SAMPLE CHAPTER

Covers Ext JS version 4.0

Ext JS IN ACTION

SECOND EDITION

Jesus Garcia
Grgur Grisogono
Jacob K. Andresen

MANNING

Ext JS in Action, Second Edition

by Jesus Garcia
Grgur Grisogono
Jacob K. Andresen

Chapter 5

brief contents

5 Exploring layouts

This chapter covers

- Using layout systems
- Exploring the `Layout` class inheritance model
- Understanding the Card layout

When building an application, many developers struggle with how to organize their UI and which tools to use to get the job done. In this chapter you'll gain the necessary experience to be able to make these decisions in an educated manner. We'll start by introducing component layouts that are new to Ext JS 4 and go on to explore the numerous container layout models and identify best practices as well as common issues you'll encounter.

The container layout management schemes are responsible for the visual organization of widgets onscreen. They include simple layout schemes such as Fit, where a single child item of a container will be sized to fit the container's body, and complex layouts such as the Border layout, which splits a container's content body into five manageable slices or regions.

We'll have some lengthy explorations of container layouts accompanied by some long examples that can serve as a great springboard for your own layouts. But before we continue our journey with in-depth descriptions of each container layout

management scheme, let's talk about how layout managers work and introduce the new component layouts.

5.1 How layout managers work

As mentioned earlier, the layout management schemes are responsible for the visual organization of widgets onscreen. To do this, they keep track of how the individual child items are placed in relation to one another. The strategy used for placement depends on which layout manager you use. The layout managers are divided into two groups: component and container layouts.

5.1.1 Component layouts

Component layouts are new to Ext JS 4 and are used to lay out the internal items of components. For everyday use, you should be familiar with the Dock component layout. The Dock layout is responsible for managing docked items like toolbars and gives you the option of adding several top and bottom toolbars, as well as adding left and right toolbars (feel free to take a look back at section 4.2 now, if you need to brush up on the details of how to dock items).

If you're implementing your own components and want to implement your own component layout management scheme, then we encourage you to familiarize yourself with the existing hierarchy of component layouts and choose a relevant class to extend.

If you've been using Ext JS 3 or earlier versions, then you may remember the Form layout as cumbersome and complex to use. In Ext JS 4 the Form layout is no longer needed due to the introduction of the Field component layout and its descendants, along with the associated functionality in the code base.

Remember that all the standard components already have their corresponding component layout, so for your everyday programming tasks you don't need detailed knowledge of each component layout. We'll focus on container layouts in this chapter.

5.1.2 Container layouts

Container layouts let you manage the position and size of child components within a container. When you add a component to or remove a component from a container, the container communicates with the parent container and resizes sibling containers or components depending on the layout management scheme.

All of the container layout managers share common functionality available from `Ext.layout.container.Container`. We'll explore each layout manager in detail.

We'll start our journey through container layouts by taking a look at the Auto layout, which is the default layout for containers. The Auto layout is the most basic layout manager, and it shares common functionality with the Container layout.

5.2 The Auto layout

As you may recall, the Auto layout is the *default* layout for any instance of a container. It places items on the screen, one on top of another. Although the Auto layout doesn't

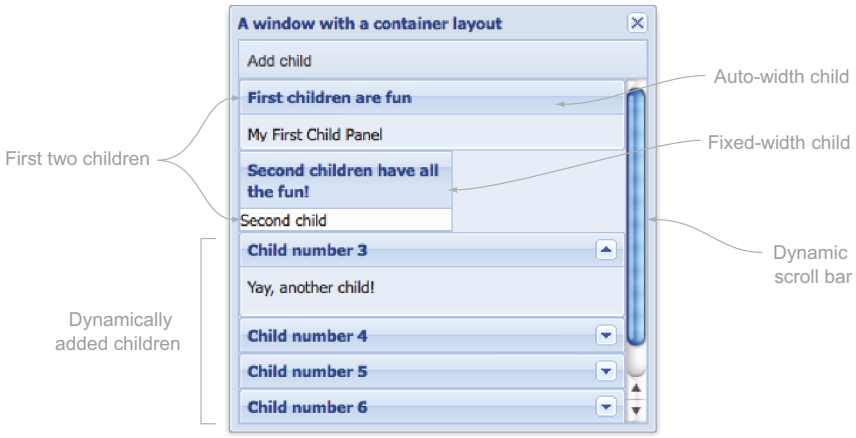

Figure 5.1 The results of your first implementation of the Auto layout

explicitly resize child items, a child's width may conform to the container's content body if it isn't constrained.

An Auto layout is the easiest to implement, requiring only that you add and remove child items. To see this you need to set up a dynamic example, using quite a few components, as shown in the next listing. When you're done, the layout will look like figure 5.1.

Listing 5.1 Implementing the Auto layout

```
var childPnl1 = {
   frame  : true,
   height : 50,
   html   : 'My First Child Panel',
   title  : 'First children are fun'
};
var childPnl2 = {
   width  : 150,
   html   : 'Second child',
   title  : 'Second children have all the fun!'
};
var myWin = Ext.create("Ext.Window", {
   height     : 300,
   width      : 300,
   title      : 'A window with a container layout',
   autoScroll : true,
   items      : [
      childPnl1,
      childPnl2
   ],
   tbar : [
      {
         text    : 'Add child',
         handler : function() {
```

1 Configures first child

2 Configures second panel

3 Creates Window

4 Sets scrollable

5 Adds child panels

6 Configures toolbar

```
                var numItems = myWin.items.getCount() + 1;
                myWin.add({
                   title       : 'Child number ' + numItems,
                   height      : 60,
                   frame       : true,
                   collapsible : true,
                   collapsed   : true,
                   html        : 'Yay, another child!'
                });
             }
          }
       ]
    });
    myWin.show();
```

In listing 5.1, the first thing you do is instantiate object references using XTypes for the two child items that'll be managed by a window: `childPnl1` ❶ and `childPnl2` ❷. These two child items are static. Next, you begin your `myWin` ❸ reference, which is an instance of `Ext.Window`. You also set the `autoScroll` property ❹ to `true`. This tells the container to set the CSS attributes `overflow-x` and `overflow-y` to `auto`, which instructs the browser to show the scroll bars only when it needs to.

Notice that you set the child `items` ❺ property to an array. The `items` property for any container can be an instance of an array used to list multiple children *or* an object reference for a single child. The window contains a toolbar ❻ that has a single button that, when clicked, adds a dynamic item to the window. Note that before Ext JS version 4, you could benefit from calling `doLayout` on the parent container after removing or adding an item; this should no longer be necessary due to the bidirectional communication in the component/container hierarchy. In earlier versions you would've called `myWin.doLayout` after adding one or more child items. If you're performing bulk updates of your component, then you set `suspendLayout` to `true` on the container to avoid calling `doLayout`. The rendered window should look like the one in figure 5.1.

Although the Auto layout provides little to manage the size of child items, it's not completely useless. It's lightweight relative to its subclasses, which makes it ideal if you want to display child items that have fixed dimensions. There are times, though, when you'll want to have the child items dynamically resize to fit the container's content body. This is where the Anchor layout can be useful.

