ProjectStack □□□□

□

CardTest

Game

Game.MetatypeMetadata

Game.MethodName

Cached StringNames for the methods contained in this class, for fast lookup.

Game.PropertyName

Cached StringNames for the properties and fields contained in this class, for fast lookup.

Game.SignalName

Cached StringNames for the signals contained in this class, for fast lookup.

Main

Main.MethodName

Cached StringNames for the methods contained in this class, for fast lookup.

Main.PropertyName

Cached StringNames for the properties and fields contained in this class, for fast lookup.

Main.SignalName

Cached StringNames for the signals contained in this class, for fast lookup.

CardTest □

□□□□: ProjectStack

□□□: ProjectStack.dll

```
public class CardTest : TestClass
```

□□

object ← TestClass ← CardTest

□□□□

CardTest(Node)

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

□□

`testScene` Node

□□

Setup()

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

SetupAll()

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

TestBottomCards()

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

TestDisconnectTopCard()

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

TestTopCards()

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

Game □

□□□□: ProjectStack

□□□: ProjectStack.dll

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

□□

object ← GodotObject ← Node ← CanvasItem ← Node2D ← Game

□□

Default

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

□□□

Game

Metatype

Generated metatype information.

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

□□□

IMetatype

MixinState

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

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

□□□

MixinBlackboard

ServiceProvider

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

□□□

IServiceProvider

□□

HasGodotClassMethod(in godot_string_name)

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

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

□□

`method` godot_string_name

Name of the method to check for.

□□

bool

InvokeGodotClassMethod(in godot_string_name, NativeVariantPtrArgs, out godot_variant)

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

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

□□

`method` godot_string_name

Name of the method to invoke.

`args` NativeVariantPtrArgs

Arguments to use with the invoked method.

`ret` godot_variant

Value returned by the invoked method.

□□

bool

OnReady()

Notification received when the node is ready.

```
public void OnReady()
```

RestoreGodotObjectData(GodotSerializationInfo)

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

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

□□

`info` GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

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

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

□□

`info` GodotSerializationInfo

Object used to save the data.

Value()

Value that is provided by the provider.

```
public IServiceProvider Value()
```

□□

IServiceProvider

_Notification(int)

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

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

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

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

□□

`what` int

Game.MetatypeMetadata □

□□□□: ProjectStack

□□□: ProjectStack.dll

```
public class Game.MetatypeMetadata
```

□□

object ← Game.MetatypeMetadata

□□

Attributes

Attributes applied to the type itself.

```
public IReadOnlyDictionary<Type, Attribute[]> Attributes { get; }
```

□□□

IReadOnlyDictionary<Type, Attribute[]>

HasInitProperties

True if the type has init-only properties that must be set at construction. If this is true for a concrete type, you may call Construct(IReadOnlyDictionary<string, object>) with a map of argument names to values to set these properties at construction.

```
public bool HasInitProperties { get; }
```

□□□

bool

MixinHandlers

Map of mixin handler invocation functions by mixin type.

```
public IReadOnlyDictionary<Type, Action<object>> MixinHandlers { get; }
```

□□□

IReadOnlyDictionary<Type, Action<object>>

Mixins

List of mixins applied to the type, in the order that they were applied.

```
public IReadOnlyList<Type> Mixins { get; }
```

□□□

IReadOnlyList<Type>

Properties

Properties on the type. Only non-partial properties marked with attributes on the current type are included. To get all of the properties, including the inherited properties from any base metatypes, see the GetProperties(Type) method.

```
public IReadOnlyList<PropertyMetadata> Properties { get; }
```

□□□

IReadOnlyList<PropertyMetadata>

Type

System type of the introspective type.

```
public Type Type { get; }
```

□□□

Type

□□

Construct(IReadOnlyDictionary<string, object?>?)

Constructs the type with the given arguments, if any. If the type is not a concrete type, this throws. If the type has init-only properties, this can be used to set them at construction.

```
public object Construct(IReadOnlyDictionary<string, object?>? args = null)
```

□□

`args` IReadOnlyDictionary<string, object>

Map of argument names to values.

□□

object

A new instance of the type.

Equals(object)

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

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

□□

`obj` object

The object to compare with the current object.

□□

bool

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

GetHashCode()

Serves as the default hash function.

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

□□

int

A hash code for the current object.

Game.MethodName □

□□□□: ProjectStack

□□□: ProjectStack.dll

Cached StringNames for the methods contained in this class, for fast lookup.

```
public class Game.MethodName : Node2D.MethodName
```

□□

object ← GodotObject.MethodName ← Node.MethodName ← CanvasItem.MethodName ← Node2D.MethodName ← Game.MethodName

□□

OnReady

Cached name for the 'OnReady' method.

```
public static readonly StringName OnReady
```

□□□

StringName

_Notification

Cached name for the '_Notification' method.

```
public static readonly StringName _Notification
```

□□□

StringName

Game.PropertyName □

□□□□: ProjectStack

□□□: ProjectStack.dll

Cached StringNames for the properties and fields contained in this class, for fast lookup.

```
public class Game.PropertyName : Node2D.PropertyName
```

□□

object ← GodotObject.PropertyName ← Node.PropertyName ← CanvasItem.PropertyName ← Node2D.PropertyName ← Game.PropertyName

Game.SignalName □

□□□□: ProjectStack

□□□: ProjectStack.dll

Cached StringNames for the signals contained in this class, for fast lookup.

```
public class Game.SignalName : Node2D.SignalName
```

□□

object ← GodotObject.SignalName ← Node.SignalName ← CanvasItem.SignalName ← Node2D.SignalName ← Game.SignalName

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.GodotObject() may be used. See also Godot.Node._EnterTree().

