Fast Unity Creation Kit Manual

Unity3D toolkit for making games easier.

Patryk Pastuszka

Table of contents

Introduction

Fast Unity Creation Kit is a framework designed to make creating games in Unity faster and easier especially for beginners and solo developers. It is designed to be easy to use and understand, and to be as flexible as possible. It is designed to be used with modern Unity versions, however most of the features should work with older versions as well.

Concepts

Interfacing

Most of the features in Fast Unity Creation Kit are accessed through C# interfaces. This is done to make it easier to use the features in your own code, and to make it easier to understand how the features work.

Interfaces are a way to implement multiple inheritance in C#, and are used to define a logic class that can be implemented by other classes. With this solution Unity Component layer and logic layer can be separated, making it easier to understand and maintain the code.

Components

Some of the features in Fast Unity Creation Kit are implemented as Unity Components, so it can be quickly added to GameObjects in the scene. These components are designed to be plug-and-play with minimal configuration required.

Scriptable Objects

Fast Unity Creation Kit tries to avoid using Scriptable Objects as much as possible, as they can lead to a lot of performance issues and are difficult to debug. However, some features are implemented as Scriptable Objects - especially those that need to be shared between multiple GameObjects like configuration data.

Systems

There are many systems in Fast Unity Creation Kit that are designed to make creating games in Unity faster and easier. Most of them are based on other games or game development frameworks and used widely across multiple game genres.

This introduction contains a brief overview of the systems in Fast Unity Creation Kit. For more detailed information, please refer to the individual system documentation in this document.

Existing systems

- Core (Core) - The core system of Fast Unity Creation Kit. It contains the basic functionality that is used by other systems.
- - The building system is used to manage the building in the game. It is used to create and manage the building of structures, houses, etc.
- - The combat system is used to manage the combat in the game. It is used to create and manage the combat between the player and enemies.
- - The entities system is used to manage the entities in the game. It is used to create and manage the entities in the game - players, enemies, NPCs, in-world destructible objects, etc.
- - The farming system is used to manage the farming in the game. It is used to create and manage the farming of crops, animals, etc.
- - The dialogue system is used to manage the dialogue in the game. It is used to create conversations between characters in the game.
- - The economy system is used to manage the economy in the game. It is used to create and manage the economy of the game - gold, but also health is considered as a local currency of an entity.
- - The inventory system is used to manage the inventory of the player. It is used to store items, weapons, armor, etc.

- - The quest system is used to manage the quests in the game. It is used to create and manage quests for the player to complete.
- - The status system is used to manage the status of objects in the game. It is used mostly to provide status effects like poison, slow, etc.
- - The time system is used to manage the time in the game. It is used to call events when the time changes, like day-night cycle, turn-based combat, etc.
- - The UI system is used to manage the user interface in the game. It is used to create and manage the user interface of the game - health bars, inventory, etc.

Core

Core is a package of commonly used classes and interfaces that are used by other systems in Fast Unity Creation Kit.

The subsystems of the Core system are:

- Events (Events) - The events system is used to manage the events in the game. It is used to create and manage the events that can be triggered by the player or the game itself. This subsystem is designed to separate other systems from each other, so they can be easily replaced or extended.
- Identification (Identification) - The identification system is used to manage the identification of objects in the game. It is used to create and manage the identification of objects in the game - players, enemies, NPCs, in-world destructible objects, etc.
- Numerics (Numerics) - The numerics system is used to manage the numerical values in the game. It is used as an overlay to C# built-in types to provide easier and more flexible way to work with numbers via interfaces like `INumber`.
- Priority (Priority) - The priority system is used to manage the priority of objects in the game. It is used to create prioritized lists of objects, so they are automatically sorted.
- Utility (Utility) - The utility system is used to manage object properties. It can handle object with name, description, icons, prefabs, etc.
- Values (Values) - The values system is used to manage the values of objects in the game. It is used to create dynamically modifiable values - for example maximum health that can be affected by status effects or equipment.

