Chapter 8: Sizing and Positioning Components

Adobe® Flex™ lays out components by determining their sizes and positions; it provides you with multiple options for controlling both sizes and positions.

Topics

About sizing and positioning

Flex controls the layout of components by using a set of rules. The layout rules are a combination of sizing rules for individual components, and sizing and positioning rules for containers. Flex supports automatic layout, so you often do not have to initially set the size or position of components. Instead, you can concentrate on building the logic of your application and let Flex control the layout. Later, you can adjust the dimensions of instances, if necessary.

Each container has its own rules for controlling layout. For example, the VBox container lays out its children in a single column. A Grid container lays out its children in rows and columns of cells. The Application container has 24-pixel padding, and many other containers have 0-pixel padding.

Although Flex has built-in default layout rules, you can use the component's properties and methods to customize the layout. All components have several properties, including `height` and `width`, for specifying the component's size in absolute or container-relative terms. Each container also has properties and styles that you can use to configure aspects of layout. You can use settings such as the `verticalGap` and `horizontalGap` styles of a Tile container to set the spacing between children, and the `direction` property to specify a row or column layout. You can also use different positioning techniques for laying out components in a container; some containers, for example, support absolute *x*- and *y*-coordinate–based positioning.

About layout in Flex

The Layout Manager controls layout in Flex. The manager uses the following three-stage process to determine the size and position of each component in an application:

Stage 1 - Commitment pass Determines the property settings of the application's components. This phase allows components whose contents depend on property settings to configure themselves before Flex determines their sizes and positions.

During the commitment pass, the Layout Manager causes each component to run its `commitProperties()` method, which determines the property values.

Stage 2 - Measurement pass Calculates the default size of every component in the application. This pass starts from the most deeply nested components and works out toward the Application container. The measurement pass determines the *measured*, or default, size of each component. The default size of each container is based on the default or explicit (if specified) sizes of its children. For example, the Box container's default width is equal to the sum of the default or explicit widths of all of its children, plus the thickness of the borders, plus the padding, plus the gaps between the children.

During the measurement pass, the Layout Manager causes each component to run its `measureSizes()` method (a private method that calls the `measure()` method) to determine the component's default size.

Stage 3 - Layout pass Lays out your application, including moving and resizing any components. This pass starts from the outermost container and works in toward the innermost component. The layout pass determines the actual size and placement of each component. It also does any programmatic drawing, such as calls to the `lineTo()` or `drawRect()` methods.

During the layout pass, Flex determines whether any component's sizing properties specify dimensions that are a percentage of the parent, and uses the setting to determine the child component's actual size. The Layout Manager causes each component to run its `updateDisplayList()` method to lay out the component's children; for this reason, this pass is also referred to as the update pass.

About Flex frames of reference

Flex uses several frames of reference in determining positions and sizes:

- The local area and local coordinate system are relative to the outer edges of the component. The component's visual elements, such as borders and scroll bars, are included in the local coordinates.
- The viewable area is the region of a component that is inside the component's visual elements; that is, it is the part of the component that is being displayed and can contain child controls, text, images, or other contents. Flex does not have a separate coordinate system for this area.
- The content area and content coordinate system include *all* of the component's contents, and do not include the visual elements. They include any regions that are currently clipped from view and must be accessed by scrolling the component. The content area of a scrolling TextArea control, for example, includes the region of text that is currently scrolled off the screen.

Flex uses the viewable area when it determines percentage-based sizes and when it performs constraint-based layout.

Flex component `x` and `y` properties, which you use to specify absolute positioning, are in the content coordinate system.

Note: *Flex coordinates increase from the upper-left corner of the frame of reference. Thus, an* x,y *position of 100,300 in the local coordinate system is 100 pixels to the right and 300 pixels down from the component's upper-left corner.*

For more information on Flex coordinate systems, see "Using Flex coordinates" on page 344.

About component sizing

The measurement and layout passes determine a component's height and width. You can get these dimensions by using the `height` and `width` properties; you can use these properties and others to control the component's size.

Flex provides several ways for you to control the size of controls and containers:

Default sizing Automatically determines the sizes of controls and containers.

Explicit sizing You set the `height` and `width` properties to absolute values.

Percentage-based sizing You specify the component size as a percentage of its container size.

Constraint-based layout You control size and position by anchoring component's sides to locations in their container.

For details on controlling component sizes, see "Sizing components" on page 149.

About component positioning

Flex positions components when your application initializes. Flex also performs a layout pass and positions or repositions components when the application or a user does something that could affect the sizes or positions of visual elements, such as the following situations:

- The application changes properties that specify sizing, such as `x`, `y`, `width`, `height`, `scaleX`, and `scaleY`.
- A change affects the calculated width or height of a component, such as when the label text for a Button control changes, or the user resizes a component.
- A child is added or removed from a container, a child is resized, or a child is moved. For example, if your application can change the size of a component, Flex updates the layout of the container to reposition its children, based on the new size of the child.
- A property or style that requires measurement and drawing, such as `horizontalScrollPolicy` or `fontFamily`, changes.

There are very few situations where an application programmer must force the layout of a component; for more information, see "Manually forcing layout" on page 149.

Flex provides two mechanisms for positioning and laying out controls:

Automatic positioning Flex automatically positions a container's children according to a set of container- and component-specific rules. Most containers, such as Box, Grid, or Form, use automatic positioning. Automatic positioning is sometimes referred to as automatic layout.

Absolute positioning You specify each child's, `x` and `y` properties, or use a constraint-based layout that specifies the distance between one or more of the container's sides and the child's sides, baseline, or center. Absolute positioning is sometimes referred to as absolute layout.

Three containers support absolute positioning:

- The Application and Panel containers use automatic positioning by default, and absolute positioning if you specify the `layout` property as `"absolute"`.
- The Canvas container always uses absolute positioning.

For details on controlling the positions of controls, see "Positioning and laying out controls" on page 163.

Component layout patterns

Flex uses different patterns to lay out different containers and their children. These patterns generally fit in the type categories listed in the following table. The table describes the general layout behavior for each type, how the default size of the container is determined, and how Flex sizes percentage-based children.

Container type	Default layout behavior
Absolute positioning: Canvas, container or Application or Panel container with `layout="absolute"`	**General layout:** Children of the container do not interact. That is, children can overlap and the position of one child does not affect the position of any other child. You specify the child positions explicitly or use constraints to anchor the sides, baselines, or centers of the children relative to the parent container. **Default sizing:** The measurement pass finds the child with the lowest bottom edge and the child with the rightmost edge, and uses these values to determine the container size. **Percentage-based children:** Sizing uses different rules depending on whether you use constraint-based layout or *x*- and *y*- coordinate positioning. See "Sizing percentage-based children of a container with absolute positioning" on page 157.
Controls that arrange all children linearly, such as Box, HBox, VBox	**General layout:** All children of the container are arranged in a single row or column. Each child's height and width can differ from all other children's heights or widths. **Default sizing:** The container fits the default or explicit sizes of all children and all gaps, borders, and padding. **Percentage based children:** If children with percentage-based sizing request more than the available space, the actual sizes are set to fit in the space, proportionate to the requested percentages.
Grid	**General layout:** The container is effectively a VBox control with rows of HBox child controls, where all items are constrained to align with each other. The heights of all the cells in a single row are the same, but each row can have a different height. The widths of all cells in a single column are the same, but each column can have a different width. You can define a different number of cells for each row or each column of the Grid container, and individual cells can span columns or rows. **Default sizing:** The grid fits the individual rows and children at their default sizes. **Percentage-based children:** If children use percentage-based sizing, the sizing rules fit the children GridItem components within their rows, and GridRow components within the grid size according to linear container sizing rules.
Tile	**General layout:** The container is a grid of equal-sized cells. The cells can be in row-first or column-first order. If you do not specify explicit or percentage-based dimensions, the control has as close as possible to an equal number of rows and columns, with the `direction` property determining the orientation with the larger number of items, if necessary. **Default sizing:** If you do not specify `tileWidth` and `tileHeight` properties, the container uses the measured or explicit size of the largest child cell for the size of each child cell. **Percentage based children:** The percentage-based sizes of a child component specify a percentage of the individual cell, not of the Tile container.
Navigators: ViewStack, Accordion, TabNavigator	**General layout:** The container displays one child at a time. **Default sizing:** The container size is determined by the measured or explicit dimensions of the initially selected child, and thereafter, all children are forced to be that initial size. If you set the `resizeToChild` property to `true` the container resizes to accommodate the measured or explicit size of each child, as that child appears. **Percentage based children:** As a general rule, you either use 100% for both height and width, which causes the children to fill the navigator bounds, or do not use percentage-based sizing.

