ProjectStack [][[]

П

<u>CardTest</u>

<u>Game</u>

<u>Launcher</u>

<u>Main</u>

CardTest [] □□□: ProjectStack □□□: ProjectStack.dll public class CardTest : TestClass CardTest(Node) public CardTest(Node testScene) testScene Node Setup() [Setup(26)] public void Setup() SetupAll() [SetupAll(19)] public void SetupAll()

TestBottomCards()

```
[Test(46)]
public void TestBottomCards()
```

TestDisconnectTopCard()

```
[Test(56)]
public void TestDisconnectTopCard()
```

TestTopCards()

```
[Test(36)]
public void TestTopCards()
```

Game []

```
□□□: ProjectStack
□□□: ProjectStack.dll
 [Meta(new Type[] { typeof(IAutoNode) })]
 [ScriptPath("res://src/Game.cs")]
 public class Game : Node2D
<u>object</u> ♂ ← GodotObject ← Node ← CanvasItem ← Node2D ← Game
Default
 public static Game Default { get; }
Game
```

Metatype

Generated metatype information.

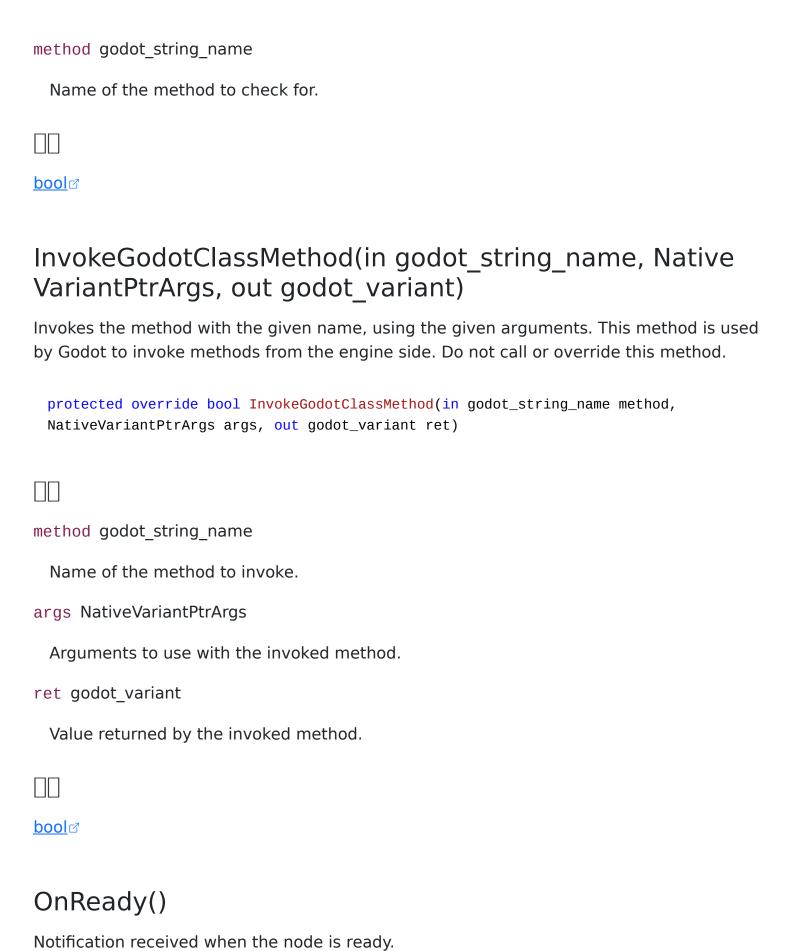
```
public IMetatype Metatype { get; }
```

IMetatype

MixinState

state in this blackboard.
<pre>public MixinBlackboard MixinState { get; }</pre>
MixinBlackboard
Recipes
<pre>public IImmutableList<irecipe> Recipes { get; }</irecipe></pre>
<u>IlmmutableList</u> ✓ < <u>IRecipe</u> >
ServiceProvider
<pre>public IServiceProvider ServiceProvider { get; }</pre>
<u>IServiceProvider</u>
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.
<pre>protected override bool HasGodotClassMethod(in godot_string_name method)</pre>

Arbitrary data that is shared between mixins. Mixins are free to store additional instance



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. IS erialization Listener.

protected override void RestoreGodotObjectData(GodotSerializationInfo info)

 \prod

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. IS erialization 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()



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 <u>int</u>♂

Launcher □□: ProjectStack □□: ProjectStack.dll [ScriptPath("res://src/Launcher.cs")] public class Launcher: Node □□ object ← GodotObject ← Node ← Launcher □□ HasGodotClassMethod(in godot_string_name)

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

protected override bool HasGodotClassMethod(in godot_string_name method)

method godot string name

Name of the method to check for.

bool₫

InvokeGodotClassMethod(in godot_string_name, Native VariantPtrArgs, out godot_variant)

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

<pre>protected override bool InvokeGodotClassMethod(in godot_string_name method, NativeVariantPtrArgs args, out godot_variant ret)</pre>
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.
<u>bool</u> ♂
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. ISerialization Listener.

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. IS erialization Listener.

protected override void SaveGodotObjectData(GodotSerializationInfo info)



info GodotSerializationInfo

Object used to save the data.

_Ready()

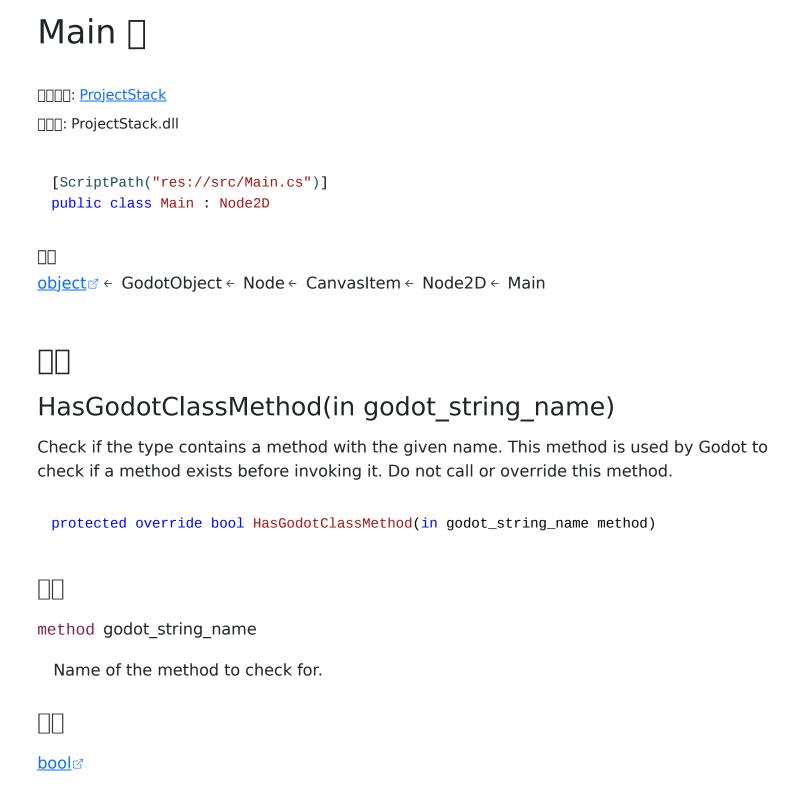
Called when the node is "ready", i.e. when both the node and its children have entered the scene tree. If the node has children, their Godot.Node._Ready() callbacks get triggered first, and the parent node will receive the ready notification afterwards.

Corresponds to the Godot.Node.NotificationReady notification in <u>Notification(int)</u>. See also the <code>@onready</code> 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()



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.

<pre>protected override bool InvokeGodotClassMethod(in godot_string_name method, NativeVariantPtrArgs args, out godot_variant ret)</pre>
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.
<u>bool</u> ♂
RestoreGodotObiectData(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. ISerialization Listener.

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. IS erialization 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 <u>Notification(int)</u>. See also the <code>@onready</code> 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()

ProjectStack.Command 🛛 🖺	
--------------------------	--

Ш

CommandAdapter

CommandAdapter []

```
ProjectStack.Command
□□□: ProjectStack.dll
 [Meta(new Type[] { typeof(IAutoNode) })]
 [ScriptPath("res://src/scripts/Command/CommandAdapter.cs")]
 public class CommandAdapter : Node
Metatype
Generated metatype information.
 public IMetatype Metatype { get; }
IMetatype
```

MixinState

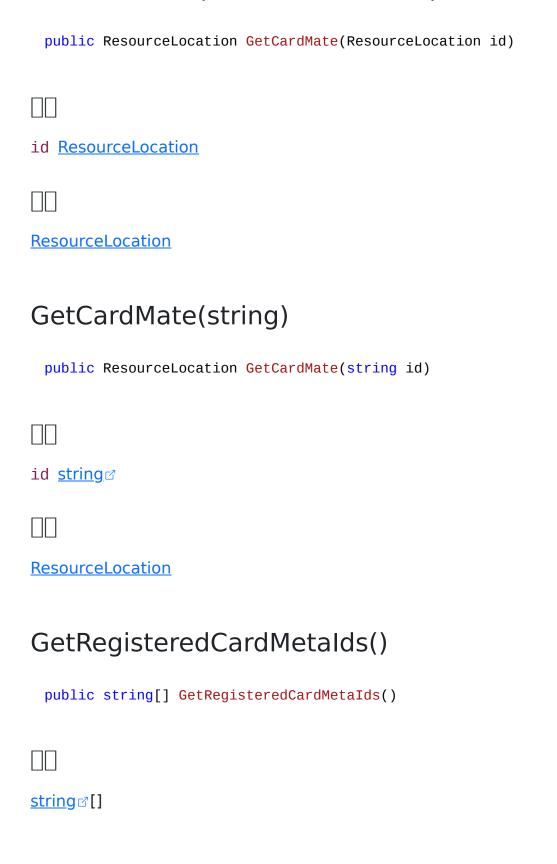
Arbitrary data that is shared between mixins. Mixins are free to store additional instance state in this blackboard.

