InputLayout

The layout manager that handles columns; each column contains two subcolumns, one is the "prompt" and the other is the "input". I wrote it long ago; feel free to use it. Just add it to your favorite package.

First, here's how to use it when you want only one column. Note that the getter methods return a JTextField. As you see, when you add components, the order is very important. The odd components are the labels (prompts), and the even components are the input fields. Odd in this sense is the serial number of the add call.

Well, actually it's not that simple. When laying out the components, the manager queries the target component (the pane) for the subcomponents, and uses the order they are provided. That's important only if you change the order of the subcomponents later. It might even cause the reordering of the components on the already displayed pane.

Sample code of using the layout manager.

```
        public JPanel createInputPane() {
        JPanel p = new JPanel(new InputLayout());

        p.add(new JLabel("Total Maximum Running Time "));
        p.add(getMaximumRuntimeField());

        p.add(new JLabel("Maximum Number Of Iterations "));
        p.add(getIterationsField());

        p.add(new JLabel("Maximum Running Time Per Iteration "));
        p.add(getIterationTimeField());

        return p;
        }
```

And the source code of the layout manager (and here you are the downloadable source code):

Source code

```
import java.awt.Component;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.Insets;
import java.awt.LayoutManager2;
/**
 * The layout manager that organizes the components into
 * columns. Each column contains a pair of components; one
 * "prompt" and one "input field". The prompts within any
 * single column has the same width (the width of the widest
 * label, if possible). Similarly, the width of the input
 * fields is the same for all of the fields of that same
 * column. The input fields are also aligned. The height of
 * the components in the same row will be set to the height
 * of the tallest component.
 * <P>
 * If there's any extra space available, it'll be
 * partitioned among the input fields, by taking into
 * account the "weights" of the columns. By default, the
 * weights are equal for all the columns, but upon
 * instantiation, it might be changed.
 * <P>
 * The number of columns might be determined upon
 * instantiation, and it's immutable, but it is possible to
 * set the weights at any time.
 * <P>
 * When calculating the "needed" space, the preferred size
 * of the components will be used. In case not enough space
 * is available, the minimum size of all the components will
 * be used, then the extra available space upon the minimum
 * will be partitioned among the input fields according to
 * the weights. The prompts won't grow ever.
 * <P>
 * If there isn't enough space to provide even when using
```

```
 * the minimum sizes, then the input fields will shrink. In
 * this case, the columns with more weight will shrink less
 * (broadly speaking, a column with half the weight of the
 * other shrinks twice as much). However, effort is taken so
 * that no input field be shorter than the size needed by
 * the capital letter "M" to appear. If the available space
 * needed for any column is even less, the layout manager
 * tries to shrink another columns with greater weights. If
 * the width of the input fields of all column reached this
 * minimum size, and the available space is even less, the
 * layout manager shrinks the prompts. When worst comes to
 * worst, the prompts will disappear sooner than the input
 * fields.
 * <P>
 * When adding components to the container managed by the
 * InputLayout, the components added to the odd indices are
 * the prompts, and the ones added to the even indices are
 * the input fields. The layout manager never checks the
 * actual type of the components.
 * <P>
 * The component serving as a "prompt" should be the
 * immediate previous one of the associated "input field" it
 * serves. That is, the component index of the prompt in the
 * container must be one less than the component index of
 * the associated input field.
 * <P>
 * When there are more columns, the first pair of components
 * is added to the first column, the second pair of
 * components is added to the second column, and so on until
 * all the columns are populated. If more components are
 * still available, the next pair will be added to the
 * second row of the first column, and so on. If the number
 * of components is odd (that is, no input field is
 * available for the last "pair"), then that space will be
 * left empty, but the corresponding prompt won't be taken
 * into account when the layout manager calculates the width
 * of the "prompt" subcolumn of the corresponding column.
 * Rather, that single prompt may grow and occupy the space
 * which otherwise would be occupied by its associated input
 * field.
 * <P>
 * If both the prompt and the associated input field is
 * invisible, it won't taken into account when calculating
 * the space they needed, but their place will be left empty
 * by default. Therefore, when they made visible, the order
 * of the component pairs won't change. However, it is
 * possible to switch on or off this feature. When it is
 * switched off (no place holders are allowed for the
 * invisible component pairs), then once the pair is made
 * visible again, all of the component pairs following will
 * be shifted one column (and consequently may appear in the
 * next row if were put into the last column previously).
