Tips 'N Tricks

Java Tip 121: Flex your grid layout

Extend java.awt.GridLayout to allow for multisized columns and rows

Summary
Unfortunately, the JDK does not include a `GridLayout` class that allows for a matrix layout with multisized cells (rows of different heights or columns of different widths). To obtain that type of layout, developers usually use `GridBagLayout`, creating long and cluttered code as they add components into a container. This tip describes a simple `GridLayout` extension that helps programmers write less and more readable code. (*1,500 words;* **December 14, 2001**)

By Bogdan Dorohonceanu

Developers often need to create graphical user interfaces (GUIs) that have a matrix-type layout with columns of different widths or rows of different heights. Those layout cells are unequal in order to minimize the space occupied by the laid out components. In those cases, you have to use the `GridBagLayout` class, whose numerous `GridBagConstraints` settings cause your code to grow quickly. However, the `GridLayout` layout manager seems more appropriate. It can help you write short and readable code while laying out components in equally sized cells. In this tip, I extend `GridLayout` to create cells of unequal size.

Note: If you use a JTable to display information in a matrix, then check out "Java Tip 116: Set Your Table Options -- at Runtime!" Sonal Goyal with John D. Mitchell.

GridLayout2

The `GridLayout2` extension is fairly simple. The `GridLayout2` class inherits from `java.awt.GridLayout` and then redefines the public methods `preferredLayoutSize()`, `minimumLayoutSize()`, and `layoutContainer()` to account for multisized grid cells. I started from the source code of those methods in the `java.awt.GridLayout` class. (Download the complete source code in Resources.)

Set preferred size

Here is the `preferredLayoutSize()` method:

```
public Dimension preferredLayoutSize(Container parent) {
  synchronized (parent.getTreeLock()) {
    Insets insets = parent.getInsets();
    int ncomponents = parent.getComponentCount();
    int nrows = getRows();
    int ncols = getColumns();
    if (nrows > 0) {
      ncols = (ncomponents + nrows - 1) / nrows;
    }
    else {
      nrows = (ncomponents + ncols - 1) / ncols;
    }
    int[] w = new int[ncols];
    int[] h = new int[nrows];
    for (int i = 0; i < ncomponents; i ++) {
      int r = i / ncols;
      int c = i % ncols;
      Component comp = parent.getComponent(i);
      Dimension d = comp.getPreferredSize();
      if (w[c] < d.width) {
        w[c] = d.width;
      }
      if (h[r] < d.height) {
```

```
        h[r] = d.height;
      }
    }
    int nw = 0;
    for (int j = 0; j < ncols; j ++) {
      nw += w[j];
    }
    int nh = 0;
    for (int i = 0; i < nrows; i ++) {
      nh += h[i];
    }
    return new Dimension(insets.left + insets.right +
                nw + (ncols-1) * getHgap(),
                insets.top + insets.bottom +
                nh + (nrows-1)*getVgap());
  }
}
```

As you can see, the code is pretty straightforward. You first ensure that you have the right number of rows and columns to lay out the components. Then you find each component's preferred size. Finally, you compute each row's height as the row components' maximum height. You compute each column width as the maximum width of the row components. The preferred layout size also accounts for the horizontal and vertical gap sizes between components and the parent container insets.

Set minimum size
The code for `minimumLayoutSize()` is basically the same as `preferredLayoutSize()`, except you use the subcomponents' minimum size dimensions.

Set container size
The `layoutContainer()` method that follows accounts for each subcomponent's preferred size. The code also scales the component according to the parent container size, horizontal and vertical gap size, and insets:

```
public void layoutContainer(Container parent) {
  synchronized (parent.getTreeLock()) {
    Insets insets = parent.getInsets();
    int ncomponents = parent.getComponentCount();
    int nrows = getRows();
    int ncols = getColumns();
    if (ncomponents == 0) {
      return;
    }
    if (nrows > 0) {
      ncols = (ncomponents + nrows - 1) / nrows;
    }
    else {
      nrows = (ncomponents + ncols - 1) / ncols;
    }
    int hgap = getHgap();
    int vgap = getVgap();
     // scaling factors
    Dimension pd = preferredLayoutSize(parent);
    double sw = (1.0 * parent.getWidth()) / pd.width;
    double sh = (1.0 * parent.getHeight()) / pd.height;
    // scale
    int[] w = new int[ncols];
    int[] h = new int[nrows];
    for (int i = 0; i < ncomponents; i ++) {
      int r = i / ncols;
      int c = i % ncols;
      Component comp = parent.getComponent(i);
      Dimension d = comp.getPreferredSize();
      d.width = (int) (sw * d.width);
      d.height = (int) (sh * d.height);
      if (w[c] < d.width) {
        w[c] = d.width;
      }
      if (h[r] < d.height) {
        h[r] = d.height;
      }
    }
    for (int c = 0, x = insets.left; c < ncols; c ++) {
```

```
        for (int r = 0, y = insets.top; r < nrows; r ++) {
          int i = r * ncols + c;
          if (i < ncomponents) {
            parent.getComponent(i).setBounds(x, y, w[c], h[r]);
          }
          y += h[r] + vgap;
        }
        x += w[c] + hgap;
      }
    }
  }
```