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

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

Main.MethodName □

□□□□: [ProjectStack]

□□□: ProjectStack.dll

Cached StringNames for the methods contained in this class, for fast lookup.

```
public class Main.MethodName : Node2D.MethodName
```

□□

[object] ← GodotObject.MethodName ← Node.MethodName ← CanvasItem.MethodName ← Node2D.MethodName ← Main.MethodName

□□

RunScene

Cached name for the 'RunScene' method.

```
public static readonly StringName RunScene
```

□□□

StringName

_Ready

Cached name for the '_Ready' method.

```
public static readonly StringName _Ready
```

□□□

StringName

Main.PropertyName □

□□□□: ProjectStack

□□□: ProjectStack.dll

Cached StringNames for the properties and fields contained in this class, for fast lookup.

```
public class Main.PropertyName : Node2D.PropertyName
```

□□

object ← GodotObject.PropertyName ← Node.PropertyName ← CanvasItem.PropertyName ← Node2D.PropertyName ← Main.PropertyName

Main.SignalName □

□□□□: ProjectStack

□□□: ProjectStack.dll

Cached StringNames for the signals contained in this class, for fast lookup.

```
public class Main.SignalName : Node2D.SignalName
```

□□

object ← GodotObject.SignalName ← Node.SignalName ← CanvasItem.SignalName ← Node2D.SignalName ← Main.SignalName

ProjectStack.Command □□□□□

CommandAdapter

CommandAdapter.MetatypeMetadata

CommandAdapter.MethodName

Cached StringNames for the methods contained in this class, for fast lookup.

CommandAdapter.PropertyName

Cached StringNames for the properties and fields contained in this class, for fast lookup.

CommandAdapter.SignalName

Cached StringNames for the signals contained in this class, for fast lookup.

CommandAdapter □

□□□□: ProjectStack.Command

□□□: ProjectStack.dll

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

□□

object ← GodotObject ← Node ← CommandAdapter

□□

Metatype

Generated metatype information.

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

□□□

IMetatype

MixinState

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

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

□□□

MixinBlackboard

□□

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

RestoreGodotObjectData(GodotSerializationInfo)

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

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

□□

`info` GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

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

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

□□

`info` GodotSerializationInfo

Object used to save the data.

CommandAdapter.MetatypeMetadata □

□□□□: ProjectStack.Command

□□□: ProjectStack.dll

```
public class CommandAdapter.MetatypeMetadata
```

□□

object ← CommandAdapter.MetatypeMetadata

□□

Attributes

Attributes applied to the type itself.

```
public IReadOnlyDictionary<Type, Attribute[]> Attributes { get; }
```

□□□

IReadOnlyDictionary<Type, Attribute[]>

HasInitProperties

True if the type has init-only properties that must be set at construction. If this is true for a concrete type, you may call Construct(IReadOnlyDictionary<string, object>) with a map of argument names to values to set these properties at construction.

```
public bool HasInitProperties { get; }
```

□□□

bool

MixinHandlers

Map of mixin handler invocation functions by mixin type.

```
public IReadOnlyDictionary<Type, Action<object>> MixinHandlers { get; }
```

□□□

IReadOnlyDictionary<Type, Action<object>>

Mixins

List of mixins applied to the type, in the order that they were applied.

```
public IReadOnlyList<Type> Mixins { get; }
```

□□□

IReadOnlyList<Type>

Properties

Properties on the type. Only non-partial properties marked with attributes on the current type are included. To get all of the properties, including the inherited properties from any base metatypes, see the GetProperties(Type) method.

```
public IReadOnlyList<PropertyMetadata> Properties { get; }
```

□□□

IReadOnlyList<PropertyMetadata>

Type

System type of the introspective type.

```
public Type Type { get; }
```

□□□

Type

□□

Construct(IReadOnlyDictionary<string, object?>?)

Constructs the type with the given arguments, if any. If the type is not a concrete type, this throws. If the type has init-only properties, this can be used to set them at construction.

```
public object Construct(IReadOnlyDictionary<string, object?>? args = null)
```

□□

`args` IReadOnlyDictionary<string, object>

Map of argument names to values.

□□

object

A new instance of the type.

Equals(object)

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

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

□□

`obj` object

The object to compare with the current object.

□□

bool

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

GetHashCode()

Serves as the default hash function.

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

□□

int

A hash code for the current object.

CommandAdapter.MethodName □

□□□□: ProjectStack.Command

□□□: ProjectStack.dll

Cached StringNames for the methods contained in this class, for fast lookup.

```
public class CommandAdapter.MethodName : Node.MethodName
```

□□

object ← GodotObject.MethodName ← Node.MethodName ← CommandAdapter.MethodName

□□

CreateNewCard

Cached name for the 'CreateNewCard' method.

```
public static readonly StringName CreateNewCard
```

□□□

StringName

CreateNewCardStack

Cached name for the 'CreateNewCardStack' method.

```
public static readonly StringName CreateNewCardStack
```

□□□

StringName

GetRegisteredCardMetaIds

Cached name for the 'GetRegisteredCardMetaIds' method.

```
public static readonly StringName GetRegisteredCardMetaIds
```

□□□

StringName

CommandAdapter.PropertyName □

□□□□: ProjectStack.Command

□□□: ProjectStack.dll

Cached StringNames for the properties and fields contained in this class, for fast lookup.

```
public class CommandAdapter.PropertyName : Node.PropertyName
```

□□

object ← GodotObject.PropertyName ← Node.PropertyName ←
CommandAdapter.PropertyName

CommandAdapter.SignalName □

□□□□: ProjectStack.Command

□□□: ProjectStack.dll

Cached StringNames for the signals contained in this class, for fast lookup.

```
public class CommandAdapter.SignalName : Node.SignalName
```

□□

object ← GodotObject.SignalName ← Node.SignalName ← CommandAdapter.SignalName

ProjectStack.Common.Card □□□□

□

CardMeta

CardMetaRegistrationHelper

CardStack

□□

ICardStack

CardMeta □

□□□□: ProjectStack.Common.Card

□□□: ProjectStack.dll

```
public record CardMeta
```

□□

object ← CardMeta

□□□□

CardMeta(ResourceLocation, string, string, ResourceLocation)

