Creating Platform-Independent Interfaces with Java Layout Managers

Type: Technical Session
Track: JBuilder

Author: Ted Faison, Faison Computing Inc.

Short Description

Java's Layout Managers are the key to creating interfaces that work correctly on different platforms. Layout Managers work hard to make your dialog boxes appear correct after being resized, or when the screen resolution changes.

Abstract

Windows programmers often use dialog boxes in the user interfaces they develop. Creating dialog boxes involves carefully laying out controls using a tool, such as Resource Workshop. Layouts that look right on one system may look wrong on systems with different video resolutions. Java does away with the fixed layout convention of dialog boxes, in favor of a dynamic technique that lays out controls at runtime, based on the video resolution available. This paper discusses the new Java Layout Managers, showing how to get the most out of them, and how to extend them to obtain custom layouts.

What is a Layout Manager?

Most GUI programs use dialog boxes to display a series of controls. We're all familiar with the tedious process of laying out the controls using tools like Resource Workshop or AppStudio. For a non-trivial dialog box, it takes a lot of time to position the controls where they look best. When you're finished coding the app, you give it to the test people. You haven't even gotten back to your office that the phone is ringing. The dialog box has clipped text, and not all the controls are visible. Sound familiar? The problem, of course, is the way Windows lays out dialog box controls, using those arcane dialog box units, The original idea was to lay out controls based on the font size. The idea was good, since it took into consideration that layout on different systems was an issue. The bad part was tying the layout policy to the font size.

The JavaSoft folks designed the Java AWT system to perform well on all platforms. Using font sizes as a reference made almost as much sense as using the size of the tires of James Goslings' car. The solution was to create a small family of Layout Managers, that at runtime would go out and look at the size of a window and lay each control out based on the amount of space available, the size of the control and the layout policy. A layout manager is an object that helps an AWT container (such as `java.awt.Frame`) position its controls in a manner that will look reasonable on any platform. Thus the burden of component positioning is shifted from the developer to Java. Does that mean you can't position controls exactly to the millimeter? Yes, for most layout policies. The idea is that *you*, the developer, don't know as much about the final appearance of your windows as the layout manager, because you don't know what the runtime platform looks like. The layout manager is sitting right there, running on that platform, so it can make some pretty reasonable decisions.

Keep in mind that layout managers are there to help, not hinder. If you don't specify any layout manager for a window, then your components will stay exactly where you put them. Just remember that your window design is tied to parameters of your design environment, so it will probably look wrong on machines with differing resolution and font size.

Basic Concepts

In order to display Java components on the screen, you need a window to hold them. In Windows we usually refer to this kind of window as the *parent window*. In Java, the window is just a component, but a special kind that can hold multiple components inside it. Such components are called *containers* (not to be confused with OOP containers like vectors and hash tables). The JDK defines a simple hierarchy of container classes, as shown in Figure 1.

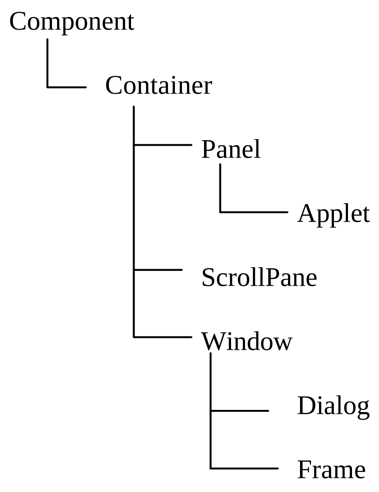


Figure 1 - The hierarchy of Java containers.

All containers maintain a list of their child components. The order of the list defines their Z-order. Unless otherwise specified, components are added at the end of the list, causing them to be displayed on top of the components that precede them in the list. Layout managers can use this child list to gain access to a container's children. Layout managers are created in a container using code like this:

```
public class MyFrame extends java.awt.Frame {

  public MyFrame() {
    setLayout(new java.awt.BorderLayout() );
  }
}
```

The layout manager is created without being passed a reference to the container to manage. For this reason, every time the container needs to have the manager layout its children, it passes it a reference to itself, like this:

```
public abstract class Container extends Component {
  //...
  public void layout() {
    LayoutManager layoutMgr = this.layoutMgr;
    if (layoutMgr != null)
      layoutMgr.layoutContainer(this);
  }
}
```

The `layout` method is normally called just before the container paints itself.

The Notion of Preferred Size

When you create a container with components embedded on it, you test its layout on your own development platform. You would like the window to look the same on other platforms and video systems, and the AWT is there to help you out. To do so, it needs a couple of bits of information. Every component has something called the `preferredSize`. This is the size you would like your component to have, if at all possible. What is possible will depend entirely on the configuration of the system your Java code is running on, and what layout policy is in effect. Some layout managers keep components their preferred size. Others, like the `BorderLayout` manager, change the dimensions of components, but use the preferred sizes as starting points. A component positioned as a `NORTH` item under a `BorderLayout` policy will be given its preferred height, but the width will be set to the width of the parent container. Layout managers look at the `preferredSize` of each component being laid out, and follow the selected layout policy. The preferred size of a component is set by overriding the `preferredSize` method like this:

```
public Dimension preferredSize() {
    return new Dimension(180, 120);
  }
```

You will normally override the `preferredSize` method only for containers, like Panels. Child controls, like buttons and list boxes, have preferred sizes derived typically from the size of their peer components.

Layout Constraints

Laying out controls in a container is not always trivial. Layout managers are often faced with situations in which there are several ways to accomplish a given task. You tell the manager exactly how you want it to determine the layout using *constraints*, which are essentially attributes that you can assign values to. Constraints always have default values, so you don't have to access and set them unless you want a non-default layout. Example: the `BorderLayout` manager puts components either along a border of a container, or in the center. Choosing a `BorderLayout` policy by itself is not sufficient to get a button to be positioned along the top border. You must specify a `North` constraint to achieve the desired results. Constraints are used by all layout managers, but often the defaults are what you want, so you don't need to change them. Constraints are used to determine the spacing between components, the margin between components and the container border, how to distribute left-over space, and so on. Most layout managers have simple constraints, since they support only a limited number of options. The exception is `GridBagLayout`, which requires you to fully understand constraints in general, and `GridBagConstraints` in particular.

