

ProjectStack 〇〇〇〇



[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._GuiInput(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 [Task](#)[↗] value.

[ResourceEditor.TaskNotifier<T>](#)

A wrapping class that can hold a [Task<TResult>](#)[↗] value.

CardTest

ProjectStack

ProjectStack.dll

```
public class CardTest : TestClass
```

```
object ← TestClass ← CardTest
```

CardTest(Node)

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

```
testScene Node
```

Setup()

```
[Setup(26)]  
public void Setup()
```

SetupAll()

```
[SetupAll(19)]  
public void SetupAll()
```

TestBottomCards()

[Test(46)]

```
public void TestBottomCards()
```

TestDisconnectTopCard()

[Test(56)]

```
public void TestDisconnectTopCard()
```

TestTopCards()

[Test(36)]

```
public void TestTopCards()
```

Game

ProjectStack

ProjectStack.dll

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

[object](#) ← GodotObject ← Node ← CanvasItem ← Node2D ← Game

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 ImmutableList<IRecipe> Recipes { get; }
```

□□□

[ImmutableList](#) <[IRecipe](#)>

ServiceProvider

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

□□□

[IServiceProvider](#)

□□

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](#)

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.

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 [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



namespace: [ProjectStack](#)

assembly: ProjectStack.dll

```
public class Game.LoadingProgressChangedEventArgs : EventArgs
```

base

[object](#) ← [EventArgs](#) ← Game.LoadingProgressChangedEventArgs

constructors

LoadingProgressChangedEventArgs(float, string)

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

properties

progress [float](#)

currentTaskName [string](#)

events

CurrentTaskName

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

return type

[string](#)

Progress

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



[float](#)

Launcher

: [ProjectStack](#)

: ProjectStack.dll

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



[object](#)  ← GodotObject ← Node ← Launcher



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](#) 

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 [_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

ProjectStack

ProjectStack.dll

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

[object](#) ← GodotObject ← Node ← CanvasItem ← Node2D ← Main

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](#) 

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 [_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 □

□□□□: [ProjectStack](#)

□□□: 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._GuiInput(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 [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 [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 [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`

`bool`

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

`bool`

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 [Task](#)) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that [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 [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 [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 [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 [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 [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 [Task](#)) and will also raise the [Property Changed](#) again for the target property when it completes. This can be used to update bindings observing that [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 [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 [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 [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 [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

[bool](#)

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

T

T

The type of the property that changed.

bool

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)
```

bool

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

comparer [IEqualityComparer](#) <T>

The [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`

`bool`

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

`T`

`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 [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`

`bool`

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

`T`

`T`

The type of the property that changed.

`PropertyChanging` and `PropertyChanged`

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
```

`PropertyChanging` and `PropertyChanged`

oldValue T

The current property value.

newValue T

The property's value after the change occurred.

comparer [IEqualityComparer](#) <T>

The [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

[bool](#)

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

TModel

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 [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`

`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 [SetProcess\(bool\)](#).

Corresponds to the `Godot.Node.NotificationProcess` notification in [_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 [_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 PropertyChangedEventHandler? PropertyChanged
```



[PropertyChangingEventHandler](#)

ResourceEditor.TaskNotifier

ProjectStack

ProjectStack.dll

A wrapping class that can hold a [Task](#) value.

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

[object](#) ← ResourceEditor.TaskNotifier

implicit operator Task?(TaskNotifier?)

Unwraps the [Task](#) value stored in the current instance.

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

notifier [ResourceEditor.TaskNotifier](#)


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

[Task](#)

ResourceEditor.TaskNotifier<T>

: [ProjectStack](#)

: ProjectStack.dll

A wrapping class that can hold a [Task<TResult>](#)  value.

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



T


The type of value for the wrapped [Task<TResult>](#)  instance.



[object](#)  ← ResourceEditor.TaskNotifier<T>



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

Unwraps the [Task<TResult>](#)  value stored in the current instance.

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



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

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



[Task](#)  <T>

ProjectStack.Command



[CommandAdapter](#)

CommandAdapter

ProjectStack: [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



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, 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](#) 

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.

