Zaval Light-Weight Visual Components Library

Version 3.0

Teaching Materials

Zaval Creative Engineering Group
http://www.zaval.org

Contents

Introduction

This document tells how to implement various things using the Zaval Light-Weight Visual Components Library (LwVCL). Actually, it's a sort of "How to…" document – it consists of real world examples that can be cat-and-paste to you application. Each example it well-commented, so it is easy enough to understand the whole process in details.

How to...

The chapter contains set of examples to illustrate the basic ideas of the library. The samples are not too complex and in most cases can be used just for training.

…create a component

The sample illustrates how to create a lightweight component using this library. The component itself is very simple. It controls a background color according mouse state. In case mouse pointer is located inside the component background will be set to "red", otherwise to "gray". If a mouse button is pressed than the background will be set to "yellow". See code example with comments below:

```
public class LwComponentSample
extends LwCanvas
implements LwMouseListener
{
   private boolean isInside;

  public LwComponentSample() {
    setBackground (Color.gray);
  }

  public void mouseEntered (LwMouseEvent e) {
    setBackground (Color.red);
    isInside = true;
  }

  public void mouseExited  (LwMouseEvent e) {
    setBackground (Color.gray);
    isInside =false;
 }

  public void mousePressed (LwMouseEvent e) {
    setBackground (Color.yellow);
  }

  public void mouseReleased(LwMouseEvent e) {
    if (isInside)  setBackground (Color.red);
    else           setBackground (Color.gray);
 }

  public void mouseClicked (LwMouseEvent e) {}
}
```

This component inherits lightweight component implementation and implements mouse listener interface to get and handle mouse events.

This method is called with the event manager when mouse pointer enters the component area. *isInside* field is set to true.

This method is called with the event manager when mouse pointer exits the component area. *isInside* field is set to true.

This method is called with the event manager when a mouse button has been pressed.

The method is called with the event manager when a mouse button has been pressed. The method determines if the mouse pointer is inside the component by *isInside* field of the class.

As you can see, it is like creating of AWT/ SWING component.

…create a view

The sample illustrates a view creation. The view is used to show colors palette that is represented with 12 boxes. Every color is formed from previous color by adding given "rgbIncremet" variable to red, green and blue values. Source code and comments are shown below:

```
public class LwMyView
extends LwView
{
  private Color basic = Color.white;
  private int   width = 15, height = 15;
  private int   rgbIncrement = -15;


  public LwMyView() {
    super(ORIGINAL);
  }

  protected Dimension calcPreferredSize() {
    return new Dimension(width*4, height*3);
  }

  public void paint (Graphics g,
                     int x, int y,
                     int w, int h, Drawable d)
  {
    int realWidth  = w/4;
    int realHeight = h/3, xx = x, yy = y;
    Color color = basic;
    for (int i=0; i < 3; i++)
    {
      for (int j=0; j < 4; j++)
      {
        g.setColor(color);
        g.fillRect(xx, yy, realWidth, realHeight);
        color=new Color(color.getRed()+rgbIncrement,
                        color.getGreen() + rgbIncrement,
                        color.getBlue()  + rgbIncrement);
        xx += realWidth;
      }
      yy += realHeight;
      xx = x;
    }
  }
}
```

This class inherits abstract view class.

Field “basic” defines a color that will be used as starting color, “width” and “height” fields define size of a color box. “rgbIncrement” field determines a value that is used to form next color.

Constructor indicates to use ORIGINAL (preferred size) to show the view.

This method defines preferred size of the view. This is very important method that will be used to calculate a preferred size of a lightweight component that utilizes this view.

This method determines functionality for appropriate abstract method of **LwView** class. It defines “face” of the view. The method is executed with a paint manager that passes a location - where the view should be painted and a size - that should be used for this view. In our case view recalculates color boxes size according to “width” and “height “ that have been provided with the owner lightweight component.

The cycle calculates colors and shows the colors boxes. The boxes are shown as a table that has four columns and three rows. Below the view face is shown:

The view usage is shown below:

```
public static void main(String[] args)
 {
   LwFrame frame = new LwFrame();
   frame.setSize(100, 80);
   frame.getRoot().setLwLayout(new LwFlowLayout());
   LwComponent comp = new LwCanvas();
   comp.getViewMan(true).setView(new LwMyView());
   frame.getRoot().add(comp);
   frame.setVisible(true);
 }
```

First of all, this method creates a lightweight frame. The frame provides a lightweight root component that should be used to add any other lightweight components. **LwFlowLayout** is set as a layout manager for the root component (default layout uses raster to layout child components). Next, it is necessary to have a lightweight component that will be used as the view owner, so the component is created and the view is set as a “face” for the component. Finally, the component is added to the root and we set frame as visible to show it.