The Built-in Layout Managers

There are several common types of layout used in applications, and the AWT contains layout managers for them. The 4 basic managers are `BorderLayout`, `CardLayout`, `FlowLayout` and `GridLayout`. The fifth, `GridBagLayout`, is a sort of catch-all manager and is considerably more complicated to use than the others. The managers form a simple hierarchy, as shown in Figure 2.

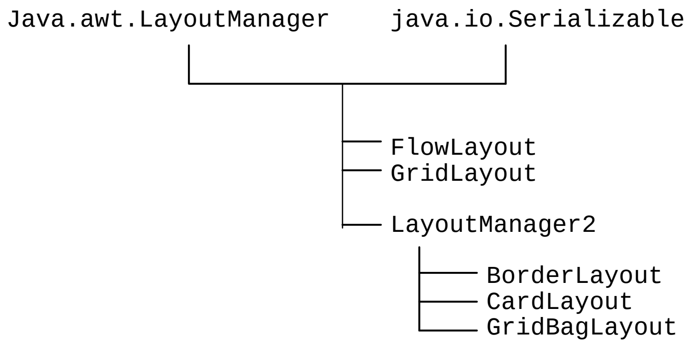


Figure 2 - The hierarchy of built-in Java Layout Managers.

All the built-in layout managers are serializable, meaning they know how to save and restore their internal state. Saving and later restoring a panel that uses a built-in layout manager will recreate the layout that was saved.

BorderLayout

This layout manager is widely used in Java programs, and lets you position a component relative to one of the borders of the parent container, or in the center. The layout options are `North`, `East`, `South` and `West` for border-relative components, and `Center` for components that occupy the remaining client area left over by border-relative components. The layout options are strings. To create a child component that sticks to the top border and has the same width as the parent component, you would use code like this:

```
add("North", new java.awt.Button() );
```

To make a component occupy the client area of the parent, you give it the `Center` option, like this:

```
add("Center", new java.awt.List() );
```

`BorderLayout` is the default policy for a number of Java containers, such as `java.awt.Window` and `java.awt.Frame`. Delphi programmers are familiar with `BorderLayout` policies, because most Delphi components have an `Align` property that lets you specify the layout option (`None`, `Left`, `Top`, `Right`, `Bottom`, `Client`) at design time. The Delphi border alignment is slightly different from the Java `BorderLayout`, because the former lets you “stack” objects against a border. For example if you add three buttons with `Top` alignment, the first button will be attached to the top border, the second will be attached to the bottom edge of the first, the third to the bottom edge of the second. The Java `BorderLayout` manager doesn’t allow this. You can only align one component on any given border. If you added two `North` buttons, the second would cover the first.

The following figure shows 5 buttons positioned using a `BorderLayout` policy. Notice how the components maintain their relative position correctly after the parent window is resized.

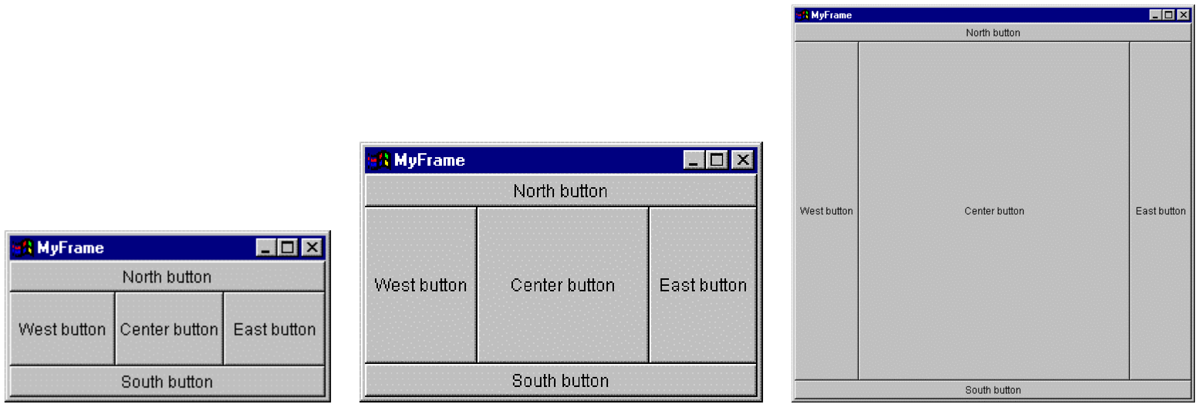


Figure 3 – How a BorderLayout manager lays out controls in a resized window.

Here is the code that produced the window in Figure 3.

```
public class MyFrame extends java.awt.Frame {

  java.awt.BorderLayout borderLayout = new java.awt.BorderLayout();

  java.awt.Button button1 = new java.awt.Button();
  java.awt.Button button2 = new java.awt.Button();
  java.awt.Button button3 = new java.awt.Button();
  java.awt.Button button4 = new java.awt.Button();
  java.awt.Button button5 = new java.awt.Button();

  //Construct the frame
  public MyFrame() {

    this.setLayout(borderLayout);
    this.setTitle("MyFrame");

    button1.setLabel("North button");
    button2.setLabel("East button");
    button3.setLabel("South button");
    button4.setLabel("West button");
    button5.setLabel("Center button");

    add("North", button1);
    add("East", button2);
    add("South", button3);
    add("West", button4);
    add("Center", button5);
  }
}
```

CardLayout

Although the name may be a bit misleading, `CardLayout` managers are commonly used. They control the visibility of different AWT containers that are kept in a sort of stack. To use a `CardLayout`, you create an AWT container and assign the `CardLayout` to it, like this:

```
java.awt.Panel deckPanel = new java.awt.Panel();
java.awt.CardLayout cardLayout = new java.awt.CardLayout();
deckPanel.setLayout(cardLayout);
```

Then you fill the deck by creating AWT containers and adding them to the deck like this:

```
java.awt.Panel panel1 = new java.awt.Panel();
deckPanel.add("Card 1", panel1);
```

When you add a card, you also give it a string tag that you can use later to select that card. You can make a specific card visible using the code

```
cardLayout.show(deckPanel, “Card 1”);
```

The first parameter is the container you want the `CardLayout` to work with. The second is the `String` tag of the card you want to select. You can also show the first or last card in the deck, using the code

```
cardLayout.first(deckPanel);
cardLayout.last(deckPanel);
```

or just move forwards or backwards by one card with the code:

```
cardLayout.next(deckPanel);
cardLayout.previous(deckPanel);
```