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

□□

`Id` ResourceLocation

`Name` string

`Description` string

`Type` ResourceLocation

□□

Description

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

□□□

string

Id

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

□□□

ResourceLocation

Name

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

□□□

string

Type

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

□□□

ResourceLocation

□□

Create(string)

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

□□

`id` string

□□

CardMeta

CardMetaRegistrationHelper □

□□□□: ProjectStack.Common.Card

□□□: ProjectStack.dll

```
public class CardMetaRegistrationHelper
```

□□

object ← CardMetaRegistrationHelper

□□□□

CardMetaRegistrationHelper(IServiceCollection)

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

□□

`services` IServiceCollection

□□

Add(CardMeta)

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

□□

`cardMeta` CardMeta

□□

CardMetaRegistrationHelper

End()

```
public IServiceCollection End()
```

□□

IServiceCollection

CardStack □

□□□□: ProjectStack.Common.Card

□□□: ProjectStack.dll

```
public record CardStack : ICardStack
```

□□

object ← CardStack

□□

ICardStack

□□□□

CardStack(IImmutableList<Card>)

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

□□

`Cards` IImmutableList<Card>

□□

Cards

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

□□□

IImmutableList<Card>

ICardStack □□

□□□□: ProjectStack.Common.Card

□□□: ProjectStack.dll

```
public interface ICardStack
```

□□

Cards

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

□□□

IImmutableList<Card>

ProjectStack.Common.Recipe □□□□

□

CardMateMatchRecipe

RecipeResult

□□

IRecipe

IRecipeInput

CardMateMatchRecipe □

□□□□: ProjectStack.Common.Recipe

□□□: ProjectStack.dll

```
public abstract class CardMateMatchRecipe : IRecipe
```

□□

object ← CardMateMatchRecipe

□□

IRecipe

□□□□

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

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

□□

`requiredCardMetas` IList<CardMeta>

`producedCardMetas` IList<CardMeta>

`isNeedOrder` bool

□□

CardViewCount

□□□□□□□□□□□□□`null`□□□□□□□

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

□□□

uint?

IsNeedOrder

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

□□□

bool

ProducedCardMetas

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

□□□

IList<CardMeta>

RequiredCardMetas

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

□□□

IList<CardMeta>

□□

Execute(ICardStack)

□□□□□□□

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

[illegible]

`cardStack` ICardStack

[illegible]

[illegible]

RecipeResult

[illegible]

IRecipe □□

□□□□: ProjectStack.Common.Recipe

□□□: ProjectStack.dll

```
public interface IRecipe
```

□□

CardViewCount

□□□□□□□□□□□□□`null`□□□□□□□

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

□□□

uint?

□□

Execute(ICardStack)

□□□□□□□

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

□□

`cardStack` ICardStack

□□□□□□□□□

□□

RecipeResult

[illegible]

IRecipeInput □□

□□□□: ProjectStack.Common.Recipe

□□□: ProjectStack.dll

```
public interface IRecipeInput
```

RecipeResult □

□□□□: ProjectStack.Common.Recipe

□□□: ProjectStack.dll

```
public record RecipeResult
```

□□

object ← RecipeResult

□□□□

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

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

□□

`IsMatch` bool

`ConsumedCards` IEnumerable<Card>

`ProducedCards` IEnumerable<Card>

□□

ConsumedCards

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

□□□

IEnumerable<Card>

IsMatch

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

□□□

bool

ProducedCards

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

□□□

IEnumerable<Card>

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.MetatypeMetadata

Card.MethodName

Cached StringNames for the methods contained in this class, for fast lookup.

Card.PropertyName

Cached StringNames for the properties and fields contained in this class, for fast lookup.

Card.SignalName

Cached StringNames for the signals contained in this class, for fast lookup.

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.MetatypeMetadata

InfoTab.MethodName

Cached StringNames for the methods contained in this class, for fast lookup.

InfoTab.PropertyName

Cached StringNames for the properties and fields contained in this class, for fast lookup.

InfoTab.SignalName

Cached StringNames for the signals contained in this class, for fast lookup.

InfoTab.TaskNotifier

A wrapping class that can hold a Task value.

InfoTab.TaskNotifier<T>

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

Card □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

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

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

□□

object ← GodotObject ← Node ← CanvasItem ← Node2D ← Card

□□

INtjObject

□□□□

Card()

```
public Card()
```

□□

BottomCard

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

□□□

Card

BottomCards

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

□□□

IImmutableList<Card>

CardMeta

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

□□□

CardMeta

CardNameLabel

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

□□□

Label

CardStack

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

□□□

ICardStack

CharacterBody

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

□□□

CharacterBody2D

CurrentStack

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

□□□

IImmutableList<Card>

IsRoot

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

□□□

bool

IsUppest

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

□□□

bool

Metatype

Generated metatype information.

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

□□□

IMetatype

MixinState

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

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

□□□

MixinBlackboard

Ntj

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

□□□

JsonObject

OnDrag

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

□□□

bool

Panel

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

□□□

Control

RootCard

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

□□□

Card

TextureRect

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

□□□

TextureRect

TopCard

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

□□□

Card

TopCards

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

□□□

IImmutableList<Card>

UppestCard

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

□□□

Card

□□

GetGodotClassPropertyValue(in godot_string_name, out godot_variant)

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

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

□□

`name` godot_string_name

Name of the property to get.

`value` godot_variant

Value of the property if it was found.

□□

bool

true if a property with the given name was found.

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

OnCardAreaEntered(Area2D)

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

□□

`body` Area2D

OnCardAreaExited(Area2D)

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

□□

`body` Area2D

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()
```

RefreshTexture()

```
public void RefreshTexture()
```

RestoreGodotObjectData(GodotSerializationInfo)

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

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

□□

`info` GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

Saves this instance's state to be restored when reloading assemblies. Do not call or override this method. To add data to be saved and restored, implement Godot.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 PropertyChanged again for the target property when it completes. This can be used to update bindings observing that Task or any of its properties. This method and its overload specifically rely on the Card.TaskNotifier 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 PropertyChanged again for the target property when it completes. This can be used to update bindings observing that Task or any of its properties. This method and its overload specifically rely on the Card.TaskNotifier<T> type, which needs to be used in the backing field for the target Task property. The field doesn't need to be initialized, as this method will take care of doing that automatically. The Card.TaskNotifier<T> type also includes an implicit operator, so it can be assigned to any Task instance directly. Here is a sample property declaration using this method:

```
private TaskNotifier<int> myTask;

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

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

□□

`taskNotifier` Card.TaskNotifier<T>

The field notifier to modify.