…create your own render

The render concept is the same as the view. Render is used to create view for an object. For example the library has render to represent view for an image (**java.awt.Image**). The sample below illustrates render to paint integer matrix (the target object for the render conforms to int[][] Java type). The render use different colors to paint the matrix items. If an item divisible by 2 than “red” color is used, if item divisible by 3 than “yellow” color is used and so on. The sample is like the view sample that has been shown in previous section. The source code and comments are shown below.

```
public class LwMyRender
```

We inherit abstract render class and implement paint() method.

```
extends LwRender
{
  private int width = 2, height = 2;

  public LwMyRender(Object target) {
    super(target);
  }

  protected Dimension calcPreferredSize() {
    int[][] target = (int[][])getTarget();
    return new Dimension(width*target.length,
                         height*target[0].length);
  }

  public void paint (Graphics g, int x, int y,
                     int w, int h, Drawable d)
  {
    int[][] target =(int[][])getTarget();
    int realWidth  = w/target.length;
    int realHeight = h/target[0].length, xx = x, yy = y;
    for (int i=0; i<target.length; i++)  {
      for (int j=0; j<target[i].length; j++)  {
        if (target[i][j]%2==0) g.setColor(Color.red);
        else
        if (target[i][j]%3==0) g.setColor(Color.yellow);
        else
        if (target[i][j]%5==0) g.setColor(Color.green);
        else
        if (target[i][j]%7==0) g.setColor(Color.black);
        else                   g.setColor(Color.gray);
        g.fillRect(xx, yy, realWidth, realHeight);
        xx += realWidth;
      }
      yy += realHeight;
      xx = x;
    }
  }
}
```

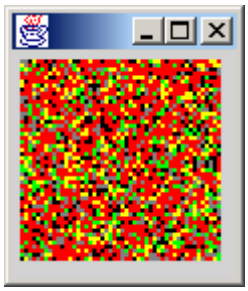
Note: The **LwRender** class inherits **LwView** class.

The target object that should be painted has to be passed to constructor. In our case, the constructor waits int[][] Java type for the target.

This method determines a preferred size for the render. The information is used with an owner component to calculate a preferred size, so it is very important to have the method. There is not necessary to think about the render insets, this method provides “pure” preferred size.

This method determines functionality for appropriate abstract method of **LwRender** class. It defines “face” of the render. The method will be executed with paint manager that passes a location - where the view should be painted and a size that should be used for the view. In our case, the view recalculates color boxes size according to “width” and “height” that have been provided with the owner lightweight component After that the target (integer matrix) will be painted according to items values.

The resulting view for a target that was generated using randomization is shown below:



To understand how the render can be used for a lightweight component see previous section, the usage is similar.

…create layout manager

Layout manager is a class that controls child components’ sizes and locations. Lightweight containers always use layout managers. The sample illustrates how to create layout manager. Lightweight library provides special interface **LwLayout** that should be used to create your own layout managers.

The sample layout manager divides a container area to four parts: left top corner, right top corner, left bottom corner and right bottom corner. Each of the parts can be used to place a child component. To determine what part of the container area should be used for a given child component the sample layout provides four constants. Both source code and comments are shown below:

```
public class LwMyLayout implements LwLayout {
 public static final Object TOPLEFT=new Integer(4);
 public static final Object TOPRIGHT=new Integer(3);
 public static final Object BOTTOMLEFT=new Integer(3);
 public static Object BOTTOMRIGHT=new Integer(1);
 Layoutable topLeft, topRight, bottomLeft, bottomRight;

  public void componentAdded(Object o,Layoutable l,int i)
  {
    if (o.equals(TOPLEFT)) topLeft = l;
    else
```

The class implements **LwLayout** interface.
The four constants have to be used to determine a child location inside the container. A container component provides special method to add a child component that has the constants and the child as input.

