Lightweight UI Toolkit

Developer's Guide

Release 1.5

E23376-03

April 2012

Describes how to use the Lightweight UI Toolkit (LWUIT) library to create appealing graphical user interface applications for mobile phones and other devices that support MIDP 2.0.

Lightweight UI Toolkit, Release 1.5

E23376-03

Contents

Preface

This document describes how to work with the Lightweight User Interface toolkit.

Before You Read This Document

This guide is intended for developers creating Mobile Information Device Profile (MIDP) applications. This book is a tutorial in Lightweight UI Toolkit programming over MIDP. You should already have basic knowledge about Java™ UI libraries (for example, AWT and SWING) and understand how to use the Mobile Information Device Profile (MIDP) and the Connected Limited Device Configuration (CLDC).

For current discussion of LWUIT issues, see these online resources:

- LWUIT home page: `http://lwuit.java.net/`
- LWUIT community discussion forum: `http://www.java.net/forums/mobile-embedded/lwuit`
- LWUIT Blog: `http://codename-1.blogspot.com/`

If you need help getting started with the Java programming language, try the New to Java Center:

http://www.oracle.com/technetwork/topics/newtojava/overview/index.html

For a quick start with MIDP programming, read Learning Path: Getting Started with MIDP 2.0:

`http://developers.sun.com/mobility/learn/midp/midp20/`

The following sites provide technical documentation related to Java ME technology:

http://download.oracle.com/javame/

How This Document Is Organized

This guide contains the following chapters and appendices:

Chapter 1 introduces the Lightweight UI Toolkit library.

Chapter 2 describes how to use Lightweight UI Toolkit widgets.

Chapter 3 explains how to use Lists.

Chapter 4 descries the Table and Tree components.

Chapter 5 describes how to use Dialogs.

Chapter 6 shows how you can use Layouts.

Chapter 7 explains how to use the Style object.

Chapter 8 describes theme elements.

Chapter 9 describes the Resource Editor utility.

Chapter 10 explains how to use Painters.

Chapter 11 describes the LWUIT implementation,

Chapter 12 describes the HTMLComponent class.

Chapter 13 describes how to use Transitions and Animations.

Chapter 14 describes how to author a new component from scratch.

Chapter 15 discusses general and device-specific portability issues.

Appendix A summarizes frequently asked questions about LWUIT.

Shell Prompts

Shell	Prompt
C shell	*machine-name*`%`
C shell superuser	*machine-name*`#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell superuser	`#`

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	**`% su`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this. To delete a file, type `rm` *filename*.

Note: Characters display differently depending on browser settings. If characters do not display correctly, change the character encoding in your browser to Unicode UTF-8.

Related Documentation

The following table lists documentation related to this product.

Topic	Title and URL
JSR 118, MIDP 2.0	*Mobile Information Device Profile* `http://jcp.org/en/jsr/detail?id=118`
JSR 139, CLDC 1.1	*Connected Limited Device Configuration* `http://jcp.org/en/jsr/detail?id=139`
JSR 184, 3D Graphics	*Mobile 3D Graphics API for J2ME* `http://jcp.org/en/jsr/detail?id=184`
AWT docs	`http://download.oracle.com/javase/6/docs/technotes/guides/awt/`

Topic	Title and URL
Swing docs	http://java.sun.com/javase/6/docs/technotes/guides/swing/index.html

We Welcome Your Comments

We are interested in improving our documentation and welcomes your comments and suggestions. Email your feedback to:

LWUIT_COMMUNITY_WW@ORACLE.COM

Introducing the Lightweight UI Toolkit Library

This book describes how to use the Lightweight UI Toolkit (LWUIT) library. The Lightweight UI Toolkit library helps you create appealing graphical user interface (GUI) applications for mobile phones and other devices that support MIDP 2.0. Lightweight UI Toolkit supports visual components and other user interface (UI) ingredients such as theming, transitions, animation and more.

After covering the basics of the Lightweight UI Toolkit, this book provides a walk through of the various widgets and uses of the LWUIT packages.