`newValue` Task<T>

The property's value after the change occurred.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

true if the property was changed, false otherwise.

□□□□

`T`

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

□□

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

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

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

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

□□

`oldValue` T

The current property value.

`newValue` T

The property's value after the change occurred.

`callback` Action<T>

A callback to invoke to update the property value.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

true if the property was changed, false otherwise.

□□□□

`T`

The type of the property that changed.

□□

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

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

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

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

□□

`oldValue` T

The current property value.

`newValue` T

The property's value after the change occurred.

`comparer` IEqualityComparer<T>

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

`callback` Action<T>

A callback to invoke to update the property value.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

true if the property was changed, false otherwise.

□□□□

`T`

The type of the property that changed.

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

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

PropertyChanged event. See additional notes about this overload in SetProperty<T>(ref T, T, string?).

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

□□

`field` T

The field storing the property's value.

`newValue` T

The property's value after the change occurred.

`comparer` IEqualityComparer<T>

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

`propertyName` string

(optional) The name of the property that changed.

□□

bool

true if the property was changed, false otherwise.

□□□□

`T`

The type of the property that changed.

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

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

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

□□

`field` T

The field storing the property's value.

`newValue` T

The property's value after the change occurred.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

true if the property was changed, false otherwise.

□□□□

`T`

The type of the property that changed.

□□

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

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

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

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

□□

`oldValue` T

The current property value.

`newValue` T

The property's value after the change occurred.

`comparer` IEqualityComparer<T>

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

`model` TModel

The model containing the property being updated.

`callback` Action<TModel, T>

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

`propertyName` string

(optional) The name of the property that changed.

□□

bool

true if the property was changed, false otherwise.

□□□□

`TModel`

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

`T`

The type of property (or field) to set.

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

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

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

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

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

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

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

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

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

□□

`oldValue` T

The current property value.

`newValue` T

The property's value after the change occurred.

`model` TModel

The model containing the property being updated.

`callback` Action<TModel, T>

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

`propertyName` string

(optional) The name of the property that changed.

□□

bool

true if the property was changed, false otherwise.

□□□□

`TModel`

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

`T`

The type of property (or field) to set.

□□

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

UpdateAllZIndex()

```
public void UpdateAllZIndex()
```

UpdateRecipe()

```
public void UpdateRecipe()
```

UpdateZIndex()

```
public void UpdateZIndex()
```

_Notification(int)

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

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

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

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

□□

`what` int

_Process(double)

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

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

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

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

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

□□

`delta` double

_Ready()

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

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

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

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

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

□□

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.MetatypeMetadata □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

```
public class Card.MetatypeMetadata
```

□□

object ← Card.MetatypeMetadata

□□

Attributes

Attributes applied to the type itself.

```
public IReadOnlyDictionary<Type, Attribute[]> Attributes { get; }
```

□□□

IReadOnlyDictionary<Type, Attribute[]>

HasInitProperties

True if the type has init-only properties that must be set at construction. If this is true for a concrete type, you may call Construct(IReadOnlyDictionary<string, object>) with a map of argument names to values to set these properties at construction.

```
public bool HasInitProperties { get; }
```

□□□

bool

MixinHandlers

Map of mixin handler invocation functions by mixin type.

```
public IReadOnlyDictionary<Type, Action<object>> MixinHandlers { get; }
```

□□□

IReadOnlyDictionary<Type, Action<object>>

Mixins

List of mixins applied to the type, in the order that they were applied.

```
public IReadOnlyList<Type> Mixins { get; }
```

□□□

IReadOnlyList<Type>

Properties

Properties on the type. Only non-partial properties marked with attributes on the current type are included. To get all of the properties, including the inherited properties from any base metatypes, see the GetProperties(Type) method.

```
public IReadOnlyList<PropertyMetadata> Properties { get; }
```

□□□

IReadOnlyList<PropertyMetadata>

Type

System type of the introspective type.

```
public Type Type { get; }
```

□□□

Type

□□

Construct(IReadOnlyDictionary<string, object?>?)

Constructs the type with the given arguments, if any. If the type is not a concrete type, this throws. If the type has init-only properties, this can be used to set them at construction.

```
public object Construct(IReadOnlyDictionary<string, object?>? args = null)
```

□□

`args` IReadOnlyDictionary<string, object>

Map of argument names to values.

□□

object

A new instance of the type.

Equals(object)

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

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

□□

`obj` object

The object to compare with the current object.

□□

bool

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

GetHashCode()

Serves as the default hash function.

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

□□

int

A hash code for the current object.

Card.MethodName □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

Cached StringNames for the methods contained in this class, for fast lookup.

```
public class Card.MethodName : Node2D.MethodName
```

□□

object ← GodotObject.MethodName ← Node.MethodName ← CanvasItem.MethodName ← Node2D.MethodName ← Card.MethodName

□□

AddCard2Group

Cached name for the 'AddCard2Group' method.

```
public static readonly StringName AddCard2Group
```

□□□

StringName

OnCardAreaEntered

Cached name for the 'OnCardAreaEntered' method.

```
public static readonly StringName OnCardAreaEntered
```

□□□

StringName

OnCardAreaExited

Cached name for the 'OnCardAreaExited' method.

```
public static readonly StringName OnCardAreaExited
```

□□□

StringName

OnMouseEntered

Cached name for the 'OnMouseEntered' method.

```
public static readonly StringName OnMouseEntered
```

□□□

StringName

OnMouseExited

Cached name for the 'OnMouseExited' method.

```
public static readonly StringName OnMouseExited
```

