

# Save Design Script Reference

## Table of Contents

1. [Interfaces](#)
    - 1.1 [IAfterInitializeCallback](#)
    - 1.2 [IAfterInitializeRollback](#)
    - 1.3 [IAfterLoadCallback](#)
    - 1.4 [IAfterLoadRollback](#)
    - 1.5 [IBeforeSaveCallback](#)
    - 1.6 [IBeforeSaveRollback](#)
    - 1.7 [ISaveDesignConfig](#)
  2. [Attributes](#)
    - 2.1 [EncryptorAttribute](#)
    - 2.2 [SaveDesignRootAttribute](#)
    - 2.3 [SharedDataAttribute](#)
    - 2.4 [SlotDataAttribute](#)
    - 2.5 [SlotMetaDataAttribute](#)
    - 2.6 [TempDataAttribute](#)
  3. [Enumerations](#)
    - 3.1 [ExceptionPolicy](#)
    - 3.2 [SerializerType](#)
    - 3.3 [TempDataResetTiming](#)
  4. [Classes](#)
    - 4.1 [Class with SaveDesignRoot attribute](#)
  5. [Third-Party Licenses](#)
-

# 1. Interfaces

## 1.1 IAfterInitializeCallback

### Description

Use this interface if you want to perform some processing **once** during data initialization.

### Cases in which callbacks are called

- When data is initialized.
- When data is loaded and the target data does not exist in the save file due to a difference in game version.

### Cases where callbacks are not called

- If the data was included in the existing saved data when the data was loaded and successfully restored.

This interface must be implemented in a class with either the `SharedData` or `SlotData` attribute, It is ignored if implemented in a class with only the `SlotMetaData` or `TempData` attributes, or in a class with none of the data attributes.

### Public Methods

Method	Description
<a href="#">OnAfterInitialize</a>	Called at data initialization.

### OnAfterInitialize

- public void **OnAfterInitialize** ();

### Description

It is called only once during data initialization.

```
using System;
using SaveDesign.Runtime;

[SharedData, Serializable]
public class ExampleClass : IAfterInitializeCallback
{
    public int money;

    void IAfterInitializeCallback.OnAfterInitialize()
    {
        money = 100;
    }
}
```

## 1.2 IAfterInitializeRollback<T>

### Description

Use this interface when an exception occurs during initialization processing and you need to roll back side effects caused by the callback implemented in [IAfterInitializeCallback](#).

### Cases where it is called

- When an exception occurred during initialization processing, [IAfterInitializeCallback.OnAfterInitialize\(\)](#) was being called.

#### Cases where it is not called

- [IAfterInitializeCallback](#) is not implemented
- When an exception occurred during initialization, [IAfterInitializeCallback.OnAfterInitialize\(\)](#) was not called

#### Public Methods

Method	Description
<a href="#">OnAfterInitializeRollback</a>	Called when rolling back data initialization processing.

#### OnAfterInitializeRollback

- public void **OnAfterInitializeRollback** (T previousData);

#### Description

Called when rolling back data initialization processing.

```
using System;
using SaveDesign.Runtime;
using UnityEngine;

[SharedData, Serializable]
public class ExampleClass : IAfterInitializeCallback, IAfterInitializeRollback<ExampleClass>
{
    [SerializedField] int frameRate;

    public void SetFrameRate(int frameRate)
    {
        this.frameRate = frameRate;
        Application.targetFrameRate = frameRate;
    }

    void IAfterInitializeCallback.OnAfterInitialize()
    {
        SetFrameRate(60);
    }

    void IAfterInitializeRollback.OnAfterInitializeRollback(ExampleClass previousData)
    {
        if (previousData != null) Application.targetFrameRate = previousData.frameRate;
        else Application.targetFrameRate = -1;
    }
}
```

### 1.3 IAfterLoadCallback

#### Description

Use this interface if you want to perform some processing after data has been read.

This interface must be implemented in a class with one of the `SharedData` , `SlotData` , or `SlotMetaData` attributes, It is ignored if implemented in a class with only the `TempData` attribute or none of the data attributes.

**Public Methods**

Method	Description
<a href="#">OnAfterLoad</a>	It is called immediately after data is read.