 * <P>
 * As it is now, InputLayout lays out the components from
 * left to right, and from top to bottom. However, after
 * calculating all the necessary information, a method with
 * the sole purpose to size and position the components is
 * called. This method gets the calculated sizes as
 * parameters. Therefore, the offsprings of InputLayout has
 * to redefine this method only if they want to change the
 * order in which the components are positioned.
 *
 * @author Istvan KETLER, Sinkó Tekla (u159905)
 *
 **/
public class InputLayout implements LayoutManager2 {
    private static enum InputLayoutType {
        PREFERRED, MINIMUM, MAXIMUM
    }
    /**
     * Number of columns (determined at instantiation). One
```

```
 * column actually means two columns: the prompt column
 * and the input fields column.
 **/
private final int nrOfColumns;
/**
 * The weights of the columns.
 **/
private final int[] columnWeights;
/**
 * Is the place should be reserved for the invisible
 * components?
 **/
private boolean isPlaceHolderAllowed = true;
// The required space
Dimension needed = null;
// The "must-have" space
Dimension badlyNeeded = null;
/**
 * Allows or forbids the use of place holders in the
 * place of the invisible component pairs.
 *
 * @param allow
 *            True if the use of the place holders is
 *            allowed, false otherwise.
 **/
public void allowPlaceHolders(final boolean allow) {
    isPlaceHolderAllowed = allow;
}
/**
 * Creates a layout where the components are organized
 * into one column. Each column contains a pair of
 * components (the "prompt" and the "input field"), and
 * each column has the same (zero) weight.
 **/
public InputLayout() {
    this(1, null);
}
/**
 * Creates a layout where the components are organized
 * into the given number of columns, and each column has
 * equal weight. Each column contains a pair of
 * components (the "prompt" and the "input field").
 *
 * @param cols
 *            The number of columns. It must be less
 *            than 100 (that is the strict limit),
 *            although more than a few might not have a
 *            reasonable visual appearance.
 **/
public InputLayout(final int cols) {
    this(cols, null);
}
/**
 * Creates a layout where the components are organized
 * into the given number of columns, and each column has
 * its given weight. The weight information will be
 * ignored if the total sum of the provided weights
 * exceeds 100, or if any of the remaining columns with
 * undefined weight would have 0 weight. (An example is
 * if there are 4 columns, and the first two have the
 * weights of 49 and 50). In this case equal weights
 * will be used for each column, as if no weight
 * information would have been provided at all. Also
 * note that in cases when the number of columns makes
 * it impossible to partition the 100 equally (eg. in
 * case of 3 columns), the last column gets the extra
 * weight so that the total sum will always be exactly
 * 100. If the weight of each column is explicitly
 * given, but the total sum is less than 100, then the
 * last column picks up all the remaining weight units.
 *
```

```
     * @param cols
     *            The number of columns. It must be less
     *            than 100 (that is the strict limit),
     *            although more than a few might not have a
     *            reasonable visual appearance.
     * @param weights
     *            The array of the weights (given in
     *            percents). The total sum of the weights
     *            should be 100. It is allowed to provide an
     *            array which has less or more number of
     *            elements than the number of columns. In
     *            case of less columns, the undefined ones
     *            will share the remaining space equally. In
     *            case of more elements, the unnecessary
     *            ones are ignored.
     **/
    public InputLayout(final int cols, final int[] weights) {
        // Actually the max number of columns should be much
        // less.
        assert (cols < 100) : "Too many columns";
        nrOfColumns = cols;
        columnWeights = new int[nrOfColumns];
        // if no weights are specified
        if (weights == null) {
            setEqualWeights(0);
            // otherwise, initialize the weight information
        } else {
            int total = 0;
            // Calculate the total sum of weights - it
            // should be <= 100
            for (int i = 0; i < columnWeights.length; i++) {
                // Oops, more weights than columns!
                if (i >= weights.length) {
                    setEqualWeights(i);
                    break;
                }
                total += (columnWeights[i] = weights[i]);
            }
            // More than 100%? How could it be? So ignore
            // it!
            if (total > 100) {
                setEqualWeights(0);
            } else if (total < 100) {
                // The last column gets all of the remaining
                // space
                columnWeights[columnWeights.length - 1] += 100 - total;
            }
        }
    }
    /**
     * Sets equal weights to the remaining columns. The
     * method calculates the weight of the columns not
     * involved, and provides the remaining value in equal
     * pieces to the remaining columns. The total sum of all
     * the column weights should be 100.