ProjectStack.Common □□□□



[ResourceLocation](#)

ResourceLocation

Package: [ProjectStack.Common](#)

Assembly: ProjectStack.dll

```
public record ResourceLocation
```

Object

[object](#) ← ResourceLocation

Constructor

ResourceLocation(string, string)

```
public ResourceLocation(string Namespace, string Path)
```

Properties

Namespace [string](#)

Path [string](#)

Static

EMPTY

```
public static ResourceLocation EMPTY { get; }
```

Methods

[ResourceLocation](#)

GetNamespace

```
public string GetNamespace { get; }
```

□□□

[string](#)

GetPath

```
public string GetPath { get; }
```

□□□

[string](#)

Namespace

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

□□□

[string](#)

Path

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

□□□

[string](#)

□□

DefaultNamespaceAndPath(string)

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

□□

path [string](#)

□□

[ResourceLocation](#)

FromNamespaceAndPath(string, string)

```
public static ResourceLocation FromNamespaceAndPath(string @namespace, string path)
```

□□

namespace [string](#)

path [string](#)

□□

[ResourceLocation](#)

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.

ProjectStack.Common.Card □□□□

□

[CardMeta](#)

[CardMetaRegistrationHelper](#)

[CardStack](#)

□□

[ICardStack](#)

CardMeta

📄: [ProjectStack.Common.Card](#)

📄: ProjectStack.dll

```
public record CardMeta
```

📄

[object](#)  ← CardMeta

📄📄📄


CardMeta(ResourceLocation, string, string, ResourceLocation)

```
public CardMeta(ResourceLocation Id, string Name, string Description, ResourceLocation Type)
```

📄

Id [ResourceLocation](#)

Name [string](#) 

Description [string](#) 

Type [ResourceLocation](#)

📄

Description

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

📄📄

[string](#)

Id

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

[ResourceLocation](#)

Name

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

[string](#)

Type

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

[ResourceLocation](#)

Create(string)

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

id [string](#)



[CardMeta](#)

CardMetaRegistrationHelper

□□□□: [ProjectStack.Common.Card](#)

ProjectStack.dll

```
public class CardMetaRegistrationHelper
```

11

```
object ↗ ← CardMetaRegistrationHelper
```

--	--	--	--

CardMetaRegistrationHelper(IServiceCollection)

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

11

services [ICollection](#)

11

Add(CardMeta)

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

11

cardMeta CardMeta

11

CardMetaRegistrationHelper

End()

```
public IServiceCollection End()
```



[IServiceCollection](#)

CardStack

namespace: [ProjectStack.Common.Card](#)

assembly: ProjectStack.dll

```
public record CardStack : ICardStack
```

class

[object](#) ← CardStack

class

[ICardStack](#)

constructor

CardStack(ImmutableList<Card>)

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

property

Cards [ImmutableList](#) <[Card](#)>

property

Cards

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

property

[ImmutableList](#) <[Card](#)>

ICardStack

□□□□: [ProjectStack.Common.Card](#)

ProjectStack.dll

```
public interface ICardStack
```

11

Cards

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

ImmutableList <Card>

ProjectStack.Common.Recipe □□□□

□

[AbstractRecipe<TRecipeInput>](#)

[CardMetaMatchRecipe](#)

[RecipeInput](#)

[RecipeOutput](#)

[RecipeRegistrationHelper](#)

[RecipeResult](#)

[ScriptRecipeInput](#)

[ScriptRecipeInput.ScriptContext](#)

[SimpleRecipe](#)

□□□□□□□□

[SimpleRecipe.Ingredient](#)

[SimpleRecipe.Product](#)

□□

[IRecipe](#)

[IRecipeInput](#)

AbstractRecipe<TRecipeInput>

namespace: [ProjectStack.Common.Recipe](#)

assembly: ProjectStack.dll

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

TRecipeInput



[object](#)  ← AbstractRecipe<TRecipeInput>

Assmble(TRecipeInput, JsonObject)

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

recipeInput TRecipeInput

ntj [JsonObject](#) 

[RecipeOutput](#)

Matches(TRecipeInput, JsonObject)

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

recipeInput TRecipeInput

ntj [JsonObject](#)

□□

[bool](#)

Type()

```
public abstract ResourceLocation Type()
```

□□

[ResourceLocation](#)

CardMetaMatchRecipe

namespace: [ProjectStack.Common.Recipe](#)

dll: ProjectStack.dll

```
public class CardMetaMatchRecipe : IRecipe
```

object

[object](#) ← CardMetaMatchRecipe

interface

[IRecipe](#)

constructor

CardMetaMatchRecipe(IList<CardMeta>, IList<CardMeta>, bool)

```
public CardMetaMatchRecipe(IList<CardMeta> requiredCardMetas, IList<CardMeta>  
producedCardMetas, bool isNeedOrder = false)
```

parameters

requiredCardMetas [IList](#) <[CardMeta](#)>

producedCardMetas [IList](#) <[CardMeta](#)>

isNeedOrder [bool](#)

property

CardViewCount

nullable integer

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

□□□

[uint](#)[↗]?