The method is called with an owner container every time when a child component has been added. The layout manager determines how the component should be placed inside the container using input arguments that

```
  if (o.equals(TOPRIGHT)) topRight = l;
  else
  if (o.equals(BOTTOMLEFT)) bottomLeft = l;
  else
  if (o.equals(BOTTOMRIGHT)) bottomRight = l;
  else throw new IllegalArgumentException();
}

public void componentRemoved (Layoutable lw, int i)
{
  if (topLeft == lw) topLeft = null;
  else  if (topRight == lw) topRight = null;
        else if (bottomRight == lw) bottomRight = null;
              else  if (bottomLeft == lw) bottomLeft = null;
}

public Dimension calcPreferredSize(LayoutContainer t)
{
  int w1 = 0, w2 = 0, h1 = 0, h2 = 0;
  if (topLeft != null && topLeft.isVisible())  {
    Dimension ps = topLeft.getPreferredSize();
    w1 = ps.width;   h1 = ps.height;
  }
  if (topRight != null && topRight.isVisible())  {
    Dimension ps = topRight.getPreferredSize();
    w2 = ps.width;  h1 = Math.max(ps.height, h1);
  }
  if (bottomLeft != null && bottomLeft.isVisible())   {
    Dimension ps = bottomLeft.getPreferredSize();
    w1 = Math.max(ps.width, w1);  h2 = ps.height;
  }
  if (bottomRight != null && bottomRight.isVisible())   {
    Dimension ps = bottomRight.getPreferredSize();
    w2=Math.max(ps.width,w2);
    h2=Math.max(h2,ps.height);
  }
  return new Dimension (w1 + w2, h1 + h2);
}

public void  layout (LayoutContainer target)  {
  int w1 = 0, w2 = 0, h1 = 0, h2 = 0;
  Insets insets = target.getInsets();

  if (topLeft != null && topLeft.isVisible())   {
    Dimension ps = topLeft.getPreferredSize();
    w1 = ps.width;  h1 = ps.height;
  }
  if (topRight != null && topRight.isVisible())   {
    Dimension ps = topRight.getPreferredSize();
    w2 = ps.width; h1 = Math.max(ps.height, h1);
  }
  if (bottomLeft != null && bottomLeft.isVisible())   {
    Dimension ps = bottomLeft.getPreferredSize();
    w1 = Math.max(ps.width, w1); h2 = ps.height;
  }
  if (bottomRight != null && bottomRight.isVisible())   {
    Dimension ps = bottomRight.getPreferredSize();
    w2 = Math.max(ps.width, w2);
    h2 = Math.max(h2, ps.height);
  }
  Point offset = target.getLayoutOffset();
  if (topLeft != null && topLeft.isVisible())   {
    topLeft.setLocation(insets.left + offset.x,
                        insets.top + offset.y);
    topLeft.setSize(w1, h1);
  }
```

have been passed to the container. In compliance with the method implementation, it is impossible to add a child anywhere, except one of the container corners. It is necessary to use method *add(Object , LwComponent)* and first argument has to be defined as one of a constant of the layout manager.

The method is called with an owner container every time when a child component has been removed. The layout sets to null appropriate child component to exclude the child from the layouting process.

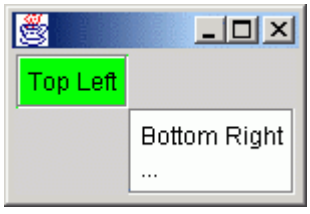
The method calculates preferred size of the owner container. The calculation algorithm should not use insets of the container the method provides “pure” preferred size.

The next method layouts child components for a given container. Take care, that the method should use insets data and *Point getLayoutOffset()* method of the owner (the method returns offset for children location and it uses to support scrolling concept). The set of images below shows different cases of the layout manager usage. Assume that we create following lightweight components:

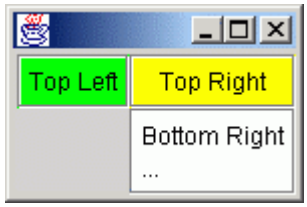
```
…
LwComponent c1 = new LwButton("Top Left");
c1.setBackground(Color.green);
LwComponent c2 = new LwButton("Top Right");
c2.setBackground(Color.yellow);
LwComponent c3 = new LwButton("Bottom Left");
c3.setBackground(Color.blue);
LwComponent c4 = new LwButton("Bottom Right\n...");
c4.setBackground(Color.white);
…
```

We can get following results for a lightweight container that uses the layout manager:

1. `root.add(LwMyLayout.TOP_LEFT, c1);`
   `root.add(LwMyLayout.BOTTOM_RIGHT, c4);`



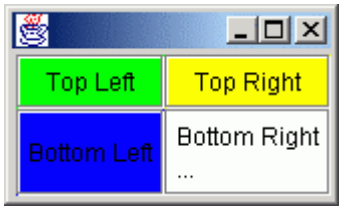
2. `root.add(LwMyLayout.TOP_LEFT, c1);`
   `root.add(LwMyLayout.TOP_RIGHT, c2);`
   `root.add(LwMyLayout.BOTTOM_RIGHT, c4);`