     *
     * @param cols
     *            The first column which is involved.
     **/
    protected void setEqualWeights(int cols) {
        // This could never happen...
        assert (cols < columnWeights.length) : "The column index is greater than the number of columns!";
        int len = columnWeights.length;
        // Were it called from the constructor, this code
        // has no use...
        // If only the first few columns would have been
        // determined weight
        int used = 0;
        for (int i = 0; i < cols; i++) {
            used += columnWeights[i];
        }
```

```
        // Not enough space for the remaining columns
        if (used > (100 - (len - cols))) {
            used = 0;
            cols = 0;
        }
        // end of "has no use" code
        int free = 100 - used;
        int remcols = len - cols;
        // The percents of the remaining columns
        // Be aware, w * remcols may or may not equal to
        // free! (divide int)
        int w = free / remcols;
        for (int i = cols; i < (len - 1); i++) {
            used += w;
            columnWeights[i] = w;
        }
        // The last column gets all of the remaining space
        columnWeights[len - 1] = 100 - used;
    }
    /**
     * Calculates the minimum, preferred, or maximum sizes
     * of the columns.
     *
     * @param target
     *            The container which size should be
     *            calculated.
     * @param nComp
     *            The number of components added to the
     *            container.
     * @param nrOfRows
     *            The number of rows the components will be
     *            laid out.
     * @param type
     *            The actual size should be used. The value
     *            might be MINIMUM, MAXIMUM, or PREFERRED.
     * @returns An array of arrays containing the sizes. The
     *          first two array gives the widths of the
     *          columns: the widths of the prompt subcolumn
     *          and the widths of the input field subcolumn.
     *          The third array gives the heights of the
     *          rows.
     **/
    private int[][] calculateDimensions(final Container target, final int nComp, final int nrOfRows, final
InputLayoutType type) {
        // The return array
        int[][] retArray = new int[3][];
        // By default, all of the elements of the arrays are
        // initialized by 0.
        // The width of the prompts
        int[] promptWidths = new int[nrOfColumns];
        // The width of the input fields
        int[] fieldWidths = new int[nrOfColumns];
        // The heights of the rows
        int[] heights = new int[nrOfRows];
        // Initialize the return array
        retArray[0] = promptWidths;
        retArray[1] = fieldWidths;
        retArray[2] = heights;
        // The next component
        Component theComponent = null;
        // Size of the prompt
        Dimension pdim = null;
        // Size of the input field
        Dimension fdim = null;
        // The actual component
        int currentComponent = 0;
        // For each row
        for (int currentRow = 0; currentRow < nrOfRows; currentRow++) {
            int height = 0;
            // for each column (However, the column is
            // increased only if
```

```
// either the place holder is allowed, or one of
// the two
// components forming the column is visible)
for (int col = 0; col < nrOfColumns;) {
    // checks which sub-column exists (prompt
    // and input field)
    int subcols = 0;
    if (currentComponent >= nComp) {
        // no more components are available
        break;
    }
    // The next component
    theComponent = target.getComponent(currentComponent++);
    // The component exists and is visible
    if ((theComponent != null) && theComponent.isVisible()) {
        // The prompt is visible
        subcols |= 1;
        // pdim gets the selected size of the
        // component
        switch (type) {
            case MINIMUM:
                pdim = theComponent.getMinimumSize();
                break;
            case MAXIMUM:
                pdim = theComponent.getMaximumSize();
                break;
            default:
            case PREFERRED:
                pdim = theComponent.getPreferredSize();
                break;
        }
    }
    // The very last input field may not exist
    if (currentComponent < nComp) {
        theComponent = target.getComponent(currentComponent++);
        // The component exists and is visible
        if ((theComponent != null) && theComponent.isVisible()) {
            // The input field is visible
            subcols |= 2;
            // fdim gets the selected size of
            // the component
            switch (type) {
                case MINIMUM:
                    fdim = theComponent.getMinimumSize();
                    break;
                case MAXIMUM:
                    fdim = theComponent.getMaximumSize();
                    break;
                default:
                case PREFERRED:
                    fdim = theComponent.getPreferredSize();
                    break;
            }
        }
    }
    // If the place holder is allowed, of if one
    // of the sub-columns is visible
    if (isPlaceHolderAllowed || (subcols > 0)) {
        // if only the label is visible, there's
        // nothing to do.
