ProjectStack □□□□

□

CardTest

Game

Game.LoadingProgressChangedEventArgs

Launcher

Main

ResourceEditor

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

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

User Interface nodes and input

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

Input events are propagated through the Godot.SceneTree from the root node to all child nodes by calling Godot.Node._Input(Godot.InputEvent). For UI elements specifically, it makes more sense to override the virtual method Godot.Control._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 类

命名空间: ProjectStack

程序集: ProjectStack.dll

```
public class CardTest : TestClass
```

继承

object ← TestClass ← CardTest

构造函数

CardTest(Node)

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

参数

`testScene` Node

方法

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 类

命名空间: ProjectStack

程序集: ProjectStack.dll

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

继承

object ← GodotObject ← Node ← CanvasItem ← Node2D ← Game

属性

Default

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

属性值

Game

Metatype

Generated metatype information.

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

属性值

IMetatype

MixinState

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

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

□□□

MixinBlackboard

Recipes

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

□□□

IImmutableList<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.ISerialization Listener.

```
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

类

命名空间: ProjectStack

程序集: ProjectStack.dll

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

继承

object ← EventArgs ← Game.LoadingProgressChangedEventArgs

构造函数

LoadingProgressChangedEventArgs(float, string)

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

参数

`progress` float

`currentTaskName` string

属性

CurrentTaskName

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

属性值

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, NativeVariantPtrArgs, out godot_variant)

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

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

□□

`method` godot_string_name

Name of the method to invoke.

`args` NativeVariantPtrArgs

Arguments to use with the invoked method.

`ret` godot_variant

Value returned by the invoked method.

□□

bool

RestoreGodotObjectData(GodotSerializationInfo)

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

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

□□

`info` GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

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

```
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.Godot Object() 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, NativeVariantPtrArgs, out godot_variant)

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

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

□□

`method` godot_string_name

Name of the method to invoke.

`args` NativeVariantPtrArgs

Arguments to use with the invoked method.

`ret` godot_variant

Value returned by the invoked method.

□□

bool

RestoreGodotObjectData(GodotSerializationInfo)

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

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

□□

`info` GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

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

```
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.Godot Object() 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.Input Event), 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, NativeVariantPtrArgs, out godot_variant)

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

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

□□

`method` godot_string_name

Name of the method to invoke.

`args` NativeVariantPtrArgs

Arguments to use with the invoked method.

`ret` godot_variant

Value returned by the invoked method.

□□

bool

OnPropertyChanged(PropertyChangedEventArgs)

Raises the PropertyChanged event.

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

□□

`e` PropertyChangedEventArgs

The input 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.ISerialization Listener.

```
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

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 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)
```

参数

`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.

返回

bool

true if the property was changed, false otherwise.

类型参数

`T`

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

备注

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 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

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

true if the property was changed, false otherwise.

□□□□

`TModel`

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

`T`

The type of property (or field) to set.

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

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

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

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

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

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

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

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

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

□□

`oldValue` T

The current property value.

`newValue` T

The property's value after the change occurred.

`model` TModel

The model containing the property being updated.

`callback` Action<TModel, T>

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

`propertyName` string

(optional) The name of the property that changed.

□□

bool

true if the property was changed, false otherwise.

□□□□

`TModel`

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

`T`

The type of property (or field) to set.

□□

The PropertyChanging and PropertyChanged events are not raised if the current and new value for the target property are the same.

_Notification(int)

Called when the object receives a notification, which can be identified in `what` by comparing it with a constant. See also 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.Godot Object() may be used. See also Godot.Node._EnterTree().

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

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

□□

PropertyChanged

Occurs when a property value changes.

```
public event PropertyChangedEventHandler? PropertyChanged
```

□□□□

PropertyChangedEventHandler

PropertyChanging

Occurs when a property value is changing.

```
public event PropertyChangingEventHandler? PropertyChanging
```

□□□□

PropertyChangingEventHandler

ResourceEditor.TaskNotifier 类

命名空间: ProjectStack

程序集: ProjectStack.dll

A wrapping class that can hold a Task value.

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

继承

object ← ResourceEditor.TaskNotifier

运算符

implicit operator Task?(TaskNotifier?)

Unwraps the Task value stored in the current instance.

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

参数

`notifier` ResourceEditor.TaskNotifier

The input ResourceEditor.TaskNotifier<T> instance.

返回

Task

ResourceEditor.TaskNotifier<T> 类

命名空间: ProjectStack

程序集: ProjectStack.dll

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

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

类型参数

T

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

继承

object ← ResourceEditor.TaskNotifier<T>

运算符

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)
```

参数

`notifier` ResourceEditor.TaskNotifier<T>

The input ResourceEditor.TaskNotifier<T> instance.

