

ProjectStack



[CardTest](#)

[Game](#)

[Game.LoadingProgressChangedEventArgs](#)

[Launcher](#)

[Main](#)

[ResourceEditor](#)

Base class for all UI-related nodes. Godot.Control features a bounding rectangle that defines its extents, an anchor position relative to its parent control or the current viewport, and offsets relative to the anchor. The offsets update automatically when the node, any of its parents, or the screen size change.

For more information on Godot's UI system, anchors, offsets, and containers, see the related tutorials in the manual. To build flexible UIs, you'll need a mix of UI elements that inherit from Godot.Control and Godot.Container nodes.

User Interface nodes and input

Godot propagates input events via viewports. Each Godot.Viewport is responsible for propagating Godot.InputEvents to their child nodes. As the Godot.SceneTree.Root is a Godot.Window, this already happens automatically for all UI elements in your game.

Input events are propagated through the Godot.SceneTree from the root node to all child nodes by calling Godot.Node._Input(Godot.InputEvent). For UI elements specifically, it makes more sense to override the virtual method Godot.Control._GUILInput(Godot.InputEvent), which filters out unrelated input events, such as by checking z-order, Godot.Control.MouseFilter, focus, or if the event was inside of the control's bounding box.

Call Godot.Control.AcceptEvent() so no other node receives the event. Once you accept an input, it becomes handled so Godot.Node._UnhandledInput(Godot.InputEvent) will not process it.

Only one Godot.Control node can be in focus. Only the node in focus will receive events. To get the focus, call Godot.Control.GrabFocus(). Godot.Control nodes lose focus when another node grabs it, or if you hide the node in focus.

Sets Godot.Control.MouseFilter to Godot.Control.MouseFilterEnum.Ignore to tell a Godot.Control node to ignore mouse or touch events. You'll need it if you place an icon on top of a button.

Godot.Theme resources change the Control's appearance. If you change the Godot.Theme on a Godot.Control node, it affects all of its children. To override some of the theme's parameters, call one of the `add_theme_*_override` methods, like Godot.Control.AddThemeFontOverride(Godot.StringName, Godot.Font). You can override the theme with the Inspector.

Note: Theme items are *not* Godot.GodotObject properties. This means you can't access their values using Godot.GodotObject.Get(Godot.StringName) and Godot.GodotObject.Set(Godot.StringName, Godot.Variant). Instead, use the `get_theme_*` and `add_theme_*_override` methods provided by this class.

[ResourceEditor.TaskNotifier](#)

A wrapping class that can hold a System.Threading.Tasks.Task value.

[ResourceEditor.TaskNotifier<T>](#)

A wrapping class that can hold a System.Threading.Tasks.Task<TResult> value.

CardTest □

□□□: [ProjectStack](#)

□□□: ProjectStack.dll

```
public class CardTest : TestClass
```

□□

```
object ← TestClass ← CardTest
```

□□□□

CardTest(Node)

```
public CardTest(Node testScene)
```

□□

```
testScene Node
```

□□

Setup()

```
[Setup(23)]  
public void Setup()
```

SetupAll()

```
[SetupAll(16)]  
public void SetupAll()
```

TestBottomCards()

```
[Test(43)]  
public void TestBottomCards()
```

TestDisconnectTopCard()

```
[Test(53)]  
public void TestDisconnectTopCard()
```

TestTopCards()

```
[Test(33)]  
public void TestTopCards()
```

Game

项目: [ProjectStack](#)

文件: ProjectStack.dll

```
[Meta(new Type[] { typeof(IAutoNode) })]  
[ScriptPath("res://src/Game.cs")]  
public class Game : Node2D
```

类

object ← GodotObject ← Node ← CanvasItem ← Node2D ← Game

方法

ActiveLevel

```
public Level? ActiveLevel
```

参数

[Level](#)

LoadedLevel

```
public List<string> LoadedLevel
```

参数

List<string>

SaveName

```
public string SaveName
```

□□□

string

□□

Default

```
public static Game Default { get; }
```

□□□

[Game](#)

Metatype

Generated metatype information.

```
public IMetatype Metatype { get; }
```

□□□

IMetatype

MixinState

Arbitrary data that is shared between mixins. Mixins are free to store additional instance state in this blackboard.

```
public MixinBlackboard MixinState { get; }
```

□□□

MixinBlackboard

Recipes

```
public IImmutableList<CardIdMatchRecipe> Recipes { get; }
```

□□□

IImmutableList<[CardIdMatchRecipe](#)>

SavePath

```
public string SavePath { get; }
```

□□□

string

ServiceProvider

```
public IServiceProvider ServiceProvider { get; }
```

□□□

IServiceProvider

□□

ChangeLevel(string)

```
public void ChangeLevel(stringlevelname)
```

□□

levelname string

GetGodotClassPropertyValue(in godot_string_name, out godot_variant)

Get the value of a property contained in this class. This method is used by Godot to retrieve property values. Do not call or override this method.

```
protected override bool GetGodotClassPropertyValue(in godot_string_name name, out  
godot_variant value)
```

□□

name godot_string_name

Name of the property to get.

value godot_variant

Value of the property if it was found.

□□

bool

[true](#) if a property with the given name was found.

GetKeylessRegistry<T>()

```
public KeylessRegistry<T> GetKeylessRegistry<T>()
```

□□

[KeylessRegistry](#)<T>

□□□□

T

GetLevel(string)

```
public Level? GetLevel(stringlevelname)
```

□□

levelname string

□□

[Level](#)

GetRegistry<T>()

```
public Registry<T> GetRegistry<T>()
```

□□

[Registry<T>](#)

□□□□

T

HasGodotClassMethod(in godot_string_name)

Check if the type contains a method with the given name. This method is used by Godot to check if a method exists before invoking it. Do not call or override this method.

```
protected override bool HasGodotClassMethod(in godot_string_name method)
```

□□

method godot_string_name

Name of the method to check for.

□□

bool

InvokeGodotClassMethod(**in** godot_string_name, Native VariantPtrArgs, **out** godot_variant)

Invokes the method with the given name, using the given arguments. This method is used by Godot to invoke methods from the engine side. Do not call or override this method.

```
protected override bool InvokeGodotClassMethod(in godot_string_name method,  
NativeVariantPtrArgs args, out godot_variant ret)
```

□□

method godot_string_name

Name of the method to invoke.

args NativeVariantPtrArgs

Arguments to use with the invoked method.

ret godot_variant

Value returned by the invoked method.

□□

bool

LoadLevel(string)

```
public Error LoadLevel(stringlevelname)
```

□□

levelname string

□□

OnReady()

Notification received when the node is ready.

```
public void OnReady()
```

RestoreGodotObjectData(GodotSerializationInfo)

Restores this instance's state after reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void RestoreGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

Saves this instance's state to be restored when reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void SaveGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object used to save the data.

SaveLevel(string)

```
public Error SaveLevel(string levelname)
```

□□

levelname string

□□

Error

SetGodotClassPropertyValue(in godot_string_name, in godot_variant)

Set the value of a property contained in this class. This method is used by Godot to assign property values. Do not call or override this method.

```
protected override bool SetGodotClassPropertyValue(in godot_string_name name, in godot_variant value)
```

□□

name godot_string_name

Name of the property to set.

value godot_variant

Value to set the property to if it was found.

□□

bool

[true](#) if a property with the given name was found.

UnloadLevel(string)

```
public Error UnloadLevel(stringlevelname)
```

□□

levelname string

□□

Error

Value()

Value that is provided by the provider.

```
public IServiceProvider Value()
```

□□

IServiceProvider

_Notification(int)

Called when the object receives a notification, which can be identified in what by comparing it with a constant. See also Godot.GodotObject.Notification(int, bool).

```
public override void _Notification(int what)
{
    if (what == NotificationPredelete)
    {
        GD.Print("Goodbye!");
    }
}
```

Note: The base Godot.GodotObject defines a few notifications (Godot.GodotObject.NotificationPostinitialize and Godot.GodotObject.NotificationPredelete). Inheriting classes such as Godot.Node define a lot more notifications, which are also received by this method.

```
public override void _Notification(int what)
```

□□

what int

□□

LoadingProgressChanged

□□□□□□□

```
public event EventHandler<Game.LoadingProgressChangedEventArgs>?  
LoadingProgressChanged
```

□□□□

EventHandler<[Game.LoadingProgressChangedEventArgs](#)>

Game>LoadingProgressChangedEventArgs

□

□□□: [ProjectStack](#)

□□□: ProjectStack.dll

```
public class Game>LoadingProgressChangedEventArgs : EventArgs
```

□□

```
object ← EventArgs ← Game>LoadingProgressChangedEventArgs
```

□□□□

LoadingProgressChangedEventArgs(float, string)

```
public LoadingProgressChangedEventArgs(float progress, string currentTaskName)
```

□□

```
progress float
```

```
currentTaskName string
```

□□

CurrentTaskName

```
public string CurrentTaskName { get; }
```

□□□

```
string
```

Progress

```
public float Progress { get; }
```



float

Launcher

项目: [ProjectStack](#)

文件: ProjectStack.dll

```
[ScriptPath("res://src/Launcher.cs")]
public class Launcher : Node
```

类

object ← GodotObject ← Node ← Launcher

方法

HasGodotClassMethod(in godot_string_name)

Check if the type contains a method with the given name. This method is used by Godot to check if a method exists before invoking it. Do not call or override this method.

```
protected override bool HasGodotClassMethod(in godot_string_name method)
```

参数

method godot_string_name

Name of the method to check for.

返回值

bool

InvokeGodotClassMethod(in godot_string_name, Native VariantPtrArgs, out godot_variant)

Invokes the method with the given name, using the given arguments. This method is used by Godot to invoke methods from the engine side. Do not call or override this method.

```
protected override bool InvokeGodotClassMethod(in godot_string_name method,  
NativeVariantPtrArgs args, out godot_variant ret)
```

□□

method godot_string_name

Name of the method to invoke.

args NativeVariantPtrArgs

Arguments to use with the invoked method.

ret godot_variant

Value returned by the invoked method.

□□

bool

RestoreGodotObjectData(GodotSerializationInfo)

Restores this instance's state after reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void RestoreGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

Saves this instance's state to be restored when reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void SaveGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object used to save the data.

_Ready()

Called when the node is "ready", i.e. when both the node and its children have entered the scene tree. If the node has children, their Godot.Node._Ready() callbacks get triggered first, and the parent node will receive the ready notification afterwards.

Corresponds to the Godot.Node.NotificationReady notification in Godot.GodotObject._Notification(int). See also the [@onready](#) annotation for variables.

Usually used for initialization. For even earlier initialization, Godot.GodotObject.GodotObject() may be used. See also Godot.Node._EnterTree().

Note: This method may be called only once for each node. After removing a node from the scene tree and adding it again, Godot.Node._Ready() will **not** be called a second time. This can be bypassed by requesting another call with Godot.Node.RequestReady(), which may be called anywhere before adding the node again.

```
public override void _Ready()
```

Main

项目: [ProjectStack](#)

文件: ProjectStack.dll

```
[ScriptPath("res://src/Main.cs")]
public class Main : Node2D
```

类

object ← GodotObject ← Node ← CanvasItem ← Node2D ← Main

方法

HasGodotClassMethod(in godot_string_name)

Check if the type contains a method with the given name. This method is used by Godot to check if a method exists before invoking it. Do not call or override this method.

```
protected override bool HasGodotClassMethod(in godot_string_name method)
```

参数

method godot_string_name

Name of the method to check for.

返回值

bool

InvokeGodotClassMethod(in godot_string_name, NativeVariantPtrArgs, out godot_variant)

Invokes the method with the given name, using the given arguments. This method is used by Godot to invoke methods from the engine side. Do not call or override this method.

```
protected override bool InvokeGodotClassMethod(in godot_string_name method,  
NativeVariantPtrArgs args, out godot_variant ret)
```

□□

method godot_string_name

Name of the method to invoke.

args NativeVariantPtrArgs

Arguments to use with the invoked method.

ret godot_variant

Value returned by the invoked method.

□□

bool

RestoreGodotObjectData(GodotSerializationInfo)

Restores this instance's state after reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void RestoreGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

Saves this instance's state to be restored when reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void SaveGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object used to save the data.

_Ready()

Called when the node is "ready", i.e. when both the node and its children have entered the scene tree. If the node has children, their Godot.Node._Ready() callbacks get triggered first, and the parent node will receive the ready notification afterwards.

Corresponds to the Godot.Node.NotificationReady notification in Godot.GodotObject._Notification(int). See also the [@onready](#) annotation for variables.

Usually used for initialization. For even earlier initialization, Godot.GodotObject.GodotObject() may be used. See also Godot.Node._EnterTree().

Note: This method may be called only once for each node. After removing a node from the scene tree and adding it again, Godot.Node._Ready() will **not** be called a second time. This can be bypassed by requesting another call with Godot.Node.RequestReady(), which may be called anywhere before adding the node again.

```
public override void _Ready()
```

ResourceEditor □

项目: [ProjectStack](#)

文件: ProjectStack.dll

Base class for all UI-related nodes. Godot.Control features a bounding rectangle that defines its extents, an anchor position relative to its parent control or the current viewport, and offsets relative to the anchor. The offsets update automatically when the node, any of its parents, or the screen size change.

For more information on Godot's UI system, anchors, offsets, and containers, see the related tutorials in the manual. To build flexible UIs, you'll need a mix of UI elements that inherit from Godot.Control and Godot.Container nodes.

User Interface nodes and input

Godot propagates input events via viewports. Each Godot.Viewport is responsible for propagating Godot.InputEvents to their child nodes. As the Godot.SceneTree.Root is a Godot.Window, this already happens automatically for all UI elements in your game.

Input events are propagated through the Godot.SceneTree from the root node to all child nodes by calling Godot.Node._Input(Godot.InputEvent). For UI elements specifically, it makes more sense to override the virtual method Godot.Control._GUILInput(Godot.InputEvent), which filters out unrelated input events, such as by checking z-order, Godot.Control.MouseFilter, focus, or if the event was inside of the control's bounding box.

Call Godot.Control.AcceptEvent() so no other node receives the event. Once you accept an input, it becomes handled so Godot.Node._UnhandledInput(Godot.InputEvent) will not process it.

Only one Godot.Control node can be in focus. Only the node in focus will receive events. To get the focus, call Godot.Control.GrabFocus(). Godot.Control nodes lose focus when another node grabs it, or if you hide the node in focus.

Sets Godot.Control.MouseFilter to Godot.Control.MouseFilterEnum.Ignore to tell a Godot.Control node to ignore mouse or touch events. You'll need it if you place an icon on top of a button.

Godot.Theme resources change the Control's appearance. If you change the Godot.Theme on a Godot.Control node, it affects all of its children. To override some of the theme's parameters, call one of the `add_theme_*_override` methods, like Godot.Control.AddTheme

`FontOverride(Godot.StringName, Godot.Font)`. You can override the theme with the Inspector.

Note: Theme items are *not* `Godot.GodotObject` properties. This means you can't access their values using `Godot.GodotObject.Get(Godot.StringName)` and `Godot.GodotObject.Set(Godot.StringName, Godot.Variant)`. Instead, use the `get_theme_*` and `add_theme_*_override` methods provided by this class.

```
[ObservableObject]
[Meta(new Type[] { typeof(IAutoNode) })]
[ScriptPath("res://src/ResourceEditor.cs")]
public class ResourceEditor : Control
```

object ← `GodotObject` ← `Node` ← `CanvasItem` ← `Control` ← `ResourceEditor`

Game

```
public Game Game { get; }
```

[Game](#)

Metatype

Generated metatype information.

```
public IMetatype Metatype { get; }
```

IMetatype

MixinState

Arbitrary data that is shared between mixins. Mixins are free to store additional instance state in this blackboard.

```
public MixinBlackboard MixinState { get; }
```



MixinBlackboard

ResourceTypeList

```
public ItemList ResourceTypeList { get; }
```



ItemList

ResourceTypes

```
public IReadOnlyList<string> ResourceTypes { get; }
```



IReadOnlyList<string>



GetGodotClassPropertyValue(in godot_string_name, out godot_variant)

Get the value of a property contained in this class. This method is used by Godot to retrieve property values. Do not call or override this method.

```
protected override bool GetGodotClassPropertyValue(in godot_string_name name, out godot_variant value)
```

□□

name godot_string_name

Name of the property to get.

value godot_variant

Value of the property if it was found.

□□

bool

[true](#) if a property with the given name was found.

HasGodotClassMethod(in godot_string_name)

Check if the type contains a method with the given name. This method is used by Godot to check if a method exists before invoking it. Do not call or override this method.

protected override bool HasGodotClassMethod(in godot_string_name method)

□□

method godot_string_name

Name of the method to check for.

□□

bool

InvokeGodotClassMethod(in godot_string_name, Native VariantPtrArgs, out godot_variant)

Invokes the method with the given name, using the given arguments. This method is used by Godot to invoke methods from the engine side. Do not call or override this method.

```
protected override bool InvokeGodotClassMethod(in godot_string_name method,  
NativeVariantPtrArgs args, out godot_variant ret)
```

□□

method godot_string_name

Name of the method to invoke.

args NativeVariantPtrArgs

Arguments to use with the invoked method.

ret godot_variant

Value returned by the invoked method.

□□

bool

OnPropertyChanged(PropertyChangedEventArgs)

Raises the [PropertyChanged](#) event.

```
protected virtual void OnPropertyChanged(PropertyChangedEventArgs e)
```

□□

e PropertyChangedEventArgs

The input System.ComponentModel.PropertyChangedEventArgs instance.

OnPropertyChanged(string?)

Raises the [PropertyChanged](#) event.

```
protected void OnPropertyChanged(string? propertyName = null)
```



propertyName string

(optional) The name of the property that changed.

OnPropertyChanging(PropertyChangingEventArgs)

Raises the [PropertyChanging](#) event.

```
protected virtual void OnPropertyChanging(PropertyChangingEventArgs e)
```



e PropertyChangingEventArgs

The input System.ComponentModel.PropertyChangingEventArgs instance.

OnPropertyChanging(string?)

Raises the [PropertyChanging](#) event.

```
protected void OnPropertyChanging(string? propertyName = null)
```



propertyName string

(optional) The name of the property that changed.

RestoreGodotObjectData(GodotSerializationInfo)

Restores this instance's state after reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void RestoreGodotObjectData(GodotSerializationInfo info)
```



info GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

Saves this instance's state to be restored when reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void SaveGodotObjectData(GodotSerializationInfo info)
```



info GodotSerializationInfo

Object used to save the data.

SetPropertyAndNotifyOnCompletion(ref TaskNotifier?, Task?, Action<Task?>, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. This method is just like [SetPropertyAndNotifyOnCompletion\(ref TaskNotifier?, Task?, string?\)](#), with the difference being an extra System.Action<T> parameter with a callback being invoked either immediately, if the new task has already completed or is [null](#), or upon completion.

```
protected bool SetPropertyAndNotifyOnCompletion(ref ResourceEditor.TaskNotifier? taskNotifier, Task? newValue, Action<Task?> callback, string? propertyName = null)
```



taskNotifier [ResourceEditor.TaskNotifier](#)

The field notifier to modify.

`newValue` Task

The property's value after the change occurred.

`callback` Action<Task>

A callback to invoke to update the property value.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

`true` if the property was changed, `false` otherwise.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetPropertyAndNotifyOnCompletion(ref TaskNotifier?, Task?, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method will also monitor the new value of the property (a generic System.Threading.Tasks.Task) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that System.Threading.Tasks.Task or any of its properties. This method and its overload specifically rely on the [ResourceEditor.TaskNotifier](#) type, which needs to be used in the backing field for the target System.Threading.Tasks.Task property. The field doesn't need to be initialized, as this method will take care of doing that automatically. The [ResourceEditor.TaskNotifier](#) type also includes an implicit operator, so it can be assigned to any System.Threading.Tasks.Task instance directly. Here is a sample property declaration using this method:

```
private TaskNotifier myTask;
```

```
public Task MyTask
{
    get => myTask;
    private set => SetAndNotifyOnCompletion(ref myTask, value);
}

protected bool SetPropertyAndNotifyOnCompletion(ref ResourceEditor.TaskNotifier?
taskNotifier, Task? newValue, string? propertyName = null)
```

□□

taskNotifier [ResourceEditor.TaskNotifier](#)

The field notifier to modify.

newValue Task

The property's value after the change occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same. The return value being [true](#) only indicates that the new value being assigned to **taskNotifier** is different than the previous one, and it does not mean the new System.Threading.Tasks.Task instance passed as argument is in any particular state.

SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>?, Task<T>?, Action<Task<T>?>, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. This method is just like [SetPropertyAndNotifyOnCompletion<T>\(ref TaskNotifier<T>?, Task<T>?, string?\)](#), with the difference being an extra System.Action<T> parameter with a callback being invoked either immediately, if the new task has already completed or is [null](#), or upon completion.

```
protected bool SetPropertyAndNotifyOnCompletion<T>(ref  
ResourceEditor.TaskNotifier<T>? taskNotifier, Task<T>? newValue, Action<Task<T>?>  
callback, string? propertyName = null)
```

□□

taskNotifier [ResourceEditor.TaskNotifier<T>](#)

The field notifier to modify.

newValue Task<T>

The property's value after the change occurred.

callback Action<Task<T>>

A callback to invoke to update the property value.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of result for the System.Threading.Tasks.Task<TResult> to set and monitor.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>?, Task<T>?, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method will also monitor the new value of the property (a generic System.Threading.Tasks.Task) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that System.Threading.Tasks.Task or any of its properties. This method and its overload specifically rely on the [ResourceEditor.TaskNotifier<T>](#) type, which needs to be used in the backing field for the target System.Threading.Tasks.Task property. The field doesn't need to be initialized, as this method will take care of doing that automatically. The [ResourceEditor.TaskNotifier<T>](#) type also includes an implicit operator, so it can be assigned to any System.Threading.Tasks.Task instance directly. Here is a sample property declaration using this method:

```
private TaskNotifier<int> myTask;

public Task<int> MyTask
{
    get => myTask;
    private set => SetAndNotifyOnCompletion(ref myTask, value);
}

protected bool SetPropertyAndNotifyOnCompletion<T>(ref
ResourceEditor.TaskNotifier<T>? taskNotifier, Task<T>? newValue, string?
propertyName = null)
```



taskNotifier [ResourceEditor.TaskNotifier<T>](#)

The field notifier to modify.

newValue Task<T>

The property's value after the change occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of result for the System.Threading.Tasks.Task<TResult> to set and monitor.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same. The return value being [true](#) only indicates that the new value being assigned to **taskNotifier** is different than the previous one, and it does not mean the new System.Threading.Tasks.Task<TResult> instance passed as argument is in any particular state.

SetProperty<T>(T, T, Action<T>, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. This overload is much less efficient than [SetProperty<T>\(ref T, T, string?\)](#) and it should only be used when the former is not viable (eg. when the target property being updated does not directly expose a backing field that can be passed by reference). For performance reasons, it is recommended to use a stateful callback if possible through the [SetProperty<TModel, T>\(T, T, TModel, Action<TModel, T>, string?\)](#) whenever possible instead of this overload, as that will allow the C# compiler to cache the input callback and reduce the memory allocations. More info on that overload are available in the related XML docs. This overload is here for completeness and in cases where that is not applicable.

```
protected bool SetProperty<T>(T oldValue, T newValue, Action<T> callback, string?)
```

`propertyName = null)`

□□

`oldValue T`

The current property value.

`newValue T`

The property's value after the change occurred.

`callback Action<T>`

A callback to invoke to update the property value.

`propertyName string`

(optional) The name of the property that changed.

□□

`bool`

`true` if the property was changed, `false` otherwise.

□□□□

`T`

The type of the property that changed.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetProperty<T>(T, T, IEqualityComparer<T>, Action<T>, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the

[PropertyChanged](#) event. See additional notes about this overload in [SetProperty<T>\(T, T, Action<T>, string?\)](#).

```
protected bool SetProperty<T>(T oldValue, T newValue, IEqualityComparer<T> comparer,  
Action<T> callback, string? propertyName = null)
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

comparer IEqualityComparer<T>

The System.Collections.Generic.IEqualityComparer<T> instance to use to compare the input values.

callback Action<T>

A callback to invoke to update the property value.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of the property that changed.

SetProperty<T>(ref T, T, IEqualityComparer<T>, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. See additional notes about this overload in [SetProperty<T>\(ref T, T, string?\)](#).

```
protected bool SetProperty<T>(ref T field, T newValue, IEqualityComparer<T>  
comparer, string? propertyName = null)
```

□□

field **T**

The field storing the property's value.

newValue **T**

The property's value after the change occurred.

comparer **IEqualityComparer<T>**

The System.Collections.Generic.IEqualityComparer<T> instance to use to compare the input values.

propertyName **string**

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of the property that changed.

SetProperty<T>(ref T, T, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event.

```
protected bool SetProperty<T>(ref T field, T newValue, string? propertyName = null)
```

□□

field T

The field storing the property's value.

newValue T

The property's value after the change occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of the property that changed.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetProperty<TModel, T>(T, T, IEqualityComparer<T>, TModel, Action<TModel, T>, string?)

Compares the current and new values for a given nested property. If the value has changed, raises the [PropertyChanging](#) event, updates the property and then raises the

[PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method is used to relay properties from a wrapped model in the current instance. See additional notes about this overload in [SetProperty<TModel, T>\(I, T, TModel, Action<TModel, T>, string?\)](#).

```
protected bool SetProperty<TModel, T>(T oldValue, T newValue, IEqualityComparer<T>
comparer, TModel model, Action<TModel, T> callback, string? propertyName = null)
where TModel : class
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

comparer IEqualityComparer<T>

The System.Collections.Generic.IEqualityComparer<T> instance to use to compare the input values.

model TModel

The model containing the property being updated.

callback Action<TModel, T>

The callback to invoke to set the target property value, if a change has occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

TModel

The type of model whose property (or field) to set.

T

The type of property (or field) to set.

SetProperty<TModel, T>(T, T, TModel, Action<TModel, T>, string?)

Compares the current and new values for a given nested property. If the value has changed, raises the [PropertyChanging](#) event, updates the property and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method is used to relay properties from a wrapped model in the current instance. This type is useful when creating wrapping, bindable objects that operate over models that lack support for notification (eg. for CRUD operations). Suppose we have this model (eg. for a database row in a table):

```
public class Person
{
    public string Name { get; set; }
}
```

We can then use a property to wrap instances of this type into our observable model (which supports notifications), injecting the notification to the properties of that model, like so:

```
[ObservableObject]
public class BindablePerson
{
    public Model { get; }

    public BindablePerson(Person model)
    {
        Model = model;
    }

    public string Name
    {
        get => Model.Name;
        set => Set(Model.Name, value, Model, (model, name) => model.Name = name);
    }
}
```

This way we can then use the wrapping object in our application, and all those "proxy" properties will also raise notifications when changed. Note that this method is not meant to be a replacement for [SetProperty<T>\(ref T, T, string?\)](#), and it should only be used when relaying properties to a model that doesn't support notifications, and only if you can't implement notifications to that model directly (eg. by having it inherit from ObservableObject). The syntax relies on passing the target model and a stateless callback to allow the C# compiler to cache the function, which results in much better performance and no memory usage.

```
protected bool SetProperty<TModel, T>(T oldValue, T newValue, TModel model,  
Action<TModel, T> callback, string? propertyName = null) where TModel : class
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

model TModel

The model containing the property being updated.

callback Action<TModel, T>

The callback to invoke to set the target property value, if a change has occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

TModel

The type of model whose property (or field) to set.

T

The type of property (or field) to set.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

_Notification(int)

Called when the object receives a notification, which can be identified in `what` by comparing it with a constant. See also `Godot.GodotObject.Notification(int, bool)`.

```
public override void _Notification(int what)
{
    if (what == NotificationPredelete)
    {
        GD.Print("Goodbye!");
    }
}
```

Note: The base `Godot.GodotObject` defines a few notifications (`Godot.GodotObject.NotificationPostinitialize` and `Godot.GodotObject.NotificationPredelete`). Inheriting classes such as `Godot.Node` define a lot more notifications, which are also received by this method.

```
public override void _Notification(int what)
```

□□

`what` int

_Process(double)

Called during the processing step of the main loop. Processing happens at every frame and as fast as possible, so the `delta` time since the previous frame is not constant. `delta` is in seconds.

It is only called if processing is enabled, which is done automatically if this method is overridden, and can be toggled with Godot.Node.SetProcess(bool).

Corresponds to the Godot.Node.NotificationProcess notification in Godot.GodotObject._Notification(int).

Note: This method is only called if the node is present in the scene tree (i.e. if it's not an orphan).

```
public override void _Process(double delta)
```



`delta` double

_Ready()

Called when the node is "ready", i.e. when both the node and its children have entered the scene tree. If the node has children, their Godot.Node._Ready() callbacks get triggered first, and the parent node will receive the ready notification afterwards.

Corresponds to the Godot.Node.NotificationReady notification in Godot.GodotObject._Notification(int). See also the `@onready` annotation for variables.

Usually used for initialization. For even earlier initialization, Godot.GodotObject.GodotObject() may be used. See also Godot.Node._EnterTree().

Note: This method may be called only once for each node. After removing a node from the scene tree and adding it again, Godot.Node._Ready() will **not** be called a second time. This can be bypassed by requesting another call with Godot.Node.RequestReady(), which may be called anywhere before adding the node again.

```
public override void _Ready()
```



PropertyChanged

Occurs when a property value changes.

```
public event PropertyChangedEventHandler? PropertyChanged
```

□□□□

PropertyChangedEventHandler

PropertyChanging

Occurs when a property value is changing.

```
public event PropertyChangingEventHandler? PropertyChanging
```

□□□□

PropertyChangingEventHandler

ResourceEditor.TaskNotifier

项目: [ProjectStack](#)

文件: ProjectStack.dll

A wrapping class that can hold a System.Threading.Tasks.Task value.

```
protected sealed class ResourceEditor.TaskNotifier
```

类

```
object ← ResourceEditor.TaskNotifier
```

方法

implicit operator Task?(TaskNotifier?)

Unwraps the System.Threading.Tasks.Task value stored in the current instance.

```
public static implicit operator Task?(ResourceEditor.TaskNotifier? notifier)
```

参数

notifier [ResourceEditor.TaskNotifier](#)

The input [ResourceEditor.TaskNotifier<T>](#) instance.

返回值

Task

ResourceEditor.TaskNotifier<T>

项目: [ProjectStack](#)

文件: ProjectStack.dll

A wrapping class that can hold a System.Threading.Tasks.Task<TResult> value.

```
protected sealed class ResourceEditor.TaskNotifier<T>
```

参数

T

The type of value for the wrapped System.Threading.Tasks.Task<TResult> instance.

示例

```
object ← ResourceEditor.TaskNotifier<T>
```

方法

implicit operator Task<T>?(TaskNotifier<T>?)

Unwraps the System.Threading.Tasks.Task<TResult> value stored in the current instance.

```
public static implicit operator Task<T>?(ResourceEditor.TaskNotifier<T>? notifier)
```

参数

notifier [ResourceEditor.TaskNotifier<T>](#)

The input [ResourceEditor.TaskNotifier<T>](#) instance.

返回值

Task<T>

ProjectStack.Attributes

□

[AttributeHolder](#)

[AttributeManager](#)

[AttributeModifier](#)

[BuiltInCardAttributes](#)

□□

[AttributeModifier.ModifierType](#)

AttributeHolder □

□□□: [ProjectStack.Attributes](#)

□□□: ProjectStack.dll

```
public class AttributeHolder
```

□□

object ← AttributeHolder

□□□□

AttributeHolder(ResourceLocation)

```
public AttributeHolder(ResourceLocation id)
```

□□

id [ResourceLocation](#)

AttributeHolder(string)

```
public AttributeHolder(string path)
```

□□

path string

□□

AttributId

```
public ResourceLocation AttributeId { get; }
```

□□□

[ResourceLocation](#)

□□

Add(AttributeModifier)

```
public void Add(AttributeModifier modifier)
```

□□

modifier [AttributeModifier](#)

Calculate(float)

```
public float Calculate(float baseValue)
```

□□

baseValue float

□□

float

CalculateBaseValue(float)

```
public float CalculateBaseValue(float value)
```

□□

value float

□□

float

CalculateTotalValue(float)

```
public float CalculateTotalValue(float value)
```

□□

value float

□□

float

Get()

```
public List<AttributeModifier> Get()
```

□□

List<[AttributeModifier](#)>

Remove(ResourceLocation)

```
public void Remove(ResourceLocation id)
```

□□

[id ResourceLocation](#)

AttributeManager

概述: [ProjectStack.Attributes](#)

位置: ProjectStack.dll

```
public class AttributeManager
```

方法

object ← AttributeManager

方法

Add(AttributeHolder)

```
public void Add(AttributeHolder attributeHolder)
```

参数

attributeHolder [AttributeHolder](#)

Get(AttributeHolder)

```
public AttributeHolder Get(AttributeHolder type)
```

参数

type [AttributeHolder](#)

返回值

[AttributeHolder](#)

Get(ResourceLocation)

```
public AttributeHolder Get(ResourceLocation id)
```

□□

id [ResourceLocation](#)

□□

[AttributeHolder](#)

Remove(ResourceLocation)

```
public void Remove(ResourceLocation id)
```

□□

id [ResourceLocation](#)

AttributeModifier □

□□□: [ProjectStack.Attributes](#)

□□□: ProjectStack.dll

```
public class AttributeModifier
```

□□

object ← AttributeModifier

□□□□

AttributeModifier(ResourceLocation, float, ModifierType)

```
public AttributeModifier(ResourceLocation id, float value,  
AttributeModifier.ModifierType type)
```

□□

id [ResourceLocation](#)

value float

type [AttributeModifier.ModifierType](#)

□□

Id

```
public ResourceLocation Id { get; }
```

□□□

[ResourceLocation](#)

Type

```
public AttributeModifier.ModifierType Type { get; }
```

□□□

[AttributeModifier.ModifierType](#)

Value

```
public float Value { get; }
```

□□□

float

AttributeModifier.ModifierType

概述: [ProjectStack.Attributes](#)

位置: ProjectStack.dll

```
public enum AttributeModifier.ModifierType
```

值

ADD_BASE = 0

ADD_MULTIPLY_BASE = 1

ADD_MULTIPLY_TOTAL = 3

ADD_TOTAL = 2

BuiltnCardAttributes □

□□□: [ProjectStack.Attributes](#)

□□□: ProjectStack.dll

```
public class BuiltnCardAttributes
```

□□

object ← BuiltnCardAttributes

□□

Attack

```
public static readonly AttributeHolder Attack
```

□□□

[AttributeHolder](#)

AttackSpeed

```
public static readonly AttributeHolder AttackSpeed
```

□□□

[AttributeHolder](#)

Defense

```
public static readonly AttributeHolder Defense
```

□□□

[AttributeHolder](#)

Health

```
public static readonly AttributeHolder Health
```



[AttributeHolder](#)

Range

```
public static readonly AttributeHolder Range
```



[AttributeHolder](#)

Speed

```
public static readonly AttributeHolder Speed
```



[AttributeHolder](#)

ProjectStack.Command

□

[CommandAdapter](#)

CommandAdapter □

□□□: [ProjectStack.Command](#)

□□□: ProjectStack.dll

```
[Meta(new Type[] { typeof(IAutoNode) })]
[ScriptPath("res://src/scripts/Command/CommandAdapter.cs")]
public class CommandAdapter : Node
```

□□

object ← GodotObject ← Node ← CommandAdapter

□□□

Metatype

Generated metatype information.

```
public IMetatype Metatype { get; }
```

□□□

IMetatype

MixinState

Arbitrary data that is shared between mixins. Mixins are free to store additional instance state in this blackboard.

```
public MixinBlackboard MixinState { get; }
```

□□□

MixinBlackboard

□□

ChangeLevel(string)

```
public string ChangeLevel(string level)
```

□□

level string

□□

string

CloseEditor()

```
public void CloseEditor()
```

CreateNewCard(ResourceLocation)

```
public void CreateNewCard(ResourceLocation id)
```

□□

id ResourceLocation

CreateNewCard(string)

```
public void CreateNewCard(string id)
```

□□

id string

CreateNewCardStack(List<ResourceLocation>)

```
public void CreateNewCardStack(List<ResourceLocation> ids)
```

□□

ids List<[ResourceLocation](#)>

CreateNewCardStack(string[])

```
public void CreateNewCardStack(string[] ids)
```

□□

ids string[]

GetCardMate(ResourceLocation)

```
public ResourceLocation GetCardMate(ResourceLocation id)
```

□□

id [ResourceLocation](#)

□□

[ResourceLocation](#)

GetCardMate(string)

```
public ResourceLocation GetCardMate(string id)
```

□□

`id` string

□□

[ResourceLocation](#)

GetRegisteredCardMetaIds()

```
public string[] GetRegisteredCardMetaIds()
```

□□

string[]

HasGodotClassMethod(`in godot_string_name`)

Check if the type contains a method with the given name. This method is used by Godot to check if a method exists before invoking it. Do not call or override this method.

```
protected override bool HasGodotClassMethod(in godot_string_name method)
```

□□

`method` godot_string_name

Name of the method to check for.

□□

bool

InvokeGodotClassMethod(`in godot_string_name`, Native VariantPtrArgs, `out godot_variant`)

Invokes the method with the given name, using the given arguments. This method is used by Godot to invoke methods from the engine side. Do not call or override this method.

```
protected override bool InvokeGodotClassMethod(in godot_string_name method,  
NativeVariantPtrArgs args, out godot_variant ret)
```

□□

method godot_string_name

Name of the method to invoke.

args NativeVariantPtrArgs

Arguments to use with the invoked method.

ret godot_variant

Value returned by the invoked method.

□□

bool

LoadLevel(string)

```
public string LoadLevel(string level)
```

□□

level string

□□

string

OpenEditor()

```
public void OpenEditor()
```

RestoreGodotObjectData(GodotSerializationInfo)

Restores this instance's state after reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void RestoreGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

Saves this instance's state to be restored when reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void SaveGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object used to save the data.

SaveLevel(string)

```
public string SaveLevel(string level)
```

□□

level string

□□

string

UnloadLevel(string)

```
public string UnloadLevel(string level)
```

□□

level string

□□

string

ProjectStack.Common



[ResourceLocation](#)

ResourceLocation □

□□□: [ProjectStack.Common](#)

□□□: ProjectStack.dll

```
public class ResourceLocation : ObservableValidator
```

□□

```
object ← ObservableObject ← ObservableValidator ← ResourceLocation
```

□□

Empty

```
public static readonly ResourceLocation Empty
```

□□□

[ResourceLocation](#)

□□

Namespace

```
[Required]  
[RegularExpression("^[a-zA-Z_][a-zA-Z0-9_]*$")]  
public string Namespace { get; init; }
```

□□□

string

Path

```
[Required]  
[RegularExpression("^[a-zA-Z0-9_/.]+$")]  
public string Path { get; init; }
```

□□□

string

□□

DefaultNamespaceAndPath(string)

```
public static ResourceLocation DefaultNamespaceAndPath(string path)
```

□□

path string

□□

[ResourceLocation](#)

Equals(ResourceLocation?)

Indicates whether the current object is equal to another object of the same type.

```
public bool Equals(ResourceLocation? other)
```

□□

other [ResourceLocation](#)

An object to compare with this object.

□□

bool

`true` if the current object is equal to the `other` parameter; otherwise, `false`.

Equals(object?)

Determines whether the specified object is equal to the current object.

```
public override bool Equals(object? obj)
```

□□

`obj` object

The object to compare with the current object.

□□

bool

`true` if the specified object is equal to the current object; otherwise, `false`.

GetHashCode()

Serves as the default hash function.

```
public override int GetHashCode()
```

□□

int

A hash code for the current object.

GetResLoc()

```
public string GetResLoc()
```

□□

string

GetResLocWithPath(string)

```
public string GetResLocWithPath(string p)
```

□□

p string

□□

string

GetResLocWithoutPrefix(string)

```
public string GetResLocWithoutPrefix(string p)
```

□□

p string

□□

string

Parse(string)

```
public static ResourceLocation Parse(string fullPath)
```

□□

fullPath string

□□

[ResourceLocation](#)

ToString()

Returns a string that represents the current object.

```
public override string ToString()
```

□□

string

A string that represents the current object.

□□□

operator ==(ResourceLocation?, ResourceLocation?)

```
public static bool operator ==(ResourceLocation? left, ResourceLocation? right)
```

□□

left [ResourceLocation](#)

right [ResourceLocation](#)

□□

bool

operator !=(ResourceLocation?, ResourceLocation?)