The number of cards in the deck can be obtained using the code:

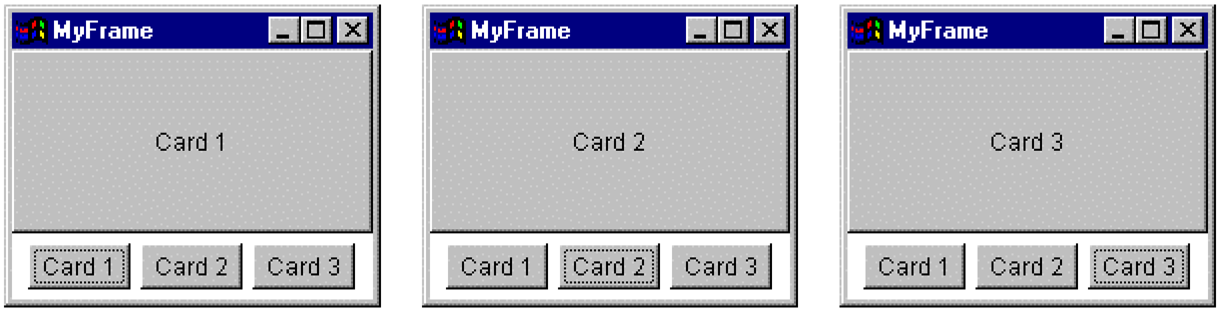
```
int cardCount = deckPanel.countComponents();
```

Basically a `CardLayout` achieves the same end as a Windows multi-page Property Sheet, except that no tabs are displayed. The name *`CardLayout`* was chosen since the panels resemble a stack of playing cards.

The following code creates a 3-card panel controlled by a `CardLayout` manager. Each card has a button that fills its client area, with a label denoting the card number. At the bottom is a panel containing buttons that let you select which card is visible. You could easily change the example to show a list box in the bottom panel. Each item in the list box could be associated with a different card.

```
public class MyFrame extends java.awt.Frame {

  // create the card deck with three page-control buttons
  java.awt.CardLayout cardLayout = new java.awt.CardLayout();
  java.awt.Panel deckPanel = new java.awt.Panel();
  java.awt.Panel buttonPanel = new java.awt.Panel();
  java.awt.Button buttonCard1 = new java.awt.Button();
  java.awt.Button buttonCard2 = new java.awt.Button();
  java.awt.Button buttonCard3 = new java.awt.Button();

  // create cards for the deck
  java.awt.Panel panel1 = new java.awt.Panel();
  java.awt.Button button1 = new java.awt.Button();

  java.awt.Panel panel2 = new java.awt.Panel();
  java.awt.Button button2 = new java.awt.Button();
  java.awt.Panel panel3 = new java.awt.Panel();
  java.awt.Button button3 = new java.awt.Button();

  //Construct the frame
  public MyFrame() {

    setLayout(new java.awt.BorderLayout() );
    setTitle("MyFrame");

    deckPanel.setLayout(cardLayout);
    add("Center", deckPanel);
    add("South", buttonPanel);

    buttonPanel.setLayout(new java.awt.FlowLayout() );

    buttonCard1.setLabel("Card 1");
    buttonPanel.add(buttonCard1);

    buttonCard2.setLabel("Card 2");
    buttonPanel.add(buttonCard2);

    buttonCard3.setLabel("Card 3");
    buttonPanel.add(buttonCard3);

    button1.setLabel("Card 1");
    panel1.setLayout(new java.awt.BorderLayout() );
    panel1.add("Center", button1);
    deckPanel.add("Card 1", panel1);

    button2.setLabel("Card 2");
    panel2.setLayout(new java.awt.BorderLayout() );
    panel2.add("Center", button2);
    deckPanel.add("Card 2", panel2);

    button3.setLabel("Card 3");
    panel3.setLayout(new java.awt.BorderLayout() );
    panel3.add("Center", button3);
    deckPanel.add("Card 3", panel3);
  }

  public boolean action(java.awt.Event theEvent, Object theSource) {
    if (theEvent.target == buttonCard1)
      cardLayout.show(deckPanel, buttonCard1.getLabel() );
    else if (theEvent.target == buttonCard2)
      cardLayout.show(deckPanel, buttonCard2.getLabel() );
    else if (theEvent.target == buttonCard3)
      cardLayout.show(deckPanel, buttonCard3.getLabel() );
    return true;
  }
}
```

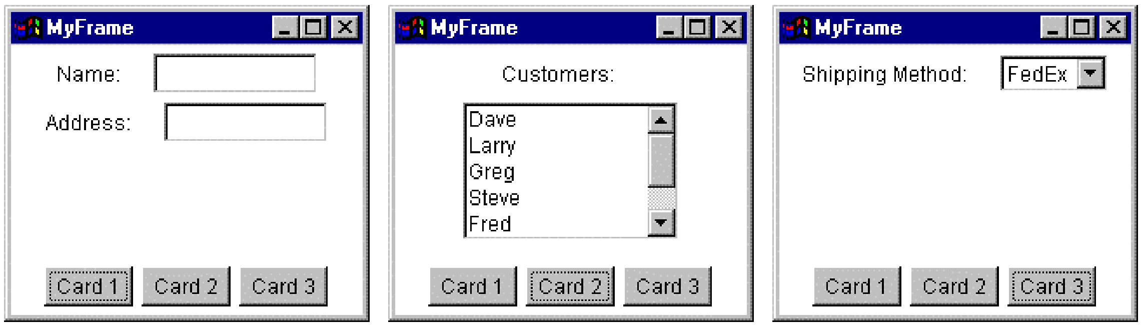
The code produces the output shown in Figure 4, which shows the results of clicking the Card 1, the Card 2 and the Card 3 buttons.



A – Card1 B – Card 2 C – Card 3

Figure 4 - Using a `CardLayout` to switch between cards.

Although my example shows each card with a simple button, you can put as many components on each as you need, with each card being completely independent of what is on the other cards. For example the 3 cards might look like this:



A – Card 1 B – Card 2 C – Card 3

Figure 5 - A deck of cards with differing components and layouts.