返回

Task<T>

ProjectStack.Command □□□□

□

CommandAdapter

CommandAdapter 类

命名空间: ProjectStack.Command

程序集: ProjectStack.dll

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

继承

object ← GodotObject ← Node ← CommandAdapter

属性

Metatype

Generated metatype information.

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

属性值

IMetatype

MixinState

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

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

属性值

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.ISerialization Listener.

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

□□

`info` GodotSerializationInfo

Object used to save the data.

ProjectStack.Common □□□□

□

ResourceLocation

ResourceLocation 类

命名空间: ProjectStack.Common

程序集: ProjectStack.dll

```
public record ResourceLocation
```

继承

object ← ResourceLocation

构造函数

ResourceLocation(string, string)

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

参数

`Namespace` string

`Path` string

属性

EMPTY

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

属性值

ResourceLocation

GetNamespace

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

属性值

string

GetPath

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

属性值

string

Namespace

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

属性值

string

Path

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

属性值

string

方法

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 类

命名空间: ProjectStack.Common.Card

程序集: ProjectStack.dll

```
public record CardMeta
```

继承

object ← CardMeta

构造函数

CardMeta(ResourceLocation, string, string, ResourceLocation)

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

参数

`Id` ResourceLocation

`Name` string

`Description` string

`Type` ResourceLocation

属性

Description

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

属性值

string

Id

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

属性值

ResourceLocation

Name

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

属性值

string

Type

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

属性值

ResourceLocation

方法

Create(string)

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

参数

id string

□□

CardMeta

CardMetaRegistrationHelper 类

命名空间: ProjectStack.Common.Card

程序集: ProjectStack.dll

```
public class CardMetaRegistrationHelper
```

继承

object ← CardMetaRegistrationHelper

构造函数

CardMetaRegistrationHelper(IServiceCollection)

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

参数

`services` IServiceCollection

方法

Add(CardMeta)

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

参数

`cardMeta` CardMeta

返回

CardMetaRegistrationHelper

End()

```
public IServiceCollection End()
```

□□

IServiceCollection

CardStack 类

命名空间: ProjectStack.Common.Card

程序集: ProjectStack.dll

```
public record CardStack : ICardStack
```

继承

object ← CardStack

实现

ICardStack

构造函数

CardStack(IImmutableList<Card>)

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

参数

`Cards` IImmutableList<Card>

属性

Cards

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

属性值

IImmutableList<Card>

ICardStack 接口

命名空间: ProjectStack.Common.Card

程序集: ProjectStack.dll

```
public interface ICardStack
```

属性

Cards

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

属性值

IImmutableList<Card>

ProjectStack.Common.Recipe 命名空间

类

AbstractRecipe<TRecipeInput>

CardMetaMatchRecipe

RecipeInput

RecipeOutput

RecipeRegistrationHelper

RecipeResult

ScriptRecipeInput

ScriptRecipeInput.ScriptContext

SimpleRecipe

表示配方的数据结构

SimpleRecipe.Ingredient

SimpleRecipe.Product

接口

IRecipe

IRecipeInput

AbstractRecipe<TRecipeInput> 类

命名空间: ProjectStack.Common.Recipe

程序集: ProjectStack.dll

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

类型参数

`TRecipeInput`

继承

object ← AbstractRecipe<TRecipeInput>

方法

Assmble(TRecipeInput, JsonObject)

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

参数

`recipeInput` TRecipeInput

`ntj` JsonObject

返回

RecipeOutput

Matchs(TRecipeInput, JsonObject)

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

参数

`recipeInput` TRecipeInput

`ntj` JsonObject

□□

bool

Type()

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

□□

ResourceLocation

CardMetaMatchRecipe 类

命名空间: ProjectStack.Common.Recipe

程序集: ProjectStack.dll

```
public class CardMetaMatchRecipe : IRecipe
```

继承

object ← CardMetaMatchRecipe

实现

IRecipe

构造函数

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

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

参数

`requiredCardMetas` IList<CardMeta>

`producedCardMetas` IList<CardMeta>

`isNeedOrder` bool

属性

CardViewCount

配方需要检查的卡牌数量。为`null`时表示不确定。

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

属性值

uint?