Events

Events is a package of classes and interfaces that are used to manage the events in the game. It is used to create and manage the events that can be triggered by the player or the game itself. This subsystem is designed to separate other systems from each other, so they can be easily replaced or extended.

Concepts

Events are a way to communicate between different parts of the game. They are used to trigger actions in response to certain conditions, like player input, game state changes, etc. Events are a way to decouple different parts of the game, so they can be easily replaced or extended without affecting other parts of the game. Thanks to this system this project is completely modular and modules are independent of each other.

The core of Events system is the `EventChannel<TSelf>` class. It is a base class for all event channels in the game. It contains methods to register and unregister event listeners, and to trigger the event. It also contains separate generic type to add ability to pass data with the event.

```
public abstract class EventChannel :
EventChannelBase<EventChannelCallback>

public abstract class EventChannel<TChannelData> :
EventChannelBase<EventChannelCallback<TChannelData>>
        where TChannelData : IEventChannelData
```

This represents a local event channel that can be attached to an object.

There are also global event channels that are used to broadcast events to all objects in the game. They are implemented as singletons, so they can be accessed from anywhere in the game.

```
public abstract class GlobalEventChannel<TSelf> : EventChannel
        where TSelf : GlobalEventChannel<TSelf>, new()

public abstract class GlobalEventChannel<TSelf, TChannelData> :
EventChannel<TChannelData>
```

```
        where TSelf : GlobalEventChannel<TSelf, TChannelData>, new()
        where TChannelData : IEventChannelData
```

> In most cases you will use GlobalEventChannel, as it is more flexible and easier to use. Local events are a possibility if you need to trigger events only on a specific object or link events to a specific object when that event is used to communicate between components on the same object.

Creating Events

To create an event you need to extend `GlobalEventChannel<TSelf>` or `EventChannel<TSelf>` via a custom class. The `TSelf` type parameter should be the type of the class you are extending as.

Event Channels also support secondary type parameter `TData` which is the type of the data that will be passed with the event, that type needs to extend `IEventChannelData` interface.

```
public class MyGlobalEventChannel :
GlobalEventChannel<MyGlobalEventChannel>
{
    // You can also override Trigger method to add custom logic
    public override void Trigger()
    {
        Debug.Log("Hello world!");
    }
}
```

> If you don't call base Trigger method in your custom Trigger method, the event will not be triggered on listeners and thus the created event can be considered a virtual event (event without listeners).

You can also create an event that passes data with it.

```
public readonly struct MyEventData : IEventChannelData
{
    public readonly string message;

    public MyEventData(string message)
    {
        this.message = message;
    }
}

public class MyGlobalEventChannelWithData :
GlobalEventChannel<MyGlobalEventChannelWithData, MyEventData>
{

}
```

Listening to Events

To listen to an event you need to access the event channel and subscribe to it. For global events, you can use the static `RegisterEventListener` method to subscribe to the event. But you can also use the instance of the event channel and call `RegisterListener` method directly.

```
public class MyListener : MonoBehaviour
{
    private void Start()
    {
        MyGlobalEventChannel.RegisterEventListener(ExampleCallback);
    }

    private void ExampleCallback()
    {
        Debug.Log("Event triggered!");
    }
}
```

If you want to pass data with the event, you need to subscribe to the event with a callback that accepts the data as a parameter.

```
public class MyListener : MonoBehaviour
{
    private void Start()
    {

MyGlobalEventChannelWithData.RegisterEventListener(ExampleCallback);
    }

    private void ExampleCallback(MyEventData data)
    {
        Debug.Log("Event triggered with data: " + data.Message);
    }
}
```

Triggering Events

To trigger an event you need to access the event channel and call the `Trigger` method. Global events also have stacic `TriggerEvent` method that can be used to trigger the event from anywhere in the game.

```
public class MyTrigger : MonoBehaviour
{
    private void Start()
    {
        MyGlobalEventChannel.TriggerEvent();
    }
}
```

If you want to pass data with the event, you need to call the `Trigger` method with the data as a parameter.

```
public class MyTrigger : MonoBehaviour
{
    private void Start()
    {
        MyGlobalEventChannelWithData.TriggerEvent(new MyEventData("Hello
world!"));
    }
}
```

Unsubscribing from Events

To unsubscribe from an event you can access the instance and use the `UnregisterListener` method or when using global events you can use the static `UnregisterEventListener` method.

