

Namespace

`HeavyCavStudios.Core.Patterns.Pooling`

Class Definition

`ObjectPoolManager : AbstractSingleton<ObjectPoolManager>`

Fields

- **`k_DefaultSize`**: The default size for new pools (`int`).
- **`m_TypeToPool`**: A dictionary that maps types to their respective object pools (`Dictionary<Type, object>`).

Methods

- **`CreatePool<T>(IFactory<T> factory, int initialSize)`**: Creates a new pool for the specified type if it does not already exist.

  - **Parameters**:
    - `IFactory<T> factory`: The factory used to create new instances.
    - `int initialSize`: The initial number of objects in the pool.
  - **Throws**: `Exception` if a pool for the specified type already exists.

- **`PoolOfTypeExists<T>()`**: Checks if a pool for the specified type exists.

  - **Returns**: `bool` indicating whether a pool of the specified type exists.

- **`GetObject<T>()`**: Retrieves an object of the specified type from the corresponding pool.

  - **Returns**: An instance of type `T`.
  - **Throws**: `Exception` if no pool for the specified type is found.

- `ReturnObject<T>(T obj)`: Returns an object to the corresponding pool.
  - **Parameters**:
    - `T obj`: The object to return.
- `Clear()`: Clears all the pools managed by the `ObjectPoolManager`.

Usage Example

```
using HeavyCavStudios.Core.Patterns.Factory;
using HeavyCavStudios.Core.Patterns.Pooling;

public class Enemy
{
    public void Spawn() { /* Spawn logic */ }
}

public class EnemyFactory : IFactory<Enemy>
{
    public Enemy Create() => new Enemy();
}

public class Game
{
    public Game()
    {
        ObjectPoolManager.Instance.CreatePool<Enemy>(new EnemyFactory(),
5);
    }

    public void SpawnEnemy()
    {
        Enemy enemy = ObjectPoolManager.Instance.GetObject<Enemy>();
        enemy.Spawn();
        ObjectPoolManager.Instance.ReturnObject(enemy);
    }
}
```

Key Features

- **Centralized Pool Management**: Manages multiple object pools, providing an easy interface for creating, retrieving, and returning objects.
- **Singleton Pattern**: Ensures only one instance of `ObjectPoolManager` exists, making it accessible globally.

Notes

- Using `ObjectPoolManager` helps reduce redundancy by centralizing pool creation and management logic.

Singleton Pattern

AbstractMonoSingleton

The `AbstractMonoSingleton<T>` class is part of the `HeavyCavStudios.Core.Patterns.Singleton` namespace and provides a base class for creating singleton instances of MonoBehaviour types in Unity. This allows easy access to a single instance of a component while ensuring only one exists at runtime.

Namespace

`HeavyCavStudios.Core.Patterns.Singleton`

Class Definition

`AbstractMonoSingleton<T> : MonoBehaviour where T : AbstractMonoSingleton<T>`

Fields

- **`k_InstanceProperty`**: A constant string used for instance property reference (`string`).
- **`k_InitializedProperty`**: A constant string used for the initialized property reference (`string`).
- **`Initialized`**: A static action triggered when the singleton is initialized (`Action`).
- **`Instance`**: The static instance of the singleton (`T`).

Methods

- **`Awake()`**: Handles the instantiation logic and ensures only one instance of the singleton exists.
  - **Notes**: Supports `DontDestroyOnLoad` to persist the instance across scenes.
- **`Instantiate()`**: Abstract method that must be implemented by derived classes to handle additional setup.
- **`OnDestroy()`**: Sets the instance to `null` when the singleton is destroyed.

Usage Example

```
using HeavyCavStudios.Core.Patterns.Singleton;
using UnityEngine;

public class GameManager : AbstractMonoSingleton<GameManager>
{
    protected override void Instantiate()
    {
        Debug.Log("GameManager instantiated.");
    }

    public void StartGame()
    {
        Debug.Log("Game started.");
    }
}

public class GameLauncher : MonoBehaviour
{
    void Start()
    {
        GameManager.Instance.StartGame();
    }
}
```

Key Features

- **MonoBehaviour Singleton**: Ensures only one instance of a MonoBehaviour exists in the scene.
- **Initialization Callback**: Provides an `Initialized` action to notify when the singleton is fully initialized.