3. `root.add(LwMyLayout.TOP_LEFT, c1);`

```
    if (topRight != null && topRight.isVisible())    {
      topRight.setLocation(insets.left + w1 + offset.x,
                           insets.top + offset.y);
      topRight.setSize(w2, h1);
    }
    if (bottomRight != null && bottomRight.isVisible())   {
      bottomRight.setLocation(insets.left + w1 + offset.x,
                              insets.top + h1 + offset.y);
      bottomRight.setSize    (w2, h2);
    }
    if (bottomLeft != null && bottomLeft.isVisible())   {
      bottomLeft.setLocation(insets.left + offset.x,
                             insets.top + h1 + offset.y);
      bottomLeft.setSize    (w1, h2);
    }
  }
}
```

```
root.add(LwMyLayout.TOP_RIGHT, c2);
root.add(LwMyLayout.BOTTOM_LEFT, c3);
root.add(LwMyLayout.BOTTOM_RIGHT, c4);
```



…create focus manager

The library allows determining your own managers for different situations. For example, we need to implement “Tab” key functionality (focus “jumps” around all components). The sample below illustrates how the focus manager (that is provided with the library) can be extended. The new focus manager additionally handles key events (the basic lightweight focus manager handles mouse events and pass focus by mouse pressed event) and if “Tab” key have been pressed than the manager tries to pass focus to a next component on a panel (it is like the MS Windows focus strategy).

```
public class LwMyFocusMan
extends LwFocusManeger
implements LwKeyListener
{
  public void keyPressed (LwKeyEvent e) {}
  public void keyReleased(LwKeyEvent e) {}

  public void keyTyped(LwKeyEvent e)
  {
    if (e.getKeyCode() != KeyEvent.VK_TAB) return;
    LwComponent t = e.getLwComponent();
    LwContainer p =(LwContainer)t.getLwParent();
    int i = (p.indexOf(t) + 1)%p.count();
    for (;i < p.count(); i++)  {
      t = (LwComponent)p.get(i);
      if (t instanceof LwFocusListener)   {
        requestFocus(t);
        break;
      }
    }
  }
}
```

This class inherits lightweight focus manager to extend its functionality and implements key listener interface to handle key events. Draw attention that the manager will be added as a key listener to lightweight manager automatically during the lightweight library bootstrap.

This method is called with lightweight manager when a key event is performed for some lightweight component. This method checks if the “Tab” key have been pressed and if it is true than tries to find next focus owner. A lightweight component can be a focus owner if it implements focus listener interface.

Note: After that you should change focus manager class name in the lightweight properties file to **LwMyFocusMan**.

…use static objects

The library provides static object concept to decrease system resource usage. If you have a class whose instance can be shared, it makes sense to use an instance of the class as a static object. For this purpose you should describe the class in the lightweight properties file:

```
...
```

obj=<old list>, myStaticObject …	First of all, it is necessary to define new unique name for the static object in the static objects list section.
obj.myStaticObject.cl=<class>	You should define the static object class name by the section. The section contains a class name relatively **org.zaval.lw** package.
obj.myStaticObject.arg=<a1>,<a2>,…	This is optional section that contains list of arguments to use as input for the static object construct.
obj.myStaticObject.key=<key> …	This section defines a key that is used to fetch an instance of the static object.

After that, an instance of the static object can be got with *getStaticObject(Object)* static method of **LwManager** class.

…use lightweight layout managers

The library contains a set of layout managers that are used to layout container child components. Every lightweight container always uses layout manager. It is impossible to set layout manager to **null** value, but it doesn't mean that the child components cannot be laid out using set locations and sizes. There are several samples to illustrate lightweight layouts usage. First sample is very simple, it shows how to layout child components using its set sizes and locations:

… LwPanel panel = new LwPanel(); panel.setLwLayout(new LwRasterLayout(false));	Creates panel and sets **LwRasterLayout** as the panel layout manager. The layout constructor gets *false* argument as input to prevent preferred sizes usage to size the child components, in this case the child components sizes are defined by *setSize* methods.

Button button = new Button("Button");
button.setSize(100, 30);
button.setLocation(20, 20);
	Creates child components and defines size and location that will be used to layout the child.

panel.add (button); …	Adds the child to the container.