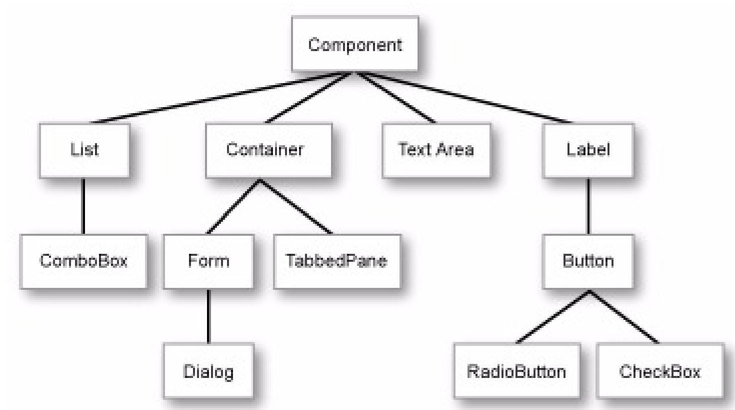
1.1 API Overview

The Lightweight UI Toolkit is a lightweight widget library inspired by Swing but designed for constrained devices such as mobile phones and set-top boxes. Lightweight UI Toolkit supports pluggable theme-ability, a component and container hierarchy, and abstraction of the underlying GUI toolkit. The term lightweight indicates that the widgets in the library draw their state in Java source without native peer rendering.

Internal interfaces and abstract classes provide abstraction of interfaces and APIs in the underlying profile. This allows portability and a migration path for both current and future devices and profiles. For example, Graphics would be an abstraction of the graphics object in the underlying profile.

The Lightweight UI Toolkit library tries to avoid the "lowest common denominator" mentality by implementing some features missing in the low-end platforms and taking better advantage of high-end platforms. Figure 1–1 shows the widget class hierarchy.

Figure 1–1 Simplified Widget Class Hierarchy



1.1.1 Scope and Portability

The Lightweight UI Toolkit library is strictly a widget UI library and does not try to abstract the underlying system services such as networking or storage. It also doesn't try to solve other UI issues related to native graphics, etcetera.

To enable portability, the Lightweight UI Toolkit library implements its own thin layer on top of the native system canvas and provides a widget abstraction. This abstraction is achieved using several key classes that hide the system specific equivalents to said classes, such as Graphics, Image and Font.

When working with the Lightweight UI Toolkit library it is critical to use the abstract classes for everything. To avoid corruption, there is no way to access the "real" underlying instances of these classes (for example, `javax.microedition.lwuit.Graphics`).

LWUIT strives to enable great functionality on small devices that might be incapable of anti-aliasing at runtime, or might choke under the weight of many images. To solve these problems the LWUIT library ships with an optional resource file format that improves resource utilization. For more details, see Chapter 9.

Example 1–1 Hello World for MIDP

This is a simple hello world example written on top of MIDP. All UI code making use of the Lightweight UI Toolkit is compatible to other platforms such as CDC.[1]

However, this example is specifically for MIDP. For MIDP the application management system (AMS) requires a MIDlet class to exist, where in a CDC environment an Xlet would be expected (and in Java SE you would expect a main class, and so forth).

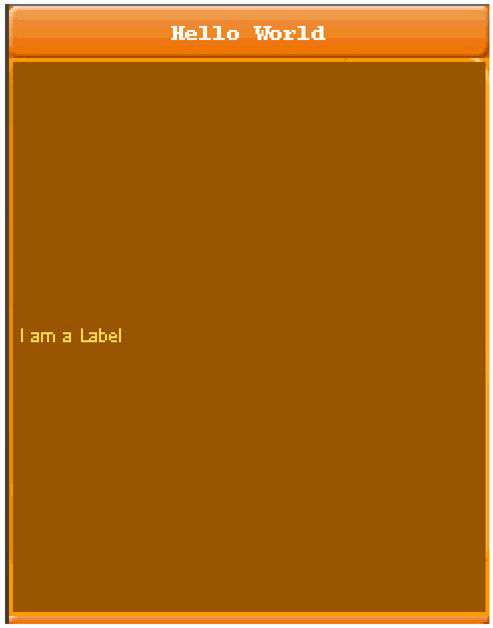
```
import com.sun.lwuit.Display;
import com.sun.lwuit.Form;
import com.sun.lwuit.Label;
import com.sun.lwuit.layouts.BorderLayout;
import com.sun.lwuit.plaf.UIManager;
import com.sun.lwuit.util.Resources;

public class HelloMidlet extends javax.microedition.midlet.MIDlet {

   public void startApp() {
       //init the LWUIT Display
       Display.init(this);

       // Setting the application theme is discussed
       // later in the theme chapter and the resources chapter
       try {
             Resources r = Resources.open("/myresources.res");
             UIManager.getInstance().setThemeProps(
               r.getTheme(r.getThemeResourceNames()[0])
              );
            } catch (java.io.IOException e) {
   }
       Form f = new Form();
       f.setTitle("Hello World");
       f.setLayout(new BorderLayout());
       f.addComponent(BorderLayout.CENTER, new Label("I am a Label"));
       f.show();
   }

   public void pauseApp() {
   }
```

[1] As of this writing the CDC version of LWUIT required for this compatibility hasn't been released to the public.

```
    public void destroyApp(boolean unconditional) {
    }
}
```

Hello world looks like Figure 1–2.