5.3 The Anchor layout

The Anchor layout is similar to other container layouts in that it stacks child items one on top of another, but it adds dynamic sizing into the mix using an `anchor` parameter specified on each child. This `anchor` parameter is used to calculate the size of the child item relative to the parent's content body size and is specified as either a pair of percentages or a pair of offsets, which are integers. The `anchor` parameter is a string, using the following format:

```
anchor : "width, height" // or "width height"
```

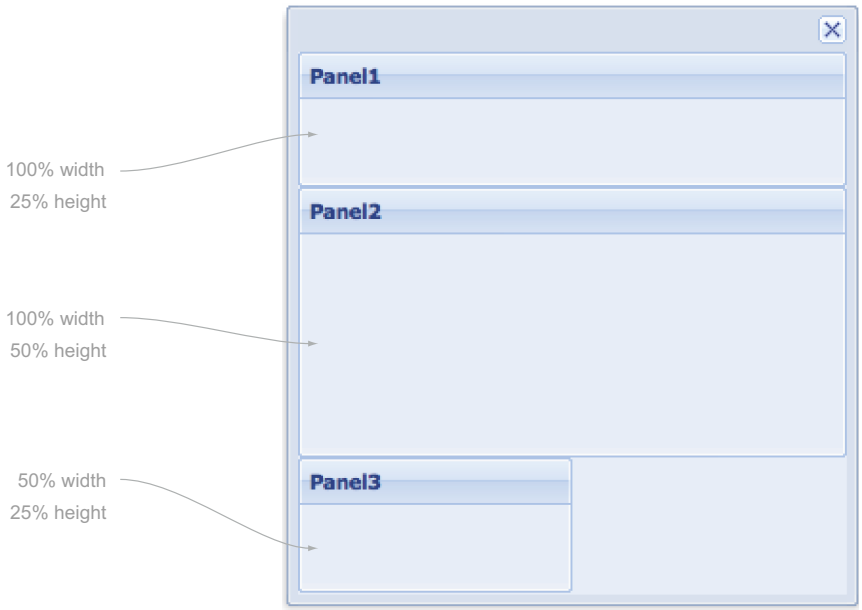

Figure 5.2 The rendered results of your first implementation of the Anchor layout in listing 5.2

Figure 5.2 shows what you'll be constructing.

In the following listing you'll take your first stab at implementing an Anchor layout using percentages.

Listing 5.2 The Anchor layout using percentages

```
var myWin = Ext.create("Ext.Window", ({
    height     : 300,
    width      : 300,
    layout     : 'anchor',
    border     : false,
    anchorSize : '400',
    items      : [
       {
          title  : 'Panel1',
          anchor : '100%, 25%',
          frame  : true
       },
       {
          title  : 'Panel2',
          anchor : '0, 50%',
          frame  : true
       },
       {
          title  : 'Panel3',
          anchor : '50%, 25%',
          frame  : true
```

1 Creates window

2 Sets layout

3 Sets dimensions

4 Configures dimensions

5 Sets size

```
        }
    ]
}));
myWin.show();
```

In listing 5.2 you instantiate `myWin` ❶, an instance of `Ext.Window`, specifying the layout as `'anchor'` ❷. The first of the child items, `Panel1`, has its `anchor` parameter ❸ specified as 100% of the parent's width and 25% of the parent's height. `Panel2` has its `anchor` parameter ❹ specified a little differently, where the `width` parameter is 0, which is shorthand for 100%. You set `Panel2`'s `height` to 50%. `Panel3`'s `anchor` parameter ❺ is set to 50% relative `width` and 25% relative `height`. The rendered item should look like figure 5.2.

Relative sizing with percentages is great, but you also have the option to specify offsets, which allows greater flexibility with the Anchor layout. Offsets are calculated as the content body dimension plus the offset. In general, offsets are specified as negative numbers to keep the child item in view. Let's put on our algebra hats for a second and remember that adding a negative integer is exactly the same as subtracting an absolute integer. Specifying a positive offset would make the child's dimensions greater than the content body's, requiring a scroll bar.

We'll explore offsets by using the previous example, modifying only the child item XTypes from listing 5.2:

```
items : [
 {
   title     : 'Panel1',
   anchor    : '-50, -150',
   frame     : true
 },
 {
   title     : 'Panel2',
   anchor    : '-10, -150',
   frame     : true
 }
]
```

The rendered panel from the preceding layout modification should look like figure 5.3. We reduced the number of child items to two to more easily illustrate how offsets work and how they can cause you a lot of trouble.

It's important to dissect what's going on, which will require you to do a little math. By inspecting the DOM with Firebug, you learn that the window's content body is 285 pixels high and 288 pixels wide. Using simple math, you can determine what the dimensions of `Panel1` and `Panel2` should be:

```
Panel1 Width  = 288px - 50px  = 238px
Panel1 Height = 285px - 150px = 135px
Panel2 Width  = 288px - 10px  = 278px
Panel2 Height = 285px - 150px = 135px
```

You can easily see that both child panels fit perfectly within the window. If you add the height of both panels, you see that they fit, with a total of only 270 pixels. But what

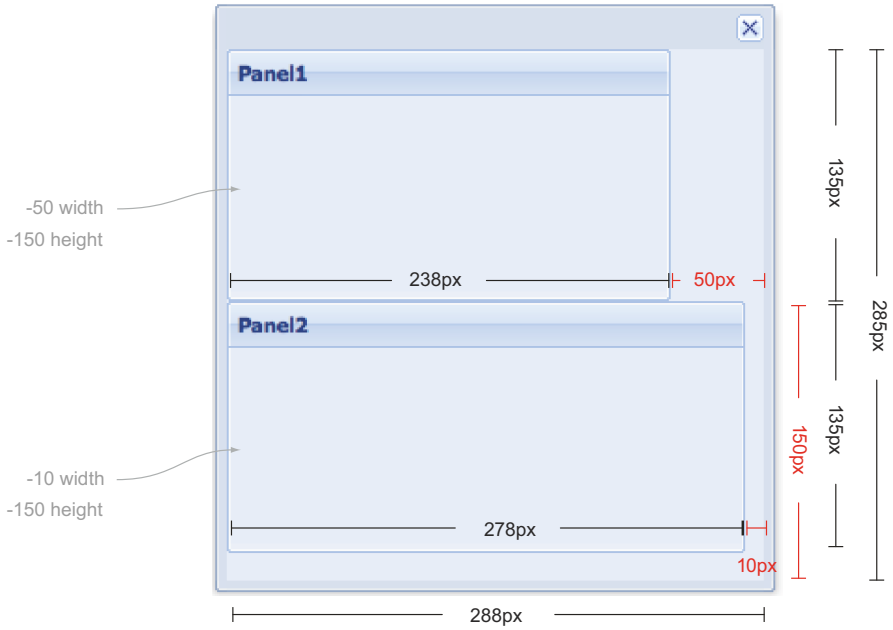

Figure 5.3 Using offsets with an Anchor layout with sizing calculations

happens if you resize the window vertically? Notice anything strange? Increasing the window's height by more than 15 pixels results in `Panel2` being pushed offscreen and scroll bars appearing in the `windowBody`.