Basic layout rules and considerations

Flex performs layout according to the following basic rules. If you remember these rules, you should be able to easily understand the details of Flex layout. These rules should help you determine why Flex lays out your application as it does and to determine how to modify your application appearance.

For a detailed description of how Flex sizes components, see "Determining and controlling component sizes" on page 151. For detailed information on component positioning, see "Positioning and laying out controls" on page 163.

- Flex first determines all components' measured (default) or explicitly set sizes *up*, from the innermost child controls to the outermost (Application) control. This is done in the measurement pass.
- After the measurement pass, Flex determines all percentage-based sizes and lays out components *down*, from the outermost container to the innermost controls. This is done in the layout pass.
- Sizes that you set to a pixel value are mandatory and fixed, and override any maximum or minimum size specifications that you set for the component.
- The default sizes determined in the measurement pass specify the sizes of components that do not have explicit or percentage-based sizes (or use constraint-based layout), and are fixed.
- Percentage-based size specifications are advisory. The layout algorithms satisfy the request if possible, and use the percentage values to determine proportional sizes, but the actual sizes can be less than the requested sizes. Percentage-based sizes are always within the component's maximum and minimum sizes, and, subject to those bounds, don't cause a container's children to exceed the container size.

Manually forcing layout

Sometimes, you must programmatically cause Flex to lay out components again. Flex normally delays processing properties that require substantial computation until the script that causes them to be set finishes executing. For example, setting the `width` property is delayed, because it may require recalculating the widths of the object's children or its parent. Delaying processing prevents it from being repeated multiple times if the script sets the object's `width` property more than once. However, in some situations, you might have to force the layout before the script completes.

Situations where you must force a layout include the following circumstances:

- When printing multiple page data grids by using the PrintDataGrid class.
- Before playing an effect, if the start values have just been set on the target.
- When capturing bitmap data after making property changes.

To force a layout, call the `validateNow()` method of the component that needs to be laid out. This method causes Flex to validate and update the properties, sizes, and layout of the object and all its children, and to redraw them, if necessary. Because this method is computation-intensive, you should be careful to call it only when it is necessary.

For an example of using the `validateNow()` method, see "Updating the PrintDataGrid layout" on page 816.

Sizing components

Flex provides several ways for controlling the size of components. You can do the following:

- Have Flex automatically determine and use default component sizes.
- Specify pixel sizes.
- Specify component size as a percentage of the parent container.
- Combine layout and sizing by specifying a constraint-based layout.

The following sections describe the basic sizing properties, provide details on how Flex determines component sizes, describe how to use automatic, explicit, and percentage-based sizing, and describe various techniques for controlling component size. For information on constraint-based layout, see "Using constraints to control component layout" on page 167.

Flex sizing properties

Several Flex properties affect the size of components. As a general rule, you use only a few properties for most applications, but a more complete understanding of these properties can help you understand the underlying Flex sizing mechanism and how Flex sizing properties interrelate. For information on the rules that Flex applies to determine the sizes of components based on the properties, see "Determining and controlling component sizes" on page 151.

Commonly used sizing properties

If you are not creating custom components, you typically use the following basic properties to specify how a component is sized:

- The `height`, `width`, `percentHeight`, and `percentWidth` properties specify the height and width of a component. In MXML tags, you use the `height` and `width` properties to specify the dimensions in pixels or as percentages of the parent container size. In ActionScript, you use the `height` and `width` properties to specify the dimensions in pixels, and use the `percentHeight` and `percentWidth` properties to specify the dimensions as a percentage of the parent container.
- The `minHeight`, `minWidth`, `maxHeight`, and `maxWidth` properties specify the minimum and maximum dimensions that a component can have if Flex determines the component size. These properties have no effect if you explicitly set the width or height in pixels.

The following tables include some properties that are only used by developers of custom components, particularly, those who must implement a custom `measure()` method.

Basic sizing characteristics and properties

The following characteristics and their associated properties determine the size of the component:

Characteristic	Associated properties	Description
Actual dimensions	Returned by the `height` and `width` properties.	The height and width of the displayed control, in pixels, as determined by the layout phase. If you set any explicit values, they determine the corresponding actual values.

Characteristic	Associated properties	Description
Explicit dimensions	`explicitHeight`, `explicitWidth` Setting the `height` and `width` properties to integer values also sets the `explicitHeight` and `explicitWidth` properties.	A dimension that you specifically set as a number of pixels. These dimensions cannot be overridden. Application developers typically use the `height` and `width` properties to set explicit dimensions. You cannot have both an explicit dimension and a percentage-based dimension, setting one unsets the other.
Percentage-based dimensions	`percentHeight`, `percentWidth` In MXML tags only, setting the `height` and `width` properties to percentage string values, such as "50%" also sets the `percentHeight` and `percentWidth` properties.	A dimension that you specifically set as a number in the range 0-100, as a percentage of the viewable area of the parent container. If you set a percentage-based dimension, the component is resizable, and grows or shrinks if the parent dimension changes.
Default dimensions	`measuredHeight`, `measuredWidth`	Not used directly by application developers. The dimensions of the component, as determined by the `measure()` method of the component. These values cannot be outside the range determined by the component's maximum and minimum height and width values. For more information on maximum and minimum default sizes, see Maximum and minimum dimensions.

Maximum and minimum dimensions

The following characteristics determine the minimum and maximum *default* and *percentage-based* dimensions that a component can have. They *do not* affect values that you set explicitly or dimensions determined by using constraint-based layout.

Characteristic	Associated Properties	Description
Minimum dimensions	`minHeight`, `minWidth` Setting the explicit minimum dimensions also sets the. `minHeight` and `minWidth` properties.	The minimum dimensions a component can have. By default, Flex sets these dimensions to the values of the minimum default dimensions.
Maximum dimensions	`maxHeight`, `maxWidth` Setting the explicit maximum dimensions also sets the `maxHeight` and `maxWidth` properties.	The maximum dimensions a component can have. The default values of these properties are component-specific, but often are 10000 pixels.
Minimum default dimensions	`measuredMinHeight`, `measuredMinWidth`	Not used by application developers. The minimum valid dimensions, as determined by the `measure()` method. The default values for these properties are component-specific; for many controls, the default values are 0.

Determining and controlling component sizes

Flex determines the sizes of controls and containers based on the components and their properties and how you can use Flex properties to control the sizes.

