<XmlLayout>

USER GUIDE

V1.00 (May 2016)

TABLE OF CONTENTS

INTRODUCTION

XmlLayout allows you to create fully functional user interfaces and UI elements using XML.

- Utilise all of the functionality of UnityGUI through XML
- Supports event handling, on any element (onClick, onMouseEnter, onMouseExit, onValueChanged, etc.)
- Easily retrieve form data from multiple elements at once with XmlLayout.GetFormData()
- Access and create Xml Elements dynamically at runtime
- Includes Xml Scheme Definition (XSD) for intellisense support (autocomplete).
- Set default values to be used by other elements (e.g. visual styles)

Getting Started

Basic Concepts

XmlLayout parses Xml, and produces fully functional Unity UI elements as a result.

The system comprises of several components:

- **XmlLayout**

  This is the primary component of the XmlLayout system. It parses the Xml, and produces the resulting UI elements.

- **Xml File** (optional)

  The XmlLayout component can optionally use an Xml File (highly recommended) instead of specifying the Xml code directly.

- **XmlLayoutController** (optional)

  The Xml Layout Controller is responsible for:

  - Loading dynamic data
  - Creating elements dynamically at run-time
  - Handling events

  Each Xml Layout Controller is a custom class which must be created specifically for the XmlLayout it will be tied to.

  Xml Layout Controllers must extend (inherit from) the XmlLayoutController class, which provides several virtual methods you may override to customise the behaviour of the controller.

  - LayoutRebuilt()

    This method is called after the XmlLayout object is rebuilt for any reason (e.g. Xml changed, RectTransform dimensions changed, Rebuild on Awake, etc.)

    As such, overriding this method is ideal for populating dynamic data fields, creating dynamic elements, and so on.

- Show()

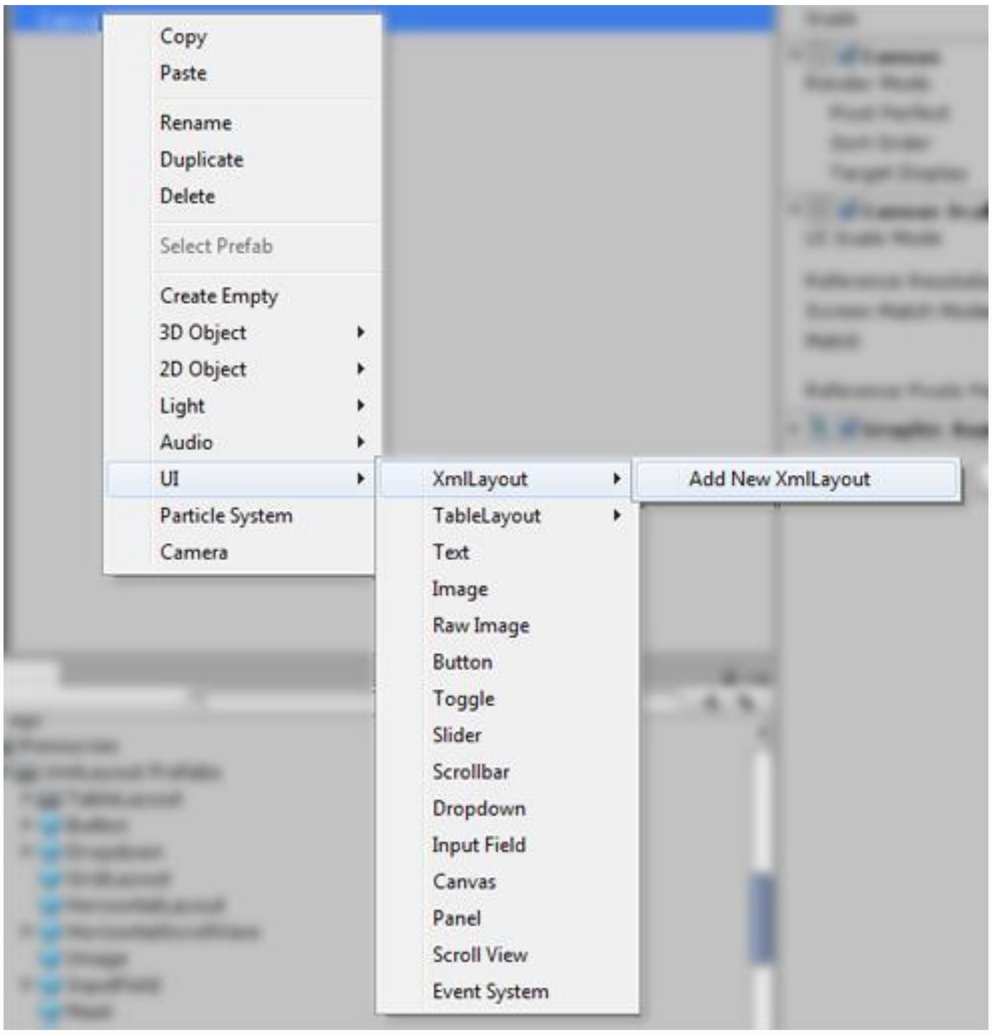
  This method will, by default, show the XmlLayout object. You may override it to add custom behaviour.

- Hide()

  This method will, by default, hide the XmlLayout object. You may override it to add custom behaviour.

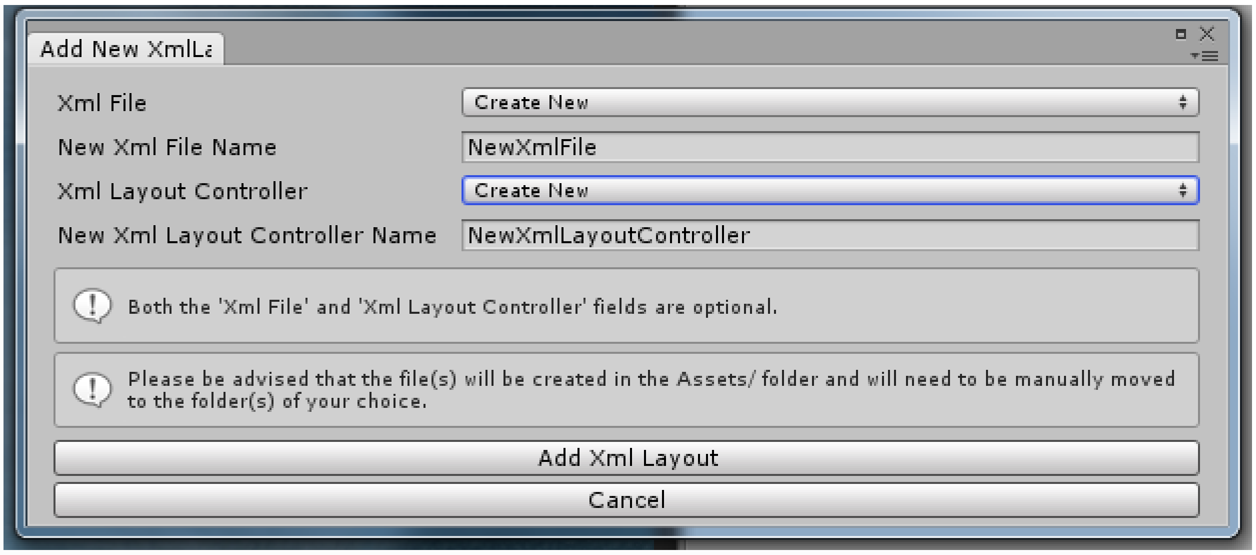
ADDING A NEW XMLLAYOUT OBJECT

To add a new XmlLayout object, right-click the desired location in the hierarchy window, and select **UI -> XmlLayout -> Add New XmlLayout**.



