Arsenal: Script Reference

Last update from: 2025-02 (Version 1.0.0 on Unity Asset Store)
Author: Simon Albou - albou.simon@gmail.com

Disclaimer: This document is **not** the user manual.

It only contains the **Script Reference**, which is the complete guide to interacting with the plugin's codebase for **advanced** usage.
For the actual user manual, go to this link.

Table of Contents: *(click on any entry to reach it)*

1 - Introduction
- 1.1 - Welcome!
- 1.2 - Using Arsenal in your code
- 1.3 - How to read this API documentation

2 - Context Manager and Lock Matrix
- class ContextManager:

3 - Input System
- class InputManager:
- class KeyBindingAsset:
- struct KeyBinding:
- struct ButtonAction:
- struct SingleButton:
- struct SingleMouseButton:
- struct AxisAction:
- struct SingleAxis:
- struct SingleKeyedAxis:
- struct SingleVirtualAxis:
- class VirtualStick:

4 - Sound Manager
- class SoundManager:
- class Soundbank:
- struct SoundEffect:
- struct SFXClip:
- class SoundEffectPlayer:
- class BackgroundMusicChanger:

5 - Bringing a value from A to B: Tweeners and Gauges

class Tweener<T>:
struct RectTransformState:
class TweenerGroup:
interface ITweener:
class Gauge:

6 - Reskinning your UI with Themes

class Theme:
struct ThemeState:
struct ThemeOverride:
class ThemeManager:
class BaseThemedElement:
class ThemedObjectGroup:
class ThemedObject:
struct ThemeVariable:
struct ThemeComponentReference:
struct ThemeEvent:
struct ThemePairing:
class MasterTheme:

7 - UI presets and prefabs

class SmartButton:
class SelectorManager:
class SpatialSelector:
class SpatialSelectorTarget:
class CanvasFader:
class DraggableElement:
class PageSystem:
class TooltipBehaviour:

8 - The top-level canvas: Overlays and Cursor

class OverlayManager:
class CursorBehaviour:
class GraphicRaycasterExposer:
class ArsenalInputModule:
class ArsenalPointerInput:

9 - Localization

class LocalizationManager:
class Localizer:

10 - Dialogue System

struct Dialogue:
struct DialoguePiece:
class DialogueLauncher:
struct DialogueEvent:
struct ReplacementRule:
struct ConditionalRule:
class DialogueManager:
class DialogueDisplayer:
class SpeakerDatabase:
struct SpeakerData:
class SpeakerDataDisplayer:
class PlayerChoiceDisplayer:
class PlayerChoice:

11 - Save System

class SaveManager:
struct SaveChannel:
class SaveFile:

12 - A sample use case for Savefiles: the Settings Panel

class SettingsSave:
class SettingsPanel:
class BaseSettingsSwitcher:
class MasterThemeSwitcher:
class FullscreenSwitcher:
class LanguageSwitcher:
class ResolutionSwitcher:
class VolumeSwitcher:
class MasterVolumeSwitcher:
class MusicVolumeSwitcher:
class SFXVolumeSwitcher:

1 - Introduction

(Chapter copied from the user manual)

1.1 - Welcome!

Hi there, thank you for choosing Arsenal! For any question / help / suggestion, you can join the Discord Server (link here), or directly contact me at **albou.simon@gmail.com**.

Special thanks to:

- Ali, for most graphic assets and various UI advice
- Everyone who helped through testing alpha and beta versions
- The BulletPro community!

Arsenal is a general utility plugin that handles a large amount of aspects of a game production, that are tedious but present in any project, thus letting you focus on the actual gameplay. This includes the following:

- Input manager
- Localization
- Dialogues
- Tweens
- Save system
- Fully functional and customizable “Settings” panel
- UI presets and prefabs
- Theme system: reskinnable UI with master “Theme” assets
- Context manager: easily allow or block any in-game action

To sum things up, Arsenal is the ideal tool to provide an optimal **initial configuration** to your project, and make sure it ships as fast as possible. In addition to this, the plugin has the following advantages:

- No dependency to any other package
- Gets fully integrated by dragging a couple of prefabs to your scene
- Clear UX: everything is self-explanatory and takes a minimal amount of clicks
- Supports all versions of Unity, from 2021 (LTS) to **6000**
- Comes with its full source code
- Thoroughly documented (well, you're here)

You can also check out my other plugin available on the Asset Store: **BulletPro** is a toolkit for everything related to **bullet** management in 2D gameplay, such as arcade games, platformer games, manic shooters, and so on. More info on the Asset Store page!

1.2 - Using Arsenal in your code

In every case, using classes from Arsenal in your scripts will require to access the proper namespace at the start of your script:

```
using UnityEngine;
using Arsenal;
```

As long as you put them in one of your Scenes at first, every **Manager** class is accessible as a singleton. They contain global utilities and are sometimes your primary interface with the related features. The public static access to each manager is called **`.instance`**.

```
if (InputManager.instance.GetButton("Test"))
{
    Debug.Log("Hello world!");
}
```

Simple example: checking for inputs is done through the InputManager.

The full list of available managers is:

- `ContextManager`
- `DialogueManager`
- `ArsenalManager` *// only used for holding all other managers together*
- `LocalizationManager`
- `OverlayManager`
- `InputManager`
- `SaveManager`
- `SelectorManager`
- `SoundManager`
- `ThemeManager`

1.3 - How to read this API documentation

- Each class has its own category.
- Under each class is a list of all its public members, properties and methods, with a short comment describing what it does.
- If a variable type or return value is shown in blue, it means it refers to a class (or struct) that has its own documentation page in the present document.
- If you find any information missing or incomplete, or want a more in-depth explanation on any point, don't hesitate to reach out! Once again, you can join the Discord Server (link here), or directly contact me at **albou.simon@gmail.com**.

2 - Context Manager and Lock Matrix

class ContextManager:

```
// All managers have their static instance exposed
public static ContextManager instance;

// Get a full list of context or feature names. Useful for debugging.
public string[] GetContextNames();
public string[] GetFeatureNames();

// Does <name> exist at all in the game?
public bool HasContext(string contextName);
public bool HasFeature(string featureName);

// Get information about a feature or context.
// Throws an error if the requested string isn't a valid name.
// If you're unsure, it is advised to check HasContext() or HasFeature() first.
public bool IsAllowed(string featureName);
public bool IsLocked(string featureName);
public bool IsOpened(string contextName);
public bool IsClosed(string contextName);

// Manipulate contexts
public void OpenContext(string contextName);
public void CloseContext(string contextName);
public void CloseAllContexts();
public void ResetContextsToStartup();

// Debug methods: sends messages to the Unity console
public void PrintOpenContexts();
public void PrintClosedContexts();
public void PrintAllowedFeatures();
public void PrintLockedFeatures();
public void PrintAllContexts(); // also states which ones are opened/closed
public void PrintAllFeatures(); // also states which ones are allowed/locked
```

