

Chapter 1: Spark containers

Spark containers provide a hierarchical structure to arrange and configure their children.

Flex also provides a set of Halo layout and Halo navigator containers. Adobe recommends that you use the Spark containers when possible. For more information on the Halo layout and Halo navigator containers, see [Using Halo Layout Containers](#) and [Using Halo Navigator Containers](#).

For an introduction to containers, including Spark containers, see [Introducing Containers](#).

About Spark containers

Spark includes the following containers:

- Group and DataGroup
- SkinnableContainer, SkinnableDataContainer, Panel, and Application

For more information on the Application container, see [Using Application Containers](#).

All Spark containers support interchangeable layouts. That means you can set the layout of a container to any of the supported layout types, such as basic, horizontal, vertical, or tiled layout. You can also define a custom layout.

To improve performance and minimize application size, some Spark containers do not support skinning. Use the Group and DataGroup containers to manage child layout. Use SkinnableContainer, SkinnableDataContainer, and Panel to manage child layout and to support custom container skins.

The Group and SkinnableContainer classes can take any visual components as children. Visual components implement the `IVisualElement` interface, and include subclasses of the `UIComponent` class and the `GraphicElement` class.

The `UIComponent` class is the base class of all Flex components. Therefore, you can use any Flex component as a child of the Group and SkinnableContainer class.

The `GraphicElement` class is the base class for the Flex drawing classes, such as the `Ellipse`, `Line`, and `Rect` classes. Therefore, you can use subclass of the `GraphicElement` class as a child of the Group and SkinnableContainer class.

The `DataGroup` and `SkinnableDataContainer` classes take as children visual components that implement the `IVisualElement` interface and are subclasses of `DisplayObject`. This include subclasses of the `UIComponent` class.

However, the `DataGroup` and `SkinnableDataContainer` containers are optimized to hold data items. Data items can be simple data items such as `String` and `Number` objects, and more complicated data items such as `Object` and `XMLNode` objects. Therefore, while these containers can hold visual children, you should use Group and SkinnableContainer for children that are visual components.

The following table lists the main characteristics of the Spark containers:

	Group	DataGroup	SkinnableContainer	SkinnableDataContainer	Panel
Children	IVisualElement	Data Item IVisualElement and DisplayObject	IVisualElement	Data item IVisualElement and DisplayObject	IVisualElement
Layout	Selectable	Selectable	Selectable	Selectable	Selectable
Skinnable	No	No	Yes	Yes	Yes
Creation policy	All	All	Selectable	Selectable	Selectable
Primary use	Lay out visual children.	Render and lay out data items.	Lay out visual children in a skinnable container.	Render and lay out data items in a skinnable container.	Subclass of SkinnableContainer that adds a title bar and other visual elements to the container.

For information on skinning, see *Creating Spark Skins*. For information on creation policy, see *About the creation policy*.

About container children

All Spark containers can take as children visual components that implement the `IVisualElement` interface. All Spark and Halo components implement the `IVisualElement` interface and can therefore be used as container children.

The following example uses the Spark `SkinnableContainer` to hold Spark Button components:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkContainerSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:SkinnableContainer>
    <s:layout>
      <s:HorizontalLayout/>
    </s:layout>
    <s:Button label="Button 1"/>
    <s:Button label="Button 2"/>
    <s:Button label="Button 3"/>
    <s:Button label="Button 4"/>
  </s:SkinnableContainer>
</s:Application>
```

The Spark `Group`, `SkinnableContainer`, and `Panel` classes can take as children any subclass of the `GraphicElement` class. The following example shows a `SkinnableContainer` with a `Line` component between two `Buttons` components:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkContainerGraphic.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:SkinnableContainer>
    <s:Button label="Button 1"/>
    <s:Line
      xFrom="0" xTo="100">
      <s:stroke>
        <s:LinearGradientStroke weight="2"/>
      </s:stroke>
    </s:Line>
    <s:Button label="Button 2"/>
  </s:SkinnableContainer>
</s:Application>
```

The `DataGroup` and `SkinnableDataContainer` classes are designed primarily to display data items as children. The following example shows a `SkinnableDataContainer` displaying an Array of Strings as children:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataContainerSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:SkinnableDataContainer itemRenderer="spark.skins.spark.DefaultItemRenderer">
    <mx:ArrayCollection>
      <fx:String>Dave Jones</fx:String>
      <fx:String>Mary Davis</fx:String>
      <fx:String>Debbie Cooper</fx:String>
    </mx:ArrayCollection>
  </s:SkinnableDataContainer>
</s:Application>
```

About Spark layouts

All Spark containers define a default layout, but let you switch the layout to suit your application requirements. To switch layout, assign a layout class to the `layout` property of the container.

Flex ships with several layout classes that you can use with the Spark containers. Additionally, you can define custom layout classes. The layout classes are defined in the `spark.layouts` package, and include the following classes:

- **BasicLayout** Uses absolute positioning. You explicitly position all container children, or use constraints to position them.
- **HorizontalLayout** Lays out children in a single horizontal row. The height of the row is fixed to the same height for all children and is typically the height of the tallest child. The width of each child is either fixed to the same value for all children, or each child can calculate its own width. By default, each child calculates its own width.
- **TileLayout** Lays out children in one or more vertical columns or horizontal rows, starting new rows or columns as necessary. The `orientation` property determines the layout direction. The valid values for the `orientation` property are `columns` for a column layout and `rows` (default) for a row layout.

All cells of the tile layout have the same size, which is the height of the tallest child and the width of the widest child.

- `VerticalLayout` Lays out children in a single vertical column. The width of the column is fixed to the same width for all children and is typically the width of the widest child. The height of each child is either fixed to the same value for all children, or each child can calculate its own height. By default, each child calculates its own height.

Setting the layout of a Spark container

By default, the `Group` container uses the `BasicLayout` class. The following example uses the `layout` property of the container to set its layout to the `HorizontalLayout` class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerHorizontal.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:Group>
    <s:layout>
      <s:HorizontalLayout/>
    </s:layout>
    <s:Button label="Button 1"/>
    <s:Button label="Button 2"/>
    <s:Button label="Button 3"/>
    <s:Button label="Button 4"/>
  </s:Group>
</s:Application>
```

To simplify the use of layouts with the `Group` container, Flex defines two subclasses of `Group`: `HGroup`, with a horizontal layout, and `VGroup`, with a vertical layout. Therefore, you can rewrite the previous example as shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkHGroupContainer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:HGroup>
    <s:Button label="Button 1"/>
    <s:Button label="Button 2"/>
    <s:Button label="Button 3"/>
    <s:Button label="Button 4"/>
  </s:HGroup>
</s:Application>
```

Setting the padding and gap of a layout

Some Spark layout classes, including `HorizontalLayout` and `VerticalLayout`, support padding and gap properties. The padding properties define the space between the container boundaries and the children. The gap properties define the space between the children, either horizontally or vertically.

The following example sets the padding to 10 pixels, and the gap to 5 pixels, for the children of a `Panel` container:

```

<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkPanelPadding.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:Panel>
    <s:layout>
      <s:HorizontalLayout
        paddingLeft="10" paddingRight="10"
        paddingTop="10" paddingBottom="10"
        gap="5"/>
    </s:layout>
    <s:Button label="Button 1"/>
    <s:Button label="Button 2"/>
    <s:Button label="Button 3"/>
    <s:Button label="Button 4"/>
  </s:Panel>
</s:Application>

```

Setting the alignment of a layout

The layout containers provide the `horizontalAlign` and `verticalAlign` properties for aligning the children of a container. For example, you can use these properties to align children to the top of a container using `HorizontalLayout`, or to the left side of a container using `VerticalLayout`.

The `horizontalAlign` property is available in the `TileLayout` and `VerticalLayout` classes. It can take the following values:

- `left` Align children to the left side of the container. This is the default value for `VerticalLayout`.
- `right` Align children to the right side of the container.
- `center` Align the children to the horizontal center of the container.
- `justify` Set the width of all children to be the same width as the container.
- `contentJustify` Set the width of all children to be the *content width* of the container. The content width of the container is the width of the largest child. If all children are smaller than the width of the container, then set the width of all the children to the width of the container.

The `verticalAlign` property is available in the `TileLayout` and `HorizontalLayout` classes. It can take the following values:

- `top` Align children to the top of the container. This is the default value for `VerticalLayout`.
- `bottom` Align children to the bottom of the container.
- `middle` Align children to the vertical middle of the container.
- `justify` Set the height of all children to be the same height as the container.
- `contentJustify` Set the height of all children to be the *content height* of the container. The content height of the container is the height of the largest child. If all children are smaller than the height of the container, then set the height of all the children to the height of the container.

The following example overrides the default vertical alignment of top for the `HorizontalLayout` to align the children of a `Group` container to the bottom of the container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkContainerSimpleAlignment.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:SkinnableContainer>
    <s:layout>
      <s:HorizontalLayout verticalAlign="bottom"/>
    </s:layout>
    <s:Button label="Button 1" fontSize="24"/>
    <s:Button label="Button 2"/>
    <s:Button label="Button 3" fontSize="36"/>
    <s:Button label="Button 4"/>
  </s:SkinnableContainer>
</s:Application>
```