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

MixinBlackboard

CreateNewCard(ResourceLocation)
<pre>public void CreateNewCard(ResourceLocation id)</pre>
id <u>ResourceLocation</u>
CreateNewCard(string)
<pre>public void CreateNewCard(string id)</pre>
id <u>string</u> ♂
CreateNewCardStack(List <resourcelocation>)</resourcelocation>
<pre>public void CreateNewCardStack(List<resourcelocation> ids)</resourcelocation></pre>
ids <u>List</u> d< <u>ResourceLocation</u> >
CreateNewCardStack(string[])
<pre>public void CreateNewCardStack(string[] ids)</pre>
ids <u>string</u> ♂[]

GetCardMate(ResourceLocation)



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.

<pre>protected override bool HasGodotClassMethod(in godot_string_name method)</pre>
method godot_string_name
Name of the method to check for.
<u>bool</u> ☑
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.
<pre>protected override bool InvokeGodotClassMethod(in godot_string_name method, NativeVariantPtrArgs args, out godot_variant ret)</pre>
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.
<u>bool</u> ♂

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. IS erialization Listener.

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.

info GodotSerializationInfo

Object used to save the data.

ProjectStack.Common 🛛	
-----------------------	--

П

ResourceLocation

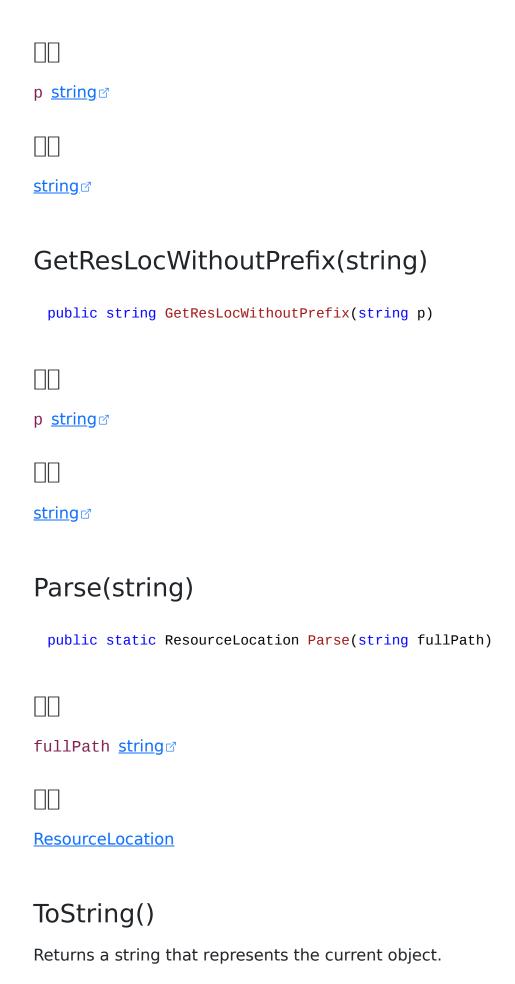
ResourceLocation [] ProjectStack.Common □□□: ProjectStack.dll public record ResourceLocation ResourceLocation(string, string) public ResourceLocation(string Namespace, string Path) Namespace <u>string</u> ☑ Path <u>string</u> □ **EMPTY** public static ResourceLocation EMPTY { get; } **ResourceLocation**

GetNamespace

```
public string GetNamespace { get; }
GetPath
 public string GetPath { get; }
Namespace
 public string Namespace { get; init; }
Path
 public string Path { get; init; }
DefaultNamespaceAndPath(string)
```

<pre>public static ResourceLocation DefaultNamespaceAndPath(string path)</pre>
path <u>string</u> ♂
ResourceLocation
FromNamespaceAndPath(string, string)
<pre>public static ResourceLocation FromNamespaceAndPath(string @namespace, string path)</pre>
namespace <u>string</u>
path <u>string</u> ♂
ResourceLocation
GetResLoc()
<pre>public string GetResLoc()</pre>
□□ string
GetResLocWithPath(string)

public string GetResLocWithPath(string p)



public override string ToString()

A string that represents the current object.

ProjectStack.Common.Card [[[[[
<u>CardMeta</u>
<u>CardMetaRegistrationHelper</u>
<u>CardStack</u>

ICardStack

ProjectStack.Common.Card □□□: ProjectStack.dll public record CardMeta object ← CardMeta CardMeta(ResourceLocation, string, string, ResourceLocation) public CardMeta(ResourceLocation Id, string Name, string Description, ResourceLocation Type) Id ResourceLocation Name <u>string</u> □ Description <u>string</u> ✓ Type ResourceLocation Description public string Description { get; init; }

CardMeta []

```
Id
 public ResourceLocation Id { get; init; }
ResourceLocation
Name
 public string Name { get; init; }
Type
 public ResourceLocation Type { get; init; }
ResourceLocation
Create(string)
 public static CardMeta Create(string id)
```

id <u>string</u>♂

<u>CardMeta</u>

ProjectStack.Common.Card □□□: ProjectStack.dll public class CardMetaRegistrationHelper <u>object</u> ← CardMetaRegistrationHelper CardMetaRegistrationHelper(IServiceCollection) public CardMetaRegistrationHelper(IServiceCollection services) Add(CardMeta) public CardMetaRegistrationHelper Add(CardMeta cardMeta) cardMeta CardMeta <u>CardMetaRegistrationHelper</u>

CardMetaRegistrationHelper []

End()

public IServiceCollection End()



<u>object</u> do ← CardStack

ICardStack



CardStack(IImmutableList<Card>)

public CardStack(IImmutableList<Card> Cards)



Cards IlmmutableListd < Card>



Cards

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



IlmmutableList

ICardStack □□

public interface ICardStack

Cards

IImmutableList<Card> Cards { get; }

IlmmutableList

ProjectStack.Common.Recipe [][]
<u>AbstractRecipe<trecipeinput></trecipeinput></u>
<u>CardMetaMatchRecipe</u>
RecipeInput
RecipeOutput
<u>RecipeRegistrationHelper</u>
RecipeResult
<u>ScriptRecipeInput</u>
ScriptRecipeInput.ScriptContext
SimpleRecipe □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
<u>SimpleRecipe.Ingredient</u>
<u>SimpleRecipe.Product</u>
<u>IRecipe</u>

<u>IRecipeInput</u>

AbstractRecipe<TRecipeInput> [] ProjectStack.Common.Recipe □□□: ProjectStack.dll public abstract class AbstractRecipe<TRecipeInput> TRecipeInput ПП <u>object</u> ∠ ← AbstractRecipe<TRecipeInput> Assmble(TRecipeInput, JsonObject) public abstract RecipeOutput Assmble(TRecipeInput recipeInput, JsonObject ntj) recipeInput TRecipeInput $\Box\Box$ **RecipeOutput** Matchs(TRecipeInput, JsonObject) public abstract bool Matchs(TRecipeInput recipeInput, JsonObject ntj)

recipeInput TRecipeInput

ntj JsonObject

□□

bool

Type()

public abstract ResourceLocation Type()

□□

ResourceLocation

CardMetaMatchRecipe []

```
ProjectStack.Common.Recipe
□□□: ProjectStack.dll
 public class CardMetaMatchRecipe : IRecipe
<u>object</u>  

← CardMetaMatchRecipe
<u>IRecipe</u>
CardMetaMatchRecipe(IList<CardMeta>,
IList<CardMeta>, bool)
 public CardMetaMatchRecipe(IList<CardMeta> requiredCardMetas, IList<CardMeta>
 producedCardMetas, bool isNeedOrder = false)
requiredCardMetas <u>|List</u> < <u>CardMeta</u> >
producedCardMetas <u>IList</u> < <u>CardMeta</u>>
isNeedOrder boold
CardViewCount
public uint? CardViewCount { get; }
```

<u>uint</u> ♂?
IsNeedOrder
<pre>public bool IsNeedOrder { get; set; }</pre>
<u>bool</u> ☑
ProducedCardMetas
<pre>public IList<cardmeta> ProducedCardMetas { get; set; }</cardmeta></pre>
<u>IList</u>
RequiredCardMetas
<pre>public IList<cardmeta> RequiredCardMetas { get; set; }</cardmeta></pre>
<u>IList</u>
Execute(ICardStack)