Recall that with this layout, the child dimensions are relative to the parent's content body plus a *constant*, which is the offset. To combat this problem, you can mix anchor offsets with fixed dimensions. To explore this concept, modify `Panel2`'s `anchor` parameter and add a fixed height:

```
{
  title     : 'Panel2',
  height    : 150,
  anchor    : '-10',
  frame     : true
}
```

This modification makes `Panel2`'s height fixed at 150 pixels. The newly rendered window can now be resized to virtually any size, and `Panel1` will grow to the window content body minus 150 pixels, which leaves just enough vertical room for `Panel2` to stay onscreen. One neat thing about this is that `Panel2` still has the relative width.

Anchors are used for a multitude of layout tasks. The Anchor layout is used by the `Ext.form.Panel` class by default, but it can be used by any container or subclass that can contain other child items, such as `Panel` or `Window`.

There are times when you need complete control over the positioning of the widget layout. The Absolute layout is perfect for this requirement.

5.4 The Absolute layout

Next to the Auto layout, the Absolute layout is by far one of the simplest to use. It fixes the position of a child by setting the CSS `'position'` attribute of the child's element to `'absolute'` and sets the `top` and `left` attributes to the `x` and `y` parameters that you set on the child items. Many designers place HTML elements as a `position: absolute` with CSS, but Ext JS uses JavaScript's DOM-manipulation mechanisms to set attributes to the elements themselves, without having to muck with CSS. Figure 5.4 shows what you'll be constructing. The next listing shows how to create a window with an Absolute layout.

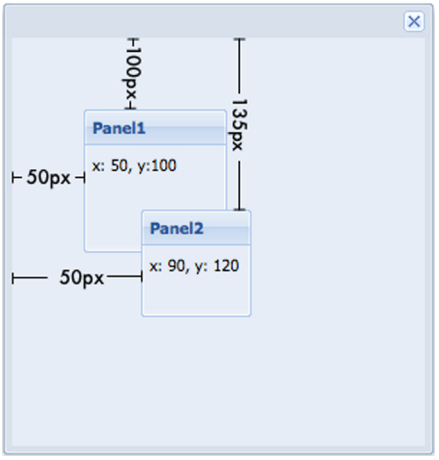

Figure 5.4 The results of your Absolute layout implementation from listing 5.3

Listing 5.3 `AbsoluteLayout` in action

```
var myWin = Ext.create("Ext.Window", {
    height     : 300,
    width      : 300,
    layout     : 'absolute',
    autoScroll : true,
    border     : false,
    items      : [
       {
          title : 'Panel1',
          x     : 50,
          y     : 50,
          height : 100,
          width  : 100,
          html   : 'x: 50, y:50',
          frame  : true
       },
       {
          title  : 'Panel2',
          x      : 90,
          y      : 120,
          height : 75,
          width  : 100,
          html   : 'x: 90, y: 120',
          frame  : true
       }
    ]
});
myWin.show();
```

1 Sets layout

2 Sets child coordinates

3 Sets child coordinates

By now, most of this code should look familiar to you, but there are a few new parameters. The first noticeable change is that the window's `layout` 1 parameter is set to

'absolute'. You attach two children to this window. Because you're using the Absolute layout, you need to specify the X and Y coordinates.

The first child, Panel1, has its X ❷ (CSS left attribute) coordinate set to 50 pixels and Y (CSS top attribute) coordinate set to 50. The second child, Panel2, has its X ❸ and Y coordinates set to 90 pixels and 120 pixels, respectively. The rendered code should look like figure 5.4.

One obvious detail in this example is that Panel2 overlaps Panel1. Panel2 is on top because of its placement in the DOM tree. Panel2's element is below Panel1's element, and because Panel2's CSS position attribute is set to 'absolute' as well, it's going to show above Panel1. Always keep the risk of overlapping in mind when you implement this layout. Also, because the positions of the child items are fixed, the Absolute layout isn't an ideal solution for parents that resize.

If you have one child item and want it to resize with its parent, the Fit layout is the best solution.

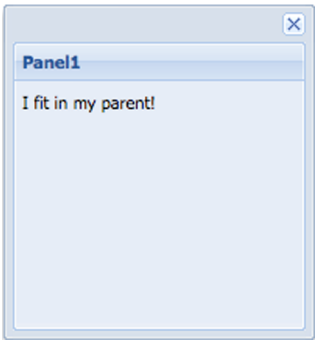

Figure 5.5 Using the Fit layout (listing 5.4)

5.5 The Fit layout

The Fit layout forces a container's single child to "fit" to its body element and is another remarkably simple layout. Figure 5.5 illustrates the end results of this exercise, as shown in the next listing.

Listing 5.4 The Fit layout

```
var myWin = Ext.create("Ext.Window", {
    height      : 200,
    width       : 200,
    layout      : 'fit',
    border      : false,
    items       : [
       {
          title : 'Panel1',
          html   : 'I fit in my parent!',
          frame  : true
       }
    ]
 });
 myWin.show();
```

❶ Configures layout

❷ Adds child

In listing 5.4 you set the window's layout property to 'fit' ❶ and instantiate a single child, an instance of Ext.Panel ❷. The child's XType is assumed by the window's defaultType property, which is automatically set to 'panel' by the window's prototype. The rendered panels should look like figure 5.5.

The Fit layout is a great solution for a seamless look when a container has one child. Often, though, multiple widgets are housed in a container. All other layout-management schemes are generally used to manage multiple children. One of the

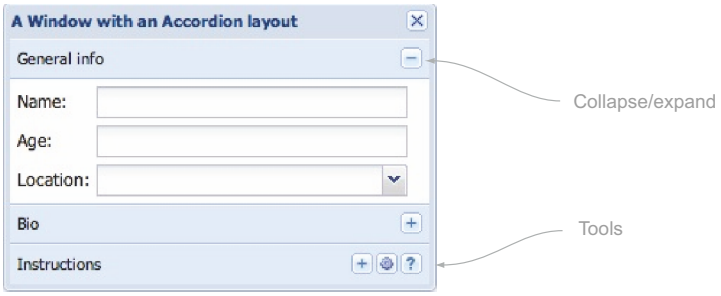

Figure 5.6 The Accordion layout is an excellent way to present the user with multiple items as a single visible component.

best-looking layouts is the Accordion layout, which allows you to vertically stack items that can be collapsed, showing the user one item at a time.

5.6 The Accordion layout

The Accordion layout, shown in the following listing, is a direct subclass of the VBox layout. It's useful when you want to display multiple panels vertically stacked, where only a single item can be expanded or contracted. Figure 5.6 shows the end result.

Listing 5.5 The Accordion layout

```
var myWin = Ext.create("Ext.Window", {
    height       : 200,
    width        : 300,
    border       : false,
    title        : 'A Window with an Accordion layout',
    layout       : 'accordion',
    layoutConfig : {
        animate  : true
    },
    items        : [
       {
          xtype         : 'form',
          title         : 'General info',
          bodyStyle     : 'padding: 5px',
          defaultType   : 'field',
          fieldDefaults : {
              labelWidth: 50
          },
          labelWidth  : 50,
          items       : [
             {
                fieldLabel : 'Name',
                anchor     : '-10'
             },
             {
                xtype      : 'field',
                fieldLabel : 'Age',
```

1 Creates delegate instance

2 Configures layout

3 Adds first child item

```
                size       : 3
             },
             {
                xtype      : 'combo',
                fieldLabel : 'Location',
                anchor     : '-10',
                store      : [ 'Here', 'There', 'Anywhere' ]
             }
          ]
       },
       {
          xtype  : 'panel',                          ❹ Creates text area
          title  : 'Bio',
          layout : 'fit',
          items  : {
             xtype : 'textarea',
             value  : 'Tell us about yourself'
          }
       },
       {
          title : 'Instructions',                    ❺ Adds panel with tools
          html  : 'Please enter information.',
          tools : [
             {id : 'gear'}, {id:'help'}
          ]
       }
    ]
});
myWin.show();
```

Listing 5.5 demonstrates the usefulness of the Accordion layout. The first thing you do is instantiate a window, `myWin`, which has its `layout` property set to `'accordion'` ❶. A configuration option you haven't seen thus far is `layoutConfig` ❷. Some layout schemes have specific configuration options, which you can define as configuration options for a component's constructor.

