CHAPTER 4

# Layout Management

As you can see from the previous chapters, ThinWire greatly simplifies Web application development. Assembling a simple form is as easy as creating a `Panel`, creating `Components`, and placing the `Components` on the `Panel`, provided you specify the exact positions and sizes (in pixels) where you want to place the `Components`. Unfortunately, by bringing the desktop development model to the Web, we've also brought along the frustrations of calculating fixed pixel coordinates.

It starts out simple enough. You can make some basic assumptions about sizes and spacing between the `Components`, and do the calculations as you add a new `Component`. But what happens when you need to insert a `Component` in the middle of your form? Just imagining having to recalculate the ycoordinates of 20 `Components` makes me shudder. Or, what if you have a `Panel` that you want to resize with the browser? You could have a `PropertyChangeListener` on the `Frame` that recalculates the dimensions of your `Panel`, but, again, this can get tedious. There has to be a better way, and there is!

## Enter Layout Managers

My long-standing rule is that if a task is too cumbersome or time-consuming, make somebody else do it. As a programmer, that "somebody else" is often times my computer. There's nothing like a computer for handling repetitive calculations, and that's pretty much the point behind a layout manager.

A layout manager is a common user interface helper found in many frameworks and in many languages. The ThinWire framework currently comes with just two layout managers that are designed for very different purposes. In this chapter, we discuss both layout managers in detail and wrap things up by showing you how to create your own layout manager.

**Layout Ignores Other Position/ Dimension Settings**

If a `Container` is managed by a layout manager, the x, y, width, and height properties should not be used on the `Components`. If you set values for these properties, they will be ignored and will be overwritten by the calculated values from the layout manager.

First, let's talk about what makes up a layout manager. Although different layout managers may employ different concepts and techniques, all layout managers do fundamentally the same thing. By definition, they arrange or "lay out" where the `Components` of a `Container` should be positioned, and in some cases, what size the `Components` should be. Because of this commonality, all layout managers for ThinWire implement the `thinwire.ui.layout.Layout` interface. The interface defines an apply method and a set of common properties. The `apply` method is where the "magic" happens. Normally, the `apply` method is called automatically by the framework, but you can invoke it at will. It will recalculate the sizes and positions of all the `Components` in the `Container`.

## Margin and Spacing

All `Layouts` have two common properties: `margin` and `spacing`. The `margin` is the space (in pixels) from the edge of the `Container` to the edge of the first `Component`. The `spacing` is the space (in pixels) between the `Components`. By having these properties be an inherent feature of a layout manager, you can line your `Components` right next to each other, and then later determine that they need to be spaced 2px apart.

**Controlling When Layout Occurs with Auto Apply**

The two `Layouts` included with ThinWire are both set to auto apply by default. This means that when the dimensions or the content of the `Container` change, the layout is recalculated and applied automatically. There is usually no need for the developer to ever invoke the `apply` method. If for some reason this behavior is not desired, you may call `setAutoApply(false)`.

# SplitLayout

The simplest layout manager in the framework is `SplitLayout`. `SplitLayout` divides a `Container` into two sections, placing one `Component` on each side. `SplitLayout` provides a draggable divider, allowing the user to adjust the split. The split can be either vertical (splitting the Components left and right) or horizontal (splitting the Components top and bottom). Example time (see Listing 4.1A).

**LISTING 4.1A Create a TabSheet with Two Buttons Split Evenly**

```
TabSheet ts = new TabSheet("Basic Split");
ts.getChildren().add(new Button("Top"));
ts.getChildren().add(new Button("Bottom"));
ts.setLayout(new SplitLayout(.5));
```

**SplitLayout Only Works with Two Components**

SplitLayout works with exactly two Components. No more, no less. May God help you if you decide it would be funny to add a third Component. Boy, you are playing with fire!

A Layout is associated to a Container via the setLayout method on Container. Only one Layout may be assigned to each Container, and a Layout can only be assigned to one Container. The preceding code uses the simplest constructor for SplitLayout. The one parameter is a double value that defines the size of the first Component in the Container. If the value is greater than 1 (a whole number), it represents the size in pixels. If the value is less than 1 (a decimal), it represents a percentage of the available space. In this example, .5 stands for 50 percent of the available space; therefore, the two Buttons split the Container evenly.