For more reference go back up to the ***Listening to Events*** section.

Handling local events

Local events are events that are only triggered on the object that the event channel is attached to. You do everything almost the same as with global events, but you need to use `EventChannel<TSelf>` instead of `GlobalEventChannel<TSelf>` and you need to call methods on the instance of the event channel.

Identification

Identification is a package that contains basic identifier and identifiable object framework. Identifiers are `readonly structs` to ensure that they are immutable and can be used as keys.

Core interfaces

IIdentifier

Represents an identifier - a unique or non-unique identifier of an object.

IUniqueIdentifier

Represents a unique identifier - an identifier that is unique within a context.

INumberIdentifier

Represents a number identifier - an identifier that is a number as a value. `TNumber` is a type of the number that inherits from `INumber` interface from `Core.Numerics` package.

IIdentifiable

Represents an object that can be identified by an identifier. `TIdentifier` is a type of the identifier which inherits from `IIdentifier` interface.

IUniqueIdentifiable

Represents an object that can be identified by a unique identifier. `TIdentifier` is a type of the identifier which inherits from `IIdentifier` interface. This interface does not require `TIdentifier` to inherit from `IUniqueIdentifier` interface as it is in programmer's responsibility to ensure that the identifier is unique within a given context.

> ⚠ While using IUniqueIdentifiable interface, it is programmer's responsibility to ensure that the identifier is unique within a given context.

Implemented identifiers

Snowflake128

A 128-bit identifier that is generated using the Snowflake algorithm. It is guaranteed to be unique within a given context. First 64 bits are a timestamp, next 32 bits are identifier data (e.g. worker ID), next 16 bits are another pack of identifier data (e.g. process ID), next 8 bits are reserved and remaining 8 bits indicates if the identifier is null.

Numeric identifiers

- ID8 - A simple 8-bit number used as an identifier.
- ID16 - A simple 16-bit number used as an identifier.
- ID32 - A simple 32-bit number used as an identifier.
- ID64 - A simple 64-bit number used as an identifier.

Creating custom identifiers

You can also create custom identifiers by implementing `IIdentifier` interface. An example of a custom identifier is shown below:

```
public readonly struct CustomIdentifier : IIdentifier
{
    // It is not recommended to use managed types within identifiers.
    // However this is just an example for demonstration purposes.
    public readonly string value;

    public CustomIdentifier(string value)
    {
        this.value = value;
    }
}
```

It is also recommended to implement IEquatable{TSelf} interface for custom identifiers.

⚠ It is not recommended to use managed types within identifiers as it may lead to performance issues.

Numerics

Numerics is a low-level API for working with numbers in Fast Unity Creation Kit. It provides a set of interfaces and classes that can be used to work with numbers in a more flexible way than the built-in C# types. Thanks to included interfaces and primitive structures you can easily extend the functionality of the built-in types and create your own types that can be used in the same way as the built-in types.

Concepts

The generic concept of the Numerics API is that all numbers can be cast to their underlying type and back without any cost. This is done thanks to explicit structure implementations with pointer/reinterpret casting, which compiled by IL2CPP compiler or Burst compiler will be optimized to the same code as built-in types.

