Flex in a Week, Flex 4.5

Video 4.01: Representing data in default item renderers

On the first Day of training, you learned about MX and Spark components, including the Spark containers like Application, Panel and Group.

These containers are used to layout and display other containers and visual controls.

In this video, you will learn about the DataGroup container, which also displays content, but based on a defined set of data.

You will also learn how to use two default Spark item renderer for the DataGroup container, that define a uniform, repeatable display for each record in the dataset.

You will first learn how to display string data in an item renderer and then you will learn how to pass visual UI components in the dataset.

Lastly, you will dynamically determine the appropriate item renderer to display based on a conditional check of the data.

This is a conceptual diagram of an item renderer for a DataGroup container.

Here is the DataGroup container on the right and here is the data set on the left.

The job of the DataGroup container is to iterate over all of the data items in the data set – or in Flex terms, the data provider – and generate a visual display in the DataGroup container for each item.

The data for each item is laid out in a specific configuration based on the directions in the layout template, or, again in Flex terms, the item renderer.

In this video, you will use two default item renderers that are available in the Spark package.

In the next video, you will learn how to create your own custom item renderer.

Let's begin with an overview of the DataGroup container basics.

First, the DataGroup container is a Spark component.

It accepts the data set for display through a dataProvider property.

In this property, you can define simple strings and numbers or complex data such as Object and XML instances.

The DataGroup class converts these data elements into visual elements for display using an item renderer.

Keep in mind that the DataGroup container is a non-skinnable container.

I will address this last point in Day 5 of the training.

This is the starter MXML application file for this video and related exercise.

I will first show you how to work with simple text data.

The data itself is a non-visual element, so I am creating it in the Declarations tag block.

Within this tag block, I am creating an ArrayList instance with an id value of employeeList.

Remember that an ArrayList is a bindable complex object similar to an ArrayCollection, but lighter weight because it does not include the sorting capabilities of ArrayCollection.

I am pasting four String instances with employee names.

This is my data set.

Below the Label control, I am creating a DataGroup container block and adding a dataProvider property.

To this property, I am binding the employeeList instance of data that I just created.

When I save the file and run the application, you can see a runtime error that says that the Flash Player "Could not create an item renderer for Samuel Ang."

The DataGroup container requires that you also define an itemRenderer property that references an item renderer class instance.

An item renderer class defines how the provided data will be visualized and is based on the type of data that will be displayed.

There are two default item renderer classes that are in the spark.skins.spark package.

The DefaultItemRenderer class is used to display simple data like strings or numbers.

The DefaultComplexItemRenderer is used to display UI components.

As I mentioned earlier, you will learn how to create a custom item renderer in the next video.

Also, remember that the DataGroup component is a container with a layout property.

By default, all of its content is laid out using the BasicLayout class, but you can change this, as you will see shortly.

Back in Flash Builder, I am adding an itemRenderer property to the DataGroup container instance and setting its value to the spark.skins.spark.DefaultItemRenderer class.

When I save the file and run the application you only see a black blob.

Remember that I said the default layout property value is the BasicLayout class.

This means that all of the data elements have x and y values of zero and are stacked on top of each other in the display.

Inside the DataGroup container tag set, I am creating a layout property tag set and assigning a VerticalLayout instance for its value.

I am adding the paddingLeft and paddingTop properties and setting both their values to 25.

When I save the file and run the application, you can see that the employee names are displayed vertically below the header text.

Earlier I mentioned that you would use the DefaultComplexItemRenderer class to display visual components.

This literally means that you assign the UI elements as the data in your dataProvider property.

In this code example, the ArrayList instance has BitmapImage Spark controls rather than string values for the data.
In the ArrayList instance, I have added four Spark BitmapImage controls below the string values.

The source property for each instance points to a JPG in this images directory.

In the DataGroup container instance, I now need to update the itemRenderer to use the DefaultComplexItemRenderer class.

When I save the file and run the application, you see only the employee images and not their names displayed despite the fact that the name strings are in the data provider.

This happens because the DefaultComplexItemRenderer only displays visual elements and does not display simple text values.

To mix the two types of data, I will implement the itemRendererFunction property, which will evaluate the type of data to be displayed and then use the proper item renderer class for the job.

You will pass each data item to the function as an Object instance and the function will use a class named ClassFactory to create a new instance of the appropriate renderer and return it to the DataGroup container.

The function itself determines the type of data in the item by doing a simple data type check.

If the data is simple, you will create a DefaultItemRenderer instance.

If the data is a visual element, then you will instantiate the DefaultComplexItemRenderer class.

Note that you can also use ClassFactory to return a custom item renderer as well.

You will learn how to do this in this video, but you will not learn how to create a custom item renderer until the next video.

In the DataGroup component instance, I am removing the itemRenderer property and adding the itemRendererFunction property with a value of rendererFunction.

My next step is to create this function to test each data element defined in the ArrayList instance to determine the correct item renderer to use.

Below the Script comment, I am creating a Script block.
Next I am creating a private function named rendererFunction, as I referenced here in the DataGroup tag.

The function takes one argument, which I am naming item and data typing to a generic Object.

Remember that each data element in the ArrayList instance will be passed to this function.

That means that the item argument will represent one of these String or BitmapImage instances.

The return type for the function is the ClassFactory class.

Within the rendererFunction() I am typing if and invoking the content assist tool – by pressing CTRL+Space twice - and selecting the if statement code template.

In the if statement I am evaluating if the item parameter value is a String instance.

If true, I am using the return keyword and then the new keyword to call the ClassFactory class constructor.

Remember that the ClassFactory class creates an instance of a class.

In this case, since the data type is a string, I am using content assist to create an instance of the DefaultItemRenderer class.

You can see that the corresponding import statement has been generated in the Script block.

Now I am creating an else statement and returning a new instance, again using the ClassFactory class, of the DefaultComplexItemRenderer class.

Again, the corresponding import statement has been generated.

When I save the file and run the application you can see that the string and image data are displayed but the string values are not displayed above the correct images.

In the ArrayList instance, I am rearranging the data objects so that the BitmapImage controls follow their respective String values.

When I save the file and run the application, you now see that the image and string values are properly displayed.

Lastly, I will use the ClassFactory class to instantiate a custom item renderer instead of these two built-in item renderers.

You will learn how to create a custom item renderer in the next video.

Right now, I will just use one that has already been created.

In the Package Explorer, I am opening the NameDisplay.mxml file from the components package.

This file contains a Label control with a text property that binds to the item renderer's data.

It also defines the backgroundColor and color of the text.

Back in the main application file, I am locating this first return statement so that it returns a new instance of the custom item renderer, NameDisplay.

Remember that if you use the content assist tool (CTRL+Space) to add the NameDisplay class, the corresponding import statement will be generated within the Script block.

I am saving the file and running the application.

You can see that the data from the custom item renderer is displayed.

For your next step, work through the exercise titled “Passing data to item renderers for display”.