Adding a new XmlLayout

This will then open the 'Add New XmlLayout' Wizard.



Xml File: Create New
New Xml File Name: NewXmlFile
Xml Layout Controller: Create New
New Xml Layout Controller Name: NewXmlLayoutController

Both the 'Xml File' and 'Xml Layout Controller' fields are optional.

Please be advised that the file(s) will be created in the Assets/ folder and will need to be manually moved to the folder(s) of your choice.

Add New XmlLayout Wizard

Options

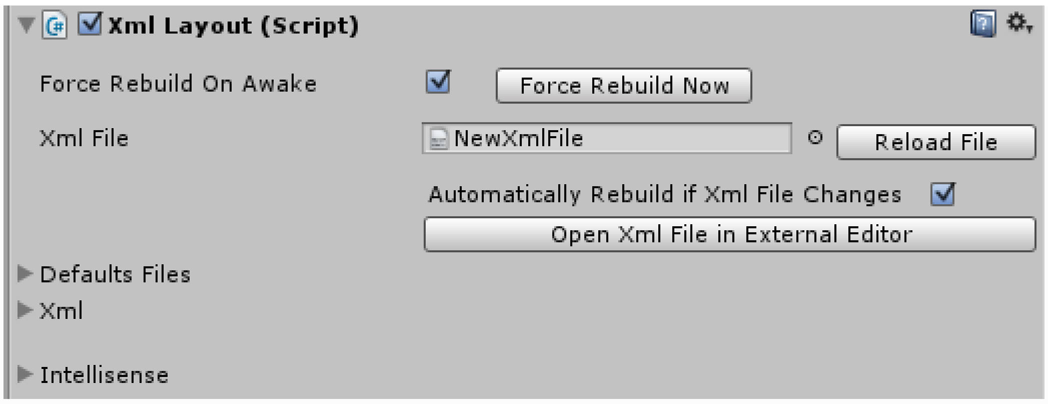
- **Xml File**:
    - *None*: The new XmlLayout will not use an Xml File.
    - *Create New*: The new XmlLayout will use a newly-created Xml File.
    - *Select Existing*: The new XmlLayout will use an existing Xml File.
- **New Xml File Name**: If creating a new Xml File, this field specifies the name of the new file.
- **Xml Layout Controller**
    - *None*: The new XmlLayout will not use an Xml Layout Controller.
    - *Create New*: The new XmlLayout will use a newly-created Xml Layout Controller.
    - *Select Existing*: The new XmlLayout will use an existing Xml Layout Controller.
- **New Xml Layout Controller Name**: If creating a new Xml Layout Controller, this field specifies the name of the new controller.

Once you have made your selections, click the **Add Xml Layout** button. If you have opted to create a new Xml Layout Controller, the Unity Editor will pause briefly while it re-compiles.

The newly created xml file and/or Xml Layout Controller file will be created in the Assets/ directory - you will need to move them to the directories of your choice.

And there you go, your newly created XmlLayout is ready for you to start coding!

The XmlLayout Component



The XmlLayout Component

- **Force Rebuild On Awake**

  If this is set, then the first time this XmlLayout becomes active in the scene, it will rebuild its contents from the Xml. This may cause a small delay the first time this XmlLayout object becomes active, but it does ensure that everything is always up-to-date.

- **Force Rebuild Now**

  If this button is clicked, then this XmlLayout will be rebuilt immediately.

- **Xml File**

  This specifies the Xml file to be used by this XmlLayout. This field is optional - if you wish, you may specify the Xml for this XmlLayout directly.

- **Reload File**

  If this button is clicked, this XmlLayout's Xml will be updated to match the contents of the Xml File, and then this XmlLayout will be rebuilt.

- **Automatically Rebuild If Xml File Changes**

  If this is set, then whenever the Xml File is changed externally (for example, in Visual Studio), then so long as the Editor is running and this project is open, this XmlLayout will be updated to match.

  Please note that this will also occur if the editor is running in Play mode, but once you return to Edit mode, the changes will be lost - so it will be necessary to click the 'Reload File' button in order to make the changes permanent.

- **Open Xml File in External Editor**

  Clicking this button will open the Xml File in the external editor that the Unity Editor has been configured to use (i.e. Visual Studio or MonoDevelop).

- **Defaults Files**

  This is an optional list of Xml files to load default element values from (See the 'Defaults' tag). Defaults files can also be specified within the Xml (See the 'Include' tag).

- **Xml**

  This field allows you to manually specify the Xml used by this XmlLayout object.

  - *Update Element*: Clicking this button will update the XmlLayout element to match the updated Xml.
  - *Save Changes to Xml File*: Clicking this button will save any changes made to the Xml File.

- **Intellisense**

  This section allows you to view/edit the XSD document used by XmlLayout, should you wish to do so.

AUTOCOMPLETE / INTELLISENSE

Visual Studio / MonoDevelop loads the XmlLayout intellisense data from an XSD document located in:

`UI/XmlLayout/Configuration/XmlLayout.xsd`

As such, each XmlLayout XmlLayout tag should be set up as follows, where `xsi:noNamespaceSchemaLocation` is the *relative* path from the Xml File to the XSD document.

```
<XmlLayout xmlns="http://www.w3schools.com"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="UI/XmlLayout/Configuration/XmlLayout.xsd">
```

EVENT HANDLING

Please note that in order to use event handling, the *Force Rebuild On Awake*property of the XmlLayout must be set to **true**.

Most Xml Elements provide the `onClick`, `onMouseEnter`, and `onMouseExit`event-handlers. Elements which are used to input values (such as InputField, Slider, Toggle, etc.) provide an additional event-handler: `onValueChanged`. Additionally, some elements have unique event-handlers, such as InputField's`onEndEdit`.

All events will be triggered on the XmlLayout's XmlLayoutController - as such, event-handling can only be used on XmlLayout elements with a controller attached.

Event field structure:

`MethodName(optional argument);`

The semi-colon is optional, as are the brackets if no argument is specified.

EXAMPLE XML

```
<!-- XmlLayoutController.ButtonClicked() will be called when
     this button is clicked -->

<Button onClick="ButtonClicked();">Button Text</Button>


<!-- Whenever this slider's value changes,
    XmlLayoutController.SliderValueChanged(x) will be called,
    where 'x' is the new value -->

<Slider onValueChanged="SliderValueChanged(selectedValue);" />


<!-- The XmlLayoutController.HighlightButton() and
     XmlLayoutControllerClearButtonHighlight() methods
     will be called when the mouse enters and exits this element.

     The button itself will be passed as an argument. -->

<Button onMouseEnter="HighlightButton(this);" onMouseExit="ClearButtonHighlight(th
is);">Button Text</Button>


<!-- XmlLayoutController.ButtonClickedTwo(aStringValue) will be
     called when this button is clicked -->

<Button onClick="ButtonClickedTwo(aStringValue);">Button Text</Button>
```

Defining methods to receive events in the controller:

Methods which receive events from the XmlLayout will be located within the XmlLayoutController through the use of Reflection. It is not necessary for the methods to be public (although they can be, should you wish to make them).

The event-handling code will attempt to convert any arguments into the type the method expects, e.g. if the method expects an integer, then the value provided will be parsed first.

You may name your event-receiving methods whatever you wish, and you may even call inherited methods (such Show() and Hide()) if you wish.