□□□

StringName

OnReady

Cached name for the 'OnReady' method.

```
public static readonly StringName OnReady
```

□□□

StringName

RefreshTexture

Cached name for the 'RefreshTexture' method.

```
public static readonly StringName RefreshTexture
```

□□□

StringName

UpdateAllZIndex

Cached name for the 'UpdateAllZIndex' method.

```
public static readonly StringName UpdateAllZIndex
```

□□□

StringName

UpdateRecipe

Cached name for the 'UpdateRecipe' method.

```
public static readonly StringName UpdateRecipe
```

□□□

StringName

UpdateState

Cached name for the 'UpdateState' method.

```
public static readonly StringName UpdateState
```

□□□

StringName

UpdateZIndex

Cached name for the 'UpdateZIndex' method.

```
public static readonly StringName UpdateZIndex
```

□□□

StringName

_Notification

Cached name for the '_Notification' method.

```
public static readonly StringName _Notification
```

□□□

StringName

_Process

Cached name for the '_Process' method.

```
public static readonly StringName _Process
```

□□□

StringName

_Ready

Cached name for the '_Ready' method.

```
public static readonly StringName _Ready
```

□□□

StringName

Card.PropertyName □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

Cached StringNames for the properties and fields contained in this class, for fast lookup.

```
public class Card.PropertyName : Node2D.PropertyName
```

□□

object ← GodotObject.PropertyName ← Node.PropertyName ← CanvasItem.PropertyName ← Node2D.PropertyName ← Card.PropertyName

□□

BottomCard

Cached name for the 'BottomCard' property.

```
public static readonly StringName BottomCard
```

□□□

StringName

CardNameLabel

Cached name for the 'CardNameLabel' property.

```
public static readonly StringName CardNameLabel
```

□□□

StringName

CharacterBody

Cached name for the 'CharacterBody' property.

```
public static readonly StringName CharacterBody
```

□□□

StringName

IsRoot

Cached name for the 'IsRoot' property.

```
public static readonly StringName IsRoot
```

□□□

StringName

IsUppest

Cached name for the 'IsUppest' property.

```
public static readonly StringName IsUppest
```

□□□

StringName

OnDrag

Cached name for the 'OnDrag' property.

```
public static readonly StringName OnDrag
```

□□□

StringName

Panel

Cached name for the 'Panel' property.

```
public static readonly StringName Panel
```

□□□

StringName

RootCard

Cached name for the 'RootCard' property.

```
public static readonly StringName RootCard
```

□□□

StringName

TextureRect

Cached name for the 'TextureRect' property.

```
public static readonly StringName TextureRect
```

□□□

StringName

TopCard

Cached name for the 'TopCard' property.

```
public static readonly StringName TopCard
```

□□□

StringName

UppestCard

Cached name for the 'UppestCard' property.

```
public static readonly StringName UppestCard
```

□□□

StringName

_bottomCard

Cached name for the '_bottomCard' field.

```
public static readonly StringName _bottomCard
```

□□□

StringName

_cardNameLabel

Cached name for the '_cardNameLabel' field.

```
public static readonly StringName _cardNameLabel
```

□□□

StringName

_characterBody

Cached name for the '_characterBody' field.

```
public static readonly StringName _characterBody
```

□□□

StringName

_game

Cached name for the '_game' field.

```
public static readonly StringName _game
```

□□□

StringName

_isRoot

Cached name for the '_isRoot' field.

```
public static readonly StringName _isRoot
```

□□□

StringName

_isUppest

Cached name for the '_isUppest' field.

```
public static readonly StringName _isUppest
```

□□□

StringName

_panel

Cached name for the '_panel' field.

```
public static readonly StringName _panel
```

□□□

StringName

_ready2Group

Cached name for the '_ready2Group' field.

```
public static readonly StringName _ready2Group
```

□□□

StringName

_textureRect

Cached name for the '_textureRect' field.

```
public static readonly StringName _textureRect
```

□□□

StringName

_topCard

Cached name for the '_topCard' field.

```
public static readonly StringName _topCard
```

▯▯▯

StringName

Card.SignalName □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

Cached StringNames for the signals contained in this class, for fast lookup.

```
public class Card.SignalName : Node2D.SignalName
```

□□

object ← GodotObject.SignalName ← Node.SignalName ← CanvasItem.SignalName ← Node2D.SignalName ← Card.SignalName

Card.TaskNotifier □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

A wrapping class that can hold a Task value.

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

□□

object ← Card.TaskNotifier

□□□

implicit operator Task?(TaskNotifier?)

Unwraps the Task value stored in the current instance.

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

□□

`notifier` Card.TaskNotifier

The input Card.TaskNotifier<T> instance.

□□

Task

Card.TaskNotifier<T> □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

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

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

□□□□

`T`

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

□□

object ← Card.TaskNotifier<T>

□□□

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

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

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

□□

`notifier` Card.TaskNotifier<T>

The input Card.TaskNotifier<T> instance.

□□

Task<T>

InfoTab □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

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

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

User Interface nodes and input

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

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

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

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

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

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

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

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

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

□□

object ← GodotObject ← Node ← CanvasItem ← Control ← InfoTab

□□

Metatype

Generated metatype information.

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

□□□

IMetatype

MixinState

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

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

□□□

MixinBlackboard

□□

GetGodotClassPropertyValue(in godot_string_name, out godot_variant)

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

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

□□

`name` godot_string_name

Name of the property to get.

`value` godot_variant

Value of the property if it was found.

□□

bool

true if a property with the given name was found.

HasGodotClassMethod(in godot_string_name)

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

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

□□

`method` godot_string_name

Name of the method to check for.

□□

bool

InvokeGodotClassMethod(in godot_string_name, NativeVariantPtrArgs, out godot_variant)

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

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

□□

`method` godot_string_name

Name of the method to invoke.

`args` NativeVariantPtrArgs

Arguments to use with the invoked method.

`ret` godot_variant

Value returned by the invoked method.