Figure 1–2 Hello World



Notice in Example 1–1 that the very first line of code for any application using the Lightweight UI Toolkit library must register the main class with the display. This behavior is tool-specific. In MIDP there is not much you can do without a reference to the parent MIDlet, so this operation must be performed in the beginning of the application.

The creation of the UI code is left within the MIDlet for simplicity but it could be separated to any class to allow full portability in any future platform to which the Lightweight UI Toolkit library would be ported.

1.1.2 Events and Threading

For increased compatibility, the Lightweight UI Toolkit library completely handles and encapsulates UI threading. It has a single main thread referred to as the "EDT" (inspired by the Event Dispatch Thread in Swing and AWT). All events and paint calls are dispatched using this thread. This guarantees that event and paint calls are serialized and do not risk causing a threading issue. It also enables portability for profiles that might have minor threading model inconsistencies. See the Display class (`com.sun.lwuit.Display` in the API documentation) for further details about integrating with the EDT and serializing calls on it.

Using Lightweight UI Toolkit Widgets

This chapter introduces the LWUIT widgets and provides sample code for several components.

2.1 Component

A *Component* is an object having a graphical representation that can be displayed on the screen and can interact with the user. The buttons, check boxes, and radio buttons in a typical graphical UI are all examples of a component. Component is the base class. All the widgets in the Lightweight UI Toolkit library use the composite pattern in a manner similar to the AWT Container and Component relationship.

2.2 Container

A *Container* is a composite pattern with a Component object. It enables nesting and arranging multiple components using a pluggable layout manager architecture. Containers can be nested one within the other to form elaborate UIs. Components added to a container are tracked in a list. The order of the list defines the components' front-to-back stacking order within the container. If you do not specify an index when you add a component to a container, it is added to the end of the list (and hence to the bottom of the stacking order).

2.3 Form

Form is a top-level component that serves as the root for the UI library. This Container handles the title and menus and allows content to be placed between them. By default the form's central content (the content pane) is scrollable. Form contains Title bar, MenuBar and a ContentPane. Invocations of Form's `addComponent` method are delegated to the content pane's `addComponent`. The same applies to most composite related methods (e.g. `setLayout`, `getComponent` and so forth).

The following code demonstrates creation and setup of a form.

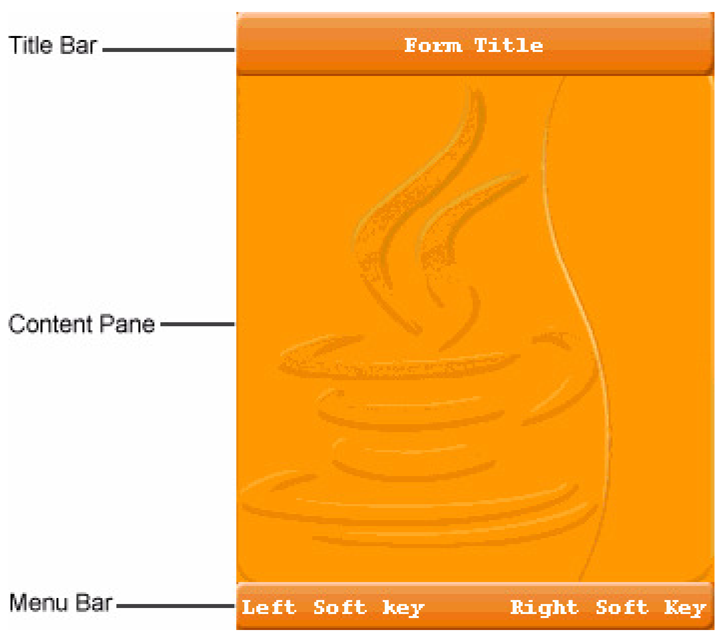
Example 2–1   Form Setup and Creation;

```
// 1. Create a Form
Form mainForm = new Form("Form Title");
// 2. Set LayoutManager
mainForm.setLayout(new BorderLayout());
// 3. Add a Label to the center of Form content pane
mainForm.addComponent(BorderLayout.CENTER, new Label(“Hello World”));
// 4. Set Transitions animation of Fade
mainForm.setTransitionOutAnimator(CommonTransitions.createFade(400));
// 5. Add Command key
mainForm.addCommand(new Command("Run", 2));
// 6. Show it
mainForm.show();
```

The following notes correspond to the comments in Example 2–1.