The following example overrides the default vertical alignment of top for the `HorizontalLayout` to use `justify`. Notice that all buttons have the same height:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkContainerSimpleAlignment.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:SkinnableContainer>
    <s:layout>
      <s:HorizontalLayout verticalAlign="justify"/>
    </s:layout>
    <s:Button label="Button 1" fontSize="24"/>
    <s:Button label="Button 2"/>
    <s:Button label="Button 3" fontSize="36"/>
    <s:Button label="Button 4"/>
  </s:SkinnableContainer>
</s:Application>
```

Setting the row height or column width of a layout

You control the way the container lays out children with different sizes by using the `VerticalLayout.variableRowHeight` and `HorizontalLayout.variableColumnWidth` properties. By default, these properties are set to `true` which lets each child determine its height (`VerticalLayout`) or width (`HorizontalLayout`).

In the following example, the `Group` container holds four buttons. By default, buttons use a 12 point font size. However, two of the buttons in this example define a larger font size. Because the `variableRowHeight` property is set to `true` by default, the container sets the height of each button appropriately for its font size:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerVarRowHeightTrue.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:Group>
    <s:layout>
      <s:VerticalLayout />
    </s:layout>
    <s:Button label="Button 1"/>
    <s:Button label="Button 2"/>
    <s:Button label="Button 3" fontSize="36"/>
    <s:Button label="Button 4" fontSize="24"/>
  </s:Group>
</s:Application>
```

If your container has many children, setting the `variableRowHeight` or `variableColumnWidth` properties to `true` can affect application performance. The reason is because the container calculates the size of every child as it appears on the screen.

Rather than having each child calculate its size, you can set the `variableRowHeight` or `variableColumnWidth` property to `false` so that each child has the same size.

Note: The following paragraphs describe setting the `variableRowHeight` property for the `VerticalLayout` class. This discussion is the same as setting the `variableColumnWidth` property for the `HorizontalLayout` class.

If you set the `variableRowHeight` property to `false`, the `VerticalLayout` class uses the following procedure to determine the height of each child:

- 1 If specified, use the `VerticalLayout.rowHeight` property to specify an explicit height of all children. Make sure that the specified height is suitable for all children.
- 2 If specified, use the `VerticalLayout.typicalLayoutElement` property to define the height of all children. This property references a component that Flex uses to define the height of all container children.
- 3 Use the preferred width of the first container child as the height of all container children. This technique is useful if the children of the container are all similar.

In the following example, a group container uses the `VerticalLayout` class to lay out four `Button` controls. The `variableRowHeight` property is `false` so that every button has the same height:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerVarRowHeightFalse.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:Group>
    <s:layout>
      <s:VerticalLayout variableRowHeight="false"/>
    </s:layout>
    <s:Button label="Button 1"/>
    <s:Button label="Button 2"/>
    <s:Button label="Button 3" fontSize="36"/>
    <s:Button label="Button 4" fontSize="24"/>
  </s:Group>
</s:Application>
```

Because you did not specify an explicit value for the `VerticalLayout.rowHeight` property or the `VerticalLayout.typicalLayoutElement` property, the `VerticalLayout` class uses the preferred height of the first button control as the height for all container children. However, because the third button and fourth button controls define a large font size, the text is truncated to the size of the button.

Alternatively, you can set the `rowHeight` property to a pixel value large enough for all the children, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerVarRowHeightFalseRowHeight.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:Group>
    <s:layout>
      <s:VerticalLayout variableRowHeight="false"
        rowHeight="40"/>
    </s:layout>
    <s:Button label="Button 1"/>
    <s:Button label="Button 2"/>
    <s:Button label="Button 3" fontSize="36"/>
    <s:Button label="Button 4" fontSize="24"/>
  </s:Group>
</s:Application>
```

In this example, all buttons are 40 pixels tall.

The following example uses the `typicalLayoutElement` property to specify to use the third button to determine the height of all container children:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerVarRowHeightFalseTypicalLE.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:Group>
    <s:layout>
      <s:VerticalLayout variableRowHeight="false"
        typicalLayoutElement="{b3}"/>
    </s:layout>
    <s:Button id="b1" label="Button 1"/>
    <s:Button id="b2" label="Button 2"/>
    <s:Button id="b3" label="Button 3" fontSize="36"/>
    <s:Button id="b4" label="Button 4" fontSize="24"/>
  </s:Group>
</s:Application>
```

You can use two common strategies for determining the typical item. One option is to use the largest item as the typical item. A second options is to calculate the average size of all items, and use the data item closest to that average size.

The Spark Group and Spark SkinnableContainer containers

The Spark Group and Spark SkinnableContainer containers take as children any components that implement the `IVisualElement` interface. Use these containers when you want to manage visual children, both visual components and graphical components.

The main difference between the Group and SkinnableContainer containers is that the SkinnableContainer can be skinned. The group container is designed for simplicity and minimal overhead, and cannot be skinned.

One of the uses of the Group container is to import graphic elements from Adobe design tools, such as Adobe Illustrator CS4. For example, if you use a design tool to create graphics imported into Flex, the graphics are often represented using FXG statements in a Group container. For more information, see [Using FXG](#).

The default layout class of the Group and SkinnableContainer container is `BasicLayout`. The following example shows a Group container with a horizontal layout and one with a vertical layout:



For complete reference information, see the *Adobe Flex Language Reference*.

Creating a Spark Group container

You use the `<s:Group>` tag to define a Group container. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example shows a Group container with four Button controls as its children:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:Group>
    <s:Button label="Button 1"
      left="10" top="13" bottom="10"/>
    <s:Button label="Button 2"
      left="110" top="13" bottom="10"/>
    <s:Button label="Button 3"
      left="210" top="13" bottom="10"/>
    <s:Button label="Button 4"
      left="310" top="13" bottom="10" right="10"/>
  </s:Group>
</s:Application>
```

In this example, the Group container uses its default layout specified by the `BasicLayout` class, which means the container uses absolute layout. The four button controls then use constraints to set their positions in the container. For more information on constraints, see [Using constraints to control component layout](#).