□□

bool

OnProcess(double)

Notification received from the tree every rendered frame when Godot.Node.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.ISerializationListener.

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

□□

`info` GodotSerializationInfo

Object used to save the data.

SetGodotClassPropertyValue(in godot_string_name, in godot_variant)

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

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

□□

`name` godot_string_name

Name of the property to set.

`value` godot_variant

Value to set the property to if it was found.

□□

bool

true if a property with the given name was found.

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

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

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

□□

`taskNotifier` InfoTab.TaskNotifier

The field notifier to modify.

`newValue` Task

The property's value after the change occurred.

`callback` Action<Task>

A callback to invoke to update the property value.

`propertyName` string

(optional) The name of the property that changed.

□□

bool

true if the property was changed, false otherwise.

□□

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

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

Compares the current and new values for a given field (which should be the backing field for a property). If the value has changed, raises the PropertyChanging event, updates the field and then raises the PropertyChanged event. The behavior mirrors that of SetProperty<T>(ref T, T, string?), with the difference being that this method will also monitor the new value of the property (a generic Task) and will also raise the 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)
```

□□

`what` int

□□

PropertyChanged

Occurs when a property value changes.

```
public event PropertyChangedEventHandler? PropertyChanged
```

□□□□

PropertyChangedEventHandler

PropertyChanging

Occurs when a property value is changing.

```
public event PropertyChangingEventHandler? PropertyChanging
```

□□□□

PropertyChangingEventHandler

InfoTab.MetatypeMetadata □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

```
public class InfoTab.MetatypeMetadata
```

□□

object ← InfoTab.MetatypeMetadata

□□

Attributes

Attributes applied to the type itself.

```
public IReadOnlyDictionary<Type, Attribute[]> Attributes { get; }
```

□□□

IReadOnlyDictionary<Type, Attribute[]>

HasInitProperties

True if the type has init-only properties that must be set at construction. If this is true for a concrete type, you may call Construct(IReadOnlyDictionary<string, object>) with a map of argument names to values to set these properties at construction.

```
public bool HasInitProperties { get; }
```

□□□

bool

MixinHandlers

Map of mixin handler invocation functions by mixin type.

```
public IReadOnlyDictionary<Type, Action<object>> MixinHandlers { get; }
```

□□□

IReadOnlyDictionary<Type, Action<object>>

Mixins

List of mixins applied to the type, in the order that they were applied.

```
public IReadOnlyList<Type> Mixins { get; }
```

□□□

IReadOnlyList<Type>

Properties

Properties on the type. Only non-partial properties marked with attributes on the current type are included. To get all of the properties, including the inherited properties from any base metatypes, see the GetProperties(Type) method.

```
public IReadOnlyList<PropertyMetadata> Properties { get; }
```

□□□

IReadOnlyList<PropertyMetadata>

Type

System type of the introspective type.

```
public Type Type { get; }
```

□□□

Type

□□

Construct(IReadOnlyDictionary<string, object?>?)

Constructs the type with the given arguments, if any. If the type is not a concrete type, this throws. If the type has init-only properties, this can be used to set them at construction.

```
public object Construct(IReadOnlyDictionary<string, object?>? args = null)
```

□□

`args` IReadOnlyDictionary<string, object>

Map of argument names to values.

□□

object

A new instance of the type.

Equals(object)

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

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

□□

`obj` object

The object to compare with the current object.

□□

bool

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

GetHashCode()

Serves as the default hash function.

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

□□

int

A hash code for the current object.

InfoTab.MethodName □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

Cached StringNames for the methods contained in this class, for fast lookup.

```
public class InfoTab.MethodName : Control.MethodName
```

□□

object ← GodotObject.MethodName ← Node.MethodName ← CanvasItem.MethodName ← Control.MethodName ← InfoTab.MethodName

□□

CreateTextBlock

Cached name for the 'CreateTextBlock' method.

```
public static readonly StringName CreateTextBlock
```

□□□

StringName

InitEvents

Cached name for the 'InitEvents' method.

```
public static readonly StringName InitEvents
```

□□□

StringName

OnProcess

Cached name for the 'OnProcess' method.

```csharp
public static readonly StringName OnProcess
```

□□□

StringName

OnReady

Cached name for the 'OnReady' method.

```csharp
public static readonly StringName OnReady
```

□□□

StringName

SetTextBlockText

Cached name for the 'SetTextBlockText' method.

```csharp
public static readonly StringName SetTextBlockText
```

□□□

StringName

UpdateInfoTexts

Cached name for the 'UpdateInfoTexts' method.

```csharp
public static readonly StringName UpdateInfoTexts
```

□□□

StringName

_Notification

Cached name for the '_Notification' method.

```
public static readonly StringName _Notification
```

□□□

StringName

InfoTab.PropertyName □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

Cached StringNames for the properties and fields contained in this class, for fast lookup.

```
public class InfoTab.PropertyName : Control.PropertyName
```

□□

object ← GodotObject.PropertyName ← Node.PropertyName ← CanvasItem.PropertyName ← Control.PropertyName ← InfoTab.PropertyName

□□

Game

Cached name for the 'Game' property.

```
public static readonly StringName Game
```

□□□

StringName

_initialized

Cached name for the '_initialized' field.

```
public static readonly StringName _initialized
```

□□□

StringName

InfoTab.SignalName □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

Cached StringNames for the signals contained in this class, for fast lookup.