Notice how one Button is on top, and the other is on bottom. This is called a horizontal split. If we want a vertical split, we set the splitVertical property to true or use the two-parameter constructor (see Listing 4.1B).

**LISTING 4.1B Create a TabSheet with Two Buttons Split Vertically**

```
TabSheet ts = new TabSheet("Vertical Split");
ts.getChildren().add(new Button("Left"));
ts.getChildren().add(new Button("Right"));
ts.setLayout(new SplitLayout(.5, true));
```

**Default Spacing for SplitLayout**

Whereas in most cases you would expect the initial margin and spacing for a layout to be 0, the spacing is set to 4 by default for `SplitLayout`.

Now we have a `Panel` with two `Buttons` split evenly and vertically (see Figure 4.1). What more could we want? Let's try changing the margin and spacing. How about a 10-pixel margin around the edge of the `Panel` and a 20-pixel space between the `Buttons` (see Listing 4.1C)?

**LISTING 4.1C SplitLayout with a 10px Margin and 20px Spacing**

```
TabSheet ts = new TabSheet("Vertical Split with Margin and Spacing");
ts.getChildren().add(new Button("Left"));
ts.getChildren().add(new Button("Right"));
ts.setLayout(new SplitLayout(.5, true, 10, 20));
```


**FIGURE 4.1** Running the SplitLayoutExample 1 and clicking the third tab.

Isn't it beautiful? Plus, if the Panel's dimensions change, so do the Buttons. SplitLayout will keep them in proportion.

Another feature of the SplitLayout is the ability to maximize one of the Components. This is accomplished with the setMaximize(Maximize) method. This method takes an enum constant that identifies whether the first or second Component is to be maximized. The options are NONE, FIRST, or SECOND. When a Component is set to be maximized, its previous state is preserved. Simply pass NONE to setMaximized to restore this state.

The only other thing you might possibly want with this layout is to make the spacer more visible. To accomplish this, we use the getSpacerStyle method, which returns a Style object. We cover the Style API in the next chapter, so just take our word that the code in Listing 4.2 will give you a green spacer (see Figure 4.2).

**LISTING 4.2 SplitLayout with Maximize and Green Spacer**

```
Frame f = Application.current().getFrame();
final SplitLayout split = new SplitLayout(.5, true, 10, 20);

// Make the spacer green
split.getSpacerStyle().getBackground().setColor(Color.LIGHTGREEN);

// Add two buttons, that when clicked will maximize or restore themselves
Button left = new Button("Left");
left.addActionListener(Button.ACTION_CLICK, new ActionListener() {
public void actionPerformed(ActionEvent ev) {
            if (split.getMaximize() == SplitLayout.Maximize.FIRST) {
                  split.setMaximize(SplitLayout.Maximize.NONE);
            } else {
                  split.setMaximize(SplitLayout.Maximize.FIRST);
            }
```

**LISTING 4.2 Continued**

```
      }
});

f.getChildren().add(left);

Button right = new Button("Right");
right.addActionListener(Button.ACTION_CLICK, new ActionListener() {
      public void actionPerformed(ActionEvent ev) {
            if (split.getMaximize() == SplitLayout.Maximize.SECOND) {
                  split.setMaximize(SplitLayout.Maximize.NONE);
            } else {
                  split.setMaximize(SplitLayout.Maximize.SECOND);
            }
      }
});

f.getChildren().add(right);

f.setLayout(split);
```