You can add a graphic element to the container to define a background for the buttons, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkGroupContainerRect.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:Group>
    <s:Rect x="0" y="0"
      radiusX="4" radiusY="4"
      height="100%" width="100%">
      <s:stroke>
        <s:LinearGradientStroke weight="1" scaleMode="normal"/>
      </s:stroke>
      <s:fill>
        <s:LinearGradient>
          <s:entries>
            <mx:GradientEntry color="0x999999"/>
          </s:entries>
        </s:LinearGradient>
      </s:fill>
    </s:Rect>
    <s:Button label="Button 1"
      left="10" top="13" bottom="10"/>
    <s:Button label="Button 2"
      left="110" top="13" bottom="10"/>
    <s:Button label="Button 3"
      left="210" top="13" bottom="10"/>
    <s:Button label="Button 4"
      left="310" top="13" right="10" bottom="10"/>
  </s:Group>
</s:Application>
```

In this example, you add an instance of the `Rect` class, a subclass of `GraphicElement`, that defines a gray background and one pixel border around the container. In this example the `Rect` is located a coordinates 0,0 in the `Group` container, and sized to fill the entire container.

Creating a Spark `SkinnableContainer` container

You use the `<s:SkinnableContainer>` tag to define a `SkinnableContainer` container. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an `ActionScript` block.

The following example shows a `SkinnableContainer` container with four `Button` controls as its children:

```

<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkContainerSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:SkinnableContainer>
    <s:layout>
      <s:HorizontalLayout/>
    </s:layout>
    <s:Button label="Button 1"/>
    <s:Button label="Button 2"/>
    <s:Button label="Button 3"/>
    <s:Button label="Button 4"/>
  </s:SkinnableContainer>
</s:Application>

```

The default layout class of the SkinnableContainer class is BasicLayout. In this example, the SkinnableContainer uses the HorizontalLayout class to arrange the buttons in a single row.

If the SkinnableContainer uses BasicLayout, you can use a Rect component as a child of the container to add a background color and border. For an example, see “Creating a Spark Group container” on page 9.

However, the SkinnableContainer class lets you apply a skin to define the visual characteristics of the container. For example, the following skin defines a gray background and a one pixel border for the container:

```

<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\mySkins\MyBorderSkin.mxml -->
<s:Skin xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  alpha.disabled="0.5">

  <fx:Metadata>
    [HostComponent("spark.components.SkinnableContainer")]
  </fx:Metadata>

  <!-- Define the skin states. -->
  <s:states>
    <s:State name="normal" />
    <s:State name="disabled" />
  </s:states>

  <!-- Define a Rect to fill the area of the skin. -->
  <s:Rect x="0" y="0"
    radiusX="4" radiusY="4"
    height="100%" width="100%">

```

```

        <s:stroke>
            <s:LinearGradientStroke weight="1"/>
        </s:stroke>
        <s:fill>
            <s:LinearGradient>
                <s:entries>
                    <mx:GradientEntry color="0x999999"/>
                </s:entries>
            </s:LinearGradient>
        </s:fill>
    </s:Rect>

    <!-- Define the content area of the container. -->
    <s:Group id="contentGroup"
        left="5" right="5" top="5" bottom="5">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
    </s:Group>
</s:Skin>

```

All `SkinnableContainer` skins must implement the view states defined by the `SkinnableContainer`. Because the `SkinnableContainer` class supports the normal and disabled view states, the skin must also support them.

The `Rect` component adds the border and gray background to the skin.

The only required part of the `SkinnableContainer` skin is `contentGroup`. All `SkinnableContainer` skins must define a skin part named `contentGroup`.

All the containers children are added to the `contentGroup` container of the skin. In this example, the `contentGroup` container is a `Group` container with a vertical layout. Setting the layout property of the `SkinnableContainer` overrides the layout specified in the skin.

The advantage to defining a skin for the `SkinnableContainer`, rather than adding the visual elements in the `SkinnableContainer` definition, is that the skin is reusable. For example, you typically define a consistent look for all `SkinnableContainer` containers in an application. By encapsulating that look in a reusable skin class, you can apply it to all containers in your application.

Use the `skinClass` property to apply the skin to the `SkinnableContainer`, as the following example shows:

```

<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkContainerSkin.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:SkinnableContainer
        skinClass="mySkins.MyBorderSkin">
        <s:layout>
            <s:HorizontalLayout gap="10"/>
        </s:layout>
        <s:Button label="Button 1"/>
        <s:Button label="Button 2"/>
        <s:Button label="Button 3"/>
        <s:Button label="Button 4"/>
    </s:SkinnableContainer>
</s:Application>

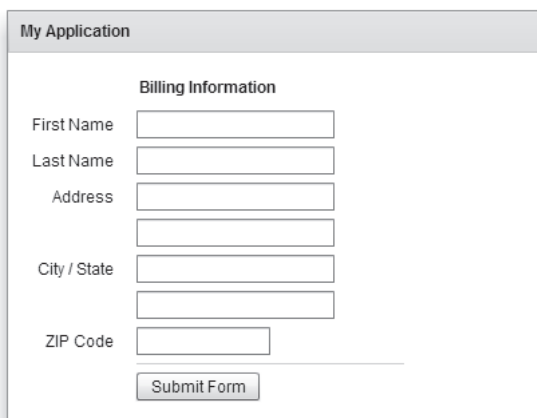
```

For more information on skinning, see [Creating Spark Skins](#).

The Spark Panel layout container

A Panel container includes a title bar, a title, a border, and a content area for its children. Typically, you use Panel containers to wrap self-contained application modules. For example, you could define several Panel containers in your application where one Panel container holds a form, a second holds a shopping cart, and a third holds a catalog.

The default layout class of the Panel container is `BasicLayout`. The following example shows a Panel container with a vertical layout:

The image shows a window titled "My Application" with a title bar. Inside the window, there is a section titled "Billing Information". Below this title, there are several text input fields arranged vertically: "First Name", "Last Name", "Address" (with a second empty line below it), "City / State" (with a second empty line below it), and "ZIP Code". At the bottom of the form is a "Submit Form" button.

For complete reference information, see the *Adobe Flex Language Reference*.

Creating a Spark Panel container

You use the `<s:Panel>` tag to define a Panel container. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example defines a Panel container that contains a form as the top-level container in your application. In this example, the Panel container provides you with a mechanism for including a title bar, as in a standard GUI window.

```

<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkPanelSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark" >
  <s:Panel id="myPanel" title="My Application" x="20" y="20">
    <mx:Form id="myForm" width="400">

      <mx:FormHeading label="Billing Information"/>

      <mx:FormItem label="First Name">
        <s:TextInput id="fname" width="100%"/>
      </mx:FormItem>

      <mx:FormItem label="Last Name">
        <s:TextInput id="lname" width="100%"/>
      </mx:FormItem>

      <mx:FormItem label="Address">
        <s:TextInput id="addr1" width="100%"/>
        <s:TextInput id="addr2" width="100%"/>
      </mx:FormItem>

      <mx:FormItem label="City / State" direction="vertical">
        <s:TextInput id="city"/>
        <s:TextInput id="state"/>
      </mx:FormItem>

      <mx:FormItem label="ZIP Code">
        <s:TextInput id="zip" width="100%"/>
      </mx:FormItem>

      <mx:FormItem>
        <mx:HRule width="200" height="1"/>
        <s:Button label="Submit Form"/>
      </mx:FormItem>
    </mx:Form>
  </s:Panel>
</s:Application>

```

The Spark DataGroup and Spark SkinnableDataContainer containers

The Spark DataGroup and Spark SkinnableDataContainer containers take as children any components that implement the IVisualElement interface and are subclasses of DisplayObject. However, these containers are primarily used to take data items as children. Data items can be simple data items such as String and Number objects, and more complicated data items such as Object and XMLNode objects.

An item renderer defines the visual representation of the data item in the container. The item renderer converts the data item into a format that can be displayed by the container. You must pass an item renderer to a DataGroup or SkinnableDataContainer container.

The main difference between the `DataGroup` and `SkinnableDataContainer` container is that the `SkinnableDataContainer` can be skinned. The `DataGroup` container is designed for simplicity and minimal overhead, and cannot be skinned.

The default layout class of the `DataGroup` container is `BasicLayout`. The default layout class of the `SkinnableDataContainer` class is `VerticalLayout`. For complete reference information, see the *Adobe Flex Language Reference*.

Creating a Spark `DataGroup` and Spark `SkinnableDataContainer` container

You use the `<s:DataGroup>` and `<s:SkinnableDataContainer>` tags to define a `DataGroup` and `SkinnableDataContainer` container. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The `DataGroup` and `SkinnableDataContainer` container are examples of data provider components. Data provider components require data for display or user interaction. To provide data, assign a collection which implements the `IList` interface, such as an `ArrayCollection` object or `XMLListCollection` object, to the container's `dataProvider` property. For more information on using data providers, see *Using Data Providers and Collections*.

The `dataProvider` property takes an array of children, as the following example shows:

```
<s:DataGroup itemRenderer=...>
  <s:dataProvider>
    <mx:ArrayCollection>
      <fx:String>Dave Jones</fx:String>
      <fx:String>Mary Davis</fx:String>
      <fx:String>Debbie Cooper</fx:String>
    </mx:ArrayCollection>
  <s:dataProvider>
</s:DataGroup>
```

If you are using Flex Components as children of the container, you can specify them as the following example shows:

```
<s:DataGroup itemRenderer=...>
  <s:dataProvider>
    <mx:ArrayCollection>
      <s:Button/>
      <s:Button/>
      <s:Button/>
    </mx:ArrayCollection>
  <s:dataProvider>
</s:DataGroup>
```

Because `dataProvider` is the default property of the `DataGroup` and `SkinnableDataContainer` container, you do not have to specify a `<s:dataProvider>` child tag. Therefore, you can write the example as shown below:

```
<s:DataGroup itemRenderer=...>
  <mx:ArrayCollection>
    <fx:String>Dave Jones</fx:String>
    <fx:String>Mary Davis</fx:String>
    <fx:String>Debbie Cooper</fx:String>
  </mx:ArrayCollection>
</s:DataGroup>
```

You can mix different types of data items in a container, or mix data items and Flex components. For example, you might mix String, Object, and XML data in the same container. However, you must define an item renderer function to apply the correct item renderer to the child. For more information, see “Using an item renderer function with a Spark container” on page 28.

You can skin the SkinnableDataContainer in the same way that you skin the container. For an example of a skin, see “Creating a Spark SkinnableContainer container” on page 10.

Using a default item renderer with a Spark container

The DataGroup and SkinnableDataContainer containers require an item renderer to draw each container child on the screen. By default, the DataGroup and SkinnableDataContainer containers do not define an item renderer. You can configure the containers to use the item renderers provided by Flex, or define your own custom item renderer.

Flex ships with two item renderers: spark.skins.spark.

- spark.skins.spark.DefaultItemRenderer Converts its data item to a single String for display in a Spark SimpleText control. It is useful when displaying a scalar data item, such as a String or a Number, that can be easily converted to a String.
- spark.skins.spark.DefaultComplexItemRenderer Displays a Flex component in a Group container. Each component is wrapped in its own Group container. Therefore, it is useful when the children of the container are visual elements, such as Flex components.

The following example uses the DefaultItemRenderer with a DataGroup container:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerDefaultRenderer.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:DataGroup itemRenderer="spark.skins.spark.DefaultItemRenderer">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
    <mx:ArrayCollection>
      <fx:String>Dave Jones</fx:String>
      <fx:String>Mary Davis</fx:String>
      <fx:String>Debbie Cooper</fx:String>
    </mx:ArrayCollection>
  </s:DataGroup>
</s:Application>
```

Each data item of the container is a String. Because you use the DefaultItemRenderer with the container, each String appears in the container in a SimpleText control.

If the data item is of type Object or is a data type that is not easily converted to a String, then you either have to convert it to a String, or define a custom item renderer to display it. For more information, see “Passing data to an item renderer” on page 20.

The following example shows a DataGroup container where all its children are Flex components. The DataGroup class uses the DefaultComplexItemRenderer to display each child:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerSimpleVisual.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:DataGroup itemRenderer="spark.skins.spark.DefaultComplexItemRenderer">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
    <mx:ArrayCollection>
      <s:Button/>
      <s:Button/>
      <s:Button/>
      <s:Button/>
    </mx:ArrayCollection>
  </s:DataGroup>
</s:Application>
```

Because you use the `DefaultComplexItemRenderer` with the container, each `Button` control appears in the container nested in its own `Group` container. By wrapping each control in a `Group` container, the item renderer can support selection highlighting for the individual children. However, if you do not want each control to appear in its own `Group` container, set the item renderer to `null`, as shown below:

```
<s:DataGroup itemRenderer="{null}">
```

Note: If you are only displaying visual elements in a `DataGroup` or `SkinnableDataContainer` container, you should instead use the `Group` or `SkinnableContainer` containers.

Adding and removing children at runtime

To modify the children of the `DataGroup` and `SkinnableDataContainer` containers at runtime, modify the `dataProvider` property. The following example uses event handlers to add and remove container children by calling the `addItem()` and `removeItemAt()` methods on the `dataProvider` property:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerAddRemoveChild.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <fx:Script>
    <![CDATA[

      private function addDGChild():void {
        var newChild:String = "new child";
        myDG.dataProvider.addItem(newChild);

        addDG.enabled = false;
        removeDG.enabled = true;
      }

      private function removeDGChild():void {
        myDG.dataProvider.removeItemAt(3);
```

```

        addDG.enabled = true;
        removeDG.enabled = false;
    }
    ]]>
</fx:Script>

<s:DataGroup id="myDG"
    itemRenderer="spark.skins.spark.DefaultItemRenderer">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>
    <mx:ArrayCollection>
        <fx:String>Dave Jones</fx:String>
        <fx:String>Mary Davis</fx:String>
        <fx:String>Debbie Cooper</fx:String>
    </mx:ArrayCollection>
</s:DataGroup>

<s:Button id="addDG" label="Add Child"
    click="addDGChild();" />
<s:Button id="removeDG" label="Remove Child"
    enabled="false"
    click="removeDGChild();" />
</s:Application>

```

For more information on using data providers, see [Using Data Providers and Collections](#).

Item renderer architecture

To better understand how item renderers work, examine the implementation of the default item renderers. Shown below is the code for the `DefaultItemRenderer` class:

```

<?xml version="1.0" encoding="utf-8"?>
<s:ItemRenderer focusEnabled="false"
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Script>
        <![CDATA[
            override public function set label(value:String):void {
                super.label = value;
                labelDisplay.text = label; |
            }
        ]]>
    </fx:Script>

    <s:states>
        <s:State name="normal" />
        <s:State name="hovered" />
        <s:State name="selected" />
        <s:State name="normalAndShowsCaret"/>
        <s:State name="hoveredAndShowsCaret"/>
        <s:State name="selectedAndShowsCaret"/>
    </s:states>

    <s:Rect left="0" right="0" top="0" bottom="0">

```

```

    <s:stroke.normalAndShowsCaret>
        <s:SolidColorStroke
            color="{selectionColor}"
            weight="1"/>
    </s:stroke.normalAndShowsCaret>
    <s:stroke.hoveredAndShowsCaret>
        <s:SolidColorStroke
            color="{selectionColor}"
            weight="1"/>
    </s:stroke.hoveredAndShowsCaret>
    <s:stroke.selectedAndShowsCaret>
        <s:SolidColorStroke
            color="{selectionColor}"
            weight="1"/>
    </s:stroke.selectedAndShowsCaret>
    <s:fill>
        <s:SolidColor
            color.normal="{contentBackgroundColor}"
            color.normalAndShowsCaret="{contentBackgroundColor}"
            color.hovered="{rollOverColor}"
            color.hoveredAndShowsCaret="{rollOverColor}"
            color.selected="{selectionColor}"
            color.selectedAndShowsCaret="{selectionColor}"
        />
    </s:fill>
</s:Rect>
<s:SimpleText id="labelDisplay"
    verticalCenter="0" left="3" right="3" top="6" bottom="4"/>
</s:ItemRenderer>

```

The base class of all item renderers is the `ItemRenderer` class. The `ItemRenderer` class is a subclass of the `Group` class, so it is itself a container. In the body of the `ItemRenderer` class, define the layout, states, and child controls of the item renderer used to represent the data item.

The default layout of the `ItemRenderer` class is `BasicLayout`. In this example, since there is no specification for the `layout` property, the item renderer uses `BasicLayout`.

The host component passes `String` data to the default item renderer by writing it to the `ItemRenderer.label` property. The default item renderer overrides this property to write the `String` data to the `SimpleText` control in the renderer.

The host component of the item renderer is called the item renderer's owner. From within the item renderer, you can access the host component by using the `ItemRenderer.owner` property. The `owner` property contains a reference back to the host component.

The item renderer can define view states. The normal view state is required. All other view states, such as `hovered`, `selected`, `normalAndShowCaret`, `hoveredAndShowCaret`, and `selectedAndShowCaret`, are optional. A custom item renderer can add additional view states. For more information on using view states in an item renderer, see "Defining item renderer view states for a Spark container" on page 25.

The `DefaultItemRenderer` class defines a rectangle, using the `Rect` component, to define the background color of the item renderer. The `Rect` component uses three colors for the three different view states of the item renderer, where:

- `contentBackgroundColor = 0xFFFFFFFF` (white)
- `rollOverColor = 0xCEDBEF`
- `selectionColor = 0xA8C6EE`

These colors are defined in the `default.css` file for the `flex4.swc` file. If you want your item renderer to mimic the color setting of Flex, you can use these same colors for setting the background color of your custom item renderers.