3 - Input System

class InputManager:

```
// All managers have their static instance exposed
public static InputManager instance;

// Input manager mode is an enum:
public InputMode inputMode; // .Arsenal, .UnityLegacy, .UnityNewInputSystem

// Component reference, only relevant to InputMode.UnityNewInputSystem
public UnityEngine.InputSystem.PlayerInput playerInput;

// For Arsenal mode: Get/set KeyBindings
public KeyBinding GetKeyBinding(); // returns the current KeyBinding
public void SetKeyBinding(KeyBinding kb);
public void SetKeyBinding(KeyBindingAsset kbAsset); // overload for convenience
public void ResetKeyBinding(); // puts back the default KeyBinding

// Usual getters.
// Based on the inputMode, they will fetch data from different sources:
// InputMode.Arsenal: get data from currentKeyBinding
// InputMode.UnityLegacy: get data from Input.GetButton() and Input.GetAxis()
// InputMode.UnityNewInputSystem: get data from playerInput.actions[buttonName]
public bool GetButton(string buttonName); // is pressed
public bool GetButtonDown(string buttonName); // became pressed this frame
public bool GetButtonUp(string buttonName); // became released this frame
public float GetAxis(string axisName); // from -1 to 1

// Two GetAxis() calls in one:
public Vector2 GetVector(string axisNameX, string axisNameY);

// Returns whether this vector has nonzero magnitude:
public bool IsStickUsed(axisNameX, axisNameY);

// These overloads are only available with new input system,
// they’re meant for Actions that have a Vector2 value
public Vector2 GetVector(string actionName);
public bool IsStickUsed(string actionName);

// The following methods are only available for InputMode.Arsenal:

// Gets whole structs to be able to fetch more info.
public AxisAction GetAxisInfo(axisName);
public ButtonAction GetButtonInfo(axisName);
// KeyBinding manipulation (rebind helpers):
// These functions modify the value of keyBinding.
public void SwapAxes(string axisNameA, string axisNameB);
public void SwapButtons(string buttonNameA, string buttonNameB);
public void ResetKeyBinding(); // sets its value to backupKeyBinding
```

class KeyBindingAsset:

```
// This inherits ScriptableObject.
// This object only contains a KeyBinding struct and exposes it in an asset.
public KeyBinding keyBinding;
```

struct KeyBinding:

```
// This is a struct, not to be confused with KeyBindingAsset which is the
ScriptableObject class holding it. KeyBindingAsset should not be manipulated.

// The array of structs is stored here.
public AxisAction[] axisActions;
public ButtonAction[] buttonActions;

// This is automatically called at Awake.
// It should be manually called after modifying the KeyBinding at runtime.
public void Initialize();

// All getters from InputManager.cs exist in this struct too.
// For instance, calling GetAxis(axisName) on InputManager actually calls
keyBinding.GetAxis(axisName).
```

struct ButtonAction:

```
public string name;
public SingleButton[] buttons; // stores runtime info for each linked KeyCode
public SingleMouseButton[] mouseButtons; // runtime info for linked mouse clicks
public ButtonBlendMode blendMode; // enum: Any, All
public bool keyUp; // read-only: did the button just become released?
public bool keyDown; // read-only: did the button just become pressed?
public bool key; // read-only: is the button pressed right now?
```

struct SingleButton:

```
// One building block of ButtonAction. Represents a single unique key.
public KeyCode keyCode;
public bool keyUp; // read-only: did the button just become released?
public bool keyDown; // read-only: did the button just become pressed?
public bool key; // read-only: is the button pressed right now?
```

struct SingleMouseButton:

```
// The building block of ButtonAction. Represents a single unique mouse click.
public MouseClickType mouseClick; // enum: LeftClick, RightClick, MiddleClick
public bool keyUp; // read-only: did the button just become released?
public bool keyDown; // read-only: did the button just become pressed?
public bool key; // read-only: is the button pressed right now?
```

struct AxisAction:

```
public string name;
public SingleAxis[] axes; // Unity axes (Project Settings)
public SingleKeyedAxis[] keyedAxes; // key pairs
public SingleVirtualAxis[] virtualAxes; // axes from Virtual Sticks
public AxisBlendMode blendMode; // enum: Add, Multiply, Average, Max
public bool invert;
public float axisValue; // read-only
```

struct SingleAxis:

```
// One of the building blocks of AxisAction. Represents a Unity axis.
public string axisName; // name as defined in the Project Settings
public float axisValue; // read-only
```

struct SingleKeyedAxis:

```
// One of the building blocks of AxisAction. Represents two keys.
public KeyCode keyMinus, keyPlus; // will directly map to -1 and 1
public float axisValue; // read-only
```

struct SingleVirtualAxis:

```
// One of the building blocks of AxisAction. Represents a Virtual Stick axis.
public string axisName; // name as defined in the VirtualStick Component
public float axisValue; // read-only
```

class VirtualStick:

```
// References
public Camera cam; // Camera linked to the canvas. Can stay null in Overlay mode.
public RectTransform detectionZone; // root object
public RectTransform stickHolder; // full stick area
public RectTransform stickMovablePart; // movable part of stick

// Parameters (see part 3.3 for definitions)
public bool placeStickUnderFinger;
public bool canBeAutoInterrupted;
public bool hideWhenUnused;
public float deadZoneThreshold;
public float stickRadius; // from 0 to 1, 1 being either screen width (if
portrait) or screen height (if landscape)

// Used by InputManager (in Arsenal mode)
public string horizontalAxisName;
public string verticalAxisName;

// Toggle finger detection and movement
public void DisableStick();
public void EnableStick();

// Set stick visibility (usually doesn't need to be called manually)
public void ShowStick();
public void HideStick();
public void ToggleStick(); // ShowStick() if hidden, HideStick() if shown

// Manually set position of movable part, to imitate some values.
// DisableStick() should be called before using these.
// Otherwise, its natural behaviour will override this.
public void DisplayStickPosition(Vector2 displayPosition); // -1 to 1 range
public void ResetStick(); // forces stick to recenter

// Fetch values that are relevant to input logic (called by InputManager)
public float GetHorizontalValue();
public float GetVerticalValue();
public Vector2 GetStickValue(); // Outputs both values at once in a Vector2
```

4 - Sound Manager

class SoundManager:

```
// All managers have their static instance exposed
public static SoundManager instance;

// DB/volume Parameters
public float minDecibelVolume;
public AnimationCurve volumeToDecibelCurve; // maps [0 : 1] to [minDBVolume : 0]

// Assets
public AudioClip startingBGM;
public Soundbank[] soundbanks;

// References
public AudioMixer audioMixer;
public AudioSource bgmSource;
public AudioSource[] sfxSources;

// Handle music
public void ChangeBGM(AudioClip clip);
public void RevertBGM(); // sets BGM to the one that was previously playing

// Has the BGM source call Play/Pause/UnPause/Stop
public void PlayBGM();
public void PauseBGM();
public void UnPauseBGM();
public void StopBGM();

// Get and set volume (0-1 range), from any AudioMixerGroup:
// multiple overloads are available
public float GetVolume(string groupName);
public float GetVolume(AudioMixerGroup group);
public void SetVolume(string groupName, float newValue);
public void SetVolume(AudioMixerGroup group, float newValue);

// Playing a sound effect: multiple overloads are available
public void PlaySFX(AudioClip clip, float volume, float pitch);
public void PlaySFX(AudioClip clip); // defaults volume and pitch to 1
public void PlaySFX(string soundKey); // plays from a soundbank
public void PlaySFX(SoundEffect effect); // plays from a soundbank, without key

// plays a specific clip with volume/pitch random info, as defined in a soundbank
public void PlaySFX(SFXClip clip);
```

class Soundbank:

```
// In a Soundbank asset, every possible SFX is stored as a SoundEffect struct.
public SoundEffect[] soundEffects;
```

struct SoundEffect:

```
// A SoundEffect links a name (key) to a list of possible SFXClips.
// When fetching a SoundEffect, one SFXClip will be picked at random.
public string soundKey;
public SFXClip[] sfxClips;
```

struct SFXClip:

```
// Each SFXClip is an AudioClip with volume and pitch settings.
// When picked, it plays with random pitch/volume within the wanted range.
public AudioClip audioClip;
public float minPitch;
public float maxPitch;
public float minVolume;
public float maxVolume;
```

class SoundEffectPlayer:

```
// global parameters
public bool useSpecificSource; // if left to false, plays from the Manager
public AudioSource source; // relevant if useSpecificSource is true
public SoundFetchMode fetchMode; // enum: FromSoundbank, RegularAudioClip

// applies only to SoundFetchMode.FromSoundbank
public string soundKey;

// applies only to SoundFetchMode.RegularAudioClip
public AudioClip audioClip;
public float minPitch;
public float maxPitch;
public float minVolume;
public float maxVolume;

// Playing a sound effect: same overloads as in the Manager
public void Play();
public void Play(AudioClip clip, float customVolume, float customPitch);
public void Play(AudioClip clip); // applies this component's volume/pitch
public void Play(string soundKey);
public void Play(SoundEffect effect);
public void Play(SFXClip clip);
```

class BackgroundMusicChanger:

```
public AudioClip musicClip;
public void Play();
```