<pre>public RecipeResult Execute(ICardStack</pre>	cardStack)
cardStack <u>ICardStack</u>	
RecipeResult	

IRecipe □□ □□□: <u>ProjectStack</u>.<u>Common</u>.<u>Recipe</u> □□□: ProjectStack.dll public interface IRecipe CardViewCount 0000000000000null000000 uint? CardViewCount { get; } uint_□? Execute(ICardStack) RecipeResult Execute(ICardStack cardStack) cardStack ICardStack **RecipeResult**



IRecipeInput □□

□□□□: ProjectStack.Common.Recipe

□□□: ProjectStack.dll

public interface IRecipeInput

RecipeInput [] □□□□: ProjectStack.Common.Recipe □□□: ProjectStack.dll public abstract record RecipeInput <u>object</u> ♂ ← RecipeInput **Derived ScriptRecipeInput** Assemble(Card) public abstract Card Assemble(Card card) card Card Card IsMatch(Card) public abstract bool IsMatch(Card card) card Card



<u>bool</u> ♂

RecipeOutput [] □□□: <u>ProjectStack</u>.<u>Common</u>.<u>Recipe</u> □□□: ProjectStack.dll public record RecipeOutput <u>object</u> ← RecipeOutput RecipeOutput(List<Card>, JsonObject) public RecipeOutput(List<Card> Cards, JsonObject Ntj) Cards <u>List</u> < <u>Card</u>> Cards public List<Card> Cards { get; init; } <u>List</u> < <u>Card</u> >

Ntj

public JsonObject Ntj { get; init; }

[][]
JsonObject

ProjectStack.Common.Recipe □□□: ProjectStack.dll public class RecipeRegistrationHelper <u>object</u> < RecipeRegistrationHelper RecipeRegistrationHelper(IServiceCollection) public RecipeRegistrationHelper(IServiceCollection services) Add(IRecipe) public RecipeRegistrationHelper Add(IRecipe recipe) recipe <u>IRecipe</u> RecipeRegistrationHelper

RecipeRegistrationHelper []

End()

public IServiceCollection End()



ProjectStack.Common.Recipe □□□: ProjectStack.dll public record RecipeResult <u>object</u> ♂ ← RecipeResult RecipeResult(bool, IEnumerable < Card > , IEnumerable < Card >) public RecipeResult(bool IsMatch, IEnumerable<Card> ConsumedCards, IEnumerable<Card> ProducedCards) IsMatch <u>bool</u> ✓ ConsumedCards | Enumerable | < Card > ProducedCards <u>IEnumerable</u> < <u>Card</u>> ConsumedCards public IEnumerable<Card> ConsumedCards { get; init; }

RecipeResult []

IsMatch

	public	bool	IsMatch	{	get;	<pre>init;</pre>	}
b							

ProducedCards

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



<u>IEnumerable</u> < <u>Card</u>>

ScriptRecipeInput □ □□: ProjectStack.Common.Recipe □□: ProjectStack.dll public record ScriptRecipeInput : RecipeInput □□ object □ ← RecipeInput ← ScriptRecipeInput

ScriptRecipeInput()

```
public ScriptRecipeInput()
```



AssembleScript

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



<u>string</u> □

AssembleScriptRunner

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



<u>ScriptRunner</u> < <u>Card</u> >

MatchScript

IsMatch(Card)

```
public string MatchScript { get; init; }
MatchScriptRunner
 public ScriptRunner<bool> MatchScriptRunner { get; }
<u>ScriptRunner</u> d < <u>bool</u> d >
Assemble(Card)
 public override Card Assemble(Card card)
card Card
Card
```

public override bool IsMatch(Card card)

Card Card

D

<u>bool</u>♂

ScriptRecipeInput.ScriptContext []

ProjectStack.Common.Recipe

ProjectStack.dll

public class ScriptRecipeInput.ScriptContext

be object ← ScriptRecipeInput.ScriptContext

card

public Card card

Card

ProjectStack.Common.Recipe □□□: ProjectStack.dll public record SimpleRecipe SimpleRecipe(ResourceLocation, string, string, float, IlmmutableList<Ingredient>, IlmmutableList<Product>) public SimpleRecipe(ResourceLocation Id, string Name, string Description, float Production, IImmutableList<SimpleRecipe.Ingredient> Ingredients, IImmutableList<SimpleRecipe.Product> Products) Id ResourceLocation Name <u>string</u> <a>d Description string Production <u>float</u> ♂ Ingredients <u>IlmmutableList</u> < <u>SimpleRecipe.Ingredient</u>>

SimpleRecipe []

Products <u>IImmutableList</u> < <u>SimpleRecipe.Product</u> >
Description
<pre>public string Description { get; init; }</pre>
<u>string</u> d
Id
<pre>public ResourceLocation Id { get; init; }</pre>
ResourceLocation
Ingredients
<pre>public IImmutableList<simplerecipe.ingredient> Ingredients { get; init; }</simplerecipe.ingredient></pre>
<u>IImmutableList</u> < <u>SimpleRecipe</u> . <u>Ingredient</u> >

Name public string Name { get; init; } <u>string</u> □ Production public float Production { get; init; } <u>float</u> ♂ **Products** public IImmutableList<SimpleRecipe.Product> Products { get; init; } SatisfactionCheck(IlmmutableList<CardMeta>) public bool SatisfactionCheck(IImmutableList<CardMeta> cards)

cards <u>IlmmutableList</u> < <u>CardMeta</u>>

□□

bool □

SimpleRecipe.Ingredient [] ProjectStack.Common.Recipe □□□: ProjectStack.dll public record SimpleRecipe.Ingredient Ingredient(ResourceLocation, int, bool) public Ingredient(ResourceLocation CardId, int Quantity, bool Consumed) CardId ResourceLocation Quantity int♂ CardId public ResourceLocation CardId { get; init; }

ResourceLocation

Consumed

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

Dool
```

Quantity

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



<u>int</u>♂

SimpleRecipe.Product [] ProjectStack.Common.Recipe □□□: ProjectStack.dll public record SimpleRecipe.Product <u>object</u> ♂ ← SimpleRecipe.Product Product(ResourceLocation, int) public Product(ResourceLocation CardId, int Quantity) CardId ResourceLocation Quantity int♂ CardId

public ResourceLocation CardId { get; init; }

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 <u>Task</u> value.

Card.TaskNotifier<T>

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

InfoTab.TaskNotifier<T>

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

Card □

ProjectStack.Component

□□□: ProjectStack.dll

bool₫

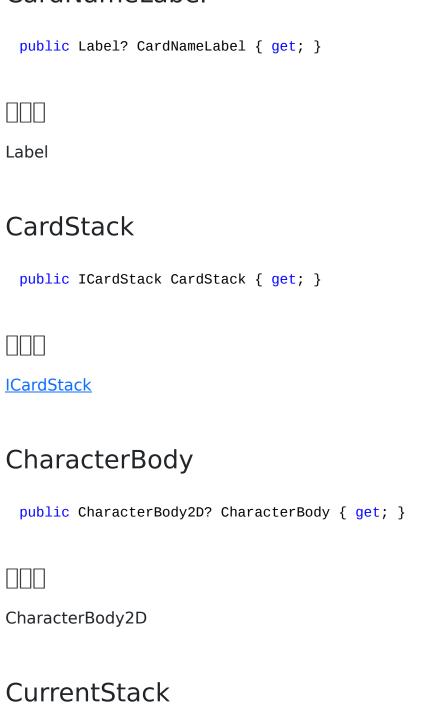
A 2D game object, with a transform (position, rotation, and scale). All 2D nodes, including physics objects and sprites, inherit from Node2D. Use Node2D as a parent node to move, scale and rotate children in a 2D project. Also gives control of the node's render order.