1. The first line of code creates a form using a constructor that lets you set the form title. The other frequently used form constructor is the no-argument constructor.
2. Next the code specifies the layout manager of the form. Layout managers are discussed later in this guide.
3. The next bit of code adds a label to the form content pane. Adding components to a Form (which is a Container) is done with `addComponent(Component cmp)` or `addComponent(Object constraints, Component cmp)`, where `constraints` are the locations in the layout manager, BorderLayout.
4. A Transition is the movement effect action that occurs when switching between forms. See the Transitions and Animation chapter.
5. Form has menus to emulate the device soft keys, for example. To set such a menu bar item, command, use the `addCommand(Command cmd)` method. The Commands are placed in the order they are added. If the Form has one Command it is placed on the right. If the Form has two Commands the first one added is placed on the left and the second one is placed on the right. If the Form has more than two Commands the first one stays on the left and a Menu is added with all the remaining Commands.
6. The show method displays the current form on the screen.

Figure 2–1 Form Element



2.4 Create and Set Up a Form Label

The Label widget can display a single line of text and/or an image and align them using multiple options. If you need to create a component that displays a string, an image, or both, you should use or extend Label. If the component is interactive and has a specific state, a Button is the most suitable widget (instead of a label).

To create a Label, use one of the following calls:

```
Label textLabel = new Label("I am a Label"); // for a text label
```

Create an image for an icon label:

```
Image icon = Image.createImage("/images/duke.png");
Label imageLabel = new Label(icon);
```

Labels can be aligned to one of the following directions: CENTER, LEFT, RIGHT. LEFT is the default. In addition the text can be aligned relative to the image position. Valid values are TOP, BOTTOM, LEFT, RIGHT, where the default is RIGHT. To update the text position use:

```
setTextPosition(int alignment);
```

Figure 2–2 displays three types of labels with text to icon alignment position of RIGHT. The container is divided into three rows, and the label in each row is as wide as possible. Figure 2–3 shows relative alignment, with the label below the icon.

Figure 2–2 Label With Text, Label With Icon, and Label with Text and Icon

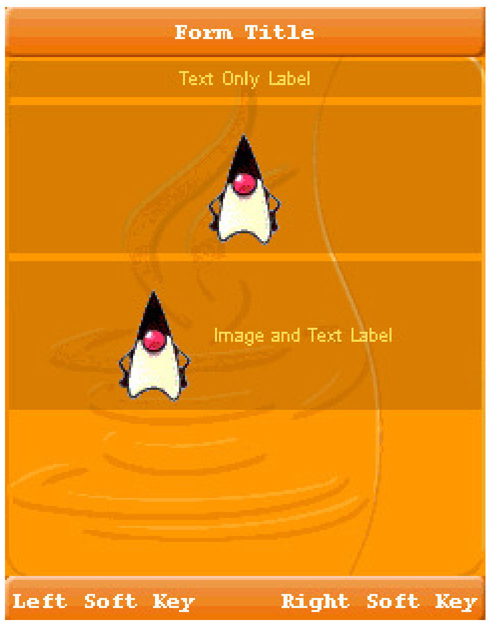
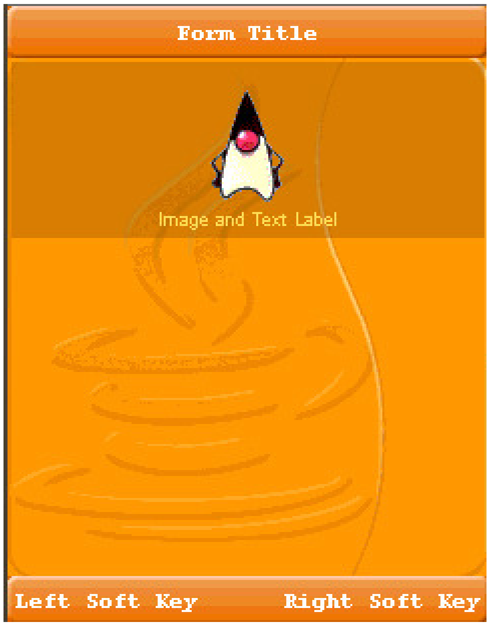