Note: *For a summary of the basic rules for component sizing, see* *"Basic layout rules and considerations" on page 149**.*

Basic sizing property rules

The following rules describe how you can use Flex sizing properties to specify the size of a component:

- Any dimension property that you set overrides the corresponding default value; for example, an explicitly set `height` property overrides any default height.
- Setting the `width`, `height`, `maxWidth`, `maxHeight`, `minWidth`, or `minHeight` property to a pixel value in MXML or ActionScript also sets the corresponding explicit property, such as `explicitHeight` or `explicitMinHeight`.
- The explicit height and width and the percentage-based height and width are mutually exclusive. Setting one value sets the other to `NaN`; for example, if you set `height` or `explicitHeight` to 50 and then set `percentHeight` to 33, the value of the `explicitHeight` property is `NaN`, not 50, and the `height` property returns a value that is determined by the `percentHeight` setting.
- If you set the `height` or `width` property to a percentage value in an MXML tag, you actually set the percentage-based value, that is, the `percentHeight` or `percentWidth` property, *not* the explicit value. In ActionScript, you cannot set the `height` or `width` property to a percentage value; instead, you must set the `percentHeight` or `percentWidth` property.
- When you get the `height` and `width` properties, the value is always the actual height or width of the control.

Determining component size

During the measurement pass, Flex determines the components' default (also called measured) sizes. During the layout pass, Flex determines the actual sizes of the components, based on the explicit or default sizes and any percentage-based size specifications.

The following list describes sizing rules and behaviors that apply to all components, including both controls and containers. For container-specific sizing rules, see "Determining container size" on page 153. For detailed information on percentage-based sizing, see "Using percentage-based sizing" on page 156.

- If you specify an explicit size for any component (that is not outside the component's minimum or maximum bounds), Flex always uses that size.
- If you specify a percentage-based size for any component, Flex determines the component's actual size as part of the parent container's sizing procedure, based on the parent's size, the component's requested percentage, and the container-specific sizing and layout rules.
- The default and percentage-based sizes are always at least as large as any minimum size specifications.
- If you specify a component size by using a percentage value and do not specify an explicit or percentage-based size for its container, the component size is the default size. Flex ignores the percentage specification. (Otherwise, an infinite recursion might result.)
- If a child or set of children require more space than is available in the parent container, the parent clips the children at the parent's boundaries, and, by default, displays scroll bars on the container so users can scroll to the clipped content. Set the `clipContent` property to `false` to configure a parent to let the child extend past the parent's boundaries. Use the `scrollPolicy` property to control the display of the scroll bars.
- If you specify a percentage-based size for a component, Flex uses the viewable area of the container in determining the sizes.
- When sizing and positioning components, Flex does not distinguish between visible and invisible components. By default, an invisible component is sized and positioned as if it were visible. To prevent Flex from considering an invisible component when it sizes and positions other components, set the component's `includeInLayout` property to `false`. This property affects the layout of the children of all containers except Accordion, FormItem, or ViewStack. For information on using the "Preventing layout of hidden controls" on page 165.

Note: *Setting a component's* `includeInLayout` *property to* `false` *does not prevent Flex from laying out or displaying the component; it only prevents Flex from considering the component when it lays out other components. As a result, the next component or components in the display list overlap the component. To prevent Flex from displaying the component, also set the visible property to* `false`*.*

Determining container size

Flex uses the following basic rules, in addition to the basic component sizing rules, to determine the size of a container:

- Flex determines all components' default dimensions during the measurement pass, and uses these values when it calculates container size during the layout pass.
- If you specify an explicit size for a container, Flex always uses that size, as with any component.
- If you specify a percentage-based size for a container, Flex determines the container's actual size as part of the parent container's sizing procedure, as with any component.
- A percentage-based container size is advisory. Flex makes the container large enough to fit its children at their minimum sizes. For more information on percentage-based sizing, see "Using percentage-based sizing" on page 156.
- If you do not specify an explicit or percentage-based size for a container, Flex determines the container size by using explicit sizes that you specify for any of its children, and the default sizes for all other children.
- Flex does not consider any percentage-based settings of a container's children when sizing the container; instead, it uses the child's default size.
- If a container uses automatic scroll bars, Flex does not consider the size of the scroll bars when it determines the container's default size in its measurement pass. Thus, if a scroll bar is required, a default-sized container might be too small for proper appearance.

Each container has a set of rules that determines the container's default size. For information on default sizes of each control and container, see the specific container sections in the *Adobe Flex Language Reference*, and in "Using the Application Container" on page 355; "Using Layout Containers" on page 371; and "Using Navigator Containers" on page 419.

Example: Determining an HBox container and child sizes

The following example code shows how Flex determines the sizes of an HBox container and its children. In this example, the width of the HBox container is the sum of the default width of the first and third buttons, the minimum width of the second button (because the default width would be smaller), and 16 for the two gaps. The default width for buttons is based on the label text width; in this example it is 66 pixels for all three buttons. The HBox width, therefore, is 66 + 70 + 66 + 16 = 218. If you change the `minWidth` property of the second button to 50, the calculation uses the button's default width, 66, so the HBox width is 214.

When Flex lays out the application, it sets the first and third button widths to the default values, 66, and the second button size to the minimum width, 70. It ignores the percentage-based specifications when calculating the final layout.

```
<?xml version="1.0"?>
<!-- components\HBoxSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:HBox id="hb1">
        <mx:Button id="b1"
            label="Label 1"
            width="50%"/>
        <mx:Button id="b2"
            label="Label 2"
            width="40%"
            minWidth="70"/>
        <mx:Button id="b3"
```

```
            label="Label 3"/>
    </mx:HBox>

    <mx:Form>
        <mx:FormItem label="HBox:">
            <mx:Label text="{hb1.width}"/>
        </mx:FormItem>
        <mx:FormItem label="Button #1:">
            <mx:Label text="{b1.width}"/>
        </mx:FormItem>
        <mx:FormItem label="Button #2:">
            <mx:Label text="{b2.width}"/>
        </mx:FormItem>
        <mx:FormItem label="Button #3">
            <mx:Label text="{b3.width}"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

In the following example, the HBox width now is 276 pixels, 50% of 552 pixels, where 552 is the Application container width of 600 minus 48 pixels for the 24-pixel left and right container padding. The button sizes are 106, 85, and 66 pixels respectively. The third button uses the default size. The variable width button sizes are five-ninths and four-ninths of the remaining available space after deducting the default-width button and the gaps, and the 1-pixel-wide border.

If you set the HBox `width` property to 20%, however, the HBox width is *not* 120 pixels, 20% of the Application container width, because this value is too small to fit the HBox container's children. Instead it is 200, the sum of 66 pixels (the default size) for buttons 1 and 3, 50 pixels (the specified minimum size) for button 2, 16 pixels for the gaps between buttons, and 2 pixels for the border. The buttons are 66, 50, and 66 pixels wide, respectively.

```
<?xml version="1.0"?>
<!-- components\HBoxSizePercent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600">
    <mx:HBox id="hb1" width="50%" borderStyle="solid">
        <mx:Button id="b1"
            label="Label 1"
            width="50%"/>
        <mx:Button id="b2"
            label="Label 2"
            width="40%"
            minWidth="50"/>
        <mx:Button id="b3"
            label="Label 3"/>
    </mx:HBox>

    <mx:Form>
        <mx:FormItem label="HBox:">
            <mx:Label text="{hb1.width}"/>
        </mx:FormItem>
        <mx:FormItem label="Button #1:">
            <mx:Label text="{b1.width}"/>
        </mx:FormItem>
        <mx:FormItem label="Button #2:">
            <mx:Label text="{b2.width}"/>
        </mx:FormItem>
        <mx:FormItem label="Button #3">
            <mx:Label text="{b3.width}"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

For more information and examples showing sizing of containers and children, see "Using Flex component sizing techniques" on page 155. For detailed information on percentage-based sizing, see "Using percentage-based sizing" on page 156.

Using Flex component sizing techniques

You can use default sizing, explicit sizing, and percentage-based sizing techniques to control the size of components. For information on using constraint-based layout for component sizing, see "Using constraints to control component layout" on page 167.

Using default sizing

If you do not otherwise specify sizes, the component's `measure()` method calculates a size based on the default sizing characteristics of the particular component and the default or explicit sizes of the component's child controls.