There is another sample that illustrates the *LwRasterLayout* usage:

… LwPanel panel = new LwPanel(); panel.setLwLayout(new LwRasterLayout(true));	Creates panel and sets **LwRasterLayout** as the panel layout manager. The layout constructor gets *true* argument as input that indicates that child components should be sized using its preferred size.
Button button = new Button("Button"); button.setLocation(20, 20);	Creates child components and defines the location that will be used to layout the child. The component's preferred size will be used to size the component by the layout manager.
LwLabel label = new LwLabel("Label"); label.setLocation(20, 100);	Creates child components and defines the location that will be used to layout the child.
label.setPSSize(100, -1);	Fixes the width of the label component that will be used by the layout manager. The height will be equal to preferred height of the child component.
panel.add (button); panel.add (label); …	Adds the child components to the container.

The next sample is a bit more complex. For example, we need to create something like a login panel:

…

```
public class LwLoginPanel
extends LwPanel
{
  public LwLoginPanel()
  {
    getViewMan(true).setBorder("br.etched");

     LwPanel p1 = new LwPanel();
     p1.setLwLayout(new LwGridLayout(2, 2));

     LwConstraints c = new LwConstraints();
     c.insets = new Insets(4,4,4,4);


     p1.add (c, new LwLabel("User name:"));
     p1.add (c, new LwTextField("", 8));
     p1.add (c, new LwLabel("Password:"));
     p1.add (c, new LwTextField("", 8));

    LwPanel p2 = new LwPanel();
    p2.setLwLayout(new LwBorderLayout(2, 2));

    LwPanel p3 = new LwPanel();
    p3.setLwLayout(new LwFlowLayout(Alignment.CENTER,
                                    Alignment.CENTER));
    p3.add(new LwButton("Login"));

    p2.add (LwBorderLayout.CENTER, p1);
    p2.add (LwBorderLayout.SOUTH, p3);

   setLwLayout(new LwFlowLayout(Alignment.CENTER,
                                Alignment.CENTER));
   add (p2);
  }
}
```

Sets border for the panel.

Creates child of **p2** panel and sets grid layout for the container.

Creates grid layout constraint and sets insets that is used as vertical and horizontal gaps for child components.

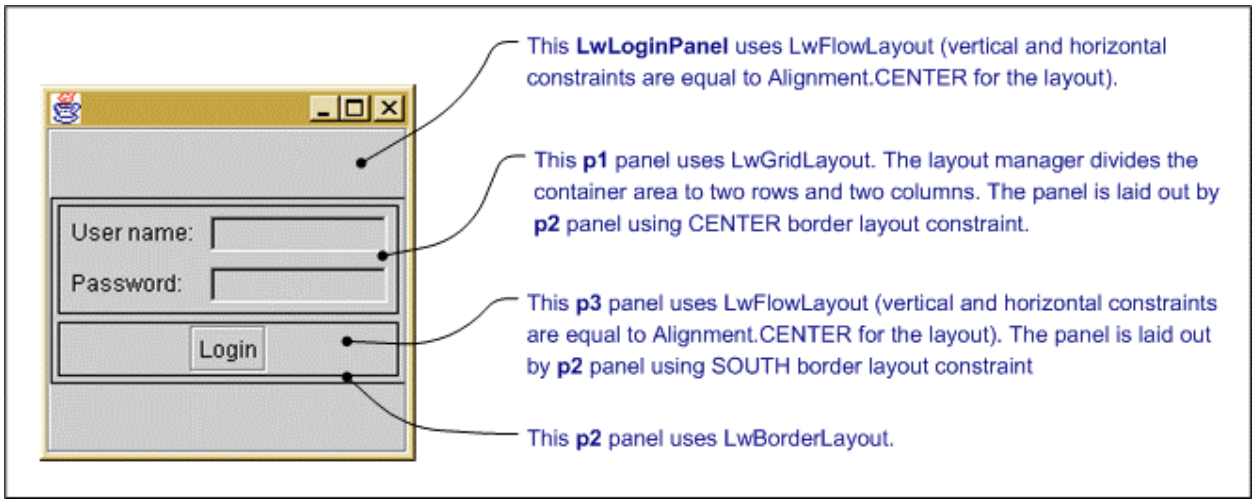
Creates child components and adds its to **p1** panel.

Creates **p2** child panel and sets grid layout for the container with appropriate vertical and horizontal gaps.

Creates child of **p2** panel and sets flow layout for the container.

Creates and adds login button to the **p3** panel. The button will be laid out in the center of the parent.