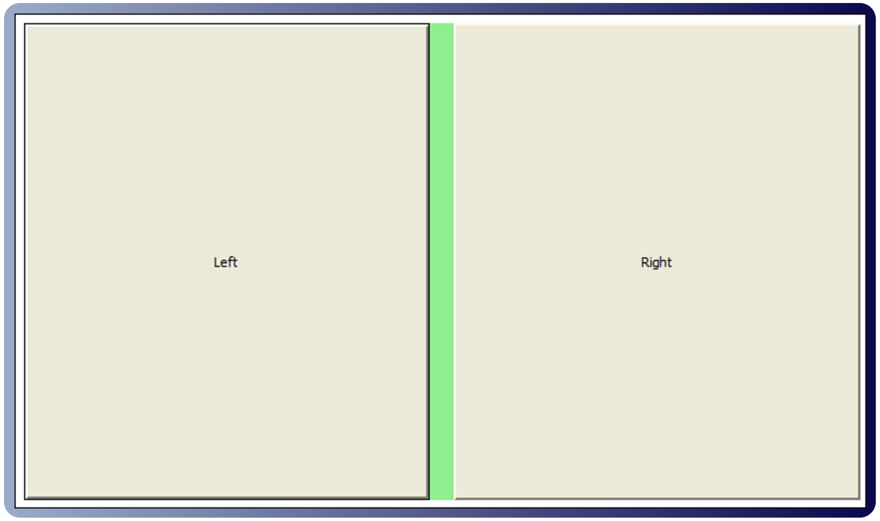



**FIGURE 4.2**
Running the SplitLayoutExample2.

## TableLayout

The other layout manager included with ThinWire is `TableLayout`. Whereas `SplitLayout` was created with a specific purpose in mind, and has a limited set of applications, `TableLayout` was created to be the ultimate, universal layout manager.

The concept of ThinWire's `TableLayout` was borrowed from an open source layout for swing called TableLayout (http://tablelayout.dev.java.net). The concept is simple.

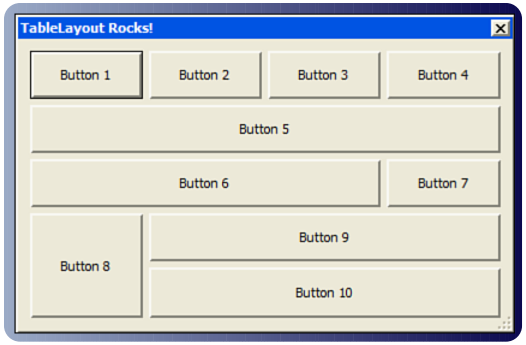
Think back to the early days of the Web. Do you remember your first masterpiece, the first (and last) time you claimed the title WebMaster? And what a master you were! Scrolling, blinking text, background music, Comic Sans font, and a hidden HTML table to lay out the site. Well, those days are long gone (thank goodness); but, although we all acknowledge the superiority of CSS for Web site layout, there is still something to be said about the simplicity of the HTML table layout. You could just basically draw the structure of the page as a table and place the content where it belonged. With `TableLayout`, this paradigm makes a triumphant return to Web design! See Listing 4.3 and Figure 4.3 for an example.

**LISTING 4.3 TableLayout**

```
Dialog dlg = new Dialog("TableLayout Rocks!");
dlg.setResizeAllowed(true);
dlg.setLayout(new TableLayout(new double[][]{
{0, 0, 0, 0}, //Column Widths
{0, 0, 0, 0, 0}}, //Row Heights
10, //Margin around edge of container
5)); //Spacing between cells

List<Component> kids = dlg.getChildren();
kids.add(new Button("Button 1").setLimit("0, 0, 1, 1"));
kids.add(new Button("Button 2").setLimit("1, 0, 1, 1"));
kids.add(new Button("Button 3").setLimit("2, 0, 1, 1"));
kids.add(new Button("Button 4").setLimit("3, 0, 1, 1"));
kids.add(new Button("Button 5").setLimit("0, 1, 4, 1"));
kids.add(new Button("Button 6").setLimit("0, 2, 3, 1"));
kids.add(new Button("Button 7").setLimit("3, 2, 1, 1"));
kids.add(new Button("Button 8").setLimit("0, 3, 1, 2"));
kids.add(new Button("Button 9").setLimit("1, 3, 3, 1"));
kids.add(new Button("Button 10").setLimit("1, 4, 3, 1"));

dlg.setVisible(true);
```

**FIGURE 4.3**
Running the TableLayoutExample1.




> **TableLayout Does NOT Use HTML Tables**
> Before you get all worked up about using HTML tables for screen layout, let me reassure you that behind the scenes we're utilizing CSS for `Component` placement. Like any other layout manager, `TableLayout` is merely an abstraction of the actual screen layout.