Figure 2–3 Text to Icon Alignment Position of BOTTOM



2.5 Button

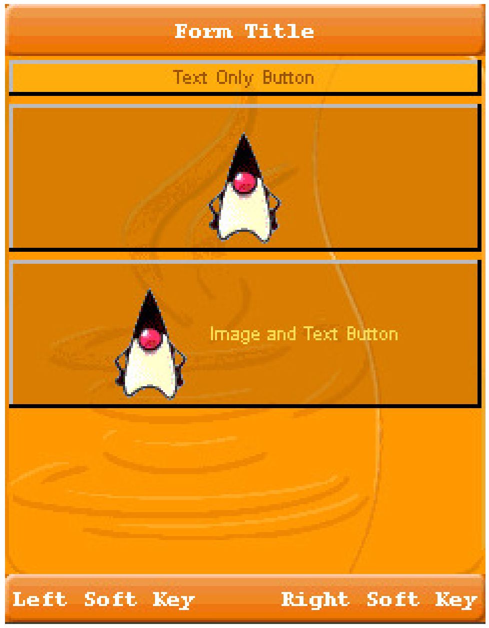
The Button component enables the GUI developer to receive action events when the user focuses on the component and clicks. In some devices a button might be more practical and usable than a command option. Button is the base class for several UI widgets that accept click actions. It has three states: rollover, pressed, and the default state. It can also have ActionListeners that react when the Button is clicked.

To get the user clicking event, you must implement an ActionListener, which is notified each time the user clicks the button. The following code snippet creates an action listener and changes the text on the button, every time the user clicks it.

```
final Button button  = new Button("Old Text");
button.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent evt) {
         button.setText("New Text");
      }
   });
```

Button extends Label, so you can create three type of buttons: text only, image only or image and text button.

Figure 2–4 Button With Text, Button With Icon, and Button With Text and Icon



2.6 RadioButton

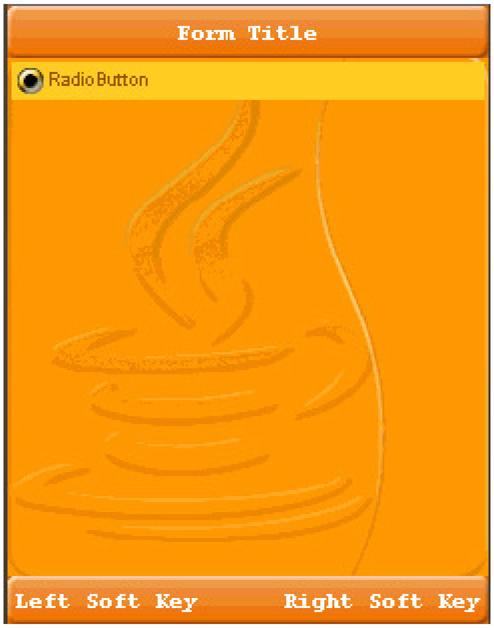
RadioButton is a Button that maintains a selection state exclusively within a specific ButtonGroup. Because RadioButton inherits from Button, radio buttons have all the usual button characteristics, as discussed in Section 2.5, "Button". For example, you can specify the image displayed in a radio button. Each time the user clicks a radio button (even if it was already selected), the button fires an action event, just as in Button.

To create a RadioButton use:

```
RadioButton radioButton = new RadioButton(“Radio Button”);
```

Figure 2–5 shows the RadioButton this code produces.

Figure 2–5 Sample Radio Button



2.7 ButtonGroup

The ButtonGroup component manages the selected and unselected states for a set of RadioButtons. For the group, the ButtonGroup instance guarantees that only one button can be selected at a time.

Initially, all RadioButtons in a ButtonGroup are unselected. Each ButtonGroup maintains the selected index, and can get a specific RadioButton by calling `getRadioButton(int index)`.

The following code snippet creates a button group made of two RadioButtons.

```
Label radioButtonsLabel = new Label("RadioButton:");
....
RadioButton rb1 = new RadioButton("First RadioButton in Group 1");
RadioButton rb2 = new RadioButton("Second RadioButton in Group 1");

   ButtonGroup group1 = new ButtonGroup();
   group1.add(rb1);
   group1.add(rb2);

exampleContainer.addComponent(radioButtonsLabel);
exampleContainer.addComponent(rb1);
exampleContainer.addComponent(rb2);
```

The code snippet result is shown in Figure 2–6.