**OnAfterLoad**

- `public void OnAfterLoad ();`

**Description**

It is called after data has been read.

```
using System;
using SaveDesign. Runtime;

[SharedData, Serializable]
public class ExampleClass : IAfterLoadCallback
{
    public int numberOfStartups;

    void IAfterLoadCallback.OnAfterLoad()
    {
        numberOfStartups++;
    }
}
```

**1.4 IAfterLoadRollback<T>**

**Description**

Use this interface when an exception occurs during the loading process and you need to roll back side effects caused by the callback implemented in [IAfterLoadCallback](#).

**Public Methods**

Method	Description
<a href="#">OnAfterLoadRollback</a>	Called when rolling back the data read operation.

**OnAfterLoadRollback**

- `public void OnAfterLoadRollback (T previousData);`

**Description**

Called when rolling back the data read operation.

```
using System;
using SaveDesign. Runtime;
```

```

using UnityEngine;

[SharedData, Serializable]
public class ExampleClass : IAfterLoadCallback, IAfterLoadRollback<ExampleClass>
{
    [SerializedField] int frameRate;

    public void SetFrameRate(int frameRate)
    {
        this.frameRate = frameRate;
        Application.targetFrameRate = frameRate;
    }

    void IAfterLoadCallback.OnAfterLoad()
    {
        SetFrameRate(frameRate);
    }

    void IAfterLoadRollback.OnAfterLoadRollback(ExampleClass previousData)
    {
        if (previousData != null) Application.targetFrameRate = previousData.frameRate;
        else Application.targetFrameRate = -1;
    }
}

```

## 1.5 IBeforeSaveCallback

### Description

Use this interface if you want to perform some processing before the data is stored.

This interface must be implemented in a class with one of the following attributes: `SharedData` , `SlotData` , or `SlotMetaData` , It is ignored if implemented in a class with only the `TempData` attribute or none of the data attributes.

### Public Methods

Method	Description
<a href="#">OnBeforeSave</a>	It is called just before data is saved.

### OnBeforeSave

- public void **OnBeforeSave** ();

### Description

It is called before data is saved.

```

using System;
using SaveDesign.Runtime;

[SharedData, Serializable]

```

```

public class ExampleClass : IBeforeSaveCallback
{
    static readonly DateTime s_epoch = new(1970, 1, 1, 0, 0, 0, DateTimeKind.Utc);

    // JsonUtility does not support serialization of DateTime type, so it is stored as a long type.
    public long saveDateTime;

    public DateTime SaveDateTime => s_epoch.AddMilliseconds(saveDateTime).ToLocalTime();

    void IBeforeSaveCallback.OnBeforeSave()
    {
        // Converts DateTime type to long and writes it just before the data is saved.
        saveDateTime = (long)(DateTime.Now.ToUniversalTime() - s_epoch).TotalMilliseconds;
    }
}

```

## 1.6 IBeforeSaveRollback

### Description

Use this interface when an exception occurs during write processing and you need to roll back side effects caused by the callback implemented in [IBeforeSaveCallback](#).

#### Public Methods

Method	Description
<a href="#">OnBeforeSaveRollback</a>	Called when rolling back a data write operation.

#### OnBeforeSaveRollback

- public void **OnBeforeSaveRollback** ();

### Description

Called when rolling back a data write operation.

## 1.7 ISaveDesignConfig

### Description

An interface that provides settings for data storage.

#### Public Methods

Method	Description
<a href="#">GetSaveDataDirectoryPath</a>	Obtain the directory path where the file is to be saved.
<a href="#">GetSharedDataFileName</a>	Obtain the name of the file in which the shared data is to be stored.
<a href="#">GetSlotDataFileName</a>	Get the file name of the data to be saved separately for each save slot.
<a href="#">GetFileExtension</a>	Gets the file extension of the file to be saved.

<a href="#">GetExceptionPolicy</a>	Get the type of exception handling.
------------------------------------	-------------------------------------

---

### GetSaveDataDirectoryPath

- `public void GetSaveDataDirectoryPath ();`

#### Description

Get the directory path where the file is stored.

If you have no particular preference, it is recommended to implement this function to return

`Application.persistentDataPath + directory name` .

Some platforms, such as Android, will not allow read/write unless the directory path under

`Application.persistentDataPath` is returned.

---

### GetSharedDataFileName

- `public void GetSharedDataFileName ();`

#### Description

Obtain the name of the file in which the shared data is to be stored.