As a general rule, you should determine whether a component's default size (as listed for the component in the *Adobe Flex Language Reference*) is appropriate for your application. If it is, you do not have to specify an explicit or percentage-based size.

The following example shows how you can use default sizing for Button children of an HBox container. In this example, none of the children of the HBox container specify a width value:

```
<?xml version="1.0"?>
<!-- components\DefaultButtonSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="400" borderStyle="solid">
        <mx:Button label="Label 1"/>
        <mx:Button label="Label 2"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

Flex, therefore, uses the default sizes of the buttons, which accommodate the button label and default padding, and draws this application as the following image shows:

Notice the empty space to the right of the third button, because the sum of the default sizes is less than the available space.

Specifying an explicit size

You use the `width` and `height` properties of a component to explicitly set its size, as follows:

```
<?xml version="1.0"?>
<!-- components\ExplicitTextSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox id="myHBox">
        <mx:TextInput id="myInput"
            text="This TextInput control is 200 by 40 pixels."
            width="200"
            height="40"/>
    </mx:HBox>
</mx:Application>
```

In this example, Flex sets the component sizes to 200 by 40 pixels.

The following example shows setting the sizes of a container and its child:

```
<?xml version="1.0"?>
<!-- components\ExplicitHBoxSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox id="myHBox"
        width="150"
        height="150"
        borderStyle="solid" paddingLeft="5" paddingTop="5" paddingRight="5"
    >
        <mx:TextInput id="myInput"
            text="Enter the zip code"
            width="200"
            height="40"
        />
    </mx:HBox>
</mx:Application>
```

Because the specified TextInput control size is larger than that of its parent HBox container, Flex clips the TextInput control at the container boundary and displays a scroll bar (if you do not disable it) so that you can scroll the container to the clipped content. For more information on scroll bar sizing considerations, see "Dealing with components that exceed their container size" on page 160.

Using percentage-based sizing

Percentage-based sizing dynamically determines and maintains a component's size relative to its container; for example, you can specify that the component's width is 75% of the container. This sizing technique has several advantages over default or explicit fixed sizing:

- You only have to specify a size relative to the container; you don't have to determine exact measurements.
- The component size changes dynamically when the container size changes.
- The sizing mechanism automatically takes into account the remaining available space and fits components even if their requested size exceeds the space.

To specify a percentage value, use one of the following coding techniques:

1 In an MXML tag, set the `height` or `width` property to a percentage value; for example:

```
<mx:TextArea id="ta1" width="70%" height="40%"/>
```

2 In an MXML tag or an ActionScript statement, set the `percentHeight` or `percentWidth` property to a numeric value; for example:

```
ta1.percentWidth=70;
```

The exact techniques Flex uses to determine the dimensions of a component that uses percentage-based sizing depend on the type of container that holds the container. For example, a Tile container has cells that are all the largest default or explicit dimensions of the largest child. Child control percentage values specify a percentage of the tile cell size, not of the Tile control size. The percentage sizes of the Box, HBox, and VBox containers, on the other hand, are relative to the container size.

Sizing percentage-based children of a linear container with automatic positioning

When Flex sizes children of a container that uses automatic positioning to lay out children in a single direction, such as a HBox or VBox container, Flex does the following:

1 Determines the size of the viewable area of the parent container, and uses the corresponding dimensions as the container dimensions for sizing calculations. The viewable area is the part of the component that is being displayed and can contain child controls, text, images, or other contents. For more information on calculating the size of containers, see "Determining component size" on page 152.

2 Determines the desired sizes of children with percentage-based sizes by multiplying the decimal value by the size of the viewable area of the container, minus any padding and inter-child gaps.

3 Reserves space for all children with explicit or default sizes.

4 If available space (parent container size minus all reserved space, including borders, padding, and gaps) cannot accommodate the percentage requests, divides the available space in proportion to the specified percentages.

5 If a minimum or maximum height or width specification conflicts with a calculated value, uses the minimum or maximum value, and recalculates all other percentage-based components based on the reduced available space.

6 Rounds the size down to the next integer.

The following examples show how the requested percentage can differ from the size when the component is laid out:

- Suppose that 50% of a HBox parent is available after reserving space for all explicit-sized and default-sized components, and for all gaps and padding. If one component requests 20% of the parent, and another component requests 60%, the first component is sized to 12.5% ((20 / 20+ 60) * 50%) of the parent container, the second component is sized to 37.5% of the parent container.
- If any component, for example, a Tile container, requests 100% of its parent Application container's space, it occupies all of the container *except* for the Application's 24-pixel-wide top, bottom, left, and right padding, unless you explicitly change the padding settings of the Application container.

Sizing percentage-based children of a container with absolute positioning

When Flex sizes children of a container that uses absolute positioning, it does the following:

1 Determines the viewable area of the parent container, and uses the corresponding dimensions as the container dimensions for sizing calculations. For more information on calculating the size of containers, see "Determining component size" on page 152.

2 Determines the sizes of children with percentage-based sizes by multiplying the decimal value by the container dimension minus the position of the control in the dimension's direction. For example, if you specify `x="10"` and `width="100%"` for a child, the child size extends only to the edge of the viewable area, not beyond.

Because controls can overlay other controls or padding, the sizing calculations do not consider padding or any other children when determining the size of a child.

3 If a minimum or maximum height or width specification conflicts with a calculated value, uses the minimum or maximum value.

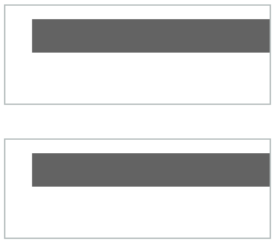
4 Rounds the size down to the next integer.

The following code shows the percentage-based sizing behavior with absolute positioning:

```
<?xml version="1.0"?>
<!-- components\PercentSizeAbsPosit.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundGradientColors="[#FFFFFF, #FFFFFF]"
    verticalGap="25">

    <mx:Canvas
        width="200" height="75"
        borderStyle="solid">

        <mx:HBox
            x="20" y="10"
            width="100%" height="25"
            backgroundColor="#666666"/>
    </mx:Canvas>

    <mx:Canvas
        width="200" height="75"
```

```
        borderStyle="solid">

        <mx:HBox
            left="20" top="10"
            width="100%" height="25"
            backgroundColor="#666666"/>
    </mx:Canvas>
</mx:Application>
```

Flex draws the following application:



Examples: Using percentage-based children of an HBox container

The following example specifies percentage-based sizes for the first two of three buttons in an HBox container:

```
<?xml version="1.0"?>
<!-- components\PercentHBoxChildren.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="400">
        <mx:Button label="Label 1" width="25%"/>
        <mx:Button label="Label 2" width="40%"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

In this example, the default width of the third button is 66 pixels. The HBox container has no padding by default, but it does put a 8-pixel horizontal gap between each component. Because this application has three components, these gaps use 16 pixels, so the available space is 384. The first button requests 25% of the available space, or 96 pixels. The second button requests 40% of 384 pixels, rounded down to 153 pixels. There is still unused space to the right of the third button.

Flex draws the following application:

Now change the percentage values requested to 50% and 40%, respectively:

```
<?xml version="1.0"?>
<!-- components\PercentHBoxChildren5040.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="400">
        <mx:Button label="Label 1"
            width="50%"/>
        <mx:Button label="Label 2"
            width="40%"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

In this example, the first button requests 50% of the available HBox space, or 192 pixels. The second button still requests 40%, or 153 pixels, for a total of 345 pixels. However, the HBox only has 318 pixels free after reserving 66 pixels for the default-width button and 16 pixels for the gaps between components. Flex divides the available space proportionally between the two buttons, giving .5/(.5 + .4) * 318 = 176 pixels, to the first button and .4/(.5 + .4) * 318 = 141 pixels, to the second button. (All calculated values are rounded down to the nearest pixel.)

Flex draws the following application:

Using minimum or maximum dimensions

You can also use the `minWidth`, `minHeight`, `maxWidth`, and `maxHeight` properties with a percentage-based component to constrain its size. Consider the following example:

```
<?xml version="1.0"?>
<!-- components\PercentHBoxChildrenMin.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="400">
        <mx:Button label="Label 1"
            width="50%"/>
        <mx:Button label="Label 2"
            width="40%"
            minWidth="150"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

To determine the widths of the percentage-based button sizes, Flex first determines the sizes as described in the second example in "Examples: Using percentage-based children of an HBox container" on page 158, which results in requested values of 176 for the first button and 141 for the second button. However, the minimum width of the second button is 150, so Flex sets its size to 150 pixels, and reduces the size of the first button to occupy the remaining available space, which results in a width of 168 pixels.