IsNeedOrder

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

属性值

bool

ProducedCardMetas

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

属性值

IList<CardMeta>

RequiredCardMetas

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

属性值

IList<CardMeta>

方法

Execute(ICardStack)

计算配方结果。

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

参数

`cardStack` ICardStack

需要进行计算的卡堆

返回

RecipeResult

配方输出

IRecipe 接口

命名空间: ProjectStack.Common.Recipe

程序集: ProjectStack.dll

```
public interface IRecipe
```

属性

CardViewCount

配方需要检查的卡牌数量。为`null`时表示不确定。

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

属性值

uint?

方法

Execute(ICardStack)

计算配方结果。

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

参数

`cardStack` ICardStack

需要进行计算的卡堆

返回

RecipeResult

□□□□

IRecipeInput □□

□□□□: ProjectStack.Common.Recipe

□□□: ProjectStack.dll

```
public interface IRecipeInput
```

RecipeInput 类

命名空间: ProjectStack.Common.Recipe

程序集: ProjectStack.dll

```
public abstract record RecipeInput
```

继承

object ← RecipeInput

Derived

ScriptRecipeInput

方法

Assemble(Card)

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

参数

`card` Card

返回

Card

IsMatch(Card)

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

参数

`card` Card

□□

bool

RecipeOutput 类

命名空间: ProjectStack.Common.Recipe

程序集: ProjectStack.dll

```
public record RecipeOutput
```

继承

object ← RecipeOutput

构造函数

RecipeOutput(List<Card>, JsonObject)

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

参数

`Cards` List<Card>

`Ntj` JsonObject

属性

Cards

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

属性值

List<Card>

Ntj

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

□□□

JsonObject

RecipeRegistrationHelper 类

命名空间: ProjectStack.Common.Recipe

程序集: ProjectStack.dll

```
public class RecipeRegistrationHelper
```

继承

object ← RecipeRegistrationHelper

构造函数

RecipeRegistrationHelper(IServiceCollection)

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

参数

`services` IServiceCollection

方法

Add(IRecipe)

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

参数

`recipe` IRecipe

返回

RecipeRegistrationHelper

End()

```
public IServiceCollection End()
```

□□

IServiceCollection

RecipeResult 类

命名空间: ProjectStack.Common.Recipe

程序集: ProjectStack.dll

```
public record RecipeResult
```

继承

object ← RecipeResult

构造函数

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

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

参数

`IsMatch` bool

`ConsumedCards` IEnumerable<Card>

`ProducedCards` IEnumerable<Card>

属性

ConsumedCards

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

属性值

IEnumerable<Card>

IsMatch

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

□□□

bool

ProducedCards

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

□□□

IEnumerable<Card>

ScriptRecipeInput 类

命名空间: ProjectStack.Common.Recipe

程序集: ProjectStack.dll

```
public record ScriptRecipeInput : RecipeInput
```

继承

object ← RecipeInput ← ScriptRecipeInput

构造函数

ScriptRecipeInput()

```
public ScriptRecipeInput()
```

属性

AssembleScript

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

属性值

string

AssembleScriptRunner

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

属性值

ScriptRunner<Card>

MatchScript

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

属性值

string

MatchScriptRunner

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

属性值

ScriptRunner<bool>

方法

Assemble(Card)

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

参数

`card` Card

返回

Card

IsMatch(Card)

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

□□

`card` Card

□□

bool

ScriptRecipeInput.ScriptContext 类

命名空间: ProjectStack.Common.Recipe

程序集: ProjectStack.dll

```
public class ScriptRecipeInput.ScriptContext
```

继承

object ← ScriptRecipeInput.ScriptContext

字段

card

```
public Card card
```

字段值

Card

SimpleRecipe 类

命名空间: ProjectStack.Common.Recipe

程序集: ProjectStack.dll

表示配方的数据结构

```
public record SimpleRecipe
```

继承

object ← SimpleRecipe

构造函数

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

表示配方的数据结构

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

参数

`Id` ResourceLocation

配方标识符

`Name` string

`Description` string

`Production` float

生产力消耗

`Ingredients` IImmutableList<SimpleRecipe.Ingredient>

原料

`Products` IImmutableList<SimpleRecipe.Product>

产品

属性

Description

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

属性值

string

Id

配方标识符

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

属性值

ResourceLocation

Ingredients

原料

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

属性值

IImmutableList<SimpleRecipe.Ingredient>

Name

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

属性值

string

Production

生产力消耗

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

属性值

float

Products

产品

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

属性值

IImmutableList<SimpleRecipe.Product>

方法

SatisfactionCheck(IImmutableList<CardMeta>)