---

### GetSlotDataFileName

- `public void GetSlotDataFileName ();`

#### Description

Obtains the name of the file that stores slot-specific data.

---

### GetFileExtension

- `public void GetFileExtension ();`

#### Description

Gets the file extension of the file to save.

Returning `null` or an empty string will generate a file with no extension.

---

### GetExceptionPolicy

- `public ExceptionPolicy GetExceptionPolicy ();`

#### Description

Retrieve the type of behavior when an exception occurs during initialization, loading, or saving.

---

## 2. Attributes

### 2.1 EncryptorAttribute

#### Description

Used to incorporate encryption and composite processing when reading/writing data.

```
using SaveDesign.Runtime;

[Encryptor]
public static class CustomEncryptor
{
    public static void Encrypt(ref byte[] data)
    {
        ...
    }

    public static void Decrypt(ref byte[] data)
    {
        ...
    }
}
```

#### Constructors

- `public EncryptorAttribute ();`

### 2.2 SaveDesignRootAttribute

#### Description

This attribute is assigned to the core class that manages the saved data.

Classes to which this attribute is assigned will automatically generate entry points for initialization, saving, loading, deleting, etc.

See [Class with SaveDesignRoot attribute](#) for details.

#### Constructors

- `public SaveDesignRootAttribute (SerializerType serializerType);`

Parameters	Description
<a href="#">serializerType</a>	Type of serializer to be used. (default value: SerializerType.JsonUtility)

```
using System;
using SaveDesign.Runtime;

[SaveDesignRoot]
internal partial class ExampleClass { }
```



```
// Accessible as follows
ExampleClass.Shared
ExampleClass.Slot
ExampleClass.Load
```

## serializerType

### DESCRIPTION

You can set the serializer to be used.

See the [SerializerType](#) section for details.

## 2.3 SharedDataAttribute

### Description

Attribute that defines data shared by all save slots.

It is suitable for player-wide progress and global settings.

### Constructors

- public **SharedDataAttribute** ();
- public **SharedDataAttribute** (string **path**);
- public **SharedDataAttribute** (params Type[] **dependsOnTypes**);
- public **SharedDataAttribute** (string **path**, params Type[] **dependsOnTypes**);

Parameters	Description
<a href="#">path</a>	Hierarchical path to access data.
<a href="#">dependsOnTypes</a>	List of data on which to rely for initialization and read/write.

```
using System;
using SaveDesign.Runtime;

[SharedData, Serializable]
public class ExampleClass
{
    public string value;
}

// Accessible as follows
SD.Shared.ExampleClass.value = "shared data example.";

var value = SD.Shared.ExampleClass.value;
```

## path

### DESCRIPTION

A hierarchical path for accessing data.

Data can be organized by hierarchy.

You can divide data into multiple hierarchies by setting slash-separated paths.

Hierarchical paths are generated by the `static partial` class and can be freely extended.

```
using System;
using SaveDesign.Runtime;

[SharedData("Path1/Path2"), Serializable]
public class ExampleClass { }

// Accessible as follows
SD.Shared.Path1.Path2.ExampleClass
```

---

## dependsOnTypes

### DESCRIPTION

You can set other data to be depended on when initializing data or reading/writing data.

By setting this, reading and writing of this data will be executed after all dependent data has been read and written.

However, all dependent data must be of the same type.

Also, you cannot set up a circular dependency.

```
using System;
using SaveDesign.Runtime;

[SharedData, Serializable]
public class A
{
    public bool flag;
}

[SharedData(typeof(A), Serializable]
public class B : IAfterLoadCallback
{
    public int value;

    void IAfterLoadCallback.OnAfterLoad()
    {
        // Dependent on data A in post-loading process
        if (SD.Shared.A.flag) value += 50;
    }
}

[SlotData, Serializable]
public class SlotData { }

// Error because the type of data is different from that of the relying party.
[SharedData(typeof(SlotData), Serializable]
public class C { }
```

---

## 2.4 SlotDataAttribute

### Description

This attribute defines data to be stored separately for each save slot.

It is suitable for data related to individual game progression, such as character status, items possessed, progression chapters, etc.