Flex draws the following application:

Sizing containers and components toolbox

You can control sizing, including setting the Application container size, handling components that exceed the container size, and using padding and custom gaps.

Setting the Application container size

When you size an application, you often start by setting the size of the Application container. The Application container determines the boundaries of your application in the Adobe® Flash® Player or Adobe® AIR™.

If you are using Adobe® Flex™ Builder™, or are compiling your MXML application on the server, an HTML wrapper page is generated automatically. The `width` and `height` properties specified in the `<mx:Application>` tag are used to set the width and height of the `<object>` and `<embed>` tags in the HTML wrapper page. Those numbers determine the portion of the HTML page that is allocated to the Adobe Flash plug-in.

If you are not autogenerating the HTML wrapper, set the `<mx:Application>` tag's `width` and `height` properties to 100%. That way, the Flex application scales to fit the space that is allocated to the Flash plug-in.

You set the Application container size by using the `<mx:Application>` tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\AppExplicit.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    height="100"
    width="150"
>

        <!-- Application children go here. -->
</mx:Application>
```

In this example, you set the Application container size to 100 by 150 pixels. Anything in the application larger than this window is clipped at the window boundaries. Therefore, if you define a 200 pixel by 200 pixel DataGrid control, it is clipped, and the Application container displays scroll bars. (You can disable, or always display, scroll bars by setting the container's `horizontalScrollPolicy` and `verticalScrollPolicy` properties.)

For more information on sizing the Application container, see "Using the Application Container" on page 355.

Dealing with components that exceed their container size

If the sum of the actual sizes of a container's children, plus the gaps and padding, exceed the dimensions of the container, by default, the container's contents are clipped at the container boundaries, and Flex displays scroll bars on the container so you can scroll to the remaining content. If you set the `horizontalScrollPolicy` and `verticalScrollPolicy` properties to `ScrollPolicy.OFF`, the scroll bars do not appear, but users do not have access to the clipped contents. If you set the `clipContent` property to `false`, container content can extend beyond the container boundary.

Using Scroll bars

If Flex cannot fit all of the components into the container, it uses scroll bars, unless you disable them by setting the `horizontalScrollPolicy` or `verticalScrollPolicy` property to `ScrollPolicy.OFF` or by setting `clipContent` to `false`. Consider the following example:

```
<?xml version="1.0"?>
<!-- components\ScrollHBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="400">
        <mx:Button label="Label 1"
            width="50%"
            minWidth="200"/>
        <mx:Button label="Label 2"
            width="40%"
            minWidth="150"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

In this example, the default width of the fixed-size button is 66 pixels, so there are 324 pixels of space available for the percentage-based buttons after accounting for the gap between components. The minimum widths of the first and second buttons are greater than the percentage-based values, so Flex assigns those buttons the set widths of 200 and 150 pixels, even though the HBox container only had 324 pixels free. The HBox container uses scroll bars to provide access to its contents because they now consume more space than the container itself.

Notice that the addition of the scroll bar doesn't increase the height of the container from its initial value. Flex considers scroll bars in its sizing calculations only if you explicitly set the scroll policy to `ScrollPolicy.ON`. So, if you use an auto scroll policy (the default), the scroll bar overlaps the buttons. To prevent this behavior, you can set the `height` property for the HBox container or allow the HBox container to resize by setting a percentage-based width. Remember that changing the height of the HBox container causes other components in your application to move and resize according to their own sizing rules. The following example adds an explicit height and permits you to see the buttons and the scroll bar:

```
<?xml version="1.0"?>
<!-- components\ScrollHBoxExplicitHeight.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox width="400" height="42">
        <mx:Button label="Label 1"
            width="50%"
            minWidth="200"/>
        <mx:Button label="Label 2"
            width="40%"
            minWidth="150"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

Flex draws the following application:

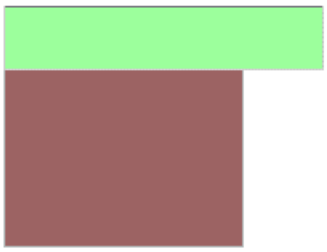
Alternately, you can set the HBox control's `horizontalScrollPolicy` property to `ScrollPolicy.ON`. This reserves space for the scroll bar during the initial layout pass, so it fits without overlapping the buttons or setting an explicit height. This also correctly handles the situation where the scroll bars change their size when you change skinning or styles. This technique places an empty scroll bar area on the container if it does not need scrolling, however.

Using the clipContent property

If you set the `clipContent` property for the parent container to `false`, the content can extend beyond the container's boundaries and no scroll bars appear, as the following example shows:

```
<?xml version="1.0"?>
<!-- components\ClipHBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="600"
    height="400"
    backgroundGradientColors="[#FFFFFF, #FFFFFF]">

    <mx:HBox id="myHBox"
        width="150"
        height="150"
        borderStyle="solid"
        backgroundColor="#996666"
        clipContent="false">

        <mx:TextInput id="myInput"
            width="200" height="40"
            backgroundColor="#99FF99"/>
    </mx:HBox>
</mx:Application>
```

The following image shows the application, with the TextInput control extending past the right edge of the HBox control:



To ensure that components fit in the container, reduce the sizes of the child components. You can do this by setting explicit sizes that fit in the container, or by specifying percentage-based sizes. If you set percentage-based sizes, Flex shrinks the children to fit the space, or their minimum sizes, whichever is larger. By default, Flex sets the minimum height and width of most components to 0. You can set these components' minimum properties to nonzero values to ensure that they remain readable.

Using padding and custom gaps

There may be situations where you want your containers to have padding around the edges. (Previous Flex releases used the term margins; Flex 2 uses the term *padding* for consistency with cascading style sheet conventions.) Some containers, such as the Application container, have padding by default; others, such as the HBox container, have padding values of 0 by default. Also, some containers have gaps between children, which you might want to change from the default values. If your application has nonzero padding and gaps, Flex reserves the necessary pixels before it sizes any percentage-based components. Consider the following example:

```
<?xml version="1.0"?>
<!-- components\PadHBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:HBox
        width="400"
        borderStyle="solid"
        paddingLeft="5"
        paddingRight="5"
        horizontalGap="5">

