Light-Weight

Version 4.50

Visual Components Library

Teaching Materials

Zaval Creative Engineering Group
http://www.lwvcl.com

Contents

Introduction....3
How to... 3
…create a component 3
…create a view....3
…create your own render 4
…create layout manager....5
…create component info manager 7
…use static objects 8
…use lightweight layout managers 9
…customize tree item view....11
…customize grid cell view....11
…customize grid cell editor....12
…use mask text field 13
…customize mask validation....14
…use desktop window....16
…use tool-tips 17
…use timer component....18
…use tree grid component....18

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:

```
class LwSampleCanvas
extends LwCanvas
implements LwMouseListener
{
   private boolean isInside;

  public LwSampleCanvas () {
    setBackground (Color.gray);
  }
```

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

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

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

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

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

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

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

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

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

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.
See the "**org/zaval/lw/samples/LwComponentSample.java**" sample.

…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:

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

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


  public LwCustomView() {
    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, Layoutable 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.

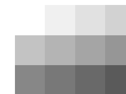
Field “basic” defines a color that will be used as starting color, “width” and “height” fields define size of a color box. “rgbIncrement” field determines a value that is used to form next color.

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
LwCustomView());
   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.

See the “**org/zaval/lw/samples/LwCustomView.java**” sample.

…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

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

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.

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

We inherit abstract render class and implement paint() method. Note: The **LwRender** class inherits **LwView** class.

```
  public LwCustomRender(Object target) {
    super(target);
  }
```

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.

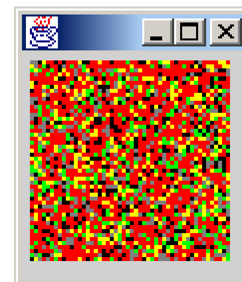
```
  protected Dimension calcPreferredSize() {
    int[][] target = (int[][])getTarget();
    return new Dimension(width*target.length,
                         height*target[0].length);
  }
```

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.

```
  public void paint (Graphics g, int x, int y,
                     int w, int h, Layoutable 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;
    }
  }
}
```

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.

See the “**org/zaval/lw/samples/LwRenderSample.java**” sample.

…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.

```
class LwCustomLayoutSample implements LwLayout {
public static final Object TOPLEFT=new Integer(4);
```

The class implements **LwLayout** interface.
The four constants have to be used to determine a child

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

```
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;
```

location inside the container. A container component provides special method to add a child component that has the constants and the child as input.

```
public void componentAdded(Object o,Layoutable l,int i)
{
  if (o.equals(TOPLEFT)) topLeft = l;
  else
  if (o.equals(TOPRIGHT)) topRight = l;
  else
  if (o.equals(BOTTOMLEFT)) bottomLeft = l;
  else
  if (o.equals(BOTTOMRIGHT)) bottomRight = l;
  else throw new IllegalArgumentException();
}
```

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 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.

```
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;
}
```

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.

```
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);
}
```

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:

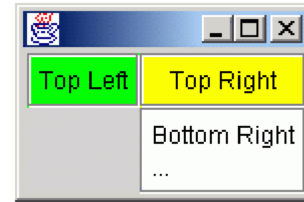
1. `root.add(LwCustomLayoutSample.TOP_LEFT, c1);`
   `root.add(LwCustomLayoutSample.BOTTOM_RIGHT, c4);`

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

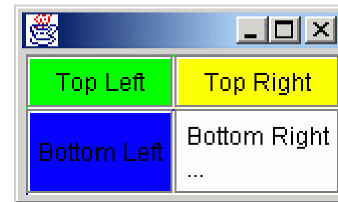
```
public void  layout (LayoutContainer target)  {
  int w1 = 0, w2 = 0, h1 = 0, h2 = 0;
  Insets insets = target.getInsets();

  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();
```

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

```
      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);
    }
    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);
    }
  }
}
```



3.
```
root.add(LwCustomLayoutSample.TOP_LEFT, c1);
root.add(LwCustomLayoutSample.TOP_RIGHT, c2);
root.add(LwCustomLayoutSample.BOTTOM_LEFT, c3);
root.add(LwCustomLayoutSample.BOTTOM_RIGHT, c4);
```



See the “**org/zaval/lw/samples/LwCustomLayoutSample.java**” sample.