```
[ObservableObject]
  [Meta(new Type[] { typeof(IAutoNode) })]
  [ScriptPath("res://src/scripts/Component/Card.cs")]
 public class Card : Node2D, INtjObject
\Pi\Pi
<u>object</u> ✓ ← GodotObject ← Node ← CanvasItem ← Node2D ← Card
INtjObject
Card()
 public Card()
ForceMotion
 public bool ForceMotion
```

InMoveing



CardNameLabel



public IImmutableList<Card> CurrentStack { get; }

IsRoot

<pre>public bool IsRoot { get; }</pre>
<u>bool</u> d'
IsUppest
<pre>public bool IsUppest { get; }</pre>
<u>bool</u> ♂
Metatype
Generated metatype information.
<pre>public IMetatype Metatype { get; }</pre>
IMetatype
MixinState
Arbitrary data that is shared between mixins. Mixins are free to store additional instance state in this blackboard.
<pre>public MixinBlackboard MixinState { get; }</pre>

MixinBlackboard

Ntj

public JsonObject Ntj { get; set; }

OnDrag

public bool OnDrag { get; set; }



<u>bool</u> ♂

Panel

public Control? Panel { get; }



Control

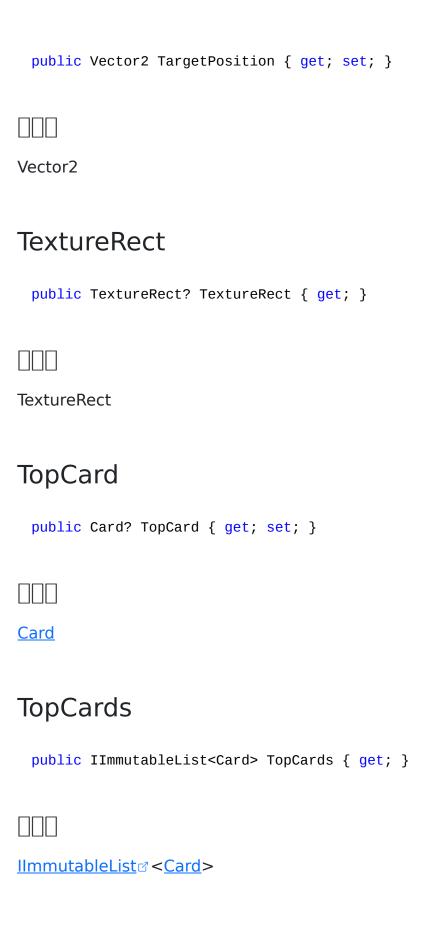
RootCard

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



Card

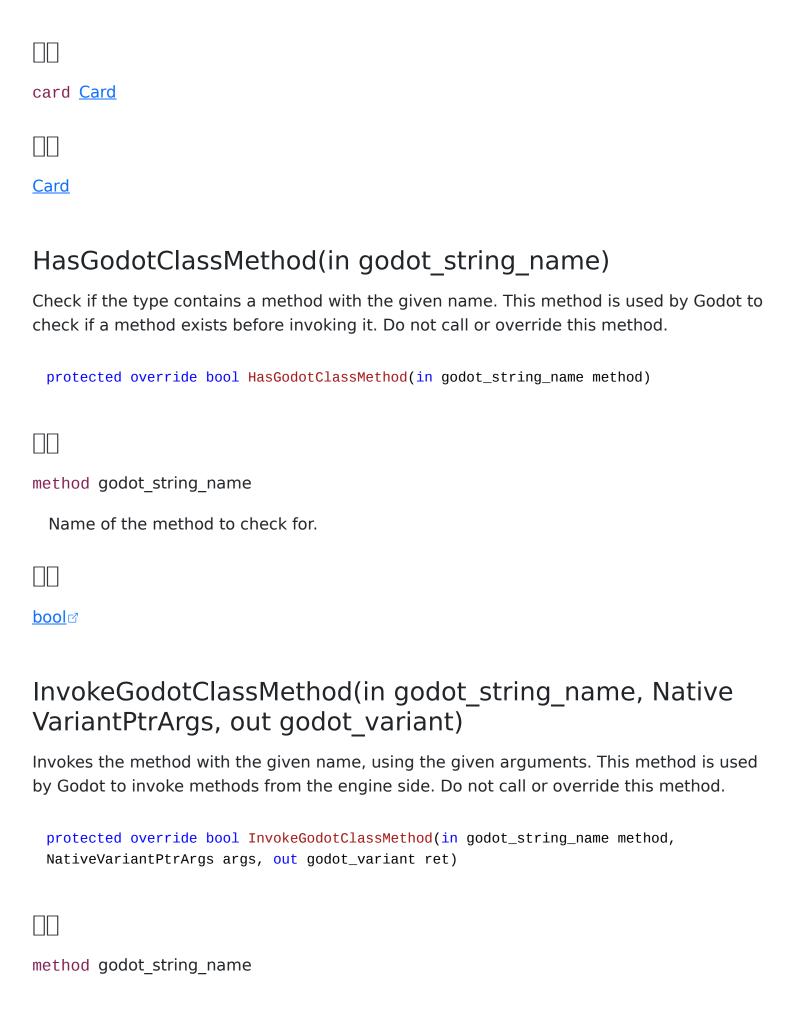
TargetPosition

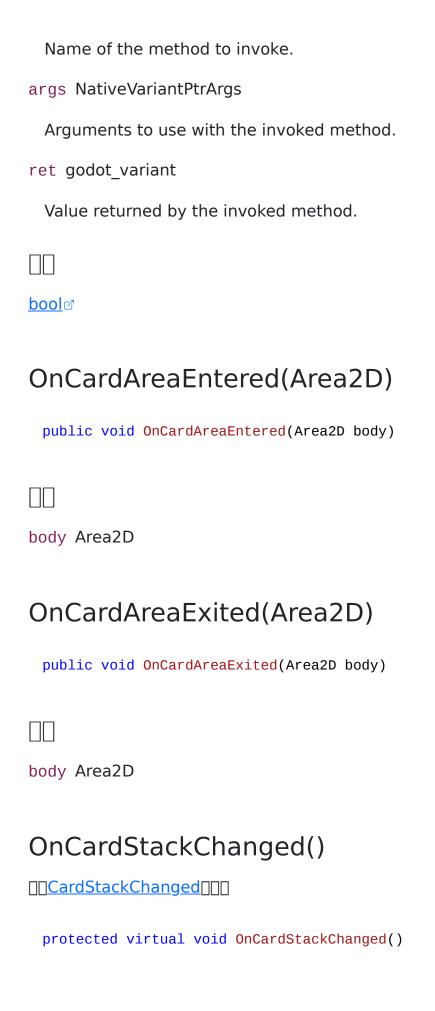


UppestCard



72 / 137





OnPropertyChanged(PropertyChangedEventArgs)

Raises the **PropertyChanged** event. protected virtual void OnPropertyChanged(PropertyChangedEventArgs e) e <u>PropertyChangedEventArgs</u>

☑ The input <u>PropertyChangedEventArgs</u> instance. OnPropertyChanged(string?) Raises the **PropertyChanged** event. protected void OnPropertyChanged(string? propertyName = null) propertyName <u>string</u>♂ (optional) The name of the property that changed. OnPropertyChanging(PropertyChangingEventArgs) Raises the **PropertyChanging** event. protected virtual void OnPropertyChanging(PropertyChangingEventArgs e)

 $e \ \underline{PropertyChangingEventArgs} \boxtimes$

The input <u>PropertyChangingEventArgs</u> 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()

RefreshTexture()

public void RefreshTexture()

RestoreGodotObjectData(GodotSerializationInfo)

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

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. IS erialization 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 ₫ <u>true</u> 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 SetPropertyAnd NotifyOnCompletion(ref TaskNotifier?, Task?, string?), with the difference being an extra <a href="Action<T>©">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 <u>Task</u>♂ The property's value after the change occurred. callback <u>Action</u> ♂ < <u>Task</u> ♂ > A callback to invoke to update the property value. propertyName <u>string</u>♂ (optional) The name of the property that changed. bool₫ <u>true</u> if the property was changed, <u>false</u> otherwise.

The <u>PropertyChanging</u> and <u>PropertyChanged</u> 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 <a href="SetProperty<">SetProperty (ref T, T, string?), with the difference being that this method will also monitor the new value of the property (a generic Taske) and will also raise the Property Changed again for the target property when it completes. This can be used to update bindings observing that Taske 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 Taske 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 Taske 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.
(optional) The name of the property that changed.
```

<u>bool</u> ♂
true do if the property was changed, false do otherwise.
The <u>PropertyChanging</u> and <u>PropertyChanged</u> events are not raised if the current and new value for the target property are the same. The return value being <u>true</u> only indicates that the new value being assigned to <u>taskNotifier</u> is different than the previous one, and it does not mean the new <u>Task</u> instance passed as argument is in any particular state.
SetPropertyAndNotifyOnCompletion <t>(ref TaskNotifier<t>?, Task<t>?, Action<task<t>?>, string?)</task<t></t></t></t>
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 <a action<t="" href="SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>?, Task<T>?, string?), with the difference being an extra @">Action<t>@</t> parameter with a callback being invoked either immediately, if the new task has already completed or is null@ , or upon completion.
<pre>protected bool SetPropertyAndNotifyOnCompletion<t>(ref Card.TaskNotifier<t>? taskNotifier, Task<t>? newValue, Action<task<t>?> callback, string? propertyName = null)</task<t></t></t></t></pre>
taskNotifier <u>Card</u> . <u>TaskNotifier</u> <t></t>
The field notifier to modify.
newValue <u>Task</u> d <t></t>
The property's value after the change occurred.
callback <u>Action</u> ♂< <u>Task</u> ♂ <t>></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.

Т

The type of result for the <u>Task<TResult></u> do set and monitor.