Each card in a `CardLayout` deck is a container – usually a `java.awt.Panel` or descendent. Other than being part of the same deck, there is no other relationship between the cards in a deck. They may each have there own (different) layout manager, so card 1 might use `FlowLayout`, card 2 `CardLayout` and card 3 a custom layout.

FlowLayout

This is the default layout policy `for java.awt.Panel` components. This manager lays the components of a container out in *flow* order. Starting from the top of the container, the controls are positioned from left to right in rows. When there is not enough space to display the next component, the manager moves below the first row of components and continues the flow. The process repeats until all components have been positioned. Components may be clipped from display if they *fall off the end* of the parent container. The controls on each row are centered horizontally by default. You can also have the layout manager left- or right-align components by passing the values `FlowLayout.LEFT` or `FlowLayout.RIGHT` to the class constructor. The following figures show how components are positioned, based on the space available.

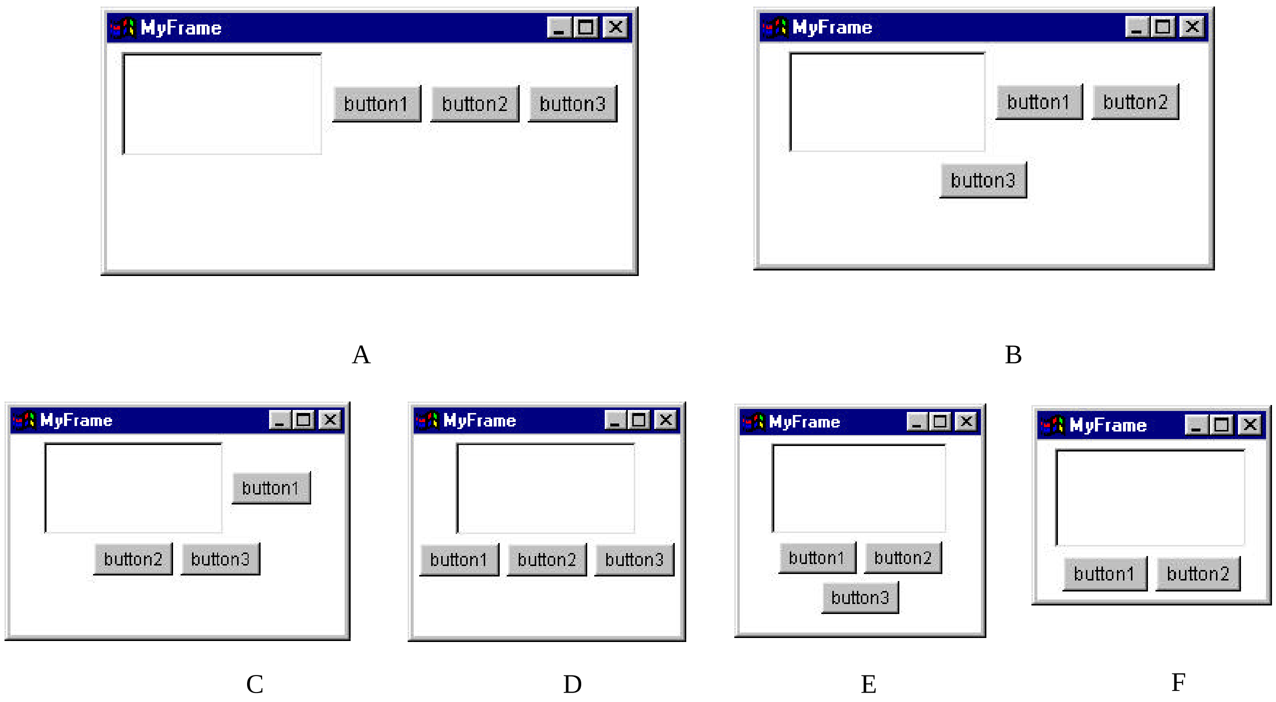


Figure 6 - How a `FlowLayout` policy positions controls based on the space available.

The figure shows a list box followed by 3 buttons. When the manager runs out of room, as shown in the Figure 6.B, the manager moves down to the next row. This row starts below the lowest component in the previous row. Notice how the components in all the rows are horizontally centered, because the code didn't specify an alignment mode. Figure 6.F shows the result of components falling off the end. There is no indication in the parent frame that some of its children were clipped.

Here is the code that produced the frame window shown in Figure 6.

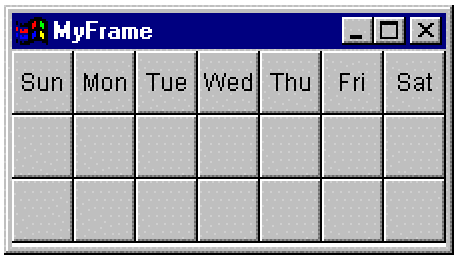
```
public class MyFrame extends java.awt.Frame {

  java.awt.FlowLayout flowLayout = new java.awt.FlowLayout();
  java.awt.List list1 = new java.awt.List();
  java.awt.Button button1 = new java.awt.Button();
  java.awt.Button button2 = new java.awt.Button();
  java.awt.Button button3 = new java.awt.Button();

public MyFrame() {

    this.setLayout(flowLayout);
    this.setTitle("MyFrame");
    button1.setLabel("button1");
    button2.setLabel("button2");
    button3.setLabel("button3");
    add(list1);
    add(button1);
    add(button2);
    add(button3);
  }
}
```

GridLayout

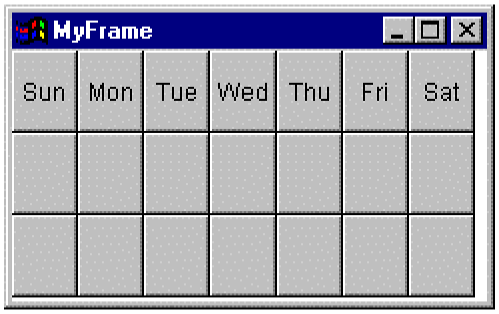
There are times when you have a form that needs to lay items out by row and column, such as when displaying a calendar. In these cases you want to be able to specify in advance the number of columns in each row, because your data makes sense only when shown with the given number of columns. A `FlowLayout` won't suffice, because the layout manager will place whatever components it can fit on each row, based on the window width. A `GridLayout` manager solves this type of problem with ease. When you create a `GridLayout`, you tell it the number of rows and columns you want, like this:

```
GridLayout myGrid = new GridLayout(3, 7);
```

Where the first parameter is the number of rows, the second the columns. Using these two numbers, the layout manager divides the area available of the window it is handling into equal-sized cells. Figure 7 shows two frames laid out by a `GridLayout`.