```
public class InfoTab.SignalName : Control.SignalName
```

□□

object ← GodotObject.SignalName ← Node.SignalName ← CanvasItem.SignalName ← Control.SignalName ← InfoTab.SignalName

InfoTab.TaskNotifier □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

A wrapping class that can hold a Task value.

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

□□

object ← InfoTab.TaskNotifier

□□□

implicit operator Task?(TaskNotifier?)

Unwraps the Task value stored in the current instance.

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

□□

`notifier` InfoTab.TaskNotifier

The input InfoTab.TaskNotifier<T> instance.

□□

Task

InfoTab.TaskNotifier<T> □

□□□□: ProjectStack.Component

□□□: ProjectStack.dll

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

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

□□□□

`T`

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

□□

object ← InfoTab.TaskNotifier<T>

□□□

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

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

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

□□

`notifier` InfoTab.TaskNotifier<T>

The input InfoTab.TaskNotifier<T> instance.

□□

Task<T>

ProjectStack.Core □□□□

□

CardMgr

CardMgr.MethodName

Cached StringNames for the methods contained in this class, for fast lookup.

CardMgr.PropertyName

Cached StringNames for the properties and fields contained in this class, for fast lookup.

CardMgr.SignalName

Cached StringNames for the signals contained in this class, for fast lookup.

ServiceCollectionExtension

CardMgr □

□□□□: ProjectStack.Core

□□□: ProjectStack.dll

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

□□

object ← GodotObject ← Node ← CardMgr

□□

RestoreGodotObjectData(GodotSerializationInfo)

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

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

□□

`info` GodotSerializationInfo

Object that contains the previously saved data.

SaveGodotObjectData(GodotSerializationInfo)

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

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

□□

`info` GodotSerializationInfo

Object used to save the data.

CardMgr.MethodName □

□□□□: ProjectStack.Core

□□□: ProjectStack.dll

Cached StringNames for the methods contained in this class, for fast lookup.

```
public class CardMgr.MethodName : Node.MethodName
```

□□

object ← GodotObject.MethodName ← Node.MethodName ← CardMgr.MethodName

CardMgr.PropertyName □

□□□□: ProjectStack.Core

□□□: ProjectStack.dll

Cached StringNames for the properties and fields contained in this class, for fast lookup.

```
public class CardMgr.PropertyName : Node.PropertyName
```

□□

object ← GodotObject.PropertyName ← Node.PropertyName ← CardMgr.PropertyName

CardMgr.SignalName □

□□□□: ProjectStack.Core

□□□: ProjectStack.dll

Cached StringNames for the signals contained in this class, for fast lookup.

```
public class CardMgr.SignalName : Node.SignalName
```

□□

object ← GodotObject.SignalName ← Node.SignalName ← CardMgr.SignalName

ServiceCollectionExtension ▯

▯▯▯▯: ProjectStack.Core

▯▯▯: ProjectStack.dll

```
public static class ServiceCollectionExtension
```

▯▯

object ← ServiceCollectionExtension

▯▯

ConfigureHoveredItemInfoDisplay(IServiceCollection)

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

▯▯

`services` IServiceCollection

▯▯

HoveredItemInfoDisplayRegistrationHelper

RegisterCardMetas(IServiceCollection)

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

▯▯

`services` IServiceCollection

▯▯

CardMetaRegistrationHelper

RegisterTextureLoader(IServiceCollection)

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

□□

`services` IServiceCollection

□□

TextureLoader

ProjectStack.NamedTagsBaseOnJson □□□□

□□

INtjObject

INtjObject □□

□□□□: ProjectStack.NamedTagsBaseOnJson

□□□: ProjectStack.dll

```
public interface INtjObject
```

□□

Ntj

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

□□□

JsonObject

ProjectStack.Resource □□□□

□

TextureLoader

TextureLoader □

□□□□: ProjectStack.Resource

□□□: ProjectStack.dll

```
public class TextureLoader
```

□□

object ← TextureLoader

□□□□

TextureLoader(IServiceCollection)

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

□□

`services` IServiceCollection

□□

AddTextureDirectory(string)

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

□□

`directoryPath` string

□□

TextureLoader

End()

```
public IServiceCollection End()
```

□□

IServiceCollection

ProjectStack.UserInterface □□□□

□

HoveredItemInfoDisplay

HoveredItemInfoDisplayRegistrationHelper

HoveredItemInfoProvider

HoveredItemInfoDisplay □

□□□□: ProjectStack.UserInterface

□□□: ProjectStack.dll

```csharp
public class HoveredItemInfoDisplay
```

□□

object ← HoveredItemInfoDisplay

□□□□

HoveredItemInfoDisplay(IServiceCollection)

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

□□

`services` IServiceCollection

□□

HoveredItemInfoProviders

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

□□□

List<HoveredItemInfoProvider>

HoveredItemInfoTexts

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

□□□

IImmutableList<string>

HoveredItems

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

□□□

ObservableCollection<Node>

HoveredItemInfoDisplayRegistrationHelper □

□□□□: ProjectStack.UserInterface

□□□: ProjectStack.dll

```
public class HoveredItemInfoDisplayRegistrationHelper
```

□□

object ← HoveredItemInfoDisplayRegistrationHelper

□□□□

HoveredItemInfoDisplayRegistrationHelper(IServiceCollection)

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

□□

`services` IServiceCollection

□□

End()

```
public IServiceCollection End()
```

□□

IServiceCollection

RegisterHoveredItemInfoProvider(HoveredItemInfoProvider)

```
public HoveredItemInfoDisplayRegistrationHelper
RegisterHoveredItemInfoProvider(HoveredItemInfoProvider hoveredItemInfoProvider)
```

□□

`hoveredItemInfoProvider` HoveredItemInfoProvider

□□

HoveredItemInfoDisplayRegistrationHelper

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

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

□□

`providerName` string

`predicate` Predicate<HoveredItemInfoDisplay>

`displayTextProvider` Func<HoveredItemInfoDisplay, string>

□□

HoveredItemInfoDisplayRegistrationHelper

HoveredItemInfoProvider □

□□□□: ProjectStack.UserInterface

□□□: ProjectStack.dll

```
public class HoveredItemInfoProvider
```

□□

object ← HoveredItemInfoProvider

□□□□

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

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

□□

`providerName` string

`predicate` Predicate<HoveredItemInfoDisplay>

`displayTextProvider` Func<HoveredItemInfoDisplay, string>

□□

ProviderName

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

□□□

string

□□

ProvideHoveredItemInfo(HoveredItemInfoDisplay)

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

□□

`hoveredItemInfoDisplay` HoveredItemInfoDisplay

□□

string

ProjectStack.Util □□□□

□

FileSystemHelper

FileSystemHelper □

□□□□: ProjectStack.Util

□□□: ProjectStack.dll