`TableLayout` takes (up to) three parameters in its constructor. The last two you should recognize from `SplitLayout`. The `margin` and `spacing` parameters have the exact same meaning for `TableLayout` as they do for `SplitLayout`. The first parameter is a two-dimensional `double` array that defines the structure of the table. The first array is the table's column widths; the second array is the table's row heights.

Column widths and row heights can be specified in three different ways:

- Fixed pixel size (whole numbers > 1)
- Percentages (decimal numbers > 0 && < 1)
- Fill (0)

In the preceding example, we create a table with four columns of equal widths and five rows of equal heights. Because we specified a size of zero (0) for each row and column, they will expand equally to fill all available space in the Container. If you were to drag and resize the Dialog, the dimensions of the Buttons in the table would change.

SplitLayout is simple in that it only ever works with two Components, and there are only two ways to arrange (left-right or top-bottom). TableLayout is designed to handle much more, and so it needs to know some extra information about the Components in the Container. Specifically, it needs to know in which cell(s) the Components belong. If this were Swing, this information would be provided by a constraint on the Component. Because this is ThinWire and not Swing, however, we have something a little ~~more sensible~~ different.

## Limits

A limit is fundamentally the same as a Swing constraint. It's called a limit in ThinWire because limit is easier to spell, and we're all about simplicity. Because we take advantage of the List interface to manage the children of a Container, we prefer not to offer an overloaded add method to specify the limit. Instead, we have a setLimit method on Component that returns the Component.

Any layout can utilize limits. A limit, as defined by the Component interface, is just an object, so it can be anything you need it to. For TableLayout, the limit defines the top-left column and row, and the size (in columns and rows). For example, a Component that was in row 0 and column 0 would have its limit defined as "0, 0, 1, 1". This simply means that the Component starts in cell 0, 0 and is 1 column wide and 1 row tall. We realize that the default case is for a Component to occupy only one cell, so the same limit can simply be defined as "0, 0".

**Limits Are Defined by the Layout Manager**

As defined by the `Component` interface, a limit is simply an object. It is left up to the layout manager to define the limit. In the case of `TableLayout`, we assign the limits as a `String`. `TableLayout` implements a protected method called `getFormatlLimit` that converts the `String` assigned to a `TableLayout.Range`, which (like `GridBox.Range`) defines `Row` and `Column` information (and justification and size in a formal structure).

## Relative Limits

When you're prototyping a form, even managing the cell positions of `Components` can become cumbersome, especially if you are constantly rearranging `Components` on the screen. If this is the case, `TableLayout` enables you to specify the limit as an offset to the previously added `Component` (see Figure 4.4). For example, `setLimit("+1, +1")` positions that `Component` one `Column` to the right of the prior `Component`, and one `Row` below the prior `Component`. If you define your `Components` with relative limits, rearranging their position on the screen is as easy as rearranging the order in which they are added to the `Container` (see Listing 4.4).

**LISTING 4.4 A Slightly More Wacky Use of TableLayout**

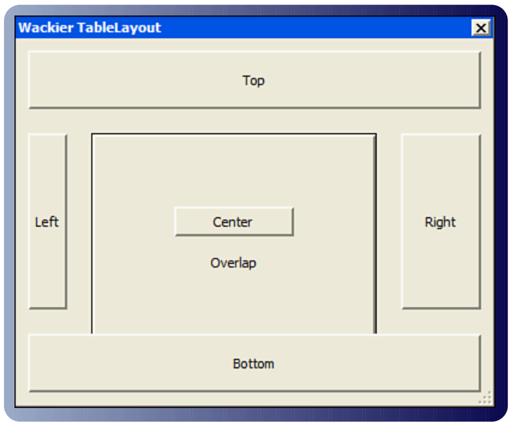
```
Dialog dlg = new Dialog("Wackier TableLayout");
dlg.setResizeAllowed(true);

dlg.setLayout(new TableLayout(new double[][]{
{10, .1, 20, 0, 20, .2, 10},
{10, .2, 20, 0, 20, .2, 10}}
));

List<Component> kids = dlg.getChildren();
kids.add(new Button("Overlap").setLimit("3, 3, 1, 3"));
```