```
public static bool operator !=(ResourceLocation? left, ResourceLocation? right)
```

□□

`left` [ResourceLocation](#)

`right` [ResourceLocation](#)

□□

bool

ProjectStack.Common.Card



CardMeta

An advanced Godot.Variant type. All classes in the engine inherit from Object. Each class may define new properties, methods or signals, which are available to all inheriting classes. For example, a Godot.Sprite2D instance is able to call Godot.Node.Add Child(Godot.Node, bool, Godot.Node.InternalMode) because it inherits from Godot.Node.

You can create new instances, using `Object.new()` in GDScript, or `new GodotObject` in C#.

To delete an Object instance, call `Godot.GodotObject.Free()`. This is necessary for most classes inheriting Object, because they do not manage memory on their own, and will otherwise cause memory leaks when no longer in use. There are a few classes that perform memory management. For example, `Godot.RefCounted` (and by extension `Godot.Resource`) deletes itself when no longer referenced, and `Godot.Node` deletes its children when freed.

Objects can have a `Godot.Script` attached to them. Once the `Godot.Script` is instantiated, it effectively acts as an extension to the base class, allowing it to define and inherit new properties, methods and signals.

Inside a `Godot.Script`, `Godot.GodotObject._GetPropertyList()` may be overridden to customize properties in several ways. This allows them to be available to the editor, display as lists of options, sub-divide into groups, save on disk, etc. Scripting languages offer easier ways to customize properties, such as with the `[annotation @GDScript.@export]` annotation.

Godot is very dynamic. An object's script, and therefore its properties, methods and signals, can be changed at run-time. Because of this, there can be occasions where, for example, a property required by a method may not exist. To prevent run-time errors, see methods such as `Godot.GodotObject.Set(Godot.StringName, Godot.Variant)`, `Godot.GodotObject.Get(Godot.StringName)`, `Godot.GodotObject.Call(Godot.StringName, params Godot.Variant[])`, `Godot.GodotObject.HasMethod(Godot.StringName)`, `Godot.GodotObject.HasSignal(Godot.StringName)`, etc. Note that these methods are **much** slower than direct references.

In GDScript, you can also check if a given property, method, or signal name exists in an object with the `in` operator:

```
var node = Node.new()
print("name" in node)          # Prints true
```

```
print("get_parent" in node)    # Prints true
print("tree_entered" in node) # Prints true
print("unknown" in node)      # Prints false
```

Notifications are int constants commonly sent and received by objects. For example, on every rendered frame, the Godot.SceneTree notifies nodes inside the tree with a Godot.Node.NotificationProcess. The nodes receive it and may call Godot.Node._Process(double) to update. To make use of notifications, see Godot.GodotObject.Notification(int, bool) and Godot.GodotObject._Notification(int).

Lastly, every object can also contain metadata (data about data). Godot.GodotObject.SetMeta(Godot.StringName, Godot.Variant) can be useful to store information that the object itself does not depend on. To keep your code clean, making excessive use of metadata is discouraged.

Note: Unlike references to a Godot.RefCounted, references to an object stored in a variable can become invalid without being set to [null](#). To check if an object has been deleted, do *not* compare it against [null](#). Instead, use [@GlobalScope.is_instance_valid](#). It's also recommended to inherit from Godot.RefCounted for classes storing data instead of Godot.GodotObject.

Note: The [script](#) is not exposed like most properties. To set or get an object's Godot.Script in code, use Godot.GodotObject.SetScript(Godot.Variant) and Godot.GodotObject.GetScript(), respectively.

[CardMeta.TaskNotifier](#)

A wrapping class that can hold a System.Threading.Tasks.Task value.

[CardMeta.TaskNotifier<T>](#)

A wrapping class that can hold a System.Threading.Tasks.Task<TResult> value.

[CardMetaRegistrationHelper](#)

[CardStack](#)



[ICardStack](#)

CardMeta □

概述: [ProjectStack.Common.Card](#)

模块: ProjectStack.dll

An advanced Godot.Variant type. All classes in the engine inherit from Object. Each class may define new properties, methods or signals, which are available to all inheriting classes. For example, a Godot.Sprite2D instance is able to call Godot.Node.AddChild(Godot.Node, bool, Godot.Node.InternalMode) because it inherits from Godot.Node.

You can create new instances, using `Object.new()` in GDScript, or `new GodotObject` in C#.

To delete an Object instance, call `Godot.GodotObject.Free()`. This is necessary for most classes inheriting Object, because they do not manage memory on their own, and will otherwise cause memory leaks when no longer in use. There are a few classes that perform memory management. For example, `Godot.RefCounted` (and by extension `Godot.Resource`) deletes itself when no longer referenced, and `Godot.Node` deletes its children when freed.

Objects can have a `Godot.Script` attached to them. Once the `Godot.Script` is instantiated, it effectively acts as an extension to the base class, allowing it to define and inherit new properties, methods and signals.

Inside a `Godot.Script`, `Godot.GodotObject._GetPropertyList()` may be overridden to customize properties in several ways. This allows them to be available to the editor, display as lists of options, sub-divide into groups, save on disk, etc. Scripting languages offer easier ways to customize properties, such as with the `[annotation @GDScript.@export]` annotation.

Godot is very dynamic. An object's script, and therefore its properties, methods and signals, can be changed at run-time. Because of this, there can be occasions where, for example, a property required by a method may not exist. To prevent run-time errors, see methods such as `Godot.GodotObject.Set(Godot.StringName, Godot.Variant)`, `Godot.GodotObject.Get(Godot.StringName)`, `Godot.GodotObject.Call(Godot.StringName, params Godot.Variant[])`, `Godot.GodotObject.HasMethod(Godot.StringName)`, `Godot.GodotObject.HasSignal(Godot.StringName)`, etc. Note that these methods are **much** slower than direct references.

In GDScript, you can also check if a given property, method, or signal name exists in an object with the `in` operator:

```

var node = Node.new()
print("name" in node)          # Prints true
print("get_parent" in node)    # Prints true
print("tree_entered" in node) # Prints true
print("unknown" in node)       # Prints false

```

Notifications are int constants commonly sent and received by objects. For example, on every rendered frame, the Godot.SceneTree notifies nodes inside the tree with a Godot.Node.NotificationProcess. The nodes receive it and may call Godot.Node._Process(double) to update. To make use of notifications, see Godot.GodotObject.Notification(int, bool) and Godot.GodotObject._Notification(int).

Lastly, every object can also contain metadata (data about data). Godot.GodotObject.SetMeta(Godot.StringName, Godot.Variant) can be useful to store information that the object itself does not depend on. To keep your code clean, making excessive use of metadata is discouraged.

Note: Unlike references to a Godot.RefCounted, references to an object stored in a variable can become invalid without being set to [null](#). To check if an object has been deleted, do *not* compare it against [null](#). Instead, use [@GlobalScope.is_instance_valid](#). It's also recommended to inherit from Godot.RefCounted for classes storing data instead of Godot.GodotObject.

Note: The [script](#) is not exposed like most properties. To set or get an object's Godot.Script in code, use Godot.GodotObject.SetScript(Godot.Variant) and Godot.GodotObject.GetScript(), respectively.

```

[ObservableObject]
[ScriptPath("res://src/scripts/Common/Card/CardMeta.cs")]
public class CardMeta : GodotObject

```

□□

object ← GodotObject ← CardMeta

□□

Description

```
public string Description { get; set; }
```

□□□

string

Id

```
public ResourceLocation Id { get; }
```

□□□

[ResourceLocation](#)

Name

```
public string Name { get; set; }
```

□□□

string

Type

```
public ResourceLocation Type { get; set; }
```

□□□

[ResourceLocation](#)

□□

Create(string)

```
public static CardMeta Create(string id)
```

□□

id string

□□

[CardMeta](#)

Deserialize(FriendlyNtjObject)

```
public void Deserialize(FriendlyNtjObject j)
```

□□

j [FriendlyNtjObject](#)

GetGodotClassPropertyValue(in godot_string_name, out godot_variant)

Get the value of a property contained in this class. This method is used by Godot to retrieve property values. Do not call or override this method.

```
protected override bool GetGodotClassPropertyValue(in godot_string_name name, out godot_variant value)
```

□□

name godot_string_name

Name of the property to get.

value godot_variant

Value of the property if it was found.

□□

bool

[true](#) if a property with the given name was found.

HasGodotClassMethod(*in godot_string_name*)

Check if the type contains a method with the given name. This method is used by Godot to check if a method exists before invoking it. Do not call or override this method.

```
protected override bool HasGodotClassMethod(in godot_string_name method)
```

□□

method *godot_string_name*

Name of the method to check for.

□□

bool

InvokeGodotClassMethod(*in godot_string_name*, *NativeVariantPtrArgs*, *out godot_variant*)

Invokes the method with the given name, using the given arguments. This method is used by Godot to invoke methods from the engine side. Do not call or override this method.

```
protected override bool InvokeGodotClassMethod(in godot_string_name method,  
NativeVariantPtrArgs args, out godot_variant ret)
```

□□

method *godot_string_name*

Name of the method to invoke.

args *NativeVariantPtrArgs*

Arguments to use with the invoked method.

ret *godot_variant*

Value returned by the invoked method.

□□

bool

OnPropertyChanged(PropertyChangedEventArgs)

Raises the [PropertyChanged](#) event.

```
protected virtual void OnPropertyChanged(PropertyChangedEventArgs e)
```

□□

e PropertyChangedEventArgs

The input System.ComponentModel.PropertyChangedEventArgs instance.

OnPropertyChanged(string?)

Raises the [PropertyChanged](#) event.

```
protected void OnPropertyChanged(string? propertyName = null)
```

□□

propertyName string

(optional) The name of the property that changed.

OnPropertyChanging(PropertyChangingEventArgs)

Raises the [PropertyChanging](#) event.

```
protected virtual void OnPropertyChanging(PropertyChangingEventArgs e)
```

□□

e PropertyChangingEventArgs

The input System.ComponentModel.PropertyChangingEventArgs instance.

OnPropertyChanging(string?)

Raises the [PropertyChanging](#) event.

```
protected void OnPropertyChanging(string? propertyName = null)
```

□□

propertyName string

(optional) The name of the property that changed.

Parse(FriendlyNtjObject)

```
public static CardMeta Parse(FriendlyNtjObject j)
```

□□

j [FriendlyNtjObject](#)

□□

[CardMeta](#)

RestoreGodotObjectData(GodotSerializationInfo)

Restores this instance's state after reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void RestoreGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

Saves this instance's state to be restored when reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void SaveGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object used to save the data.

Serialize(FriendlyNtjObject)

```
public void Serialize(FriendlyNtjObject j)
```

□□

j [FriendlyNtjObject](#)

SetGodotClassPropertyValue(in godot_string_name, in godot_variant)

Set the value of a property contained in this class. This method is used by Godot to assign property values. Do not call or override this method.

```
protected override bool SetGodotClassPropertyValue(in godot_string_name name, in godot_variant value)
```

□□

name godot_string_name

Name of the property to set.

value godot_variant

Value to set the property to if it was found.

□□

bool

[true](#) if a property with the given name was found.

SetPropertyAndNotifyOnCompletion(ref TaskNotifier?, Task?, Action<Task?>, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. This method is just like [SetPropertyAndNotifyOnCompletion\(ref TaskNotifier?, Task?, string?\)](#), with the difference being an extra System.Action<T> parameter with a callback being invoked either immediately, if the new task has already completed or is [null](#), or upon completion.

```
protected bool SetPropertyAndNotifyOnCompletion(ref CardMeta.TaskNotifier?  
taskNotifier, Task? newValue, Action<Task?> callback, string? propertyName = null)
```

□□

taskNotifier [CardMeta.TaskNotifier](#)

The field notifier to modify.

newValue Task

The property's value after the change occurred.

callback Action<Task>

A callback to invoke to update the property value.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetPropertyAndNotifyOnCompletion(ref TaskNotifier?, Task?, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method will also monitor the new value of the property (a generic System.Threading.Tasks.Task) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that System.Threading.Tasks.Task or any of its properties. This method and its overload specifically rely on the [CardMeta.TaskNotifier](#) type, which needs to be used in the backing field for the target System.Threading.Tasks.Task property. The field doesn't need to be initialized, as this method will take care of doing that automatically. The [CardMeta.TaskNotifier](#) type also includes an implicit operator, so it can be assigned to any System.Threading.Tasks.Task instance directly. Here is a sample property declaration using this method:

```
private TaskNotifier myTask;

public Task MyTask
{
    get => myTask;
    private set => SetPropertyAndNotifyOnCompletion(ref myTask, value);
}

protected bool SetPropertyAndNotifyOnCompletion(ref CardMeta.TaskNotifier?
taskNotifier, Task? newValue, string? propertyName = null)
```



`taskNotifier` [CardMeta.TaskNotifier](#)

The field notifier to modify.

`newValue` Task

The property's value after the change occurred.

`propertyName` string

(optional) The name of the property that changed.



bool

[true](#) if the property was changed, [false](#) otherwise.



The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same. The return value being [true](#) only indicates that the new value being assigned to `taskNotifier` is different than the previous one, and it does not mean the new System.Threading.Tasks.Task instance passed as argument is in any particular state.

SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>?, Task<T>?, Action<Task<T>?>, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. This method is just like [SetPropertyAndNotifyOnCompletion<T>\(ref TaskNotifier<T>?, Task<T>?, string?\)](#), with the difference being an extra System.Action<T> parameter with a callback being invoked either immediately, if the new task has already completed or is [null](#), or upon completion.

```
protected bool SetPropertyAndNotifyOnCompletion<T>(ref CardMeta.TaskNotifier<T>?  
taskNotifier, Task<T>? newValue, Action<Task<T>?> callback, string? propertyName
```

= `null`)

□□

`taskNotifier` [CardMeta.TaskNotifier<T>](#)

The field notifier to modify.

`newValue` [Task<T>](#)

The property's value after the change occurred.

`callback` [Action<Task<T>>](#)

A callback to invoke to update the property value.

`propertyName` string

(optional) The name of the property that changed.

□□

`bool`

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of result for the `System.Threading.Tasks.Task<TResult>` to set and monitor.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

`SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>?, Task<T>?, string?)`

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the

field and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method will also monitor the new value of the property (a generic System.Threading.Tasks.Task) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that System.Threading.Tasks.Task or any of its properties. This method and its overload specifically rely on the [CardMeta.TaskNotifier<T>](#) type, which needs to be used in the backing field for the target System.Threading.Tasks.Task property. The field doesn't need to be initialized, as this method will take care of doing that automatically. The [CardMeta.TaskNotifier<T>](#) type also includes an implicit operator, so it can be assigned to any System.Threading.Tasks.Task instance directly. Here is a sample property declaration using this method:

```
private TaskNotifier<int> myTask;

public Task<int> MyTask
{
    get => myTask;
    private set => SetAndNotifyOnCompletion(ref myTask, value);
}

protected bool SetPropertyAndNotifyOnCompletion<T>(ref CardMeta.TaskNotifier<T>?
taskNotifier, Task<T>? newValue, string? propertyName = null)
```

□□

taskNotifier [CardMeta.TaskNotifier<T>](#)

The field notifier to modify.

newValue Task<T>

The property's value after the change occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□

T

The type of result for the System.Threading.Tasks.Task<TResult> to set and monitor.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same. The return value being [true](#) only indicates that the new value being assigned to [taskNotifier](#) is different than the previous one, and it does not mean the new System.Threading.Tasks.Task<TResult> instance passed as argument is in any particular state.

SetProperty<T>(T, T, Action<T>, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. This overload is much less efficient than [SetProperty<T>\(ref T, T, string?\)](#) and it should only be used when the former is not viable (eg. when the target property being updated does not directly expose a backing field that can be passed by reference). For performance reasons, it is recommended to use a stateful callback if possible through the [SetProperty<TModel, T>\(T, T, TModel, Action<TModel, T>, string?\)](#) whenever possible instead of this overload, as that will allow the C# compiler to cache the input callback and reduce the memory allocations. More info on that overload are available in the related XML docs. This overload is here for completeness and in cases where that is not applicable.

```
protected bool SetProperty<T>(T oldValue, T newValue, Action<T> callback, string?  
propertyName = null)
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

`callback Action<T>`

A callback to invoke to update the property value.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

`true` if the property was changed, `false` otherwise.

□□□□

T

The type of the property that changed.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetProperty<T>(T, T, IEqualityComparer<T>, Action<T>, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. See additional notes about this overload in [SetProperty<T>\(T, T, Action<T>, string?\)](#).

```
protected bool SetProperty<T>(T oldValue, T newValue, IEqualityComparer<T> comparer, Action<T> callback, string? propertyName = null)
```

□□

`oldValue` T

The current property value.

`newValue` `T`

The property's value after the change occurred.

`comparer` `IEqualityComparer<T>`

The `System.Collections.Generic.IEqualityComparer<T>` instance to use to compare the input values.

`callback` `Action<T>`

A callback to invoke to update the property value.

`propertyName` `string`

(optional) The name of the property that changed.

□□

`bool`

`true` if the property was changed, `false` otherwise.

□□□□

`T`

The type of the property that changed.

`SetProperty<T>(ref T, T, IEqualityComparer<T>, string?)`

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. See additional notes about this overload in [`SetProperty<T>\(ref T, T, string?\)`](#).

```
protected bool SetProperty<T>(ref T field, T newValue, IEqualityComparer<T>
comparer, string? propertyName = null)
```

□□

field T

The field storing the property's value.

newValue T

The property's value after the change occurred.

comparer IEqualityComparer<T>

The System.Collections.Generic.IEqualityComparer<T> instance to use to compare the input values.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of the property that changed.

SetProperty<T>(ref T, T, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event.

```
protected bool SetProperty<T>(ref T field, T newValue, string? propertyName = null)
```

□□

field T

The field storing the property's value.

`newValue` `T`

The property's value after the change occurred.

`propertyName` `string`

(optional) The name of the property that changed.

`oldValue`

`bool`

`true` if the property was changed, `false` otherwise.

`propertyType`

`T`

The type of the property that changed.

`oldValue`

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

`SetProperty<TModel, T>(T, T, IEqualityComparer<T>, TModel, Action<TModel, T>, string?)`

Compares the current and new values for a given nested property. If the value has changed, raises the [PropertyChanging](#) event, updates the property and then raises the [PropertyChanged](#) event. The behavior mirrors that of [`SetProperty<T>\(ref T, T, string?\)`](#), with the difference being that this method is used to relay properties from a wrapped model in the current instance. See additional notes about this overload in [`SetProperty<TModel, T>\(I, T, TModel, Action<TModel, T>, string?\)`](#).

```
protected bool SetProperty<TModel, T>(T oldValue, T newValue, IEqualityComparer<T> comparer, TModel model, Action<TModel, T> callback, string? propertyName = null)  
where TModel : class
```

`oldValue`

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

comparer IEqualityComparer<T>

The System.Collections.Generic.IEqualityComparer<T> instance to use to compare the input values.

model TModel

The model containing the property being updated.

callback Action<TModel, T>

The callback to invoke to set the target property value, if a change has occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

TModel

The type of model whose property (or field) to set.

T

The type of property (or field) to set.

SetProperty<TModel, T>(T, T, TModel, Action<TModel, T>, string?)

Compares the current and new values for a given nested property. If the value has changed, raises the [PropertyChanging](#) event, updates the property and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method is used to relay properties from a wrapped model in the current instance. This type is useful when creating wrapping, bindable objects that operate over models that lack support for notification (eg. for CRUD operations). Suppose we have this model (eg. for a database row in a table):

```
public class Person
{
    public string Name { get; set; }
}
```

We can then use a property to wrap instances of this type into our observable model (which supports notifications), injecting the notification to the properties of that model, like so:

```
[ObservableObject]
public class BindablePerson
{
    public Model { get; }

    public BindablePerson(Person model)
    {
        Model = model;
    }

    public string Name
    {
        get => Model.Name;
        set => Set(Model.Name, value, Model, (model, name) => model.Name = name);
    }
}
```

This way we can then use the wrapping object in our application, and all those "proxy" properties will also raise notifications when changed. Note that this method is not meant to be a replacement for [SetProperty<T>\(ref T, T, string?\)](#), and it should only be used when relaying properties to a model that doesn't support notifications, and only if you can't implement notifications to that model directly (eg. by having it inherit from `ObservableObject`). The syntax relies on passing the target model and a stateless callback to allow the C# compiler to cache the function, which results in much better performance and no memory usage.

```
protected bool SetProperty<TModel, T>(T oldValue, T newValue, TModel model,  
Action<TModel, T> callback, string? propertyName = null) where TModel : class
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

model TModel

The model containing the property being updated.

callback Action<TModel, T>

The callback to invoke to set the target property value, if a change has occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

TModel

The type of model whose property (or field) to set.

T

The type of property (or field) to set.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.



PropertyChanged

Occurs when a property value changes.