### Constructors

- `public SlotDataAttribute ();`
- `public SlotDataAttribute (string path);`
- `public SlotDataAttribute (params Type[] dependsOnTypes);`
- `public SlotDataAttribute (string path, params Type[] dependsOnTypes);`

Parameters	Description
<a href="#">path</a>	Hierarchical path to access data.
<a href="#">dependsOnTypes</a>	List of data on which to rely for initialization and read/write.

```
using System;
using SaveDesign.Runtime;

[SlotData, Serializable]
public class ExampleClass
{
    public string value;
}

// Accessible as follows
SD.Slot.ExampleClass.value = "slot data example.";

var value = SD.Slot.ExampleClass.value;
```

### path

#### DESCRIPTION

A hierarchical path for accessing data.

Data can be organized by hierarchy.

You can divide data into multiple hierarchies by setting slash-separated paths.

Hierarchical paths are generated by the `static partial` class and can be freely extended.

```
using System;
using SaveDesign.Runtime;

[SlotData("Path1/Path2"), Serializable]
public class ExampleClass { }
```

```
// Accessible as follows  
SD.Slot.Path1.Path2.ExampleClass
```

---

## dependsOnTypes

### DESCRIPTION

You can set other data to be depended on when initializing data or reading/writing data.

By setting this, reading and writing of this data will be executed after all dependent data has been read and written.

However, all dependent data must be of the same type.

Also, you cannot set up a circular dependency.

```
using System;  
using SaveDesign.Runtime;  
  
[SlotData, Serializable]  
public class A  
{  
    public bool flag;  
}  
  
[SlotData(typeof(A)), Serializable]  
public class B : IAfterLoadCallback  
{  
    public int value;  
  
    void IAfterLoadCallback.OnAfterLoad()  
    {  
        // Dependent on data A in post-loading process  
        if (SD.Slot.A.flag) value += 50;  
    }  
}  
  
[SharedData, Serializable]  
public class SharedData { }  
  
// Error because the type of data is different from that of the relying party.  
[SlotData(typeof(SharedData)), Serializable]  
public class C { }
```

---

## 2.5 SlotMetaDataAttribute

### Description

This attribute defines the meta information associated with each save slot.

It is separated from the actual save data and can be used to display a list of save slots.

However, unlike other types of data, meta information is created as new data each time it is saved.

Therefore, directly rewriting the value of loaded meta information will not save it.

This is a mechanism to ensure that meta information is automatically generated from other types of data.

---

## Constructors

- `public SlotDataAttribute ();`

```
using System;
using SaveDesign.Runtime;

[SlotMetaData, Serializable]
public class ExampleClass : IBeforeSaveCallback
{
    public string playerName;
    public int level;
    public int money;

    void IBeforeSaveCallback.OnBeforeSave()
    {
        var player = SD.Slot.Player;
        playerName = player.name;
        level = player.level;
        money = player.money;
    }
}

// It can be read as follows
if (SD.Load.SlotMeta(slotIndex, out var meta))
{
    var info = meta.playerName + ", " + meta.level + ", " + meta.money;

    meta.playerName = "dummy name"; // ✗ Rewriting meta information values is not saved.
}

// The save is saved with the SlotData when it is saved.
SD.Save.Slot(0);
```

---

## 2.6 TempDataAttribute

### Description

Temporary data that is not saved.

It is used to store flags and temporary states that are only valid during a game session.

When it is reset can be controlled by [TempDataResetTiming](#).

---

## Constructors

- `public TempDataAttribute ();`
- `public TempDataAttribute (string path);`
- `public TempDataAttribute (TempDataResetTiming resetTiming);`
- `public TempDataAttribute (params Type[] dependsOnTypes);`
- `public TempDataAttribute (string path, TempDataResetTiming resetTiming);`
- `public TempDataAttribute (string path, params Type[] dependsOnTypes);`

- public TempDataAttribute ([TempDataResetTiming](#) resetTiming, params Type[] \* *dependsOnTypes*\*);
- public TempDataAttribute (string path, [TempDataResetTiming](#) resetTiming, params Type[] dependsOnTypes);

Parameters	Description
<a href="#">path</a>	Hierarchical path to access data.
<a href="#">resetTiming</a>	When to reset temporary data. (default value: TempDataResetTiming.OnSharedDataLoad)
<a href="#">dependsOnTypes</a>	List of data on which to rely for initialization and read/write.

```
using SaveDesign.Runtime;

[TempData]
public class ExampleClass
{
    public string value;
}

// Accessible as follows
SD.Temp.ExampleClass.value = "temp data example.";

var value = SD.Temp.ExampleClass.value;
```

## path

### DESCRIPTION

A hierarchical path for accessing data.