The <u>PropertyChanging</u> and <u>PropertyChanged</u> 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 <a href="SetProperty<">SetProperty (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 <a href="Card.TaskNotifier<T>">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 <a href="Card.TaskNotifier<T>">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 Taskd<T>
  The property's value after the change occurred.
(optional) The name of the property that changed.
bool♂
  <u>true</u> if the property was changed, <u>false</u> otherwise.
Т
  The type of result for the <u>Task<TResult></u> do set and monitor.
The <u>PropertyChanging</u> and <u>PropertyChanged</u> 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
```

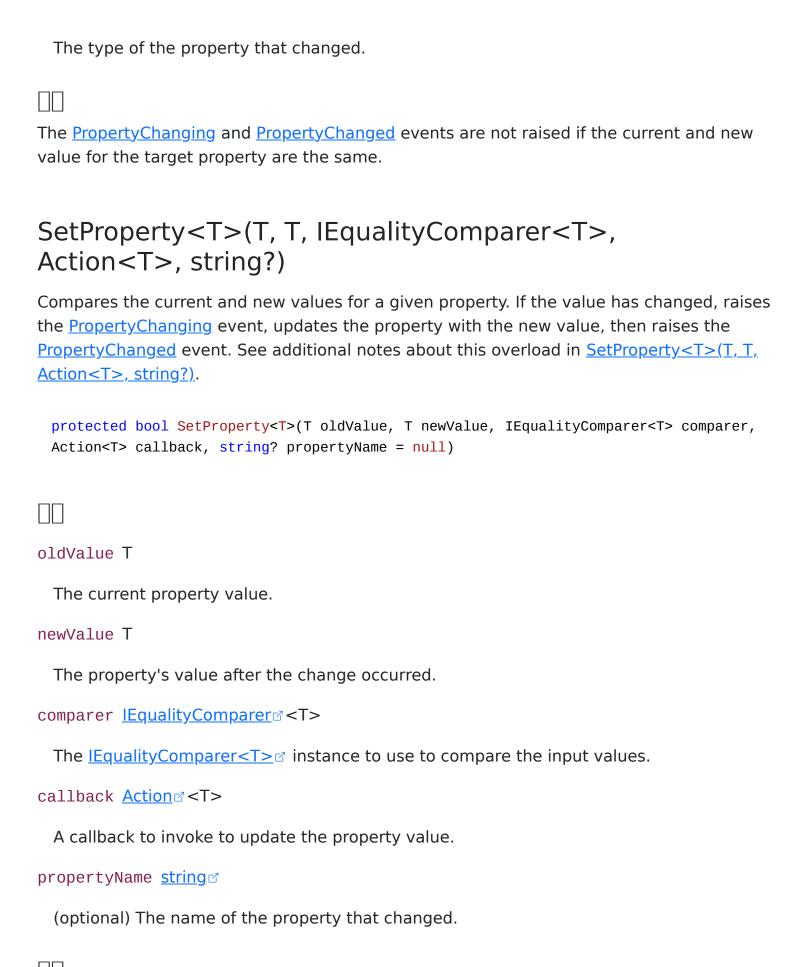
the new value being assigned to taskNotifier is different than the previous one, and it does not mean the new <a href="mailto:Task Instance passed as argument is in any particular state.

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 <a href="SetProperty<T>(ref T, T, string?">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 <a href="SetProperty<TModel">SetProperty<TModel, TotAT, T, TModel, <a href="Action<TModel">Action<TModel, TotAT, 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.

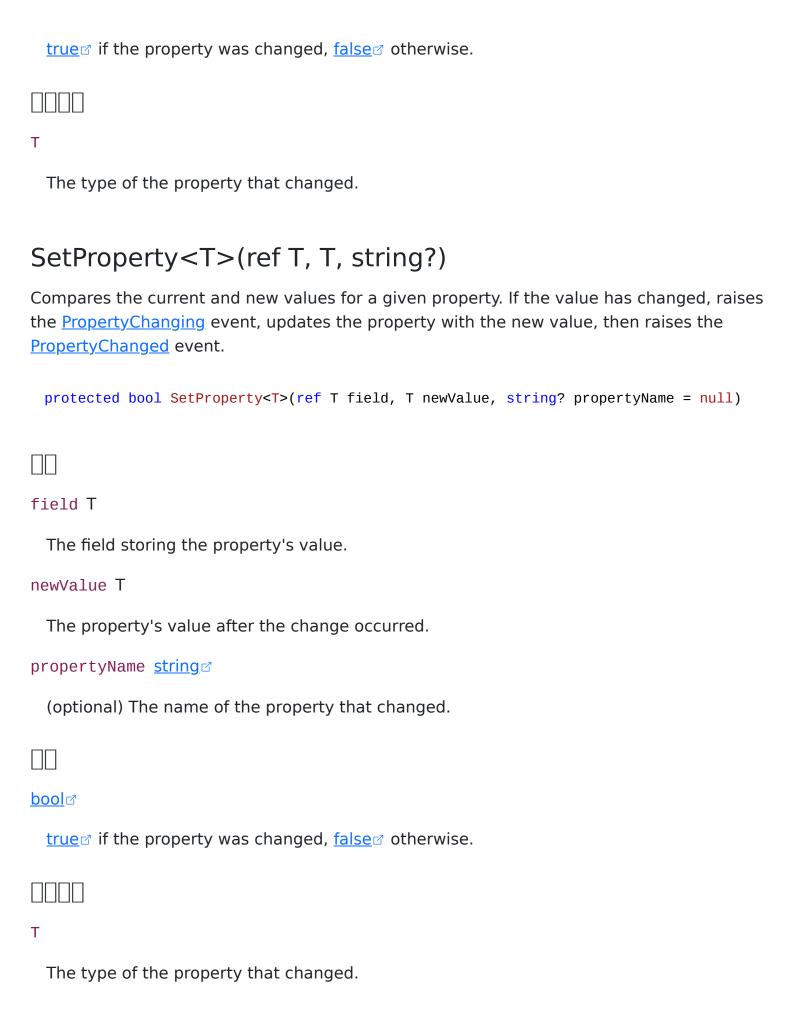
```
protected bool SetProperty<T>(T oldValue, T newValue, Action<T> callback, string?
 propertyName = null)
oldValue T
  The current property value.
newValue T
  The property's value after the change occurred.
callback <u>Action</u> < < T >
  A callback to invoke to update the property value.
propertyName string
  (optional) The name of the property that changed.
\Box\Box
bool ₫
  <u>true</u> if the property was changed, <u>false</u> otherwise.
```

Т



bool₫ <u>true</u> if the property was changed, <u>false</u> otherwise. Т 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 <u>PropertyChanging</u> event, updates the property with the new value, then raises the <u>PropertyChanged</u> event. See additional notes about this overload in <u>SetProperty<T>(ref T,</u> 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 <u>IEqualityComparer</u> < T> The <u>IEqualityComparer<T></u> instance to use to compare the input values. propertyName <u>string</u>♂ (optional) The name of the property that changed.

bool₫





The <u>PropertyChanging</u> and <u>PropertyChanged</u> 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 <a href="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 <a href="SetProperty<TModel">SetProperty<TModel, <a href="Total Total Content of the Instance of the Instance of Total Content of Total Conten

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 <u>IEqualityComparer</u> < T>

The <u>IEqualityComparer<T></u> 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 d

(optional) The name of the property that changed.



<u>true</u> if the property was changed, <u>false</u> otherwise.



TModel

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

Т

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 <u>PropertyChanging</u> event, updates the property and then raises the <u>PropertyChanged</u> event. The behavior mirrors that of <u>SetProperty<T>(ref T, T, string?)</u>, 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 <a href="SetProperty<T>(ref T, T, string?">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 Callback Action
```

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

```
propertyName <u>string</u>♂
```

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).

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 <u>int</u> □

_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 <u>Notification(int)</u> ...

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 <u>double</u>♂

_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 <u>Notification(int)</u>. See also the <code>@onready</code> 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



<u>Action</u> < <u>ICardStack</u>>

PropertyChanged

Occurs when a property value changes.