For example:

```
public void Test()
{
    float32 a = 1.0f;
    float32 b = 2.0f;

    float32 c = a + b;
}
```

Will be compiled to same code as:

```
public void Test()
{
    float a = 1.0f;
    float b = 2.0f;

    float c = a + b;
}
```

Because at assembly-level types do not exist. This allows for way more flexible numbers approach and `INumber` interface used to provide an easy way to require a number as a

generic type in your methods or classes.

Interfaces

INumber

This is base interface for all numbers in Numerics API. It provides basic arithmetic operations, however those operations are only used if direct number type is not known - if you perform an operation where at least one of the operands is interface itself.

Those math operations requires both numbers to support generic version of `ISupportsFloatConversion` interface.

ISupportsFloatConversion

Represents a number that can be converted to float (or double). This is used to provide a way to convert any number to float or double, which is required for basic arithmetic operations on unknown types. This is especially useful for `INumber` interface, where you can't know the exact type of the number and thus for modifiable values / modifiers.

IWithMaxLimit

Represents that an object has maximum limit. This is not supported by numbers themselves, but is a numeric-related utility interface that can be used to provide maximum limit for a number. An example usage can be status stacking with a limit of stack count - like burning status can stack up to 5 times.

IWithMinLimit

Same as `IWithMaxLimit`, but for minimum limit.

IWithDefaultValue

Represents that an object has default value. This is not supported by numbers themselves, but is a numeric-related utility interface that can be used to provide default value for a number. An example could be a modifiable value that has default value like health points with default value of 100.

> ⚠ In case of **IWithMaxLimit**, **IWithMinLimit** and **IWithDefaultValue** interfaces, you should implement them in your class and provide custom logic for handling those limits and default values. The interfaces are only utility markers used to indicate your intentions.

IFloatingPointNumber

Marker interface indicating that number is floating point number.

ISignedNumber

Marker interface indicating that number is signed number.

IUnsignedNumber

Marker interface indicating that number is unsigned number.

IVectorizedNumber

Marker interface indicating that number is vectorized number - tries to use SIMD/AVX instructions for calculations.

Supported numbers

The implemented numbers are:

- `float32` - 32-bit floating point number with internal value of `float`
- `float64` - 64-bit floating point number with internal value of `double`
- `int8` - 8-bit signed integer with internal value of `sbyte`
- `int16` - 16-bit signed integer with internal value of `short`
- `int32` - 32-bit signed integer with internal value of `int`
- `int64` - 64-bit signed integer with internal value of `long`
- `uint8` - 8-bit unsigned integer with internal value of `byte`
- `uint16` - 16-bit unsigned integer with internal value of `ushort`
- `uint32` - 32-bit unsigned integer with internal value of `uint`
- `uint64` - 64-bit unsigned integer with internal value of `ulong`
- `v128` - 128-bit vectorized number with internal value of `int4`
- `v256` - 256-bit vectorized number with internal value of `int4x2`
- `v512` - 512-bit vectorized number with internal value of `int4x4`

If you're unfamiliar with **int4**, **int4x2** and **int4x4** types, they are numbers that are included in Unity.Mathematics package and are used to provide SIMD/AVX instructions for calculations. They are not directly supported by Numerics API, but are used to provide vectorized numbers.

Priority

Priority is a simple package that allows to sort objects in lists by their priority. It can be either used to create a `PrioritizedList<TObject>` of objects which is automatically sorted when a new object is "added" (aka. inserted) or to use a custom extension method `SortByPriority<TObject>` provided in `PriorityExtensions` class to sort any `IList<TObject>` by priority.

All `TObject` should inherit from `IPrioritySupport` interface which provides a `Priority` property.

> The priority is sorted in ascending order, so the lower the priority, the closer to the beginning of the list the object will be.
>
> **For example:** object A with priority 3 will be always before object B with priority 5. If both objects have the same priority, the order of insertion is unknown and the order of objects is not guaranteed.
>
> In case of PrioritizedList, the order of insertion allows to somewhat predict the order of objects with the same priority - objects will always be sorted by the inverse order of insertion - the first inserted object will be at the end of the list. You can consider this similar to a reverse-order sorting.