The application that uses the login panel and explanations are shown bellow (the real application will not have black borders, it's just for demonstrational purposes):



…customize tree item view

Lightweight tree view component provides ability to customize tree view items views. It means that you can bind any tree view item with the specified view to customize rendering process. For example,

it is necessary to use additional state icon for some tree view items. The sample below illustrates how it can be implemented:

```
public class CustomViewProvider
implements LwViewProvider
{

  public LwView getView(Drawable d, Object obj)
  {
    LwPanel panel = new  LwPanel();
    panel.setLwLayout(new LwFlowLayout());
    LwImage  icon = new LwImage("icon.gif");
    String v = (String) ((Item)obj).getValue();
    LwLabel   title = new LwLabel(v);
    panel.add (icon);
    panel.add (title);
    return new LwCompRender(panel);
  }
}

…

Item root = new Item("root");
Tree data = new Tree(root);
data.add (root, new Item("Child 1"));
data.add (root, new Item("Child 2"));
LwTree tree = new LwTree(data);

tree.setViewProvider(new CustomViewProvider());

…
```

First of all it is necessary to implement your own view provider. This is very simple interface that creates view for the specified component and the given tree model value.

The method creates panel where image and label components are laid out by the flow layout. It is supposed that tree data model stores strings as the tree items values and the value is used as content for the label component. After that the method creates and returns component render with the panel as the rendered target.

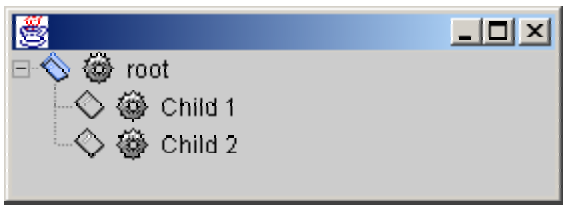
The next step shows how to apply the new view provider for a concrete tree view component.

Creates and fills the tree model.

Creates tree view component with the tree data model.

Sets the new view provider for the tree view component.

The sample is show below:



…customize grid cell view

Lightweight grid component provides ability to customize cells views. It means that you can bind any grid cell with the specified view to customize rendering process. For example, you have a grid column that renders boolean data ("Yes" or "No") and you would like to render it as check box view. The sample below illustrates how it can be implemented:

```
public class CustomViewProvider
implements LwGridViewProvider
{

  public LwView getView(int row, int col, Object o)
  {
    String value = (String)o;
    if (col !=2)
      return new LwTextRender(value==null?""
                                          :value);

    int type =value != null && value.equals("Yes")
```

First of all it is necessary to implement own view provider. This interface defines views for the grid cells, vertical and horizontal alignments for the views and cells background colors.

It is supposed that grid data model stores strings as the cells values and it is supposed that column number two is used to store boolean string value ("Yes" or "No"). The method returns appropriate box view if the column has number two and returns text render for others.

```
                    ?LwBoxView.CHECK_ON
                    :LwBoxView.CHECK_OFF;
    return new LwBoxView(type);
  }

  public int getXAlignment(int row, int col)  {
    return Alignment.CENTER;
  }

  public int getYAlignment(int row, int col)  {
    return Alignment.CENTER;
  }

  public Color getCellColor (int row, int col) {
    return null;
  }
}

…

Matrix data = new Matrix(10, 3);
data.put (1, 2, “Yes”);
…
LwGrid grid = new LwGrid(data);

grid.setViewProvider(new CustomViewProvider());
…
```

The method returns horizontal alignment that is used to align the grid view inside the grid cell area.

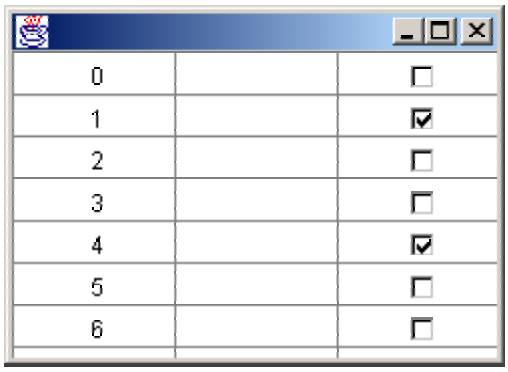
The method returns vertical alignment that is used to align the grid view inside the grid cell area.

The method returns a color that should be used to fill the cell background. In this case the method returns **null**, it means that grid background will be used.

The next step is applying the new grid view provider for a concrete grid component. Firstly the grid data model is created and filled.
Creates the grid component with the data model.

Sets the new grid views provider for the component.