These `layoutConfig` parameters can change the way a layout behaves or functions. In this case, you set `layoutConfig` for the Accordion layout, specifying `animate: true`, which instructs the Accordion layout to animate the collapse and expansion of a child item. Another behavior-changing configuration option is `activeOnTop`, which, if set to `true`, will move the active item to the top of the stack. When you're working with a layout for the first time, we suggest consulting the API for all the options available to you.

Next you start to define child items, which build on some of the knowledge you've gained so far. The first child is `FormPanel` ❸, which uses the `anchor` parameters you learned about earlier in this chapter. Next you specify a panel ❹ that has its `layout` property set to `'fit'` and contains a child `TextArea`. You then define the last child item ❺ as a vanilla panel with some tools. The rendered code should look like figure 5.6.

Another way to configure layouts

Instead of using both the `layout (String)` as well as the `layoutConfig (Object)` configurations, you can set the `layout` configuration to an `Object` that contains both the layout type and any options for that layout. For example:

```
layout  : {
    type    : 'accordion',
    animate : true
}
```

It's important to note that the Accordion layout can only function well with an `Ext.panel.Panel` and two of its subclasses, `Ext.grid.Panel` and `Ext.tree.Panel`. This is because `Panel` (and the two specified subclasses) has what's required for the Accordion layout to function properly. If you need anything else inside an Accordion layout, such as a tab panel, wrap a panel around it and add that panel as a child of the container that has the Accordion layout.

Although the Accordion layout is a good solution for having more than one panel onscreen, it has limitations. For instance, what if you needed to have 10 components in a particular container? The sum of the heights of the title bars for each item would take up a lot of valuable screen space. The Card layout is perfect for this requirement, because it allows you to show and hide or flip through child components.

5.7 The Card layout

The Card layout ensures that its children conform to the size of the container. Unlike the Fit layout, the Card layout can have multiple children under its control. This tool gives you the flexibility to create components that mimic wizard interfaces.

Except for the initial active item, the Card layout leaves all of the flipping to the end developer with its publicly exposed `setActiveItem` method. To create a wizard-like interface, you need to create a method to control the card flipping:

```
var handleNav = function(btn) {
   var activeItem   = myWin.layout.activeItem,
       index        = myWin.items.indexOf(activeItem),
       numItems     = myWin.items.getCount(),
       indicatorEl  = Ext.getCmp('indicator').el;

   if (btn.text == 'Forward' && index < numItems - 1) {
       index++;
       myWin.layout.setActiveItem(index);
       index++;
       indicatorEl.update(index + ' of ' + numItems);
   }
   else if (btn.text == 'Back' && index > 0) {
       myWin.layout.setActiveItem(index - 1);
       indicatorEl.update(index + ' of ' + numItems);
   }
}
```

Here you control the card flipping by determining the active item's `index` and setting the active item based on whether the Forward or Back button is clicked. You then update the indicator text on the bottom toolbar. Next let's implement your Card layout. The code example in the next listing is rather long and involved, so please stick with us.

Listing 5.6 The Card layout in action

```
var myWin = Ext.create("Ext.Window", {
    height       : 200,
    width        : 300,
    border       : false,
    title        : 'A Window with a Card layout',
    layout       : 'card',
    activeItem   : 0,
    defaults     : { border : false },
    items        : [
       {
          xtype       : 'form',
          title       : 'General info',
          bodyStyle   : 'padding: 5px',
          defaultType : 'field',
          labelWidth  : 50,
          items       : [
             {
                fieldLabel : 'Name',
                anchor     : '-10',
             },
             {
                xtype      : 'numberfield',
                fieldLabel : 'Age',
                size       : 3
             },
             {
                xtype      : 'combo',
                fieldLabel : 'Location',
                anchor     : '-10',
                store      : [ 'Here', 'There', 'Anywhere' ]
             }
          ]
       },
       {
          xtype  : 'panel',
          title  : 'Bio',
          layout : 'fit',
          items  : {
             xtype : 'textarea',
             value : 'Tell us about yourself'
          }
       },
       {
          title : 'Congratulations',
          html  : 'Thank you for filling out our form!'
```

1 **Sets card layout**

2 **Configures active item**

```
        }
    ],
    dockedItems : [
        {
            xtype : 'toolbar',
            dock  : 'bottom',
            items : [
                {
                    text    : 'Back',
                    handler : handleNav
                },
                '-',
                {
                    text    : 'Forward',
                    handler : handleNav
                },
                '->',
                {
                    type    : 'component',
                    id      : 'indicator',
                    style   : 'margin-right: 5px',
                    html    : '1 of 3'
                }
            ]
        }
    ]
});
myWin.show();
```

❸ Adds navigation buttons

❹ Adds indicator component

Listing 5.6 details the creation of a window that uses the Card layout. Although most of this should be familiar to you, we should point out a few things. The first obvious item is the `layout` property ❶, which is set to `'card'`. Next is the `activeItem` property ❷, which the container passes to the layout at render time. You set this to `0` (zero), which tells the layout to call the child component's `render` method when the container renders.

Next you define the bottom toolbar, which contains the Forward and Back ❸ buttons, which call your previously defined `handleNav` method and a generic component ❹ that you use to display the index of the current active item. The rendered container should look like the one in figure 5.7.

Clicking Forward or Back will invoke the `handleNav` method, which will take care of the card flipping and update the indicator component. Remember that with the

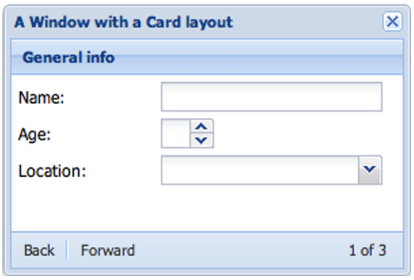

Figure 5.7 Your first Card layout implementation with a fully interactive navigation toolbar (listing 5.6)

Card layout, the logic of the active item switching is completely up to the end developer to create and manage.

In addition to the previously discussed layouts, Ext JS offers a few more schemes. The Column layout is one of the favorite schemes among UI developers for organizing UI columns that can span the entire width of the parent container.

5.8 The Column layout

Organizing components into columns allows you to display multiple components in a container side by side. Like the Anchor layout, the Column layout allows you to set the absolute or relative width of the child components. There are some things to look out for when using this layout. We'll highlight these in a bit, but first let's construct a Column layout window, as shown in the following listing.