…create component info manager

The library allows determining your own managers for different situations. For example, we need to get information about a lightweight component where the mouse cursor is located by pressing “F1” key. The information includes the component class name, location and size. The sample below illustrates how the task can be solved by creation the special manager. The manager handles key and mouse events and if “F1” key has been pressed than the manager shows information window where the component class name, location and size are displayed.

```
public class LwCompInfoManager
implements LwMouseListener, LwKeyListener,
LwMouseMotionListener, LwManager
{
  private LwComponent target;
```

This class implements **LwMouseListener**, **LwKeyListener**, **LwMouseMotionListener** to listen all performed key and mouse events. The class implements **LwManager** interface since this is LwVCL manager.

```
  private int lastX, lastY;
```

Defines two fields to store current cursor location.

```
  public void mouseMoved (LwMouseEvent e)
  {
    lastX = e.getAbsX();
    lastY = e.getAbsY();
  }
```

The method is called whenever the mouse cursor location has been changed. The manager stores the cursor location in the special fields.

```
  public void mouseEntered (LwMouseEvent e) {
    target = e.getLwComponent();
  }
```

The method is called whenever the mouse cursor enters into a component.

```
  public void mouseExited  (LwMouseEvent e)
  {
    if (target != null)
    {
```

The method is called whenever the mouse cursor exits the component area. We use the method to hide the information window for the component.

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

Code	Description

`LwDesktop d = LwToolkit.getDesktop(target);`
`LwWinLayer wl=(LwWinLayer)d.getLayer("win");`
`wl.removeAll();`
`target = null;`
`}`
`}`	

`public void keyPressed (LwKeyEvent e)`	The method is called whenever a key has been pressed.

`{`
`if (target != null &&`
`e.getKeyCode() == KeyEvent.VK_F1)`
	Tests if the target component is not null and the "F1" key has been pressed.

`{` `LwDesktop d = LwToolkit.getDesktop(target);`	Gets the desktop where the target component lives.
`LwWinLayer wl= (LwWinLayer)d.getLayer("win");`	Gets the window layer for the desktop.
`wl.removeAll();`	Closes all opened windows on the window layer (to close previous information window if it has been shown).
`LwLabel lab = new LwLabel(new Text(""));`	Creates the information window.
`lab.setBackground (new Color (220, 220, 220));`	Sets the information window background.
`lab.getViewMan(true).setBorder("br.plain");`	Sets the information window border.

`lab.setText("Class: "+`
`target.getClass().getName()+"\n"+`
`"x = " + target.getX() + ", y = " +`
`target.getY() + "\n" +`
`"width = "+target.getWidth()+", height = "+`
`target.getHeight());`
	Sets the information window content that includes the target component class name, location and size.

`Dimension ps = lab.getPreferredSize();`	Gets the information window preferred size.
`lab.setSize(ps.width, ps.height);`	Sizes the information window according to its preferred size.
`lab.setLocation (lastX, lastY);`	Locates the window to the current mouse cursor location.

`wl.add (LwWinLayer.INFO_WIN, lab);`
`}`
`}`
	Shows the information window.

`public void keyReleased(LwKeyEvent e) {}`
`public void keyTyped(LwKeyEvent e) {}`
`public void startDragged(LwMouseEvent e) {}`
`public void endDragged (LwMouseEvent e) {}`
`public void mouseDragged(LwMouseEvent e) {}`
`public void mousePressed (LwMouseEvent e) {}`
`public void mouseReleased(LwMouseEvent e) {}`
`public void mouseClicked (LwMouseEvent e) {}`	

`public void dispose() {}` `}`	Implements manager interface method.

After the manager has been implemented it is necessary to add the manager description into "lw.properties" file. Open the file, you will see "obj" key at the beginning of the file. Add your manager identification key, for example "info" at the end of the managers list as follow:

```
obj=paint,focus,cursor,clip,pop,tt,info
```

Next, add the "info" key to describe the manager class as follow:

```
info.cl=org.zaval.lw.samples.LwCompInfoManager
```

(**Note**: in your case the class name can be different)

See the "**org/zaval/lw/samples/LwCompInfoManager.java**" sample.

…use static objects

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

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:

Code	Description
… *myStaticObjectKey=<class>:<a1>,<a2>,…* …	You should specify a property name that will be used as the static object key. You should define the static object class name as the property value just after “:” character. The class name is relative to **org.zaval.lw** package if package has not been specified. You can specify arguments list just after class name plus shift. This is optional part that contains list of arguments to use as input for the static object constructor.