The sample application is shown bellow:



…customize grid cell editor

Lightweight grid component provides ability to customize cells editing process. It means that you can use a lightweight component to edit a cell value. The sample is like previous, but in this case it is necessary to organize editing for boolean cells using checkbox component. The sample below illustrates how it can be implemented:

```
public class CustomEditorProvider
implements LwEditorProvider
{


  public LwComponent getEditor(int r, int c, Object o)
  {
    if (c !=2) return null;

    String value = (String)o;
    boolean state = value!=null&&value.equals(“Yes”);
```

First of all it is necessary to implement own editor provider. This interface defines a component that should be used as the cell editor, defines how to fetch a model value from the editor component and defines how the editor should be initialized.

It is supposed that grid data model stores strings as the cells values and it is supposed that column number two is used to store boolean string value (“Yes” or “No”). The method returns **null** if this is not boolean column, it means that the column cannot be edited. In a case when the column number is two the method initializes and returns checkbox component

```
    LwCheckbox box = new LwCheckbox();
    box.setFocusComponent(null);
    box.setState(state);
    return box;
  }


  public Object fetchEditedValue(int row,
                                 int col,
                                 LwComponent c)
  {
      return ((LwCheckbox)c).getState()
                              ? "Yes"
                              : "No"  ;
  }

  public boolean shouldStartEdit (int row,
                                  int col,
                                  LwAWTEvent e)
  {
     return true;
  }

}

…

Matrix data = new Matrix(10, 3);
data.put (1, 2, "Yes");
…
LwGrid grid = new LwGrid(data);

grid.setViewProvider(new CustomViewProvider());
…
```

as the cell editor.

The method fetches the data model value for the specified cell from the given editor component (that has been used to edit the cell value). In this case, a checkbox state is converted to appropriate string value.

The method checks if it is necessary to initiate editing process for the specified cell and for the given lightweight event. In this case the editing process will be initiated immediately.

The next step is applying the new grid editor provider for a concrete grid component. Firstly the grid data model is created and filled.

Creates the grid component with the data model.

Sets the new grid editor provider for the component.

…use mask text field

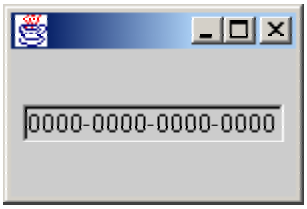
The sample illustrates lightweight masked text field usage. First simple sample shows creating credit number input filed (Visa Electron). The credit number contains four numeric slots, every slots contains four figures and the slots are separated by "-" character. For example "7876-9873-3435-1237". The application below provides input field to enter the credit number type:

```
public class LwCreditInputApp
{
  public static void main(String[] args)
  {
    LwFrame frame = new LwFrame();
    frame.setSize(150, 100);

    LwMaskTextField tf = new LwMaskTextField() ;
    tf.setMask("nnnn-nnnn-nnnn-nnnn");



    LwContainer root = frame.getRoot();
    LwLayout l = new LwFlowLayout(Alignment.CENTER,
                                  Alignment.CENTER));
    root.setLwLayout(l);
    root.add (tf);

    frame.setVisible(true) ;
  }
}
```

Creates the application frame.

Creates mask text field.
Specifies the mask. The mask contains four numeric slots separated by "-" character and every slot consists of four ciphers.

Adds the mask component to the application.
The result application is shown below:



The next sample illustrates creation of the input date field. The library has **LwDateMaskValidator** class that provides ability to customize input for date fields. For example, the date format is like "01/Jun/2001", it is necessary to provide input field that supports the date format. The table below shows the date input field creating:

```
public class LwCreditInputApp
{
  public static void main(String[] args)
  {
    LwFrame frame = new LwFrame();
    frame.setSize(150, 100);

    LwMaskTextField tf = new LwMaskTextField() ;
    tf.setValidator(new DateMaskValidator());


    tf.setMask("dd/MMM/yyyy");




    LwContainer root = frame.getRoot();
    LwLayout l = new LwFlowLayout(Alignment.CENTER,
                                  Alignment.CENTER));
    root.setLwLayout(l);
    root.add (tf);

    frame.setVisible(true) ;
  }
}
```

Creates the application frame.

Creates mask text field.
Sets the date validator that knows how validate the date input.

Specifies the mask. The mask contains day, month, year slots separated by "/" character. Draw attention that the month slot represents short month name (for example "Feb", "Jun" and so on).

Adds the mask component to the application.
The result application is shown below (the first window is date mask field just after starting application and the second window is date mask after inputting some data):

…customize mask validation

The sample illustrates how the mask validation process can be customized. For example, we need to create input field that can be filled with hex numeric only. For this purpose we should implement our own mask validator. The validator uses 'h' character as the tag name and defines HEX_TYPE mask element type. The mask converts every hex letter to upper case. The table below shows the mask validator code and comments:

```
public class HexMaskValidator
implements MaskValidator
{
  public static final int HEX_TYPE=1;

  public boolean isHandledTag  (char tag) {
    return tag == 'h';
  }
```

Defines hex mask element type.