Notes

- This pattern is useful for managers or services that need to be accessed globally within the Unity scene.

AbstractSingleton

The `AbstractSingleton<T>` class is part of the `HeavyCavStudios.Core.Patterns.Singleton` namespace and provides a base class for creating non-MonoBehaviour singleton instances. It ensures only one instance of a class is created, providing a global access point.

Namespace

`HeavyCavStudios.Core.Patterns.Singleton`

Class Definition

`AbstractSingleton<T> where T : AbstractSingleton<T>, new()`

Fields

- **`s_Instance`**: The static instance of the singleton (`T`).

Properties

- **`Instance`**: Retrieves the singleton instance, creating it if it does not already exist.

  - **Returns**: The singleton instance of type `T`.

Constructor

- `AbstractSingleton()`: Protected constructor to prevent multiple instances from being created.
  - **Throws**: `Exception` if an attempt is made to instantiate multiple instances.

Methods

- `Initialize()`: Abstract method that must be implemented by derived classes to handle additional setup when the singleton is created.

Usage Example

```csharp
using HeavyCavStudios.Core.Patterns.Singleton;
using System;

public class ConfigManager : AbstractSingleton<ConfigManager>
{
    protected override void Initialize()
    {
        Console.WriteLine("ConfigManager initialized.");
    }

    public void LoadConfig()
    {
        Console.WriteLine("Configuration loaded.");
    }
}

public class Program
{
    public static void Main()
    {
        ConfigManager.Instance.LoadConfig();
    }
}
```

Key Features

- **Non-MonoBehaviour Singleton**: Provides a simple way to implement the singleton pattern for non-MonoBehaviour classes.
- **Lazy Initialization**: Creates the instance when it is first accessed.

Notes

- This pattern is useful for utility classes, configuration managers, or other non-MonoBehaviour singletons that need a global point of access.

Reflection Utility

ReflectionUtility

The `ReflectionUtility` class is part of the `HeavyCavStudios.Core.Reflection` namespace and provides various utility methods for working with reflection in C#. These methods help retrieve information about assemblies, types, constructors, properties, and more.

Namespace

`HeavyCavStudios.Core.Reflection`

Class Definition

`ReflectionUtility`

Methods

- **`GetImplementedConstructorsForNonGenericInterface(Assembly assembly, Type interfaceType, List<Type> attributesToIgnore)`**: Retrieves constructors for all non-generic implementations of the specified interface in the given assembly.
  - **Parameters**:
    - `Assembly assembly`: The assembly to search within.
    - `Type interfaceType`: The non-generic interface type to look for.
    - `List<Type> attributesToIgnore`: List of attributes that, if present on a type, should exclude it from the search.
  - **Returns**: A list of functions that invoke the constructors for the implementations found.
- **`GetImplementedConstructorsForGenericInterface(Assembly assembly, Type genericInterfaceType, List<Type> attributesToIgnore)`**: Retrieves constructors for all generic implementations of the specified interface in the given assembly.
  - **Parameters**:
    - `Assembly assembly`: The assembly to search within.
    - `Type genericInterfaceType`: The generic interface type to look for.
    - `List<Type> attributesToIgnore`: List of attributes that, if present on a type, should exclude it from the search.
  - **Returns**: A dictionary mapping the generic type argument to a list of functions that invoke the constructors.
- **`ImplementsGenericInterface(object obj, Type genericInterfaceType)`**: Checks whether the given object implements a specific generic interface.
  - **Parameters**:

        - `object obj`: The object to check.
        - `Type genericInterfaceType`: The generic interface type to check for.
    - **Returns**: `bool` indicating whether the object implements the specified generic interface.

- **`ExtractGenericArgumentType(Type type, Type genericInterface)`**: Extracts the generic type argument from a given type that implements a specific generic interface.