Figure 2–6 RadioButton Group

2.8 CheckBox

Check boxes are similar to RadioButtons but their selection model is different, because they can flip the selection state between selected and unselected modes. A group of radio buttons, on the other hand, can have only one button selected. Because CheckBox inherits from Button, check boxes have all the usual button characteristics, as discussed in Section 2.5, "Button". For example, you can specify the image displayed in a check box. Each time the user select a check box (even if it was already selected), it fires an action event, just as in Button.

To create a CheckBox use:

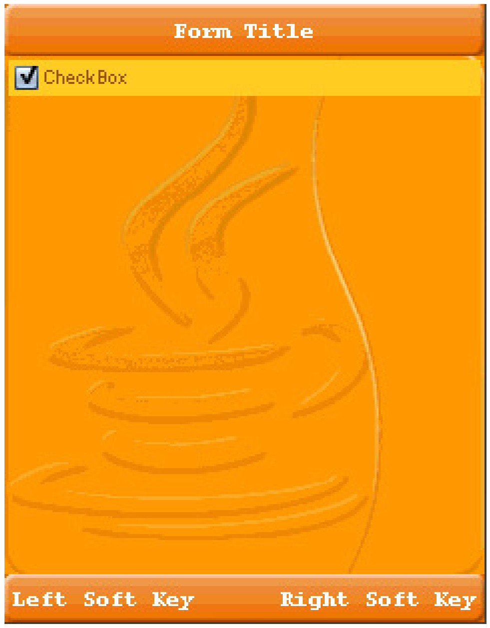
```
final CheckBox checkBox = new CheckBox(“Check Box”);
```

This code produces the CheckBox shown in Figure 2–7.

To catch select and unselect events you can try this:

```
checkBox.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent evt) {
      if(checkBox.isSelected()) {
         System.out.println("CheckBox got selected");
      } else {
         System.out.println("CheckBox got unselected");
      }
   }
});
```

Figure 2–7 CheckBox Sample



2.9 ComboBox

A combo box is a list that allows only one selection at a time. When a user clicks the combo box button, a drop-down list of elements allows the user to select a single element. The combo box is driven by the list model and allows all the renderer features of the List as well.

Other components that can display one-of-many choices are groups of radio buttons, check boxes, buttons, and lists. Groups of radio buttons are generally the easiest for users to understand, but combo boxes can be more appropriate when space is limited or more than a few choices are available. Lists are not always attractive, but they are more appropriate than combo boxes when the number of items is large (say, over five).

The following code creates a combo box (a list model that is built from check boxes) and sets it up:

```
String[] content = { "Red", "Blue", "Green", "Yellow" };

// 1. Creating the combo box
ComboBox comboBox = new ComboBox(content);

// 2. Setting a checkBox renderer
comboBox.setListCellRenderer(new checkBoxRenderer());

// 3. Adding a action listener to catch user clicking
//    to open the ComboBox

comboBox.addActionListener(myActionListener......);
```

The following notes correspond to the comments in the code above.

1. This combo box code contains an array of strings, but you could just as easily use labels instead.

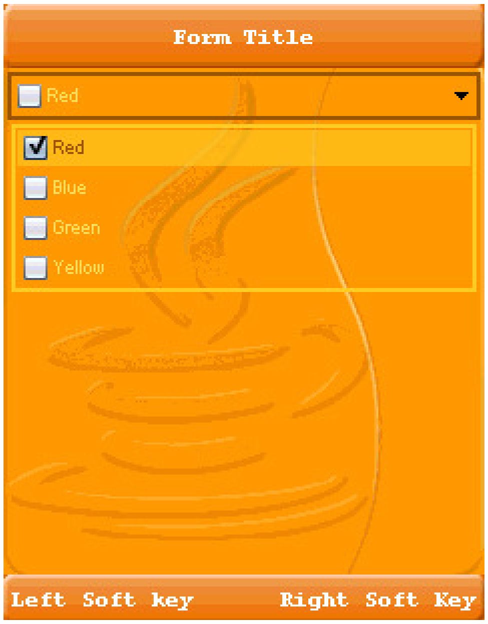
2. To put anything else into a combo box or to customize how the items in a combo box look, you need to write a custom renderer.
3. The next line of code (which calls setListCellRender) registers an action listener on the combo box.