A – No left over space



B – A grid with some left-over space

Figure 7 - A 7 by 3 grid of buttons.

Because a `GridLayout` makes all the cells the same size, there is a possibility of left over space on the right and bottom borders. The space corresponds to the remainder you get when you divide the window width by the number of columns, and conversely for the rows. Figure 7.B shows a grid with some left-over space. If left-over space is unacceptable, you can snap the window size to the correct size by overriding the `java.awt.Container.layout` method and computing the nearest allowable width and height, like this:

```
public void layout() {
  Dimension d = getSize();
  Insets insets = insets();
  int xMargins = insets.left + insets.right;
  int yMargins = insets.top + insets.bottom;
  int width = (int) ( (d.width - xMargins) / 7) * 7 + xMargins;
  int height = (int) ( (d.height - yMargins) / 3) * 3 + yMargins;
  resize(width, height);
  super.layout();
}
```

Here is the code used to create Figure 7.

```
public class MyFrame extends java.awt.Frame {

  public MyFrame() {

    setLayout(new java.awt.GridLayout(3, 7) );
    setTitle("MyFrame");
    add(new java.awt.Button("Sun") );
    add(new java.awt.Button("Mon") );
    add(new java.awt.Button("Tue") );
    add(new java.awt.Button("Wed") );
    add(new java.awt.Button("Thu") );
    add(new java.awt.Button("Fri") );
    add(new java.awt.Button("Sat") );

    for (int i = 0; i < 14; i++)
      add(new java.awt.Button() );
  }

  public boolean handleEvent(java.awt.Event theEvent) {
    if (theEvent.id == java.awt.Event.WINDOW_DESTROY)
      System.exit(0);
    return false;
  }
}
```

When laying out the cell components in the grid, the `GridLayout` manager ignores the preferred sizes of the components. All the cells are the same size, and the size depends only on the parent window size and the number of cells.

GridBagLayout

The AWT designers provided simple classes to support the most common layout policies. When you have a layout that can't be handled using the conventional managers discussed so far, you have only two options: create your own manager or use `GridBagLayout`. A lot of programmers get confused by `GridBagLayout` and would almost prefer creating their own manager rather than use it. Although this manager is definitely more complex than the others, with a little practice you can get the hang of what it's doing and how to control it.

Because `GridBagLayout` is intended to be very flexible, it requires a lot of options to work. You tell it those options using a full-fledged `GridBagConstraints` object that has the following fields.

Field	**Description**
`anchor`	This option is used when the size of a component is smaller than the grid cell created for it. It tells the layout manager which corner, point or border you want the component positioned by. The default is `CENTER`, meaning the component is centered vertically and horizontally inside the cell. Possible values are `NORTH`, `NORTHEAST`, `EAST`, `SOUTHEAST`, `SOUTH`, `SOUTHWEST`, `WEST`, `NORTHWEST` and `CENTER`. Choosing a corner like `NORTHWEST` places the component's top left corner on that point. Choosing a border, like `NORTH`, causes the component to be placed on the top edge of the bounding cell and centered horizontally in the cell
`fill`	This option tells the manager how you want the component to fill its grid cell if the cell size is different from the component's preferred size. For example if the component is wider than the cell, you can make the manager shrink the component horizontally to the exact cell size using the option `HORIZONTAL`. If the component were narrower than the cell, the manager would expand it. In either case, the vertical size wouldn't be affected. The `VERTICAL` options causes resizing in the vertical direction. The value `NONE` causes the component size to be left alone. If the component is smaller than the cell, empty space is left around it. If the component is larger than the cell, it is clipped. Using the option `BOTH` causes the manager to resize the

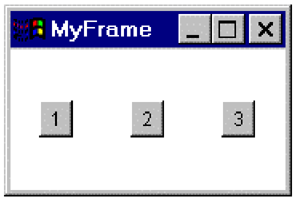
	component to fit exactly in the cell. See Figure 8 for examples.
`gridwidth` `gridheight`	These two options can be a bit confusing, possibly because of their names. Maybe `numberOfCellsX` and `numberOfCellsY` would have been better choices. These options control the number of grid cells a given component occupies. The `GridBagLayout` manager lays out a rectangular area divided into cells. By default each component occupies a single cell. Setting `gridWidth` and `gridHeight` to 2 or higher will cause the associated component to spread over 2 or more cells in the X or Y direction. The default value is 1. Two other important values you can use are `RELATIVE`, which makes a component the next-to-last item in its row or column, and `REMAINDER`, which makes the component the last item in its row or column. See Figure 9 for examples.
`gridx` `gridy`	The cells in a grid are numbered. The leftmost cell in a row has `gridx` = 0. The cell at the beginning (top) of a column has `gridy` = 0. As the `GridBagLayout` manager is laying out cells, it assigns items to cells, based on the options you set. You can also force an item to be in a particular row or column by setting `gridx`/`gridy`. The default value is `RELATIVE`, but assigning a non-negative value to it tells the layout manager you want an item in a specific place. See Figure 10 for examples.
`insets`	Insets are the space the layout manager places around each component in its cell. Adding an inset doesn't make a component smaller: it makes the cell larger. The default insets are `Insets(0, 0, 0, 0)`. The parameters specify the margin for the top, left, bottom and right borders. See Figure 11 for examples.
`ipadx` `ipady`	You can make individual components larger by padding them. Padding makes the layout manager add space in the x or y direction. See Figure 12 for examples.
`weightx` `weighty`	Determines how space is allocated to rows and columns. For rows/columns with 0 weight, the manager makes cells based on their preferred sizes. For larger weights, the manager adds space using a simple algorithm. It computes the ratio of the cell's weight to the sum of all the weights in the same row and column. Cells with higher weights will get more space than other cells. See Figure 14 for examples.

Table 1 - The fields of `GridBagConstraints`.