PropertyChangedEventHandler

PropertyChanging

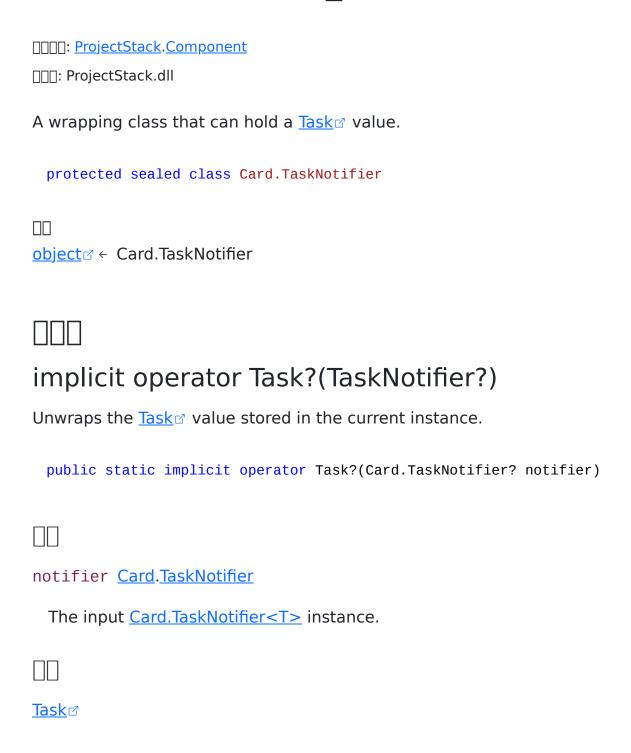
Occurs when a property value is changing.

public event PropertyChangingEventHandler? PropertyChanging

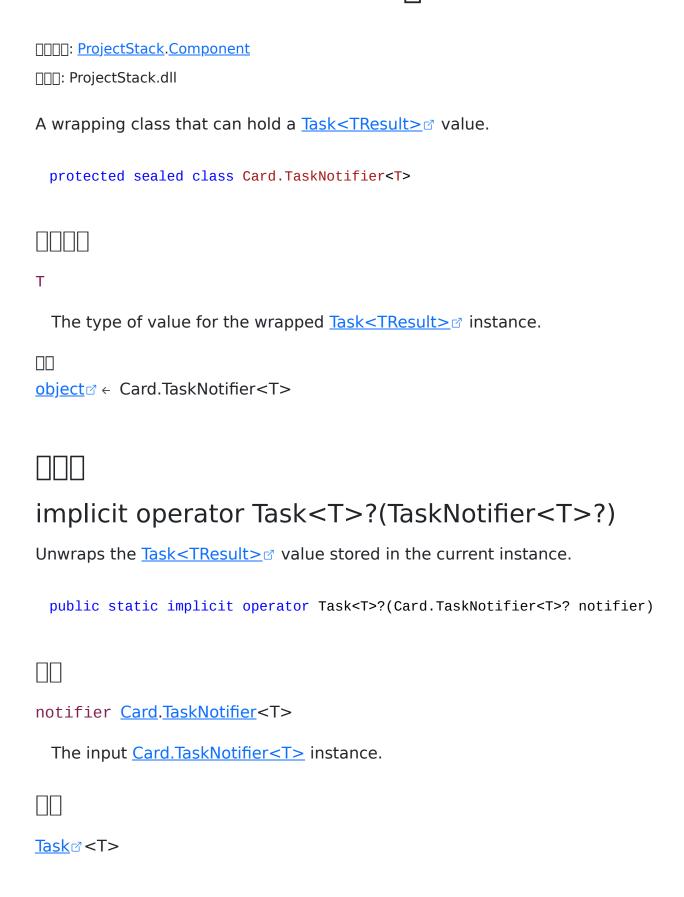
public event PropertyChangedEventHandler? PropertyChanged



Card.TaskNotifier []



Card.TaskNotifier<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.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/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.
<pre>protected override bool GetGodotClassPropertyValue(in godot_string_name name, out godot_variant value)</pre>
name godot_string_name
Name of the property to get.
value godot_variant
Value of the property if it was found.
<u>bool</u> ♂
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.
<pre>protected override bool HasGodotClassMethod(in godot_string_name method)</pre>
method godot_string_name
Name of the method to check for.



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.

OnProcess(double)

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

public void OnProcess(double delta)



delta <u>double</u>♂

Time since the last process update, in seconds.

OnPropertyChanged(PropertyChangedEventArgs)

Raises the **PropertyChanged** event.

 ${\tt protected\ virtual\ void\ OnPropertyChanged(PropertyChangedEventArgs\ e)}$

e <u>PropertyChangedEventArgs</u>

☑

The input PropertyChangedEventArgs instance.

OnPropertyChanged(string?)

Raises the <u>PropertyChanged</u> event.

protected void OnPropertyChanged(string? propertyName = null)

propertyName <u>string</u> ☐

(optional) The name of the property that changed.

OnPropertyChanging(PropertyChangingEventArgs)

Raises the <u>PropertyChanging</u> event.

protected virtual void OnPropertyChanging(PropertyChangingEventArgs e)

e <u>PropertyChangingEventArgs</u>

☑

The input <u>PropertyChangingEventArgs</u> instance.

OnPropertyChanging(string?)

Raises the <u>PropertyChanging</u> event.

protected void OnPropertyChanging(string? propertyName = null)

propertyName <u>string</u>♂

(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. IS erialization Listener.

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. IS erialization 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 ₫ <u>true</u> 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 SetPropertyAnd NotifyOnCompletion(ref TaskNotifier?, Task?, string?), with the difference being an extra <a href="Action<T><a href="Action<T><a href="Action<T>Action<T>TaskNotifier?, Task?, string?), with the difference being an extra Action<T>TaskNotifier?, TaskNotifier?, TaskNotifier?, ref">TaskNotifier?, TaskNotifier?, <a

protected bool SetPropertyAndNotifyOnCompletion(ref InfoTab.TaskNotifier?

taskNotifier, Task? newValue, Action<Task?> callback, string? propertyName = null) taskNotifier InfoTab.TaskNotifier The field notifier to modify. newValue <u>Task</u>♂ The property's value after the change occurred. callback <u>Action</u> ♂ < <u>Task</u> ♂ > A callback to invoke to update the property value. propertyName <u>string</u>♂ (optional) The name of the property that changed. bool₫ <u>true</u> if the property was changed, <u>false</u> otherwise. \prod The <u>PropertyChanging</u> and <u>PropertyChanged</u> 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 <a href="SetProperty<">SetProperty (ref T, T, string?), with the difference being that this method will also monitor the new value of the property (a generic Taske) and will also raise the Property Changed again for the target property when it completes. This can be used to update bindings observing that Taske 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 Taske 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 Taske 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.
(optional) The name of the property that changed.
```

<u>bool</u> ♂
true do if the property was changed, false do otherwise.
The <u>PropertyChanging</u> and <u>PropertyChanged</u> events are not raised if the current and new value for the target property are the same. The return value being <u>true</u> only indicates that the new value being assigned to <u>taskNotifier</u> is different than the previous one, and it does not mean the new <u>Task</u> instance passed as argument is in any particular state.
SetPropertyAndNotifyOnCompletion <t>(ref TaskNotifier<t>?, Task<t>?, Action<task<t>?>, string?)</task<t></t></t></t>
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 <a action<t="" href="SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>?, Task<T>?, string?), with the difference being an extra @">Action<t>@</t> parameter with a callback being invoked either immediately, if the new task has already completed or is null@ , or upon completion.
<pre>protected bool SetPropertyAndNotifyOnCompletion<t>(ref InfoTab.TaskNotifier<t>? taskNotifier, Task<t>? newValue, Action<task<t>?> callback, string? propertyName = null)</task<t></t></t></t></pre>
taskNotifier <u>InfoTab.TaskNotifier</u> <t></t>
The field notifier to modify.
newValue <u>Task</u> d <t></t>
The property's value after the change occurred.
callback <u>Action</u> ♂< <u>Task</u> ♂ <t>></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.

Т

The type of result for the <u>Task<TResult></u> do set and monitor.

The <u>PropertyChanging</u> and <u>PropertyChanged</u> 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 <a href="SetProperty<T>(ref T, T, string?">SetProperty<T>(ref T, T, string?), with the difference being that this method will also monitor the new value of the property (a generic Taske") and will also raise the Property
Changed again for the target property when it completes. This can be used to update bindings observing that Taske or any of its properties. This method and its overload specifically rely on the <a href="InfoTab.TaskNotifier<T>">InfoTab.TaskNotifier<T> type, which needs to be used in the backing field for the target Taske property. The field doesn't need to be initialized, as this method will take care of doing that automatically. The <a href="InfoTab.TaskNotifier<T>">InfoTab.TaskNotifier<T> type also includes an implicit operator, so it can be assigned to any Taske 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 Taskd<T>
  The property's value after the change occurred.
(optional) The name of the property that changed.
bool₫
  <u>true</u> if the property was changed, <u>false</u> otherwise.
Т
  The type of result for the <u>Task<TResult></u> do set and monitor.
The <u>PropertyChanging</u> and <u>PropertyChanged</u> events are not raised if the current and new
```