Interfaces

IPrioritySupport

This interface should be implemented by all objects that should be sorted by priority.

Example usage

Example prioritized object:

```
public class PrioritizedObjectA : IPrioritySupport
{
    public uint Priority => 3;

     public int CompareTo(IPrioritySupport other) =>
```

```
Priority.CompareTo(other.Priority);
}

public class PrioritizedObjectB : IPrioritySupport
{
    public uint Priority => 5;

     public int CompareTo(IPrioritySupport other) =>
Priority.CompareTo(other.Priority);
}
```

Example usage of `PrioritizedList<TObject>`:

```
var list = new PrioritizedList<IPrioritySupport>();
list.Add(new PrioritizedObjectB());
list.Add(new PrioritizedObjectA());

// list[0] is PrioritizedObjectA
// list[1] is PrioritizedObjectB
```

Of course, you can use the extension method `SortByPriority<TObject>` on any `IList<TObject>`:

```
var list = new List<IPrioritySupport>();
list.Add(new PrioritizedObjectB());
list.Add(new PrioritizedObjectA());

list.SortByPriority();

// list[0] is PrioritizedObjectA
// list[1] is PrioritizedObjectB
```

It is strongly recommended to avoid using SortByPriority on the list as it is not as efficient as PrioritizedList.

Utility

Utility is a package of commonly used classes and interfaces that support other systems of Fast Unity Creation Kit. It contains quite a bunch of other subsystems that can manage object properties like initialization state, name, description, icons, prefabs, etc.

The subsystems of the Utility system are:

- Initialization (Initialization) - The initialization system is used to manage the initialization of objects in the game.
- Properties (Properties and configuration) - The properties system is used to manage the properties of objects in the game.
- Singleton (Singleton) - The singleton system is used to manage the singleton objects in the game. It is used to create singleton objects that can be accessed from anywhere in the game.

Initialization

Initialization is a package that contains the `IInitializable` interface responsible for initializing objects in the game. Any initializable object should implement this interface and provide custom logic calls to request initialization when needed.

Implementation

You need to implement the `IInitializable` interface in your class to make it initializable.

Example for regular C# class:

```
public class MyInitializable : IInitializable
{
    bool IInitializable.InternalInitializationStatusStorage { get; set;
}

    void IInitializable.OnInitialize()
    {
        // Initialization logic
        Debug.Log("MyInitializable initialized");
    }

    public MyInitializable()
    {
        // Request initialization
        this.Initialize();
    }
}
```

Example for Unity MonoBehaviour:

```
public class MyInitializable : MonoBehaviour, IInitializable
{
    bool IInitializable.InternalInitializationStatusStorage { get; set;
}

    void IInitializable.OnInitialize()
```

```
    {
        // Initialization logic
        Debug.Log("MyInitializable initialized");
    }

    private void Awake()
    {
        // Request initialization
        this.Initialize();
    }
}
```

Property InternalInitializationStatusStorage on the interface is used to store the initialization status of the object. This is an internal property and should not be accessed directly, for that purposes you should use `IsInitialized` property.

Initialization with multiple entry paths

Initialization can be requested multiple times, but it will be executed only once. This is useful when you need to initialize an object from multiple entry points, and you don't know which one will be called first.

Also, if you want to avoid using this.Initialize() method you can use `EnsureInitialized()` method that will initialize the object if it's not initialized yet. This method is clearer and more readable in some cases - when you want to ensure that the object is initialized, but you are not sure if it was already initialized.

Using `EnsureInitialized()` method requires casing the object to `IInitializable` interface.

Properties and configuration

Properties are used to define a generic structure for basic object information, especially things like names, description, icons, prefabs, etc. This system is used to manage the properties of objects in the game.

Concepts

All properties have a context in which they exist. The context is a class that implements the `IUsageContext` interface. Thanks to this, the object can have multiple names/descriptions/icons/prefabs, etc. depending on the context in which it is used.