Listing 5.7 Exploring the Column layout

```
var myWin = Ext.create("Ext.Window", {
    height       : 200,
    width        : 400,
    autoScroll   : true,
    id           : 'myWin',
    title        : 'A Window with a Column layout',
    layout       : 'column',
    defaults     : {
       frame : true
    },
    items        : [
       {
          title       : 'Col 1',
          id          : 'col1',
          columnWidth : .3
       },
       {
          title       : 'Col 2',
          html        : "20% relative width",
          columnWidth : .2
       },
       {
          title : 'Col 3',
          html  : "100px fixed width",
          width : 100
       },
       {
          title       : 'Col 4',
          frame       : true,
          html        : "50% relative width",
          columnWidth : .5
       }
    ]
});
myWin.show();
```

1 Sets scrollable

2 Configures layout

3 Sets relative width

4 Fixes width to 100 pixels

5 Configures relative width

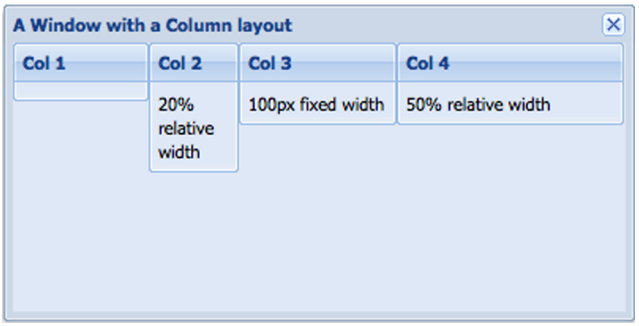

Figure 5.8 Your first Column layout, which uses relative column widths with a fixed-width entity

In a nutshell, the Column layout is easy to use. Declare child items and specify relative or absolute widths or a combination of both, as you do here. In listing 5.7, you set the `autoScroll` property ❶ of the container to `true`, which ensures that scroll bars will appear if the composite of the child component dimensions grows beyond those of the container. Next you set the `layout` property to `'column'` ❷. You then declare four child components, the first of which has its relative `width` set to 30% via the `columnWidth` ❸ property. Set the second child's relative `width` to 20%. You mix things up a bit by setting a fixed `width` for the third child of 100 pixels ❹. Last, you set a relative `width` ❺ of 50% for the last child. The rendered example should look like figure 5.8.

If you tally up the relative widths, you'll see that they total up to 100%. How can that be? Three components, taking 100% width, *and* a fixed-width component? To understand how this is possible you need to dissect how the Column layout sets the sizes of all of the child components. Put your math cap back on for a moment.

The meat of the Column layout is its `onLayout` method, which calculates the dimensions of the container's body, which in this case is 388 pixels. It then goes through all of its direct children to determine the amount of available space to give to any of the children with relative widths.

To do this, it first subtracts the width of each of the absolute-width child components from the known width of the container's body. In this example, you have one child with an absolute width of 100 pixels. The Column layout calculates the difference between 388 and 100, which is 288 (pixels).

Now that the Column layout knows exactly how much horizontal space it has left, it can set the size of each of the child components based on the percentage. It goes through each of the children and sizes each one based on the known available horizontal width of the container's body. It does this by multiplying the percentage (decimal) by the available width. Once complete, the sum of the widths of relatively sized components turns out to be about 288 pixels.

Now that you understand the width calculations for this layout, let's change our focus to the height of the child items. Notice how the height of the child components doesn't equal the height of the container body; this is because the Column layout doesn't manage the height of the child components. This causes an issue with child items that may grow beyond the height of their containers' bodies. This is precisely

why you set `autoScroll` to `true` for the window. You can exercise this theory by adding an extra-large child to the `'Col 1'` component. Enter the following code inside Firebug's JavaScript input console. Make sure you have a virgin copy of listing 5.7 running in your browser:

```
Ext.getCmp('col1').add({
   height : 250,
   title  : 'New Panel',
   frame  : true
});
```

You should now see a panel embedded into the `'Col 1'` panel with its height exceeding that of the window's body. Notice how scroll bars appear in the window. If you didn't set `autoScroll` to `true`, your UI would look cut off and might have its usability reduced or halted. You can scroll vertically and horizontally. The reason you can scroll vertically is that `Col1`'s overall height is greater than that of the window's body. That's acceptable. The horizontal scrolling is the problem in this case. Recall that the Column layout calculated only 288 pixels to properly size the three columns with relative widths. Because the vertical scroll bar is now visible, the physical amount of space in which the columns can be displayed is reduced by the width of the vertical scroll bar. In Ext JS 4, the parent's `doLayout` method is automatically called when adding a component to any of the direct children (in earlier versions you would have to call `doLayout` on the parent to keep your UIs looking great).

As you can see, the Column layout is great for organizing your child components in columns. With this layout, you have two limitations. All child items are always left-justified, and their heights are unmanaged by the parent container. Ext JS offers the HBox layout to help overcome the limitations of the Column layout and extend it far beyond its capabilities.

5.9 The HBox and VBox layouts

The HBox layout's behavior is similar to that of the Column layout because it displays items in columns, but it allows for much greater flexibility. For instance, you can change the alignment of the child items both vertically and horizontally. Another great feature of this layout scheme is the ability to allow the columns or rows to stretch to their parent's dimensions if required.

Let's dive into the HBox layout, shown in the next listing, where you'll create a container with three child panels to manipulate. But first, check out figure 5.9 to see what you're trying to accomplish.

Listing 5.8 HBox layout: exploring the packing configuration

```
Ext.create("Ext.Window", {
    layout       : 'hbox',                          <-- 1 Sets layout to 'hbox'
    height       : 300,
    width        : 300,
    title        : 'A Container with an HBox layout',
```

```
    layoutConfig : {
       pack : 'start'
    },
    defaults : {
       frame : true,
       width : 75
    },
    items : [
       {
          title  : 'Panel 1',
          height : 100
       },
       {
          title  : 'Panel 2',
          height : 75,
          width  : 100
       },
       {
          title  : 'Panel 3',
          height : 200
       }
    ]
}).show();
```

Specifies layout configuration ❷

In listing 5.8 you set `layout` to `'hbox'` ❶ and specify the `layoutConfig` ❷ configuration object. You create the three child panels with irregular shapes, allowing you to properly exercise the different layout configuration parameters. Of these you can specify two, `pack` and `align`, where `pack` means "vertical alignment" and `align` means "horizontal alignment." Understanding the meanings for these two parameters is important because they're flipped for the HBox layout's cousin, the VBox layout. The `pack` parameter accepts three possible values: `'start'`, `'center'`, and `'end'`. In this context, we like to think of them as left, center, and right. Modifying that parameter in listing 5.8 will result in one of the rendered windows in figure 5.9. The default value for the `pack` attribute is `'start'`.

The `align` parameter accepts four possible values: `'top'`, `'middle'`, `'stretch'`, and `'stretchmax'`. Remember that with the HBox layout, the `align` property specifies vertical alignment.