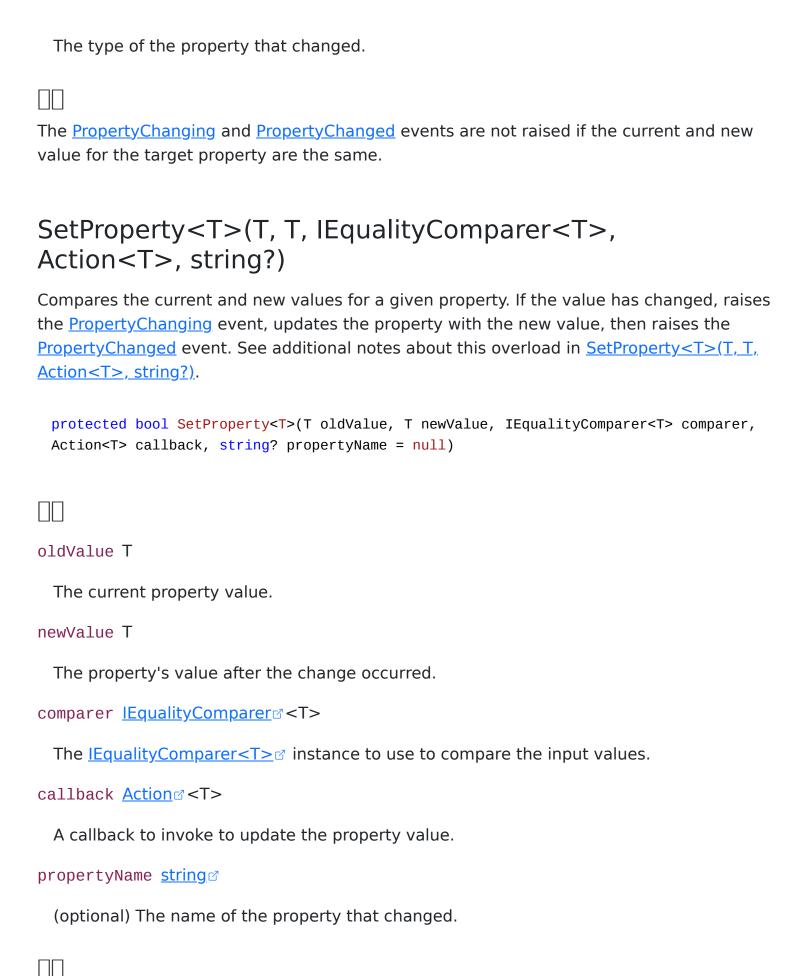
value for the target property are the same. The return value being true only indicates that the new value being assigned to taskNotifier is different than the previous one, and it does not mean the new <u>Task<TResult></u> instance passed as argument is in any particular state.

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 <a href="SetProperty<T>(ref T, T, string?">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 <a href="SetProperty<TModel">SetProperty<TModel, <a href="TotAction<TModel">TotAction<TModel, <a href="TotActio

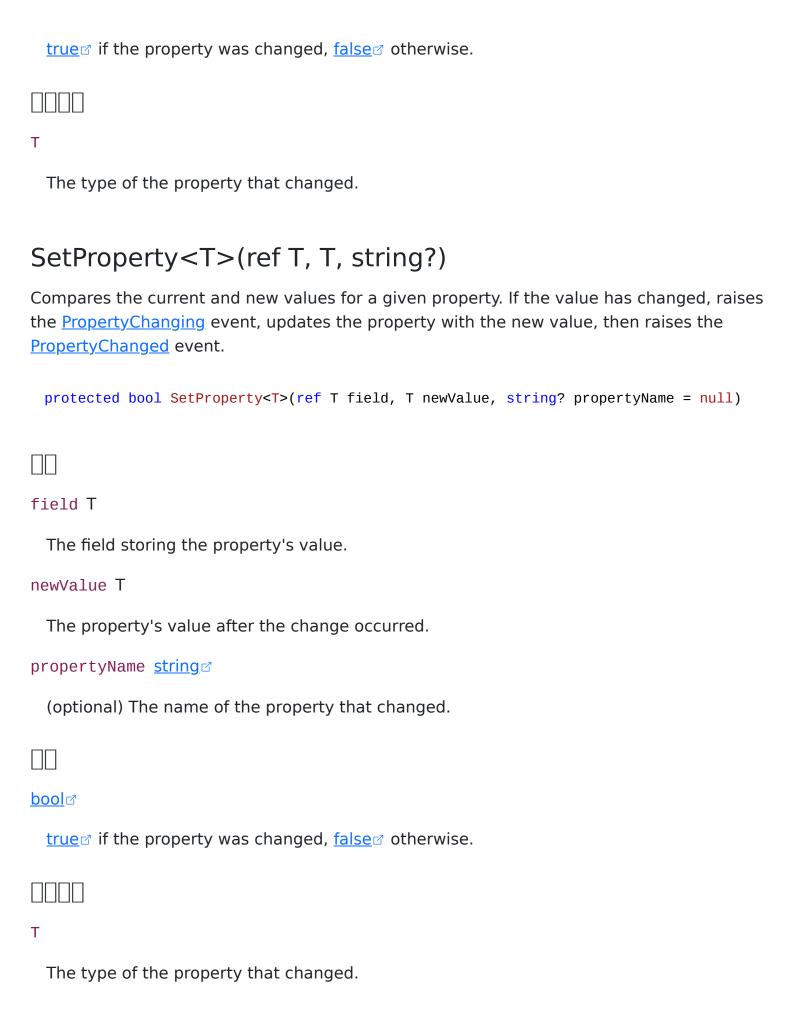
```
protected bool SetProperty<T>(T oldValue, T newValue, Action<T> callback, string?
 propertyName = null)
oldValue T
  The current property value.
newValue T
  The property's value after the change occurred.
callback <u>Action</u> < < T >
  A callback to invoke to update the property value.
propertyName string
  (optional) The name of the property that changed.
\Box\Box
bool ₫
  <u>true</u> if the property was changed, <u>false</u> otherwise.
```

Т



bool₫ <u>true</u> if the property was changed, <u>false</u> otherwise. Т 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 <u>PropertyChanging</u> event, updates the property with the new value, then raises the <u>PropertyChanged</u> event. See additional notes about this overload in <u>SetProperty<T>(ref T,</u> 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 <u>IEqualityComparer</u> < T> The <u>IEqualityComparer<T></u> instance to use to compare the input values. propertyName <u>string</u>♂ (optional) The name of the property that changed.

bool♂





The <u>PropertyChanging</u> and <u>PropertyChanged</u> 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 <u>PropertyChanging</u> event, updates the property and then raises the <u>PropertyChanged</u> event. The behavior mirrors that of <u>SetProperty<T>(ref T, T, string?)</u>, 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 <u>SetProperty<TModel, T>(T, T, TModel, Action<TModel, T>, string?)</u>.

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 <u>IEqualityComparer</u> < T>

The <u>IEqualityComparer<T></u> 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 d

(optional) The name of the property that changed.



<u>true</u> if the property was changed, <u>false</u> otherwise.



TModel

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

Т

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 <u>PropertyChanging</u> event, updates the property and then raises the <u>PropertyChanged</u> event. The behavior mirrors that of <u>SetProperty<T>(ref T, T, string?)</u>, 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 <a href="SetProperty<T>(ref T, T, string?">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 <u>string</u>♂
```

(optional) The name of the property that changed.

bool ₫

<u>true</u> if the property was changed, <u>false</u> otherwise.

TModel

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

Т

The type of property (or field) to set.

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

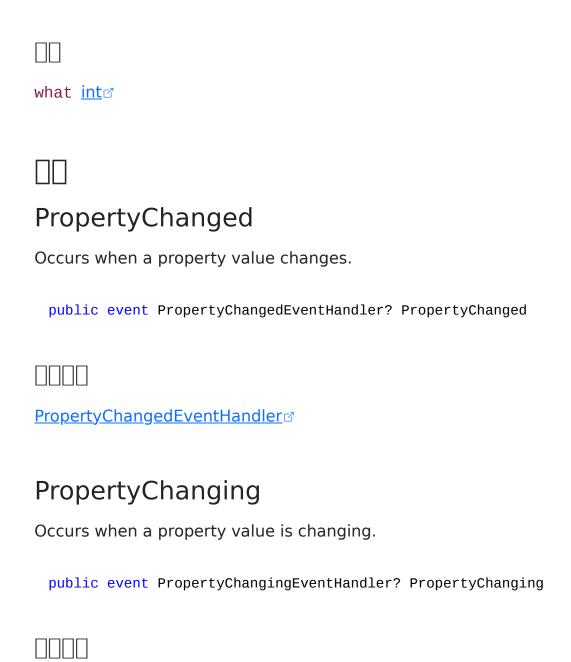
Notification(int)

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!");
      }
  }
```

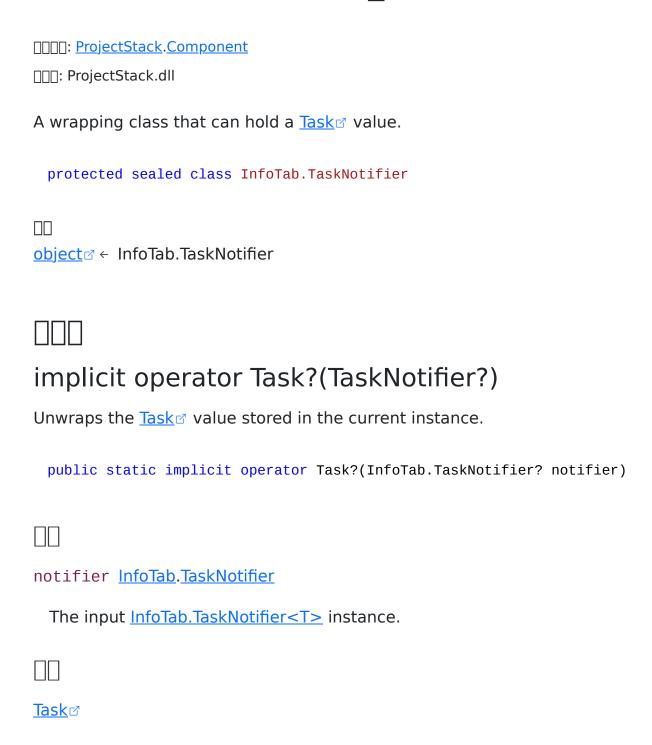
Note: The base Godot.GodotObject defines a few notifications (Godot.GodotObject. NotificationPostinitialize and Godot.GodotObject.NotificationPredelete). Inheriting classes such as Godot. Node define a lot more notifications, which are also received by this method.