```
public event PropertyChangedEventHandler? PropertyChanged
```



PropertyChangedEventHandler

PropertyChanging

Occurs when a property value is changing.

```
public event PropertyChangingEventHandler? PropertyChanging
```



PropertyChangingEventHandler

CardMeta.TaskNotifier □

概述: [ProjectStack.Common.Card](#)

位置: ProjectStack.dll

A wrapping class that can hold a System.Threading.Tasks.Task value.

```
protected sealed class CardMeta.TaskNotifier
```

□□

```
object ← CardMeta.TaskNotifier
```

□□□

implicit operator Task?(TaskNotifier?)

Unwraps the System.Threading.Tasks.Task value stored in the current instance.

```
public static implicit operator Task?(CardMeta.TaskNotifier? notifier)
```

□□

```
notifier CardMeta.TaskNotifier
```

The input [CardMeta.TaskNotifier<T>](#) instance.

□□

Task

CardMeta.TaskNotifier<T>

所在文件: [ProjectStack.Common.Card](#)

所在程序集: ProjectStack.dll

A wrapping class that can hold a System.Threading.Tasks.Task<TResult> value.

```
protected sealed class CardMeta.TaskNotifier<T>
```

参数

T

The type of value for the wrapped System.Threading.Tasks.Task<TResult> instance.

示例

```
object <-- CardMeta.TaskNotifier<T>
```

方法

implicit operator Task<T>?(TaskNotifier<T>?)

Unwraps the System.Threading.Tasks.Task<TResult> value stored in the current instance.

```
public static implicit operator Task<T>?(CardMeta.TaskNotifier<T>? notifier)
```

参数

notifier [CardMeta.TaskNotifier<T>](#)

The input [CardMeta.TaskNotifier<T>](#) instance.

返回值

Task<T>

CardMetaRegistrationHelper □

□□□: [ProjectStack.Common.Card](#)

□□□: ProjectStack.dll

```
public class CardMetaRegistrationHelper
```

□□

```
object ← CardMetaRegistrationHelper
```

□□□□

CardMetaRegistrationHelper(IServiceCollection)

```
public CardMetaRegistrationHelper(IServiceCollection services)
```

□□

```
services IServiceCollection
```

□□

Add(CardMeta)

```
public CardMetaRegistrationHelper Add(CardMeta cardMeta)
```

□□

```
cardMeta CardMeta
```

□□

[CardMetaRegistrationHelper](#)

End()

```
public IServiceCollection End()
```



IServiceCollection

CardStack

类名: [ProjectStack.Common.Card](#)

位置: ProjectStack.dll

```
public record CardStack : ICardStack
```

类

object ← CardStack

接口

[ICardStack](#)

参数

CardStack(IImmutableList<Card>)

```
public CardStack(IImmutableList<Card> Cards)
```

参数

Cards IImmutableList<[Card](#)>

参数

Cards

```
public IImmutableList<Card> Cards { get; init; }
```

返回值

IImmutableList<[Card](#)>

ICardStack

宣告: [ProjectStack.Common.Card](#)

位置: ProjectStack.dll

```
public interface ICardStack
```



Cards

```
IImmutableList<Card> Cards { get; }
```



IImmutableList<[Card](#)>

ProjectStack.Common.Recipe

□

[AbstractRecipe<TRecipeInput>](#)

[CardIdMatchRecipe](#)

[RecipeInput](#)

[RecipeOutput](#)

[RecipeRegistrationHelper](#)

[RecipeResult](#)

[ScriptRecipeInput](#)

[ScriptRecipeInput.ScriptContext](#)

[SimpleRecipe](#)

□□□□□□□□□

[SimpleRecipe.Ingredient](#)

[SimpleRecipe.Product](#)

□□

[IRecipe](#)

[IRecipeInput](#)

AbstractRecipe<TRecipeInput>

项目: [ProjectStack.Common.Recipe](#)

文件: ProjectStack.dll

```
public abstract class AbstractRecipe<TRecipeInput> : ObservableValidator
```

参数

TRecipeInput

类

```
object ← ObservableObject ← ObservableValidator ← AbstractRecipe<TRecipeInput>
```

属性

Type

[Required]

```
public abstract ResourceLocation Type { get; }
```

参数

[ResourceLocation](#)

方法

Assmble(TRecipeInput, JsonObject)

```
public abstract RecipeOutput Assmble(TRecipeInput recipeInput, JsonObject ntj)
```

参数

recipeInput TRecipeInput

ntj JsonObject

□□

[RecipeOutput](#)

Matches(TRecipeInput, JsonObject)

```
public abstract bool Matches(TRecipeInput recipeInput, JsonObject ntj)
```

□□

recipeInput TRecipeInput

ntj JsonObject

□□

bool

CardIdMatchRecipe

空间: [ProjectStack.Common.Recipe](#)

文件: ProjectStack.dll

```
public class CardIdMatchRecipe : ObservableValidator, IRecipe
```

类
object ← ObservableObject ← ObservableValidator ← CardIdMatchRecipe
实现
[IRecipe](#)

方法
CardIdMatchRecipe()

```
public CardIdMatchRecipe()
```

CardIdMatchRecipe(List<ResourceLocation>,
List<ResourceLocation>, bool)

```
public CardIdMatchRecipe(List<ResourceLocation> requiredCardIds,  
List<ResourceLocation> producedCardIds, bool isNeedOrder = false)
```

参数
requiredCardIds List<[ResourceLocation](#)>
producedCardIds List<[ResourceLocation](#)>
isNeedOrder bool

方法

CardViewCount

卡牌视图计数
null

```
public uint? CardViewCount { get; }
```

uint

uint?

IsNeedOrder

```
public bool IsNeedOrder { get; }
```

bool

bool

ProducedCardIds

```
public IReadOnlyList<ResourceLocation> ProducedCardIds { get; }
```

IReadOnlyList

<[ResourceLocation](#)>

RequiredCardIds

```
public IReadOnlyList<ResourceLocation> RequiredCardIds { get; }
```

IReadOnlyList

<[ResourceLocation](#)>

□□

Execute(ICardStack)

□□□□□□□

```
public RecipeResult Execute(ICardStack cardStack)
```

□□

cardStack [ICardStack](#)

□□□□□□□□□

□□

[RecipeResult](#)

□□□□

IRecipe

概述: [ProjectStack.Common.Recipe](#)

位置: ProjectStack.dll

```
public interface IRecipe
```



CardViewCount

返回卡堆中卡的数量。如果堆为 null，则返回 0。

```
uint? CardViewCount { get; }
```



uint?



Execute(ICardStack)

执行操作。

```
RecipeResult Execute(ICardStack cardStack)
```



cardStack [ICardStack](#)

返回操作结果。



[RecipeResult](#)

IRecipeInput

命名空间: [ProjectStack.Common.Recipe](#)

位置: ProjectStack.dll

```
public interface IRecipeInput
```

RecipeInput

空间: [ProjectStack.Common.Recipe](#)

文件: ProjectStack.dll

```
public abstract record RecipeInput
```

□□

object ← RecipeInput

Derived

[ScriptRecipeInput](#)

□□

Assemble(Card)

```
public abstract Card Assemble(Card card)
```

□□

card [Card](#)

□□

[Card](#)

IsMatch(Card)

```
public abstract bool IsMatch(Card card)
```

□□

card [Card](#)

□□

bool

RecipeOutput

项目: [ProjectStack.Common.Recipe](#)

文件: ProjectStack.dll

```
public record RecipeOutput
```

类

object ← RecipeOutput

方法

RecipeOutput(List<Card>, JsonObject)

```
public RecipeOutput(List<Card> cards, JsonObject ntj)
```

参数

Cards List<[Card](#)>

ntj JsonObject

返回值

Cards

```
public List<Card> Cards { get; init; }
```

参数

List<[Card](#)>

Ntj

```
public JsonObject Ntj { get; init; }
```

□□□

JsonObject

RecipeRegistrationHelper

项目: [ProjectStack.Common.Recipe](#)

位置: ProjectStack.dll

```
public class RecipeRegistrationHelper
```

类

```
object ← RecipeRegistrationHelper
```

方法

RecipeRegistrationHelper(IServiceCollection)

```
public RecipeRegistrationHelper(IServiceCollection services)
```

参数

```
services IServiceCollection
```

方法

Add(IRecipe)

```
public RecipeRegistrationHelper Add(IRecipe recipe)
```

参数

```
recipe IRecipe
```

方法

[RecipeRegistrationHelper](#)

End()

```
public IServiceCollection End()
```



IServiceCollection

RecipeResult

空间: [ProjectStack.Common.Recipe](#)

模块: ProjectStack.dll

```
public record RecipeResult
```

类

object ← RecipeResult

方法

RecipeResult(bool, IEnumerable<Card>,
IEnumerable<Card>)

```
public RecipeResult(bool IsMatch, IEnumerable<Card> ConsumedCards,  
IEnumerable<Card> ProducedCards)
```

参数

.IsMatch bool

ConsumedCards IEnumerable<[Card](#)>

ProducedCards IEnumerable<[Card](#)>

属性

ConsumedCards

```
public IEnumerable<Card> ConsumedCards { get; init; }
```

方法

IEnumerable<[Card](#)>

IsMatch

```
public bool IsMatch { get; init; }
```

□□□

bool

ProducedCards

```
public IEnumerable<Card> ProducedCards { get; init; }
```

□□□

IEnumerable<[Card](#)>

ScriptRecipeInput □

□□□: [ProjectStack.Common.Recipe](#)

□□□: ProjectStack.dll

```
public record ScriptRecipeInput : RecipeInput
```

□□

```
object ← RecipeInput ← ScriptRecipeInput
```

□□□□

ScriptRecipeInput()

```
public ScriptRecipeInput()
```

□□

AssembleScript

```
public string AssembleScript { get; init; }
```

□□□

string

AssembleScriptRunner

```
public ScriptRunner<Card> AssembleScriptRunner { get; }
```

□□□

ScriptRunner<[Card](#)>

MatchScript

```
public string MatchScript { get; init; }
```

□□□

string

MatchScriptRunner

```
public ScriptRunner<bool> MatchScriptRunner { get; }
```

□□□

ScriptRunner<bool>

□□

Assemble(Card)

```
public override Card Assemble(Card card)
```

□□

card [Card](#)

□□

[Card](#)

IsMatch(Card)

```
public override bool IsMatch(Card card)
```

□□

card Card

□□

bool

ScriptRecipeInput.ScriptContext □

□□□: [ProjectStack.Common.Recipe](#)

□□□: ProjectStack.dll

```
public class ScriptRecipeInput.ScriptContext
```

□□

```
object ← ScriptRecipeInput.ScriptContext
```

□□

card

```
public Card card
```

□□□

[Card](#)

SimpleRecipe □

□□□: [ProjectStack.Common.Recipe](#)

□□□: ProjectStack.dll

□□□□□□□□

```
public record SimpleRecipe
```

□□

```
object ← SimpleRecipe
```

□□□□

SimpleRecipe(ResourceLocation, string, string, float,
IImmutableList<Ingredient>, IImmutableList<Product>)

□□□□□□□□

```
public SimpleRecipe(ResourceLocation Id, string Name, string Description, float  
Production, IImmutableList<SimpleRecipe.Ingredient> Ingredients,  
IImmutableList<SimpleRecipe.Product> Products)
```

□□

Id [ResourceLocation](#)

□□□□□

Name string

Description string

Production float

□□□□□

Ingredients IImmutableList<[SimpleRecipe.Ingredient](#)>

□□

Products `IImmutableList<SimpleRecipe.Product>`

□□

□□

Description

```
public string Description { get; init; }
```

□□□

string

Id

□□□□□

```
public ResourceLocation Id { get; init; }
```

□□□

[ResourceLocation](#)

Ingredients

□□

```
public IImmutableList<SimpleRecipe.Ingredient> Ingredients { get; init; }
```

□□□

`IImmutableList<SimpleRecipe.Ingredient>`

Name

```
public string Name { get; init; }
```

□□□

string

Production

□□□□□

```
public float Production { get; init; }
```

□□□

float

Products

□□

```
public IImmutableList<SimpleRecipe.Product> Products { get; init; }
```

□□□

IImmutableList<[SimpleRecipe.Product](#)>

□□

SatisfactionCheck(IImmutableList<CardMeta>)

```
public bool SatisfactionCheck(IImmutableList<CardMeta> cards)
```

□□

cards ImmutableList<CardMeta>

□□

bool

SimpleRecipe.Ingredient □

▪▪▪▪: [ProjectStack.Common.Recipe](#)

▪▪▪: ProjectStack.dll

```
public record SimpleRecipe.Ingredient
```

□□

object ← SimpleRecipe.Ingredient

□□□□

Ingredient(ResourceLocation, int, bool)

```
public Ingredient(ResourceLocation CardId, int Quantity, bool Consumed)
```

□□

CardId [ResourceLocation](#)

Quantity int

Consumed bool

□□

CardId

```
public ResourceLocation CardId { get; init; }
```

□□□

[ResourceLocation](#)

Consumed

```
public bool Consumed { get; init; }
```

□□□

bool

Quantity

```
public int Quantity { get; init; }
```

□□□

int

SimpleRecipe.Product

位于: [ProjectStack.Common.Recipe](#)

在: ProjectStack.dll

```
public record SimpleRecipe.Product
```

类

object ← SimpleRecipe.Product

方法

Product(ResourceLocation, int)

```
public Product(ResourceLocation CardId, int Quantity)
```

参数

CardId [ResourceLocation](#)

Quantity int

属性

CardId

```
public ResourceLocation CardId { get; init; }
```

类型

[ResourceLocation](#)

Quantity

```
public int Quantity { get; init; }
```

□□□

int

ProjectStack.Component □□□



[Card](#)

A 2D game object, with a transform (position, rotation, and scale). All 2D nodes, including physics objects and sprites, inherit from Node2D. Use Node2D as a parent node to move, scale and rotate children in a 2D project. Also gives control of the node's render order.

[Card.TaskNotifier](#)

A wrapping class that can hold a System.Threading.Tasks.Task value.

[Card.TaskNotifier<T>](#)

A wrapping class that can hold a System.Threading.Tasks.Task<TResult> value.

[InfoTab](#)

Base class for all UI-related nodes. Godot.Control features a bounding rectangle that defines its extents, an anchor position relative to its parent control or the current viewport, and offsets relative to the anchor. The offsets update automatically when the node, any of its parents, or the screen size change.

For more information on Godot's UI system, anchors, offsets, and containers, see the related tutorials in the manual. To build flexible UIs, you'll need a mix of UI elements that inherit from Godot.Control and Godot.Container nodes.

User Interface nodes and input

Godot propagates input events via viewports. Each Godot.Viewport is responsible for propagating Godot.InputEvents to their child nodes. As the Godot.SceneTree.Root is a Godot.Window, this already happens automatically for all UI elements in your game.

Input events are propagated through the Godot.SceneTree from the root node to all child nodes by calling Godot.Node._Input(Godot.InputEvent). For UI elements specifically, it makes more sense to override the virtual method Godot.Control._GUILInput(Godot.InputEvent), which filters out unrelated input events, such as by checking z-order, Godot.Control.MouseFilter, focus, or if the event was inside of the control's bounding box.

Call Godot.Control.AcceptEvent() so no other node receives the event. Once you accept an input, it becomes handled so Godot.Node._UnhandledInput(Godot.InputEvent) will not process it.

Only one Godot.Control node can be in focus. Only the node in focus will receive events. To get the focus, call Godot.Control.GrabFocus(). Godot.Control nodes lose focus when another node grabs it, or if you hide the node in focus.

Sets Godot.Control.MouseFilter to Godot.Control.MouseFilterEnum.Ignore to tell a Godot.Control node to ignore mouse or touch events. You'll need it if you place an icon on top of a button.

Godot.Theme resources change the Control's appearance. If you change the Godot.Theme on a Godot.Control node, it affects all of its children. To override some of the theme's parameters, call one of the `add_theme_*_override` methods, like Godot.Control.AddThemeFontOverride(Godot.StringName, Godot.Font). You can override the theme with the Inspector.

Note: Theme items are *not* Godot.GodotObject properties. This means you can't access their values using Godot.GodotObject.Get(Godot.StringName) and Godot.GodotObject.Set(Godot.StringName, Godot.Variant). Instead, use the `get_theme_*` and `add_theme_*_override` methods provided by this class.

[InfoTab.TaskNotifier](#)

A wrapping class that can hold a System.Threading.Tasks.Task value.

[InfoTab.TaskNotifier<T>](#)

A wrapping class that can hold a System.Threading.Tasks.Task<TResult> value.

Card

项目: [ProjectStack.Component](#)

文件: ProjectStack.dll

A 2D game object, with a transform (position, rotation, and scale). All 2D nodes, including physics objects and sprites, inherit from Node2D. Use Node2D as a parent node to move, scale and rotate children in a 2D project. Also gives control of the node's render order.

```
[ObservableObject]
[Meta(new Type[] { typeof(IAutoNode) })]
[ScriptPath("res://src/scripts/Component/Card.cs")]
public class Card : Node2D, INtjObject
```

类

object ← GodotObject ← Node ← CanvasItem ← Node2D ← Card

接口

[INtjObject](#)

方法

Card()

```
public Card()
```

属性

ForceMotion

bool

```
public bool ForceMotion
```

方法

bool

InMoveing

□□□□□

```
public bool InMoveing
```

□□□

bool

Loaded

```
public bool Loaded
```

□□□

bool

□□

BottomCard

□□□□□□□

```
public Card? BottomCard { get; set; }
```

□□□

Card

BottomCards

□□□□□□□□

```
public IImmutableList<Card> BottomCards { get; }
```

□□□

IImmutableList<[Card](#)>

CardMeta

```
public CardMeta CardMeta { get; set; }
```

□□□

[CardMeta](#)

CardNameLabel

```
public Label? CardNameLabel { get; }
```

□□□

Label

CardStack

□□

```
public ICardStack CardStack { get; }
```

□□□

[ICardStack](#)

CharacterBody

```
public CharacterBody2D? CharacterBody { get; }
```

□□□

CharacterBody2D

CurrentStack

□□□□□□□□

```
public ImmutableList<Card> CurrentStack { get; }
```

□□□

IImmutableList<[Card](#)>

IsRoot

□□□□□□

```
public bool IsRoot { get; }
```

□□□

bool

IsUppest

□□□□□□□

```
public bool IsUppest { get; }
```

□□□

bool

Metatype

Generated metatype information.

```
public IMetatype Metatype { get; }
```



IMetatype

MixinState

Arbitrary data that is shared between mixins. Mixins are free to store additional instance state in this blackboard.

```
public MixinBlackboard MixinState { get; }
```



MixinBlackboard

Ntj

JSON object

```
public JsonObject Ntj { get; set; }
```



JsonObject

OnDrag

Boolean value

```
public bool OnDrag { get; set; }
```

□□□

bool

Panel

```
public Control? Panel { get; }
```

□□□

Control

RootCard

□□□

```
public Card RootCard { get; }
```

□□□

Card

TargetPosition

□□□□□□

```
public Vector2 TargetPosition { get; set; }
```

□□□

Vector2

TextureRect

□□

```
public TextureRect? TextureRect { get; }
```

□□□

TextureRect

TopCard

□□□□□□□

```
public Card? TopCard { get; set; }
```

□□□

Card

TopCards

□□□□□□□□

```
public ImmutableList<Card> TopCards { get; }
```

□□□

ImmutableList<Card>

UppestCard

□□□

```
public Card UppestCard { get; }
```

□□□

Card

□□

CanDrag()

```
public bool CanDrag()
```

□□

bool

Deserialize(FriendlyNtjObject)

```
public void Deserialize(FriendlyNtjObject ntj)
```

□□

ntj [FriendlyNtjObject](#)

GetAttributeHolder(AttributeHolder)

```
public AttributeHolder GetAttributeHolder(AttributeHolder type)
```

□□

type [AttributeHolder](#)

□□

[AttributeHolder](#)

GetAttributeHolder(ResourceLocation)

```
public AttributeHolder GetAttributeHolder(ResourceLocation id)
```

□□

id [ResourceLocation](#)

□□

[AttributeHolder](#)

GetCardsInRadius()

```
public IImmutableList<Card> GetCardsInRadius()
```

□□

IImmutableList<[Card](#)>

GetCardsInRadius(float)

□□□□□□□□□□

```
public IImmutableList<Card> GetCardsInRadius(float radius)
```

□□

radius float

□□

IImmutableList<[Card](#)>

GetCollidingCards(ShapeCast2D)

```
public List<Card> GetCollidingCards(ShapeCast2D cast2D)
```

□□

cast2D ShapeCast2D



List<[Card](#)>

GetGodotClassPropertyValue(in godot_string_name, out godot_variant)

Get the value of a property contained in this class. This method is used by Godot to retrieve property values. Do not call or override this method.