Special Arguments

By default, arguments will simply be converted to the required type, e.g. if a value of 'xyz' is passed by the Xml as the argument, and the method expects a string, then the method will just be passed the 'xyz' value as a string. However, some string values have special meanings:

- **this**: The game object which raised the event will be passed as an argument. The event-handling code will attempt to convert it into the type required by the method, through the use of gameObject.GetComponent().
- **value**: (InputField only) The current text value of this InputField.
- **selectedValue**: (Toggle/ToggleButton/ToggleGroup/Slider) The currently selected int/float value of this element.
- **selectedIndex**: (Dropdown only) The currently selected index of this Dropdown element.
- **selectedText**: (Dropdown only) The currently selected text value of this Dropdown element.
- **xy**: (Scrollviews only) A Vector2() representing this scrollview's current x and y values.
- **x**: (Scrollviews only) A float representing this scrollview's current x value.
- **y**: (Scrollviews only) A float representing this scrollview's current y value.

EXAMPLE CONTROLLER

```
void ButtonClicked()
{
    Debug.Log("The button was clicked.");
}

void SliderValueChanged(float newValue)
{
    Debug.Log("The slider's value was changed to " + newValue + ".");
}

void HighlightButton(Button button)
{
    // Code to highlight the button
}

void ClearButtonHighlight(Button button)
{
    // Code to clear the button highlight
}

void ButtonClickedTwo(string stringValue)
{
    // Will output: 'ButtonClickedTwo - value received = 'aStringValue'
    Debug.Log("ButtonClickedTwo - value received = '" + stringValue + "'");
}
```

CREATING ELEMENTS DYNAMICALLY

There are several methods which you can use to create elements dynamically. The easiest, and perhaps the most efficient, is to create a hidden template element and clone it, as done in the `Example Menu` and `Shop` examples.

This works well, because you can easily style and clone the template as with any other Xml element.

While not strictly necessary, it is recommended that you add dynamically created elements to layout groups to minimise the effort it takes to position and scale them.

EXAMPLE XML

```
<VerticalLayout padding="10" spacing="10" id="menuButtons">

    <!-- This is a template which will be used by the XmlLayoutController to creat
e the menu buttons dynamically -->

    <Button id="menuButtonTemplate" active="false"></Button>

</VerticalLayout>
```

EXAMPLE CONTROLLER

```
public override void LayoutRebuilt(ParseXmlResult parseResult)
{
    if (parseResult != ParseXmlResult.Changed) return;

    // get the menu button container
    menuButtonGroup = xmlLayout.GetElementById("menuButtons");

    // get the menu button template so that we can clone it
    var menuButtonTemplate = xmlLayout.GetElementById("menuButtonTemplate");

    // in this case, 'Examples' is populated in the editor
    foreach (var example in Examples)
    {
        var name = example.name;

        // Create a copy of the template
        var menuButton = GameObject.Instantiate(menuButtonTemplate);
        menuButton.name = name;

        // Access the XmlElement component and initialise it for this new button
        var xmlElement = menuButton.GetComponent<XmlElement>();
        xmlElement.Initialise( xmlLayout,
                               (RectTransform)menuButton.transform,
                               menuButtonTemplate.tagHandler);


        // Add the xmlElement to the menuButtonGroup
        menuButtonGroup.AddChildElement(menuButton);


        // Set the necessary attributes, and Apply them
        xmlElement.SetAttribute("text", name);

        // the template is inactive (so as not to be visible),
        // so we need to activate this button
        xmlElement.SetAttribute("active", "true");

        // Call the SelectExample function (in this XmlEventReceiver)
        // when this button is clicked
        xmlElement.SetAttribute("onClick", "SelectExample(" + name + ");");

        xmlElement.ApplyAttributes();

    }
}
```

WORKING WITH FORM DATA

Receiving values for individual elements using event-handlers is easy, but what if you need to retrieve the values of multiple elements at once (e.g. for a login form, or an options menu)?

To this end, XmlLayout provides the `GetFormData()` method, which returns the values of all the form elements in a`Dictionary<string,string>` (where they key is the id value of the element).

The `Options Menu` example demonstrates how form data can be read easily using XmlLayout.GetFormData().

XMLLAYOUT TAGS

ATTRIBUTE TYPES

Most attribute value types are straightforward, however, there are a few which warrant an explanation.

- **Color**

  For color values, several formats are acceptable:

  - **HTML (6 character hex) color codes** e.g. `#FFFFFF` (white 100% opacity)
  - **8 character hex color codes** e.g. `#FFFFFFCC` (white 80% opacity)
  - **RGB color codes:** e.g. `rgb(1,1,1)` (white 100% opacity)
  - **RGBA color codes:** e.g. `rgba(1,1,1,0.8)` (white 80% opacity)

- **Color Block**

  Color block values are used to specify the colors for elements such as buttons and input fields.

  Format: (normalColor|highlightedColor|pressedColor|disabledColor) where each color is formatted as above,
  e.g. `#FFFFFF|#FFFFFF|#C8C8C8|rgba(0.78,0.78,0.78,0.5)`

- **Paths**

  All paths are passed through Unity's `Resources.Load()`, and as such any referenced Sprites, AudioClips, Prefabs, etc. *must* be located within a Resources folder.

- **Bool**

  Boolean attributes can accept `1` (true), `true`, `0` (false), or `false`.

- **RectOffset**

  RectOffsets, e.g. padding, accept the following formats:

  - float(left) float(right) float(top) float(bottom) e.g. `5 5 10 10` (padding value of 5 on the left and right sides, and 10 on the top and bottom)
  - float (all sides) e.g. `10` (padding value of 10 on all sides)

Common Attributes

These attributes can be applied to most Xml Elements.

Attributes

Name	Description	Type / Options	Default Value
Identification			
id	Specifies an id for this element which can be used to access it later on, via XmlLayout.GetElementById(). This value should be unique within each XmlLayout.	string	
internalId	Specifies an internal id for this element which can be used to access it later on, via XmlElement.GetElementByInternalId(). This can be used to access child elements of any XmlElement.	string	
name	Specifies the name of this element within the Unity Hierarchy.	string	
Defaults			
class	Specifies the class(es) of this element, which allows you to use default values defined within a tag.	list of classes separated by spaces	
Active			

active	Specifies whether or not this element is active.	bool	true
Image **(Only applies to elements with an image component)**			
image	Image to use for this element	Path/To/Image	Varies per element
color	Color for this element's image	color	clear or #FFFFFF
type	Image Type	▪ Simple ▪ Sliced ▪ Filled ▪ Tiled	Varies per element, usually 'Sliced'
raycastTarget	Should this element block raycasts or not?	bool	true
Event Handling			
onClick	Defines a method to be called when this element is clicked.	MethodName(argument);	
onMouseEnter	Defines a method to be called when the mouse enters this element.	MethodName(argument);	
onMouseExit	Defines a method to be called when the mouse exits this element.	MethodName(argument);	
Appearance			
shadow	Defines the shadow color of this element	color	None

shadowDistance	Defines the distance of the shadow for this element	float(x) float(y)	1 -1
outline	Defines the outline color of this element	color	None
outlineSize	Defines the size of this elements outline	float(x) float(y)	1 -1
prefabPath	Allows you to use a non-standard prefab for any element.	Path/To/Prefab	Varies per element
Layout Elements (Only applies to elements within a layout group)			
ignoreLayout	Should this element ignore its parent layout group?	bool	false
minWidth	Minimum width for this element	float	
minHeight	Minimum height for this element	float	
preferredWidth	Preferred width for this element	float	
preferredHeight	Preferred height for this element	float	
flexibleWidth	Should the width of this element be flexible?	▪ 1 ▪ 0	
flexibleHeight	Should the height of this element be flexible?	▪ 1 ▪ 0	