    - **Parameters**:
        - `Type type`: The type to analyze.
        - `Type genericInterface`: The generic interface to look for.
    - **Returns**: The generic type argument (`Type`) or `null` if not found.

- **`TrySetProperty(object target, string propertyName, object propertyValue)`**: Attempts to set a property on the target object.

    - **Parameters**:
        - `object target`: The target object on which to set the property.
        - `string propertyName`: The name of the property to set.
        - `object propertyValue`: The value to set the property to.
    - **Returns**: `bool` indicating whether the property was successfully set.

- **`GetProperty<T>(object target, string property)`**: Retrieves a property value from the target object.

    - **Parameters**:
        - `object target`: The target object from which to get the property value.
        - `string property`: The name of the property to retrieve.
    - **Returns**: The property value cast to type `T`.

- **`GetProperty(object target, string property)`**: Retrieves a property value from the target object.

    - **Parameters**:
        - `object target`: The target object from which to get the property value.
        - `string property`: The name of the property to retrieve.
    - **Returns**: The property value as an `object`.

- **`GetImplementationsOfInterface<TInterface>(IEnumerable<Assembly> assemblies, bool instantiate = false)`**: Gets all types implementing a specified interface and optionally instantiates them if they have parameterless constructors.

    - **Parameters**:
        - `IEnumerable<Assembly> assemblies`: The assemblies to search within.
        - `bool instantiate`: If true, instantiates the found types.
    - **Returns**: A list of implementations or instances of the specified interface.

Helper Methods

- **`ContainsGenericInterfaceInAncestors(Type type)`**: Checks if the given type has a generic interface in its ancestors.

    - **Parameters**:
        - `Type type`: The type to check.
    - **Returns**: `bool` indicating whether the type has a generic interface in its ancestors.

- **`ContainsAnyOfAttributes(Type type, List<Type> attributes)`**: Checks if the given type contains any of the specified attributes.

    - **Parameters**:
        - `Type type`: The type to check.
        - `List<Type> attributes`: The list of attributes to look for.
    - **Returns**: `bool` indicating whether the type has any of the specified attributes.

Usage Example

```
using HeavyCavStudios.Core.Reflection;
using System;
using System.Collections.Generic;
using System.Reflection;

public interface IExampleInterface
{
    void Execute();
}

public class ExampleImplementation : IExampleInterface
{
    public void Execute()
    {
        Console.WriteLine("Executing...");
    }
}

public class ReflectionExample
{
    public void FindImplementations()
    {
        Assembly assembly = Assembly.GetExecutingAssembly();
        List<Func<object[], object>> constructors =
ReflectionUtility.GetImplementedConstructorsForNonGenericInterface(
            assembly,
            typeof(IExampleInterface),
            new List<Type>()
        );

        foreach (var constructor in constructors)
        {
            IExampleInterface instance =
(IExampleInterface)constructor.Invoke(null);
            instance.Execute();
        }
        }
    }
```

Key Features

- **Interface Implementation Search**: Finds all types that implement a given interface, both generic and non-generic.
- **Constructor Invocation**: Provides an easy way to invoke constructors via reflection.
- **Property Manipulation**: Allows getting and setting properties dynamically.

Notes

- These utilities are helpful for implementing dynamic type loading and service registration systems.

Core Serialization Utilities

MultidimensionalArray

The `MultidimensionalArray<T>` class is part of the `HeavyCavStudios.Core.Serialization` namespace and represents a multidimensional array while maintaining serialization capabilities for Unity. It uses a one-dimensional array to store data, allowing for efficient serialization.

Namespace

`HeavyCavStudios.Core.Serialization`

Class Definition

`MultidimensionalArray<T>`

Fields

- **`m_Rows`**: The number of rows in the array (`int`).
- **`m_Columns`**: The number of columns in the array (`int`).
- **`m_Array`**: A one-dimensional array to store the elements of the multidimensional array (`T[]`).

Constructor

- **`MultidimensionalArray(int rows, int columns)`**: Initializes the multidimensional array with the specified number of rows and columns.