```
protected override bool GetGodotClassPropertyValue(in godot_string_name name, out godot_variant value)
```



name godot_string_name

Name of the property to get.

value godot_variant

Value of the property if it was found.



bool

[true](#) if a property with the given name was found.

GetNearestSameCard(Card)



```
public Card? GetNearestSameCard(Card card)
```



card [Card](#)

□□□□□□□

□□

[Card](#)

□□

GetNearestSameCard(IImmutableList<Card>, Card)

public Card? GetNearestSameCard(IImmutableList<Card> cards, Card card)

□□

cards IImmutableList<[Card](#)>

card [Card](#)

□□

[Card](#)

HasGodotClassMethod(in godot_string_name)

Check if the type contains a method with the given name. This method is used by Godot to check if a method exists before invoking it. Do not call or override this method.

protected override bool HasGodotClassMethod(in** godot_string_name method)**

□□

method godot_string_name

Name of the method to check for.

□□

bool

InvokeGodotClassMethod(**in** godot_string_name, Native VariantPtrArgs, **out** godot_variant)

Invokes the method with the given name, using the given arguments. This method is used by Godot to invoke methods from the engine side. Do not call or override this method.

```
protected override bool InvokeGodotClassMethod(in godot_string_name method,  
NativeVariantPtrArgs args, out godot_variant ret)
```

□□

method godot_string_name

Name of the method to invoke.

args NativeVariantPtrArgs

Arguments to use with the invoked method.

ret godot_variant

Value returned by the invoked method.

□□

bool

Load()

```
public void Load()
```

OnAssemble()

```
public void OnAssemble()
```

OnCardStackChanged()

□□[CardStackChanged](#)□□□

```
protected virtual void OnCardStackChanged()
```

OnClick()

```
public void OnClick()
```

OnDraging()

```
public void OnDraging()
```

OnDrop()

```
public void OnDrop()
```

OnHover()

```
public void OnHover()
```

OnPropertyChanged(PropertyChangedEventArgs)

Raises the [PropertyChanged](#) event.

```
protected virtual void OnPropertyChanged(PropertyChangedEventArgs e)
```

□□

e PropertyChangedEventArgs

The input System.ComponentModel.PropertyChangedEventArgs instance.

OnPropertyChanged(string?)

Raises the [PropertyChanged](#) event.

```
protected void OnPropertyChanged(string? propertyName = null)
```

□□

propertyName string

(optional) The name of the property that changed.

OnPropertyChanging(PropertyChangingEventArgs)

Raises the [PropertyChanging](#) event.

```
protected virtual void OnPropertyChanging(PropertyChangingEventArgs e)
```

□□

e PropertyChangingEventArgs

The input System.ComponentModel.PropertyChangingEventArgs instance.

OnPropertyChanged(string?)

Raises the [PropertyChanged](#) event.

```
protected void OnPropertyChanged(string? propertyName = null)
```

□□

propertyName string

(optional) The name of the property that changed.

OnReady()

□□□

```
public void OnReady()
```

OnRecipeMatch()

```
public void OnRecipeMatch()
```

RefreshTexture()

□□□□

```
public void RefreshTexture()
```

RestoreGodotObjectData(GodotSerializationInfo)

Restores this instance's state after reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void RestoreGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object that contains the previously saved data.

Save()

```
public void Save()
```

SaveGodotObjectData(GodotSerializationInfo)

Saves this instance's state to be restored when reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void SaveGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object used to save the data.

Serialize(FriendlyNtjObject)

```
public void Serialize(FriendlyNtjObject ntj)
```

□□

ntj [FriendlyNtjObject](#)

SetGodotClassPropertyValue(in godot_string_name, in godot_variant)

Set the value of a property contained in this class. This method is used by Godot to assign property values. Do not call or override this method.

```
protected override bool SetGodotClassPropertyValue(in godot_string_name name, in godot_variant value)
```

□□

name godot_string_name

Name of the property to set.

value godot_variant

Value to set the property to if it was found.

□□

bool

[true](#) if a property with the given name was found.

SetPropertyAndNotifyOnCompletion(ref TaskNotifier?, Task?, Action<Task?>, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. This method is just like [SetPropertyAndNotifyOnCompletion\(ref TaskNotifier?, Task?, string?\)](#), with the difference being an extra System.Action<T> parameter with a callback being invoked either immediately, if the new task has already completed or is [null](#), or upon completion.

```
protected bool SetPropertyAndNotifyOnCompletion(ref Card.TaskNotifier? taskNotifier,  
Task? newValue, Action<Task?> callback, string? propertyName = null)
```

□□

taskNotifier [Card.TaskNotifier](#)

The field notifier to modify.

newValue Task

The property's value after the change occurred.

callback Action<Task>

A callback to invoke to update the property value.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.



The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetPropertyAndNotifyOnCompletion(ref TaskNotifier?, Task?, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method will also monitor the new value of the property (a generic System.Threading.Tasks.Task) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that System.Threading.Tasks.Task or any of its properties. This method and its overload specifically rely on the [Card.TaskNotifier](#) type, which needs to be used in the backing field for the target System.Threading.Tasks.Task property. The field doesn't need to be initialized, as this method will take care of doing that automatically. The [Card.TaskNotifier](#) type also includes an implicit operator, so it can be assigned to any System.Threading.Tasks.Task instance directly. Here is a sample property declaration using this method:

```
private TaskNotifier myTask;

public Task MyTask
{
    get => myTask;
    private set => SetAndNotifyOnCompletion(ref myTask, value);
}

protected bool SetPropertyAndNotifyOnCompletion(ref Card.TaskNotifier? taskNotifier,
Task? newValue, string? propertyName = null)
```



taskNotifier [Card.TaskNotifier](#)

The field notifier to modify.

`newValue` Task

The property's value after the change occurred.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

`true` if the property was changed, `false` otherwise.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same. The return value being `true` only indicates that the new value being assigned to `taskNotifier` is different than the previous one, and it does not mean the new System.Threading.Tasks.Task instance passed as argument is in any particular state.

SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>?, Task<T>?, Action<Task<T>?>, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. This method is just like [SetPropertyAndNotifyOnCompletion<T>\(ref TaskNotifier<T>?, Task<T>?, string?\)](#), with the difference being an extra System.Action<T> parameter with a callback being invoked either immediately, if the new task has already completed or is `null`, or upon completion.

```
protected bool SetPropertyAndNotifyOnCompletion<T>(ref Card.TaskNotifier<T>? taskNotifier, Task<T>? newValue, Action<Task<T>?> callback, string? propertyName = null)
```

□□

`taskNotifier` [Card.TaskNotifier<T>](#)

The field notifier to modify.

newValue Task<T>

The property's value after the change occurred.

callback Action<Task<T>>

A callback to invoke to update the property value.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of result for the System.Threading.Tasks.Task<TResult> to set and monitor.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>?, Task<T>?, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method will also monitor the new value of the property (a generic System.Threading.Tasks.Task) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that System.Threading.Tasks.Task or any of its properties. This method and its overload specifically rely on the [Card.TaskNotifier<T>](#) type, which needs to be used in the backing field for the target System.Threading.Tasks.Task

property. The field doesn't need to be initialized, as this method will take care of doing that automatically. The [Card.TaskNotifier<T>](#) type also includes an implicit operator, so it can be assigned to any System.Threading.Tasks.Task instance directly. Here is a sample property declaration using this method:

```
private TaskNotifier<int> myTask;

public Task<int> MyTask
{
    get => myTask;
    private set => SetAndNotifyOnCompletion(ref myTask, value);
}

protected bool SetPropertyAndNotifyOnCompletion<T>(ref Card.TaskNotifier<T>?
taskNotifier, Task<T>? newValue, string? propertyName = null)
```

□□

taskNotifier [Card.TaskNotifier<T>](#)

The field notifier to modify.

newValue Task<T>

The property's value after the change occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of result for the System.Threading.Tasks.Task<TResult> to set and monitor.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same. The return value being [true](#) only indicates that the new value being assigned to [taskNotifier](#) is different than the previous one, and it does not mean the new `System.Threading.Tasks.Task<TResult>` instance passed as argument is in any particular state.

SetProperty<T>(T, T, Action<T>, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. This overload is much less efficient than [SetProperty<T>\(ref T, T, string?\)](#) and it should only be used when the former is not viable (eg. when the target property being updated does not directly expose a backing field that can be passed by reference). For performance reasons, it is recommended to use a stateful callback if possible through the [SetProperty<TModel, T>\(T, T, TModel, Action<TModel, T>, string?\)](#) whenever possible instead of this overload, as that will allow the C# compiler to cache the input callback and reduce the memory allocations. More info on that overload are available in the related XML docs. This overload is here for completeness and in cases where that is not applicable.

```
protected bool SetProperty<T>(T oldValue, T newValue, Action<T> callback, string?  
propertyName = null)
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

callback Action<T>

A callback to invoke to update the property value.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□

T

The type of the property that changed.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetProperty<T>(T, T, IEqualityComparer<T>, Action<T>, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. See additional notes about this overload in [SetProperty<T>\(T, T, Action<T>, string?\)](#).

```
protected bool SetProperty<T>(T oldValue, T newValue, IEqualityComparer<T> comparer,  
Action<T> callback, string? propertyName = null)
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

comparer IEqualityComparer<T>

The System.Collections.Generic.IEqualityComparer<T> instance to use to compare the input values.

`callback Action<T>`

A callback to invoke to update the property value.

`propertyName` string

(optional) The name of the property that changed.

□□

`bool`

`true` if the property was changed, `false` otherwise.

□□□□

T

The type of the property that changed.

SetProperty<T>(ref T, T, IEqualityComparer<T>, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. See additional notes about this overload in [SetProperty<T>\(ref T, T, string?\)](#).

```
protected bool SetProperty<T>(ref T field, T newValue, IEqualityComparer<T> comparer, string? propertyName = null)
```

□□

`field` T

The field storing the property's value.

`newValue` T

The property's value after the change occurred.

`comparer` IEqualityComparer<T>

The `System.Collections.Generic.IEqualityComparer<T>` instance to use to compare the input values.

`propertyName` `string`

(optional) The name of the property that changed.

□□

`bool`

`true` if the property was changed, `false` otherwise.

□□□□

`T`

The type of the property that changed.

`SetProperty<T>(ref T, T, string?)`

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event.

```
protected bool SetProperty<T>(ref T field, T newValue, string? propertyName = null)
```

□□

`field` `T`

The field storing the property's value.

`newValue` `T`

The property's value after the change occurred.

`propertyName` `string`

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of the property that changed.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetProperty<TModel, T>(T, T, IEqualityComparer<T>, TModel, Action<TModel, T>, string?)

Compares the current and new values for a given nested property. If the value has changed, raises the [PropertyChanging](#) event, updates the property and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method is used to relay properties from a wrapped model in the current instance. See additional notes about this overload in [SetProperty<TModel, T>\(T, T, TModel, Action<TModel, T>, string?\)](#).

```
protected bool SetProperty<TModel, T>(T oldValue, T newValue, IEqualityComparer<T> comparer, TModel model, Action<TModel, T> callback, string? propertyName = null)  
where TModel : class
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

comparer IEqualityComparer<T>

The System.Collections.Generic.IEqualityComparer<T> instance to use to compare the input values.

model TModel

The model containing the property being updated.

callback Action<TModel, T>

The callback to invoke to set the target property value, if a change has occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

TModel

The type of model whose property (or field) to set.

T

The type of property (or field) to set.

SetProperty<TModel, T>(T, T, TModel, Action<TModel, T>, string?)

Compares the current and new values for a given nested property. If the value has changed, raises the [PropertyChanging](#) event, updates the property and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method is used to relay properties from a wrapped model in the current instance. This type is useful when creating wrapping, bindable objects that operate over models that lack support for notification (eg. for CRUD operations). Suppose we have this model (eg. for a database row in a table):

```
public class Person
{
    public string Name { get; set; }
}
```

We can then use a property to wrap instances of this type into our observable model (which supports notifications), injecting the notification to the properties of that model, like so:

```
[ObservableObject]
public class BindablePerson
{
    public Model { get; }

    public BindablePerson(Person model)
    {
        Model = model;
    }

    public string Name
    {
        get => Model.Name;
        set => Set(Model.Name, value, Model, (model, name) => model.Name = name);
    }
}
```

This way we can then use the wrapping object in our application, and all those "proxy" properties will also raise notifications when changed. Note that this method is not meant to be a replacement for [SetProperty<T>\(ref T, T, string?\)](#), and it should only be used when relaying properties to a model that doesn't support notifications, and only if you can't implement notifications to that model directly (eg. by having it inherit from `ObservableObject`). The syntax relies on passing the target model and a stateless callback to allow the C# compiler to cache the function, which results in much better performance and no memory usage.

```
protected bool SetProperty<TModel, T>(T oldValue, T newValue, TModel model,
Action<TModel, T> callback, string? propertyName = null) where TModel : class
```



`oldValue` `T`

The current property value.

`newValue` T

The property's value after the change occurred.

`model` TModel

The model containing the property being updated.

`callback` Action<TModel, T>

The callback to invoke to set the target property value, if a change has occurred.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

TModel

The type of model whose property (or field) to set.

T

The type of property (or field) to set.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

UpdateAllZIndex()

□□□□Z□□□□□□□□

```
public void UpdateAllZIndex()
```

UpdateRecipe()

方法

```
public void UpdateRecipe()
```

UpdateZIndex()

方法

```
public void UpdateZIndex()
```

_Notification(int)

Called when the object receives a notification, which can be identified in `what` by comparing it with a constant. See also `Godot.GodotObject.Notification(int, bool)`.

```
public override void _Notification(int what)
{
    if (what == NotificationPredelete)
    {
        GD.Print("Goodbye!");
    }
}
```

Note: The base `Godot.GodotObject` defines a few notifications (`Godot.GodotObject.NotificationPostinitialize` and `Godot.GodotObject.NotificationPredelete`). Inheriting classes such as `Godot.Node` define a lot more notifications, which are also received by this method.

```
public override void _Notification(int what)
```

参数

`what` int

_Process(double)

◆◆◆

```
public override void _Process(double delta)
```

◆◆

delta double

_Ready()

Called when the node is "ready", i.e. when both the node and its children have entered the scene tree. If the node has children, their Godot.Node._Ready() callbacks get triggered first, and the parent node will receive the ready notification afterwards.

Corresponds to the Godot.Node.NotificationReady notification in Godot.GodotObject._Notification(int). See also the `@onready` annotation for variables.

Usually used for initialization. For even earlier initialization, Godot.GodotObject.GodotObject() may be used. See also Godot.Node._EnterTree().

Note: This method may be called only once for each node. After removing a node from the scene tree and adding it again, Godot.Node._Ready() will **not** be called a second time. This can be bypassed by requesting another call with Godot.Node.RequestReady(), which may be called anywhere before adding the node again.

```
public override void _Ready()
```

◆◆

CardStackChanged

◆◆◆◆◆◆◆◆◆◆

```
public event Action<ICardStack>? CardStackChanged
```

◆◆◆◆

Action<[ICardStack](#)>

PropertyChanged

Occurs when a property value changes.

```
public event PropertyChangedEventHandler? PropertyChanged
```



PropertyChangedEventHandler

PropertyChanging

Occurs when a property value is changing.

```
public event PropertyChangingEventHandler? PropertyChanging
```



PropertyChangingEventHandler

Card.TaskNotifier □

▪▪▪▪: [ProjectStack.Component](#)

▪▪▪: ProjectStack.dll

A wrapping class that can hold a System.Threading.Tasks.Task value.

```
protected sealed class Card.TaskNotifier
```

□□

```
object ← Card.TaskNotifier
```

□□□

implicit operator Task?(TaskNotifier?)

Unwraps the System.Threading.Tasks.Task value stored in the current instance.

```
public static implicit operator Task?(Card.TaskNotifier? notifier)
```

□□

```
notifier Card.TaskNotifier
```

The input [Card.TaskNotifier<T>](#) instance.

□□

Task

Card.TaskNotifier<T>

项目: [ProjectStack.Component](#)

位置: ProjectStack.dll

A wrapping class that can hold a System.Threading.Tasks.Task<TResult> value.

```
protected sealed class Card.TaskNotifier<T>
```

参数

T

The type of value for the wrapped System.Threading.Tasks.Task<TResult> instance.

示例

```
object ← Card.TaskNotifier<T>
```

方法

implicit operator Task<T>?(TaskNotifier<T>?)

Unwraps the System.Threading.Tasks.Task<TResult> value stored in the current instance.

```
public static implicit operator Task<T>?(Card.TaskNotifier<T>? notifier)
```

参数

notifier [Card.TaskNotifier<T>](#)

The input [Card.TaskNotifier<T>](#) instance.

返回值

Task<T>

InfoTab □

类别: [ProjectStack.Component](#)

文件: ProjectStack.dll

Base class for all UI-related nodes. Godot.Control features a bounding rectangle that defines its extents, an anchor position relative to its parent control or the current viewport, and offsets relative to the anchor. The offsets update automatically when the node, any of its parents, or the screen size change.

For more information on Godot's UI system, anchors, offsets, and containers, see the related tutorials in the manual. To build flexible UIs, you'll need a mix of UI elements that inherit from Godot.Control and Godot.Container nodes.

User Interface nodes and input

Godot propagates input events via viewports. Each Godot.Viewport is responsible for propagating Godot.InputEvents to their child nodes. As the Godot.SceneTree.Root is a Godot.Window, this already happens automatically for all UI elements in your game.

Input events are propagated through the Godot.SceneTree from the root node to all child nodes by calling Godot.Node._Input(Godot.InputEvent). For UI elements specifically, it makes more sense to override the virtual method Godot.Control._GUILInput(Godot.InputEvent), which filters out unrelated input events, such as by checking z-order, Godot.Control.MouseFilter, focus, or if the event was inside of the control's bounding box.

Call Godot.Control.AcceptEvent() so no other node receives the event. Once you accept an input, it becomes handled so Godot.Node._UnhandledInput(Godot.InputEvent) will not process it.

Only one Godot.Control node can be in focus. Only the node in focus will receive events. To get the focus, call Godot.Control.GrabFocus(). Godot.Control nodes lose focus when another node grabs it, or if you hide the node in focus.

Sets Godot.Control.MouseFilter to Godot.Control.MouseFilterEnum.Ignore to tell a Godot.Control node to ignore mouse or touch events. You'll need it if you place an icon on top of a button.

Godot.Theme resources change the Control's appearance. If you change the Godot.Theme on a Godot.Control node, it affects all of its children. To override some of the theme's parameters, call one of the `add_theme_*_override` methods, like Godot.Control.AddTheme

`FontOverride(Godot.StringName, Godot.Font)`. You can override the theme with the Inspector.

Note: Theme items are *not* `Godot.GodotObject` properties. This means you can't access their values using `Godot.GodotObject.Get(Godot.StringName)` and `Godot.GodotObject.Set(Godot.StringName, Godot.Variant)`. Instead, use the `get_theme_*` and `add_theme_*_override` methods provided by this class.

```
[ObservableObject]
[Meta(new Type[] { typeof(IAutoNode) })]
[ScriptPath("res://src/scripts/Component/InfoTab.cs")]
public class InfoTab : Control
```



`object ← GodotObject ← Node ← CanvasItem ← Control ← InfoTab`



Metatype

Generated metatype information.

```
public IMetatype Metatype { get; }
```



`IMetatype`

MixinState

Arbitrary data that is shared between mixins. Mixins are free to store additional instance state in this blackboard.

```
public MixinBlackboard MixinState { get; }
```



`MixinBlackboard`



GetGodotClassPropertyValue(*in godot_string_name*, *out godot_variant*)

Get the value of a property contained in this class. This method is used by Godot to retrieve property values. Do not call or override this method.

```
protected override bool GetGodotClassPropertyValue(in godot_string_name name, out godot_variant value)
```



name *godot_string_name*

Name of the property to get.

value *godot_variant*

Value of the property if it was found.



bool

[true](#) if a property with the given name was found.