> With the properties system you can have both an icon for a enemy that is displayed in the quest log and a image that is displayed in bestiary.

Usage

To use the properties system, you need to create a class that implements the `IUsageContext` interface.

```
public class BestiaryContext : IUsageContext{}

public class QuestLogContext : IUsageContext{}
```

Then you can use properties in your own classes.

```
public class Enemy : MonoBehaviour, IWithIcon<BestiaryContext>,
IWithIcon<QuestLogContext>
{
    [SerializeField] private Sprite bestiaryIcon;
    [SerializeField] private Sprite questLogIcon;

    Sprite IWithIcon<BestiaryContext>.Icon => bestiaryIcon;
    Sprite IWithIcon<QuestLogContext>.Icon => questLogIcon;
}
```

Of course there are way more properties that you can use.

Properties

The properties system contains the following properties:

- IWithName - Interface that allows you to get the name of the object.
- IWithLocalizedNames - Interface that allows you to get the localized names of the object (compatible with Unity.Localization package)
- IWithDescription - Interface that allows you to get the description of the object.
- IWithLocalizedDescription - Interface that allows you to get the localized description of the object (compatible with Unity.Localization package)
- IWithIcon - Interface that allows you to get the icon of the object.
- IWithPrefab - Interface that allows you to get the prefab of the object - this one requires to specify prefab type in the generic parameter.
- IWithConfiguration - Interface that allows you to get the configuration of the object.
- IWithAssetReference - Interface that allows you to get the asset reference for the object - compatible with Unity.Addressables package.

There are also some additional interfaces that require implementation of custom logic:

- IConditionallyRemovable - Interface that allows you to check if the object should be removed.

About the configuration

By default, Fast Unity Creation Kit uses standard C# classes for logic processing which is not the best way to integrate it with Unity. That's why the configuration is a `ScriptableObject` that can be created in the Unity Editor. This way you can easily create a configuration for your object and assign it to the object.

To implement it within FastUnityCreationKit you need to create a class that implements the `IWithConfiguration` interface.

```
public class EnemyConfiguration : ScriptableObject
{
    [SerializeField] private LocalizedString name;
    [SerializeField] private LocalizedString description;
    [SerializeField] private Sprite icon;
}
```

Then you can use it in your class.

```
public abstract class Enemy : MonoBehaviour,
IWithConfiguration<EnemyConfiguration>,
 IWithLocalizedName<BestiaryContext>,
IWithLocalizedDescription<BestiaryContext>,
 IWithIcon<BestiaryContext>
{
    [SerializeField] private EnemyConfiguration configuration;

    EnemyConfiguration
IWithConfiguration<EnemyConfiguration>.Configuration => configuration;

    LocalizedString IWithLocalizedName<BestiaryContext>.Name =>
configuration.name;
    LocalizedString
IWithLocalizedDescription<BestiaryContext>.Description =>
configuration.description;
    Sprite IWithIcon<BestiaryContext>.Icon => configuration.icon;
}
```

This solution allows for easy proxy-passing of the configuration to the object and way better safety on exposing object data to other systems like UI as you can easily validate if desired context is supported by the object and throw an exception if not.

If you want you can also skip implementation of configuration and hard-code the values in the object, but this is not recommended as it is not a good practice to hard-code values in the code.

Alternatively you can use constants that are stored in single static class and use them with the "Properties" system.