After that, an instance of the static object can be got with *getStaticObj(String)* static method of the **LwToolkit** 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:

Code	Description
… 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:

Code	Description
… 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.

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

…

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

```
…

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(LwToolkit.CENTER,
                                   LwToolkit.CENTER));
   p3.add(new LwButton("Login"));

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

  setLwLayout(new LwFlowLayout(LwToolkit.CENTER,
                               LwToolkit.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 inset that is used as vertical and horizontal gaps for child components.

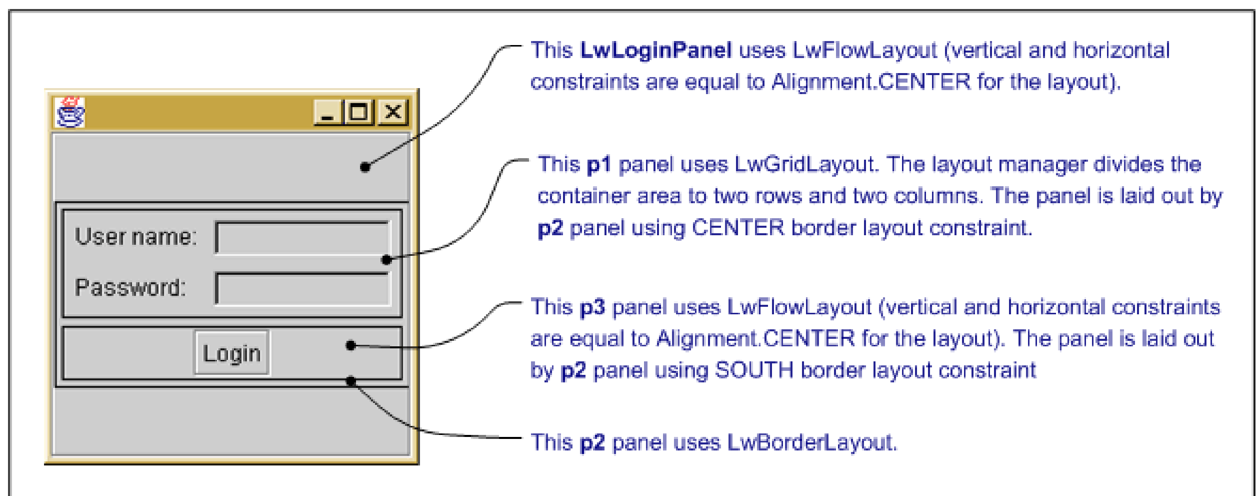
Creates child components and adds its to **p1** panel.

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 below (the real application will not have black borders, it's just for demonstrational purposes):



© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

See the “**org/zaval/lw/samples/LwLayoutUsageSample.java**” sample.

...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:

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.

```
class LwCustomViewProvider
implements LwViewProvider
{

  public LwView getView(LwComponent 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);
  }
}
```

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

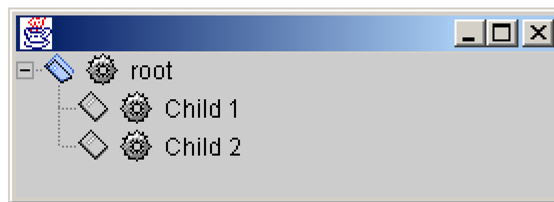
Creates and fills the tree model.

Creates tree view component with the tree data model.

Sets the new view provider for the tree view component.

```
…

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 LwCustomViewProvider());

…
```

The sample is show below:



See the “**org/zaval/lw/samples/LwTreeCustomViewSample.java**” sample.

...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:

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.

```
public class LwGridBooleanView
implements LwGridViewProvider
{
```

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

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

  String type =value != null && value.equals("Yes")
                            ?"check.on"
                            :"check.off";
    return LwToolkit.getView(type);
  }
```

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.

The views are fetched by the **LwToolkit.**

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

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

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

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

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

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.