HasGodotClassMethod(*in godot_string_name*)

Check if the type contains a method with the given name. This method is used by Godot to check if a method exists before invoking it. Do not call or override this method.

```
protected override bool HasGodotClassMethod(in godot_string_name method)
```



method *godot_string_name*

Name of the method to check for.

□□

bool

InvokeGodotClassMethod(*in* godot_string_name, Native VariantPtrArgs, *out* godot_variant)

Invokes the method with the given name, using the given arguments. This method is used by Godot to invoke methods from the engine side. Do not call or override this method.

```
protected override bool InvokeGodotClassMethod(in godot_string_name method,  
NativeVariantPtrArgs args, out godot_variant ret)
```

□□

method godot_string_name

Name of the method to invoke.

args NativeVariantPtrArgs

Arguments to use with the invoked method.

ret godot_variant

Value returned by the invoked method.

□□

bool

OnProcess(double)

Notification received from the tree every rendered frame when Godot.Node.IsPhysics Processing() returns true.

```
public void OnProcess(double delta)
```

□□

`delta` double

Time since the last process update, in seconds.

OnPropertyChanged(PropertyChangedEventArgs)

Raises the [PropertyChanged](#) event.

```
protected virtual void OnPropertyChanged(PropertyChangedEventArgs e)
```

□□

`e` PropertyChangedEventArgs

The input System.ComponentModel.PropertyChangedEventArgs instance.

OnPropertyChanged(string?)

Raises the [PropertyChanged](#) event.

```
protected void OnPropertyChanged(string? propertyName = null)
```

□□

`propertyName` string

(optional) The name of the property that changed.

OnPropertyChanging(PropertyChangingEventArgs)

Raises the [PropertyChanging](#) event.

```
protected virtual void OnPropertyChanging(PropertyChangingEventArgs e)
```

□□

`e` PropertyChangingEventArgs

The input System.ComponentModel.PropertyChangingEventArgs instance.

OnPropertyChanging(string?)

Raises the [PropertyChanging](#) event.

```
protected void OnPropertyChanging(string? propertyName = null)
```

□□

propertyName string

(optional) The name of the property that changed.

OnReady()

Notification received when the node is ready.

```
public void OnReady()
```

RestoreGodotObjectData(GodotSerializationInfo)

Restores this instance's state after reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void RestoreGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

Saves this instance's state to be restored when reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void SaveGodotObjectData(GodotSerializationInfo info)
```



info GodotSerializationInfo

Object used to save the data.

SetGodotClassPropertyValue(in godot_string_name, in godot_variant)

Set the value of a property contained in this class. This method is used by Godot to assign property values. Do not call or override this method.

```
protected override bool SetGodotClassPropertyValue(in godot_string_name name, in godot_variant value)
```



name godot_string_name

Name of the property to set.

value godot_variant

Value to set the property to if it was found.



bool

[true](#) if a property with the given name was found.

SetPropertyAndNotifyOnCompletion(ref TaskNotifier?, Task?, Action<Task?>, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. This method is just like [SetPropertyAndNotifyOnCompletion\(ref TaskNotifier?, Task?, string?\)](#), with the difference being an extra System.Action<T> parameter with a callback being invoked either immediately, if the new task has already completed or is [null](#), or upon completion.

```
protected bool SetPropertyAndNotifyOnCompletion(ref InfoTab.TaskNotifier?  
taskNotifier, Task? newValue, Action<Task?> callback, string? propertyName = null)
```

□□

taskNotifier [InfoTab.TaskNotifier](#)

The field notifier to modify.

newValue Task

The property's value after the change occurred.

callback Action<Task>

A callback to invoke to update the property value.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetPropertyAndNotifyOnCompletion(ref TaskNotifier?, Task?, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method will also monitor the new value of the property (a generic System.Threading.Tasks.Task) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that System.Threading.Tasks.Task or any of its properties. This method and its overload specifically rely on the [InfoTab.TaskNotifier](#) type, which needs to be used in the backing field for the target System.Threading.Tasks.Task property. The field doesn't need to be initialized, as this method will take care of doing that automatically. The [InfoTab.TaskNotifier](#) type also includes an implicit operator, so it can be assigned to any System.Threading.Tasks.Task instance directly. Here is a sample property declaration using this method:

```
private TaskNotifier myTask;

public Task MyTask
{
    get => myTask;
    private set => SetAndNotifyOnCompletion(ref myTask, value);
}
```

```
protected bool SetPropertyAndNotifyOnCompletion(ref InfoTab.TaskNotifier?
taskNotifier, Task? newValue, string? propertyName = null)
```



taskNotifier [InfoTab.TaskNotifier](#)

The field notifier to modify.

newValue Task

The property's value after the change occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same. The return value being [true](#) only indicates that the new value being assigned to `taskNotifier` is different than the previous one, and it does not mean the new System.Threading.Tasks.Task instance passed as argument is in any particular state.

SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>?, Task<T>?, Action<Task<T>?>, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. This method is just like

[SetPropertyAndNotifyOnCompletion<T>\(ref TaskNotifier<T>?, Task<T>?, string?\)](#), with the difference being an extra System.Action<T> parameter with a callback being invoked either immediately, if the new task has already completed or is [null](#), or upon completion.

```
protected bool SetPropertyAndNotifyOnCompletion<T>(ref InfoTab.TaskNotifier<T>? taskNotifier, Task<T>? newValue, Action<Task<T>?> callback, string? propertyName = null)
```

□□

`taskNotifier` [InfoTab.TaskNotifier<T>](#)

The field notifier to modify.

`newValue` Task<T>

The property's value after the change occurred.

`callback` Action<Task<T>>

A callback to invoke to update the property value.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of result for the System.Threading.Tasks.Task<TResult> to set and monitor.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>?, Task<T>?, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method will also monitor the new value of the property (a generic System.Threading.Tasks.Task) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that System.Threading.Tasks.Task or any of its properties. This method and its overload specifically rely on the [InfoTab.TaskNotifier<T>](#) type, which needs to be used in the backing field for the target System.Threading.Tasks.Task property. The field doesn't need to be initialized, as this method will take care of doing that automatically. The [InfoTab.TaskNotifier<T>](#) type also includes an implicit operator, so it can be assigned to any System.Threading.Tasks.Task instance directly. Here is a sample property declaration using this method:

```
private TaskNotifier<int> myTask;

public Task<int> MyTask
{
    get => myTask;
    private set => SetAndNotifyOnCompletion(ref myTask, value);
}

protected bool SetPropertyAndNotifyOnCompletion<T>(ref InfoTab.TaskNotifier<T>?
taskNotifier, Task<T>? newValue, string? propertyName = null)
```

□□

taskNotifier [InfoTab.TaskNotifier](#)<T>

The field notifier to modify.

newValue Task<T>

The property's value after the change occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of result for the System.Threading.Tasks.Task<TResult> to set and monitor.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same. The return value being [true](#) only indicates that the new value being assigned to [taskNotifier](#) is different than the previous one, and it does

not mean the new System.Threading.Tasks.Task<TResult> instance passed as argument is in any particular state.

SetProperty<T>(T, T, Action<T>, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. This overload is much less efficient than [SetProperty<T>\(ref T, T, string?\)](#) and it should only be used when the former is not viable (eg. when the target property being updated does not directly expose a backing field that can be passed by reference). For performance reasons, it is recommended to use a stateful callback if possible through the [SetProperty<TModel, T>\(T, T, TModel, Action<TModel, T>, string?\)](#) whenever possible instead of this overload, as that will allow the C# compiler to cache the input callback and reduce the memory allocations. More info on that overload are available in the related XML docs. This overload is here for completeness and in cases where that is not applicable.

```
protected bool SetProperty<T>(T oldValue, T newValue, Action<T> callback, string?  
propertyName = null)
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

callback Action<T>

A callback to invoke to update the property value.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□

T

The type of the property that changed.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetProperty<T>(T, T, IEqualityComparer<T>, Action<T>, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. See additional notes about this overload in [SetProperty<T>\(T, T, Action<T>, string?\)](#).

```
protected bool SetProperty<T>(T oldValue, T newValue, IEqualityComparer<T> comparer,  
Action<T> callback, string? propertyName = null)
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

comparer IEqualityComparer<T>

The System.Collections.Generic.IEqualityComparer<T> instance to use to compare the input values.

callback Action<T>

A callback to invoke to update the property value.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

`true` if the property was changed, `false` otherwise.

□□□□

T

The type of the property that changed.

SetProperty<T>(ref T, T, IEqualityComparer<T>, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. See additional notes about this overload in [SetProperty<T>\(ref T, T, string?\)](#).

```
protected bool SetProperty<T>(ref T field, T newValue, IEqualityComparer<T>
comparer, string? propertyName = null)
```

□□

`field` T

The field storing the property's value.

`newValue` T

The property's value after the change occurred.

`comparer` IEqualityComparer<T>

The System.Collections.Generic.IEqualityComparer<T> instance to use to compare the input values.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

`true` if the property was changed, `false` otherwise.

□□□□

T

The type of the property that changed.

SetProperty<T>(ref T, T, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event.

```
protected bool SetProperty<T>(ref T field, T newValue, string? propertyName = null)
```

□□

`field` T

The field storing the property's value.

`newValue` T

The property's value after the change occurred.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□

T

The type of the property that changed.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetProperty<TModel, T>(T, T, IEqualityComparer<T>, TModel, Action<TModel, T>, string?)

Compares the current and new values for a given nested property. If the value has changed, raises the [PropertyChanging](#) event, updates the property and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method is used to relay properties from a wrapped model in the current instance. See additional notes about this overload in [SetProperty<TModel, T>\(T, T, TModel, Action<TModel, T>, string?\)](#).

```
protected bool SetProperty<TModel, T>(T oldValue, T newValue, IEqualityComparer<T> comparer, TModel model, Action<TModel, T> callback, string? propertyName = null)  
where TModel : class
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

comparer IEqualityComparer<T>

The System.Collections.Generic.IEqualityComparer<T> instance to use to compare the input values.

model TModel

The model containing the property being updated.

callback Action<TModel, T>

The callback to invoke to set the target property value, if a change has occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

TModel

The type of model whose property (or field) to set.

T

The type of property (or field) to set.

SetProperty<TModel, T>(T, T, TModel, Action<TModel, T>, string?)

Compares the current and new values for a given nested property. If the value has changed, raises the [PropertyChanging](#) event, updates the property and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method is used to relay properties from a wrapped model in the current instance. This type is useful when creating wrapping, bindable objects that operate over models that lack support for notification (eg. for CRUD operations). Suppose we have this model (eg. for a database row in a table):

```
public class Person
{
```

```
    public string Name { get; set; }  
}
```

We can then use a property to wrap instances of this type into our observable model (which supports notifications), injecting the notification to the properties of that model, like so:

```
[ObservableObject]  
public class BindablePerson  
{  
    public Model { get; }  
  
    public BindablePerson(Person model)  
    {  
        Model = model;  
    }  
  
    public string Name  
    {  
        get => Model.Name;  
        set => Set(Model.Name, value, Model, (model, name) => model.Name = name);  
    }  
}
```

This way we can then use the wrapping object in our application, and all those "proxy" properties will also raise notifications when changed. Note that this method is not meant to be a replacement for [SetProperty<T>\(ref T, T, string?\)](#), and it should only be used when relaying properties to a model that doesn't support notifications, and only if you can't implement notifications to that model directly (eg. by having it inherit from `ObservableObject`). The syntax relies on passing the target model and a stateless callback to allow the C# compiler to cache the function, which results in much better performance and no memory usage.

```
protected bool SetProperty<TModel, T>(T oldValue, T newValue, TModel model,  
Action<TModel, T> callback, string? propertyName = null) where TModel : class
```



`oldValue` `T`

The current property value.

`newValue` `T`

The property's value after the change occurred.

model TModel

The model containing the property being updated.

callback Action<TModel, T>

The callback to invoke to set the target property value, if a change has occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

TModel

The type of model whose property (or field) to set.

T

The type of property (or field) to set.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

_Notification(int)

Called when the object receives a notification, which can be identified in **what** by comparing it with a constant. See also Godot.GodotObject.Notification(int, bool).

```
public override void _Notification(int what)
{
    if (what == NotificationPredelete)
    {
```

```
        GD.Print("Goodbye!");
    }
}
```

Note: The base Godot.GodotObject defines a few notifications (Godot.GodotObject.NotificationPostinitialize and Godot.GodotObject.NotificationPredelete). Inheriting classes such as Godot.Node define a lot more notifications, which are also received by this method.

```
public override void _Notification(int what)
```



`what` int



PropertyChanged

Occurs when a property value changes.

```
public event PropertyChangedEventHandler? PropertyChanged
```



`PropertyChangedEventHandler`

PropertyChanging

Occurs when a property value is changing.

```
public event PropertyChangingEventHandler? PropertyChanging
```



`PropertyChangingEventHandler`

InfoTab.TaskNotifier

项目: [ProjectStack.Component](#)

文件: ProjectStack.dll

A wrapping class that can hold a System.Threading.Tasks.Task value.

```
protected sealed class InfoTab.TaskNotifier
```

类

```
object ← InfoTab.TaskNotifier
```

方法

implicit operator Task?(TaskNotifier?)

Unwraps the System.Threading.Tasks.Task value stored in the current instance.

```
public static implicit operator Task?(InfoTab.TaskNotifier? notifier)
```

参数

notifier [InfoTab.TaskNotifier](#)

The input [InfoTab.TaskNotifier<T>](#) instance.

返回值

Task

InfoTab.TaskNotifier<T>

父类: [ProjectStack.Component](#)

位置: ProjectStack.dll

A wrapping class that can hold a System.Threading.Tasks.Task<TResult> value.

```
protected sealed class InfoTab.TaskNotifier<T>
```

参数

T

The type of value for the wrapped System.Threading.Tasks.Task<TResult> instance.

示例

```
object ← InfoTab.TaskNotifier<T>
```

方法

implicit operator Task<T>?(TaskNotifier<T>?)

Unwraps the System.Threading.Tasks.Task<TResult> value stored in the current instance.

```
public static implicit operator Task<T>?(InfoTab.TaskNotifier<T>? notifier)
```

参数

notifier [InfoTab.TaskNotifier<T>](#)

The input [InfoTab.TaskNotifier<T>](#) instance.

返回值

Task<T>

ProjectStack.Core



[FrozenGameResource<T>](#)

[GameResource<T>](#)

[GameResources](#)

[Level](#)

[ServiceCollectionExtension](#)



[IGameResource](#)

FrozenGameResource<T>

空间: [ProjectStack.Core](#)

位置: ProjectStack.dll

```
public class FrozenGameResource<T> : IGameResource where T : class
```

参数

T

类

object ← FrozenGameResource<T>

接口

[IGameResource](#)

方法

Count

Gets the number of elements in the collection.

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

参数

int

The number of elements in the collection.

方法

GetEnumerator()

Returns an enumerator that iterates through the collection.

```
public IEnumarator<T> GetEnumarator()
```

□□

IEnumarator<T>

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

GameResource<T>

空间: [ProjectStack.Core](#)

文件: ProjectStack.dll

```
public class GameResource<T> : IGameResource where T : class
```

参数

T

类

object ← GameResource<T>

接口

[IGameResource](#)

方法

Count

Gets the number of elements contained in the System.Collections.Generic.ICollection<T>.

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

参数

int

The number of elements contained in the System.Collections.Generic.ICollection<T>.

IsReadOnly

Gets a value indicating whether the System.Collections.Generic.ICollection<T> is read-only.

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

□□□

bool

[true](#) if the System.Collections.Generic.ICollection<T> is read-only; otherwise, [false](#).

□□

Add(T)

Adds an item to the System.Collections.Generic.ICollection<T>.

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

□□

item T

The object to add to the System.Collections.Generic.ICollection<T>.

□□

NotSupportedException

The System.Collections.Generic.ICollection<T> is read-only.

Clear()

Removes all items from the System.Collections.Generic.ICollection<T>.

```
public void Clear()
```

□□

NotSupportedException

The System.Collections.Generic.ICollection<T> is read-only.

Contains(T)

Determines whether the System.Collections.Generic.ICollection<T> contains a specific value.

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

□□

item T

The object to locate in the System.Collections.Generic.ICollection<T>.

□□

bool

[true](#) if item is found in the System.Collections.Generic.ICollection<T>; otherwise, [false](#).

CopyTo(T[], int)

Copies the elements of the System.Collections.Generic.ICollection<T> to an System.Array, starting at a particular System.Array index.

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

□□

array T[]

The one-dimensional System.Array that is the destination of the elements copied from System.Collections.Generic.ICollection<T>. The System.Array must have zero-based indexing.

arrayIndex int

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

□□

ArgumentNullException

`array` is `null`.

ArgumentOutOfRangeException

`arrayIndex` is less than 0.

ArgumentException

The number of elements in the source `System.Collections.Generic.ICollection<T>` is greater than the available space from `arrayIndex` to the end of the destination `array`.

GetEnumerator()

Returns an enumerator that iterates through the collection.

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



`IEnumerator<T>`

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

Remove(T)

Removes the first occurrence of a specific object from the `System.Collections.Generic.ICollection<T>`.

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



`item` `T`

The object to remove from the `System.Collections.Generic.ICollection<T>`.



bool

[true](#) if `item` was successfully removed from the `System.Collections.Generic.ICollection<T>`; otherwise, [false](#). This method also returns [false](#) if `item` is not found in the original `System.Collections.Generic.ICollection<T>`.



NotSupportedException

The `System.Collections.Generic.ICollection<T>` is read-only.

GameResources □

□□□: [ProjectStack.Core](#)

□□□: ProjectStack.dll

```
public class GameResources
```

□□

object ← GameResources

□□

Count

Gets the number of elements contained in the System.Collections.Generic.ICollection<T>.

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

□□□

int

The number of elements contained in the System.Collections.Generic.ICollection<T>.

IsReadOnly

Gets a value indicating whether the System.Collections.Generic.ICollection<T> is read-only.

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

□□□

bool

[true](#) if the System.Collections.Generic.ICollection<T> is read-only; otherwise, [false](#).



Add(IGameResource)

Adds an item to the System.Collections.Generic.ICollection<T>.

```
public void Add(IGameResource item)
```



`item` [IGameResource](#)

The object to add to the System.Collections.Generic.ICollection<T>.



NotSupportedException

The System.Collections.Generic.ICollection<T> is read-only.

Clear()

Removes all items from the System.Collections.Generic.ICollection<T>.

```
public void Clear()
```



NotSupportedException

The System.Collections.Generic.ICollection<T> is read-only.

Contains(IGameResource)

Determines whether the System.Collections.Generic.ICollection<T> contains a specific value.

```
public bool Contains(IGameResource item)
```

□□

item [IGameResource](#)

The object to locate in the System.Collections.Generic.ICollection<T>.

□□

bool

[true](#) if **item** is found in the System.Collections.Generic.ICollection<T>; otherwise, [false](#)

.

CopyTo(IGameResource[], int)

Copies the elements of the System.Collections.Generic.ICollection<T> to an System.Array, starting at a particular System.Array index.

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

□□

array [IGameResource](#)[]

The one-dimensional System.Array that is the destination of the elements copied from System.Collections.Generic.ICollection<T>. The System.Array must have zero-based indexing.

arrayIndex int

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

□□

ArgumentNullException

array is [null](#).

ArgumentOutOfRangeException

arrayIndex is less than 0.

ArgumentException

The number of elements in the source System.Collections.Generic.ICollection<T> is greater than the available space from `arrayIndex` to the end of the destination `array`.

GetEnumerator()

Returns an enumerator that iterates through the collection.

```
public Ienumerator<IGameResource> GetEnumerator()
```



`Ienumerator<IGameResource>`

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

Remove(IGameResource)

Removes the first occurrence of a specific object from the System.Collections.Generic.ICollection<T>.

```
public bool Remove(IGameResource item)
```



`item IGameResource`

The object to remove from the System.Collections.Generic.ICollection<T>.



`bool`

`true` if `item` was successfully removed from the System.Collections.Generic.ICollection<T>; otherwise, `false`. This method also returns `false` if `item` is not found in the original System.Collections.Generic.ICollection<T>.



NotSupportedException

The System.Collections.Generic.ICollection<T> is read-only.