Again, you first ensure you have the right number of rows and columns to lay out the components. Then you compute the scaling factors on `x` and `y` coordinates as the ratios between the parent container's current height and width and the height and width of its preferred layout size. You find each component's preferred size and scale it using the scaling factors. You also compute each row's height as the maximum height of the scaled components in the row. You compute each column's width as the maximum width of the scaled components in the row. Finally, you lay out the components, considering the horizontal and vertical gap sizes between components and the parent container insets.

You could also scale the horizontal and vertical gap sizes. I will leave that as an exercise for the reader.

Compare GridLayout and GridBagLayout to GridLayout2

Figures 1, 2, and 3 show a random arrangement of 14 multisized `JLabels` using the three grid layouts: `GridLayout`, `GridBagLayout`, and `GridLayout2`. Figure 3 shows the desired view.

Figure 1. Arrange 14 multisized JLabels with GridLayout layout manager

Figure 2. Arrange 14 multisized JLabels with GridBagLayout layout manager

Figure 3. Arrange the same 14 labels with GridLayout2 layout manager

As you can see, `GridLayout` sizes all matrix cells equally; `GridBagLayout` allows for each column/row cell to have its own width/height; and `GridLayout2` looks the same as `GridBagLayout` without the extra baggage and hassle.

Create a compact panel

Figures 4, 5, and 6 show a sample password panel using the same three grid layouts: `GridLayout`, `GridBagLayout`, and `GridLayout2`. As the above exercise, Figures 5 and 6 show the desired view: a compact change-password panel that doesn't consume as much screen space as Figure 4.

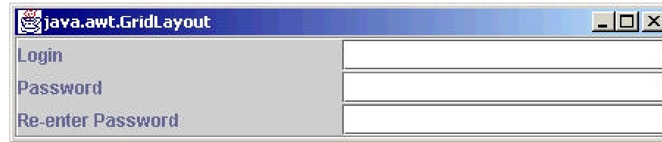


Figure 4. Sample change-password panel using GridLayout layout manager

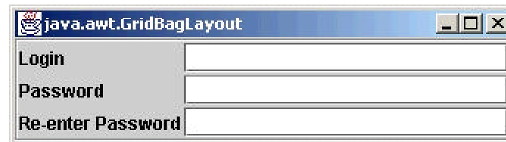


Figure 5. Sample change-password panel using GridBagLayout layout manager

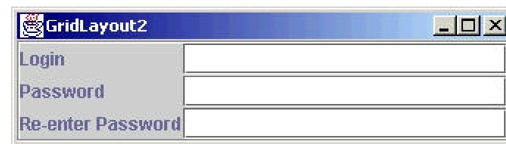


Figure 6. Same panel using GridLayout2 layout manager

These screenshots run the simple `Test` class shown below. Since `GridLayout2` is a `GridLayout` subclass, you can use the same code to construct the samples for either layout (the `Test` class for `GridBagLayout` will follow):

```
  public Test(GridLayout layout, JComponent[] component) {
    super(layout.getClass().getName());
   JPanel panel = new JPanel(layout);
    //--- code needed to add the components
    // less than using a java.awt.GridBagLayout
    for (int i = 0; i < component.length; i ++) {
      panel.add(component[i]);
    }
    //---
    panel.setBorder(new EtchedBorder());
    setContentPane(panel);
    pack();
    show();
  }
```

The `Test` class extends the `JFrame` class, so you don't need to do anything special to create the window frame -- just make sure the constructor invokes `super()` appropriately. You then set the frame's content pane to be a `JPanel` instance, and the given layout to be the panel's layout manager. Then you add a component array to the panel by invoking the `add()` method for each component (see the loop iteration). Pretty simple. Finally, you `pack()` and `show()` the window.

Note that the test programs don't actually perform any actions. The test programs only show how the layout managers behave when the containers (whose layouts they manage) are resized.

On the other hand, `GridBagLayout` requires a separate routine to construct it. `GridBagLayout` requires more (convoluted) code to specify the constraints applied when laying out the components for the desired behavior:

```
  public Test(int columns, Insets insets, JComponent[] component) {
    super(GridBagLayout.class.getName());
```

```
    GridBagLayout layout = new GridBagLayout();
    JPanel panel = new JPanel(layout);
    //--- code needed to add the components
    GridBagConstraints constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.WEST;
    constraints.insets = insets;
    constraints.fill = GridBagConstraints.BOTH;
    for (int i = 0; i < component.length; i ++) {
      constraints.gridx = i % columns;
      constraints.gridy = i / columns;
      layout.setConstraints(component[i], constraints);
      panel.add(component[i]);
    }
    //---
    panel.setBorder(new EtchedBorder());
    setContentPane(panel);
    pack();
    show();
  }
```

Here, you use a `GridBagConstraints` instance to create the desired layout. To simply lay out the components like in the previous example, use the constraints instance's `gridx` and `gridy` fields to specify each component's location in the grid layout. You can easily observe that using a `java.awt.GridBagLayout` manager requires writing more code; for example, the loop iteration contains three more lines. Also, as you'll see below, without more complex and convoluted constraint management, the `GridBagLayout` doesn't always behave as you would like when faced with functions like resizing windows.