Data can be organized by hierarchy.

You can divide data into multiple hierarchies by setting slash-separated paths.

Hierarchical paths are generated by the `static partial` class and can be freely extended.

```
using SaveDesign.Runtime;

[TempData("Path1/Path2")]
public class ExampleClass { }

// Accessible as follows
SD.Temp.Path1.Path2.ExampleClass
```

## resetTiming

### DESCRIPTION

You can set the timing for resetting temporary data.

If you have temporary data that you want to use separately for each save slot, for example, you can set this value to automatically reset the data at the appropriate time.

This mechanism prevents bugs caused by forgetting to initialize.

```

using SaveDesign.Runtime;

[TempData(TempDataResetTiming.OnSlotDataLoad)]
public class ExampleClass
{
    public string value = "reset";
}

SD.Temp.ExampleClass.value = "example";

var value = SD.Temp.ExampleClass.value; // example

if (SD.Load.Slot("identifier"))
{
    value = SD.Temp.ExampleClass.value; // reset
}

```

## dependsOnTypes

### DESCRIPTION

You can set other data on which the data initialization process depends.

By setting this, the initialization process for this data will be executed after the initialization process for all dependent data has been completed.

However, all dependent data must be of the same type,

**In the case of temporary data, the reset timing must also be the same as that of the dependent data.**

In addition, circular dependencies cannot be set.

```

using SaveDesign.Runtime;

[TempData]
public class A
{
    public bool flag;
}

[TempData(typeof(A))]
public class B
{
    public int value;

    public B()
    {
        // Depends on the value of data A in the initialization process.
        if (SD.Temp.A.flag) value += 50;
    }
}

[SlotData, System.Serializable]
public class SlotData { }

```

```
// ✖ Error because the type of data is different from that of the relying party.  
[TempData(typeof(SlotData))]  
public class C { }
```

```
// ✖ Error due to different reset timing from the data of the relying party.  
[TempData(TempDataResetTiming.OnGameStart, typeof(A))]  
public class D { }
```

---



### 3. Enumerations

#### 3.1 ExceptionPolicy

**Description**

The type of exception handling for exceptions occurring during initialization or read/write operations.

**Properties**

Parameters	Description
Throw	Throws the exception as-is. Use this when the caller wants to perform a try-catch and exercise fine-grained control over exception handling.
LogAndSuppress	The exception is logged using <code>UnityEngine.Debug.LogException</code> and then suppressed.
Suppress	Swallows the thrown exception. Since no logs are output, this is used when you don't want to show errors to users or wish to silently ignore failures.

#### 3.2 SerializerType

**Description**

Serializer type.

You can switch the type of serializer to use by setting this to the `SaveDesignRoot` attribute.

```
using SaveDesign.Runtime;

[SaveDesignRoot(SerializerType.MessagePack)]
internal partial class SD { }
```

**Properties**

Properties	Description
JsonUtility	Unity standard JSON library.
MessagePack	MessagePack for C# [MIT License]
Newtonsoft.Json	Newtonsoft.Json [MIT License]
MemoryPack	MemoryPack [MIT License]

#### 3.3 TempDataResetTiming

**Description**

Type of timing for resetting temporary data.

**Properties**

Properties	Description
OnGameStart	Reset only once at game startup.
OnSharedDataLoad	Reset when initializing or reading shared data.
OnSlotDataLoad	Reset when initializing or loading a save slot.
Manual	Reset manually.

## 4. Classes

### 4.1 Class with `SaveDesignRoot` attribute

Description

Static properties

Properties	Description
<a href="#">config</a>	Data Storage Settings.
<a href="#">currentSlotIndex</a>	The save slot number currently being read. (Read only)

**config**

public static [ISaveDesignConfig](#) config;

DESCRIPTION

The directory path and file name set in this will be used to store the data.

This must be set before reading or writing data.

See the [ISaveDesignConfig](#) section for details.

**currentSlotIndex**

- public static int **currentSlotIndex**;

DESCRIPTION

Returns the number of the save slot currently being read. (read only)

Available if there is one or more classes with the `SlotData` attribute.