IsNeedOrder

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

□□□

[bool](#)[↗]

ProducedCardMetas

```
public IList<CardMeta> ProducedCardMetas { get; set; }
```

□□□

[IList](#)[↗] <[CardMeta](#)>

RequiredCardMetas

```
public IList<CardMeta> RequiredCardMetas { get; set; }
```

□□□

[IList](#)[↗] <[CardMeta](#)>

□□

Execute(ICardStack)

□□□□□□□

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

11

cardStack [ICardStack](#)

□□□□□□□□

11

RecipeResult

□□□□

IRecipe

namespace: [ProjectStack.Common.Recipe](#)

dll: ProjectStack.dll

```
public interface IRecipe
```

CardViewCount

CardViewCount

CardViewCount is null

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

uint?

[uint](#)?

Execute(ICardStack)

Execute(ICardStack)

RecipeResult

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

cardStack

[ICardStack](#)

RecipeResult

[RecipeResult](#)

[RecipeResult](#)

IRecipeInput []

[][]: [ProjectStack.Common.Recipe](#)

[][]: ProjectStack.dll

```
public interface IRecipeInput
```


RecipeInput

namespace: [ProjectStack.Common.Recipe](#)

assembly: ProjectStack.dll

```
public abstract record RecipeInput
```

base

[object](#) ← RecipeInput

Derived

[ScriptRecipeInput](#)

base

Assemble(Card)

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

base

card [Card](#)

base

[Card](#)

IsMatch(Card)

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

base

card [Card](#)



[bool](#) 

RecipeOutput

namespace: [ProjectStack.Common.Recipe](#)

assembly: ProjectStack.dll

```
public record RecipeOutput
```

class

[object](#) ← RecipeOutput

constructor

RecipeOutput(List<Card>, JsonObject)

```
public RecipeOutput(List<Card> Cards, JsonObject Ntj)
```

fields

Cards [List](#)<[Card](#)>

Ntj [JsonObject](#)

properties

Cards

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

getter

[List](#)<[Card](#)>

Ntj

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

□□□

[JsonObject](#)↗

RecipeRegistrationHelper

□□□□: [ProjectStack.Common.Recipe](#)

ProjectStack.dll

```
public class RecipeRegistrationHelper
```

11

```
object ↗ ← RecipeRegistrationHelper
```

--	--	--	--

RecipeRegistrationHelper(IServiceCollection)

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

11

services [ICollection](#)

11

Add(IRecipe)

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

11

recipe IRecipe

11

RecipeRegistrationHelper

End()

```
public IServiceCollection End()
```



[IServiceCollection](#)

RecipeResult

□□□□: [ProjectStack.Common.Recipe](#)

ProjectStack.dll

```
public record RecipeResult
```

11

```
object ↗ ← RecipeResult
```

--	--	--	--

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

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

11

IsMatch [bool](#)

ConsumedCards [IEnumerable](#) [<Card>](#)ProducedCards [IEnumerable](#) [<Card>](#)

11

ConsumedCards

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

1111

IEnumerable [↗](#) <Card>

IsMatch

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



[bool](#) 

ProducedCards

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



[IEnumerable](#)  [<Card>](#)

ScriptRecipeInput

namespace: [ProjectStack.Common.Recipe](#)

assembly: ProjectStack.dll

```
public record ScriptRecipeInput : RecipeInput
```

using

```
object <a href="#"></a> <a href="#"></a> <a href="#"></a> ScriptRecipeInput
```

ScriptRecipeInput

ScriptRecipeInput()

```
public ScriptRecipeInput()
```

using

AssemblyScript

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

using

```
string <a href="#"></a>
```

AssemblyScriptRunner

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

using

```
ScriptRunner <a href="#"></a> <a href="#"></a>
```

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
```

11

```
object  ← ScriptRecipeInput.ScriptContext
```

11

card

```
public Card card
```

Card

SimpleRecipe

Package: [ProjectStack.Common.Recipe](#)

File: ProjectStack.dll

Namespace