        <mx:Button label="Label 1"
            width="50%"/>
        <mx:Button label="Label 2"
            width="40%"
            minWidth="150"/>
        <mx:Button label="Label 3"/>
    </mx:HBox>
</mx:Application>
```

The default width of the fixed-size button is 66 pixels. All horizontal padding and gaps in the HBox control are 5 pixels wide, so the Flex application reserves 5 pixels for the left padding, 5 pixels for the right padding, and 10 pixels total for the two gaps between components, which leaves 314 pixels free for the two percentage-based components. Flex reserves 66 pixels for the default-sized (third) button; the second button requires its minimum size, 150 pixels; and the padding and gap take 20 pixels; this leaves 164 pixels available for the first button. The first button requests 200 pixels; therefore, it uses all available pixels and is 164 pixels wide.

Flex draws the following application:

Positioning and laying out controls

By default, Flex automatically positions all components, except for the children of a Canvas container. If you have a Canvas container, or an Application or Panel container with the `layout` property set to `absolute`, you specify absolute positions for its children, or use constraint-based layout. You can use automatic positions and absolute positioning by using `x` and `y` properties. For information on using constraint-based layout, which can control both positioning and sizing, see "Using constraints to control component layout" on page 167.

Using automatic positioning

For most containers, Flex automatically positions the container children according to the container's layout rules, such as the layout direction, the container padding, and the gaps between children of that container.

For containers that use automatic positioning, setting the `x` or `y` property directly or calling `move()` has no effect, or only a temporary effect, because the layout calculations set the *x* position to the calculation result, not the specified value. You can, however, specify absolute positions for the children of these containers under some circumstances; for more information, see "Disabling automatic positioning temporarily" on page 164.

You can control aspects of the layout by specifying container properties; for details on the properties, see the property descriptions for the container in the *Adobe Flex Language Reference*. You also control the layout by controlling component sizes and by using techniques such as adding spacers.

Using the Spacer control to control layout

Flex includes a Spacer control that helps you lay out children within a parent container. The Spacer control is invisible, but it does allocate space within its parent.

In the following example, you use a percentage-based Spacer control to push the Button control to the right so that it is aligned with the right edge of the HBox container:

```
<?xml version="1.0"?>
<!-- components\SpacerHBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            [Embed(source="assets/flexlogo.jpg")]
            [Bindable]
            public var imgCls:Class;
        ]]>
    </mx:Script>

    <mx:HBox width="400">
        <mx:Image source="{imgCls}"/>
        <mx:Label text="Company XYZ"/>
        <mx:Spacer width="100%"/>
        <mx:Button label="Close"/>
    </mx:HBox>
</mx:Application>
```

In this example, the Spacer control is the only percentage-based component in the HBox container. Flex sizes the Spacer control to occupy all available space in the HBox container that is not required for other components. By expanding the Spacer control, Flex pushes the Button control to the right edge of the container.

You can use all sizing and positioning properties with the Spacer control, such as `width`, `height`, `maxWidth`, `maxHeight`, `minWidth`, and `minHeight`.

Disabling automatic positioning temporarily

You can use effects, such as the Move and Zoom effects, to modify the size or position of a child in response to a user action. For example, you might define a child so that when the user selects it, the child moves to the top of the container and doubles in size. These effects modify the `x` and `y` properties of the child as part of the effect. Similarly, you might want to change the position of a control by changing its *x* or *y* coordinate value, for example, in response to a button click.

Containers that use automatic positioning ignore the values of the `x` and `y` properties of their children during a layout update. Therefore, the layout update cancels any modifications to the `x` and `y` properties performed by the effect, and the child does not remain in its new location.

You can prevent Flex from performing automatic positioning updates that conflict with the requested action of your application by setting the `autoLayout` property of a container to `false`. Setting this property to `false` prevents Flex from laying out the container's contents when a child moves or resizes. Flex defines the `autoLayout` property in the Container class, and all containers inherit it; its default value is `true`, which enables Flex to update layouts.

Even when you set the `autoLayout` property of a container to `false`, Flex updates the layout when you add or remove a child. Application initialization, deferred instantiation, and the `<mx:Repeater>` tag add or remove children, so layout updates always occur during these processes, regardless of the value of the `autoLayout` property. Therefore, during container initialization, Flex defines the initial layout of the container children regardless of the value of the `autoLayout` property.

The following example disables layout updates for a VBox container:

```
<?xml version="1.0"?>
<!-- components\DisableVBoxLayout.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox autoLayout="false"
        width="200"
        height="200"
    >
        <mx:Button id="b1"
            label="Button 1"
        />
        <mx:Button id="b2"
            label="Button 2"
            click="b2.x += 10;"
        />
        <mx:Button id="b3"
            label="Button 3"
            creationComplete="b3.x = 100; b3.y = 75;"
        />
    </mx:VBox>
</mx:Application>
```

In this example, Flex initially lays out all three Button controls according to the rules of the VBox container. The creationComplete event listener for the third button is dispatched after the VBox control has laid out its children, but before Flex displays the buttons. Therefore, when the third button appears, it is at the *x* and *y* positions specified by the creationComplete listener. After the buttons appear, Flex shifts the second button 10 pixels to the right each time a user clicks it.

Setting the `autoLayout` property of a container to `false` prohibits Flex from updating a container's layout after a child moves or resizes, so you should set it to `false` only when absolutely necessary. You should always test your application with the `autoLayout` property set to the default value of `true`, and set it to `false` only as necessary for the specific container and specific actions of the children in that container.

For more information on effects, see "Using Behaviors" on page 431.

Preventing layout of hidden controls

By default, Flex lays out and reserves space for all components, including hidden components, but it does not display the hidden controls. You see blank spots where the hidden controls will appear when you make them visible. In place of the hidden controls, you see their container's background. However if the container is any of the following components, you can prevent Flex from considering the child component when it lays out the container's other children by setting the child component's `includeInLayout` property of the component to `false`:

- Box, or any of its subclasses: HBox, VBox, DividedBox, HDividedBox, VdividedBox, Grid, GridItem, GridRow, ControlBar, and ApplicationControlBar,
- Form
- Tile and its subclass, Legend
- ToolBar

When a component's `includeInLayout` property is `false`, Flex does not include it in the layout calculations for other components, but still lays it out. In other words, Flex does not reserve space for the component, but still draws it. As a result, the component can appear underneath the components that follow it in the layout order. To prevent Flex from drawing the component, you must also set its `visible` property to `false`.

The following example shows the effects of the `includeInLayout` and `visible` properties. It lets you toggle each of these properties independently on the middle of three Panel controls in a VBox control.

```
<?xml version="1.0"?>
<!-- components\HiddenBoxLayout.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:VBox>
        <mx:Panel id="p1"
            title="Panel 1"
            backgroundColor="#FF0000"/>
        <mx:Panel id="p2"
            title="Panel 2"
            backgroundColor="#00FF00"/>
        <mx:Panel id="p3"
            title="Panel 3"
            backgroundColor="#0000FF"/>
    </mx:VBox>

    <mx:HBox>
        <mx:Button label="Toggle Panel 2 Visible"
            click="{p2.visible=!p2.visible;}"/>
        <mx:Button label="Toggle Panel 2 in Layout"
            click="{p2.includeInLayout=!p2.includeInLayout;}"/>
    </mx:HBox>
</mx:Application>
```

Run this application and click the buttons to see the results of different combinations of `visible` and `includeInLayout` properties. The example shows the following behaviors:

- If you include the second Panel control in the layout and make it invisible, Flex reserves space for it; you see the background of its VBox container in its place.

- If you do not include the second Panel control in the layout, the VBox resizes and the HBox with the buttons moves up. If you then include it in the layout, the VBox resizes again, and the HBox and buttons move down.
- If you do not include the second Panel control in the layout and make it visible, Flex still draws it, but does not consider it in laying out the third Panel control, so the two panels overlap. Because the title of a Panel control has a default alpha of 0.5, you see the combination of the second and third Panel controls in the second Panel position.

Using absolute positioning

Three containers support absolute positioning:

- Application and Panel controls use absolute positioning if you specify the `layout` property as `"absolute"` (`ContainerLayout.ABSOLUTE`).
- The Canvas container always uses absolute positioning.

With absolute positioning, you specify the child control position by using its `x` and `y` properties, or you specify a constraint-based layout; otherwise, Flex places the child at position 0,0 of the parent container. When you specify the *x* and *y* coordinates, Flex repositions the controls only when you change the property values. The following example uses absolute positioning to place a VBox control inside a Canvas control:

```
<?xml version="1.0"?>
<!-- components\CanvasLayout.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundGradientColors="[#FFFFFF, #FFFFFF]">

    <mx:Canvas
        width="100" height="100"
        backgroundColor="#999999">

        <mx:VBox id="b1"
            width="80" height="80"
            x="20" y="20"
            backgroundColor="#A9C0E7">
        </mx:VBox>
    </mx:Canvas>
</mx:Application>
```

This example produces the following image:

When you use absolute positioning, you have full control over the locations of the container's children. This lets you overlap components. The following example adds a second VBox to the previous example so that it partially overlaps the initial box.