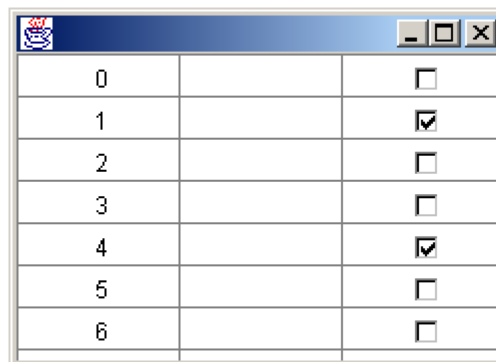
```
…

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

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.

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

Sets the new grid views provider for the component.

The sample application is shown below:



See the "**org/zaval/lw/samples/LwGridBooleanView.java**" sample.

…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:

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

```
public class LwGridBooleanEditor
implements org.zaval.lw.grid.LwEditorProvider
{
```

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.

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

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 as the cell editor.

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

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.

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

}
```

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.

```
…

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

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

```
…
LwGrid grid = new LwGrid(data);
```

Creates the grid component with the data model.

```
grid.setViewProvider(new LwGridBoleanView());
grid.setEditorProvider(new LwGridBoleanEditor());
…
```

Sets the new grid view (see the previous sample) and editor providers for the component.

See the "**org/zaval/lw/samples/LwGridBooleanEditor.java**" sample.

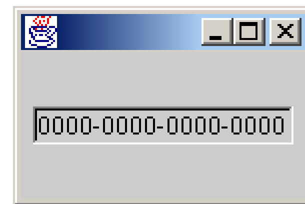
…use mask text field

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" value. The application below provides input field to enter the credit number type:

```
public class LwCreditCardInputSample
{
  public static void main(String[] args)
  {
```

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

Code	Description
`LwFrame frame = new LwFrame();` `frame.setSize(150, 100);`	Creates the application frame.
`LwMaskTextField tf = new LwMaskTextField() ;` `tf.setMask("nnnn-nnnn-nnnn-nnnn");`	Creates mask text field. Specifies the mask. The mask contains four numeric slots separated by "-" character and every slot consists of four ciphers.

`LwContainer root = frame.getRoot();`
`LwLayout l = new LwFlowLayout(LwToolkit.CENTER, LwToolkit.CENTER));`
`root.setLwLayout(l);`
`root.add (tf);`
 	
`frame.setVisible(true) ;`
`}`
`}`
	Adds the mask component to the application.
	The result application is shown below:
	

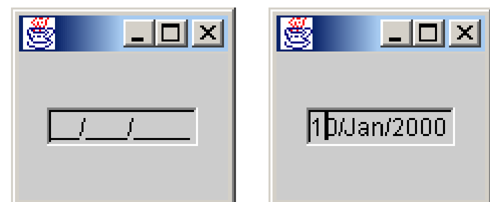
See the "**org/zaval/lw/samples/LwCreditCardInputSample.java**" sample.

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:

Code	Description

`public class LwDateInputSample`
`{`
`public static void main(String[] args)`
`{`	

`LwFrame frame = new LwFrame();` `frame.setSize(150, 100);`	Creates the application frame.
`LwMaskTextField tf = new LwMaskTextField() ;` `tf.setValidator(new DateMaskValidator());`	Creates mask text field. Sets the date validator that knows how validate the date input.
`tf.setMask("dd/MMM/yyyy");`	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).

`LwContainer root = frame.getRoot();`
`LwLayout l = new LwFlowLayout(LwToolkit.CENTER, LwToolkit.CENTER));`
`root.setLwLayout(l);`
`root.add (tf);`
 	
`frame.setVisible(true) ;`
`}`
`}`
	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):
	

See the "**org/zaval/lw/samples/LwDateInputSample.java**" sample.

<u>…customize mask validation</u>

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

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:

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

Defines hex mask element type.

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

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;
  }
```

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

```
  public char getBlankChar(char tag) {
    return ‘F’;
  }
```

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

```
  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;
  }
```

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.

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

Converts the new value to upper case.

The next table illustrates the 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);
```

Creates the application frame.

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

Creates mask text field.

```
   tf.setValidator(new HexMaskValidator());
```

Sets the hex validator.

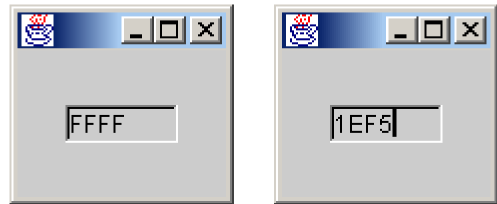
```
   tf.setMask(“hhhh”);
```

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

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

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

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

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):



See the "**org/zaval/lw/samples/LwCustomInputSample.java**" sample.

<u>…use desktop window</u>

This 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 below illustrates how it can be implemented:

```
public class LwWinShowerSample
implements LwActionListener
{
  public void actionPerformed(Object src, Object data)
  {

    LwDesktop desk = LwToolkit.getDesktop(src);
LwWinLayer
wl=(LwWinLayer)desk.getLayer(LwWinLayer.ID);

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

    wl.add(LwWinLayer.MDI_WIN, win);

    wl.activate(win);
  }

  public static void main(String[] args)
  {
    LwFrame frame = new LwFrame();
    frame.setSize(600, 600);
    LwButton button = new LwButton("Show Window");
    button.setSize(120, 25);
    button.addActionListener(new
LwWinShowerSample());
    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.
Gets the window layer where windows can be resided

Creates lightweight window class, locates and sizes it.

Sets border layout for the root container of the window to layout the text field component.

Adds the text area as the window content.

Opens MDI window on the desktop.

Activates the window.

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

See the "**org/zaval/lw/samples/LwWinShowerSample.java**" sample.

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

…use tool-tips

This sample illustrates the lightweight tool-tips usage. For example, we want to show current mouse cursor location for the specified component via tool-tip. There are two ways to implement this feature:

- If this is a new component it is possible to implement **TooltipInfo** interface.
- If it is necessary to use the tool-tip for existing lightweight component it is possible to use **LwTooltipMan** manager to specify the **TooltipInfo** interface for the component.

The sample below illustrates both ways:

First implementation way

The class implements **TooltipInfo** interface to define tool-tip component.

```
class LwTooltipCanvas
extends LwCanvas
implements TooltipInfo
{
```

This method returns a label component that will be used as a tool-tip that indicates current mouse cursor location.

```
  public LwComponent getTooltip(LwComponent c,
                                int x, int y)
  {
    LwLabel l = new LwLabel("[x=" + x + ",y="+y + "]");
    l.setBackground((Color)LwToolkit.getStaticObj(
                                        "tt.bgcolor"));
    return l;
  }

  public static void main(String[] args)
  {
```

Creates lightweight window class, locates and sizes it.

```
    LwFrame frame = new LwFrame();
    frame.setSize(200, 200);
```

Creates, locates and sizes the component that shows mouse cursor location using tool-tip.

```
    LwTooltipCanvas tp = new LwTooltipCanvas ();
    tp.setSize(100, 100);
    tp.getViewMan(true).setBorder("br.plain");
```

Adds component to the root and shows the window.

```
    frame.getRoot().add(tp);
    frame.setVisible(true);
  }
}
```

Second implementation way

In this case we define the tool tip for a button component:

The class implements **TooltipInfo** interface to define tool-tip component.

```
public class LwTooltipSample2
implements TooltipInfo
{
```

This method returns a label component that will be used as a tool-tip that indicates current mouse cursor location.

```
  public LwComponent getTooltip(LwComponent c,
                                int x, int y)
  {
    LwLabel l = new LwLabel("[x=" + x + ",y="+y + "]");
    l.setBackground(((Color)LwToolkit.getStaticObj(
                                        "tt.bgcolor"));
    return l;
  }

  public static void main(String[] args)
  {
```

Creates lightweight window class, locates and sizes it.

```
    LwFrame frame = new LwFrame();
    frame.setSize(300, 300);
```

Creates, locates and sizes the button component.

```
    LwButton b = new LwButton ("Test");
    b.setLocation(50, 50);
```

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

Code	Description
`b.setSize(100, 30);` `LwToolkit.getTooltipManager().setTooltip(b, new LwTooltipSample2 ());`	Creates and binds the specified tool-tip interface with the button component.

`frame.getRoot().add(b);`
`frame.setVisible(true);`
`}`
`}`
	Adds button to the root and shows the window.

See the "**org/zaval/lw/samples/LwTooltipSample1.java**" and "**org/zaval/lw/samples/LwTooltipSample2.java**" samples.

<u>…use timer component</u>

This sample illustrates the timer component usage. This component is not a GUI component that provides ability to register **Runnable** interfaced classes to be executed periodically. The sample implements a lightweight canvas that periodically changes its background color:

Code	Description

`public class LwTimerSample`
`extends LwCanvas`
`implements Runnable`
`{`
	This class extends **LwCanvas** and implements **Runnable** interface to have ability be executed by the timer component.

`int r,g,b,d = 3;`	The field defines red, blue, green background color entries and a value that will be used as increment/decrement for the entries.