        // That is, no label is allowed to be
        // longer than the width
        // of the sum of the prompt and the
        // input field columns
        switch (subcols) {
            case 3: {
                // Finding the widest prompt in
                // the column
                promptWidths[col] = Math.max(promptWidths[col], pdim.width);
                // Finding the max height in
                // this row
```

```
                        height = Math.max(height, pdim.height);
                        // continues with the next case
                    }
                    case 2: {
                        // Finding the widest input
                        // field in the column
                        fieldWidths[col] = Math.max(fieldWidths[col], fdim.width);
                        // Finding the max height in
                        // this row
                        height = Math.max(height, fdim.height);
                        break;
                    }
                }
                // The column number is increased only
                // if either the place
                // holder is allowed, or at least one of
                // the components
                // out of the two is visible.
                col++;
            }
        }
        // Store the calculated max height of the row
        heights[currentRow] = height;
        // No more components
        if (currentComponent >= nComp) {
            break;
        }
    }
    return retArray;
}
private Dimension calculateLayoutSize(final Container target, final InputLayoutType inputLayout,
        final int columnSeparatingWidth, final int rowSeparatingHeight){
    int nComp = target.getComponentCount();
    // The number of actual rows might be less if there
    // are invisible components added to the container.
    int nrOfRows = (((nComp / 2) + nrOfColumns) - 1) / nrOfColumns;
    int[][] dims = calculateDimensions(target, nComp, nrOfRows, inputLayout);
    int[] promptWidths = dims[0];
    int[] fieldWidths = dims[1];
    int[] heights = dims[2];
    // Sum up the widths of the columns
    // (the columns are separated by columnSeparatingWidth pixels):
    int w = columnSeparatingWidth;
    for (int i = 0; i < nrOfColumns; i++) {
        w += promptWidths[i];
        w += fieldWidths[i];
        w += columnSeparatingWidth;
    }
    // If the number of rows is less than it was
    // calculated, the unoccupied
    // rows will have the height of 0 due to how
    // Java initializes the arrays.
    int h = rowSeparatingHeight;
    for (int i = 0; i < nrOfRows; i++) {
        h += heights[i];
        h += rowSeparatingHeight;
    }
    // The insets values should be taken into account
    Insets ins = target.getInsets();
    w += ins.left + ins.right;
    h += ins.top + ins.bottom;
    return new Dimension(w, h);
}
/**
 * Calculates the minimum size this container needs in
 * order to lay out all of the components it contains.
 * Upon calculation, only the space needed by the
 * visible components will taken into account. If both
 * element of a prompt/input field pair are invisible,
 * then depending on the current setting of the place
 * holders, the next pair still appears in its own
```

```
     * column, or replaces the invisible pair.
     *
     * @param target
     *            The container of which the minimum size
     *            should be calculated.
     * @returns The dimension minimally needed to lay out
     *          the container.
     * @see #allowPlaceHolders(boolean)
     *
     * @see java.awt.LayoutManager#minimumLayoutSize(java.awt.Container)
     */
    @Override
    public Dimension minimumLayoutSize(final Container target) {
        InputLayoutType inputLayout = InputLayoutType.MINIMUM;
        int columnSeparatingWidth = 0;
        int rowSeparatingHeight = 0;
        return calculateLayoutSize(target, inputLayout, columnSeparatingWidth, rowSeparatingHeight);
    }
    /**
     * Calculates the preferred size this container needs in
     * order to lay out all of the components it contains.
     * Upon calculation, only the space needed by the
     * visible components will taken into account. If both
     * element of a prompt/input field pair are invisible,
     * then depending on the current setting of the place
     * holders, the next pair still appears in its own
     * column, or replaces the invisible pair.
     *
     * @param target
     *            The container of which the preferred size
     *            should be calculated.
     * @returns The dimension preferably needed to lay out
     *          the container.
     * @see #allowPlaceHolders(boolean)
     *
     * @see java.awt.LayoutManager#preferredLayoutSize(java.awt.Container)
     */
    @Override
    public Dimension preferredLayoutSize(final Container target) {
        InputLayoutType inputLayout = InputLayoutType.PREFERRED;
        // the columns are separated by 4 pixels:
        int columnSeparatingWidth = 4;
        // the rows are separated by 4 pixels:
        int rowSeparatingHeight = 4;
        return calculateLayoutSize(target, inputLayout, columnSeparatingWidth, rowSeparatingHeight);
    }
    /**
     * Calculates the maximum size this container needs in
     * order to lay out all of the components it contains.