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

参数

`cards` IImmutableList<CardMeta>

□□

bool

SimpleRecipe.Ingredient 类

命名空间: ProjectStack.Common.Recipe

程序集: ProjectStack.dll

```
public record SimpleRecipe.Ingredient
```

继承

object ← SimpleRecipe.Ingredient

构造函数

Ingredient(ResourceLocation, int, bool)

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

参数

`CardId` ResourceLocation

`Quantity` int

`Consumed` bool

属性

CardId

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

属性值

ResourceLocation

Consumed

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

□□□

bool

Quantity

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

□□□

int

SimpleRecipe.Product 类

命名空间: ProjectStack.Common.Recipe

程序集: ProjectStack.dll

```
public record SimpleRecipe.Product
```

继承

object ← SimpleRecipe.Product

构造函数

Product(ResourceLocation, int)

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

参数

`CardId` ResourceLocation

`Quantity` int

属性

CardId

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

属性值

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 类

命名空间: ProjectStack.Component

程序集: 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
```

继承

object ← GodotObject ← Node ← CanvasItem ← Node2D ← Card

实现

INtjObject

构造函数

Card()

```
public Card()
```

字段

ForceMotion

是否强制移动。

```
public bool ForceMotion
```

字段值

bool

InMoveing

是否正在移动。

```
public bool InMoveing
```

字段值

bool

属性

BottomCard

卡牌下方的卡牌。

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

属性值

Card

BottomCards

卡牌下方所有卡牌。

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

属性值

IImmutableList<Card>

CardMeta

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

属性值

CardMeta

CardNameLabel

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

属性值

Label

CardStack

卡堆

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

属性值

ICardStack

CharacterBody

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

属性值

CharacterBody2D

CurrentStack

卡堆中的所有卡牌。

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

属性值

IImmutableList<Card>

IsRoot

是否是根卡牌。

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

属性值

bool

IsUppest

是否是顶部卡牌。

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

属性值

bool

Metatype

Generated metatype information.

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

属性值

IMetatype

MixinState

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

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

属性值

MixinBlackboard

Ntj

额外数据，以Json对象形式存储。

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

属性值

JsonObject

OnDrag

是否在拖拽状态。

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

属性值

bool

Panel

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

属性值

Control

RootCard

根卡牌。

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

属性值

Card

TargetPosition

目标移动位置。

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

属性值

Vector2

TextureRect

材质

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

属性值

TextureRect

TopCard

卡牌上方的卡牌。

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

属性值

Card

TopCards

卡牌上方所有卡牌。

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

属性值

IImmutableList<Card>

UppestCard

顶部卡牌。

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

属性值

Card

方法

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)
```

参数

`name` godot_string_name

Name of the property to get.

`value` godot_variant

Value of the property if it was found.

返回

bool

true if a property with the given name was found.

GetNearestSameCard(Card)

获取附近的相同卡牌。

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

参数

`card` Card

需要寻找的卡牌

返回

Card

结果

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

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)
```

参数

e PropertyChangingEventArgs

The input PropertyChangingEventArgs instance.

OnPropertyChanging(string?)

Raises the PropertyChanging event.

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

参数

propertyName string

(optional) The name of the property that changed.

OnReady()

初始化。

```
public void OnReady()
```

RefreshTexture()

更新材质。

```
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.ISerialization Listener.

```
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 => SetAndNotifyOnCompletion(ref myTask, value);
}
```

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

□□

`taskNotifier` Card.TaskNotifier

The field notifier to modify.

`newValue` Task

The property's value after the change occurred.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

true if the property was changed, false otherwise.

□□

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

true if the property was changed, false otherwise.

□□□□

`TModel`

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

`T`

The type of property (or field) to set.

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

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

```
public class Person
{
```

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

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

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

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

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

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

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

□□

`oldValue` T

The current property value.