`public void run ()`
`{`
`if (r + d >= 255) d = -3;`
`else`
`if (r + d < 0) d = 3;`
`r += d;`
`g += d;`
`b += d;`
`setBackground(new Color(r, g, b));`
`}`
	This is **Runnable** method implementation. The method will be called by the timer component regularly in the specified time frame. It increments or decrements current RGB entries of the background color and sets the new color as the background.

`public static void main(String[] args)` `{`	
`LwFrame frame = new LwFrame();` `frame.setSize(300, 300);`	Creates lightweight window class, locates and sizes it.

`LwTimerSample ts = new LwTimerSample();`
`ts.setLocation(50, 50);`
`ts.setSize(100, 100);`
	Creates the canvas instance, locates and sizes it.

`Timer.getTimer(false).add(ts, 20, 20);`	Gets shared instance of the timer and adds the canvas to be executed in the specified time with the specified period.

`frame.getRoot().add(ts);`
`frame.setVisible(true);`
`}`
`}`
	Adds canvas to the root and shows the window.

See the "**org/zaval/lw/samples/LwTimerSample.java**"

<u>…use tree grid component</u>

This sample application is a simple lightweight library resources viewer. The viewer shows information about managers and static objects that are used by the library at the moment. Taking into account complexity of the task it would be described as a several sub-tasks:

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

1. The first sub-task creates a tree model for the tree grid component according to the list of lightweight managers and static objects (components views). So, the tree model describes two main branches: lightweight managers and static objects.
2. The second sub-task illustrates how the tree model items can be bound with the tree grid model items. The task shows classes names that are bound with appropriate lightweight managers and static objects in the second tree grid column.
3. And the last sub-task shows how we can customize the tree grid cells rendering to show static objects views "faces" columns.

```
public class LwPropertiesViewer
extends LwPanel
{
 public LwPropertiesViewer(Hashtable p)
 {
```

The constructor gets a hashtable as an input argument. The hashtable contains properties that are read from the "lw.properties" file.

```
   TreeModel  treeModel = createTreeModel(p);
   LwTreeGrid treeGrid  = createTreeGrid (treeModel);
```

Creates tree model.
Creates tree grid component with the specified tree model.

```
   for (int i=0; i < treeGrid.getCols(); i++)
     treeGrid.setColWidth(i, 180);
```

Sizes the tree grid columns

```
   add (LwBorderLayout.CENTER,
          new    LwScrollPan(treeGrid));
 }
```

Adds the created tree component in the scroll panel and adds the scroll panel to the panel.

```
 protected LwTreeGrid createTreeGrid(TreeModel m)
 {
```

This method creates tree grid component with the specified tree model.

```
   LwTreeGrid treeGrid = new LwTreeGrid(m);
   LwGridCaption cp = new LwGridCaption(treeGrid);
   caption.putTitle (0, "Object Key");
   caption.putTitle (1, "Class Name");
   caption.putTitle (2, "View");
```

Creates tree grid component.
Creates caption for the tree grid component.
Defines three titles as a tree grid columns captions.

```
   treeGrid.add (LwGrid.TOP_CAPTION_EL, cp);
   return treeGrid;
 }
```

Adds the created caption to the tree grid component.
Returns the created tree grid component.

```
 protected TreeModel createTreeModel(Hashtable p)
 {
```

This method creates tree model for the specified properties.

```
   Item root = new Item("root");
   TreeModel treeModel = new Tree(root);
   Item man  = new Item("Managers");
   Item sobj = new Item("Static Objects");
```

Creates a root of the tree model.
Creates tree model with the specified root.
Creates two branches where the first will contain all managers and second will contain all static objects.

```
   treeModel.add (root, man);
   treeModel.add (root, sobj);
   String mans = (String)p.get("man");
   StringTokenizer st=new StringTokenizer(mans, ",");
   while (st.hasMoreTokens())
     treeModel.add(man, new Item(st.nextToken()));
```