Example for the configuration with constants:

```
public static class EnemyConfigurationConstants
{
    public const string ENEMY_NAME_BESTIARY_WOLF = "Wolf";
    public const string ENEMY_DESCRIPTION_BESTIARY_WOLF = "A wolf is a
large canine native to North America.";
}

public class Enemy : MonoBehaviour, IWithName<BestiaryContext>,
IWithDescription<BestiaryContext>
{
    string IWithName<BestiaryContext>.Name =>
EnemyConfigurationConstants.ENEMY_NAME_BESTIARY_WOLF;
    string IWithDescription<BestiaryContext>.Description =>
EnemyConfigurationConstants.ENEMY_DESCRIPTION_BESTIARY_WOLF;
}
```

That allows for implementation of the configuration while avoiding the need to create a `ScriptableObject` for each type of object - this solution is intended for small projects where the configuration is not a big deal or projects maintained mostly by software developers who can easily handle code-level configuration.

Property calls are often inlined by the compiler (JIT or IL2CPP), so there should be no performance overhead, but this is not guaranteed as it was not properly tested (it is just an assumption based on the observation and analysis of the generated code).

Singleton

Singleton is an easy way to create a class that has only one instance and provides a global point of access to it.

Example implementations

```
public sealed class SingletonExample : ISingleton<SingletonExample>
{

}

public sealed class MonoSingletonExample :
IMonoBehaviourSingleton<MonoSingletonExample>
{

}
```

Acquiring the singleton instance

```
SingletonExample singleton = ISingleton<SingletonExample>.GetInstance();
MonoSingletonExample monoSingleton =
IMonoBehaviourSingleton<MonoSingletonExample>.GetInstance();
```

This is way simpler than creating a singleton class with a static instance and a static method to access it.

> ⚠ It is not recommended to add custom static Instance properties to singleton classes that implement the ISingleton interface as it may lead to undefined behaviour as ISingleton instance already contains a protected static Instance property.
>
> This implementation/usage is not supported and any issues related to it will not be addressed.

Values

Values is a subsystem for handling custom values assigned to objects. Even if it supports adding static values to object instances, it is mainly used for dynamic values that are calculated on the fly.

Low-level Interfaces

- IValue - represents a value and has a property to return the current value.
- IModifiableValue - represents a value that can be modified on the fly either by arithmetic or modifiers.
- IModifier - represents a modifier that can be applied to a value.

High-level API

StaticValue

An abstract class that takes `INumber` as a type parameter. It is used to represent a non-changing value assigned to an object. It is rarely used and was implemented to provide compatibility with the dynamic values via the `IValue` interface.

> It is highly recommended to avoid using `StaticValue` and use `ModifiableValue` instead.

ModifiableValue

An abstract class that takes `INumber` as a type parameter. It is used to represent a value that can be modified on the fly. It is the most commonly used value type.

It has been designed to allow for easy modification of the value by using arithmetic operations or applying modifiers. This value type is used by features like Resources, Stats, and other dynamic values.

> It is heavily recommended to use `ModifiableValue` for any entity property like speed, max health etc. as it allows for easy modification and handling of

modifiers that can be applied to the value by things like buffs, debuffs, equipment, etc.

Creating a custom value

To create a custom value, you need to create a class that inherits from `ModifiableValue` and implement the abstract methods.

```
public sealed class MaxHealthValue : ModifiableValue<int32>
{

}
```

You can also set up an initial value in the constructor, however that approach is not recommended.

Value set in the constructor:

```
public sealed class MaxHealthValue : ModifiableValue<int32>
{
    public MaxHealthValue()
    {
        SetBaseValue(100);
    }
}
```

It is recommended to set the initial value using IWithDefaultValue interface from the Numerics package.

Example of setting the initial value using IWithDefaultValue:

```
public sealed class MaxHealthValue : ModifiableValue<int32>,
IWithDefaultValue<int32>
{
    public MaxHealthValue()
    {
        SetBaseValue(this.GetDefaultValue());
```

```
    }

    public int32 DefaultValue => 100;
}
```

Of course according to that also the `IWithMinLimit` and `IWithMaxLimit` interfaces can be used to set the limits of the value. Limits are respected both by arithmetic operations and modifiers.