```
<?xml version="1.0"?>
<!-- components\CanvasLayoutOverlap.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundGradientColors="[#FFFFFF, #FFFFFF]">

    <mx:Canvas
        width="100" height="100"
        backgroundColor="#999999">

        <mx:VBox id="b1"
```

```
            width="80" height="80"
            x="20" y="20"
            backgroundColor="#A9C0E7">
        </mx:VBox>

        <mx:VBox id="b2"
            width="50" height="50"
            x="0" y="50"
            backgroundColor="#FF0000">
        </mx:VBox>
    </mx:Canvas>
</mx:Application>
```

This example produces the following image:

Note: *If you use percentage-based sizing for the children of a control that uses absolute positioning, the percentage-based components resize when the parent container resizes, and the result may include unwanted overlapping of controls.*

Using constraints to control component layout

You can manage a child component's size and position simultaneously by using constraint-based layout, or by using constraint rows and columns. Constraint-based layout lets you anchor the sides or center of a component to positions relative to the viewable region of the component's container. The *viewable region* is the part of the component that is being displayed, and it can contain child controls, text, images, or other contents.

Constraint rows and columns let you subdivide a container into vertical and horizontal constraint regions to control the size and positioning of child components with respect to each other and within the parent container.

Creating a constraint-based layout

You can use constraint-based layout to determine the position and size of the immediate children of any container that supports absolute positioning. With constraint-based layout you can do the following:

- Anchor one or more edges of a component at a pixel offset from the corresponding edge of its container's viewable region. The anchored child edge stays at the same distance from the parent edge when the container resizes. If you anchor both edges in a dimension, such as top and bottom, the component resizes if the container resizes.
- Anchor the child's horizontal or vertical center (or both) at a pixel offset from the center of the container's viewable region. The child does not resize in the specified dimension unless you also use percentage-based sizing.
- Anchor the baseline of a component at a pixel offset from the top edge of its parent container.

You can specify a constraint-based layout for any Flex framework component (that is, any component that extends the UIComponent class). The following rules specify how to position and size components by using constraint-based layout:

- Place the component directly inside a Canvas container, or directly inside an Application or Panel container with the `layout` property set to `absolute`.
- Specify the constraints by using the `top`, `bottom`, `left`, `right`, `horizontalCenter`, or `verticalCenter` styles.

The `top`, `bottom`, `left`, and `right` styles specify the distances between the component sides and the corresponding container sides.

The `baseline` constraint specifies the distance between the baseline position of a component and the upper edge of its parent container. Every component calculates its baseline position as the y-coordinate of the baseline of the first line of text of the component. The baseline of a UIComponent object that does not contain any text is calculated as if the UIComponent object contained a UITextField object that uses the component's styles, and the top of the UITextField object coincides with the component's top.

The `horizontalCenter` and `verticalCenter` styles specify distance between the component's center point and the container's center, in the specified direction; a negative number moves the component left or up from the center.

The following example anchors the Form control's left and right sides 20 pixels from its container's sides:

```
<mx:Form id="myForm" left="20" right="20"/>
```

- Do not specify a `top` or `bottom` style with a `verticalCenter` style; the `verticalCenter` value overrides the other properties. Similarly, do not specify a `left` or `right` style with a `horizontalCenter` style.
- A size determined by constraint-based layout overrides any explicit or percentage-based size specifications. If you specify `left` and `right` constraints, for example, the resulting constraint-based width overrides any width set by a `width` or `percentWidth` property.

Precedence rules for constraint-based components

- If you specify a single edge constraint (`left, right, top,` or `bottom`) without any other sizing or positioning parameter, the component size is the default size and its position is determined by the constraint value. If you specify a size parameter (width or height), the size is determined by that parameter.
- If you specify a pair of constraints (`left-right` or `top-bottom`), the size and position of the component is determined by those constraint values. If you also specify a center constraint (`horizontalCenter` or `verticalCenter`), the size of the component is calculated from the edge constraints and its position is determined by the center constraint value.
- Component size determined by a pair of constraint-based layout properties (`left-right` or `top-bottom`) overrides any explicit or percentage-based size specifications. For example, if you specify both `left` and `right` constraints, the calculated constraint-based width overrides the width set by a `width` or `percentWidth` property.
- Edge constraints override `baseline` constraints.

Example: Using constraint-based layout for a form

The following example code shows how you can use constraint-based layout for a form. In this example, the Form control uses a constraint-based layout to position its top just inside the canvas padding. The form left and right edges are 20 pixels from the Canvas container's left and right edges. The HBox that contains the buttons uses a constraint-based layout to place itself 20 pixels from the Canvas right edge and 10 pixels from the Canvas bottom edge.

If you change the size of your browser, stand-alone Flash Player, or AIR application, you can see the effects of dynamically resizing the Application container on the Form layout. The form and the buttons overlap as the application grows smaller, for example. In general, you should include the buttons in the last FormItem of the form. However, in the following example, the buttons are separated in an HBox container to better show the effects of resizing.

```
<?xml version="1.0"?>
<!-- components\ConstraintLayout.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Use a Canvas container in the Application to prevent
        unnecessary scroll bars on the Application. -->
    <mx:Canvas width="100%" height="100%">
```

```
        <!-- Anchor the top of the form at the top of the canvas.
           Anchor the form sides 20 pixels from the canvas sides. -->
        <mx:Form id="myForm"
                backgroundColor="#DDDDDD"
                top="0"
                left="20"
                right="20">

            <mx:FormItem label="Product:" width="100%">
                <!-- Specify a fixed width to keep the ComboBox control from
                    resizing as you change the application size. -->
                <mx:ComboBox width="200"/>
            </mx:FormItem>

            <mx:FormItem label="User" width="100%">
                <mx:ComboBox width="200"/>
            </mx:FormItem>

            <mx:FormItem label="Date">
                <mx:DateField/>
            </mx:FormItem>

            <mx:FormItem width="100%"
                    direction="horizontal"
                    label="Hours:">
                <mx:TextInput width="75"/>
                <mx:Label text="Minutes" width="48"/>
                <mx:TextInput width="75"/>
            </mx:FormItem>
        </mx:Form>

        <!-- Anchor the box with the buttons 20 pixels from the canvas
            right edge and 10 pixels from the bottom. -->
        <mx:HBox id="okCancelBox"
                right="20"
                bottom="10">
            <mx:Button label="OK"/>
            <mx:Button label="Cancel"/>
        </mx:HBox>
    </mx:Canvas>
</mx:Application>
```

Using constraint rows and columns

You can subdivide a container that supports absolute positioning into vertical and horizontal constraint regions to control the size and positioning of child components with respect to each other, or with respect to the parent container.

You define the horizontal and vertical constraint regions of a container by using the `constraintRows` and `constraintColumns` properties. These properties contain Arrays of constraint objects that partition the container horizontally (ConstraintColumn objects) and vertically (ConstraintRow objects). ConstraintRow objects are laid out in the order they are defined, from top to bottom in their container; ConstraintColumn objects are laid out from left to right in the order they are defined.

The following example shows a `Canvas` container partitioned into two vertical regions and two horizontal regions. The first constraint column occupies 212 pixels from the leftmost edge of the `Canvas`. The second constraint column occupies 100% of the remaining Canvas width. The rows in this example occupy 80% and 20% of the Canvas container's height from top to bottom, respectively.