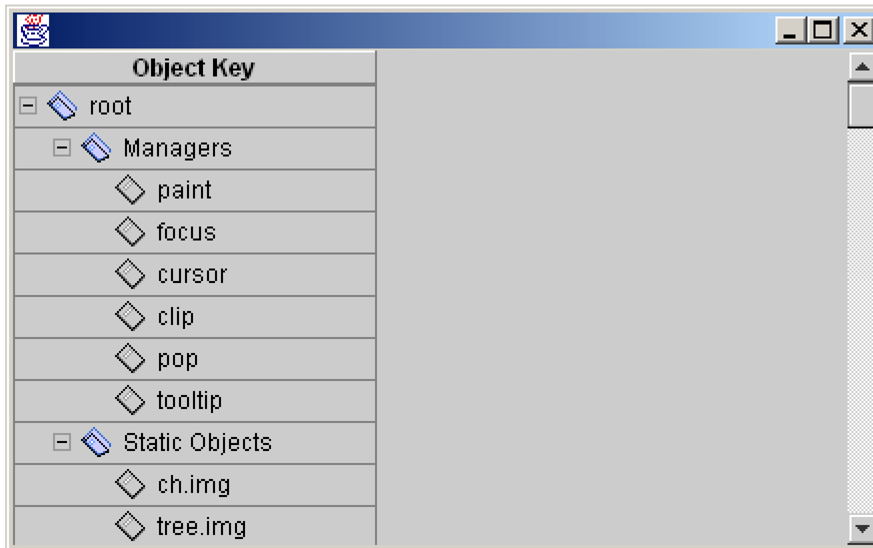
Adds the two branches as root child items of the tree model.
Gets a list of managers sections.
Fetches sections names and adds them as child items of the manager branch in the tree model.

```
   st = new StringTokenizer((String)p.get("obj"), ",");
   while (st.hasMoreTokens())
   {
      String key = p.get("obj." + st.nextToken() + ".key");
      treeModel.add(sobj, new Item(key));
   }

   return treeModel;
 }
}
```

Fetches static objects' names and adds their keys as child items of the static objects branch in the tree model.

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

The result is shown below:



At the next step we will add second column to the tree grid that will show a class name for corresponding managers and static objects. For this purpose we have to add *createTreeGridModel* method to the **LwPropertiesViewer** class as follows (the red color indicates changes in existing code):

```
public class LwPropertiesViewer
extends LwPanel
{
 public LwPropertiesViewer(Hashtable p)
 {
  TreeModel  treeModel = createTreeModel(p);
  LwTreeGrid treeGrid  = createTreeGrid (treeModel);
  createTreeGridModel(treeGrid);
  for (int i=0; i < treeGrid.getCols(); i++)
    treeGrid.setColWidth(i, 180);
  add (LwBorderLayout.CENTER,
        new    LwScrollPan(treeGrid));
 }
 …
 protected void createTreeGridModel(LwTreeGrid grid)
 {
  MatrixModel matrixModel = grid.getModel();
  TreeModel   treeModel   = grid.getTreeModel();

  int row = 2;
  Item man = treeModel.getChildAt(treeModel.getRoot(), 0);

  for (int i=0;i<treeModel.getChildrenCount(man); i++)
  {
    Item item = treeModel.getChildAt(man, i);
    Object obj=LwToolkit.getStaticObj((String)item.getValue());
    matrixModel.put (row, 1, obj.getClass().getName());
    row++;
  }
  Item sobj = treeModel.getChildAt(treeModel.getRoot(), 1);
  row++;
  for (int i=0;i<treeModel.getChildrenCount(sobj); i++)
  {
    Item item = treeModel.getChildAt(sobj, i);
    Object obj = LwToolkit.getStaticObj ((String)item.getValue());
    matrixModel.put (i + row, 1, obj.getClass().getName());
  }
 }
```

Creates the second column with the class name.

Fills the second tree grid column with the corresponding class names.
Gets grid model.
Gets current tree model.

Defines starting row.
Gets manager item from the tree model.
Goes through the manager' child items to get corresponding object instance, determine the instance class and set the class name to the grid model (for the second column and the row that corresponds to the tree model item).

Gets the static objects item from the tree model.
Goes through the static object's child items to get appropriate object instance, determine the instance class and set the class name to the grid model (for the second column and the row that corresponds to the tree model item).