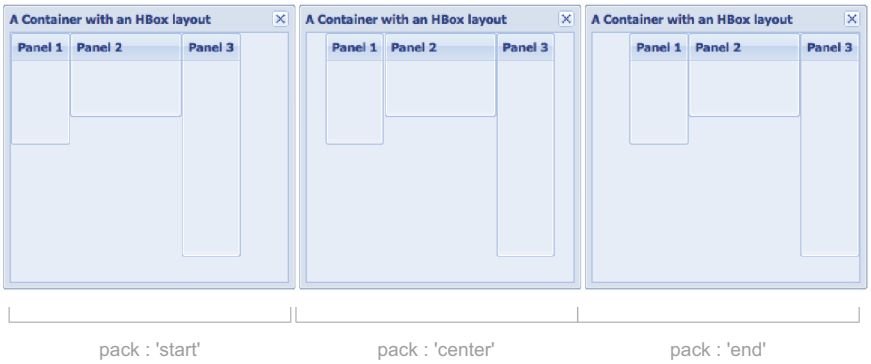

Figure 5.9 The HBox layout options (listing 5.8)

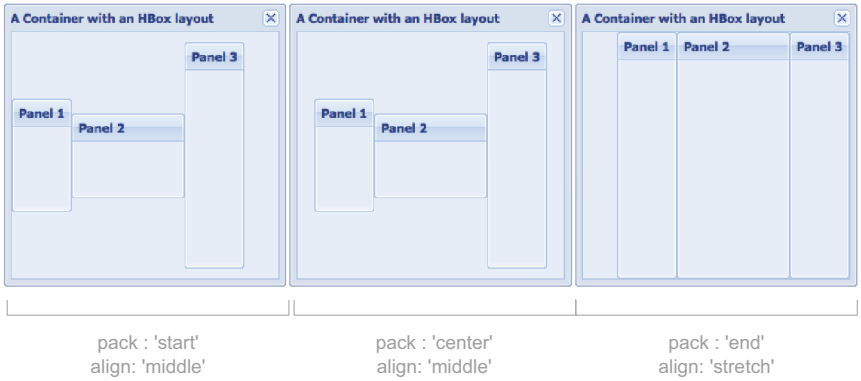

Figure 5.10 The `'stretch'` alignment will always override any height values specified by the child items.

The default parameter for `align` is `'top'`. To change how the child panels are vertically aligned, you need to override the default by specifying it in the `layoutConfig` object for the container. Figure 5.10 illustrates how you can change the way the children are sized and arranged based on a few different combinations.

Specifying a value of `'stretch'` for the `align` attribute instructs the HBox layout to resize the child items to the height of the container's body, which overcomes one limitation of the Column layout.

The last configuration parameter that we must explore is `flex`, which is similar to the `columnWidth` parameter for the Column layout and gets specified on the child items. Unlike the `columnWidth` parameter, the `flex` parameter is interpreted as a weight or a priority instead of a percentage of the columns. Let's say, for instance, you'd like each of the columns to have equal widths. Set each column's `flex` to the same value, and they'll all have equal widths. If you wanted to have two of the columns expand to a total of one half of the width of the parent's container and the third to expand to the other half, make sure that the `flex` value for each of the first two columns is exactly half that of the third column. For instance:

```
defaults : {
   frame : true,
   width : 75
},
items    : [
   {
       title : 'Panel 1',
       flex  : 1
   },
   {
       title : 'Panel 2',
       flex  : 1
   },
   {
       title : 'Panel 3',
```

```
        flex  : 2
    }
]
```

Stacking items vertically is also possible with the VBox layout, which follows the same syntax as the HBox layout. To use the VBox layout, modify listing 5.8 by changing `layout` to `'vbox'`, and refresh the page. Next, you can apply the `flex` parameters described earlier to make each of the panels relative in height to the parent container. We like to think of the VBox layout as the Auto layout on steroids.

Contrasting the VBox layout with the HBox layout, there's one parameter change. Recall that the `align` parameter for the HBox layout accepts a value of `'top'`. For the VBox layout, though, you specify `'left'` instead of `'top'`.

Now that you've mastered HBox and VBox layouts, we'll switch gears to the Table layout, where you can position child components, such as a traditional HTML table.

5.10 The Table layout

The Table layout gives you complete control over how you visually organize your components. Many of you are used to building HTML tables the traditional way, where you write the HTML code. Building a table of Ext JS components is different because you specify the content of the table cells in a single-dimension array, which can get a little confusing.

We're sure that once you've finished these exercises you'll be an expert in this layout. In the next listing you'll create a basic 3x3 Table layout like the one in figure 5.11.

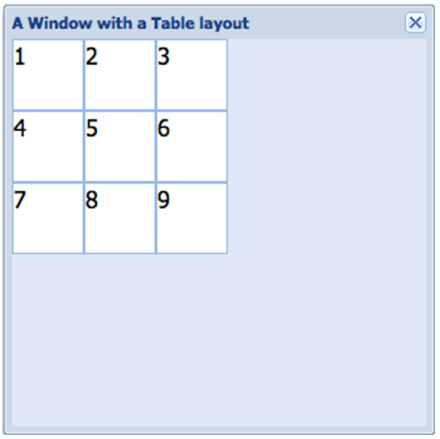

Figure 5.11 The results of your first Table layout in listing 5.9

Listing 5.9 A vanilla Table layout

```
var myWin = Ext.create("Ext.Window", {
    height       : 300,
    width        : 300,
    border       : false,
    autoScroll   : true,
    title        : 'A Window with a Table layout',
    layout       : {
       type      : 'table',
       columns   : 3
    },
    defaults     : {
       height : 50,
       width  : 50
    },
```

1 Specifies layout as 'table'

2 Sets number of columns

3 Configures default size

```
    items        : [
        {
            html : '1'
        },
        {
            html : '2'
        },
        {
            html : '3'
        },
        {
            html : '4'
        },
        {
            html : '5'
        },
        {
            html : '6'
        },
        {
            html : '7'
        },
        {
            html : '8'
        },
        {
            html : '9'
        }
    ]
});
myWin.show();
```

The code in listing 5.9 creates a window container that has nine boxes stacked in a 3x3 formation like in figure 5.11. By now most of this should seem familiar to you, but we want to highlight a few items. The most obvious of these should be the layout `type` parameter ❶, set to `'table'`. Next, you set a layout `column` property ❷, which sets the number of columns. Always remember to set this property when using this layout. Last, you set `defaults` ❸ for all the child items to 50 pixels wide by 50 pixels high.

Often you need sections of the table to span multiple rows or multiple columns. To accomplish this you must specify either the `rowspan` or the `colspan` parameter explicitly on the child items. When you're done your layout will look like figure 5.12.

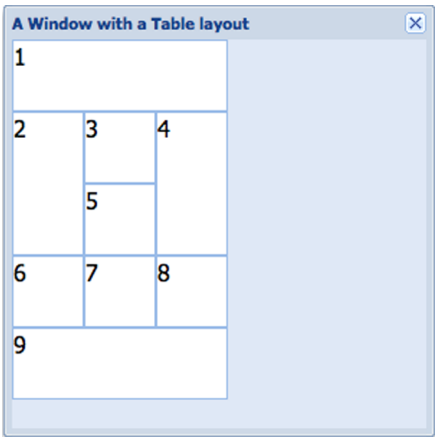

Figure 5.12 When using the Table layout you could specify `rowspan` and `colspan` for a particular component, which will make it occupy more than one cell in the table.

Let's modify your table so the child items can span multiple rows or columns, as shown in the following listing.