Indexer

- **`this[int i, int j]`**: Provides two-dimensional indexing to access or modify the elements.
  - **Throws**: `IndexOutOfRangeException` if indices are out of range.

Key Features

- **Unity Serialization**: Optimized for use with Unity's serialization system.
- **Validation**: Ensures indices are within bounds during access.

NullableInt

The `NullableInt` class is a serializable representation of a nullable integer (`int?`), allowing for serialization compatibility in Unity.

Namespace

`HeavyCavStudios.Core.Serialization`

Class Definition

`NullableInt`

Fields

- **`m_HasValue`**: Indicates whether the value is present (`bool`).
- **`m_Value`**: Stores the actual value (`int`).

Properties

- **`Value`**: Gets or sets the nullable integer value.

Key Features

- **Custom Equality**: Provides custom equality and hash code methods to compare instances.

RuntimeScriptableObject

The `RuntimeScriptableObject` class provides a mechanism to create runtime instances of `ScriptableObject` types in Unity, while preserving a reference to the original asset.

Namespace

`HeavyCavStudios.Core.Serialization`

Class Definition

`RuntimeScriptableObject`

Fields

- **`m_Original`**: Stores the original `ScriptableObject` reference.

- `m_RuntimeInstance`: Stores the runtime-created instance of the `ScriptableObject`.

Methods

- `CreateRuntimeInstance()`: Instantiates a runtime version of the original `ScriptableObject`.
- `Get<T>()`: Retrieves the runtime instance, casted to the specified type.
- `Get()`: Retrieves the runtime instance as a `ScriptableObject`.

Key Features

- **Runtime Instantiation**: Creates runtime instances of `ScriptableObject` while retaining the original reference.

SerializableRange

The `SerializableRange<T>` class represents a serializable range with a minimum and maximum value.

Namespace

`HeavyCavStudios.Core.Serialization`

Class Definition

`SerializableRange<T>`

Fields

- `m_MinValue`: The minimum value of the range (`T`).
- `m_MaxValue`: The maximum value of the range (`T`).

Properties

- `MinValue`: Gets the minimum value.
- `MaxValue`: Gets the maximum value.

Key Features

- **Custom Equality**: Supports equality comparison and provides a custom hash code implementation.

SerializableType

The `SerializableType` class allows a type to be serialized by storing its name as a string.

Namespace

`HeavyCavStudios.Core.Serialization`

Class Definition

`SerializableType`

Fields

- **`m_TypeName`**: Stores the fully qualified type name as a string (`string`).

Properties

- **`Type`**: Retrieves the `Type` object from the stored type name.

Constructor

- **`SerializableType(Type type)`**: Initializes the `SerializableType` with the given `Type`.

Key Features

- **Type Retrieval**: Allows for serialization of types by storing their assembly-qualified names.

SerializedDictionary<TKey, TValue>

The `SerializedDictionary<TKey, TValue>` class is a custom dictionary that supports Unity serialization. It serializes the dictionary keys and values into lists for Unity's serialization system.

Namespace

`HeavyCavStudios.Core.Serialization`

Class Definition

`SerializedDictionary<TKey, TValue> : Dictionary<TKey, TValue>, ISerializationCallbackReceiver`

Fields

- **`m_KeyData`**: Stores the keys for serialization (`List<TKey>`).
- **`m_ValueData`**: Stores the values for serialization (`List<TValue>`).

Methods

- **`OnAfterDeserialize()`**: Restores the dictionary from serialized lists after deserialization.
- **`OnBeforeSerialize()`**: Prepares the dictionary for serialization by converting it to lists.

Key Features

- **Unity Serialization**: Enables serialization of dictionary data for Unity.

Usage Example

```
using HeavyCavStudios.Core.Serialization;
using System;
using UnityEngine;

public class ExampleUsage : MonoBehaviour
{
    [SerializeField] SerializedDictionary<string, int>
serializedDictionary;

    void Start()
    {
        serializedDictionary = new SerializedDictionary<string, int>();
        serializedDictionary.Add("Key1", 100);
        Debug.Log(serializedDictionary["Key1"]);
    }
}
```

Key Features Summary

- **Serialization Support**: Provides multiple classes that facilitate the serialization of complex data structures for Unity.
- **Custom Utility Classes**: Includes classes like `NullableInt`, `SerializableRange`, `SerializedDictionary` for specialized use cases in Unity development.
- **Runtime and Reflection Utilities**: Offers tools like `RuntimeScriptableObject` to manage runtime assets, and `SerializableType` to serialize type information.