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

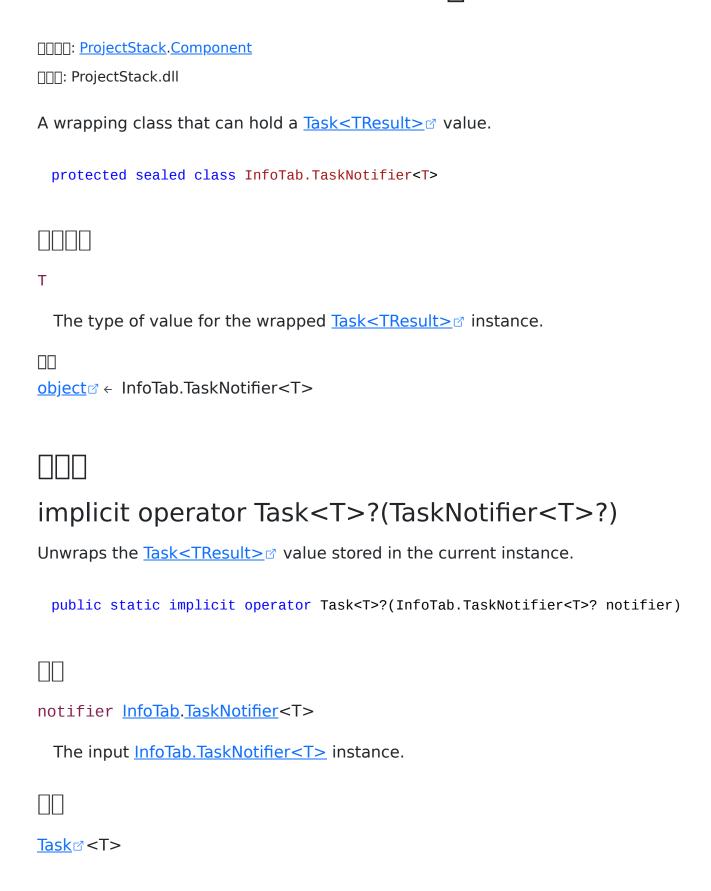


 $\underline{PropertyChangingEventHandler} \boxdot$

InfoTab.TaskNotifier []



InfoTab.TaskNotifier<T> []



Pro _.	jects	Stac	k.(Cor	e			1
------------------	-------	------	-----	-----	---	--	--	---

П

<u>CardMgr</u>

<u>ServiceCollectionExtension</u>

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. IS erialization Listener.

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. IS erialization Listener.

protected override void SaveGodotObjectData(GodotSerializationInfo info)



info GodotSerializationInfo

Object used to save the data.

ServiceCollectionExtension []
□□□: ProjectStack.Core □□□: ProjectStack.dll
<pre>public static class ServiceCollectionExtension</pre>
<pre>□□ object ← ServiceCollectionExtension</pre>
Configure Hovered Item Info Display (IS ervice Collection)
<pre>public static HoveredItemInfoDisplayRegistrationHelper ConfigureHoveredItemInfoDisplay(this IServiceCollection services)</pre>
services <u>IServiceCollection</u>
<u>HoveredItemInfoDisplayRegistrationHelper</u>
RegisterCardMetas(IServiceCollection)
<pre>public static CardMetaRegistrationHelper RegisterCardMetas(this IServiceCollection services)</pre>
services <u>IServiceCollection</u>

RegisterRecipes(IServiceCollection)

<pre>public static RecipeRegistrationHelper RegisterRecipes(this IServiceCollection services)</pre>
services <u>IServiceCollection</u>
<u>RecipeRegistrationHelper</u>
RegisterTextureLoader(IServiceCollection) public static TextureLoader RegisterTextureLoader(this IServiceCollection services)
services <u>IServiceCollection</u>
<u>TextureLoader</u>

ProjectStack.NamedTagsBaseOnJson	

<u>INtjObject</u>

INtjObject []

□□□□: ProjectStack.NamedTagsBaseOnJson
□□: ProjectStack.dll

public interface INtjObject

Ntj

Ntj

JsonObject Ntj { get; }

ProjectStack.Resource [
-------------------------	--

П

<u>TextureLoader</u>

ProjectStack.Resource □□□: ProjectStack.dll public class TextureLoader <u>object</u> ← TextureLoader TextureLoader(IServiceCollection) public TextureLoader(IServiceCollection services) AddTextureDirectory(string) public TextureLoader AddTextureDirectory(string directoryPath) directoryPath string <u>TextureLoader</u>

TextureLoader []

End()

public IServiceCollection End()



${\sf ProjectStack.UserInterface} \ \square \square$	
--	--

 $\underline{\mathsf{HoveredItemInfoDisplay}}$

 $\underline{Hovered Item Info Display Registration Helper}$

<u>HoveredItemInfoProvider</u>

HoveredItemInfoDisplay [] ProjectStack. UserInterface □□□: ProjectStack.dll public class HoveredItemInfoDisplay <u>object</u> < HoveredItemInfoDisplay HoveredItemInfoDisplay(IServiceCollection) public HoveredItemInfoDisplay(IServiceCollection services) HoveredItemInfoProviders public List<HoveredItemInfoProvider> HoveredItemInfoProviders { get; } List ≥ < HoveredItemInfoProvider > HoveredItemInfoTexts

public IImmutableList<string> HoveredItemInfoTexts { get; }



<u>IlmmutableList</u> ♂ < <u>string</u> ♂ >

HoveredItems

public ObservableCollection<Node> HoveredItems { get; }





RegisterHoveredItemInfoProvider(HoveredItemInfoProvider)

<pre>public HoveredItemInfoDisplayRegistrationHelper RegisterHoveredItemInfoProvider(HoveredItemInfoProvider hoveredItemInfoProvider)</pre>
hoveredItemInfoProvider <u>HoveredItemInfoProvider</u>
<u>HoveredItemInfoDisplayRegistrationHelper</u>
RegisterHoveredItemInfoProvider(string, Predicate <hoverediteminfodisplay>, Func<hoverediteminfodisplay, string="">) public HoveredItemInfoDisplayRegistrationHelper</hoverediteminfodisplay,></hoverediteminfodisplay>
RegisterHoveredItemInfoProvider(string providerName, Predicate <hoverediteminfodisplay> predicate, Func<hoverediteminfodisplay, string=""> displayTextProvider)</hoverediteminfodisplay,></hoverediteminfodisplay>
providerName <u>string</u> ♂
predicate <u>Predicate</u> ✓ < <u>HoveredItemInfoDisplay</u> >
displayTextProvider <u>Func</u> < <u>HoveredItemInfoDisplay</u> , <u>string</u> ♂ >
<u>HoveredItemInfoDisplayRegistrationHelper</u>

HoveredItemInfoProvider []

```
□□□□: ProjectStack.UserInterface
□□□: ProjectStack.dll
 public class HoveredItemInfoProvider
object  

← HoveredItemInfoProvider
HoveredItemInfoProvider(string,
Predicate < Hovered Item Info Display > ,
Func<HoveredItemInfoDisplay, string>)
 public HoveredItemInfoProvider(string providerName,
 Predicate<HoveredItemInfoDisplay> predicate, Func<HoveredItemInfoDisplay,
  string> displayTextProvider)
providerName <u>string</u>♂
predicate Predicate <a href="Predicate">Predicate</a> <a href="Predicate">HoveredItemInfoDisplay</a>
displayTextProvider Func <a href="Func">Func</a> <a href="Func">HoveredItemInfoDisplay</a>, <a href="string">string</a> >
ProviderName
 public string ProviderName { get; }
```

<u>string</u> [™]
ProvideHoveredItemInfo(HoveredItemInfoDisplay)
<pre>public string? ProvideHoveredItemInfo(HoveredItemInfoDisplay hoveredItemInfoDisplay)</pre>
hoveredItemInfoDisplay <u>HoveredItemInfoDisplay</u>
<u>string</u> ♂

ProjectStack.Util 🛮 🔻	
-----------------------	--

<u>FileSystemHelper</u>

FileSystemHelper []

```
ProjectStack.Util
□□□: ProjectStack.dll
 public class FileSystemHelper
<u>object</u>  

✓ FileSystemHelper
Default
 public static FileSystemHelper Default { get; }
<u>FileSystemHelper</u>
GetAllFilesInDirectory(string)
 public IEnumerable<string> GetAllFilesInDirectory(string directoryPath)
directoryPath string ≥
<u>IEnumerable</u> ♂ < <u>string</u> ♂ >
```