Listing 5.10 Exploring `rowspan` and `colspan`

```
items : [
    {
       html    : '1',
       colspan : 3,
       width   : 150
    },
    {
       html    : '2',
       rowspan : 2,
       height  : 100
    },
    {
       html : '3'
    },
    {
       html    : '4',
       rowspan : 2,
       height  : 100
    },
    {
       html : '5'
    },
    {
       html : '6'
    },
    {
       html : '7'
    },
    {
       html : '8'
    },
    {
       html    : '9',
       colspan : 3,
       width   : 150
    }
 ]
```

❶ Sets colspan to 3, width to 150 px

❷ Sets rowspan to 2, height to 100 px

❸ Sets rowspan to 2, height to 100 px

❹ Sets colspan to 3, width to 150 px

In listing 5.10 you reuse the existing `Container` code from listing 5.9 and replace the child `items` array. You set the `colspan` attribute for the first panel ❶ to `3` and manually set its width to fit the total known width of the table, which is 150 pixels. Remember that you have three columns of default 50x50 child containers. Next, you set the `rowspan` property of the second child item ❷ to `2` and its height to the total of two rows, which is 100 pixels. You do the same thing for panel 4 ❸. The last change involves panel 9, which has the exact same attributes as panel 1 ❹. The rendered table after the changes should look like figure 5.12.

When using the Table layout, keep a few things in mind. First, determine the total number of columns that'll be used and specify it in the layout `column` config property. Also, if you're going to have components span rows and/or columns, be sure to set their dimensions accordingly; otherwise the components laid out in the table won't seem to be aligned correctly. The Table layout is extremely versatile and can be used to create any type of box-based layout that your imagination conjures up, with the main limitation being that there's no parent-child size management.

Moving to our last stop on the Ext JS layout journey, we reach the ever-popular Border layout, which lets you divide any container into five collapsible regions that manage their children's size.

5.11 The Border layout

The Border layout made its debut in 2006, back when Ext was little more than a mere extension to the YUI library. It has since matured into an extremely flexible and easy-to-use layout that provides full control over its subparts, or regions. The Border layout has seen widespread use as an easy way to divide complex applications into manageable regions. These regions are aptly named by polar coordinates: North, South, East, West, and Center. Figure 5.13 illustrates a Border layout implementation from the Ext JS SDK.

Depending on the configuration options provided, the region can be resized or collapsed by the user. Options are also available to limit the resize of the region or prevent it from being resized altogether.

To explore the Border layout we'll use the `Viewport` class, shown in the next listing, which will make it easier for you to see the final result of this exercise.

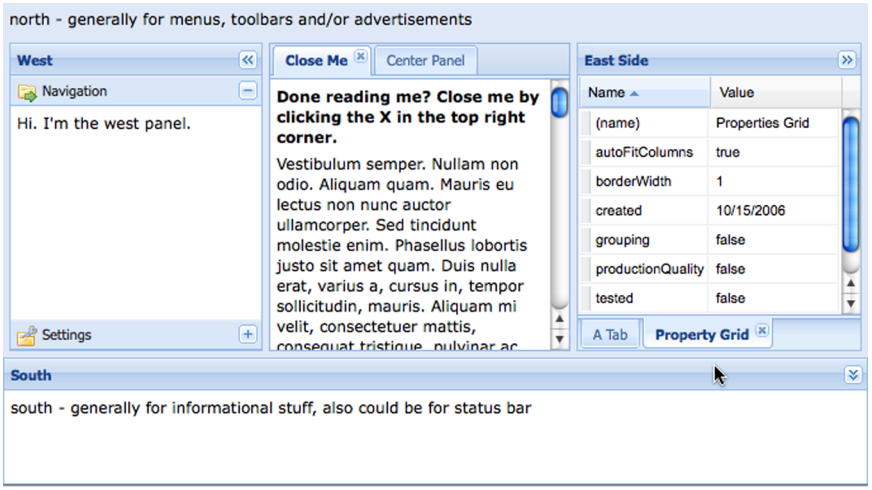

Figure 5.13 The Border layout is what attracts many new developers to the Ext JS framework and is widely used in many applications to divide the screen into task-specific functional areas.

Listing 5.11 Flexing the Border layout

```
Ext.create('Ext.Viewport', {
    layout   : 'border',
    defaults : {
       frame  : true,
       split  : true
    },
    items : [
       {
          title       : 'North Panel',
          region      : 'north',
          height      : 100,
          minHeight   : 100,
          maxHeight   : 150,
          collapsible : true
       },
       {
          title       : 'South Panel',
          region      : 'south',
          height      : 75,
          split       : false,
          margins     : {
             top : 5
          }
       },
       {
          title       : 'East Panel',
          region      : 'east',
          width       : 100,
          minWidth    : 75,
          maxWidth    : 150,
          collapsible : true
       },
       {
          title        : 'West Panel',
          region       : 'west',
          collapsible  : true,
          collapseMode : 'mini',
          width        : 100
       },
       {
          title  : 'Center Panel',
          region : 'center'
       }
    ]
 });
```

❶ Splits regions, allowing for resize

❷ Adds north region

❸ Sets resizable south region

❹ Configures the east region

❺ Adds west region

In listing 5.11 you accomplish a lot using `Viewport` in a few lines of code. You set `layout` to `'border'` ❶ and set `split` to `true` in the default configuration object. There's a lot going on here at once, so feel free to reference figure 5.14, which depicts what the rendered code will look like.

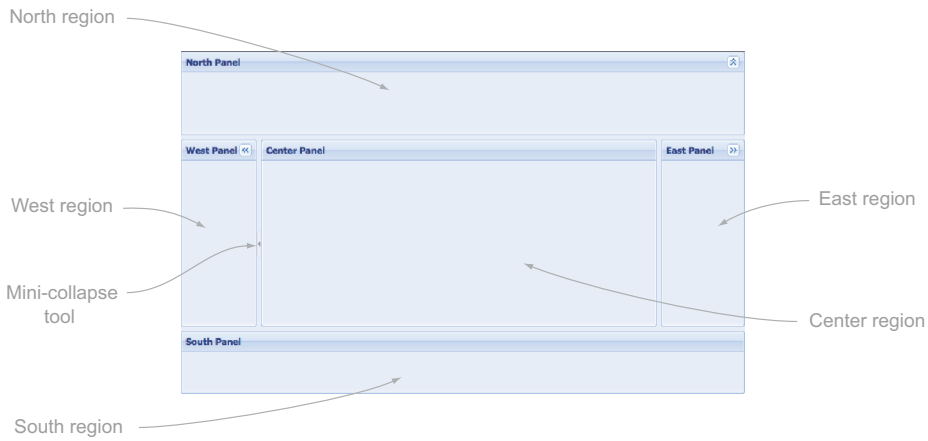

Figure 5.14 The Border layout's versatility and ease of use make it one of the most widely used in Ext JS–based RIAs.

Next, you begin to instantiate child items, which have Border layout region–specific parameters. To review many of them you'll make each region's behavior different from the other (see figure 5.14).

For the first child ❷, you set the `region` property to `'north'` to ensure that it's at the top of the Border layout. You play a little game with the box component–specific parameter, `height`, and the region-specific parameters, `minHeight` and `maxHeight`. By specifying a `height` of `100`, you're instructing the region to render the panel with an initial height of 100 pixels. `minHeight` instructs the region to not allow the split bar to be dragged beyond the coordinates that'd make the northern region the minimum height of 100. The same is true for the `maxHeight` parameter, except it applies to expanding the region's height. You also specify the panel-specific parameter `collapsible` as `true`, which instructs the region to allow it to be collapsed to a mere 30 pixels high.