Resizing

Now, let's resize Figures 1, 2, and 3 from the first exercise. (You can try this yourself by running the `Test` program in Resources.) Figures 7, 8, and 9 show the screenshots from my system.

Figure 7. Resize 14 multisized JLabels with a GridLayout layout manager

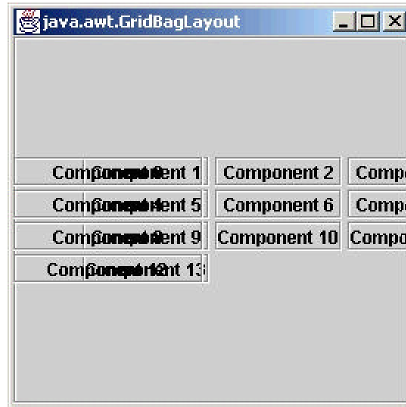


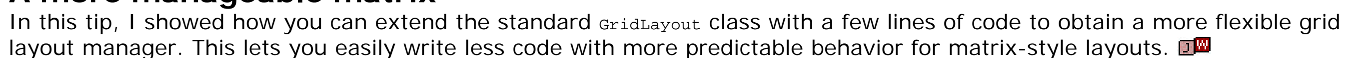
Figure 8. Resize 14 multisized JLabels with a GridBagLayout layout manager

Figure 9. Resize 14 multisized labels using a GridLayout2 layout manager

In resizing the first test batch, you can see that the `GridLayout2` code in Figure 9 remains the only one that displays the desired view. I will let you resize the figures in the compact panel exercise. Note particularly the `GridBagLayout` example's poor behavior when you resize the window to something smaller than its desired size.

A more manageable matrix

In this tip, I showed how you can extend the standard `GridLayout` class with a few lines of code to obtain a more flexible grid layout manager. This lets you easily write less code with more predictable behavior for matrix-style layouts.



About the author

Bogdan Dorohonceanu is a graduate research assistant at CAIP Center, Rutgers University, where he uses Java to build user interfaces for distributed collaborative applications.

Resources

- Complete GridLayout2 source code: http://www.javaworld.com/javaworld/javatips/javatip121/GridLayout2.java
- Example test program source code: http://www.javaworld.com/javaworld/javatips/javatip121/Test.java
- The java.awt.GridLayout class documentation: http://java.sun.com/j2se/1.4/docs/api/java/awt/GridLayout.html
- "How to Use GridLayout," an online chapter from *The Java Tutorial*, Mary Campione et al. (Addison-Wesley, December 2000; ISBN: 0201703939): http://java.sun.com/docs/books/tutorial/uiswing/layout/grid.html

- "Java Tip 116: Set Your Table Options -- at Runtime!" Sonal Goyal with John D. Mitchell (*JavaWorld*, September 2001):
  http://www.javaworld.com/javaworld/javatips/jw-javatip116.html
- Browse *JavaWorld*'s **AWT/Swing** Index:
  http://www.javaworld.com/channel_content/jw-awt-index.shtml
- Browse *JavaWorld*'s **Foundation Classes** Index:
  http://www.javaworld.com/channel_content/jw-foundation-index.shtml
- Browse *JavaWorld*'s **User Interface Design** Index:
  http://www.javaworld.com/channel_content/jw-ui-index.shtml
- View all previous **Java Tips** and submit your own:
  http://www.javaworld.com/javatips/jw-javatips.index.html
- Learn Java from the ground up in *JavaWorld*'s **Java 101** column:
  http://www.javaworld.com/javaworld/topicalindex/jw-ti-java101.html
- Java experts answer your toughest Java questions in *JavaWorld*'s **Java Q&A** column:
  http://www.javaworld.com/javaworld/javaqa/javaqa-index.html
- Sign up for *JavaWorld*'s free weekly email newsletters:
  http://www.idg.net/jw-subscribe
- Speak out in *JavaWorld*'s Java Forum:
  http://forums.devworld.com/webx?13@@.ee6b802
  You'll find a wealth of IT-related articles from our sister publications at IDG.net

HOME | FEATURED TUTORIALS | COLUMNS | NEWS & REVIEWS | FORUM | JW RESOURCES | ABOUT JW | FEEDBACK

Copyright © 2003 JavaWorld.com, an IDG company