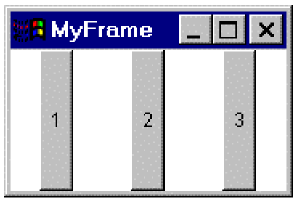
You specify the layout constraints when you add a component to a container, using code like this:

```
GridBagLayout layout = new GridBagLayout();
GridBagConstraints constraints = new GridBagConstraints();
Button button = new Button("Label");
constraints.weightx = 1;
constraints.gridwidth = GridBagConstraints.REMAINDER;
layout.setConstraints(button, constraints);
add(button);
```

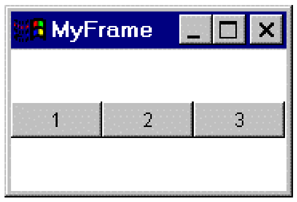
The following figure shows how the `fill` option affects the layout of components.



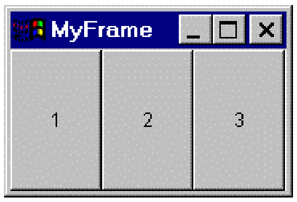
A
`fill` = `NONE`



B
`fill` = `VERTICAL`



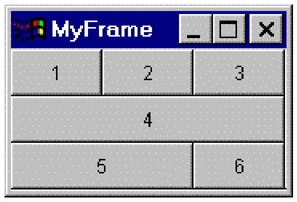
C
`fill` = `HORIZONTAL`



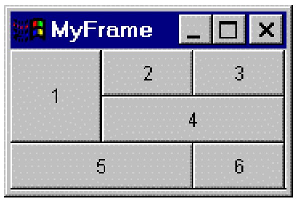
D
`fill` = `BOTH`

Figure 8 - The effect of changing the `fill` option.

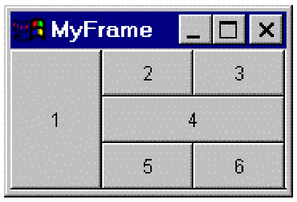
In Figure 8, the `weightx`/`weighty` options were set to 1, to better show the grid cell sizes. The next figure shows how the `gridwidth` and `gridheight` options affect the layout.



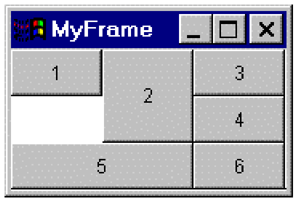
A
`All gridheights` = 1



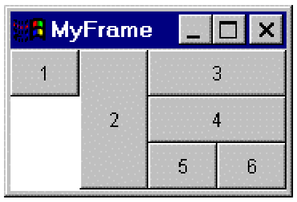
B
Button 1:
`gridheight` = 2



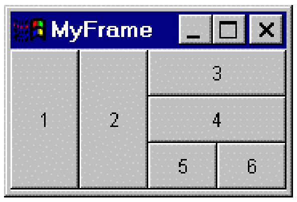
C
Button 1:
`gridheight` >= 3



D
Button 2:
`gridheight` = 2



E
Button 2:
`gridheight` = 3

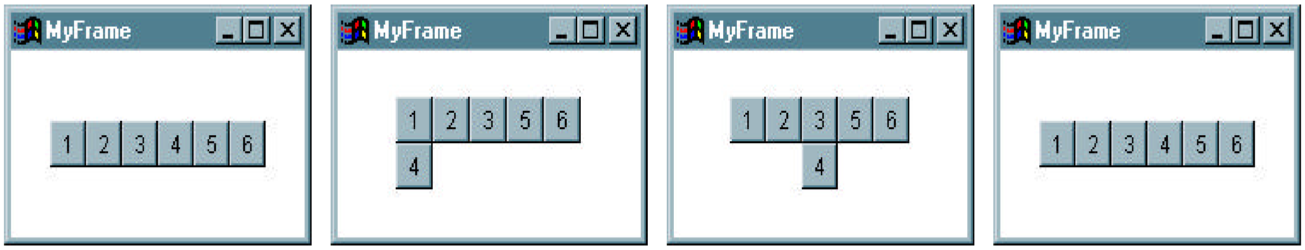


E
Button 1 & 2:
`gridheight` = 3

Figure 9 - The effect of changing the `gridheight` option.

In Figure 9 the `weightx` and `weighty` options were set to 1. The fill was set to `BOTH`.

The next figure shows how the layout is affected by the `gridx` and `gridy` options.

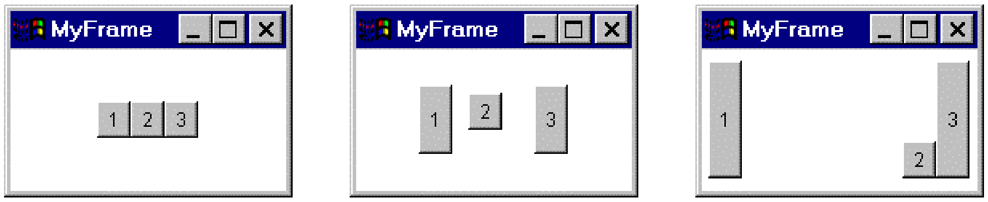


A `gridx` = `RELATIVE` B button 4: `gridx` = 0 C Button 4: `gridx` = 2 D Button 4: `gridx` > 2

Figure 10 - The effect of changing the `gridx` of button 4.

In Figure 10, the component `fill` constraints are set to BOTH, to better show the grid cell sizes. Figure 10.A shows a series of buttons laid out using the default `gridx` option, which is `GridBagOptions.RELATIVE`. Figure 10.B shows how button 4 is positioned after setting its `gridx` option to 0. Notice that the layout manager placed it on the next row. When you specify a grid value that is smaller than the current cell, the layout manager places the item on the next row or column. Figure 10.D shows button 4 with a `gridx` value of 3. Setting the `gridx` value of an item to a value greater than or equal to the current cell makes the manager place the item on the current row, as shown in Figure 10.D.

The next figure shows how the `inset` option affects the layout.



A Button 2: Insets(0, 0, 0, 0) B Button 2: Insets(5, 10, 15, 20) C Button 2: Insets(50, 100, 0, 0)

Figure 11 - The effect of changing a component's insets.

In Figure 11, the component `fill` constraints are set to `BOTH`, to better show the grid cell sizes. Because the values of an inset contribute toward cell size, changing one component’s insets can affect the size of other components on the same row or column. Just by looking at Figure 11.C, it is hard to imagine that it differs from Figure 11.A only in the value of a single cell inset.