**LISTING 4.4 Continued**

```
kids.add(new Button("Center").setSize(100, 25).setLimit("3, 3, c, c"));
kids.add(new Button("Right").setLimit("5, 3, 1, 1"));
kids.add(new Button("Left").setLimit("1, 3, 1, 1"));
kids.add(new Button("Bottom").setLimit("1, 5, 5, 1"));
kids.add(new Button("Top").setLimit("1, 1, 5, 1"));

dlg.setVisible(true);
```

**FIGURE 4.4**
Running the TableLayoutExample2.




If a `Component` has a fixed size, a justification may be specified in the limit rather than column and row sizes. In the preceding example, the Center button has a width of 100 and a height of 25, and is centered in cell 3, 3. Its limit is defined as "`3, 3, c, c`". The first `c` centers the `Component` horizontally, and the second `c` centers it vertically. `Components` may also be left (`l`), right (`r`), top (`t`), or bottom (`b`) justified within a single cell.

We believe that this model allows the flexibility to lay out `Components` in any fashion while still remaining very simple. But wait, there's more.

## Use TableLayout Like a GridBox

As we were implementing the `TableLayout`, the thought occurred that conceptually a table could be considered the same as a grid. In Chapter 2, "Component Overview," we introduced our `GridBox Component` that allows data to be displayed and manipulated in a grid. Wouldn't it be cool if the `TableLayout` also implemented the same interface as the `GridBox`? We certainly thought so.

The `Grid` API takes the `TableLayout` to a whole new level. The definition of the layout itself is now dynamic. `Rows` and `Columns` can be added and removed at any point in your application (see Figure 4.5). `Components` can be referenced directly by their location in the `Grid`. And we decided to add some gravy by allowing you to toggle the visibility of `Rows` and `Columns` and have the layouts adjust to fill in the gaps (see Listing 4.5).

**LISTING 4.5 The Editable GridBox**

```
class EditableGridBox extends Panel {
List<TableLayout.Column> columns;
List<TableLayout.Row> rows;

public EditableGridBox() {
TableLayout layout = new TableLayout();
            layout.setMargin(1);
            layout.setSpacing(1);
            columns = layout.getColumns();
            rows = layout.getRows();
            setLayout(layout);
```

**LISTING 4.5 Continued**

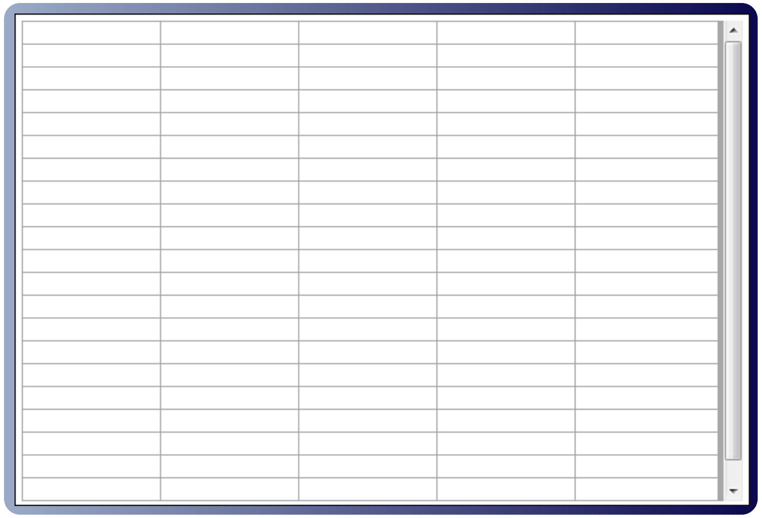
```
            getStyle().getBackground().setColor(Color.DARKGRAY);
            setScrollType(ScrollType.AS_NEEDED);
      }

      public void addColumn() {
            TableLayout.Column col = new TableLayout.Column(0);
            columns.add(col);
            for (int i = 0, cnt = col.size(); i < cnt; i++) {
col.set(i, newCell());
}
      }

      public void addRow() {
            TableLayout.Row row = new TableLayout.Row(18);
            rows.add(row);
            for (int i = 0, cnt = row.size(); i < cnt; i++) {
row.set(i, newCell());
            }
      }

      public String getText(int column, int row) {
            return ((TextField) rows.get(row).get(column)).getText();
      }

      public void setText(int column, int row, String text) {
```