The `SimpleText` control is centered vertically in the display area of the item renderer, and is constrained to be three pixels in from the left border, three pixels in from the right border, six pixels in from the top border, and four pixels in from the bottom border. You can use these same settings in your custom item renderers to mimic the look of the default Flex item renderers, or change them as necessary for your application.

The id of the `SimpleText` control in the item renderer is `labelDisplay`. This is a specially named component in an item renderers that Flex uses to determine the baseline position of the text for display.

The `DefaultComplexItemRenderer` class is similar, but it uses a `Group` container to hold the Flex components defined as children of the container. You can see the source code for the `DefaultComplexItemRenderer` in the `spark/skins/default` directory of your Flex installation.

Defining a custom item renderer for a Spark container

You might be able to create your application by using just the `DefaultItemRenderer` and `DefaultComplexItemRenderer` classes. However, you typically define a custom item renderer if your data items are not simple values, or if you want more control over the appearance of your container children.

Passing data to an item renderer

The base class for item renderers, `ItemRenderer`, defines three properties that the item renderer owner uses to pass information to the renderer:

- `label` A `String` representation of the data item. The original data item is either a `String`, or the item renderer owner converts the data item to a `String`.
- `data` The original data item in its original representation.
- `owner` The component that hosts the item renderer. For example, the `SkinnableDataContainer` can be the owner of an item renderer.

The owner of an item renderer, including `SkinnableDataContainer`, must implement the `IItemRendererOwner` interface. That interface defines the following methods to write information to the item renderer:

- `itemToLabel()` Converts the data item to a `String` representation.
Components can override this method to customize the `String` conversion.
- `updateRenderer()` Write the data item as a `String` to the `ItemRenderer.label` property. Updates the `ItemRenderer.owner` property with a reference to the host component.

Components can override this method to write additional information to the item renderer.

There is no method to update the `ItemRenderer.data` property. That property is always initialized with the unprocessed data item.

Note: While `DataGroup` supports item renderers, it does not implement the `IItemRendererOwner` interface. However, it does still pass data to the item renderer by setting the `label`, `owner`, and `data` properties.

Before you create a custom item renderer, you must decide how to pass the data item to the item renderer. In some situations, you want the host component to perform any processing on the data item before it passes it to the item renderer. If so, override the `itemToLabel()` and `updateRenderer()` methods in the host component. The item renderer then access the data by using the `label` property.

Instead of the item renderer owner processing the data item, the item renderer can perform all of the processing. If you want the item renderer to process the data item, use `ItemRenderer.data` property to pass it. The item renderer then accesses the `data` property and perform any processing on the data item before displaying it.

For an example the overrides the `itemToLabel()` and `updateRenderer()` methods, see “Passing data using the `ItemRenderer.label` property” on page 21. For an example using the `data` property, see “Passing data using the `ItemRenderer.data` property” on page 24.

Passing data using the `ItemRenderer.label` property

If the data item is a `String`, or a value that can easily be converted to a `String`, you can use the `ItemRenderer.label` property to pass it to the item renderer. If the data item is in a format that must be converted to a `String` representation, override the `itemToLabel()` method in the owner to customize the conversion.

In the following example, the children of the `SkinnableDataContainer` container are `Strings` specifying different colors:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataContainerColor.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:MyComps="myComponents.*">

  <s:SkinnableDataContainer
    itemRenderer="myComponents.MySimpleColorRenderer">
    <mx:ArrayCollection>
      <fx:String>red</fx:String>
      <fx:String>green</fx:String>
      <fx:String>blue</fx:String>
    </mx:ArrayCollection>
  </s:SkinnableDataContainer>
</s:Application>
```

This example uses a custom item renderer named `MySimpleColorRenderer`, defined in the file `MySimpleColorRenderer.mxml`, that shows the `String` text in a background of the matching color. Shown below is the item renderer:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleColorRenderer.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <fx:Script>
    <![CDATA[

      // Property to hold the RGB color value.
      [Bindable]
      public var myColor:uint;

      // Write String to labelDisplay component.
      override public function set label(value:String):void
      {
        super.label = value;
        labelDisplay.text = label;

        // Determine the RGB color value from the data item.
```

```

        if (label == "red")
            myColor = 0xFF0000;
        if (label == "green")
            myColor = 0x00FF00;
        if (label == "blue")
            myColor = 0x0000FF;
    }
    ]]>
</fx:Script>

<s:states>
    <s:State name="normal" />
</s:states>

<!-- Set the background color to the RGB color value.-->
<s:Rect width="100%" height="100%" alpha="0.5">
    <s:fill>
        <s:SolidColor color="{myColor}" />
    </s:fill>
</s:Rect>

<!-- Display the color name -->
<s:SimpleText id="labelDisplay"/>
</s:ItemRenderer>

```

In this example, the item renderer overrides the `label` property to write the color to the `SimpleText` control, and set the fill color of the `Rect` component.

This item renderer displays the data with no background colors, or any visual changes based on state. This item renderer is useful for displaying data in the container when it does not have any user interaction. For an example of an item renderer that changes its display based on user interaction, see “Defining item renderer view states for a Spark container” on page 25.

If you want to modify the `String` passed to the `label` property, you can override the `itemToLabel()` method in the host component. The `itemToLabel()` method has the following signature:

```
itemToLabel(item:Object):String
```

The method takes a single argument representing the data item. It returns a `String` representation of the data item for display in the item renderer.

In the following example, each data item is represented by an `Object` containing three fields. The custom `SkinnableDataContainer` container, called `MyDataGroup`, overrides the `itemToLabel()` method to format the `Object` as a `String` before passing the `String` to the item renderer. The example then uses the `DefaultItemRenderer` to display the text:

```

<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerOverride.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:MyComps="myComponents.*">

  <!-- Define a custom DataGroup container to override the itemToLabel() method. -->
  <MyComps:MyDataGroup itemRenderer="spark.skins.spark.DefaultItemRenderer">
    <MyComps:layout>
      <s:VerticalLayout/>
    </MyComps:layout>

    <mx:ArrayCollection>
      <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
      <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
      <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
      <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
    </mx:ArrayCollection>
  </MyComps:MyDataGroup>
</s:Application>

```

The MyDataGroup.mxml file defines the custom SkinnableDataContainer container that overrides the itemToLabel() method:

```

<?xml version="1.0" encoding="utf-8"?>
<s:SkinnableDataContainer xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:s="library://ns.adobe.com/flex/spark"
  xmlns:mx="library://ns.adobe.com/flex/halo">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <fx:Script>
    <![CDATA[
      import mx.core.IVisualElement;
      import spark.components.IItemRenderer;

      // Override to return the Object as a formatted String.
      override public function itemToLabel(item:Object):String {
        var tempString:String;
        if (item == null)
          return " ";

        tempString = item.firstName + " " + item.lastName
          + " " + ", ID: " + item.companyID;
        return tempString;
      }
    ]]>
  </fx:Script>
</s:SkinnableDataContainer>

```

Passing data using the `ItemRenderer.data` property

Rather than processing the data in the host component, you can let the item renderer perform all the processing of the data item for display. In this situation, use the `ItemRenderer.data` property to pass the data item to the item renderer. This technique lets you define a set of item renderers that display the same data in different ways depending on your application requirements.

In the next example, each data item is represented by an Object that defines three fields:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerSimple.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:SkinnableDataContainer
    itemRenderer="myComponents.MySimpleItemRenderer">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
    <mx:ArrayCollection>
      <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
      <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
      <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
      <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
    </mx:ArrayCollection>
  </s:SkinnableDataContainer>
</s:Application>
```

The `SkinnableDataContainer` uses a custom item renderer named `MySimpleItemRenderer.mxml`. The custom item renderer displays the `firstName` and `lastName` fields in a single `SimpleText` control, and display the `companyID` in a second `SimpleText` control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleItemRenderer.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:states>
    <s:State name="normal"/>
  </s:states>

  <s:HGroup verticalCenter="0" left="2" right="2" top="2" bottom="2">
    <s:SimpleText text="{data.lastName}, {data.firstName}"/>
    <s:SimpleText text="{data.companyID}"/>
  </s:HGroup>
</s:ItemRenderer>
```

The `data` property contains an Object passed from the `DataGroup` container. The renderer uses data binding to populate the controls in the item renderer from the `data` property. The two `SimpleText` controls are defined in a `Group` container so that they can be layed out horizontally.