The two `GridBagConstaints` options `ipadx` and `ipady` can be used to increase the size of component cells. If you set the `fill` option to `BOTH`, the components are inflated to fill the cell, making the cell size visible. Figure 12 shows 3 buttons using different padding values. The next figure shows how the `ipadx` and `ipady` options affect the layout.

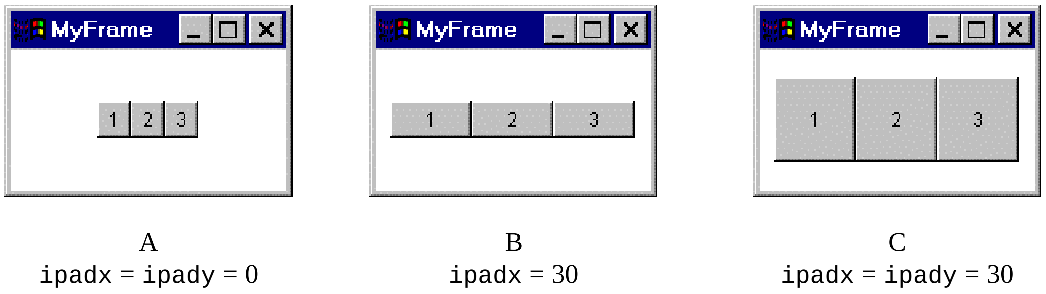


Figure 12 - The effect of changing `ipadx/ipady`.

To make the cell size visible, the `fill` option in Figure 12 was set to `BOTH`.

As for insets, padding a cell on one row can affect other cells on the same row or column, as shown in the next figure.

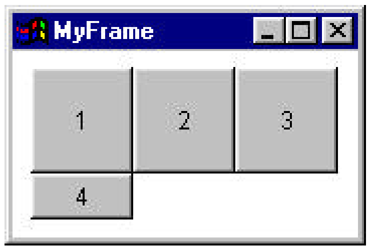
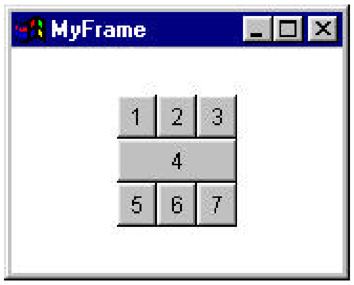


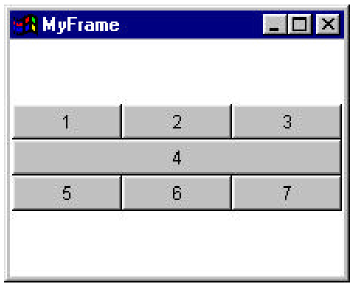
Figure 13 - How changing the padding of one row can affect other rows.

The components on the first row have x and y padding of 30. The component on the second row has x and y padding of 0. All components have `fill` set to `BOTH`, to make the cell size obvious. Because the first row sets x padding to 30, all cells in the first column will be at least as wide. The cell on the second row would have been narrower, had no padding been applied to the cell above it.

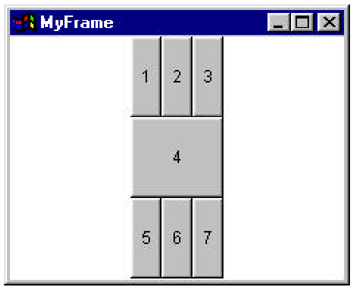
The next figure shows how the layout is affected by changing `weightx` and `weighty`.



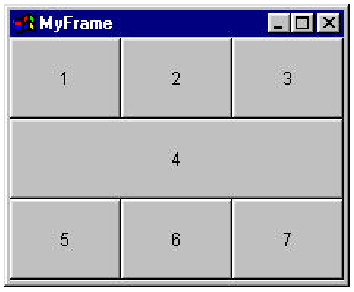
A
`All rows: weights = 0`



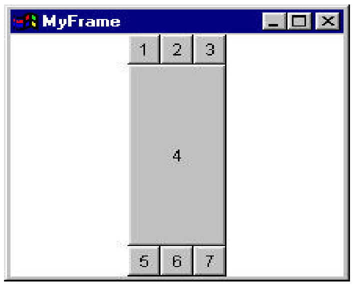
B
All rows: `weightx` = 1



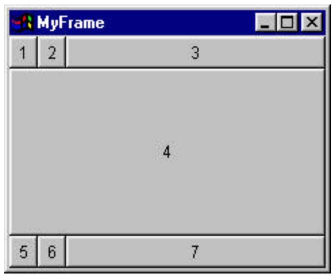
C
All rows: `weighty` = 1



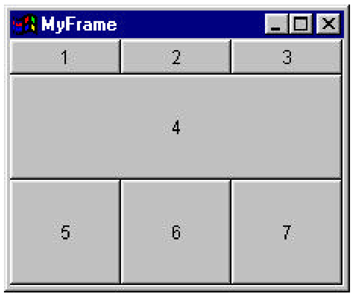
D
All rows: weights = 1



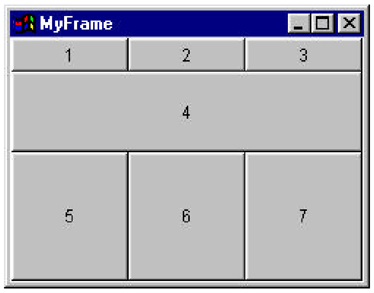
E
Row 2: `weighty` = 1



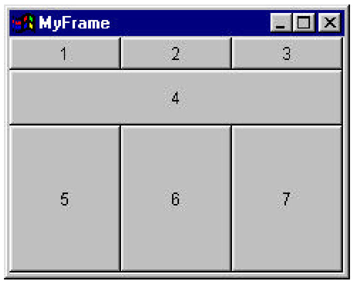
F
Row 2: Both weights = 1



G
Row 2 weighty = 1
Row 3 weighty = 1



H
Row 2 weighty = 1
Row 3 weighty = 3



I
Row 2 weighty = 1
Row 3 weighty = 5

Figure 14 - The effect of the `weightx`/`weighty` `GridBagConstraints` options.

In Figure 14, the component `fill` constraints are set to `BOTH`, to better show the grid cell sizes. As you can see, changing weights can have a great effect on the layout.

Rolling your own manager

It's crunch time. You have this really cool layout, but can't seem to support it with one of the built-in managers. Playing with `GridBagLayout` doesn't quite get what you want. Before going to the effort of creating a brand new layout manager of your own, you should investigate whether a series of nested panels will solve the problem. Assuming it doesn't, it's coding time.