Position and Size (Simple)			
rectAlignment	Defines this elements position within its parent. Only applies to elements not contained within layout groups.	▪ UpperLeft ▪ UpperCenter ▪ UpperRight ▪ MiddleLeft ▪ MiddleCenter ▪ MiddleRight ▪ LowerLeft ▪ LowerCenter ▪ LowerRight	MiddleCente r
width	Defines the width of this element.	▪ float (fixed width) ▪ or a Percentage value	100%
height	Defines the height of this element.	▪ float (fixed width) ▪ or a Percentage value	100%
offsetXY	Defines an offset to the position of this element, e.g. a value of `-32 0`will cause this element to be 32 pixels to the left of where it would otherwise be.	float (x) float (y)	0 0
Position and Size (Advanced RectTransform properties)			
anchorMin		float(x) float(y)	
anchorMax		float(x) float(y)	
sizeDelta		float(x) float(y)	
pivot		float(x) float(y)	
rotation		float(x) float(y) float(z)	
scale		float(x) float(y)	

Dragging			
allowDragging	Allow this element to be dragged? (Note: does not work on child elements of layout groups)	bool	false
restrictDraggingToParentBounds	Prevent this element from being dragged outside of its parent?	bool	true
Animation			
showAnimation		▪ None ▪ Grow ▪ FadeIn ▪ SlideIn_Left ▪ SlideIn_Right ▪ SlideIn_Top ▪ SlideIn_Bottom	None
hideAnimation		▪ None ▪ Shrink ▪ FadeOut ▪ SlideOut_Left ▪ SlideOut_Right ▪ SlideOut_Top ▪ SlideOut_Bottom	None
showAnimationDelay	Adds a short delay before playing this element's show animation.	float	0
hideAnimationDelay	Adds a short delay before playing this element's hide animation.	float	0
Audio			
onClickSound	Audioclip to play when this element is clicked.	Path/To/AudioClip	None

onMouseEnterSound	Audioclip to play when the mouse enters this element.	Path/To/AudioClip	None
onMouseExitSound	Audioclip to play when mouse exits this element.	Path/To/AudioClip	None
audioVolume	Volume to play Click/MouseEnter/MouseExit Audioclips at.	float	1

XMLLAYOUT

The XmlLayout tag is the *root* tag of any XmlLayout Xml document - it must be present, and only elements contained within the XmlLayout tag will be parsed.

ATTRIBUTES

Name	Description	Type / Options	Default Value
xmlns			http://www.w3schools.com
xmlns:xsi			http://www.w3.org/2001/XMLSchema-instance
xsi:noNamespaceSchemaLocation			*Relative Path to XmlLayout.xsd*

Please note that for the XmlLayout tag, these values must all be set in order to use Intellisense in Visual Studio

Defaults

The defaults Tag allows you to set default values for UI elements. Primarily this will be used to set styles and the like, but there is no restriction on what default values you can set.

Defaults can be applied to all instances of a particular tag type, or only those of a particular class (as set by the class attribute). Please note that elements may use more than one class (separated by spaces).

As with its HTML counterpart (CSS), XmlLayout Defaults are applied in a *cascading* fashion. This means that an element will always use the *most recent* value for an attribute - for example, if an element implements a class, it will use the attribute values defined by that class *except* when the element itself also defines those attributes (attributes defined on the element will always take precedence).

Defaults tags can be placed anywhere in the Xml document, but will only apply to elements *after* it.

Example

```
<Defaults>
    <!-- Set the default color and font size for all Text elements -->
    <Text color="#DDDDDD" fontSize="16" />

    <!-- Set the default color for all Text elements using the 'darker' class -->
    <Text class="darker" color="#AAAAAA" />

    <!-- Set the default background image for all Button images -->
    <Button image="Sprites/Layout/Button" />
</Defaults>

<!-- This text's color will be "#DDDDDD" and its font size will be "16" -->
<Text>Text</Text>

<!-- This text's color will be "#AAAAAA" and its font size will be "16" -->
<Text class="darker">Text</Text>

<!-- This button will use the background image "Sprites/Layout/Button" -->
<Button>Button Text</Button>
```

Attributes

None

INCLUDE

The Include tag allows you to load the contents of another XmlFile and include them within the current Xml document. This is commonly used to load Defaults files, but can be used to include any other Xml document.

EXAMPLE

```
<Include path="Xml/Styles.xml" />
```

ATTRIBUTES *(CLICK TO SHOW/HIDE)*

Name	Description	Type / Options	Default Value
path	Specifies the path to the Xml file. Please note that the Xml file **must**be located within a Resources folder.	Path/To/XmlFile	

BASIC ELEMENTS

TEXT

The text tag is a Unity UI `Text` object.

This tag supports *Rich Text* as shown in the following example:

EXAMPLE

```xml
<!-- Standard Text element -->
<Text>Some Text</Text>


<!-- Rich Text -->
<Text>
    This text is <b>Bold</b>, <i>Italic</i>,
    and <textcolor color="#00FF00">Green</textcolor>.
    <br>
    This text is <textsize size="18">Larger</textsize>.
</Text>
```

ATTRIBUTES

Name	**Description**	**Type / Options**	**Default Value**
alignment		▪ UpperLeft ▪ UpperCenter ▪ UpperRight ▪ MiddleLeft ▪ MiddleCenter ▪ MiddleRight ▪ LowerLeft ▪ LowerCenter ▪ LowerRight	MiddleCenter
color		color	#323232
font		Path to font resource	Arial

fontStyle		▪ Normal ▪ Bold ▪ Italic ▪ BoldItalic	Normal
fontSize		float	14
resizeTextForBestFit	Resize text to fit?	bool	false
resizeTextMinSize	Minimum font size	float	10
resizeTextMaxSize	Maximum font size	float	40
horizontalOverflow		▪ Wrap ▪ Overflow	Overflow
verticalOverflow		▪ Truncate ▪ Overflow	Truncate

IMAGE

The image tag is a Unity UI `Image` object.

EXAMPLE

```
<Image image="Path/To/Image" />
```

ATTRIBUTES (CLICK TO SHOW/HIDE)

Name	**Description**	**Type / Options**	**Default Value**
image		Path/To/Sprite	None
color		Color	#FFFFFF

type	Image Type	▪ Simple ▪ Sliced ▪ Filled ▪ Tiled	Simple
raycastTarget	Should this image block raycasts or not?	bool	true

Layout Elements

Panel

The panel tag is a simple layout element.

Example

```
<Panel>
    <Text>Text contained within Panel</Text>
</Panel>
```

Attributes

Name	Description	Type / Options	Default Value
padding	Specifies the padding for this panel. Please note that if padding is specified, the panel will function as a LayoutGroup (which it does not do by default).	float(left) float(right) float(top) float(bottom)	None

HorizontalLayout

The HorizontalLayout tag is a Unity UI `HorizontalLayoutGroup` object.