Rather than using data binding, you can override the `data` property in the item renderer. Within the override, you can then set any other properties in the renderer.

Defining item renderer view states for a Spark container

Item renderers must define the normal view state. The normal view states correspond to the normal state of the item when the item has no user interaction. The other view states defined in the `DefaultItemRenderer` and `DefaultComplexItemRenderer` classes include the following:

- `hovered` The mouse is over the item.
- `selected` The item is selected.
- `normalAndShowCaret` The item is in the normal state, and it has focus in the item list.
- `hoveredAndShowCaret` The item is in the hovered state, and it has focus in the item list.
- `selectedAndShowCaret` The item is in the normal state, and it has focus in the item list.

The `selected`, `normalAndShowCaret`, `hoveredAndShowCaret`, and `selectedAndShowCaret` view states are supported by the list-based components. The list-based components are subclasses of the `spark.components.supportClasses.ListBase` class. The `DataGroup` and `SkinnableDataContainer` containers do not implement these view states. However, they are supported by the `DefaultItemRenderer` and `DefaultComplexItemRenderer` classes so that you can use those item renderers with list-based classes. For more information, see *Working with item renderers*.

Your item renderer can define additional states based on the requirements of your application.

The container hosting the item renderer sets the view state of the item renderer. You can choose to alter the display of the data renderer based on the view state, or you can do nothing on a change of view state. The only required view state is the normal view state. If the owner of the item renderer attempts to set the item renderer to an undefined view state, the item renderer uses the normal view state.

In the next example, you define a `SkinnableDataContainer` to display an `Object` with four fields: `firstName`, `lastName`, `companyID`, and `phone`:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerSimpleStates.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:SkinnableDataContainer itemRenderer="myComponents.MySimpleItemRendererWithStates">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
    <mx:ArrayCollection>
      <fx:Object firstName="Bill" lastName="Smith"
        companyID="11233" phone="617-555-1212"/>
      <fx:Object firstName="Dave" lastName="Jones"
        companyID="13455" phone="617-555-1213"/>
      <fx:Object firstName="Mary" lastName="Davis"
        companyID="11543" phone="617-555-1214"/>
      <fx:Object firstName="Debbie" lastName="Cooper"
        companyID="14266" phone="617-555-1215"/>
    </mx:ArrayCollection>
  </s:SkinnableDataContainer>
</s:Application>
```

The following item renderer displays the text of the `SimpleText` controls in bold, blue font for the hover state:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleItemRendererWithStates.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:states>
    <s:State name="normal"/>
    <s:State name="hovered"/>
  </s:states>

  <s:Rect left="0" right="0" top="0" bottom="0">
    <s:fill>
      <s:SolidColor color="0xFFFFFFFF" />
    </s:fill>
    <s:fill.hovered>
      <s:SolidColor color="0xCEDBEF" />
    </s:fill.hovered>
  </s:Rect>

  <s:HGroup verticalCenter="0" horizontalCenter="0">
    <s:SimpleText text="{data.lastName}, {data.firstName}"
      color.hovered="blue"
      fontWeight.hovered="bold"/>
    <s:SimpleText text="{data.companyID}"
      color.hovered="blue"
      fontWeight.hovered="bold"/>
    <s:SimpleText text="{data.phone}"
      color.hovered="blue"
      fontWeight.hovered="bold"/>
  </s:HGroup>
</s:ItemRenderer>
```

Because a `SkinnableDataContainer` does not support the selected view state, the item renderer does not define any settings for the selected state.

You can also include a transition in an item renderer. A transition plays whenever you change a view state. The following item renderer uses a transition to display the company ID and telephone number of the employee on mouse over:

```

<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleItemRendererWithStates.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:states>
    <s:State name="normal"/>
    <s:State name="hovered"/>
  </s:states>

  <s:transitions>
    <s:Transition fromState="normal">
      <s:Sequence>
        <s:Resize target="{this}" />
        <mx:SetPropertyAction targets="{[cID, empPhone]}"
          name="visible" value="true" />
      </s:Sequence>
    </s:Transition>
    <s:Transition toState="normal">
      <s:Sequence>
        <mx:SetPropertyAction targets="{[cID, empPhone]}"
          name="visible" value="false" />
        <s:Resize target="{this}" />
      </s:Sequence>
    </s:Transition>
  </s:transitions>

  <s:Rect left="0" right="0" top="0" bottom="0">
    <s:fill>
      <s:SolidColor color="0xFFFFFFFF" />
    </s:fill>
    <s:fill.hovered>
      <s:SolidColor color="0xCEDBEF" />
    </s:fill.hovered>
  </s:Rect>

  <s:VGroup verticalCenter="0" horizontalCenter="0">
    <s:SimpleText text="{data.lastName}, {data.firstName}"
      color.hovered="blue"
      fontWeight.hovered="bold"/>
    <s:SimpleText id="cID"
      includeIn="hovered"
      includeInLayout.normal="false"
      text="{data.companyID}"/>
    <s:SimpleText id="empPhone"
      includeIn="hovered"
      includeInLayout.normal="false"
      text="{data.phone}"/>
  </s:VGroup>
</s:ItemRenderer>

```

The following application uses this item renderer:

```

<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerSimpleStatesTransition.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:SkinnableDataContainer itemRenderer="myComponents.MySimpleItemRendererWithTrans">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
    <mx:ArrayCollection>
      <fx:Object firstName="Bill" lastName="Smith"
        companyID="11233" phone="617-555-1212"/>
      <fx:Object firstName="Dave" lastName="Jones"
        companyID="13455" phone="617-555-1213"/>
      <fx:Object firstName="Mary" lastName="Davis"
        companyID="11543" phone="617-555-1214"/>
      <fx:Object firstName="Debbie" lastName="Cooper"
        companyID="14266" phone="617-555-1215"/>
    </mx:ArrayCollection>
  </s:SkinnableDataContainer>
</s:Application>

```

For more information on view states, see [Using View States](#). For more information on transitions, see [Transitions](#).

Using an item renderer function with a Spark container

In some applications, you display different types of data items in one container. In this scenario, each type of data item needs its own item renderer. Or, you mix data items and Flex components in a container. To mix data items and Flex components, define different item renderers for the data items and the Flex components.

You can use an item renderer function to examine each child to determine which item renderer to use. The `DataGroup.itemRendererFunction` and `SkinnableDataContainer.itemRendererFunction` property takes a function with the following signature:

```
function itemRendererFunction(item:Object):ClassFactory
```

Where `item` is the data item, and the return value is the item renderer. If the child is a Flex component, return `DefaultComplexItemRenderer` to display the child in a Group container, or return `null` to display the child with no renderer.