It will be **automatically updated** to the appropriate value upon initialization or read/write.

Operation	Change in currentSlotIndex
Initialize.Slot()	Set to -1.
Load.Slot(identifier)	Set to -1.
Load.Slot(slotIndex)	The slotIndex is set.
Save.Slot(slotIndex)	The slotIndex is set.
Save.Slot(identifier)	<b>Unchanged</b> (values remain unchanged)

Delete.Slot(slotIndex)	Unchanged (values remain unchanged)
Delete.Slot(identifier)	Unchanged (values remain unchanged)

```
var slotIndex = SD.currentSlotIndex; // -1

if (SD.Load.Slot(0))
{
    slotIndex = SD.currentSlotIndex; // 0
}

if (SD.Save.Slot(1))
{
    slotIndex = SD.currentSlotIndex; // 0
}

if (SD.Load.Slot("identifier"))
{
    slotIndex = SD.currentSlotIndex; // -1
}
```

Entry point

Classes with the `SaveDesignRoot` attribute automatically generate entry points to perform initialization, read/write, etc.

Entry points are generated by the `static partial` class and can be freely extended.

The generated entry points are as follows.

Entry point	Description
<a href="#">Initialize</a>	Initialize data.
<a href="#">Load</a>	Load data.
<a href="#">Save</a>	Save data.
<a href="#">Delete</a>	Delete data.
<a href="#">Shared</a>	Entry point for accessing shared data.
<a href="#">Slot</a>	Entry points for accessing data to be stored separately per save slot.
<a href="#">Temp</a>	Entry point for accessing temporary data that will not be stored.

Initialize

Description

Entry point to data initialization function.

Static Methods

- public static void **Shared** ();

- public static void **Shared** ([ExceptionPolicy](#) exceptionPolicy);

#### DESCRIPTION

Initialize shared data.

For shared data, it is recommended to implement a process that performs a read only once at game startup and initializes the data if it fails.

```
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
static void InitSaveDesignConfig()
{
    // Storage-related settings
    SD.config = Resources.Load<SaveDesignConfig>("SaveDesignConfig");

    // Initialize if shared data fails to load
    if (!SD.Load.Shared()) SD.Initialize.Shared();
}
```

- public static void **Slot** ();
- public static void **Slot** ([ExceptionPolicy](#) exceptionPolicy);

#### DESCRIPTION

Initialize the data to be saved separately for each save slot.

Run it when you start a new game.

```
public void NewGame()
{
    SD.Slot.Player.money = 100; // ❌ Error because data not initialized

    SD.Initialize.Slot();

    SD.Slot.Player.money = 100; // ✅ OK
}
```

### Load

#### Description

Entry point to the data read function.

#### Static Methods

- public static bool **Shared** ();
- public static bool **Shared** ([ExceptionPolicy](#) exceptionPolicy);

#### DESCRIPTION

Load shared data.

For shared data, it is recommended to implement a process that performs loading only once at game startup and initializes the data if it fails.

```
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
static void InitSaveDesignConfig()
```

```

{
    // Storage-related settings
    SD.config = Resources.Load<SaveDesignConfig>("SaveDesignConfig");

    // Initialize if shared data fails to load
    if (!SD.Load.Shared()) SD.Initialize.Shared();
}

```

- public static bool **Slot** (int slotIndex);
- public static bool **Slot** (int slotIndex, [ExceptionPolicy](#) exceptionPolicy);
- public static bool **Slot** (string identifier);
- public static bool **Slot** (string identifier, [ExceptionPolicy](#) exceptionPolicy);

#### DESCRIPTION

Load the data to be saved separately for each save slot, specifying the **slot number** or **identifier**.

Execute this function when starting a game by taking over the data saved in a save slot.

In addition, the read function by slot number is used **when reading data in a save slot**, and the read function by identifier is used for autosaves, checkpoints, etc. We recommend that you use the loading function by identifier when loading data that has nothing to do with **save slots**.

```

public void LoadGame(int slotIndex)
{
    if (SD.Load.Slot(slotIndex))
    {
        // Transition to the next scene when data is read.
        SceneManager.LoadScene("Next Scene");
    }
}

```

- public static bool **SlotMeta** (int slotIndex, out ? meta);
- public static bool **SlotMeta** (int slotIndex, out ? meta, [ExceptionPolicy](#) exceptionPolicy);
- public static bool **SlotMeta** (string identifier, out ? meta);
- public static bool **SlotMeta** (string identifier, out ? meta, [ExceptionPolicy](#) exceptionPolicy);

#### DESCRIPTION

Load the meta information associated with each save slot, specifying the **slot number** or **identifier**.