Example

```
<HorizontalLayout>
    <Button>Button One</Button>
    <Button>Button Two</Button>
    <Button>Button Three</Button>
</HorizontalLayout>
```

ATTRIBUTES

Name	Description	Type / Options	Default Value
padding		float(left) float(right) float(top) float(bottom)	0 0 0 0
spacing	Spacing between child elements	float	0
childAlignment		▪ UpperLeft ▪ UpperCenter ▪ UpperRight ▪ MiddleLeft ▪ MiddleCenter ▪ MiddleRight ▪ LowerLeft ▪ LowerCenter ▪ LowerRight	UpperLeft
childForceExpandWidth		bool	true
childForceExpandHeight		bool	true

VERTICALLAYOUT

The VerticalLayout tag is a Unity UI `VerticalLayoutGroup` object.

EXAMPLE

```
<VerticalLayout>
    <Button>Button One</Button>
    <Button>Button Two</Button>
    <Button>Button Three</Button>
</VerticalLayout>
```

ATTRIBUTES

Name	Description	Type / Options	Default Value
padding		float(left) float(right) float(top) float(bottom)	0 0 0 0
spacing	Spacing between child elements	float	0
childAlignment		▪ UpperLeft ▪ UpperCenter ▪ UpperRight ▪ MiddleLeft ▪ MiddleCenter ▪ MiddleRight ▪ LowerLeft ▪ LowerCenter ▪ LowerRight	UpperLeft
childForceExpandWidth		bool	true
childForceExpandHeight		bool	true

GRIDLAYOUT

The GridLayout tag is a Unity UI `GridLayoutGroup` object.

EXAMPLE

```
<GridLayout>
    <Button>Button One</Button>
    <Button>Button Two</Button>
    <Button>Button Three</Button>
</GridLayout>
```

Attributes

Name	Description	Type / Options	Default Value
padding		float(left) float(right) float(top) float(bottom)	0 0 0 0
spacing	Spacing between child elements	float(x) float(y)	0 0
cellSize		float(x) float(y)	100 100
startCorner		▪ UpperLeft ▪ UpperRight ▪ LowerLeft ▪ LowerRight	UpperLeft
startAxis		▪ Horizontal ▪ Vertical	Horizontal
childAlignment		▪ UpperLeft ▪ UpperCenter ▪ UpperRight ▪ MiddleLeft ▪ MiddleCenter ▪ MiddleRight ▪ LowerLeft ▪ LowerCenter ▪ LowerRight	UpperLeft
constraint		▪ Flexible ▪ FixedColumnCount ▪ FixedRowCount	Flexible
constraintCount		integer	2

TableLayout

The TableLayout tag is a layout element based on HTML tables, allowing you to specify the position of elements in specific rows/columns.

Example

```
<TableLayout>
    <!-- Row 1 -->
    <Row>
        <Cell><Button>Button One</Button></Cell>
        <Cell><Button>Button Two</Button></Cell>
    </Row>
    <!-- Row 2 -->
    <Row>
        <Cell><Button>Button One</Button></Cell>
        <Cell><Button>Button Three</Button></Cell>
    </Row>
</TableLayout>
```

Attributes

Name	**Description**	**Type / Options**	**Default Value**

padding		float(left)
		float(right)
		float(top)
		float(bottom)
			0 0 0 0

cellSpacing	Spacing between each cell.	float	0
columnWidths	(Optional) Explicitly set the width of each column. Use a value of 0 to auto-size a specific column.	float list - e.g. '32 0 0 32'	
automaticallyAddColumns	If more cells are added to a row than are accounted for by*columnWidths*, should this TableLayout automatically add one or more new auto-sized entries (0) to *columnWidths*?	bool	true

automaticallyRemoveEmptyColumns	If there are more entries in*columnWidths* than there are cells in any row, should this TableLayout automatically remove entries from*columnWidths* until their are no 'empty' columns?	bool	true
autoCalculateHeight	If set to true, then the height of this TableLayout will automatically be calculated as the **sum** of each rows *preferredHeight* value. This option cannot be used without explicitly sized rows.	bool	false
useGlobalCellPadding	If set to true, then all cells will use this TableLayout's *cellPadding*value.	bool	true
cellPadding	Padding for each cell.	float(left) float(right) float(top) float(bottom)	0 0 0 0
cellBackgroundImage	Image to use as the background for each cell.	Path/To/Sprite	Sprites/Outline_Sharp
cellBackgroundColor	Color for each cells background image.	color	rgba(1,1,1,0.4)
rowBackgroundImage	Image to use as the background for each row.	Path/To/Sprite	None
rowBackgroundColor	Color to use for each rows background image.	color	clear

ROW

A row within a TableLayout.

ATTRIBUTES

Name	Description	Type / Options	Default Value
preferredHeight	Sets the height for this row. Use a value of '0' to specify that this row should be auto-sized.	float	0
dontUseTableRowBackground	If set to true, then this row will ignore the tables'*rowBackgroundImage* and*rowBackgroundColor values, allowing you to override those values for this row.*	bool	false

CELL

A cell within a TableLayout.

ATTRIBUTES

Name	Description	Type / Options	Default Value
columnSpan	How many columns should this cell span?	int	1
dontUseTableCellBackground	If set to true, then this cell will ignore the tables'*cellBackgroundImage* and <cellbackgroundcolor < i="" style="box-sizing: border-box;">values, allowing you to override those values for this cell.</cellbackgroundcolor<>	bool	false
overrideGlobalCellPadding	If set to true, then this cell will ignore the tables' *cellPadding* value, allowing you to set unique cell padding for this cell.	bool	false

padding	Padding values to use for this cell if*overrideGlobalCellPadding* is set to true.	float(left) float(right) float(top) float(bottom)	0 0 0 0

Input Fields

InputField

A UnityUI text `InputField` which can be used to create both single and multi-line input fields.

Example

```
<InputField>Default Text</InputField>
```

Attributes

Name	Description	Type / Options	Default Value
interactable		bool	true
colors		colorblock (normalColor | highlightedColor | pressedColor | disabledColor)	#FFFFFF|#FFFFFF|#C8C8C8|rgba(0.78,0.78,0.78,0.5)
contentType		▪ Standard ▪ AutoCorrected ▪ IntegerNumber ▪ DecimalNumber ▪ AlphaNumeric ▪ Name ▪ EmailAddress ▪ Password ▪ Pin ▪ Custom	Standard
lineType		▪ SingleLine ▪ MultiLineSubmit ▪ MultiLineNewLine	SingleLine
inputType		▪ Standard ▪ AutoCorrect ▪ Password	Standard

keyboardType		▪ Default ▪ ASCIICapable ▪ NumbersAndPunctuation ▪ URL ▪ NumberPad ▪ NamePhonePad ▪ EmailAddress ▪ NintendoNetworkAccount	Default
characterValidation		▪ None ▪ Integer ▪ Decimal ▪ Alphanumeric ▪ Name ▪ EmailAddress	None
caretBlinkRate			0.85
caretWidth			1
caretColor		color	#323232
selectionColor		color	rgba(0.65,0.8,1,0.75)
hideMobileInput		bool	false
readOnly		bool	false
onValueChanged	Defines a method to be called when this element's value changes.	MethodName(argument);	

onEndEdit	Defines a method to be called when editing has ended.	MethodName(argument);	
textColor		color	#323232
characterLimit		int	0 (no limit)

Button

A UnityUI `Button` object with optional text and/or icon.