**LISTING 4.5 Continued**

```
            ((TextField) rows.get(row).get(column)).setText(text);
        }

        private static TextField newCell() {
            TextField tf = new TextField();
            tf.getStyle().getBorder().setSize(0);
            return tf;
        }
}
```

**FIGURE 4.5**
Running the TableLayoutExample3.



One of the most requested features is the ability to type directly into the cells in a `GridBox`. Although the `GridBox Component` does not currently support this, we can create a reasonable facsimile using a `Panel`, many `TextFields`, and `TableLayout`. The API is identical to the `GridBox` with one exception. `TableLayout.Rows` and `TableLayout.Columns` need a size specified to their constructors.

## Custom Layout Manager

Although `TableLayout` should be able to handle any layout task you can throw at it, you might have a desire to create your own layout manager. If the mood should strike you, we have provided a default implementation of most of the methods in the abstract class `AbstractLayout`. As an example, we walk you through the process of creating a simple flow layout.

First we create a new class that extends `AbstractLayout`. A flow layout arranges `Components` in a directional flow, much like lines of text in a paragraph. The `FlowLayout` included with Swing allows for many configurations. For simplicity, our `FlowLayout` arranges `Components` left-to-right and aligns them along the top (see Listing 4.6).

Every layout manager must implement its own `apply` method. This is where all the positioning work is done (see Figure 4.6).

**LISTING 4.6 Custom Flow Layout**

```
public class FlowLayout extends AbstractLayout {
public void apply() {
// If the layout is not associated with a Container,
// this method should do nothing.

if (container == null) return;

// Determine the maximum width available
```

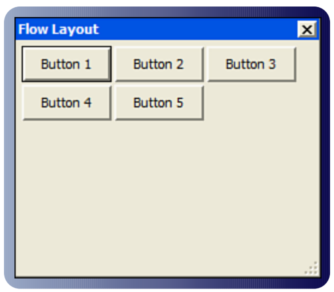
**LISTING 4.6 Continued**

```
int width = container.getInnerWidth() - margin;

// Set initial coordinate values
int rowHeight = 0;
int x = margin;
int y = margin;
boolean first = true;
for (Component c : container.getChildren()) {
int compWidth = c.getWidth();
int compHeight = c.getHeight();

// If the component doesn't fit on this row
// and it is not the first Component on the row,
// start a new row
if (x + compWidth > width && !first) {
y += rowHeight + spacing;
x = margin;
rowHeight = compHeight;
} else {
if (rowHeight < compHeight) rowHeight = compHeight;
first = false;
}

c.setPosition(x, y);
```

**LISTING 4.6 Continued**

```
// Determine the x coordinate of the next Component
x += compWidth + spacing;
}
}

public FlowLayout(int margin, int spacing) {
setMargin(margin);
setSpacing(spacing);
setAutoApply(true);
      }
}
```

**FIGURE 4.6**
Running the CustomLayoutExample and resizing Dialog.




Notice that we make use of some member variables: `container`, `margin`, and `spacing`. These are defined in `AbstractLayout` as protected members.

For convenience, we have the constructor take the `margin` and `spacing` as parameters. Using the setter methods rather than the properties directly allow us to take advantage of some range checking. What's important about the constructor is the call to `setAutoApply(true)`. By calling this method and passing it `true`, we're instructing `AbstractLayout` to call the `apply` method any time the dimensions of the `Container` change, a `Component` is added or removed, `margin` is changed, or `spacing` is changed. Everything required to make this layout dynamically applied is taken care of by `setAutoApply(true)`.

## Summary

ThinWire is all about making the developer's life easier, and layout managers provide the means to quickly get an application "all laid out like that." `TableLayout` should take care of 90 percent of your layout needs. For the other 10 percent, we have `SplitLayout` and the `AbstractLayout` (which implements all the common features and allows the developer to quickly create a new layout).

We're getting closer to completeness in our discussion of ThinWire. You know how easy it can be to create very complex and interactive applications. The next chapter shows you how to customize the look of the `Components` and your application as a whole.