`newValue` T

The property's value after the change occurred.

`model` TModel

The model containing the property being updated.

`callback` Action<TModel, T>

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

`propertyName` string

(optional) The name of the property that changed.

返回

bool

true if the property was changed, false otherwise.

类型参数

`TModel`

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

`T`

The type of property (or field) to set.

备注

The PropertyChanging and PropertyChanged events are not raised if the current and new value for the target property are the same.

UpdateAllZIndex()

更新卡堆Z顺序以避免闪烁。

```
public void UpdateAllZIndex()
```

UpdateRecipe()

□□□□□

```
public void UpdateRecipe()
```

UpdateZIndex()

□□Z□□

```
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)

□□□□

```
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.Godot Object() 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 类

命名空间: ProjectStack.Component

程序集: ProjectStack.dll

A wrapping class that can hold a Task value.

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

继承

object ← Card.TaskNotifier

运算符

implicit operator Task?(TaskNotifier?)

Unwraps the Task value stored in the current instance.

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

参数

`notifier` Card.TaskNotifier

The input Card.TaskNotifier<T> instance.

返回

Task

Card.TaskNotifier<T> 类

命名空间: ProjectStack.Component

程序集: ProjectStack.dll

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

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

类型参数

T

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

继承

object ← Card.TaskNotifier<T>

运算符

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)
```

参数

notifier Card.TaskNotifier<T>

The input Card.TaskNotifier<T> instance.

返回

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.IsPhysicsProcessing() 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.ISerialization Listener.

```
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 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 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 => SetAndNotifyOnCompletion(ref myTask, value);
}
```

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

□□

`taskNotifier` InfoTab.TaskNotifier

The field notifier to modify.

`newValue` Task

The property's value after the change occurred.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

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

□□

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

true if the property was changed, false otherwise.

□□□□

`TModel`

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

`T`

The type of property (or field) to set.

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

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

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

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

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

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

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

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

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

□□

`oldValue` T

The current property value.

`newValue` T

The property's value after the change occurred.

`model` TModel

The model containing the property being updated.

`callback` Action<TModel, T>

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

`propertyName` string

(optional) The name of the property that changed.

□□

bool

true if the property was changed, false otherwise.

□□□□

`TModel`

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

`T`

The type of property (or field) to set.

□□

The PropertyChanging and PropertyChanged events are not raised if the current and new value for the target property are the same.

_Notification(int)

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

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

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

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

参数

what int

事件

PropertyChanged

Occurs when a property value changes.

```
public event PropertyChangedEventHandler? PropertyChanged
```

事件类型

PropertyChangedEventHandler

PropertyChanging

Occurs when a property value is changing.

```
public event PropertyChangingEventHandler? PropertyChanging
```

事件类型

PropertyChangingEventHandler

InfoTab.TaskNotifier 类

命名空间: ProjectStack.Component

程序集: ProjectStack.dll

A wrapping class that can hold a Task value.

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

继承

object ← InfoTab.TaskNotifier

运算符

implicit operator Task?(TaskNotifier?)

Unwraps the Task value stored in the current instance.

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

参数

`notifier` InfoTab.TaskNotifier

The input InfoTab.TaskNotifier<T> instance.

返回

Task

InfoTab.TaskNotifier<T> 类

命名空间: ProjectStack.Component

程序集: ProjectStack.dll

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

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

类型参数

T

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

继承

object ← InfoTab.TaskNotifier<T>

运算符

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)
```

参数

notifier InfoTab.TaskNotifier<T>

The input InfoTab.TaskNotifier<T> instance.

返回

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.ISerialization Listener.

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

□□

`info` GodotSerializationInfo

Object used to save the data.