```
public record SimpleRecipe
```

Object

[object](#) ← SimpleRecipe

Constructor

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

Constructor

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

Properties

Id [ResourceLocation](#)

Property

Name [string](#)

Description [string](#)

Production [float](#)

Property

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

□□

Products [ImmutableList](#) <[SimpleRecipe.Product](#)>

□□

□□

Description

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

□□□

[string](#)

Id

□□□□□

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

□□□

[ResourceLocation](#)

Ingredients

□□

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

□□□

[ImmutableList](#) <[SimpleRecipe.Ingredient](#)>

Name

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

□□□

[string](#)[↗]

Production

□□□□□

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

□□□

[float](#)[↗]

Products

□□

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

□□□

[ImmutableList](#)[↗] <[SimpleRecipe.Product](#)>

□□

SatisfactionCheck(ImmutableList<CardMeta>)

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

□□

cards [ImmutableList](#) <[CardMeta](#)>

□□

[bool](#)

SimpleRecipe.Ingredient

namespace: [ProjectStack.Common.Recipe](#)

assembly: ProjectStack.dll

```
public record SimpleRecipe.Ingredient
```

class

[object](#) ← SimpleRecipe.Ingredient

constructor

Ingredient(ResourceLocation, int, bool)

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

properties

CardId [ResourceLocation](#)

Quantity [int](#)

Consumed [bool](#)

property

CardId

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

property

[ResourceLocation](#)

Consumed

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



[bool](#)

Quantity

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



[int](#)

SimpleRecipe.Product

namespace: [ProjectStack.Common.Recipe](#)

assembly: ProjectStack.dll

```
public record SimpleRecipe.Product
```

class

[object](#) ← SimpleRecipe.Product

constructor

Product(ResourceLocation, int)

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

fields

CardId [ResourceLocation](#)

Quantity [int](#)

properties

CardId

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

properties

[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 [Task](#)[↗] value.

[Card.TaskNotifier<T>](#)

A wrapping class that can hold a [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._GuiInput(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 [Task](#) value.

[InfoTab.TaskNotifier<T>](#)

A wrapping class that can hold a [Task<TResult>](#) value.

Card


 : [ProjectStack.Component](#)

 : 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
```

[object](#)  ← GodotObject ← Node ← CanvasItem ← Node2D ← Card

[IntjObject](#)

Card()

```
public Card()
```

ForceMotion

```
public bool ForceMotion
```

[bool](#) 

InMoveing

```
public bool InMoveing
```

☐☐☐

[bool](#) 

☐☐

BottomCard

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

☐☐☐

[Card](#)

BottomCards

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

☐☐☐

[ImmutableList](#)  [<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; }
```



[ImmutableList](#) [<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

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



[JsonObject](#)

OnDrag

```
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](#)

☐☐

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](#)

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](#) 

OnCardAreaEntered(Area2D)

```
public void OnCardAreaEntered(Area2D body)
```

□□

body Area2D

OnCardAreaExited(Area2D)

```
public void OnCardAreaExited(Area2D body)
```

□□

body Area2D

OnCardStackChanged()

□□[CardStackChanged](#)□□□

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

OnPropertyChanged(PropertyChangedEventArgs)

Raises the [PropertyChanged](#) event.

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

□□

e [PropertyChangedEventArgs](#)↗

The input [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 [PropertyChangingEventArgs](#)↗ instance.

OnPropertyChanging(string?)

Raises the [PropertyChanging](#) event.

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

□□

propertyName [string](#) 

(optional) The name of the property that changed.

OnReady()

Notification received when the node is ready.

```
public void OnReady()
```

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.

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 [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 [Task](#)) and will also raise the [Property Changed](#) again for the target property when it completes. This can be used to update bindings observing that [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 [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 [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 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 [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 [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 [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 [Task](#)) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that [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 [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 [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 [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 [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 [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 [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 [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

[bool](#)

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

TModel

[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`

`bool`

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

`TModel`

`TModel`

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

`T`

The type of property (or field) to set.