Example

```
<!-- Standard Button -->
<Button>Button Text</Button>

<!-- Button with Icon -->
<Button icon="Sprites/Icons/Arrow_Up" />

<!-- Button with Icon and Text -->
<Button icon="Sprites/Icons/Arrow_Left">Button With Icon</Button>
```

Attributes

Name	**Description**	**Type / Options**	**Default Value**
interactable		bool	true
textColor		color	#000000

colors		colorblock (normalColor | highlightedColor | pressedColor | disabledColor)	#FFFFFF|#FFFFFF|#C8C8C8|rgba(0.78,0.78,0.78,0.5)
textShadow		color	None
textOutline		color	None
textAlignment		▪ UpperLeft ▪ UpperCenter ▪ UpperRight ▪ MiddleLeft ▪ MiddleCenter ▪ MiddleRight ▪ LowerLeft ▪ LowerCenter ▪ LowerRight	UpperLeft
icon		Path/To/Sprite	None
iconWidth		float	
iconColor		color	
iconAlignment		▪ Left ▪ Right	Left
padding		float(left) float(right) float(top) float(bottom)	0 0 0 0

TOGGLE

A UnityUI `Toggle` object with optional text.

EXAMPLE

```
<Toggle>Toggle Text</Toggle>

<!-- Toggle which is selected by default -->
<Toggle isOn="true">Toggle Text</Toggle>
```

ATTRIBUTES

Name	Description	Type / Options	Default Value
interactable		bool	true
textColor		color	#000000
colors		colorblock (normalColor | highlightedColor | pressedColor | disabledColor)	#FFFFFF|#FFFFFF|#C8C8C8|rgba(0.78,0.78,0.78,0.5)
isOn	Is this toggle selected?	bool	false
onValueChanged	Defines a method to be called when this element's value changes.	MethodName(argument);	

ToggleButton

A UnityUI `Toggle` object styled as a button with optional text and/or icon.

Example

```
<ToggleButton>Toggle Button Text</Toggle>
```

Attributes

Name	Description	Type / Options	Default Value
interactable		bool	true
textColor		color	#000000
colors		colorblock (normalColor | highlightedColor | pressedColor | disabledColor)	#FFFFFF|#FFFFFF|#C8C8C8|rgba(0.78,0.78,0.78,0.5)
isOn	Is this toggle button selected?	bool	false
onValueChanged	Defines a method to be called when this element's value changes.	MethodName(argument);	
textShadow		color	None
textOutline		color	None

textAlignment		▪ UpperLeft ▪ UpperCenter ▪ UpperRight ▪ MiddleLeft ▪ MiddleCenter ▪ MiddleRight ▪ LowerLeft ▪ LowerCenter ▪ LowerRight	UpperLeft
icon		Path/To/Sprite	None
iconWidth		float	
iconColor		color	
iconAlignment		▪ Left ▪ Right	Left
padding		float(left) float(right) float(top) float(bottom)	0 0 0 0

ToggleGroup

A UnityUI `ToggleGroup` object which can consist of a group of Toggle elements (whether the are Toggles or ToggleButtons).

A LayoutGroup (VerticalLayout/HorizontalLayout/TableLayout/GridLayout) can be used to position the Toggle elements.

ToggleExample

```
<ToggleGroup>
    <VerticalLayout>
        <Toggle>Toggle A</Toggle>
        <Toggle>Toggle B</Toggle>
        <Toggle>Toggle C</Toggle>
    </VerticalLayout>
</ToggleGroup>
```

ToggleButtonExample

```
<ToggleGroup>
    <HorizontalLayout>
        <ToggleButton>ToggleButton A</ToggleButton>
        <ToggleButton>ToggleButton B</ToggleButton>
        <ToggleButton>ToggleButton C</ToggleButton>
    </HorizontalLayout>
</ToggleGroup>
```

Attributes

Name	**Description**	**Type / Options**	**Default Value**
allowSwitchOff	If this is set to true, then the user may clear their selection from within the ToggleGroup by clicking on the selected Toggle.	bool	False

toggleBackgroundImage	Sets the default background image to use for nested Toggle elements.	Path/To/Sprite	Sprites/Elements/ToggleGroup_Background
toggleBackgroundColor		color	#FFFFFF
toggleSelectedImage	Sets the default image to use for selected (checked) nested Toggle elements.	Path/To/Sprite	Sprites/Elements/ToggleGroup_Selected
toggleSelectedColor		color	#FFFFFF

SLIDER

A UnityUI `Slider` object.

EXAMPLE

```
<Slider minValue="0" maxValue="1" value="0.5" />
```

ATTRIBUTES

Name	**Descriptio n**	**Type / Options**	**Default Value**
interactable		bool	true
colors		colorblock (normalColor | highlightedColor | pressedColor | disabledColor)	#FFFFFF|#FFFFFF|#C8C8C8|rgba(0.78,0.78,0.78,0.5)
minValue		float	0

maxValue		float	1
value		float	0
wholeNumbers		bool	false
onValueChanged	Defines a method to be called when this element's value changes.	MethodName(argument);	
direction		▪ LeftToRight ▪ RightToLeft ▪ TopToBottom ▪ BottomToTop	LeftToRight

Dropdown

A UnityUI `Dropdown` object.

Example

```
<Dropdown>
    <Option selected="true">Option 1</Option>
    <Option>Option 2</Option>
    <Option>Option 3</Option>
    <Option>Option 4</Option>
</Dropdown>
```

Attributes

Name	Description	Type / Options	Default Value
interactable		bool	true

onValueChanged	Defines a method to be called when this element's value changes.	MethodName(argument);	
textColor		color	#000000
itemBackgroundColors		colorblock (normalColor | highlightedColor | pressedColor | disabledColor)	#FFFFFF|#FFFFFF|#C8C8C8|rgba(0.78,0.78,0.78,0.5)
itemTextColor		color	#000000

Scroll Views

HorizontalScrollView

A UnityUI `ScrollRect` object set up to scroll horizontally.

A layout element such as a Panel, HorizontalLayout, GridLayout, or TableLayout can be used to position child elements within the Scroll View.

Example

```
<HorizontalScrollView>
    <HorizontalLayout>
        <Panel>
            <Text>1</Text>
        </Panel>
        <Panel>
            <Text>2</Text>
        </Panel>
        <Panel>
            <Text>3</Text>
        </Panel>
        <Panel>
            <Text>4</Text>
        </Panel>
    </HorizontalLayout>
</HorizontalScrollView>
```

Attributes

Name	Description	Type / Options	Default Value
horizontal		bool	true
vertical		bool	false
movementType		▪ Unrestricted ▪ Elastic ▪ Clamped	Clamped
elasticity		float	0.1

inertia		bool	true
decelerationRate		float	0.135
scrollSensitivity		float	1
horizontalScrollbarVisibility		▪ Permanent ▪ AutoHide ▪ AutoHideAndExpandViewport	AutoHide
verticalScrollbarVisibility		▪ Permanent ▪ AutoHide ▪ AutoHideAndExpandViewport	None
onValueChanged	Defines a method to be called when this element's value changes.	MethodName(argument);	
noScrollbars	If set to true, then this scroll view will have no visible scrollbars.	bool	false
scrollbarBackgroundColor		color	#FFFFFF
scrollbarColors		colorblock (normalColor | highlightedColor | pressedColor | disabledColor)	#FFFFFF|#FFFFFF|#C8C8C8|rgba(0.78,0.78,0.78,0.5)
scrollbarImage		Path/To/Sprite	UISprite

VerticalScrollView

A UnityUI `ScrollRect` object set up to scroll vertically.

A layout element such as a Panel, VerticalLayout, GridLayout, or TableLayout can be used to position child elements within the Scroll View.