ServiceCollectionExtension 类

命名空间: ProjectStack.Core

程序集: ProjectStack.dll

```
public static class ServiceCollectionExtension
```

继承

object ← ServiceCollectionExtension

方法

ConfigureHoveredItemInfoDisplay(IServiceCollection)

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

参数

`services` IServiceCollection

返回

HoveredItemInfoDisplayRegistrationHelper

RegisterCardMetas(IServiceCollection)

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

参数

`services` IServiceCollection

返回

CardMetaRegistrationHelper

RegisterEditors(IServiceCollection)

```
public static EditorRegistrationHelper RegisterEditors(this
IServiceCollection services)
```

□□

`services` IServiceCollection

□□

EditorRegistrationHelper

RegisterRecipes(IServiceCollection)

```
public static RecipeRegistrationHelper RegisterRecipes(this
IServiceCollection services)
```

□□

`services` IServiceCollection

□□

RecipeRegistrationHelper

RegisterTextureLoader(IServiceCollection)

```
public static TextureLoader RegisterTextureLoader(this IServiceCollection services)
```

□□

`services` IServiceCollection

□□

TextureLoader

ProjectStack.Editor □□□□

□

CardMetaEditor

CardMetaEditorModel

EditorRegistrationHelper

SingleTypeEditor

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

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

User Interface nodes and input

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

Input events are propagated through the Godot.SceneTree from the root node to all child nodes by calling Godot.Node._Input(Godot.InputEvent). For UI elements specifically, it makes more sense to override the virtual method Godot.Control._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.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) , SingleTypeEditor.Model

□□

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.ISerialization Listener.

```
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)
```

□□

```
what int
```

CardMetaEditorModel 类

命名空间: ProjectStack.Editor

程序集: ProjectStack.dll

```
public class CardMetaEditorModel : ISingleTypeEditorModel
```

继承

object ← CardMetaEditorModel

实现

ISingleTypeEditorModel

构造函数

CardMetaEditorModel(ObservableCollection<CardMeta>)

```
public CardMetaEditorModel(ObservableCollection<CardMeta> cardMetas)
```

参数

`cardMetas` ObservableCollection<CardMeta>

属性

CardMetas

```
public ObservableCollection<CardMeta> CardMetas { get; }
```

属性值

ObservableCollection<CardMeta>

ResourceIds

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

□□□

IReadOnlyList<string>

SelectedResourceIdIndex

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

□□□

int?

EditorRegistrationHelper 类

命名空间: ProjectStack.Editor

程序集: ProjectStack.dll

```
public class EditorRegistrationHelper
```

继承

object ← EditorRegistrationHelper

构造函数

EditorRegistrationHelper(IServiceCollection)

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

参数

`services` IServiceCollection

方法

Add(string, ISingleTypeEditorModel, PackedScene)

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

参数

`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 接口

命名空间: ProjectStack.Editor

程序集: ProjectStack.dll

```
public interface IEditorModel
```

属性

ResourceTypes

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

属性值

IReadOnlyList<string>

SelectedResourceType

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

属性值

string

方法

GetSingleTypeEditorModel(string)

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

参数

`resourceType` string

□□

ISingleTypeEditorModel

ISingleTypeEditorModel 接口

命名空间: ProjectStack.Editor

程序集: ProjectStack.dll

```
public interface ISingleTypeEditorModel
```

属性

ResourceIds

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

属性值

IReadOnlyList<string>

SelectedResourceIdIndex

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

属性值

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.ISerialization Listener.

```
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 => SetAndNotifyOnCompletion(ref myTask, value);
}
```

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

□□

`taskNotifier` SingleTypeEditor.TaskNotifier

The field notifier to modify.

`newValue` Task

The property's value after the change occurred.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

true if the property was changed, false otherwise.

□□

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

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 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

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

true if the property was changed, false otherwise.

□□□□

`TModel`

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

`T`

The type of property (or field) to set.

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

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

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

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

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

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

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

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

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

□□

`oldValue` T

The current property value.

`newValue` T

The property's value after the change occurred.

`model` TModel

The model containing the property being updated.

`callback` Action<TModel, T>

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

`propertyName` string

(optional) The name of the property that changed.

□□

bool

true if the property was changed, false otherwise.

□□□□

`TModel`

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

`T`

The type of property (or field) to set.

□□

The PropertyChanging and PropertyChanged events are not raised if the current and new value for the target property are the same.

□□

PropertyChanged

Occurs when a property value changes.

```
public event PropertyChangedEventHandler? PropertyChanged
```

□□□□

PropertyChangedEventHandler

PropertyChanging

Occurs when a property value is changing.

```
public event PropertyChangingEventHandler? PropertyChanging
```

□□□□

PropertyChangingEventHandler

SingleTypeEditor.TaskNotifier 类

命名空间: ProjectStack.Editor

程序集: ProjectStack.dll

A wrapping class that can hold a Task value.

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

继承

object ← SingleTypeEditor.TaskNotifier

运算符

implicit operator Task?(TaskNotifier?)