Notes

- These utilities are designed to integrate seamlessly with Unity's serialization system and enhance the flexibility of Unity projects.

Core Factories

GameObjectComponentFactory

The `GameObjectComponentFactory<T>` class is part of the `HeavyCavStudios.Core.Factories` namespace and implements the `IFactory<T>` interface to create components attached to GameObjects. This factory is particularly useful for generating GameObject components in Unity with control over their parent transform and default active state.

Namespace

`HeavyCavStudios.Core.Factories`

Class Definition

`GameObjectComponentFactory<T> : IFactory<T> where T : Component`

Fields

- **`m_Prefab`**: The GameObject prefab used for instantiation (`GameObject`).
- **`m_ParentTransform`**: The transform that acts as the parent of the instantiated GameObject (`Transform`).
- **`m_IsActiveByDefault`**: Specifies whether the instantiated GameObject should be active by default (`bool`).

Constructor

- **`GameObjectComponentFactory(GameObject prefab, Transform parentTransform, bool isActiveByDefault)`**: Initializes the factory with the specified prefab, parent transform, and active state.
  - **Parameters**:
    - `GameObject prefab`: The prefab to be instantiated.
    - `Transform parentTransform`: The parent transform for the instantiated GameObject.
    - `bool isActiveByDefault`: Whether the instantiated GameObject should be active by default.

Methods

- **`Create()`**: Creates and returns an instance of the component of type `T` attached to a new instance of the GameObject.
  - **Returns**: An instance of type `T`.

Key Features

- **Prefab Instantiation**: Creates instances of components from a given prefab.
- **Transform Management**: Allows specification of the parent transform for instantiated GameObjects.

GameObjectFactory

The `GameObjectFactory` class is part of the `HeavyCavStudios.Core.Factories` namespace and implements the `IFactory<GameObject>` interface to create GameObjects using a customizable instantiation function.

Namespace

`HeavyCavStudios.Core.Factories`

Class Definition

`GameObjectFactory : IFactory<GameObject>`

Fields

- **`m_CreateFunc`**: A function that defines how the GameObject should be instantiated (`Func<GameObject, GameObject>`).
- **`m_Prefab`**: The GameObject prefab used for instantiation (`GameObject`).

Constructor

- **`GameObjectFactory(GameObject prefab, Func<GameObject, GameObject> createFunc)`**: Initializes the factory with the specified prefab and create function.
  - **Parameters**:
    - `GameObject prefab`: The prefab to be instantiated.
    - `Func<GameObject, GameObject> createFunc`: A function that handles the instantiation logic.

Methods

- **`Create()`**: Creates and returns a new instance of the GameObject by applying the `m_CreateFunc` to the prefab.
  - **Returns**: A new instance of `GameObject`.

Key Features

- **Customizable Creation Logic**: Uses a function to customize how the prefab is instantiated.

Usage Example

```csharp
using HeavyCavStudios.Core.Factories;
using UnityEngine;

public class ExampleUsage : MonoBehaviour
{
    [SerializeField] GameObject prefab;

    void Start()
    {
        // Create a factory for instantiating prefabs with a custom function
        var factory = new GameObjectFactory(prefab, p => Instantiate(p));
        GameObject newObject = factory.Create();
        Debug.Log(newObject.name);
    }
}
```

Key Features Summary

- **Flexible GameObject and Component Creation**: Provides factories for instantiating GameObjects and their components, allowing for better code reuse and decoupling.
- **Integration with Factory Pattern**: Implements the `IFactory<T>` interface to standardize object creation.

Notes

- These factory classes are useful in Unity projects to streamline the process of creating GameObjects and components while providing control over their instantiation and hierarchy.