`bool`

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

UpdateAllZIndex()

```
public void UpdateAllZIndex()
```

UpdateRecipe()

```
public void UpdateRecipe()
```

UpdateZIndex()

```
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 [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 [SetProcess\(bool\)](#).

Corresponds to the `Godot.Node.NotificationProcess` notification in [_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 [_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

Source: [ProjectStack.Component](#)

File: ProjectStack.dll

A wrapping class that can hold a [Task](#) value.

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

[object](#) ← Card.TaskNotifier

implicit operator Task?(TaskNotifier?)

Unwraps the [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>

: [ProjectStack.Component](#)


: ProjectStack.dll

A wrapping class that can hold a [Task<TResult>](#) value.

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



T


The type of value for the wrapped [Task<TResult>](#) instance.



[object](#) ← Card.TaskNotifier<T>



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

Unwraps the [Task<TResult>](#) value stored in the current instance.

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



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

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



[Task](#) <T>

InfoTab

: [ProjectStack.Component](#)

: 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._GuiInput(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, 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](#)

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 [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 [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 [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 [Task](#)) and will also raise the [Property Changed](#) again for the target property when it completes. This can be used to update bindings observing that [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 [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 [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 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 [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 [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 [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 [Task](#)) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that [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 [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 [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 [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 [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 [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 [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 [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

[bool](#)

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

TModel

[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`

[bool](#)

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

`TModel`

`T`

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

`T`

The type of property (or field) to set.

`bool`

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 [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

InfoTab: [ProjectStack.Component](#)

InfoTab: ProjectStack.dll

A wrapping class that can hold a [Task](#) value.

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

InfoTab

[object](#) ← InfoTab.TaskNotifier

InfoTab

implicit operator Task?(TaskNotifier?)

Unwraps the [Task](#) value stored in the current instance.

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

InfoTab

notifier [InfoTab.TaskNotifier](#)

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


InfoTab

[Task](#)

InfoTab.TaskNotifier<T>

: [ProjectStack.Component](#)


: ProjectStack.dll

A wrapping class that can hold a [Task<TResult>](#)  value.

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

T


The type of value for the wrapped [Task<TResult>](#)  instance.



[object](#)  ← InfoTab.TaskNotifier<T>

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

Unwraps the [Task<TResult>](#)  value stored in the current instance.


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



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

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



[Task](#)  <T>

ProjectStack.Core □□□□



[CardMgr](#)

[ServiceCollectionExtension](#)

CardMgr

ProjectStack.Core

ProjectStack.dll

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

[object](#) ← GodotObject ← Node ← CardMgr

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
```

11

```
object ↗ ← ServiceCollectionExtension
```

11

ConfigureHoveredItemInfoDisplay(IServiceCollection)

```
public static HoveredItemInfoDisplayRegistrationHelper  
ConfigureHoveredItemInfoDisplay(this IServiceCollection services)
```

11

services [ICollection](#)

11

[HoveredItemInfoDisplayRegistrationHelper](#)

RegisterCardMetas(IServiceCollection)

```
public static CardMetaRegistrationHelper RegisterCardMetas(this
IServiceCollection services)
```

11

services [ICollection](#)

11

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._GuiInput(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 [Task](#) value.

[SingleTypeEditor.TaskNotifier<T>](#)

A wrapping class that can hold a [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 [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)
```



CardMetaEditorModel

ProjectStack: [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
```

11

```
object ↗ ← EditorRegistrationHelper
```

--	--	--	--

EditorRegistrationHelper(IServiceCollection)

```
public EditorRegistrationHelper(IServiceCollection services)
```

11

services [ICollection](#)

11

Add(string, ISingleTypeEditorModel, PackedScene)

```
public EditorRegistrationHelper Add(string resourceType, ISingleTypeEditorModel
model, PackedScene editorPrototype)
```

11

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

namespace: [ProjectStack.Editor](#)

assembly: ProjectStack.dll

```
public interface IEditorModel
```

ResourceTypes

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

[IReadOnlyList](#) <[string](#)>

SelectedResourceType

```
string? SelectedResourceType { get; set; }
```

[string](#)

GetSingleTypeEditorModel(string)

```
ISingleTypeEditorModel? GetSingleTypeEditorModel(string resourceType)
```

resourceType [string](#)[↗]



[ISingleTypeEditorModel](#)

ISingleTypeEditorModel

namespace: [ProjectStack.Editor](#)

assembly: ProjectStack.dll

```
public interface ISingleTypeEditorModel
```

ResourceIds

```
IReadOnlyList<string> ResourceIds { get; }
```

[IReadOnlyList](#) <[string](#)>

SelectedResourceIdIndex

```
int? SelectedResourceIdIndex { get; set; }
```

[int](#)?

SingleTypeEditor □

□□□□: [ProjectStack.Editor](#)

□□□: 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._GuiInput(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](#) ↗ ← 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 [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 [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 [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 [Task](#)) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that [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 [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 [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 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 [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 [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`

`bool`

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

`T`

`T`

The type of result for the [Task<TResult>](#) to set and monitor.