IGameResource

命名空间: [ProjectStack.Core](#)

文件: ProjectStack.dll

```
public interface IGameResource
```

Level

模块: [ProjectStack.Core](#)

文件: ProjectStack.dll

```
[ScriptPath("res://src/scripts/Core/Level.cs")]
public class Level : Node2D
```

类

object ← GodotObject ← Node ← CanvasItem ← Node2D ← Level

方法

Level()

```
public Level()
```

方法

AddAttribute(AttributeHolder)

```
public void AddAttribute(AttributeHolder attributeHolder)
```

参数

attributeHolder [AttributeHolder](#)

GetAttribute(AttributeHolder)

```
public AttributeHolder GetAttribute(AttributeHolder type)
```

参数

type [AttributeHolder](#)

□□

[AttributeHolder](#)

GetAttribute(ResourceLocation)

```
public AttributeHolder GetAttribute(ResourceLocation id)
```

□□

[id ResourceLocation](#)

□□

[AttributeHolder](#)

RemoveAttribute(ResourceLocation)

```
public void RemoveAttribute(ResourceLocation id)
```

□□

[id ResourceLocation](#)

RestoreGodotObjectData(GodotSerializationInfo)

Restores this instance's state after reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void RestoreGodotObjectData(GodotSerializationInfo info)
```

□□

[info GodotSerializationInfo](#)

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

Saves this instance's state to be restored when reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void SaveGodotObjectData(GodotSerializationInfo info)
```



info GodotSerializationInfo

Object used to save the data.

ServiceCollectionExtension □

□□□: [ProjectStack.Core](#)

□□□: ProjectStack.dll

```
public static class ServiceCollectionExtension
```

□□

```
object ← ServiceCollectionExtension
```

□□

AddKeylessRegistry<T>(IServiceCollection,
Action<KeylessRegistry<T>>?)

```
public static IServiceCollection AddKeylessRegistry<T>(this IServiceCollection  
services, Action<KeylessRegistry<T>>? configure = null)
```

□□

```
services IServiceCollection
```

```
configure Action<KeylessRegistry<T>>
```

□□

```
IServiceCollection
```

□□□□

T

AddRegistry<T>(IServiceCollection,
Action<Registry<T>>?)

```
public static IServiceCollection AddRegistry<T>(this IServiceCollection services,  
Action<Registry<T>>? configure = null)
```

□□

services IServiceCollection

configure Action<[Registry](#)<T>>

□□

IServiceCollection

□□□

T

ConfigureHoveredItemInfoDisplay(IServiceCollection)

```
public static HoveredItemInfoDisplayRegistrationHelper  
ConfigureHoveredItemInfoDisplay(this IServiceCollection services)
```

□□

services IServiceCollection

□□

[HoveredItemInfoDisplayRegistrationHelper](#)

RegisterCardMetas(IServiceCollection)

```
public static CardMetaRegistrationHelper RegisterCardMetas(this  
IServiceCollection services)
```

□□

```
services IServiceCollection
```



[CardMetaRegistrationHelper](#)

RegisterEditors(IServiceCollection)

```
public static EditorRegistrationHelper RegisterEditors(this  
IServiceCollection services)
```



```
services IServiceCollection
```



[EditorRegistrationHelper](#)

RegisterRecipes(IServiceCollection)

```
public static RecipeRegistrationHelper RegisterRecipes(this  
IServiceCollection services)
```



```
services IServiceCollection
```



[RecipeRegistrationHelper](#)

RegisterTextureLoader(IServiceCollection)

```
public static TextureLoader RegisterTextureLoader(this IServiceCollection services)
```

□□

services IServiceCollection

□□

[TextureLoader](#)

ProjectStack.Editor

□

[CardMetaEditor](#)

[CardMetaEditorModel](#)

[EditorRegistrationHelper](#)

[SingleTypeEditor](#)

Base class for all UI-related nodes. Godot.Control features a bounding rectangle that defines its extents, an anchor position relative to its parent control or the current viewport, and offsets relative to the anchor. The offsets update automatically when the node, any of its parents, or the screen size change.

For more information on Godot's UI system, anchors, offsets, and containers, see the related tutorials in the manual. To build flexible UIs, you'll need a mix of UI elements that inherit from Godot.Control and Godot.Container nodes.

User Interface nodes and input

Godot propagates input events via viewports. Each Godot.Viewport is responsible for propagating Godot.InputEvents to their child nodes. As the Godot.SceneTree.Root is a Godot.Window, this already happens automatically for all UI elements in your game.

Input events are propagated through the Godot.SceneTree from the root node to all child nodes by calling Godot.Node._Input(Godot.InputEvent). For UI elements specifically, it makes more sense to override the virtual method Godot.Control._GUILInput(Godot.InputEvent), which filters out unrelated input events, such as by checking z-order, Godot.Control.MouseFilter, focus, or if the event was inside of the control's bounding box.

Call Godot.Control.AcceptEvent() so no other node receives the event. Once you accept an input, it becomes handled so Godot.Node._UnhandledInput(Godot.InputEvent) will not process it.

Only one Godot.Control node can be in focus. Only the node in focus will receive events. To get the focus, call Godot.Control.GrabFocus(). Godot.Control nodes lose focus when another node grabs it, or if you hide the node in focus.

Sets Godot.Control.MouseFilter to Godot.Control.MouseFilterEnum.Ignore to tell a Godot.Control node to ignore mouse or touch events. You'll need it if you place an icon on top of a button.

Godot.Theme resources change the Control's appearance. If you change the Godot.Theme on a Godot.Control node, it affects all of its children. To override some of the theme's parameters, call one of the `add_theme_*_override` methods, like `Godot.Control.AddThemeFontOverride(Godot.StringName, Godot.Font)`. You can override the theme with the Inspector.

Note: Theme items are *not* `Godot.GodotObject` properties. This means you can't access their values using `Godot.GodotObject.Get(Godot.StringName)` and `Godot.GodotObject.Set(Godot.StringName, Godot.Variant)`. Instead, use the `get_theme_*` and `add_theme_*_override` methods provided by this class.

[SingleTypeEditor.TaskNotifier](#)

A wrapping class that can hold a `System.Threading.Tasks.Task` value.

[SingleTypeEditor.TaskNotifier<T>](#)

A wrapping class that can hold a `System.Threading.Tasks.Task<TResult>` value.



[IEditorModel](#)

[ISingleTypeEditorModel](#)

CardMetaEditor □

□□□: [ProjectStack.Editor](#)

□□□: ProjectStack.dll

```
[Meta(new Type[] { typeof(IAutoNode) })]  
[ScriptPath("res://src/scripts/Editor/CardMetaEditor.cs")]  
public class CardMetaEditor : SingleTypeEditor
```

□□

object ← GodotObject ← Node ← CanvasItem ← Control ← [SingleTypeEditor](#) ← CardMetaEditor

□□□

```
SingleTypeEditor.\_model , SingleTypeEditor.Model , SingleTypeEditor.PropertyChanged ,  
SingleTypeEditor.PropertyChanging ,  
SingleTypeEditor.OnPropertyChanged\(PropertyChangedEventArgs\) ,  
SingleTypeEditor.OnPropertyChanging\(PropertyChangingEventArgs\) ,  
SingleTypeEditor.OnPropertyChanged\(string\) ,  
SingleTypeEditor.OnPropertyChanging\(string\) ,  
SingleTypeEditor SetProperty<T>\(ref T, T, string\) ,  
SingleTypeEditor SetProperty<T>\(ref T, T, IEqualityComparer<T>, string\) ,  
SingleTypeEditor SetProperty<T>\(T, T, Action<T>, string\) ,  
SingleTypeEditor SetProperty<T>\(T, T, IEqualityComparer<T>, Action<T>, string\) ,  
SingleTypeEditor SetProperty<TModel, T>\(T, T, TModel, Action<TModel, T>, string\) ,  
SingleTypeEditor SetProperty<TModel, T>\(T, T, IEqualityComparer<T>, TModel, Action<TModel, T>, string\) ,  
SingleTypeEditor SetPropertyAndNotifyOnCompletion\(ref SingleTypeEditor.TaskNotifier, Task, string\) ,  
SingleTypeEditor SetPropertyAndNotifyOnCompletion\(ref SingleTypeEditor.TaskNotifier, Task, Action<Task>, string\) ,  
SingleTypeEditor SetPropertyAndNotifyOnCompletion<T>\(ref SingleTypeEditor.TaskNotifier<T>, Task<T>, string\) ,  
SingleTypeEditor SetPropertyAndNotifyOnCompletion<T>\(ref SingleTypeEditor.TaskNotifier<T>, Task<T>, Action<Task<T>>, string\)
```

□□

Metatype

Generated metatype information.

```
public IMetatype Metatype { get; }
```



IMetatype

MixinState

Arbitrary data that is shared between mixins. Mixins are free to store additional instance state in this blackboard.

```
public MixinBlackboard MixinState { get; }
```



MixinBlackboard

NameList

```
public ItemList NameList { get; }
```



ItemList



GetGodotClassPropertyValue(in godot_string_name, out godot_variant)

Get the value of a property contained in this class. This method is used by Godot to retrieve property values. Do not call or override this method.

```
protected override bool GetGodotClassPropertyValue(in godot_string_name name, out  
godot_variant value)
```

□□

name godot_string_name

Name of the property to get.

value godot_variant

Value of the property if it was found.

□□

bool

[true](#) if a property with the given name was found.

HasGodotClassMethod(**in** godot_string_name)

Check if the type contains a method with the given name. This method is used by Godot to check if a method exists before invoking it. Do not call or override this method.

```
protected override bool HasGodotClassMethod(in godot_string_name method)
```

□□

method godot_string_name

Name of the method to check for.

□□

bool

InvokeGodotClassMethod(**in** godot_string_name, Native VariantPtrArgs, **out** godot_variant)

Invokes the method with the given name, using the given arguments. This method is used by Godot to invoke methods from the engine side. Do not call or override this method.

```
protected override bool InvokeGodotClassMethod(in godot_string_name method,  
NativeVariantPtrArgs args, out godot_variant ret)
```

□□

method godot_string_name

Name of the method to invoke.

args NativeVariantPtrArgs

Arguments to use with the invoked method.

ret godot_variant

Value returned by the invoked method.

□□

bool

OnReady()

Notification received when the node is ready.

```
public void OnReady()
```

RestoreGodotObjectData(GodotSerializationInfo)

Restores this instance's state after reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void RestoreGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

Saves this instance's state to be restored when reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void SaveGodotObjectData(GodotSerializationInfo info)
```



info GodotSerializationInfo

Object used to save the data.

_Notification(int)

Called when the object receives a notification, which can be identified in **what** by comparing it with a constant. See also Godot.GodotObject.Notification(int, bool).

```
public override void _Notification(int what)
{
    if (what == NotificationPredelete)
    {
        GD.Print("Goodbye!");
    }
}
```

Note: The base Godot.GodotObject defines a few notifications (Godot.GodotObject.NotificationPostinitialize and Godot.GodotObject.NotificationPredelete). Inheriting classes such as Godot.Node define a lot more notifications, which are also received by this method.

```
public override void _Notification(int what)
```



`what int`

CardMetaEditorModel □

□□□: [ProjectStack.Editor](#)

□□□: ProjectStack.dll

```
public class CardMetaEditorModel : ISingleTypeEditorModel
```

□□

object ← CardMetaEditorModel

□□

[ISingleTypeEditorModel](#)

□□□□

CardMetaEditorModel(ObservableCollection<CardMeta>)

```
public CardMetaEditorModel(ObservableCollection<CardMeta> cardMetas)
```

□□

cardMetas ObservableCollection<[CardMeta](#)>

□□

CardMetas

```
public ObservableCollection<CardMeta> CardMetas { get; }
```

□□□

ObservableCollection<[CardMeta](#)>

ResourceIds

```
public IReadOnlyList<string> ResourceIds { get; }
```

□□□

IReadOnlyList<string>

SelectedResourceIdIndex

```
public int? SelectedResourceIdIndex { get; set; }
```

□□□

int?

EditorRegistrationHelper □

□□□: [ProjectStack.Editor](#)

□□□: ProjectStack.dll

```
public class EditorRegistrationHelper
```

□□

object ← EditorRegistrationHelper

□□□□

EditorRegistrationHelper(IServiceCollection)

```
public EditorRegistrationHelper(IServiceCollection services)
```

□□

services IServiceCollection

□□

Add(string, ISingleTypeEditorModel, PackedScene)

```
public EditorRegistrationHelper Add(string resourceType, ISingleTypeEditorModel model, PackedScene editorPrototype)
```

□□

resourceType string

model [ISingleTypeEditorModel](#)

editorPrototype PackedScene

□□

[EditorRegistrationHelper](#)

Add(string, Func<ISingleTypeEditorModel>,
PackedScene)

```
public EditorRegistrationHelper Add(string resourceType,  
Func<ISingleTypeEditorModel> modelFactory, PackedScene editorPrototype)
```

□□

resourceType string

modelFactory Func<[ISingleTypeEditorModel](#)>

editorPrototype PackedScene

□□

[EditorRegistrationHelper](#)

End()

```
public IServiceCollection End()
```

□□

IServiceCollection

IEditorModel

命名空间: [ProjectStack.Editor](#)

文件: ProjectStack.dll

```
public interface IEditorModel
```



ResourceTypes

```
IReadOnlyList<string> ResourceTypes { get; }
```



IReadOnlyList<string>

Selected ResourceType

```
string? Selected ResourceType { get; set; }
```



string



GetSingleTypeEditorModel(string)

```
ISingleTypeEditorModel? GetSingleTypeEditorModel(string resourceType)
```



`resourceType` string

□□

[ISingleTypeEditorModel](#)

ISingleTypeEditorModel □□

□□□: [ProjectStack.Editor](#)

□□□: ProjectStack.dll

```
public interface ISingleTypeEditorModel
```



ResourceIds

```
IReadOnlyList<string> ResourceIds { get; }
```



IReadOnlyList<string>

SelectedResourceIdIndex

```
int? SelectedResourceIdIndex { get; set; }
```



int?

SingleTypeEditor ▾

类别: [ProjectStack.Editor](#)

文件: ProjectStack.dll

Base class for all UI-related nodes. Godot.Control features a bounding rectangle that defines its extents, an anchor position relative to its parent control or the current viewport, and offsets relative to the anchor. The offsets update automatically when the node, any of its parents, or the screen size change.

For more information on Godot's UI system, anchors, offsets, and containers, see the related tutorials in the manual. To build flexible UIs, you'll need a mix of UI elements that inherit from Godot.Control and Godot.Container nodes.

User Interface nodes and input

Godot propagates input events via viewports. Each Godot.Viewport is responsible for propagating Godot.InputEvents to their child nodes. As the Godot.SceneTree.Root is a Godot.Window, this already happens automatically for all UI elements in your game.

Input events are propagated through the Godot.SceneTree from the root node to all child nodes by calling Godot.Node._Input(Godot.InputEvent). For UI elements specifically, it makes more sense to override the virtual method Godot.Control._GUILInput(Godot.InputEvent), which filters out unrelated input events, such as by checking z-order, Godot.Control.MouseFilter, focus, or if the event was inside of the control's bounding box.

Call Godot.Control.AcceptEvent() so no other node receives the event. Once you accept an input, it becomes handled so Godot.Node._UnhandledInput(Godot.InputEvent) will not process it.

Only one Godot.Control node can be in focus. Only the node in focus will receive events. To get the focus, call Godot.Control.GrabFocus(). Godot.Control nodes lose focus when another node grabs it, or if you hide the node in focus.

Sets Godot.Control.MouseFilter to Godot.Control.MouseFilterEnum.Ignore to tell a Godot.Control node to ignore mouse or touch events. You'll need it if you place an icon on top of a button.

Godot.Theme resources change the Control's appearance. If you change the Godot.Theme on a Godot.Control node, it affects all of its children. To override some of the theme's parameters, call one of the `add_theme_*_override` methods, like Godot.Control.AddTheme

`FontOverride(Godot.StringName, Godot.Font)`. You can override the theme with the Inspector.

Note: Theme items are *not* `Godot.GodotObject` properties. This means you can't access their values using `Godot.GodotObject.Get(Godot.StringName)` and `Godot.GodotObject.Set(Godot.StringName, Godot.Variant)`. Instead, use the `get_theme_*` and `add_theme_*_override` methods provided by this class.

```
[ObservableObject]
[ScriptPath("res://src/scripts/Editor/SingleTypeEditor.cs")]
public abstract class SingleTypeEditor : Control
```

object
object ← `GodotObject` ← `Node` ← `CanvasItem` ← `Control` ← `SingleTypeEditor`

Derived

[CardMetaEditor](#)

_model

```
[ObservableProperty]
protected ISingleTypeEditorModel? _model
```

ISingleTypeEditorModel

Model

```
public ISingleTypeEditorModel? Model { get; set; }
```

ISingleTypeEditorModel



OnPropertyChanged(PropertyChangedEventArgs)

Raises the [PropertyChanged](#) event.

```
protected virtual void OnPropertyChanged(PropertyChangedEventArgs e)
```



e `PropertyChangedEventArgs`

The input `System.ComponentModel.PropertyChangedEventArgs` instance.

OnPropertyChanged(string?)

Raises the [PropertyChanged](#) event.

```
protected void OnPropertyChanged(string? propertyName = null)
```



propertyName `string`

(optional) The name of the property that changed.

OnPropertyChanging(PropertyChangingEventArgs)

Raises the [PropertyChanging](#) event.

```
protected virtual void OnPropertyChanging(PropertyChangingEventArgs e)
```



e `PropertyChangingEventArgs`

The input System.ComponentModel.PropertyChangingEventArgs instance.

OnPropertyChanging(string?)

Raises the [PropertyChanging](#) event.

```
protected void OnPropertyChanging(string? propertyName = null)
```

□□

propertyName string

(optional) The name of the property that changed.

RestoreGodotObjectData(GodotSerializationInfo)

Restores this instance's state after reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void RestoreGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

Saves this instance's state to be restored when reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.ISerializationListener.

```
protected override void SaveGodotObjectData(GodotSerializationInfo info)
```

□□

info GodotSerializationInfo

Object used to save the data.

SetPropertyAndNotifyOnCompletion(ref TaskNotifier?, Task?, Action<Task?>, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. This method is just like [SetPropertyAndNotifyOnCompletion\(ref TaskNotifier?, Task?, string?\)](#), with the difference being an extra System.Action<T> parameter with a callback being invoked either immediately, if the new task has already completed or is [null](#), or upon completion.

```
protected bool SetPropertyAndNotifyOnCompletion(ref SingleTypeEditor.TaskNotifier?  
taskNotifier, Task? newValue, Action<Task?> callback, string? propertyName = null)
```

□□

taskNotifier [SingleTypeEditor.TaskNotifier](#)

The field notifier to modify.

newValue Task

The property's value after the change occurred.

callback Action<Task>

A callback to invoke to update the property value.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetPropertyAndNotifyOnCompletion(ref TaskNotifier?, Task?, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method will also monitor the new value of the property (a generic System.Threading.Tasks.Task) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that System.Threading.Tasks.Task or any of its properties. This method and its overload specifically rely on the [SingleTypeEditor.TaskNotifier](#) type, which needs to be used in the backing field for the target System.Threading.Tasks.Task property. The field doesn't need to be initialized, as this method will take care of doing that automatically. The [SingleTypeEditor.TaskNotifier](#) type also includes an implicit operator, so it can be assigned to any System.Threading.Tasks.Task instance directly. Here is a sample property declaration using this method:

```
private TaskNotifier myTask;

public Task MyTask
{
    get => myTask;
    private set => SetAndNotifyOnCompletion(ref myTask, value);
}

protected bool SetPropertyAndNotifyOnCompletion(ref SingleTypeEditor.TaskNotifier?
taskNotifier, Task? newValue, string? propertyName = null)
```



taskNotifier [SingleTypeEditor.TaskNotifier](#)

The field notifier to modify.

newValue Task

The property's value after the change occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same. The return value being [true](#) only indicates that the new value being assigned to [taskNotifier](#) is different than the previous one, and it does not mean the new System.Threading.Tasks.Task instance passed as argument is in any particular state.

SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>?, Task<T>?, Action<Task<T>?>, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. This method is just like [SetPropertyAndNotifyOnCompletion<T>\(ref TaskNotifier<T>?, Task<T>?, string?\)](#), with the difference being an extra System.Action<T> parameter with a callback being invoked either immediately, if the new task has already completed or is [null](#), or upon completion.