     * Upon calculation, only the space needed by the
     * visible components will taken into account. If both
     * element of a prompt/input field pair are invisible,
     * then depending on the current setting of the place
     * holders, the next pair still appears in its own
     * column, or replaces the invisible pair.
     *
     * @param target
     *            The container of which the maximum size
     *            should be calculated.
     * @returns The maximum dimension needed to lay out the
     *          container.
     * @see #allowPlaceHolders(boolean)
     *
     * @see java.awt.LayoutManager2#maximumLayoutSize(java.awt.Container)
     */
    @Override
    public Dimension maximumLayoutSize(final Container target) {
        InputLayoutType inputLayout = InputLayoutType.MAXIMUM;
        int columnSeparatingWidth = 0;
        // the rows are separated by 4 pixels:
        int rowSeparatingHeight = 0;
```

```
        return calculateLayoutSize(target, inputLayout, columnSeparatingWidth, rowSeparatingHeight);
    }
    /**
     * Laying out the components in the container.
     *
     * @see java.awt.LayoutManager#layoutContainer(java.awt.Container)
     */
    @Override
    public void layoutContainer(final Container target) {
        // Number of components
        int nComp = target.getComponentCount();
        // The number of actual rows might be less if there
        // are invisible
        // components added to the container.
        int nrOfRows = (((nComp / 2) + nrOfColumns) - 1) / nrOfColumns;
        // The current size of the container
        Dimension dim = target.getSize();
        Insets insets = target.getInsets();
        // The space remaining for the components. Note that
        // in case of
        // negative insets, the space is bigger
        int currentWidth = dim.width - (insets.left + insets.right);
//        int currentHeight = dim.height - (insets.top + insets.bottom);
        if (needed == null) {
            needed = preferredLayoutSize(target);
            needed.setSize(needed.width - (insets.left + insets.right), needed.height - (insets.top + insets.
bottom));
        }
        if (currentWidth >= needed.width) {
            int[][] dims = calculateDimensions(target, nComp, nrOfRows, InputLayoutType.PREFERRED);
            double extra = currentWidth - needed.width;
            createActualLayout(target, extra, dims, 2);
        } else {
            if (badlyNeeded == null) {
                badlyNeeded = minimumLayoutSize(target);
                badlyNeeded.setSize(badlyNeeded.width - (insets.left + insets.right), badlyNeeded.height
                        - (insets.top + insets.bottom));
            }
            if (currentWidth >= badlyNeeded.width) {
                int[][] dims = calculateDimensions(target, nComp, nrOfRows, InputLayoutType.MINIMUM);
                double extra = currentWidth - badlyNeeded.width;
                createActualLayout(target, extra, dims, 0);
            } else {
                // shrink
                int[][] dims = calculateDimensions(target, nComp, nrOfRows, InputLayoutType.MINIMUM);
                createActualLayout(target, 0.0d, dims, 0);
            }
        }
    }
    /**
     * Distributes the extra space among the columns.
     *
     * @param target
     *            The container which contains the
     *            components.
     * @param extra
     *            The extra space that should be distributed.
     * @param dims
     *            The array of int arrays which contains the
     *            already calculated width and height
     *            informations. The first subarray is the
     *            widths of the prompt subcolumns, and the
     *            second one is the widths of the input
     *            field subcolumns. The third subarray
     *            contains the heights of the rows.
     * @param ins
     *            The gap between the subcolumns. This gap
     *            should be used before and after of each
     *            component. The same gap also should be
     *            used above and below of each component.
     */
```

```
    private void createActualLayout(final Container target, final double extra, final int[][] dims, final int
ins) {
        int[] fieldWidths = dims[1];
        int all = 0;
        if (extra > 0) {
            for (int i = 0; i < nrOfColumns; i++) {
                int add = (int) ((extra * columnWeights[i]) / 100);
                all += add;
                fieldWidths[i] += add;
            }
        }
        if (all != extra) {
            fieldWidths[nrOfColumns - 1] += extra - all;
        }
        layoutComponents(target, dims, ins);
    }
    /**
     * Lays out the components on the container. Sizing and
     * locating the components is done from left to right,
     * and from top to bottom.
     *
     * @param target
     *            The container which contains the
     *            components.