Defines mask tags that are handled by the validator (in our case it is 'h' character).

```
  public int getTypeByTag(char tag) {
     return tag=='h'?HEX_TYPE:UNDEF_TYPE;
  }

  public char getBlankChar(char tag) {
    return 'F';
  }

  public boolean isValidValue  (MaskElement e,
                                String newValue)
  {
    char[] buf = newValue.toCharArray();
    for (int i=0;i<buf.length; i++)
    {
      if (!Character.isDigit(buf[i]) &&
          buf[i] != 'A' && buf[i] != 'B' && buf[i] != 'C' &&
          buf[i] != 'D' && buf[i] != 'E' && buf[i] != 'F'       )
        return false;
    }
    return true;
  }

  public String  completeValue (MaskElement e,
                                String newValue)
  {
    return  newValue. toUpperCase();
  }
}
```

Defines the type for the specified tag. In our case the HEX_TYPE is bound with 'h' tag.

Defines default character that is used to fill mask value for not inputted characters.

Validates the specified value for the mask element. Since the mask validator supports only one mask element type, the method doesn't tests what the specified mask element type is, in our case the type will have always hex type.

The new value is valid if it consists of 0 - 9 or A – F characters.

Converts the new value to upper case.

The next table illustrates validator usage. The sample application creates hex input field that consists of four hex ciphers:

```
public class LwCreditInputApp
{
  public static void main(String[] args)
  {
    LwFrame frame = new LwFrame();
    frame.setSize(150, 100);

    LwMaskTextField tf = new LwMaskTextField() ;
    tf.setPSSize(50, -1);
    tf.setValidator(new Validator());

    tf.setMask("hhhh");


    LwContainer root = frame.getRoot();
    LwLayout l = new LwFlowLayout(Alignment.CENTER,
                                  Alignment.CENTER));
    root.setLwLayout(l);
    root.add (tf);

    frame.setVisible(true) ;
  }
}
```

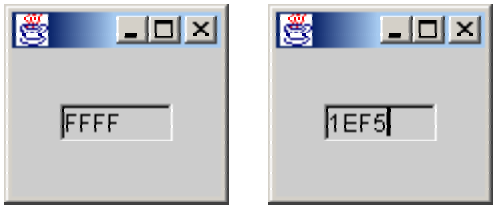
Creates the application frame.

Creates mask text field.

Sets the hex validator.

Specifies the mask. The mask defines input for hex numeric that contains four ciphers.

Adds the mask component to the application.
The result application is shown below (the first window is hex mask field just after starting application and the second window is hex mask after inputting some data):



…use desktop window

The sample illustrates the lightweight window usage. For example, we want to show window on the lightweight desktop with editor area. To open a new editor window you should press “Show Window” button. The sample bellow illustrates how it can be implemented:

```
public class LwWinShower
implements LwActionListener
{
  public void actionPerformed(LwActionEvent e)
  {

     LwComponent src = (LwComponent)e.getSource();
     LwDesktop desk = LwToolkit.getDesktop(src);

     LwWindow win = new LwWindow();
     win.setLocation(30, 30);
     win.setSize (300, 300);
     win.setLwLayout(new LwBorderLayout());
     LwTextField editor = new LwTextField();
     win.add(LwBorderLayout.CENTER, editor);

     desk.openWin(win, null);


     desk.activateWin(win);
  }

  public static void main(String[] args)
  {
    LwFrame frame = new LwFrame();
    frame.setSize(600, 600);
    LwButton button = new LwButton(“Show Window”);
    button.addActionListener(new LwWinShower());
    frame.getRoot().add(button);
    frame.setVisible(true);
  }
}
```

The class implements **LwActionListener** interface to get action event generated by “Show Window” button.

This method is invoked whenever “Show Window” button has been pressed.

Gets source of the event.
Gets desktop where the source component is resided.

Creates lightweight window class, locates and sizes it.

Sets border layout for the window to layout the text field component.

Opens the window on the desktop. The parent of the window is null.

Activates the window.

This main method creates the lightweight application. Firstly **LwFrame** is created and initialized, than “Show Window” button is created and the **LwWinShower** are registered as the button listener.

Support available

If you feel this document does not cover any important issues – let us know. You can request additional teaching materials and support from our team – drop a mail to support@zaval.org with subject line set to “LwVCL: teaching materials”.