```
protected bool SetPropertyAndNotifyOnCompletion<T>(ref SingleTypeEditor.TaskNotifier<T>? taskNotifier, Task<T>? newValue, Action<Task<T>?> callback, string? propertyName = null)
```

□□

taskNotifier [SingleTypeEditor.TaskNotifier<T>](#)

The field notifier to modify.

newValue Task<T>

The property's value after the change occurred.

callback Action<Task<T>>

A callback to invoke to update the property value.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of result for the System.Threading.Tasks.Task<TResult> to set and monitor.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>?, Task<T>?, string?)

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the [PropertyChanging](#) event, updates the field and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method will also monitor the new value of the property (a generic System.Threading.Tasks.Task) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that System.Threading.Tasks.Task or any of its properties. This method and its overload specifically rely on the [SingleTypeEditor.TaskNotifier<T>](#) type, which needs to be used in the backing field for the target System.Threading.Tasks.Task property. The field doesn't need to be initialized, as this method will take care of doing that automatically. The [SingleTypeEditor.TaskNotifier<T>](#)

type also includes an implicit operator, so it can be assigned to any System.Threading.Tasks.Task instance directly. Here is a sample property declaration using this method:

```
private TaskNotifier<int> myTask;

public Task<int> MyTask
{
    get => myTask;
    private set => SetAndNotifyOnCompletion(ref myTask, value);
}

protected bool SetPropertyAndNotifyOnCompletion<T>(ref
SingleTypeEditor.TaskNotifier<T>? taskNotifier, Task<T>? newValue, string?
propertyName = null)
```

□□

taskNotifier [SingleTypeEditor.TaskNotifier<T>](#)

The field notifier to modify.

newValue Task<T>

The property's value after the change occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of result for the System.Threading.Tasks.Task<TResult> to set and monitor.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same. The return value being [true](#) only indicates that the new value being assigned to [taskNotifier](#) is different than the previous one, and it does not mean the new `System.Threading.Tasks.Task<TResult>` instance passed as argument is in any particular state.

SetProperty<T>(T, T, Action<T>, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. This overload is much less efficient than [SetProperty<T>\(ref T, T, string?\)](#) and it should only be used when the former is not viable (eg. when the target property being updated does not directly expose a backing field that can be passed by reference). For performance reasons, it is recommended to use a stateful callback if possible through the [SetProperty<TModel, T>\(T, T, TModel, Action<TModel, T>, string?\)](#) whenever possible instead of this overload, as that will allow the C# compiler to cache the input callback and reduce the memory allocations. More info on that overload are available in the related XML docs. This overload is here for completeness and in cases where that is not applicable.

```
protected bool SetProperty<T>(T oldValue, T newValue, Action<T> callback, string?  
propertyName = null)
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

callback Action<T>

A callback to invoke to update the property value.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□

T

The type of the property that changed.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetProperty<T>(T, T, IEqualityComparer<T>, Action<T>, string?)

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. See additional notes about this overload in [SetProperty<T>\(T, T, Action<T>, string?\)](#).

```
protected bool SetProperty<T>(T oldValue, T newValue, IEqualityComparer<T> comparer,  
Action<T> callback, string? propertyName = null)
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

comparer IEqualityComparer<T>

The System.Collections.Generic.IEqualityComparer<T> instance to use to compare the input values.

`callback Action<T>`

A callback to invoke to update the property value.

`propertyName` string

(optional) The name of the property that changed.

□□

`bool`

`true` if the property was changed, `false` otherwise.

□□□□

T

The type of the property that changed.

`SetProperty<T>(ref T, T, IEqualityComparer<T>, string?)`

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event. See additional notes about this overload in [`SetProperty<T>\(ref T, T, string?\)`](#).

```
protected bool SetProperty<T>(ref T field, T newValue, IEqualityComparer<T> comparer, string? propertyName = null)
```

□□

`field` T

The field storing the property's value.

`newValue` T

The property's value after the change occurred.

`comparer` IEqualityComparer<T>

The `System.Collections.Generic.IEqualityComparer<T>` instance to use to compare the input values.

`propertyName` `string`

(optional) The name of the property that changed.

□□

`bool`

`true` if the property was changed, `false` otherwise.

□□□□

`T`

The type of the property that changed.

`SetProperty<T>(ref T, T, string?)`

Compares the current and new values for a given property. If the value has changed, raises the [PropertyChanging](#) event, updates the property with the new value, then raises the [PropertyChanged](#) event.

```
protected bool SetProperty<T>(ref T field, T newValue, string? propertyName = null)
```

□□

`field` `T`

The field storing the property's value.

`newValue` `T`

The property's value after the change occurred.

`propertyName` `string`

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

T

The type of the property that changed.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

SetProperty<TModel, T>(T, T, IEqualityComparer<T>, TModel, Action<TModel, T>, string?)

Compares the current and new values for a given nested property. If the value has changed, raises the [PropertyChanging](#) event, updates the property and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method is used to relay properties from a wrapped model in the current instance. See additional notes about this overload in [SetProperty<TModel, T>\(T, T, TModel, Action<TModel, T>, string?\)](#).

```
protected bool SetProperty<TModel, T>(T oldValue, T newValue, IEqualityComparer<T> comparer, TModel model, Action<TModel, T> callback, string? propertyName = null)  
where TModel : class
```

□□

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

comparer IEqualityComparer<T>

The System.Collections.Generic.IEqualityComparer<T> instance to use to compare the input values.

model TModel

The model containing the property being updated.

callback Action<TModel, T>

The callback to invoke to set the target property value, if a change has occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

TModel

The type of model whose property (or field) to set.

T

The type of property (or field) to set.

SetProperty<TModel, T>(T, T, TModel, Action<TModel, T>, string?)

Compares the current and new values for a given nested property. If the value has changed, raises the [PropertyChanging](#) event, updates the property and then raises the [PropertyChanged](#) event. The behavior mirrors that of [SetProperty<T>\(ref T, T, string?\)](#), with the difference being that this method is used to relay properties from a wrapped model in the current instance. This type is useful when creating wrapping, bindable objects that operate over models that lack support for notification (eg. for CRUD operations). Suppose we have this model (eg. for a database row in a table):

```
public class Person
{
    public string Name { get; set; }
}
```

We can then use a property to wrap instances of this type into our observable model (which supports notifications), injecting the notification to the properties of that model, like so:

```
[ObservableObject]
public class BindablePerson
{
    public Model { get; }

    public BindablePerson(Person model)
    {
        Model = model;
    }

    public string Name
    {
        get => Model.Name;
        set => Set(Model.Name, value, Model, (model, name) => model.Name = name);
    }
}
```

This way we can then use the wrapping object in our application, and all those "proxy" properties will also raise notifications when changed. Note that this method is not meant to be a replacement for [SetProperty<T>\(ref T, T, string?\)](#), and it should only be used when relaying properties to a model that doesn't support notifications, and only if you can't implement notifications to that model directly (eg. by having it inherit from `ObservableObject`). The syntax relies on passing the target model and a stateless callback to allow the C# compiler to cache the function, which results in much better performance and no memory usage.

```
protected bool SetProperty<TModel, T>(T oldValue, T newValue, TModel model,
Action<TModel, T> callback, string? propertyName = null) where TModel : class
```



`oldValue` `T`

The current property value.

newValue T

The property's value after the change occurred.

model TModel

The model containing the property being updated.

callback Action<TModel, T>

The callback to invoke to set the target property value, if a change has occurred.

propertyName string

(optional) The name of the property that changed.

□□

bool

[true](#) if the property was changed, [false](#) otherwise.

□□□□

TModel

The type of model whose property (or field) to set.

T

The type of property (or field) to set.

□□

The [PropertyChanging](#) and [PropertyChanged](#) events are not raised if the current and new value for the target property are the same.

□□

PropertyChanged

Occurs when a property value changes.

```
public event PropertyChangedEventHandler? PropertyChanged
```

□□□□

PropertyChangedEventHandler

PropertyChanging

Occurs when a property value is changing.

```
public event PropertyChangingEventHandler? PropertyChanging
```

□□□□

PropertyChangingEventHandler

SingleTypeEditor.TaskNotifier □

□□□: [ProjectStack.Editor](#)

□□□: ProjectStack.dll

A wrapping class that can hold a System.Threading.Tasks.Task value.

```
protected sealed class SingleTypeEditor.TaskNotifier
```

□□

```
object ← SingleTypeEditor.TaskNotifier
```

□□□

implicit operator Task?(TaskNotifier?)

Unwraps the System.Threading.Tasks.Task value stored in the current instance.

```
public static implicit operator Task?(SingleTypeEditor.TaskNotifier? notifier)
```

□□

```
notifier SingleTypeEditor.TaskNotifier
```

The input [SingleTypeEditor.TaskNotifier<T>](#) instance.

□□

Task

SingleTypeEditor.TaskNotifier<T>

项目: [ProjectStack.Editor](#)

位置: ProjectStack.dll

A wrapping class that can hold a System.Threading.Tasks.Task<TResult> value.

```
protected sealed class SingleTypeEditor.TaskNotifier<T>
```

参数

T

The type of value for the wrapped System.Threading.Tasks.Task<TResult> instance.

方法

```
object < SingleTypeEditor.TaskNotifier<T>
```

参数

implicit operator Task<T>?(TaskNotifier<T>?)

Unwraps the System.Threading.Tasks.Task<TResult> value stored in the current instance.

```
public static implicit operator Task<T>?(SingleTypeEditor.TaskNotifier<T>? notifier)
```

参数

```
notifier SingleTypeEditor.TaskNotifier<T>
```

The input [SingleTypeEditor.TaskNotifier<T>](#) instance.

返回

```
Task<T>
```

ProjectStack.NamedTagsBaseOnJson

□

[FriendlyNtjObject](#)

□□

[INtjObject](#)

FriendlyNtjObject □

▪▪▪▪: [ProjectStack.NamedTagsBaseOnJson](#)

▪▪▪: ProjectStack.dll

```
public class FriendlyNtjObject
```

□□

object ← FriendlyNtjObject

□□

Ntj

```
public JsonObject Ntj { get; }
```

□□□

JsonObject

□□

AsJsonObject()

```
public JsonObject AsJsonObject()
```

□□

JsonObject

AsString()

```
public string AsString()
```

□□

string

Empty()

```
public static FriendlyNtjObject Empty()
```

□□

[FriendlyNtjObject](#)

FromJsonObject(JsonObject)

```
public static FriendlyNtjObject FromJsonObject(JsonObject n)
```

□□

n JsonObject

□□

[FriendlyNtjObject](#)

GetArray(string)

```
public JSONArray GetArray(string key)
```

□□

key string

□□

JSONArray

GetFriendlyNtjObject(string)

```
public FriendlyNtjObject GetFriendlyNtjObject(string key)
```

□□

key string

□□

[FriendlyNtjObject](#)

GetObject(string)

```
public JsonObject? GetObject(string key)
```

□□

key string

□□

JsonObject

GetResourceLocation(string)

```
public ResourceLocation GetResourceLocation(string key)
```

□□

key string

□□

[ResourceLocation](#)

Get<T>(string)

```
public T? Get<T>(string key)
```

□□

key string

□□

T

□□□□

T

PraseString(string)

```
public static FriendlyNtjObject PraseString(string str)
```

□□

str string

□□

[FriendlyNtjObject](#)

Write(string, bool)

```
public void Write(string key, bool value)
```

□□

key string

value bool

Write(string, double)

```
public void Write(string key, double value)
```

□□

key string

value double

Write(string, int)

```
public void Write(string key, int value)
```

□□

key string

value int

Write(string, float)

```
public void Write(string key, float value)
```

□□

key string

value float

Write(string, string)

```
public void Write(string key, string value)
```

□□

key string

value string

Write(string, JSONArray)

```
public void Write(string key, JSONArray value)
```

□□

key string

value JSONArray

Write(string, JsonObject)

```
public void Write(string key, JsonObject value)
```

□□

key string

value JsonObject

WriteFriendlyNtjObject(string, FriendlyNtjObject)

```
public void WriteFriendlyNtjObject(string key, FriendlyNtjObject value)
```

□□

key string

value [FriendlyNtjObject](#)

WriteResourceLocation(string, ResourceLocation)

```
public void WriteResourceLocation(string key, ResourceLocation value)
```

□□

key string

value [ResourceLocation](#)

IIntjObject

项目名: [ProjectStack.NamedTagsBaseOnJson](#)

文件名: ProjectStack.dll

```
public interface IIntjObject
```

类名

Ntj

```
JsonObject Ntj { get; }
```

方法名

JsonObject

ProjectStack.Registration

□

[KeylessRegistry<T>](#)

[Registry<T>](#)

KeylessRegistry<T>

项目: [ProjectStack.Registration](#)

文件: ProjectStack.dll

```
public class KeylessRegistry<T>
```

类

T

类

```
object ← KeylessRegistry<T>
```

方法

Count

Gets the number of elements in the collection.

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

方法

int

The number of elements in the collection.

this[int]

Gets the element at the specified index in the read-only list.

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

方法

`index` int

The zero-based index of the element to get.

□□□

T

The element at the specified index in the read-only list.

□□

GetEnumerator()

Returns an enumerator that iterates through the collection.

```
[MustDisposeResource]  
public IEnumarator<T> GetEnumerator()
```

□□

IEnumerator<T>

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

Register(T)

```
public KeylessRegistry<T> Register(T value)
```

□□

`value` T

□□

[KeylessRegistry<T>](#)

Registry<T>

位于: [ProjectStack.Registration](#)

在: ProjectStack.dll

```
public class Registry<T>
```

类

T

类

```
object ← Registry<T>
```

方法

Count

Gets the number of elements in the collection.

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

方法

int

The number of elements in the collection.

this[ResourceLocation]

Gets the element that has the specified key in the read-only dictionary.

```
public T this[ResourceLocation key] { get; }
```

方法

`key` [ResourceLocation](#)

The key to locate.



T

The element that has the specified key in the read-only dictionary.



ArgumentNullException

`key` is [null](#).

KeyNotFoundException

The property is retrieved and `key` is not found.

Keys

Gets an enumerable collection that contains the keys in the read-only dictionary.

```
public IEnumerable<ResourceLocation> Keys { get; }
```



[IEnumerable<ResourceLocation>](#)

An enumerable collection that contains the keys in the read-only dictionary.

Values

Gets an enumerable collection that contains the values in the read-only dictionary.

```
public IEnumerable<T> Values { get; }
```



IEnumerable<T>

An enumerable collection that contains the values in the read-only dictionary.



ContainsKey(ResourceLocation)

Determines whether the read-only dictionary contains an element that has the specified key.

```
public bool ContainsKey(ResourceLocation key)
```



key [ResourceLocation](#)

The key to locate.



bool

[true](#) if the read-only dictionary contains an element that has the specified key; otherwise, [false](#).



[ArgumentNullException](#)

key is [null](#).

GetEnumerator()

Returns an enumerator that iterates through the collection.

```
[MustDisposeResource]
```

```
public IEnumarator<KeyValuePair<ResourceLocation, T>> GetEnumerator()
```



IEnumerator<KeyValuePair<[ResourceLocation](#), T>>

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

Register(ResourceLocation, T)

```
public Registry<T> Register(ResourceLocation key, T value)
```



key [ResourceLocation](#)

value T



[Registry](#)<T>

TryGetValue(ResourceLocation, out T)

Gets the value that is associated with the specified key.

```
public bool TryGetValue(ResourceLocation key, out T value)
```



key [ResourceLocation](#)

The key to locate.

value T

When this method returns, the value associated with the specified key, if the key is found; otherwise, the default value for the type of the **value** parameter. This parameter is passed uninitialized.



bool

[true](#) if the object that implements the System.Collections.Generic.IReadOnlyDictionary<TKey, TValue> interface contains an element that has the specified key; otherwise, [false](#).

□□

ArgumentNullException

key is [null](#).

TryRegister(ResourceLocation, T)

public bool TryRegister(ResourceLocation key, T value)

□□

key [ResourceLocation](#)

value T

□□

bool

ProjectStack.Resource

□

[TextureLoader](#)

TextureLoader

资源: [ProjectStack.Resource](#)

位置: ProjectStack.dll

```
public class TextureLoader
```

类

```
object ← TextureLoader
```

方法

TextureLoader(IServiceCollection)

```
public TextureLoader(IServiceCollection services)
```

参数

```
services IServiceCollection
```

方法

AddTextureDirectory(string)

```
public TextureLoader AddTextureDirectory(string directoryPath)
```

参数

```
directoryPath string
```

方法

[TextureLoader](#)

End()

```
public IServiceCollection End()
```



IServiceCollection

ProjectStack.UserInterface

□

[HoveredItemInfoDisplay](#)

[HoveredItemInfoDisplayRegistrationHelper](#)

[HoveredItemInfoProvider](#)

HoveredItemInfoDisplay □

□□□: [ProjectStack.UserInterface](#)

□□□: ProjectStack.dll

```
public class HoveredItemInfoDisplay
```

□□

```
object ← HoveredItemInfoDisplay
```

□□□□

HoveredItemInfoDisplay(IServiceCollection)

```
public HoveredItemInfoDisplay(IServiceCollection services)
```

□□

```
services IServiceCollection
```

□□

HoveredItemInfoProviders

```
public List<HoveredItemInfoProvider> HoveredItemInfoProviders { get; }
```

□□□

```
List<HoveredItemInfoProvider>
```

HoveredItemInfoTexts

```
public IImmutableList<string> HoveredItemInfoTexts { get; }
```

☰

IImmutableList<string>

HoveredItems

```
public ObservableCollection<Node> HoveredItems { get; }
```

☰

ObservableCollection<Node>

☰

Update()

```
public void Update()
```

HoveredItemInfoDisplayRegistrationHelper

□

□□□: [ProjectStack.UserInterface](#)

□□□: ProjectStack.dll

```
public class HoveredItemInfoDisplayRegistrationHelper
```

□□

```
object ← HoveredItemInfoDisplayRegistrationHelper
```

□□□□

HoveredItemInfoDisplayRegistrationHelper(IServiceCollection)

```
public HoveredItemInfoDisplayRegistrationHelper(IServiceCollection services)
```

□□

```
services IServiceCollection
```

□□

End()

```
public IServiceCollection End()
```

□□

```
IServiceCollection
```

RegisterHoveredItemInfoProvider(HoveredItemInfoProvider)

```
public HoveredItemInfoDisplayRegistrationHelper  
RegisterHoveredItemInfoProvider(HoveredItemInfoProvider hoveredItemInfoProvider)
```

□□

hoveredItemInfoProvider [HoveredItemInfoProvider](#)

□□

[HoveredItemInfoDisplayRegistrationHelper](#)

RegisterHoveredItemInfoProvider(string, Predicate<HoveredItemInfoDisplay>, Func<HoveredItemInfoDisplay, string>)

```
public HoveredItemInfoDisplayRegistrationHelper  
RegisterHoveredItemInfoProvider(string providerName,  
Predicate<HoveredItemInfoDisplay> predicate, Func<HoveredItemInfoDisplay,  
string> displayTextProvider)
```

□□

providerName string

predicate Predicate<[HoveredItemInfoDisplay](#)>

displayTextProvider Func<[HoveredItemInfoDisplay](#), string>

□□

[HoveredItemInfoDisplayRegistrationHelper](#)

HoveredItemInfoProvider □

□□□: [ProjectStack.UserInterface](#)

□□□: ProjectStack.dll

```
public class HoveredItemInfoProvider
```

□□

```
object ← HoveredItemInfoProvider
```

□□□□

```
HoveredItemInfoProvider(string,  
Predicate<HoveredItemInfoDisplay>,  
Func<HoveredItemInfoDisplay, string>)
```

```
public HoveredItemInfoProvider(string providerName,  
Predicate<HoveredItemInfoDisplay> predicate, Func<HoveredItemInfoDisplay,  
string> displayTextProvider)
```

□□

```
providerName string
```

```
predicate Predicate<HoveredItemInfoDisplay>
```

```
displayTextProvider Func<HoveredItemInfoDisplay, string>
```

□□

ProviderName

```
public string ProviderName { get; }
```

□□□

string

□□

ProvideHoveredItemInfo(HoveredItemInfoDisplay)

```
public string? ProvideHoveredItemInfo(HoveredItemInfoDisplay hoveredItemInfoDisplay)
```

□□

hoveredItemInfoDisplay [HoveredItemInfoDisplay](#)

□□

string

ProjectStack.Util



[FileSystemHelper](#)

FileSystemHelper

命名空间: [ProjectStack.Util](#)

文件: ProjectStack.dll

```
public class FileSystemHelper
```

类

```
object ← FileSystemHelper
```

方法

Default

```
public static FileSystemHelper Default { get; }
```

方法

[FileSystemHelper](#)

方法

GetAllFilesInDirectory(string)

```
public IEnumerable<string> GetAllFilesInDirectory(string directoryPath)
```

参数

directoryPath string

返回值

IEnumerable<string>