The first step is to implement the `java.awt.LayoutManager` interface and implement its 5 basic methods, like this:

```
public class MyLayoutManager implements java.awt.LayoutManager {

  public void addLayoutComponent(String theString, Component theComponent) {}
  public void removeLayoutComponent(Component theComponent) {}
  public Dimension preferredLayoutSize(Container theContainer) {}
  public Dimension minimumLayoutSize(Container theContainer) {}
  public void layoutContainer(Container theContainer) {}
}
```

You may or not want to also implement the `java.io.Serializable` interface to make your layout manager persistent. The next step is to decide how to code the 5 methods. Obviously that will depend on the type of layout policy you need. Let's say you want a manager that always lays components out vertically, like this:

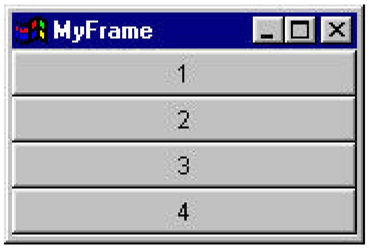


Figure 15 - Laying components out vertically with a custom layout manager.

The first two methods, `addLayoutComponent` and `removeLayoutComponent` are used if you want to use your own fields to store the objects added to a layout. This is often not necessary, because a layout manager can get the list of components from the container that calls `layoutContainer`. You may want to keep your own list if you have a special parameters that control layout. For example, a `BorderLayout` uses the parameters "`NORTH`", "`EAST`", "`SOUTH`", "`WEST`" and "`CENTER`" to position components and uses a separate field to store components for each border. Your layout manager can use any strings it wants as options, including empty strings. Class `BorderLayout` implements `addLayoutComponent` like this:

```
public void addLayoutComponent(String name, Component comp) {

      /* Special case:  treat null the same as "Center". */
      if (name == null) {
          name = "Center";
      }

      /* Assign the component to one of the known regions of the layout.
       */
      if ("Center".equals(name)) {
          center = comp;
      } else if ("North".equals(name)) {
          north = comp;
      } else if ("South".equals(name)) {
          south = comp;
```

```
        } else if ("East".equals(name)) {
            east = comp;
        } else if ("West".equals(name)) {
            west = comp;
        } else {
            throw new IllegalArgumentException("cannot add to layout: “ +
                                          “unknown constraint: " + name);
        }
    }
```

My simple vertical layout manager will use no placement strings, so it will not override `addLayoutComponent` or `removeLayoutComponent`.

The two methods `minimumLayoutSize` and `preferredLayoutSize` calculate the minimum and ideal size of the container. For my simple example, the minimum size is the greatest of the minimum sizes of the components, plus the parent container’s insets. The preferred size is the greatest of the preferred sizes of the components, plus the parent container’s insets.

The bulk of the work in any layout manager is typically in the `layoutContainer` method. `MyLayoutManager` simply positions each component so the left side in indented by the container’s left inset. The size is set so the right edge of each component extends to the right border of the container minus the right inset. The layout manager maintains the preferred heights of all the components. The implementation of `MyLayoutManager` looks like this:

```
import java.awt.*;

class MyLayoutManager implements LayoutManager{

  public void addLayoutComponent(String theString, Component theComponent) {}
  public void removeLayoutComponent(Component theComponent) {}

  public Dimension preferredLayoutSize(Container theContainer) {

    int width = 0;
    int height = 0;
    int componentCount = theContainer.countComponents();
    Insets insets = theContainer.insets();

    for (int i = 0; i < componentCount-1; i++) {
      Component component = theContainer.getComponent(i);
      width = Math.max(width, component.preferredSize().width);
      height += component.preferredSize().height;
    }

    return new Dimension(insets.left + width + insets.right,
                         insets.top + height + insets.bottom);
  }

  public Dimension minimumLayoutSize(Container theContainer) {
    int width = 0;
    int height = 0;
    int componentCount = theContainer.countComponents();
    Insets insets = theContainer.insets();

    for (int i = 0; i < componentCount-1; i++) {
      Component component = theContainer.getComponent(i);
      width = Math.max(width, component.minimumSize().width);
      height += component.minimumSize().height;
    }

    return new Dimension(insets.left + width + insets.right,
                         insets.top + height + insets.bottom);
  }
  public void layoutContainer(Container theContainer) {

    Insets insets = theContainer.insets();
    int x = insets.left;
    int y = insets.top;
    int width = theContainer.size().width - insets.left - insets.right;

    for (int i = 0; i < theContainer.countComponents(); i++) {
      Component component = theContainer.getComponent(i);
      int height = component.preferredSize().height;
      component.reshape(x, y, width, height);
      y += height;
    }
  }
}
```

The manager is quite simple, but still useful. Of course a vertical layout could have been achieved using a `GridBagLayout`, but I wanted to keep the layout policy simple to keep the discussion focused. If your layout manager has any fields that aren't initialized in the constructor, you should seriously consider making the class `Serializable`, so you can save and restore the class from persistent storage.

Conclusion

Often Windows programmers find layout managers to be a initially something of a nuisance. Instead of just dropping controls on a form, you have to set up a certain amount of infrastructure required by layout managers. After some experience with the AWT, programmers come to realize the importance of centralized layout policies. Under Windows, dialog boxes are generally not resizable, because controls are laid out in a static pre-arranged order. If you enlarge a dialog box, you just get empty space along the right and bottom borders. If you shrink one, you clip off some of the controls. With Windows, laying out components in a parent-window-size-dependent manner requires lots of coding and many applications devote pages of code just to solve layout problems.

Layout managers take the pain out of delivering platform-independent interfaces. Even a complicated layout can often be created using nested panels, each with its own layout manager. Layout managers allow users to resize forms while preserving spatial relationships between the components on them. Because layout is a mundane and essential issue in user interfaces, the AWT provides built-in layout managers that solve most of the presentation situations applications deal with. The built-in managers were designed to be extensible with little coding. Creating your own layout manager is as simple as implementing 5 methods in a custom layout class. Layout managers centralize the code that deals with dynamic positioning and sizing of components, allowing you to focus on the particular task at hand. Hopefully layout managers will help you not only develop better user interfaces, but using less code, with the bonus that they will look good on any machine.