Execute this when displaying the information for each save slot on the save data load screen.

The use of slot numbers and identifiers is the same as for data saved separately for each save slot.

Also, no asynchronous functions are generated.

```

// UI list of save slots
[SerializeField] SaveSlotUI[] slots;

public void DisplaySaveSlots()
{
    for(int i = 0; i < slots.Length; i++)
    {

```

```

        if (SD.Load.SlotMeta(i, out var meta))
        {
            // If there is meta information, it means there is stored data.
            slots[i].UpdateUI(i, meta);
        }
        else
        {
            // If there is no meta information, the save slot is empty.
            slots[i].UpdateUI(i, "no data");
        }
    }
}

```

## Save

### Description

Entry point to the data save function.

Meta information is automatically saved when the `Slot` is saved.

### Static Methods

- `public static bool Shared ()`;
- `public static bool Shared (ExceptionPolicy exceptionPolicy)`;

### DESCRIPTION

Save shared data.

It is recommended to recall it after changing game settings or at the end of a game.

```

public void SaveSetting()
{
    if (SD.Save.Shared())
    {
        ...
    }
}

```

- `public static bool Slot (int slotIndex)`;
- `public static bool Slot (int slotIndex, ExceptionPolicy exceptionPolicy)`;
- `public static bool Slot (string identifier)`;
- `public static bool Slot (string identifier, ExceptionPolicy exceptionPolicy)`;

### DESCRIPTION

The data to be saved separately for each save slot is saved by specifying the **slot number** or **identifier**.

It is recommended that saving by **slot number** be used when saving **save slot data** and saving by **identifier** be used when saving data unrelated to **save slots**, such as autosaves, checkpoints, etc. We recommend that you use it when saving data that has nothing to do with a **save slot**.

```

public void SaveGame(int slotIndex)
{
    if (SD.Save.Slot(slotIndex))

```

```
{  
    ...  
}  
}
```

---

## Delete

### Description

Entry point to the delete data function.

Meta information is automatically deleted when the `Slot` is deleted.

---

### Static Methods

- public static bool **Shared** ();
- public static bool **Shared** ([ExceptionPolicy](#) exceptionPolicy);

### DESCRIPTION

Delete shared data.

```
SD.Delete.Shared();
```

- 
- public static bool **Slot** (int slotIndex);
  - public static bool **Slot** (int slotIndex, [ExceptionPolicy](#) exceptionPolicy);
  - public static bool **Slot** (string identifier);
  - public static bool **Slot** (string identifier, [ExceptionPolicy](#) exceptionPolicy);

### DESCRIPTION

Delete data to be saved separately for each save slot, specifying the **slot number** or **identifier**.

```
SD.Delete.Slot(0);
```

---

## Shared

### Description

Entry point to shared data.

It is generated by the `static partial` class and can be freely extended.

```
using System;  
using SaveDesign.Runtime;  
  
[SharedData, Serializable]  
public class ExampleClass  
{  
    public int value;  
}  
  
// Accessible as follows  
SD.Shared.ExampleClass.value = 10;
```

## Slot

### Description

Entry point to data to be stored separately for each save slot.

It is generated by the `static partial` class and can be freely extended.

```
using System;
using SaveDesign.Runtime;

[SlotData, Serializable]
public class ExampleClass
{
    public int value;
}

// Accessible as follows
SD.Slot.ExampleClass.value = 10;
```

## Temp

### Description

Entry point to unstored temporary data.

It is generated by the `static partial` class and can be freely extended.

```
using SaveDesign.Runtime;

[TempData]
public class ExampleClass
{
    public int value;
}

// Accessible as follows
SD.Temp.ExampleClass.value = 10;
```

## 5. Third-Party Licenses

This package may generate code that references the following libraries:

- [MessagePack for C#](#) — MIT License
- [Newtonsoft.Json](#) — MIT License
- [MemoryPack](#) — MIT License

These libraries are **not included** in the package.

For license details, see `Third-Party Notices.txt`.