     * @param dims
     *            The array of int arrays which contains the
     *            already calculated width and height
     *            informations. The first subarray is the
     *            widths of the prompt subcolumns, and the
     *            second one is the widths of the input
     *            field subcolumns. The third subarray
     *            contains the heights of the rows.
     * @param ins
     *            The gap between the subcolumns. This gap
     *            should be used before and after of each
     *            component. The same gap also should be
     *            used above and below of each component.
     **/
    protected void layoutComponents(final Container target, final int[][] dims, final int ins) {
        int nComp = target.getComponentCount();
        int[] promptWidths = dims[0];
        int[] fieldWidths = dims[1];
        int[] heights = dims[2];
        Insets insets = target.getInsets();
        int gap = ins + ins;
        Component prompt = null;
        Component field = null;
        int currentComponent = 0;
        int top = insets.top + gap;
        for (int currentRow = 0; currentRow < heights.length; currentRow++) {
            int left = insets.left + ins;
            for (int col = 0; col < nrOfColumns;) {
                int subcols = 0;
                if (currentComponent >= nComp) {
                    // no more components are available
                    break;
                }
                prompt = target.getComponent(currentComponent++);
                if ((prompt != null) && prompt.isVisible()) {
                    subcols |= 1;
                }
                if (currentComponent < nComp) {
                    field = target.getComponent(currentComponent++);
                    if ((field != null) && field.isVisible()) {
                        subcols |= 2;
                    }
                }
                if (isPlaceHolderAllowed || (subcols > 0)) {
                    switch (subcols) {
                        // Only the prompt exists and/or is
                        // visible.
```

```
                        case 1: {
                            prompt.setBounds(left, top, promptWidths[col] + fieldWidths[col] + gap, heights
[currentRow]);
                            break;
                        }
                        // Both the prompt and the input
                        // field exist and are visible.
                        case 3: {
                            prompt.setBounds(left, top, promptWidths[col], heights[currentRow]);
                            // continues with the next case
                        }
                        // Only the input field exists
                        // and/or is visible.
                        case 2: {
                            field.setBounds(left + promptWidths[col] + gap, top, fieldWidths[col], heights
[currentRow]);
                            break;
                        }
                    }
                    left += promptWidths[col] + gap;
                    left += fieldWidths[col] + ins;
                    col++;
                }
            }
            top += heights[currentRow] + gap;
            if (currentComponent >= nComp) {
                break;
            }
        }
    }
    /**
     * Adds the component to the layout. This method is
     * provided for compatibility considerations, and
     * delegates the call to the newer form of the
     * overloaded method.
     *
     * @see #addLayoutComponent(java.awt.Component,
     *      java.lang.Object)
     **/
    @Override
    public void addLayoutComponent(final String name, final Component comp) {
        this.addLayoutComponent(comp, name);
    }
    /**
     * Adds the component to the layout. This operation
     * invalidates the layout.
     *
     * @see java.awt.LayoutManager2#addLayoutComponent(java.awt.Component, java.lang.Object)
     **/
    @Override
    public void addLayoutComponent(final Component comp, final Object constraints) {
        doInvalidate();
    }
    /**
     * Removes the component from the layout. This operation
     * invalidates the layout.
     *
     * @see java.awt.LayoutManager#removeLayoutComponent(java.awt.Component)
     */
    @Override
    public void removeLayoutComponent(final Component comp) {
        doInvalidate();
    }
    /**
     * Invalidates the information stored by the layout
     * manager. The only information currently cached is the
     * dimension of the preferred and the minimum size of
     * the container.
     *
     * @see java.awt.LayoutManager2#invalidateLayout(java.awt.Container)
     */
```

```
    @Override
    public void invalidateLayout(final Container target) {
        doInvalidate();
    }
    /**
     * Performs the invalidate layout functionality. That
     * is, deletes all of the cached information about the
     * layout.
     **/
    private void doInvalidate() {
        needed = null;
        badlyNeeded = null;
    }
    /**
     * Provides alignment information. This is a no-op.
     *
     * @see java.awt.LayoutManager2#getLayoutAlignmentX(java.awt.Container)
     */
    @Override
    public float getLayoutAlignmentX(final Container target) {
        return 0.0f;
    }
    /**
     * Provides alignment information. This is a no-op.
     *
     * @see java.awt.LayoutManager2#getLayoutAlignmentY(java.awt.Container)
     */
    @Override
    public float getLayoutAlignmentY(final Container target) {
        return 0.0f;
    }
}
```