The following example defines an item renderer function to return one item renderer for data items defined as `Object`, and another item renderer for data items defined as `String`:

```

<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerFunction.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <fx:Script>
    <![CDATA[

      import myComponents.MySimpleItemRendererFunction;
      import spark.skins.spark.DefaultItemRenderer;

      private function selectRenderer(item:Object):ClassFactory {
        var classFactory:ClassFactory;
        if (item is String) {
          // If the item is a String, use DefaultItemRenderer.
          classFactory = new ClassFactory(DefaultItemRenderer);
        }
        else {
          // If the item is an Object, use MySimpleItemRendererFunction.
          classFactory = new ClassFactory(MySimpleItemRendererFunction);
        }
        return classFactory;
      }
    ]]>
  </fx:Script>

  <s:DataGroup itemRendererFunction="selectRenderer">
    <s:layout>
      <s:TileLayout requestedColumnCount="3"/>
    </s:layout>
    <mx:ArrayCollection>
      <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
      <fx:String>617-555-1212</fx:String>
      <fx:String>Newton</fx:String>
      <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
      <fx:String>617-555-5555</fx:String>
      <fx:String>Newton</fx:String>
      <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
      <fx:String>617-555-6666</fx:String>
      <fx:String>Newton</fx:String>
    </mx:ArrayCollection>
  </s:DataGroup>
</s:Application>

```

You also use item renderer function when mixing data items and Flex components in the container. Flex components implement the `IVisualElement` interface, and therefore do not need an item renderer to draw them on the screen. In your item renderer function, you can determine if the data item corresponds to a Flex component. If so, the item renderer function returns the `DefaultComplexItemRenderer`, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerFunctionVisual.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <fx:Script>
    <![CDATA[
      import mx.core.IVisualElement;

      import myComponents.MySimpleItemRenderereEmployee;
      import spark.skins.spark.DefaultComplexItemRenderer;

      private function selectRenderer(item:Object):ClassFactory {
        var classFactory:ClassFactory;
        if(item is IVisualElement){
          // If the item is a Flex component, use DefaultComplexItemRenderer.
          classFactory = new ClassFactory(DefaultComplexItemRenderer);
        }
        else if (item is Object){
          // If the item is an Object, use MySimpleItemRenderereFunction.
          classFactory = new ClassFactory(MySimpleItemRenderereEmployee);
        }
        return classFactory;
      }
    ]]>
  </fx:Script>

  <s:DataGroup itemRendererFunction="selectRenderer">
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
    <mx:ArrayCollection>
      <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
      <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
      <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
      <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
      <s:Button label="Add Employee"/>
    </mx:ArrayCollection>
  </s:DataGroup>
</s:Application>
```

Defining an inline item renderer for a Spark container

The examples of item renderers shown above are all defined in an MXML file. That makes the item renderer highly reusable because you can reference it from multiple containers.

You can also define inline item renderers in the MXML definition of a component. By using an inline item renderer, your code can all be defined in a single file. However, it is not easy to reuse an inline item renderer.

The following example uses an inline item renderer with the DataGroup container:

```

<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerInline.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:DataGroup>
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
    <mx:ArrayCollection>
      <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
      <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
      <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
      <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
    </mx:ArrayCollection>
    <s:itemRenderer>
      <fx:Component>
        <s:ItemRenderer>
          <s:states>
            <s:State name="normal"/>
            <s:State name="hovered"/>
            <s:State name="selected"/>
          </s:states>

          <s:Group verticalCenter="0" left="2" right="2" top="2" bottom="2">
            <s:layout>
              <s:HorizontalLayout/>
            </s:layout>
            <s:SimpleText text="{data.lastName}, {data.firstName}"/>
            <s:SimpleText text="{data.companyID}"/>
          </s:Group>
        </s:ItemRenderer>
      </fx:Component>
    </s:itemRenderer>
  </s:DataGroup>
</s:Application>

```

Notice that you define the item renderer inline by using the `itemRenderer` property of the `DataGroup` container. The first child tag of the `itemRenderer` property is always the `<fx:Component>` tag. Inside the `<fx:Component>` tag is the same code as was in the `MySimpleItemRenderer.mxml` file shown above.

Items allowed in an inline component

There is only one restriction on what you can and cannot do in an inline item renderer. You cannot create an empty `<fx:Component></fx:Component>` tag. For example, you can combine effect and style definitions in an inline item renderer along with your rendering logic.

You can include the following items in an inline item renderer:

- Binding tags
- Effect tags
- Metadata tags
- Model tags
- Scripts tags

- Service tags
- State tags
- Style tags
- XML tags
- `id` attributes, except for the top-most component

Using the Component tag

The `<fx:Component>` tag defines a new scope in an MXML file, where the local scope of the item renderer is defined by the MXML code block delimited by the `<fx:Component>` and `</fx:Component>` tags. To access elements outside the local scope of the item renderer, you prefix the element name with the `outerDocument` keyword.

For example, you define one variable named `localVar` in the scope of the main application, and another variable with the same name in the scope of the item renderer. From within the item renderer, you access the application's `localVar` by prefixing it with `outerDocument` keyword, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerInlineScope.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <fx:Script>
    <![CDATA[
      // Variable in the Application scope.
      [Bindable]
      public var localVar:String="Application scope";
    ]]>
  </fx:Script>

  <s:DataGroup>
    <s:layout>
      <s:VerticalLayout/>
    </s:layout>
    <mx:ArrayCollection>
      <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
      <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
      <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
      <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
    </mx:ArrayCollection>
    <s:itemRenderer>
      <fx:Component>
        <s:ItemRenderer>
          <fx:Script>
            <![CDATA[
              // Variable in the Renderer scope.
              [Bindable]
              public var localVar:String="Renderer scope";
            ]]>
          </fx:Script>
          <s:states>
            <s:State name="normal"/>
          </s:states>
        </s:ItemRenderer>
      </fx:Component>
    </s:itemRenderer>
  </s:DataGroup>
</s:Application>
```

```

        <s:State name="hovered"/>
        <s:State name="selected"/>
    </s:states>

    <s:Group verticalCenter="0" left="2" right="2" top="2" bottom="2">
        <s:layout>
            <s:HorizontalLayout/>
        </s:layout>
        <s:SimpleText text="{data.lastName}, {data.firstName}"/>
        <s:SimpleText text="{data.companyID}"/>
        <s:SimpleText
            text="'Renderer localVar = ' + localVar"/>
        <s:SimpleText
            text="'Application localVar = ' + outerDocument.localVar"/>
    </s:Group>
</s:ItemRenderer>
</fx:Component>
</s:itemRenderer>
</s:DataGroup>
</s:Application>

```

Creating a reusable inline item renderer

Rather than defining an inline item renderer in the definition of a component, you can define a reusable inline item renderer for use in multiple locations in your application.

To create a reusable inline item renderer, specify the `className` property of the `<fx:Component>` tag. By naming the class, you define a way to reference the item renderer, and the elements of the item renderer.

The following example uses the `<fx:Component>` tag to define an inline item renderer for two containers:

```

<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerInlineReuse.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Declarations>
        <fx:Component id="inlineRenderer">
            <s:ItemRenderer>
                <s:states>
                    <s:State name="normal"/>
                    <s:State name="hovered"/>
                    <s:State name="selected"/>
                </s:states>

                <s:Group verticalCenter="0" horizontalCenter="0">
                    <s:layout>
                        <s:HorizontalLayout/>
                    </s:layout>
                    <s:SimpleText text="{data.lastName}, {data.firstName}"/>
                    <s:SimpleText text="{data.companyID}"/>
                </s:Group>
            </s:ItemRenderer>

```

```

        </fx:Component>
    </fx:Declarations>

    <s:DataGroup itemRenderer="{inlineRenderer}">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayCollection>
            <fx:Object firstName="Bill" lastName="Smith" companyID="11233"/>
            <fx:Object firstName="Dave" lastName="Jones" companyID="13455"/>
            <fx:Object firstName="Mary" lastName="Davis" companyID="11543"/>
            <fx:Object firstName="Debbie" lastName="Cooper" companyID="14266"/>
        </mx:ArrayCollection>
    </s:DataGroup>
    <s:SkinnableDataContainer itemRenderer="{inlineRenderer}">
        <s:layout>
            <s:VerticalLayout/>
        </s:layout>
        <mx:ArrayCollection>
            <fx:Object firstName="Jim" lastName="Sullivan" companyID="11233"/>
            <fx:Object firstName="Joan" lastName="Connors" companyID="13455"/>
            <fx:Object firstName="Jack" lastName="Wilson" companyID="11543"/>
            <fx:Object firstName="Jeff" lastName="Lodge" companyID="14266"/>
        </mx:ArrayCollection>
    </s:SkinnableDataContainer>
</s:Application>

```

In this example, you use data binding to specify the renderer as the value of the `itemRenderer` property for the two containers.

Spark item renderer precedence

The `DataGroup` and `SkinnableDataContainer` containers use the follow rules to determine the item renderer for a child:

- 1 If the `itemRendererFunction` property is defined, call the associated function to obtain the item renderer. If the function returns `null`, go to rule 2.
- 2 If the `itemRenderer` property is defined, use the specified item renderer to display the item.
- 3 If the item implements `mx.core.IVisualElement` and is of type `flash.display.DisplayObject`, use it directly.
- 4 Dispatch a runtime error if no item renderer found.