5 - Bringing a value from A to B: Tweeners and Gauges

class Tweener<T>:

```
// Tweeners inherit from Tweener<Color>, <float>, <Vector2> or <Vector3>.
// For instance, a SpriteAlphaTweener inherits Tweener<float>.
// You can create your own Tweeners which inherit Tweener<T> for any T.

// Parameters
public T defaultTargetValue;
public float defaultDuration;
public AnimationCurve defaultCurve;
public bool disallowLaunchWhileAlreadyRunning;

// Events
public UnityEvent onStarted;
public UnityEvent onComplete;
public UnityEvent onCancel;
// Events that pass the new progress value (0-1):
public UnityEvent<float> onProgressChanged; // passes value affected by curve
public UnityEvent<float> onProgressChangedRaw; // passes value unaffected by curve

// Launching the tweeners: multiple overloads available.
// Whenever an argument is used, it overrides the corresponding default value.
public void Launch();
public void Launch(T targetValue);
public void Launch(AnimationCurve curve);
public void Launch(T targetValue, float duration);
public void Launch(T targetValue, AnimationCurve curve);
public void Launch(float duration, AnimationCurve curve);
public void Launch(T targetValue, float duration, AnimationCurve curve);
// Launch(float duration) does not exist: it would create an ambiguity in classes
that inherit from Tweener<float>.

// Manipulate progress value during a tween: usually called automatically
public float Smooth(float progress); // returns a value affected by curve
public void SetProgress(float newValue); // expects a value from 0 to 1
public void Cancel(); // Interrupts and stops a tween

// Get information on the current tween:
public float GetProgress(); // returns a value from 0 to 1, unaffected by curve
public T GetStartValue(); // returns the value it had at progress=0
public T GetTargetValue(); // returns the value it will have at progress=1
public float GetDuration(); // returns total duration of this tween
public AnimationCurve GetCurve(); // returns curve used for this tween

// Internally called while performing a tween.
// Override these to create your own tweener inheriting this class:
public T GetValue(); // gets current value (at this frame)
public void SetValue(T newValue); // sets current value in relevant component
public T TweenValue(T start, T end, float ratio); // performs an interpolation
```

struct RectTransformState:

```
// Describes the configuration of a RectTransform so it can be tweened.
// Each parameter corresponds to the same-named value in the RectTransform class.
// RectPositionTweener inherits from Tweener<RectTransformState>, which exists for
this sole purpose.

public Vector2 anchoredPos;
public Vector2 anchorMin;
public Vector2 anchorMax;
public Vector2 offsetMin;
public Vector2 offsetMax;
```

class TweenerGroup:

```
public MonoBehaviour[] tweenerScripts; // Holds all group content
public ITweener[] tweeners; // Same list, but provides access to Tweener functions

// Calls corresponding functions in all children:
public void Launch();
public void Cancel();
public void SetProgress(float t); // expects value from 0 to 1
```

interface ITweener:

```
// All tweeners implement this interface. Helps mass-manipulate these Components.

void Launch();
void Cancel();
float GetProgress();
void SetProgress(float t); // expects value from 0 to 1
```

class Gauge:

```
// Value parameters
public float minValue;
public float maxValue;
public float initialFillAmount;

// Behaviour parameters
public AutomaticGaugeMode startingMode; // enum: Immobile, Charging, Emptying
public float chargingTime;
public float emptyingTime;

// Event parameters
public bool callEventsAtStart;

// (cont. next page)
```

```
// Value events
public UnityEvent<float> onChangeValue; // passes the new value
public UnityEvent<float> onIncreaseValue; // only called if prev value was smaller
public UnityEvent<float> onDecreaseValue; // only called if prev value was higher
public UnityEvent onBecomeEmpty;
public UnityEvent onBecomeFull;

// Passes <previous mode, new mode>
public UnityEvent<AutomaticGaugeMode, AutomaticGaugeMode> onChangeMode;

// Other behaviour events
public UnityEvent onStartImmobile;
public UnityEvent onQuitImmobile;
public UnityEvent onStartCharging;
public UnityEvent onQuitCharging;
public UnityEvent onStartEmptying;
public UnityEvent onQuitEmptying;

// Runtime data: these five values are read-only
public AutomaticGaugeMode currentMode;
public bool isFull;
public bool isEmpty;
public float fillAmount;
public float fillRatio; // normalized from 0 to 1, 0 being min, 1 being max

// Mode change
public void SetMode(AutomaticGaugeMode mode);
public void SetImmobile(); // Shortcut to SetMode(AutomaticGaugeMode.Immobile)
public void StartCharging(); // Shortcut to SetMode(AutomaticGaugeMode.Charging)
public void StartEmptying(); // Shortcut to SetMode(AutomaticGaugeMode.Emptying)

// Instantaneous value change:
public void FillCompletely();
public void EmptyCompletely();

// Standard method to modify value. (instantaneous)
public void SetAmount(float newValue);

// Adds to existing value.
// Enter a negative delta to decrease the gauge.
public void ChangeAmount(float delta);

// Like SetAmount() but with normalized values (from 0 to 1).
// 0 represents min value, 1 represents max value.
public void SetRatio(float newRatio);

// Like ChangeAmount() but with normalized values (from 0 to 1).
// 0 represents min value, 1 represents max value.
public void ChangeRatio(float delta);
```

6 - Reskinning your UI with Themes

class Theme:

```
// Theme inherits ScriptableObject.
public ThemeState defaultState;
public bool allowStackingStates;
public ThemeState[] states;
```

struct ThemeState:

```
// Contains basic info for a single state.
public string stateName;
public float transitionDuration;
public AnimationCurve transitionCurve;
public ThemeOverride[] overrides;
```

struct ThemeOverride:

```
public bool enabled;

// Large enum of 19 possible values, defines the override:
public ThemeOverrideMode overrideMode;

// Copied from the parent ThemeState at serialization time
public float transitionDuration;
public AnimationCurve transitionCurve;

// One of these values will be relevant based on overrideMode:
public Color colorValue;
public float floatValue;
public Vector2 vector2Value;
public bool boolValue;
public int intValue;
public string stringValue;
public UnityEngine.Object objectReferenceValue;

// Blend mode: one of these values will be relevant based on overrideMode.
public ThemeBoolBlend boolBlendMode; // enum: Set, Toggle, And, Or, Xor
public ThemeColorBlend colorBlendMode; // enum: Set, Add, Mult, Overlay, Invert
public ThemeNumberBlend numberBlendMode; // enum: Set, Add, Multiply, Min, Max

// Per-ThemedObject advanced parameters
public string componentReferenceId; // identifies the affected object(s)
public bool isThemeVariable; // returns whether linked to a theme variable
public string variableId; // relevant if isThemeVariable is true
```