Example value with limits:

```
public class ExampleLimitedValue : ModifiableValue<float32>,
IWithMinLimit<float32>, IWithMaxLimit<float32>
{
    public float32 MinLimit => -10f;
    public float32 MaxLimit => 10f;
}
```

Modifiers

Modifiers are used to modify the value of a `ModifiableValue`. They can be added and removed at any time. There are some basic modifiers implemented:

- AddFlatModifier - adds a flat value to the value (is executed as first modifier with priority 536_870_912)
- MultiplyModifier - multiplies the value by a given factor (is executed as second modifier with priority 1_073_741_824)
- AddModifier - adds a value to the value (is executed as third modifier with priority 2_147_483_648)

All modifiers support IWithPriority interface to set the priority of the modifier. The lower the priority value, the earlier the modifier is applied. Those classes are abstract and should be inherited to create custom modifiers (to give context to the modifier).

Example of a custom modifier:

```
public sealed class ExampleModifier : AddModifier<float32>
{
    public ExampleModifier(float32 value) : base(value)
    {
    }
}
```

Example usage:

```
var maxHealth = new MaxHealthValue();
maxHealth.AddModifier(new ExampleModifier(10));
```

This allows for easy interpretation of the modifiers (unfortunately at quite large overhead cost, but nothing forbids you to create a few simple modifiers and use them in the code instead of having modifier for each different context).

Removing modifiers is also possible:

```
maxHealth.RemoveModifier<ExampleModifier>();
```

Advanced modifier creation

You can also create way more advanced modifiers with custom logic. To do that you need to create a class that implements the `IModifier` interface.

```
public sealed class ExampleModifier : IModifier<float32>
{
    public uint Priority => 3_000_000_000; // Last in the queue
(usually)

     public void Apply<TNumberType>(IModifiableValue<TNumberType> value)
            where TNumberType : struct, INumber
        {
            if(value is not MaxHealthValue)
                return;

            if(amount is TNumberType valueToAdd)
```

```
                value.Multiply((float32) 0.5f);
        }

        public void Remove<TNumberType>(IModifiableValue<TNumberType>
value)
            where TNumberType : struct, INumber
        {
            if(value is not MaxHealthValue)
                return;

            if(amount is TNumberType valueToAdd)
                value.Multiply((float32) 2f);
        }

        public int CompareTo(IPrioritySupport other) =>
Priority.CompareTo(other.Priority);
}
```

This example is a purely custom modifier that halves the value of the `MaxHealthValue` when applied and doubles it when removed. It also can be applied only to the `MaxHealthValue` modifiable value thanks to additional safety check.

Arithmetic operations

All arithmetic operations are supported by the `ModifiableValue` class. They are implemented as methods that take a value to apply the operation to.

```
var maxHealth = new MaxHealthValue();
maxHealth.Add(10);
maxHealth.Multiply(2);
maxHealth.Divide(2);
maxHealth.Subtract(10);
```

> ⚠ **Modifiers and arithmetic operations are not compatible!**
>
> Arithmetic operations are designed to modify values that are somewhat predictable and can be easily calculated. Modifiers are designed to modify

values in a more complex dynamic way with custom logic.

A good example for arithmetic operations is the Economy subsystem where you can easily add or subtract resource value thanks to it being a ModifiableValue.

Conditionally removable modifiers

There are also some cases when you want to remove a modifier for example after a certain time. To do that you can use the `IConditionallyRemovable` interface from Core.Utility package.

```
public sealed class ExampleModifier : AddModifier<float32>,
IConditionallyRemovable
{
    public ExampleModifier(float32 value) : base(value)
    {
    }

    public bool IsRemovalConditionMet()
    {
        // This is just an example, you can use any condition here
        // it's just for god sake simplicity, you can also take custom
        // time from your game manager or any other source
        return DateTime.UtcNow.Year > 2025;
    }
}
```

This way you can easily remove the modifier after a certain time or when any other condition is met. Dynamically removable modifiers are always validated when the modifier is applied or removed or when the value is acquired.