Unwraps the Task value stored in the current instance.

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

参数

`notifier` SingleTypeEditor.TaskNotifier

The input SingleTypeEditor.TaskNotifier<T> instance.

返回

Task

SingleTypeEditor.TaskNotifier<T> 类

命名空间: ProjectStack.Editor

程序集: ProjectStack.dll

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

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

类型参数

T

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

继承

object ← SingleTypeEditor.TaskNotifier<T>

运算符

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)
```

参数

`notifier` SingleTypeEditor.TaskNotifier<T>

The input SingleTypeEditor.TaskNotifier<T> instance.

返回

Task<T>

ProjectStack.NamedTagsBaseOnJson □□□□□□

INtjObject

INtjObject 接口

命名空间: ProjectStack.NamedTagsBaseOnJson

程序集: ProjectStack.dll

```
public interface INtjObject
```

属性

Ntj

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

属性值

JsonObject

ProjectStack.Resource □□□□

□

TextureLoader

TextureLoader 类

命名空间: ProjectStack.Resource

程序集: ProjectStack.dll

```
public class TextureLoader
```

继承

object ← TextureLoader

构造函数

TextureLoader(IServiceCollection)

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

参数

`services` IServiceCollection

方法

AddTextureDirectory(string)

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

参数

`directoryPath` string

返回

TextureLoader

End()

```
public IServiceCollection End()
```

□□

IServiceCollection

ProjectStack.UserInterface □□□□

□

HoveredItemInfoDisplay

HoveredItemInfoDisplayRegistrationHelper

HoveredItemInfoProvider

HoveredItemInfoDisplay 类

命名空间: ProjectStack.UserInterface

程序集: ProjectStack.dll

```
public class HoveredItemInfoDisplay
```

继承

object ← HoveredItemInfoDisplay

构造函数

HoveredItemInfoDisplay(IServiceCollection)

```
public HoveredItemInfoDisplay(IServiceCollection services)
```

参数

`services` IServiceCollection

属性

HoveredItemInfoProviders

```
public List<HoveredItemInfoProvider> HoveredItemInfoProviders { get; }
```

属性值

List<HoveredItemInfoProvider>

HoveredItemInfoTexts

```
public IImmutableList<string> HoveredItemInfoTexts { get; }
```

□□□

IImmutableList<string>

HoveredItems

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

□□□

ObservableCollection<Node>

□□

Update()

```
public void Update()
```

HoveredItemInfoDisplayRegistrationHelper 类

命名空间: ProjectStack.UserInterface

程序集: ProjectStack.dll

```
public class HoveredItemInfoDisplayRegistrationHelper
```

继承

object ← HoveredItemInfoDisplayRegistrationHelper

构造函数

HoveredItemInfoDisplayRegistrationHelper(IServiceCollection)

```
public HoveredItemInfoDisplayRegistrationHelper(IServiceCollection services)
```

参数

`services` IServiceCollection

方法

End()

```
public IServiceCollection End()
```

返回

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 类

命名空间: ProjectStack.UserInterface

程序集: ProjectStack.dll

```
public class HoveredItemInfoProvider
```

继承

object ← HoveredItemInfoProvider

构造函数

HoveredItemInfoProvider(string, Predicate<HoveredItemInfoDisplay>, Func<HoveredItemInfoDisplay, string>)

```
public HoveredItemInfoProvider(string providerName,
Predicate<HoveredItemInfoDisplay> predicate, Func<HoveredItemInfoDisplay,
string> displayTextProvider)
```

参数

`providerName` string

`predicate` Predicate<HoveredItemInfoDisplay>

`displayTextProvider` Func<HoveredItemInfoDisplay, string>

属性

ProviderName

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

属性值

string

□□

ProvideHoveredItemInfo(HoveredItemInfoDisplay)

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

□□

`hoveredItemInfoDisplay` HoveredItemInfoDisplay

□□

string

ProjectStack.Util □□□□

□

FileSystemHelper

FileSystemHelper 类

命名空间: ProjectStack.Util

程序集: ProjectStack.dll

```
public class FileSystemHelper
```

继承

object ← FileSystemHelper

属性

Default

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

属性值

FileSystemHelper

方法

GetAllFilesInDirectory(string)

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

参数

`directoryPath` string

返回

IEnumerable<string>