class ThemeManager:

```
// All managers have their static instance exposed
public static ThemeManager instance;

public ThemeVariable[] globalVariables;
public MasterTheme defaultMasterTheme;
public MasterTheme[] availableMasterThemes;
public UnityEvent onMasterThemeChanged;

// Interact with the MasterTheme
public MasterTheme GetMasterTheme();
public void SetMasterTheme(MasterTheme newMasterTheme); // triggers the event
// Overload that draws from the availableMasterThemes array:
public void SetMasterTheme(int masterThemeIndex);

// Obtains the relevant component reference for a given ThemeOverrideMode
// This works regardless of obj's type, it can be a GameObject or any component.
// "mode" is the same enum used in ThemeOverride.overrideMode.
public T Extract<T>(UnityEngine.Object obj, ThemeOverrideMode mode);

// Advanced: manually cleans up memory taken by cached components.
// Very rarely necessary.
// If called, it should ideally be during loading times, such as loading a scene.
public void FlushCaches();

// Get theme variables (has no fallback, being the highest level available)
public bool HasThemeVariable(string varName);
public Color GetColor(string varName);
public float GetFloat(string varName);
public Vector2 GetVector2(string varName);
public bool GetBool(string varName);
public UnityEngine.Object GetObjectReference(string varName);
public int GetInt(string varName);
public string GetString(string varName);

// Set theme variables
public void SetColor(string varName, Color newValue);
public void SetFloat(string varName, float newValue);
public void SetVector2(string varName, Vector2 newValue);
public void SetBool(string varName, bool newValue);
public void SetObjectReference(string varName, UnityEngine.Object newValue);
public void SetInt(string varName, int newValue);
public void SetString(string varName, string newValue);
```

class BaseThemedElement:

```
// The base class from which ThemedObject and ThemedObjectGroup both inherit.
// This is a MonoBehaviour.

public Theme startingTheme; // GetTheme() and SetTheme() should be used instead.
public Theme currentTheme; // Read-only
public bool applyAtStart;
public bool autoAdaptToMasterTheme;
public ThemePairing[] alternativeThemes;
public ThemedObjectGroup parentGroup; // Group holding this, if any.
public ThemeVariable[] themeVariables;

// Manipulate Themes
public Theme GetTheme();
public void SetTheme(Theme newTheme);

// Usually called automatically, but appearance can be recalculated manually too
public void RefreshTheme();

// Get theme variables.
// In case of error/fail, uses variables from parent group as fallback, then from
MasterTheme, then from ThemeManager.
public bool HasThemeVariable(string varName);
public Color GetColor(string varName);
public float GetFloat(string varName);
public Vector2 GetVector2(string varName);
public bool GetBool(string varName);
public UnityEngine.Object GetObjectReference(string varName);
public int GetInt(string varName);
public string GetString(string varName);

// Set theme variables
// Doing this at runtime will set var.pointsToHigherLevelVariable to false.
public void SetColor(string varName, Color newValue);
public void SetFloat(string varName, float newValue);
public void SetVector2(string varName, Vector2 newValue);
public void SetBool(string varName, bool newValue);
public void SetObjectReference(string varName, UnityEngine.Object newValue);
public void SetInt(string varName, int newValue);
public void SetString(string varName, string newValue);

// Tells a Theme variable to point to a higher-level one, like the Manager or a
MasterTheme. Sets var.pointsToHigherLevelVariable to true.
public void SetVariableRedirection(string varName, string higherVarReference);
```

class ThemedObjectGroup:

```
// This class inherits BaseThemedElement.

public BaseThemedElement[] groupContents; // reference loops will be auto-avoided

// Usually called automatically. Sets Theme of all objects in groupContents.
public override void RefreshTheme();
```

class ThemedObject:

```
// This class inherits BaseThemedElement.

// Used for referencing all objects affected by Theme state changes.
// Multiple references can (and should) share the same ID if you want a change to
affect multiple objects at once.
public ThemeComponentReference[] componentReferences;

// Contains UnityEvents (with a key) that can be called by Theme state changes.
public ThemeEvent[] events;

// Retrieves all referenced objects under the same key
public UnityEngine.Object[] GetComponentReferences(string key);

// Manipulate states: most common use cases.
// Arguments can have spaces to reference multiple states at once,
// ie. AddState("hovered pressed") would add both states in a single call.
// Logic operators (!^&|) should not be used in arguments here.
public void AddState(string stateName);
public void RemoveState(string stateName);
public void ToggleState(string stateName);
public void SetState(string stateName, bool stateValue);
public bool HasState(string stateName); // can't accept spaces in argument

// (advanced, usually not needed) Bypass state logic to directly apply changes:
public void ApplyThemeState(ThemeState state);
public void ApplyOverride(ThemeOverride tor, float duration, AnimationCurve
curve);
```

struct ThemeVariable:

```
// Holds data for all variables declared inside a ThemedObject.
public string name;

// enum: Color, Object, Float, Vector2, String, Bool, Int
public ThemeVariableType variableType;

// The variable can redirect to another variable declared at a higher level,
// such as a MasterTheme or the ThemeManager:
public bool pointsToHigherLevelVariable;
public string higherVariableName;

// otherwise, one of these values will be relevant, based on variableType
public Color colorValue;
public UnityEngine.Object objectReferenceValue;
public float floatValue;
public Vector2 vector2Value;
public string stringValue;
public bool boolValue;
public int intValue;
```

struct ThemeComponentReference:

```
// Stores data for any reference to objects affected/modified by Themes.
public string referenceId;
public UnityEngine.Object objectReference;
```

struct ThemeEvent:

```
// Stores data for any event that should be called by Themes.
public string name;
public UnityEvent onTriggered;
```

struct ThemePairing:

```
// Represents the link between a MasterTheme and a Theme.
public MasterTheme masterTheme;
public Theme localTheme;
```

class MasterTheme:

```
// Represents a reference to a MasterTheme, so that one MasterTheme at a time can be active in the ThemeManager.

// While this MasterTheme is active, all ThemedObjects are considered to be in <stateName>.
public string stateName;

// MasterTheme assets also hold Theme Variables.
public ThemeVariable[] themeVariables;

// Get theme variables.
// In case of invalid var name, uses ThemeManager's theme variables as fallback.
public bool HasThemeVariable(string varName);
public Color GetColor(string varName);
public float GetFloat(string varName);
public Vector2 GetVector2(string varName);
public bool GetBool(string varName);
public UnityEngine.Object GetObjectReference(string varName);
public int GetInt(string varName);
public string GetString(string varName);

// Set theme variables.
// Doing this at runtime will set var.pointsToHigherLevelVariable to false.
public void SetColor(string varName, Color newValue);
public void SetFloat(string varName, float newValue);
public void SetVector2(string varName, Vector2 newValue);
public void SetBool(string varName, bool newValue);
public void SetObjectReference(string varName, UnityEngine.Object newValue);
public void SetInt(string varName, int newValue);
public void SetString(string varName, string newValue);

// Tells a Theme variable to point to a higher-level one, like the Manager or a MasterTheme. Sets var.pointsToHigherLevelVariable to true.
public void SetVariableRedirection(string varName, string higherVarReference);
```