`PropertyChanging` and `PropertyChanged`

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 [Task](#)) and will also raise the [PropertyChanged](#) again for the target property when it completes. This can be used to update bindings observing that [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 [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 [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 [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 [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 [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 [IEqualityComparer](#)<T> instance to use to compare the input values.

propertyName [string](#)

(optional) The name of the property that changed.

bool

bool

true if the property was changed, false otherwise.

T

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

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

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 [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

bool

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

TModel

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 PropertyChangedEventHandler? PropertyChanged
```



[PropertyChangedEventHandler](#) 

SingleTypeEditor.TaskNotifier

□□□□: [ProjectStack.Editor](#)

ProjectStack.dll

A wrapping class that can hold a [Task](#) value.

```
protected sealed class SingleTypeEditor.TaskNotifier
```

11

[object](#) ← SingleTypeEditor.TaskNotifier

111

implicit operator Task?(TaskNotifier?)

Unwraps the [Task](#) value stored in the current instance.

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

11

notifier [SingleTypeEditor.TaskNotifier](#)

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


11

Task 

SingleTypeEditor.TaskNotifier<T>

: [ProjectStack.Editor](#)

: ProjectStack.dll

A wrapping class that can hold a [Task<TResult>](#)  value.

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

T


The type of value for the wrapped [Task<TResult>](#)  instance.



[object](#)  ← SingleTypeEditor.TaskNotifier<T>

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

Unwraps the [Task<TResult>](#)  value stored in the current instance.

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



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

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



[Task](#)  <T>

ProjectStack.NamedTagsBaseOnJson □□□□

□□

[IntjObject](#)

INtjObject

📄: [ProjectStack.NamedTagsBaseOnJson](#)

📄: ProjectStack.dll

```
public interface INtjObject
```

📄

Ntj

```
JsonObject Ntj { get; }
```

📄

[JsonObject](#)🔗

ProjectStack.Resource □□□□

□

[TextureLoader](#)

TextureLoader

□□□□: [ProjectStack.Resource](#)

ProjectStack.dll

```
public class TextureLoader
```

11

```
object ↗ ← TextureLoader
```

--	--	--	--

TextureLoader(IServiceCollection)

```
public TextureLoader(IServiceCollection services)
```

11

services [ICollection](#)

11

AddTextureDirectory(string)

```
public TextureLoader AddTextureDirectory(string directoryPath)
```

11

directoryPath [string](#)

11

TextureLoader

End()

```
public IServiceCollection End()
```



[IServiceCollection](#)

ProjectStack.UserInterface □□□□



[HoveredItemInfoDisplay](#)

[HoveredItemInfoDisplayRegistrationHelper](#)

[HoveredItemInfoProvider](#)

HoveredItemInfoDisplay

namespace: [ProjectStack.UserInterface](#)

assembly: 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 ImmutableList<string> HoveredItemInfoTexts { get; }
```

□□□

[ImmutableList](#) <[string](#)>

HoveredItems

```
public ObservableCollection<Node> HoveredItems { get; }
```

□□□

[ObservableCollection](#) <Node>

□□

Update()

```
public void Update()
```

HoveredItemInfoDisplayRegistrationHelper



namespace: [ProjectStack.UserInterface](#)

assembly: 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)
```

11

providerName [string](#)

predicate [Predicate](#) [<HoveredItemInfoDisplay>](#)

displayTextProvider [Func](#) <[HoveredItemInfoDisplay](#), [string](#)>

11

ProviderName

```
public string ProviderName { get; }
```

--	--	--

[string](#)



ProvideHoveredItemInfo(HoveredItemInfoDisplay)

```
public string? ProvideHoveredItemInfo(HoveredItemInfoDisplay hoveredItemInfoDisplay)
```



hoveredItemInfoDisplay [HoveredItemInfoDisplay](#)



[string](#)

ProjectStack.Util □□□□



[FileSystemHelper](#)

FileSystemHelper □

□□□□: [ProjectStack.Util](#)

ProjectStack.dll

```
public class FileSystemHelper
```

11

```
object ↗ ← FileSystemHelper
```

11

Default

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

101

FileSystemHelper

11

GetAllFilesInDirectory(string)

```
public IEnumerable<string> GetAllFilesInDirectory(string directoryPath)
```

11

directoryPath [string](#)

11

IEnumerable <string >