The following is a sample of renderer code:

```
/**
 * Demonstrates implementation of a renderer derived from a CheckBox
 */
private static class checkBoxRenderer extends CheckBox implements ListCellRenderer
{

      /** Creates a new instance of checkBoxRenderer */
      public checkBoxRenderer() {
         super("");
      }

      // Setting the current check box text and status
      public Component getListCellRendererComponent(List list,
                       Object value, int index, boolean isSelected) {
         setText("" + value);
         if (isSelected) {
            setFocus(true);
            setSelected(true);
         } else {
            setFocus(false);
            setSelected(false);
         }
         return this;
      }

      // Returning the list focus component
      public Component getListFocusComponent(List list) {
            setText("");
            setFocus(true);
            setSelected(true);
            return this;
      }
   }
```

The sample code produces the combo box in Figure 2–8.

Figure 2–8 Combo Box



2.10 Tabs

Tabs are containers that let the user switch between a group of components that all share the same space by focusing on a tab with a title, an icon, or both. The user chooses which component to view by selecting the tab corresponding to the desired component.

To create a tab pane, instantiate Tabs, create the components you wish it to display, and then add the components to the tabbed pane using the `addTab` or `insertTab` methods. Tabs has the ability to remove tabs as well, by calling `removeTabAt(int index)` at a given position index. A tab is represented by an index corresponding to the position it was added in, where the first tab has an index equal to 0 and the last tab has an index equal to the tab count minus 1.

If the tab count is greater than 0, then there is always a selected index, which by default is initialized to the first tab. If the tab count is 0, then the selected index is -1.

Tabs has four different tab placement orientations. The default tab placement is set to the TOP location. You can change the tab placement to LEFT, RIGHT, TOP or BOTTOM using the `setTabPlacement` method.

The following code creates a pane with tab placement of bottom, and places a Label in the center of the first (and only) tab.

```
Tabs tabs = new Tabs(Tabs.TOP);
tabs.addTab("Tab 1", new Label("I am a Tab!"));
tabs.addTab("Tab 2", new Label("Tab number 2"));
...
```

2.11 TextArea

The text area represents text that might be editable using the system native editor (it might occur in a new screen). The native editor is used to enable complex input methods (such as T9) and application internationalization. The following code creates and initializes the text area:

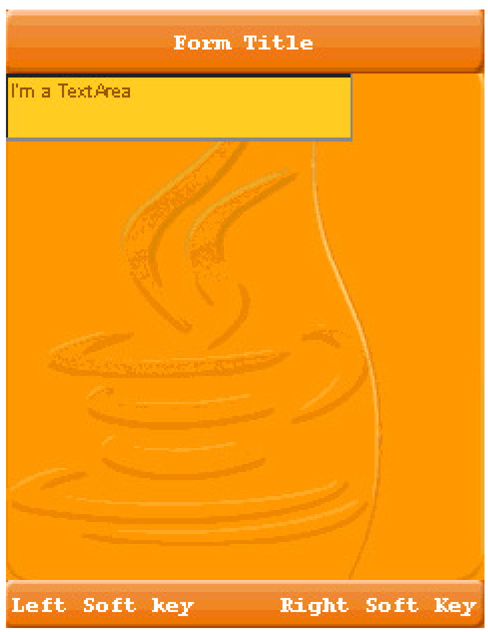
```
TextArea textArea = new TextArea(5, 20, TextArea.NUMERIC);
textArea.setEditable(false);
```

The first two arguments to the TextArea constructor are hints as to the number of rows and columns, respectively, that the text area should display. The third one is a constraint that is passed into the native text editor. Valid values can be one of `ANY`, `EMAILADDR`, `NUMERIC`, `PHONENUMBER`, `URL`, or `DECIMAL`. In addition it can be bitwise OR'd with one of `PASSWORD`, `UNEDITABLE`, `SENSITIVE`, `NON_PREDICTIVE`, `INITIAL_CAPS_SENTENCE`, `INITIAL_CAPS_WORD`. For example, `ANY | PASSWORD`. The default value is `ANY`. In the above example `NUMERIC` only allows the user to type numbers.

Text areas are editable by default. The code `setEditable(false)` makes the text area uneditable. It is still selectable, but the user cannot change the text area's contents directly.

A 5 x 20 text area is shown in Figure 2–9.

Figure 2–9 Form With Text Area



2.12 TextField

TextArea doesn't always allow in-place editing on existing devices and doesn't provide "fine grained control" over the input. This allows a text area to be lightweight, and portable for all possible devices. These restrictions sometimes cause a poor user experience because it requires users to go into a different screen for input (since all input is handled natively by the device). From a developer standpoint the native input can be a problem since it doesn't send change events and doesn't provide control over allowed input.