7 - UI presets and prefabs

class SmartButton:

```
// actual button effect
public UnityEvent onActivated; // contains callbacks
public void Activate(); // triggers the event. Shouldn't be put into onActivated.

// text handling
public string buttonText; // does nothing by its own, needs RefreshText()
public TextMeshProUGUI[] texts; // reference to all sub-elements that have text
public void RefreshText(); // assigns 'buttonText' to all elements of 'texts'
public void SetText(string str); // sets 'buttonText' then calls RefreshText()

// for debug purposes: prints to the console
public void DebugHelloWorld();
```

class SelectorManager:

```
// All managers have their static instance exposed
public static SelectorManager instance;

// Cached reference, frequently used in the script
public InputManager input;

// Input
public float axisThreshold;
public float startCooldown;
public float endCooldown;
public float cooldownEvolutionDuration;
public AnimationCurve cooldownEvolution;

// Read-only: value of Time.time when currentSelector last moved.
// Internally used by CursorBehaviour (see part 8).
public float lastShiftTimestamp;

// Input names, as defined in the InputManager
// Input will be checked differently, based on InputManager.inputMode.
public string validateButton;
public string backButton;
public string selectAxisX;
public string selectAxisY;

// Selector management
public SpatialSelector GetCurrentActiveSelector();
public void SwitchSelector(SpatialSelector newSelector);
public void TerminateSelector(); // sets the current active one to null
```

class SpatialSelector:

```
public Vector2Int size; // total grid size, (0,0) being upper-left
public Vector2Int defaultSelected; // grid index of default option
public Vector2Int currentPosition; // grid index that has focus. Read-only.
public bool memorizeLast; // if true, treats last selected as the new default
public bool wrapX; // if true, connects left and right edges
public bool wrapY; // if true, connects upper and lower edges

// if feature exists and is locked, this selector won't be navigable
public string requiredFeature;
// if context exists, it will be opened while this selector is active
public string relatedContext;

// References to selectable options (children)
public SpatialSelectorTarget[] possibleTargets;
public SpatialSelectorTarget GetCurFocusedTarget(); // if none, returns null

// indexers: return the child option located at grid position
public SpatialSelectorTarget this[int x, int y];
public SpatialSelectorTarget this[Vector2Int p];

// Events
public UnityEvent onSelectorEnabled;
public UnityEvent onSelectorDisabled;
public UnityEvent onValidate;
public UnityEvent onCancel;
public UnityEvent onMoved;
public UnityEvent onMovedUp;
public UnityEvent onMovedDown;
public UnityEvent onMovedLeft;
public UnityEvent onMovedRight;
public UnityEvent onStuck; // refers to inputs that fail, due to grid edges
public UnityEvent onStuckUp;
public UnityEvent onStuckDown;
public UnityEvent onStuckLeft;
public UnityEvent onStuckRight;

// Main controls: the following functions usually shouldn't be called manually.
// Instead, SelectorManager internally calls them in response to inputs.
// Still, they're available for niche/advanced usage, like faking any situation.
public void EnableSelector();
public void DisableSelector();
public void EnableSelector(bool ignoreContext); // if true, won't open context
public void DisableSelector(bool ignoreContext); // if true, won't close context
public bool ValidateTarget(); // 'validate' input: returns whether it worked
public bool PressCancel(); // 'cancel' input: returns whether event was nonempty
public bool Move(SelectDirection dir); // enum: .Left, .Right, .Up, .Down
public bool Focus(Vector2Int pos); // sets focus on a certain grid position
public void RegisterMouseEnter(); // ensures pointer input compatibility
public void RegisterMouseExit(); // ensures pointer input compatibility
```

class SpatialSelectorTarget:

```
// Events
public UnityEvent onGainedFocus;
public UnityEvent onLostFocus;
public UnityEvent onSelected;
public UnityEvent onOtherWasSelected;

// The remaining members of this class aren't meant to be called manually.
// Still, they're available, but only for niche/advanced usage.

// Links to parent selector that contains this
public SpatialSelector GetSelector();

// Setters don't have to be called manually, except for faking another situation,
// or relocating a pre-existing SpatialSelectorTarget.
public void SetSelector(SpatialSelector newSelector);
public void SetCoords(Vector2Int coords);

// Read-only
public bool hasFocus;

// Returns whether focus has changed.
// This doesn't affect the parent selector. It shouldn't be called manually.
// Instead, please use ObtainSelectorFocus and LoseSelectorFocus.
public bool SetFocus(bool newValue);

// The following functions don't have to be called manually:
// Instead, the prefab calls them from UnityEvents.
// In addition, they make the parent selector the active one if it's not already.
public void TurnOnSelectorAndFocus(); // called on pointer enter
public void TurnOnSelectorAndValidate(); // called on pointer click

// More utility functions that can be called from UnityEvents.
// These don't need to be called manually either.
// Safety net: they don't run if the parent selector isn't the current active one.
public void ObtainSelectorFocus(); // called on pointer enter
public void LoseSelectorFocus(); // called on pointer exit
public void ObtainSelectorValidation(); // called on pointer click
```

class CanvasFader:

```
// note: this class inherits Tweener<float>. Relevant inherited members are:
public float defaultDuration;
public AnimationCurve defaultCurve;
public UnityEvent onStarted;
public UnityEvent onComplete;
public UnityEvent onCancel;
public UnityEvent<float> onProgressChanged; // passes the new progress value (0-1)
public void Cancel();
public float GetValue(); // returns canvasGroup.alpha
public void SetValue(float newValue); // sets canvasGroup.alpha

// Reference to the related group
public CanvasGroup canvasGroup;

// Drives the "Interactable" and "Blocks Raycasts" parameters from canvasGroup.
public bool canContainInteraction;

// If true, fading in/out will also enable/disable the GameObject accordingly.
public bool handleGameObjectActivation;

// Alias for Launch(targetValue, duration), works the same
public void FadeTo(float targetValue, float duration);

// Shortcuts that use defaultDuration
public void FadeTo(float targetValue); // Expects target alpha from 0 to 1
public void FadeIn(); // Uses 1 as target
public void FadeOut(); // Uses 0 as target
public void ToggleFade(); // Knows to use 0 or 1 as target

// Shortcuts with explicit duration
public void FadeInInSeconds(float duration); // Uses 1 as target
public void FadeOutInSeconds(float duration); // Uses 0 as target
public void ToggleFade(float duration); // Knows to use 0 or 1 as target

// “Pop” means “instantly appear or disappear”, ie. with a duration of 0
public void Pop(float targetValue); // Expects target alpha from 0 to 1
public void PopIn(); // Uses 1 as target
public void PopOut(); // Uses 0 as target
public void TogglePop(); // Knows to use 0 or 1 as target

// Automatically called during events to handle interaction and activation
public void PreFadeRefresh(); // Called at onStarted
public void PostFadeRefresh(); // Called at onComplete
```

class DraggableElement:

```
public RectTransform objectToMove; // object that receives pointer movement

// Settings
public float movementMultiplier;
public bool preventOffscreenMovement;
public bool isBeingDragged; // read-only

// This implements IInitializePotentialDragHandler, IBeginDragHandler,
IDragHandler and IEndDragHandler.
// See the Unity docs for more info: Click here.
public UnityEvent onDragCanStart;
public UnityEvent onDragStarted;
public UnityEvent onDragEnded;

// Passes the movement delta, in pixels, multiplied by movementMultiplier
public UnityEvent<Vector2> onDragged;
```

class PageSystem:

```
public bool wrap; // if true, considers first page comes right after last page
public float fadeDuration;
public CanvasFader[] pages;
public CanvasFader defaultPage; // opens at start
public UnityEvent<int> onPageChanged; // passes index of new page

public int GetCurrentIndex(); // returns index of current page
public CanvasFader GetCurrentPage(); // returns page at current index
public void GoToPage(int index); // uses indexes from the ‘pages’ array
public void GoToPage(CanvasFader page); // available overload
public void GoToNextPage();
public void GoToPreviousPage();
```

class TooltipBehaviour:

```
public string tooltipTag;
public bool dynamicPosition;
public bool followMouseWhileActive; // only if dynamic position.
public Vector2 deltaTowardsScreenCenter; // only if dynamic position. In pixels.

// References
public RectTransform rectTransform;
public CanvasFader canvasFader;
public Camera relatedCamera; // used at runtime for screen-space recalculations
public Canvas relatedCanvas;

// Enum: .None, .MainCamera, .SpecifiedCamera, .SpecifiedCanvas
public TooltipCameraType relatedCameraType;

// Activation and deactivation
public bool isVisible; // read-only
public void ShowTooltip();
public void HideTooltip();
public void ToggleTooltip(); // shows if hidden, hides if shown
public void SetTooltipVisible(bool targetValue); // shows if true, hides if false

// Automatic called functions, for repositioning in the overlay canvas
public void ReturnToBaseParent(); // called after fading out
public void HookToPoint(Transform hook); // called for dynamic position
```