Defining the south region, the viewport's second child ❸, you set some configuration items to prevent it from being resized but keeping the layout's 5-pixel split between the regions. By setting `split` to `false` you instruct the region to not allow it to be resized. Doing this also instructs the region to omit the 5-pixel split bar, which would make the layout somewhat visually incomplete. To achieve a façade split bar, you use a region-specific `margins` parameter, which specifies that you want the south region to have a 5-pixel buffer between itself and anything above it. One word of caution about this: although the layout now looks complete, end users may try to resize it, possibly causing frustration on their end.

The third child ❹ is defined as the east region. This region is configured much the same as the north panel, but it has sizing constraints that are a bit more flexible. Whereas the north region starts its life out at its minimum size, the east region starts its life between its `minWidth` and `maxWidth`. Specifying size parameters like these

allows the UI to present a region in a default or suggested size while also allowing the panel to be resized beyond its original dimensions.

The west region ❺ has a special region-specific parameter, `collapseMode`, set to the string `'mini'`. Setting the parameter in this way instructs Ext JS to collapse a panel to a mere 5 pixels, providing more visual space for the center region. Figure 5.15 illustrates how small the region will be when collapsed. By allowing the `split` parameter to remain `true` (remember the `defaults` object) and by not specifying minimum or maximum size parameters, the west region can be resized as far as the browser will physically allow.

The last region is the center region, which is the only required region for the Border layout. Although the center region seems a bit bare, it's special indeed. The center region is generally the canvas in which developers place the bulk of their RIA UI components, and its size is dependent on the dimensions of its sibling regions.

For all of its strengths, the Border layout has one huge disadvantage, which is that once a child in a region is defined or created it can't be changed. The fix for this is extremely simple. For each region where you wish to replace components, specify a container as a region. Let's try this by replacing the center region section for listing 5.11:

```
{
    xtype  : 'container',
    region : 'center',
    layout : 'fit',
    id     : 'centerRegion',
    items  : {
       title : 'Center Region',
       id    : 'centerPanel',
       html  : 'I am disposable',
       frame : true
    }
 }
```

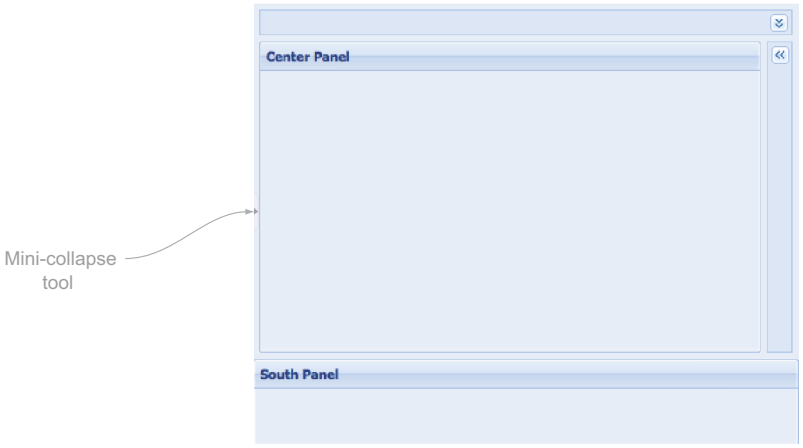

Figure 5.15 The Border layout, where two of the regions, north and east, are collapsed in regular mode and the west panel is collapsed in miniature mode

Remember that the viewport can be created only once, so a refresh of the page where the example code lies is required. The refreshed viewport should look nearly identical to figure 5.15 except that the center region now has HTML showing that it's disposable. In the previous example you define the container XType with a `layout` of `'fit'` and an `id` that you can use with Firebug's JavaScript console.

Think back to our previous discussion and exercises related to adding and removing child components to and from a container—can you recall how to get a reference to a component from its `id` and remove a child? If you can, excellent work! If you can't, we've already worked it out for you. But be sure to review the prior sections because they're extremely important to managing the Ext JS UI. Take a swipe at replacing the center region's child component, as shown in the next listing.

Listing 5.12 Replacing a component in the center region

```
var centerPanel  = Ext.getCmp('centerPanel'),
    centerRegion = Ext.getCmp('centerRegion');

centerRegion.remove(centerPanel, true);

centerRegion.add({
    xtype       : 'form',
    frame       : true,
    bodyStyle   : 'padding: 5px',
    defaultType : 'field',
    title       : 'Please enter some information',
    defaults    : {
        anchor : '-10'
    },
    items       : [
        {
            fieldLabel : 'First Name'
        },
        {
            fieldLabel : 'Last Name'
        },
        {
            xtype      : 'textarea',
            fieldLabel : 'Bio'
        }
    ]
});
```

Listing 5.12 uses everything you've learned so far regarding components, containers, and layouts, providing you with the flexibility to replace the center region's child, a panel, with a form panel, with relative ease. You can use this pattern in any of the regions to replace items at will.

5.12 Summary

This chapter explored the many and versatile Ext JS layout schemes. You learned about some of the strengths, weaknesses, and pitfalls associated with the various layouts. Remember that although many layouts can do similar things, each has its place

in a UI. The correct layout to display components may not be immediately apparent and will take some practice to find if you're new to UI design.

If you aren't 100% comfortable with the material as you finish this chapter, we suggest moving forward and returning to it after some time has passed and the material has had some time to sink in. A good time to revisit this chapter is when you start part 3, "Building an application."

Now that we've covered many of the core topics, put your seatbelt on, because you're going to be in for a wild ride. Next, you'll learn more about Ext JS's UI widgets, starting with forms.

JAVASCRIPT

EXTJS IN ACTION, Second Edition

Garcia • Grisogono • Andresen

Free eBook
SEE INSERT

Ext JS is a mature JavaScript web application framework that provides modern UI widgets and an advanced MVC architecture. It helps you manage tedious boilerplate and minimize hand-coded HTML and browser incompatibilities.

Ext JS in Action, Second Edition starts with a quick overview of the framework and then explores the core components by diving into complete examples, engaging illustrations, and clear explanations. You'll feel like you have an expert guide at your elbow as you learn the best practices for building and scaling full-featured web applications.

What's Inside

- Building professional web apps with Ext JS
- Stamping out DOM fragments with templates
- Customizing and building Ext widgets
- Masterful UI design

A working knowledge of JavaScript is assumed. No prior experience with Ext JS is required.

Jay Garcia is a well-known member of the Ext JS community and a contributor to the framework. He wrote *Sencha Touch in Action*. Grgur Grisogono founded SourceDevCon in London, UK and Split, Croatia. Jacob Andresen is a consultant specializing in large scale internet applications.

"A must-have book to learn Ext JS properly."
—Loiane Groner, Citibank

"A guided tour for an action-packed journey from novice to expert."
—Jeet Marwah, gen-E

"Master every detail of Ext JS 4.0 and how to build desktop-like UIs for the web."
—Efran Cobisi
Microsoft MVP, MCSD

"The missing link between experimenting with Ext JS and being productive with it."
—Raul Cota
Virtual Materials Group

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/ExtJSinActionSecondEdition

ISBN 13: 978-1-617290-32-9
ISBN 10: 1-617290-32-7
54999
9 781617 290329

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]