Example

```
<VerticalScrollView>
    <VerticalLayout>
        <Panel>
            <Text>1</Text>
        </Panel>
        <Panel>
            <Text>2</Text>
        </Panel>
        <Panel>
            <Text>3</Text>
        </Panel>
        <Panel>
            <Text>4</Text>
        </Panel>
    </VerticalLayout>
</VerticalScrollView>
```

Attributes

Name	Description	Type / Options	Default Value
horizontal		bool	false
vertical		bool	true
movementType		▪ Unrestricted ▪ Elastic ▪ Clamped	Clamped
elasticity		float	0.1
inertia		bool	true

decelerationRate		float	0.135
scrollSensitivity		float	1
horizontalScrollbarVisibility		▪ Permanent ▪ AutoHide ▪ AutoHideAndExpandViewport	None
verticalScrollbarVisibility		▪ Permanent ▪ AutoHide ▪ AutoHideAndExpandViewport	AutoHide
onValueChanged	Defines a method to be called when this element's value changes.	MethodName(argument);	
noScrollbars	If set to true, then this scroll view will have no visible scrollbars.	bool	false
scrollbarBackgroundColor		color	#FFFFFF
scrollbarColors		colorblock (normalColor | highlightedColor | pressedColor | disabledColor)	#FFFFFF|#FFFFFF|#C8C8C8|rgba(0.78,0.78,0.78,0.5)
scrollbarImage		Path/To/Sprite	UISprite

Masks

Mask

A UnityUI `Mask` object which can be used to mask nested elements.

Example

```
<Mask image="Sprites/MaskImage">
    <Panel>
        <Text>Masked Content</Text>
    </Panel>
</Mask>
```

Attributes

Name	Description	Type / Options	Default Value
image	Mask graphic	Path/To/Sprite	UIMask
showMaskGraphic	Should the mask graphic be visible?	bool	false

CUSTOMISING XMLLAYOUT

CUSTOM ATTRIBUTES

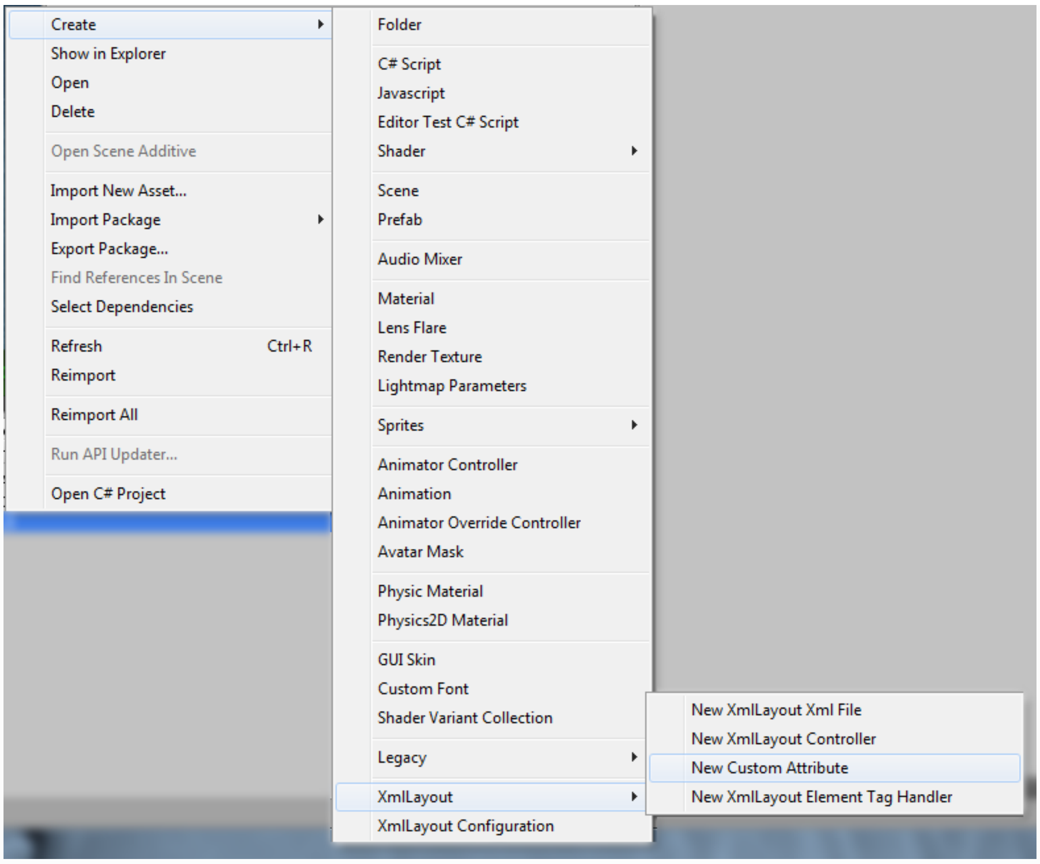
If you wish to add custom attributes of your own which apply to one or more element types, then you can create a `CustomXmlAttribute` which will define how to handle it.

Please note that if you wish to create an attribute for a single tag, and that tag is a custom tag (see below), then it is not strictly necessary to create a CustomXmlAttribute as the attribute can be handled by the TagHandler if you wish.

Creating a new CustomXmlAttribute

In the Project window, right-click the folder in which you wish to store the CustomXmlAttribute code.

Then, select **Create->XmlLayout->New Custom Attribute**



Add New Custom Xml Attribute

You will then be prompted to enter a name for the new attribute's C# file. You will also need to change the name of the class in the file itself.

Naming Custom Attributes

Custom Xml Attributes use a strict naming standard, which must be followed if the attribute is to be matched in Xml. The format is as follows:

`MyAttributeNameAttribute`

This would match the Xml attribute 'myAttributeName', e.g.

```
<Element myAttributeName="myAttributeValue" />
```

Once the CustomXmlAttribute has been added to the project and named correctly, it will automatically be located and used by XmlLayout.

CustomXmlAttribute Virtual methods/properties

The CustomXmlAttribute class defines two virtual methods and three virtual properties which you can override to define how this attribute behaves:

Methods

- **Apply()** - This method allows you to alter the element directly as required by the attribute's value.
- **Convert()** - This method allows you to define and return new attribute values which will be processed (effectively allowing you to translate one attribute into one or more others).

Your Custom Xml Attribute may implement one or both of the above methods.

Properties

- **RestrictToPermittedElementsOnly** - If this property returns true, then this attribute will only be applied to elements in PermittedElements. If this property returns false, then it will be applied to any element. **Default: False**
- **PermittedElements** - Defines which elements this attribute may be applied to.
- **KeepOriginalTag** - If this property returns true, then this attribute will remain in the list of attributes after it has been processed by this CustomXmlAttribute. If it returns false, then the attribute will be removed after processing. **Default: False**

For examples of CustomXmlAttributes, see the `Assets/UI/XmlLayout/CustomAttributes` folder.

Custom Tags

If you wish to create custom tags of your own, then you can create a new `ElementTagHandler` which will define how to handle it.