```
public class FileSystemHelper
```

□□

object ← FileSystemHelper

□□

Default

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

□□□

FileSystemHelper

□□

GetAllFilesInDirectory(string)

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

□□

`directoryPath` string

□□

IEnumerable<string>

ProjectStack.src.scripts.Common □□□□

□

ResourceLocation

ResourceLocation □

□□□□: ProjectStack.src.scripts.Common

□□□: ProjectStack.dll

```
public record ResourceLocation
```

□□

object ← ResourceLocation

□□□□

ResourceLocation(string, string)

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

□□

`Namespace` string

`Path` string

□□

EMPTY

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

□□□

ResourceLocation

GetNamespace

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

□□□

string

GetPath

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

□□□

string

Namespace

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

□□□

string

Path

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

□□□

string

□□

DefaultNamespaceAndPath(string)

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

□□

`path` string

□□

ResourceLocation

FromNamespaceAndPath(string, string)

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

□□

`namespace` string

`path` string

□□

ResourceLocation

GetResLoc()

```
public string GetResLoc()
```

□□

string

GetResLocWithPath(string)

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

□□

`p` string

□□

string

GetResLocWithoutPrefix(string)

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

□□

`p` string

□□

string

Parse(string)

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

□□

`fullPath` string

□□

ResourceLocation

ToString()

Returns a string that represents the current object.

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

□□

string

A string that represents the current object.

ProjectStack.src.scripts.Common.Recipe □□□□

□

AbstractRecipe<TRecipeInput>

RecipeInput

RecipeOutput

ScriptRecipeInput

ScriptRecipeInput.ScriptContext

SimpleRecipe
  □□□□□□□□□

SimpleRecipe.Ingredient

SimpleRecipe.Product

AbstractRecipe<TRecipeInput> □

□□□□: ProjectStack.src.scripts.Common.Recipe

□□□: ProjectStack.dll

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

□□□□

`TRecipeInput`

□□

object ← AbstractRecipe<TRecipeInput>

□□

Assmble(TRecipeInput, JsonObject)

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

□□

`recipeInput` TRecipeInput

`ntj` JsonObject

□□

RecipeOutput

Matchs(TRecipeInput, JsonObject)

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

□□

`recipeInput` TRecipeInput

`ntj` JsonObject

□□

bool

Type()

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

□□

ResourceLocation

RecipeInput □

□□□□: ProjectStack.src.scripts.Common.Recipe

□□□: ProjectStack.dll

```
public abstract record RecipeInput
```

□□

object ← RecipeInput

Derived

ScriptRecipeInput

□□

Assemble(Card)

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

□□

`card` Card

□□

Card

IsMatch(Card)

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

□□

`card` Card

□□

bool

RecipeOutput □

□□□□: ProjectStack.src.scripts.Common.Recipe

□□□: ProjectStack.dll

```
public record RecipeOutput
```

□□

object ← RecipeOutput

□□□□

RecipeOutput(List<Card>, JsonObject)

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

□□

`Cards` List<Card>

`Ntj` JsonObject

□□

Cards

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

□□□

List<Card>

Ntj

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

□□□

JsonObject

ScriptRecipeInput □

□□□□: ProjectStack.src.scripts.Common.Recipe

□□□: ProjectStack.dll

```
public record ScriptRecipeInput : RecipeInput
```

□□

object ← RecipeInput ← ScriptRecipeInput

□□□□

ScriptRecipeInput()

```
public ScriptRecipeInput()
```

□□

AssembleScript

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

□□□

string

AssembleScriptRunner

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

□□□

ScriptRunner<Card>

MatchScript

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

□□□

string

MatchScriptRunner

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

□□□

ScriptRunner<bool>

□□

Assemble(Card)

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

□□

`card` Card

□□

Card

IsMatch(Card)

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

□□

`card` Card

□□

bool

ScriptRecipeInput.ScriptContext □

□□□□: ProjectStack.src.scripts.Common.Recipe

□□□: ProjectStack.dll

```
public class ScriptRecipeInput.ScriptContext
```

□□

object ← ScriptRecipeInput.ScriptContext

□□

card

```
public Card card
```

□□□

Card

SimpleRecipe □

□□□□: ProjectStack.src.scripts.Common.Recipe

□□□: ProjectStack.dll

□□□□□□□□□

```
public record SimpleRecipe
```

□□

object ← SimpleRecipe

□□□□

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

□□□□□□□□□

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

□□

`Id` ResourceLocation

□□□□□

`Name` string

`Description` string

`Production` float

□□□□□

`Ingredients` IImmutableList<SimpleRecipe.Ingredient>

□□

`Products` IImmutableList<SimpleRecipe.Product>

□□

□□

Description

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

□□□

string

Id

□□□□□

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

□□□

ResourceLocation

Ingredients

□□

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

□□□

IImmutableList<SimpleRecipe.Ingredient>

Name

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

□□□

string

Production

□□□□□

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

□□□

float

Products

□□

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

□□□

IImmutableList<SimpleRecipe.Product>

□□

SatisfactionCheck(IImmutableList<CardMeta>)

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

□□

`cards` IImmutableList<CardMeta>

□□

bool

SimpleRecipe.Ingredient □

□□□□: ProjectStack.src.scripts.Common.Recipe

□□□: ProjectStack.dll

```
public record SimpleRecipe.Ingredient
```

□□

object ← SimpleRecipe.Ingredient

□□□□

Ingredient(ResourceLocation, int, bool)

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

□□

`CardId` ResourceLocation

`Quantity` int

`Consumed` bool

□□

CardId

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

□□□

ResourceLocation

Consumed

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

□□□

bool

Quantity

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

□□□

int

SimpleRecipe.Product □

□□□□: ProjectStack.src.scripts.Common.Recipe

□□□: ProjectStack.dll

```
public record SimpleRecipe.Product
```

□□

object ← 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