8 - The top-level canvas: Overlays and Cursor

class OverlayManager:

```
// All managers have their static instance exposed
public static OverlayManager instance;

// Utility: gets the camera used for screen-space operations with a given canvas.
public static Camera GetCameraFromCanvas(Canvas canvas);

// References
public RectTransform tooltipParent;
public Image monochromeImage;
public CursorBehaviour cursor;
public CanvasFader monochrome;
public CanvasFader tooltips;
public CanvasFader[] customOverlays;
public List<TooltipBehaviour> activeTooltips; // read-only
public List<GraphicRaycaster> activeRaycasters; // read-only

// Tooltip management: usually automatically called, but still available
public void RegisterTooltip(TooltipBehaviour tooltip);
public void UnregisterTooltip(TooltipBehaviour tooltip);

// Flush all tooltips. Useful when closing a panel.
public void HideAllTooltips();

// Only hides tooltips that share a common tag. Useful when closing a panel.
public void HideTooltipsWithTag(string tooltipTag);

// Retrieves a tooltip's position. Uses a different camera based on whether it's
in the overlay (arg true) or in its "home" canvas (arg false).
public Vector2 GetScreenSpacePositionOfTooltip(TooltipBehaviour tooltip, bool
isTooltipActiveInOverlay);

// Sets a tooltip's position to match a certain set of screen-space coordinates.
// Uses a different camera based on whether it's in the overlay (arg true) or in
its "home" canvas (arg false).
public void SetTooltipWorldPositionFromScreenSpace(TooltipBehaviour tooltip,
Vector2 screenSpacePos, bool isTooltipActiveInOverlay);

// Raycaster management: usually automatically called, but still available
public void RegisterRaycaster(GraphicRaycaster gr);
public void UnregisterRaycaster(GraphicRaycaster gr);

// Toggles all registered raycasters, which enables/disables UI input
public void SetRaycastersActive(bool newValue);

// (cont. next page)
```

```
// Fades
public float defaultFadeDuration; // overrides duration from the canvas groups
public void BlackFadeIn();
public void BlackFadeOut();
public void WhiteFadeIn();
public void WhiteFadeOut();
public void MonochromeFadeIn(Color color);
public void MonochromeFadeOut();
public void TooltipsFadeIn();
public void TooltipsFadeOut();
public void CustomFadeIn(int index); // uses customOverlays[index]
public void CustomFadeOut(int index);

// Providing an extra argument overrides defaultFadeDuration:
public void BlackFadeIn(float duration);
public void BlackFadeOut(float duration);
public void WhiteFadeIn(float duration);
public void WhiteFadeOut(float duration);
public void MonochromeFadeIn(Color color, float duration);
public void MonochromeFadeOut(float duration);
public void TooltipsFadeIn(float duration);
public void TooltipsFadeOut(float duration);
public void CustomFadeIn(int index, float duration); // uses customOverlays[index]
public void CustomFadeOut(int index, float duration);

// A “pop” is an instant fade, always using a duration of zero
public void TooltipsPopIn();
public void TooltipsPopOut();
public void BlackPopIn();
public void BlackPopOut();
public void WhitePopIn();
public void WhitePopOut();
public void MonochromePopIn(Color color);
public void MonochromePopOut();
public void CustomPopIn(int index); // uses customOverlays[index]
public void CustomPopOut(int index);
```

class CursorBehaviour:

```
// References
public RectTransform self;
public RectTransform overlayCanvasTransform;

// Directional Inputs
public string xAxisName; // name as defined in KeyBinding asset
public string yAxisName;
public float cursorSpeed; // in pixels per second
public bool clampPositionToScreenEdges;

// Idle Behaviour
public bool isIdle; // read-only
public float idleTime; // cursor goes idle after this much time without moving
public UnityEvent onCursorWentIdle;
public UnityEvent onCursorExitedIdle;

// After X seconds without moving, raycasters will reboot, retriggering a
potential hover event on selectors. Setting this to 0 disables this behaviour.
public float raycasterResetTime;
```

class GraphicRaycasterExposer:

```
// This component is placed on any GameObject with a Graphic Raycaster, to make
the raycaster known to the OverlayManager, so it can control it later.
// No interaction with this component is necessary.
// Only assigning the reference from the inspector is needed to perform the
registration.

public GraphicRaycaster self;
```

class ArsenalInputModule:

```
// Has the same properties and methods as the StandaloneInputManager class.
// See documentation on Unity's website: click here.
// Members marked as Obsolete in that page have also been removed.
```

class ArsenalPointerInput:

```
// This class inherits from UnityEngine.EventSystems.BaseInput: see Unity docs.

// Reference to the active input module from Event System.
// This should be a ArsenalInputModule, like included in the package.
public BaseInputModule inputModule;

// Input names as defined in the KeyBinding asset.
// Their key binding definition doesn't have to include the actual mouse click.
// These values can be left empty, in which case clicks will never be simulated.
public string simulatedLeftClick;
public string simulatedRightClick;
public string simulatedMiddleClick;
```

9 - Localization

class LocalizationManager:

```
// All managers have their static instance exposed
public static LocalizationManager instance;

// Short string identifiers, in the header of all text file
public string[] availableLanguages;

// CSV Files
public TextAsset[] sharedStrings;
public TextAsset[] stringChunks;
public TextAsset[] sharedDialogues;
public TextAsset[] dialogueChunks;

// If true, a Localizer with no key won't apply at all.
public bool disableLocalizersWithEmptyKeys;

// Called after (not before) setting the language
public UnityEvent onLanguageChanged;

// Chunk management. Loading a chunk non-additively unloads all others.
public void LoadChunk(string chunkName);
public void LoadChunkAdditive(string chunkName);
public void UnloadAllChunks();

// Get/set language. Accepts two formats:
// Short string from 'availableLanguages', or index from the same array.
public string GetCurrentLanguage();
public int GetCurrentLanguageIndex();
public void SetLanguage(string newLanguage);
public void SetLanguage(int newLanguageIndex);
public string LanguageIndexToString(int index);
public int LanguageStringToIndex(string str);

// Utility: translates a string on-the-go, without needing a Localizer.
// Example: Localize("settingsButton_Tooltip", "EN");
public string Localize(string key, string language);
public string Localize(string key, int languageIndex);
public string Localize(string key); // simpler version, uses current language

// Utility: retrieved a localized dialogue and all related data.
// Strictly identical to DialogueManager.GetDialogue(), available for clarity.
public Dialogue GetDialogue(string key, string language);
public Dialogue GetDialogue(string key, int languageIndex);
public Dialogue GetDialogue(string key); // simpler version, uses current language
```

class Localizer:

```
// Read-only. Uses current language.
public string translatedString;

// Text component references
public Text text;
public TextMeshProUGUI textTMP;

// Get/set localization key
public string GetKey();
public void SetKey(string newKey); // automatically updates text
public void SetKeyWithoutRefresh(string newKey); // leaves previous text in place

// Refreshes text components, updating texts with translation.
// Usually automatically called by SetKey().
public void RefreshText();

// Called after (not before) translating and setting text contents.
// Automatically passes translated text as argument.
public UnityEvent<string> onRefreshed;
```