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

```
}
```

The result is shown below:

Object Key	Class Name
⊟ root	
⊟ Managers	
paint	org.zaval.lw.LwPaintManImpl
focus	org.zaval.lw.LwFocusManager
cursor	org.zaval.lw.LwCursorManager
clip	org.zaval.lw.LwClipboardMan
pop	org.zaval.lw.LwPopupManager
tooltip	org.zaval.lw.LwTooltipMan
⊟ Static Objects	
ch.img	java.lang.String
tree.img	java.lang.String

At the next step we will add third column to the tree grid that will show a view bound with the static object if it is possible. To render the third column cells it is necessary to implement special view provider for the tree grid component (since the default tree grid render cannot render views) class as follows (red color indicates additional changes that had been made on this step):

```
public class LwPropertiesViewer
extends LwPanel
{
  public LwPropertiesViewer(Hashtable p)
  {
    TreeModel  treeModel = createTreeModel(p);
    LwTreeGrid treeGrid  = createTreeGrid (treeModel);
    createTreeGridModel(treeGrid);
    for (int i=0; i < treeGrid.getCols(); i++)
      treeGrid.setColWidth(i, 180);
    treeGrid.setViewProvider(new LwCellViewProvider());
    add (LwBorderLayout.CENTER,
           new    LwScrollPan(treeGrid));
  }
  …
  protected void createTreeGridModel(LwTreeGrid grid)
  {
     …
    Item sobj = treeModel.getChildAt(treeModel.getRoot(), 1);
    row++;
    for (int i=0;i<treeModel.getChildrenCount(sobj); i++)
    {
      Item item = treeModel.getChildAt(sobj, i);
      Object obj = LwToolkit.getStaticObj((String)item.getValue());
      matrixModel.put (i + row, 1, obj.getClass().getName());
      matrixModel.put (i + row, 2, obj);
    }
  }
}
```

Sets the custom view provider.

Adds instance of the static object to the third column.

The custom tree grid view provider can render views instances if it is necessary. The provider code is shown below:

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

```
public class LwCellViewProvider
extends LwDefViews
{
  LwCanvas        target  = new LwCanvas();
  LwCompRender render = new LwCompRender(target);

  public LwView getView(int row, int col, Object obj)
  {
    if (col == 2)
    {
      if (obj instanceof LwView)
      {
        LwView view = (LwView)obj;
        if (view.getType() != LwView.ORIGINAL)
        {
          target.getViewMan(true).setView(view);
          Dimension d = view.getPreferredSize();
          target.setPSSize(d.width, d.height);
          return render;
        }
        else return view;
      }
      else return super.getView(row, col, "---");
    }
    else
    if (col == 1) return super.getView(row, col, obj);
    return null;
  }
}
```

The view provider extends standard view provider.

Defines target object and his render.

Tests if the column number is 2 and the object instance is **LwView.**

Checks if the view type is not ORIGINAL. In this case the view is set for the target canvas. It is necessary to render the view using its preferred size.

Returns view as is.

Returns “---” string if the object cannot be rendered (if the object doesn’t represent any view)

If the column number is 1 than the view works as the **LwDefViews** view provider.

The final application is shown below:

See the “**org/zaval/lw/samples/LwPropertiesViewer.java.java**”

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.

Object Key	Class Name	View
layout.raster	org.zaval.lw.LwRasterLayout	---
check.off	org.zaval.lw.LwImgSetRender	
check.on	org.zaval.lw.LwImgSetRender	
check.dison	org.zaval.lw.LwImgSetRender	
check.disoff	org.zaval.lw.LwImgSetRender	
radio.off	org.zaval.lw.LwImgSetRender	
radio.on	org.zaval.lw.LwImgSetRender	
radio.dison	org.zaval.lw.LwImgSetRender	
radio.disoff	org.zaval.lw.LwImgSetRender	
scroll.hbg	org.zaval.lw.LwImgSetRender	
scroll.vbg	org.zaval.lw.LwImgSetRender	
bt.left.out	org.zaval.lw.LwImgSetRender	
bt.left.pressed	org.zaval.lw.LwImgSetRender	
bt.left.over	org.zaval.lw.LwImgSetRender	
bt.left.disabled	org.zaval.lw.LwImgSetRender	
bt.right.out	org.zaval.lw.LwImgSetRender	
bt.right.pressed	org.zaval.lw.LwImgSetRender	
bt.right.over	org.zaval.lw.LwImgSetRender	
bt.right.disabled	org.zaval.lw.LwImgSetRender	
bt.bottom.out	org.zaval.lw.LwImgSetRender	
bt.bottom.pressed	org.zaval.lw.LwImgSetRender	
bt.bottom.over	org.zaval.lw.LwImgSetRender	

© Zaval Creative Engineering Group, 2001-2005. All rights reserved.