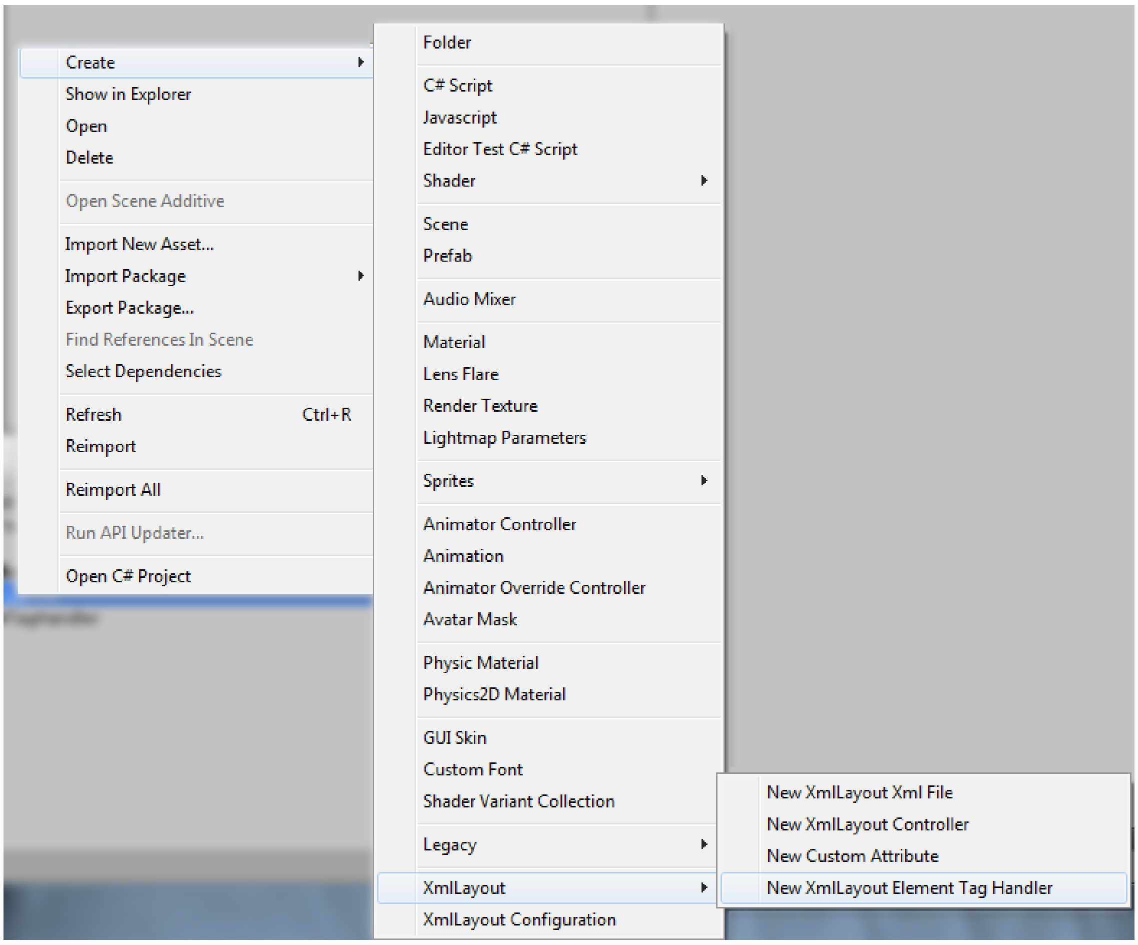
Creating a new Element Tag Handler

The first step in creating a new Element Tag Handler is to create a prefab which will serve as the 'base' for the element. For examples of tag prefabs, look in the `Assets/UI/XmlLayout/Tags/Resources/XmlLayout Prefabs/` folder.

Once you have created a prefab, you will then need to create the Element Tag Handler class.

In the Project window, right-click the folder in which you wish to store the Element Tag Handler code.

Then, select **Create->XmlLayout->New Element Tag Handler**



Add New Element Tag Handler

You will then be prompted to enter a name for the new attribute's C# file. You will also need to change the name of the class in the file itself.

Naming Element Tag Handlers

Element Tag Handlers use a strict naming standard, which must be followed if the handler is to be matched to an Xml tag. The format is as follows:

`MyTagNameTagHandler`

This would match the Xml tag 'MyTagName', e.g.

```
<MyTagName />
```

Once the Element Tag Handler has been added to the project and named correctly, it will automatically be located and used by XmlLayout.

ElementTagHandler Virtual methods/properties

The ElementTagHandler class defines three virtual methods and four virtual properties which you can override to define how this tag behaves:

Methods

- **ApplyAttributes()**
  This method should be used to process any complex attributes as required by this tag.

  In generally, you should call `base.ApplyAttributes()` for standard attribute handling - XmlLayout will use reflection to match attributes and their values to the element, which means that most attributes will automatically be used - only complex attributes, or attributes which do not directly match a property on the class need to have special handling set up here.

  The base ApplyAttributes() method will attempt to match attributes to components in the following order (and will return once it finds a match):

  - The Primary Component (as defined by `primaryComponent`)
  - The `RectTransform`
  - The `LayoutElement` component (which is automatically added to each element by XmlLayout)
  - The `XmlElement` component (which is automatically added to each element by XmlLayout)
  - The `Image` component (if there is one)

  If this tag does not require any special attribute handling, this function can safely be omitted.

- **HandleEventAttribute()**
  This method should be used to process event attributes. It will be called for each element defined by `eventAttributeNames` (as long as the base ApplyAttributes() method is called at some point).

  `base.HandleEventAttribute()` will handle the `onClick`, `onMouseEnter`, and `onMouseExit` events attributes.

  For examples of how to implement this method for custom events, see the `UI.Xml.InputFieldTagHandler`, `UI.Xml.SliderTagHandler`, and `UI.Xml.DropdownTagHandler` classes (located in Assets/UI/XmlLayout/Tags).

  If this tag does not require any special event handling, this function can safely be omitted.

- **ParseChildElements()**
  This method allows you to control how child elements of this custom tag are parsed, e.g. for tags like Dropdown.

  If this method returns true, then XmlLayout will not attempt to parse the child elements itself (as this method has already parsed them).

  Default return value: **False**

  If this tag does not require any child element handling, this function can safely be omitted.

Your Element Tag Handler may implement any number of the above methods.

Properties

- **primaryComponent**
  This property should return the component (MonoBehaviour) which would be considered to be the 'primary' component of this tag. e.g. <Button /> == UnityEngine.UI.Button

  This is usually best achieved through the use of `currentInstanceTransform.GetComponent();`

  Default return value: **null**

- **prefabPath**
  By default, XmlLayout will use the following path: `Resources/XmlLayout Prefabs/{Element Name}TagHandler`

  If the prefab used by this tag is located in a different location (or is named differently), then override this function and return the correct location of the prefab.

- **transformToAddChildrenTo**
  Generally, if this element supports child elements, child elements will be added directly to this element's transform. In some cases, however, this is not the desired behaviour, e.g. child elements of ScrollViews need to be added to the ScrollViews content object, not the ScrollView itself.

  Default return value: **currentInstanceTransform**

- **eventAttributeNames**
  This property defines which attributes will be processed by HandleEventAttribute(). The names of any custom events should be added here.

  Default return value: **onClick, onMouseEnter, onMouseExit**

Additionally, the ElementTagHandler class defines several non-virtual properties which may be useful when implementing custom tag behaviour:

- **currentInstanceTransform**
  This is the `RectTransform` of the element that is being processed.

- **currentXmlElement**
  This is the `XmlElement` component of the element that is being processed.

- **currentXmlLayoutInstance**
  This is the `XmlLayout` which contains the element that is being processed.

For examples of CustomXmlAttributes, see the `Assets/UI/XmlLayout/CustomAttributes` folder.

Using custom prefabs with existing tags

If you wish, you may substitute the default prefab used by an existing tag by setting the `prefabPath` attribute to the path of your choice.

It is highly recommended that you ensure that any custom prefabs contain the required components (e.g. prefabs for the Button element should always contain a Button component) in order to prevent errors from ocurring.