10 - Dialogue System

struct Dialogue:

```
// This object contains full data for one *localized* dialogue.
// Which means, it's available for one specific language.
// For another language, there's another Dialogue value stored in game data.

public string key; // dialogue key, used by the DialogueLauncher
public string language; // language indicator, as defined in LocalizationManager
public DialoguePiece[] timeline; // chronological list of all entries
```

struct DialoguePiece:

```
// Data equivalent to one single row of a CSV file.
public DialoguePieceType pieceType; // enum: Line, Event, Option, Redirection
public string dialogueLine; // for Lines. Contains actual text.
public string speakerKey; // for Lines. Links to SpeakerDatabase.
public string eventName; // for Events. Links to name defined in DialogueLauncher.
public string redirectionKey; // for Options and Redirections.
```

class DialogueLauncher:

```
// Links to UI: targetDisplayer is only relevant if useDefaultDisplayer is false.
public bool useDefaultDisplayer;
public DialogueDisplayer targetDisplayer;

// Main function
public void LaunchDialogue();

// Get/set content
public string GetDialogueKey();
public void SetDialogueKey(string newKey);
public DialogueEvent[] dialogueEvents;
public ReplacementRule[] GetReplacementRules();
public ConditionalRule[] GetConditionalRules();
public void SetReplacementRules(ReplacementRule[] newRules);
public void SetConditionalRules(ConditionalRule[] newRules);

// Access launcher history
public string lastDialoguePlayed; // returns a dialogueKey
public int timesTalked; // how many times has the same dialogue been launched?
public void ForceResetCounter(); // resets timesTalked to zero
```

struct DialogueEvent:

```
// identifier used in spreadsheets
public string name;

// Should the text be frozen, hidden, or continue while the event plays?
// enum: KeepLastText, LaunchNextText, HideText
public TextBehaviourDuringEvent textBehaviour;

// If text is frozen/hidden, how long before it gets back to normal?
// enum: Instant, TimedHold, IndefiniteHold
public EventTiming timing;

// Delay before text resumes playing. Only relevant for EventTiming.TimedHold.
public float heldDuration;

// How does this event interact with a dialogue being skipped?
// enum: QuitFastForwarding, SkipEvent, PlayEvent
public FastForwardEventMode fastForwardBehaviour;

// Actual event content
public UnityEvent onTriggered;
```

struct ReplacementRule:

```
// Upon reading a line, <sourceString> will be replaced by <replacementMethod()>.
public string sourceString;
public Func<string> replacementMethod; // evaluated at the exact moment of display

// Constructor
public ReplacementRule(string oldString, Func<string> newString);
```

struct ConditionalRule:

```
// See part 10.8 for explanation and example.
public string startTag;
public string separatorTag;
public string endTag;
public Func<bool> checkingMethod; // evaluated at the exact moment of display

// Constructor
public ConditionalRule(string start, string separator, string end, Func<bool>
checkMethod);
```

class DialogueManager:

```
// All managers have their static instance exposed
public static DialogueManager instance;

// References
public DialogueDisplayer defaultDialogueDisplayer;
public PlayerChoiceDisplayer defaultPlayerChoiceDisplayer;

// Access speaker data
public SpeakerDatabase[] speakerDatabases;
public SpeakerData GetSpeakerData(string key);

// Fetch dialogue from key, as defined in spreadsheet.
// Strictly identical to LocalizationManager.GetDialogue(), available for clarity.
public Dialogue GetDialogue(string key);
```

class DialogueDisplayer:

```
// Dialogue References
public CanvasFader wholeDialogueGroup;
public Gauge skipGauge;
public TextMeshProUGUI[] texts;

// Player Choices
public bool useDefaultChoiceDisplayer;
public PlayerChoiceDisplayer customChoiceDisplayer; // if default isn't used

// UI Behaviour
public bool hideAtAwake;
public float dialogueFadeTime;
public float baseCharacterTime; // delay, in seconds, between two characters
public float textAcceleration; // multiplier applied when "talk" input is pressed

// Formatting
public string autoAdvanceSuffix;

// Context Matrix Names
public string dialogueContextName;
public string dialogueFeatureName;

// Input Names (as defined in the KeyBinding asset)
public string talkButtonID;
public string skipButtonID;

// (cont. next page)
```

```
// Events
public UnityEvent<string> onChangedSpeakerKey;
public UnityEvent<Dialogue> onDialogueStarted;
public UnityEvent<Dialogue> onDialogueEnded;
public UnityEvent<Dialogue> onDialogueSkipped;
public UnityEvent<Dialogue, Dialogue> onDialogueRedirected; // passes <from, to>
public UnityEvent<string> onLineStarted; // passes speaker key
public UnityEvent<string> onLineEnded; // passes speaker key

// Usually automatically called by a DialogueLauncher, but can be manually forced
public void LaunchDialogue(string key); // shortcut with no events or rules
public void LaunchDialogue(
	Dialogue dialogue,
	Dictionary<string, DialogueEvent> eventDico,
	ReplacementRule[] rRules,
	ConditionalRule[] cRules); // full overload with all settings

// Main controls on the ongoing dialogue (usually automatically called)
public void AdvanceDialogue(); // goes to next timeline entry (ie. next CSV row)
public void RedirectDialogueTo(string key); // launches a redirection to <key>
public void FastForward(); // skips to the next choice or unskippable event

// FastForward() has overloads with enum arguments.
// These help specify how to interact with redirections and choices.
public void FastForward(FastForwardRedirectionMode redirectionMode,
FastForwardOptionMode optionMode);
public void FastForward(FastForwardRedirectionMode redirectionMode);
public void FastForward(FastForwardOptionMode optionMode);

// How does skipping interact with redirections?
public enum FastForwardRedirectionMode {
	QuitFastForwarding, // skipping stops at the next redirection
	FastForwardThroughRedirection // skipping doesn't stop at the redirection
}

// How does skipping interact with player choices?
public enum FastForwardOptionMode {
	QuitFastForwarding, // skipping stops at the next player choice
	PickFirstChoice, // skipping goes through, assumes choice 1 has been picked
	PickLastChoice, // same as above, but last choice from the list
	PickRandomChoice // same as above, but random choice from the list
}
```

class SpeakerDatabase:

```
// SpeakerDatabase is simply a ScriptableObject holding data.
public SpeakerData[] speakers;
```

struct SpeakerData:

```
public string speakerKey; // as called in the CSV file
public SpeakerNameMode nameMode; // enum: Hidden, Unlocalized, UseLocaKey
public string speakerNameLocalizationKey;
public string speakerUnlocalizedName;
public SpeakerPortraitMode portraitMode; // enum: Hidden, Sprite, Prefab
public Sprite portraitSprite;
public GameObject portraitPrefab; // must carry a RectTransform, invalid otherwise
```

class SpeakerDataDisplayer:

```
// References
public Image speakerPortrait;
public RectTransform speakerPortraitPrefabHolder;
public TextMeshProUGUI speakerNameText;
public Localizer speakerNameLocalizer;
public CanvasFader portraitGroup;
public CanvasFader nameGroup;
public GameObject currentPortraitPrefabInstance;

// Settings
public float fadeTime;

// Main function, has two overloads
public void LoadSpeakerInfo(string speakerKey);
public void LoadSpeakerInfo(SpeakerData speakerInfo);
```

class PlayerChoiceDisplayer:

```
// Behaviour
public bool hideAtAwake;
public float fadeDuration;
public string contextName;
public UnityEvent onBuildDisplayer;

// References
public RectTransform self;
public SpatialSelector choiceSelector;
public CanvasFader choicePanelGroup;
public PlayerChoice[] choiceObjects;

// The DialogueDisplayer that opened this, if any.
// Flushed after use by the PlayerChoice object.
public DialogueDisplayer boundDialogueDisplayer;

// Creates and enables the displayer. Usually automatically called.
public void BuildDisplayer(List<DialoguePiece> pieces);

// Called when the player has selected a choice and the panel can disappear.
public void TerminateDisplayer();

// An example of a method that could be called by the onBuildDisplayer event.
// Resizes the displayer background so all options are the same size.
// This example assumes a vertical layout, left-aligned.
// See setup in the sample prefab.
public void AdjustSizes();
```

class PlayerChoice:

```
// References
public RectTransform rectTransform; // the object itself
public RectTransform textTransform; // text holder, child of this object
public TextMeshProUGUI text;
public PlayerChoiceDisplayer parentDisplayer;

// Data
public UnityEvent<DialoguePiece> onLoadChoiceContent; // called on initialization
public DialoguePiece correspondingOption;

// Usually automatically called. Initializes the option along with the displayer.
public void LoadDialoguePiece(DialoguePiece newPiece);

// Usually automatically called when the option is selected. Closes the displayer.
public void ValidateOption();
```