Using virtualization with Spark `DataGroup` and `SkinnableDataContainer`

A `DataGroup` or `SkinnableDataContainer` container can represent any number of children. However, each child requires an instance of an item renderer. If the container has many children, you might notice performance degradation as you add more children to the container.

Instead of creating an item renderer for each child, you can configure the container to use a virtual layout. With virtual layout, the container reuses item renderers so that it only creates item renderers for the currently visible children of the container. As a child is moved off the screen, possible by scrolling the container, a new child being scrolled onto the screen can reuse its item renderer.

To configure a container to use virtual layout, set the `useVirtualLayout` property to `true` for the layout associated with the container. Only the `DataGroup` or `SkinnableDataContainer` with the `VerticalLayout`, `HorizontalLayout`, and `TileLayout` supports virtual layout.

Note: *If you define an `itemRendererFunction` to determine the item renderer for each data item, you cannot use virtualization. The `itemRendererFunction` must examine each data item and create the item renderers as necessary for the specific data item type.*

There are a few differences between the way a layout class works when virtual layout is enabled and when it is disabled:

- A layout with virtual layout enabled does not support the layout's major axis percent size property. This axis corresponds to the `percentHeight` property for the `VerticalLayout` class, and the `percentWidth` property for the `HorizontalLayout` class.
- A container using a virtual layout that contains few children whose sizes vary widely can respond poorly to interactive scrolling using the scroll thumb. No performance degradation occurs when scrolling using the scroll arrows or by clicking in the scroll track. Responsiveness improves as the variation in size decreases or the number of children increases.

Use virtual layout when the cost of creating or measuring a `DataGroup` is prohibitive because of the number of data elements or the complexity of the item renderers.

Creating a recyclable item renderer

With virtual layout disabled, the `DataGroup` and `SkinnableDataContainer` containers create one instance of the item renderer for each child. With virtual layout enabled, the container only creates enough item renderers to display its currently visible children. Virtual layout greatly reduces the overhead required to use the `DataGroup` and `SkinnableDataContainer` containers.

With virtual layout enabled, when a child is moved off the visible area of the container, its item renderer is recycled. First, the item renderer's `data` property is set to `null`. When the item renderer is reused, its `data` property is set to the data item representing the new child. Therefore, if a recycled item renderer performs any actions based on the value of the `data` property, it must first check that the property is not `null`.

When the item renderer is reassigned, Flex also calls the `updateRenderer()` method of the item renderer owner. This method must set the `owner` and `label` properties on the item renderer. Subclasses of `SkinnableDataContainer`, can use the `updateRenderer()` method to set additional properties on the item renderer.

Because a container can reuse an item renderer, ensure that you fully define its state. For example, you use a `CheckBox` control in an item renderer to display a `true` (checked) or `false` (unchecked) value based on the current value of the `data` property. A common mistake is to assume that the `CheckBox` control is always in its default state of unchecked and only inspect the `data` property for a value of `true`.

However, remember that the `CheckBox` can be recycled and had previously been checked. Therefore, inspect the `data` property for a value of `false`, and explicitly uncheck the control if it is checked, as the following example shows:

```

<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleItemRendererCB.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark"
  dataChange="setMgr();">

  <fx:Script>
    <![CDATA[
      private function setMgr():void {
        // Check to see if the data property is null.
        if (data == null)
          return;
        // If the data property is not null,
        // set the CheckBox control appropriately..
        if (data.manager == "yes") {
          mgr.selected = true;
        }
        else {
          mgr.selected = false;
        }
      }
    ]]>
  </fx:Script>

  <s:states>
    <s:State name="normal"/>
    <s:State name="hovered"/>
    <s:State name="selected"/>
  </s:states>

  <s:HGroup verticalCenter="0" left="2" right="2" top="2" bottom="2">
    <s:SimpleText text="{data.lastName}, {data.firstName}"/>
    <s:SimpleText text="{data.companyID}"/>
    <s:CheckBox id="mgr"/>
  </s:HGroup>
</s:ItemRenderer>

```

Use the `dataChange` event of the `ItemRenderer` class to detect the change to its `data` property. This event is dispatched whenever the `data` property changes. Alternatively, you can override the `data` property.

Alternatively, you can override the `ItemRenderer.data` property itself, as the following example shows:

```

<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\myComponents\MySimpleItemRendererCBData.mxml -->
<s:ItemRenderer xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <fx:Script>
    <![CDATA[

      override public function set data(value:Object):void {
        super.data = value;

        // Check to see if the data property is null.
        if (value== null)
          return;
        // If the data property is not null,
        // set the CheckBox control appropriately..
        if (value.manager == "yes") {
          mgr.selected = true;
        }
        else {
          mgr.selected = false;
        }
      }
    ]]>
  </fx:Script>

  <s:states>
    <s:State name="normal"/>
    <s:State name="hovered"/>
    <s:State name="selected"/>
  </s:states>

  <s:HGroup verticalCenter="0" left="2" right="2" top="2" bottom="2">
    <s:SimpleText text="{data.lastName}, {data.firstName}"/>
    <s:SimpleText text="{data.companyID}"/>
    <s:CheckBox id="mgr"/>
  </s:HGroup>
</s:ItemRenderer>

```

Defining a typical item for determining the size of an item renderer

When using virtual layout with the `DataGroup` and `SkinnableDataContainer` containers, you can pass to the container a data item that defines a typical data item. The container then uses the typical data item, and the associated item renderer, to determine the default size of the child. By defining the typical item, the container does not have to size each child as it is drawn on the screen.

Use the `typicalItem` property of the container to specify the data item, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- containers\spark\SparkDataGroupContainerTypicalItem.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
  xmlns:mx="library://ns.adobe.com/flex/halo"
  xmlns:s="library://ns.adobe.com/flex/spark">

  <s:layout>
    <s:VerticalLayout/>
  </s:layout>

  <fx:Script>
    <![CDATA[
      [Bindable]
      public var typicalObj:Object = {
        firstName:"Long first name",
        lastName:"Even longer last name",
        companyID:"123456",
        manager:"yes"
      };
    ]]>
  </fx:Script>

  <s:Scroller>
    <s:DataGroup itemRenderer="myComponents.MySimpleItemRendererCB"
      height="100"
      typicalItem="{typicalObj}" >
      <s:layout>
        <s:VerticalLayout useVirtualLayout="true"/>
      </s:layout>
      <mx:ArrayCollection>
        <fx:Object firstName="Bill" lastName="Smith"
          companyID="11233" manager="yes"/>
        <fx:Object firstName="Dave" lastName="Jones"
          companyID="13455" manager="no"/>
        <fx:Object firstName="Mary" lastName="Davis"
          companyID="11543" manager="yes"/>
        <fx:Object firstName="Debbie" lastName="Cooper"
          companyID="14266" manager="no"/>
      </mx:ArrayCollection>
    </s:DataGroup>
  </s:Scroller>
</s:Application>
```

```

    <fx:Object firstName="Bob" lastName="Martins"
      companyID="11233" manager="yes"/>
    <fx:Object firstName="Jack" lastName="Jones"
      companyID="13455" manager="no"/>
    <fx:Object firstName="Sam" lastName="Johnson"
      companyID="11543" manager="yes"/>
    <fx:Object firstName="Tom" lastName="Fitz"
      companyID="14266" manager="no"/>
    <fx:Object firstName="Dave" lastName="Mead"
      companyID="11233" manager="yes"/>
    <fx:Object firstName="Dave" lastName="Jones"
      companyID="13455" manager="no"/>
    <fx:Object firstName="Mary" lastName="Davis"
      companyID="11543" manager="yes"/>
    <fx:Object firstName="Debbie" lastName="Cooper"
      companyID="14266" manager="no"/>
  </mx:ArrayCollection>
</s:DataGroup>
</s:Scroller>
</s:Application>

```

In this example, you define `typicalObj`, an Object that represents a data item with a long value for the `firstName` and `lastName` fields. You then pass `typicalObj` to the `typicalItem` property of the container. The container uses that data item, and the associated item renderer, to determine the size of the children.

Specifying a value for the `typicalItem` property passes that value, and the associated item renderer, to the `typicalLayoutElement` property of the layout of the container. For more information on the `typicalLayoutElement` property, see “Setting the row height or column width of a layout” on page 6.