```
<?xml version="1.0"?>
```

```
<!-- constraints\BasicRowColumn.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Canvas>
        <mx:constraintColumns>
            <mx:ConstraintColumn id="col1" width="212"/>
            <mx:ConstraintColumn id="col2" width="100%"/>
        </mx:constraintColumns>
        <mx:constraintRows>
            <mx:ConstraintRow id="row1" height="80%"/>
            <mx:ConstraintRow id="row2" height="20%"/>
        </mx:constraintRows>

    <!-- Position child components based on
        the constraint columns and rows. -->

    </mx:Canvas>
</mx:Application>
```

Constraint rows and columns do not have to occupy 100% of the available area in a container. The following example shows a single constraint column that occupies 20% of the Canvas width; 80% of the container is unallocated.

```
<?xml version="1.0"?>
<!-- constraints\BasicColumn_20Percent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Canvas>
        <mx:constraintColumns>
            <mx:ConstraintColumn id="col1" width="20%"/>
        </mx:constraintColumns>
    </mx:Canvas>

    <!-- Position child components based on
        the constraint column. -->

</mx:Application>
```

Creating constraint rows and columns

Constraint columns and rows have three sizing options: fixed, percent, and content. These options dictate the amount of space that the constraint region occupies in the container. As child components are added to or removed from the parent container, the space allocated to each ConstraintColumn and ConstraintRow instance is computed according to its sizing option.

- *Fixed size* means the space allocated to the constraint region is a fixed pixel size. In the following example, you set the fixed with of a ConstraintColumn instance to 100 pixels:

  ```
  <mx:ConstraintColumn id="col1" width="100"/>
  ```

  As the parent container grows or shrinks, the ConstraintColumn instance remains 100 pixels wide.

- *Percent size* means that the space allocated to the constraint row or column is calculated as a percentage of the space remaining in the parent container after the space allocated to fixed and content size child objects has been deducted from the available space.

  In the following example, you set the with of a ConstraintColumn instance to 80%:

  ```
  <mx:ConstraintColumn  id="col1" width="80%"/>
  ```

  As the parent container grows or shrinks, the ConstraintColumn always takes up 80% of the available width.

A best practice in specifying percent constraints is to ensure that the sum of all percent constraints is less than or equal to 100%. However, if the total value of percent specifications is greater than 100%, the actual allocated percentages are calculated so that the proportional values for all constraints total 100%. For example, if the percentages for two constraint objects are specified as 100% and 50%, the values are adjusted to 66.6% and 33.3% (two-thirds for the first value and one-third for the second).

- *Content size* (default) means that the space allocated to the region is dictated by the size of the child objects in that space. As the size of the content changes, so does the size of the region. Content sizing is the default when you do not specify either fixed or percentage sizing parameters.

  In the following example, you specify content size by omitting any explicit width setting:

  ```
  <mx:ConstraintColumn  id="col1"/>
  ```

  The width of this ConstraintColumn is determined by the width of its largest child. When children span multiple content sized constraint rows or constraint columns, Flex divides the space consumed by the children among the rows and columns.

For the ConstraintColumn class, you can also use the `maxWidth` and `minWidth` properties to limit the width of the column. For the ConstraintRow class, you can use the `maxHeight` and `minHeight` properties to limit the height of the row. Minimum and maximum sizes for constraint columns and rows limit how much the constraint regions grow or shrink when you resize their parent containers. If the parent container with a constraint region shrinks to less than the minimum size for that region when you resize the container, scrollbars appear to show clipped content.

Note: *Minimum and maximum limits are only applicable to percentage and content sized constraint regions. For fixed size constraint regions, minimum and maximum values, if specified, are ignored.*

Positioning child components based on constraint rows and constraint columns

Anchor a child component to a constraint row or constraint column by prepending the constraint region's ID to any of the child's constraint parameters. For example, if the ID of a ConstraintColumn is "col1", you can specify a set of child constraints as `left="col1:10"`, `right="col1:30"`, `horizontalCenter="col1:0"`.

If you do not qualify constraint parameters (`left, right, top`, and `bottom`) a constraint region ID, the component is constrained relative to the edges of its parent container. Components can occupy a single constraint region (row or column) or can span multiple regions.

The following example uses constraint rows and constraint columns to position three Button controls in a Canvas container:

```
<?xml version="1.0"?>
<!-- constraints\ConstrainButtons.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Canvas id="myCanvas" backgroundColor="0x6699FF">

        <mx:constraintColumns>
            <mx:ConstraintColumn id="col1" width="100"/>
            <mx:ConstraintColumn id="col2" width="100"/>
        </mx:constraintColumns>
        <mx:constraintRows>
            <mx:ConstraintRow id="row1" height="100"/>
            <mx:ConstraintRow id="row2" height="100"/>
        </mx:constraintRows>

        <mx:Button label="Button 1"
            top="row1:10" bottom="row1:10"
            left="10"/>
        <mx:Button label="Button 2"
            left="col2:10" right="col2:10"/>
        <mx:Button label="Button 3"
```

```
            top="row2:10" bottom="row2:10"
            left="col1:10" right="10"/>
    </mx:Canvas>

    <mx:Label text="canvas width:{myCanvas.width}"/>
    <mx:Label text="canvas height:{myCanvas.height}"/>
</mx:Application>
```

The next example defines the constraint rows and columns by using percentages. As you resize the application, the Button controls resize accordingly.

```
<?xml version="1.0"?>
<!-- constraints\ConstrainButtonsPercent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Canvas id="myCanvas"
        backgroundColor="0x6699FF"
        width="100%" height="100%">

        <mx:constraintColumns>
            <mx:ConstraintColumn id="col1" width="30%"/>
            <mx:ConstraintColumn id="col2" width="40%"/>
            <mx:ConstraintColumn id="col3" width="30%"/>
        </mx:constraintColumns>
        <mx:constraintRows>
            <mx:ConstraintRow id="row1" height="35%"/>
            <mx:ConstraintRow id="row2" height="55%"/>
        </mx:constraintRows>

        <mx:Button label="Button 1"
            top="row1:10" bottom="row2:10"
            left="10"/>
        <mx:Button label="Button 2"
            left="col2:10" right="col3:10"/>
        <mx:Button label="Button 3"
            top="row2:10" bottom="row2:10"
            left="col3:10" right="col3:10"/>
    </mx:Canvas>

    <mx:Label text="canvas width:{myCanvas.width}"/>
    <mx:Label text="canvas height:{myCanvas.height}"/>
</mx:Application>
```

While you may specify any combination of qualified and unqualified constraints, some constraint properties may be overridden. The priority of sizing and positioning constraints are as follows:

1. Center constraint specifications override all other constraint values when determining the position of a control.
2. Next, left edge and top positions are determined.
3. Finally, right edge and bottom positions are calculated to best fit the component.

The following table defines the behavior of constrained components when they are contained in a single constraint region. *Edge 1* is the first specified edge constraint for a child component (`left`, `right`, `top`, or `bottom`). *Edge 2* is the second specified edge constraint of a pair (`left` and `right`, `top` and `bottom`). *Size* is an explicit size for a child component (`width`, `height`). *Center* is the positioning constraint to center the child object (`horizontalCenter` or `verticalCenter`).

Constraint Parameters				**Behavior**	
Edge 1	**Edge 2**	**Size**	**Center**	**Size**	**Position**
x				Default component size	Relative to specified edge constraint
x	x			Calculated from specified edge constraints	Relative to specified edge constraints
x	x	x		Determined from specified edge constraints. Explicit size is overridden	Relative to specified edge constraints
x		x		Specified size	Relative to specified edge
x			x	Default component size	Centered in constraint region; edge constraint is ignored
x	x		x	Calculated from edge constraints	Centered in constraint region
x		x	x	Explicit size	Centered in constraint region; single edge constraint is ignored
			x	Default size of component	Centered in constraint region