11 - Save System

class SaveManager:

```
// All managers have their static instance exposed
public static SaveManager instance;

// Slot management
public int currentSaveSlotIndex; // read-only
public int numberOfSaveSlots;
public void SwitchSlot(int newValue); // switches the active slot
public void DuplicateSlot(int src, int dst); // copy contents of src into dst
public void DeleteSlot(int slot); // Erases contents of a slot. (all channels)

// Per-channel management
public SaveChannel GetChannel(string channelName);
public SaveChannel GetChannel(int index);

// Most common operations
public void SaveGame(string channel);
public void LoadGame(string channel);
public void ResetToDefault(string channel); // sets JSON to factory settings
public void EraseSavefile(string channel); // destroys JSON file

// Fake a gamestate for debug purposes
public void LoadDebugFile(string channel, SaveFile savefile);

// Overloads available for those operations:
public void SaveGame(int channelIndex);
public void LoadGame(int channelIndex);
public void ResetToDefault(int channelIndex);
public void EraseSavefile(int channelIndex);
public void LoadDebugFile(int channelIndex, SaveFile savefile);
public void SaveGame(SaveChannel channel);
public void LoadGame(SaveChannel channel);
public void ResetToDefault(SaveChannel channel);
public void EraseSavefile(SaveChannel channel);
public void LoadDebugFile(ref SaveChannel channel, SaveFile savefile);

// Access path where JSON files are stored
public string folderPath; // read-only, contains full path
public string saveFolderName;

// prevents users from time-traveling
public int versionNumber;

// (cont. next page)
```

```
// One callback for each data transfer: json <-> asset <-> game.
// Events automatically pass the involved channel as an argument.
// SaveManagerErrorType is an enum. For now, it only knows .WrongVersionNumber.
public UnityEvent<SaveChannel> onLoadedJsonToAsset;
public UnityEvent<SaveChannel> onLoadedAssetToGame;
public UnityEvent<SaveChannel> onSavedGameToAsset;
public UnityEvent<SaveChannel> onSavedAssetToJson;
public UnityEvent<SaveChannel, SaveManagerErrorType> onError;
```

struct SaveChannel:

```
public string name;

// if true, won't use the slot system and will generate one single JSON file,
instead of generating one file per slot
public bool isSharedBetweenAllSlots;

// if true, will load this channel at game start
public bool loadOnAwake;

// contents of this file don't matter, since it will change at runtime
public SaveFile boundFile;

// holds factory settings
public SaveFile defaultSaveFile;
```

class SaveFile:

```
// SaveFile inherits from ScriptableObject.
// All your custom save classes will inherit from SaveFile.

// Prevents time travel
public int versionNumber;

// Reads the content of this file and updates the game state accordingly.
// Override this in your own class.
public virtual void LoadIntoGame();

// Reads the game state and updates the content of this file accordingly.
// Override this in your own class.
public virtual void SaveFromGame();
```

12 - A sample use case for Savefiles: the Settings Panel

class SettingsSave:

```
// This class inherits SaveFile. It contains save data for settings.
public int masterTheme; // Mapped on ThemeManager.availableThemes indexes
public int fullscreenMode; // Mapped on Unity's FullScreenMode enum
public int resolutionWidth;
public int resolutionHeight;
public int languageIndex;
public float masterVolume;
public float bgmVolume;
public float sfxVolume;

// Names that should point to AudioMixerGroups used by the SoundManager
public string masterGroupName;
public string bgmGroupName;
public string sfxGroupName;
```

class SettingsPanel:

```
// A MonoBehaviour at the root of the panel object, running its global behaviour.
public CanvasFader canvasFader; // reference to self, for fades
public BaseSettingSwitcher[] switchers; // reference to panel contents

// Name of the "close" button, as defined in the InputManager.
// Input will be checked differently, based on InputManager.inputMode.
public string closeButtonName;

// Name of the channel used by the SaveManager
public string saveChannelName;

// Context, as defined in ContextManager, will be opened while the panel is.
public string relatedContextName;

// Opening the panel counts as this feature, as defined in the ContextManager.
public string relatedFeatureName;

// Open or close the panel: (include fades and context update)
public void OpenPanel();
public void ClosePanel();

// The following methods are automatically called by UnityEvents from buttons, or
from OpenPanel() and ClosePanel():
public void Initialize(); // Reads game state and adapts display
public void Apply(); // Reads switchers and adapts game state
public void Revert(); // Resets switchers to where they were before editing
public void FactorySettings(); // Resets whole panel to default settings
public void FlushTooltips(); // Removes tooltips tagged "settingsPanel"
```

class BaseSettingsSwitcher:

```
// A MonoBehaviour that connects the game state, the SaveFile asset and a UI text.
// For each parameter, there is a separate script that inherits from this class.

public TextMeshProUGUI text; // Reference to the UI object displaying text
public string saveChannelName; // Name of the channel used by the SaveManager
public int currentIndex; // Index of the value currently selected by this switcher

// Automatically called by UnityEvents when clicking the "<" or ">" buttons.
// These point to other functions that update game state and/or display.
public void GoToPreviousValue();
public void GoToNextValue();

// Describes the number of possible different values. Useful for wrapping.
public virtual int GetNumberOfIndexes();

// Handles display in the UI (text)
public virtual string IndexToString(int index);

// Switcher logic: reads the game state and associates it to an index
public virtual int GetIndexBasedOnSaveFile();

// Refreshes the display (text) based on the current game state
public void SyncWithSaveFile();

// Sets the game state based on the currently wanted index.
public void ApplyCurrentIndex();

// Applies setting to the actual game
public virtual void ApplyIndexToGameState(int index);

// Applies setting to the SaveFile asset. Does NOT mean the JSON is modified!
public virtual void ApplyIndexToSaveFile(int index);
```

class MasterThemeSwitcher:

```
// Switcher for the game's active MasterTheme. Inherits from BaseSettingsSwitcher.

// Name of the theme variable storing the MasterTheme's name.
// The theme variable value is stored as a localization key.
public string nameVar;

// Localization key corresponding to the "None" entry, in case of an error.
public string errorKey;
```

class FullscreenSwitcher:

```
// Switcher for the game's FullScreenMode. Inherits from BaseSettingsSwitcher.

// Maps each value of the FullScreenMode enum to a localization key.
public string windowedKey;
public string fsWindowKey;
public string exclusiveFSKey;
public string maxWindowKey;
```

class LanguageSwitcher:

```
// Switcher for the game's language. Inherits from BaseSettingsSwitcher.
// Draws the list of languages from LocalizationManager, and has no other member.
```

class ResolutionSwitcher:

```
// Switcher for the game's resolution. Inherits from BaseSettingsSwitcher.
public Vector2Int[] resolutions;
```

class VolumeSwitcher:

```
// Base Switcher for the game's audio volume. Inherits from BaseSettingsSwitcher.

// Group name as declared in the AudioMixer used by SoundManager
public string mixerGroupName;

// Clicking the button once will increase or decrease volume by this value.
// Volume goes from 0 to 1.
public float increment; // defaults to 0.05f
```

class MasterVolumeSwitcher:

```
// Switcher for the game's master volume. Inherits from VolumeSwitcher.
```

class MusicVolumeSwitcher:

```
// Switcher for the game's music volume. Inherits from VolumeSwitcher.
```

class SFXVolumeSwitcher:

```
// Switcher for the game's sound effect volume. Inherits from VolumeSwitcher.
```