LWUIT provides the TextField component to support direct mobile phone input from within LWUIT. Unlike a TextArea, TextField is completely implemented in LWUIT.

Developers can override almost all of its features to provide deep customization (for example, masked input, localization, and more).

TextField inherits the TextArea component and all of its features. It also supports moving to the native text editor.

The constructor also accepts several arguments, similar to the TextArea component.

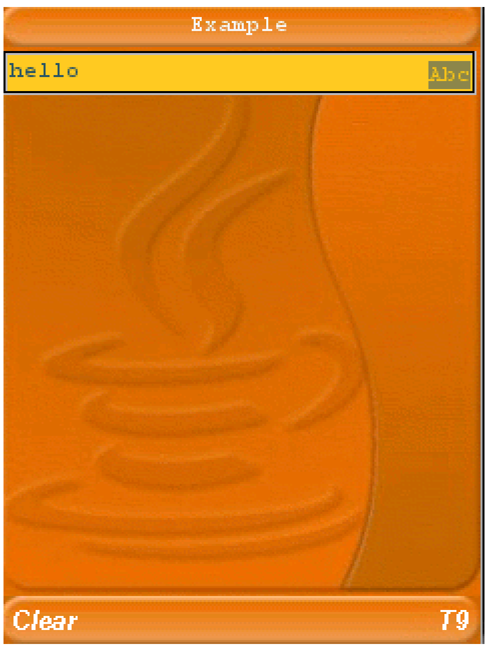
TextField also has some limitations:

- Does not support input in foreign locales unless you provide code for foreign input
- Does not support device features, such as T9 input
- Might not correctly detect QWERTY devices
- Does not work on devices with unique keyboards, such as the Perl

Creating a text field is trivial:

```
TextField f = new TextField();
```

Figure 2–10 Sample Text Field



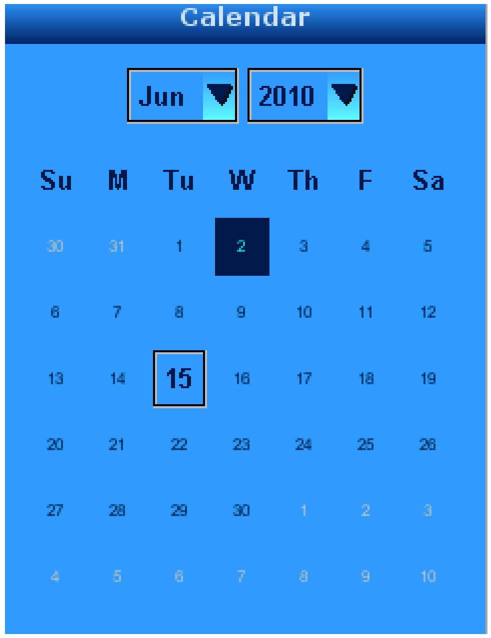
2.13 Calendar

The LWUIT calendar component allows users to pick a date using a monthly calendar user interface. Use the calendar component to navigate and pick a date, as shown in the following code:

```
Calendar cal = new Calendar();
```

Developers can monitor state changes within the calendar using a data change listener or an action listener.

Figure 2–11 Calendar Component



2.14 Tickering

Label (and all its subclasses) includes ticker support. A ticker scrolls the content of a long label across the screen. Ticker ability in labels is usually indicated by displaying three dots "..." after the end of the label. When the label (button, checkbox, etcetera) receives focus, these three dots disappear and the label starts animating like a stock ticker.

A ticker can be started explicitly using a call to `startTicker` or `stopTicker` in Label. It can also be prevented by invoking `setTickerEnabled(false)`. To prevent the three dots from appearing at the end of labels or components that support tickering, use `setEndsWith3Points(false)`.

2.15 Bidi

BiDi refers to bidirectional language support, generally used for right-to-left (RTL) languages. There is plenty of information about RTL languages (Arabic, Hebrew, Syriac, Thaana) on the internet, but as a brief primer here is a minor summary.

Most western languages are written from left to right (LTR), however some languages are normally written from right to left (RTL). Speakers of these languages expect the UI to flow in the opposite direction, otherwise it seems "weird" just like reading this word in RTL would look: "driew" to most English speakers.

The problem posed by RTL languages is known as bi-directional) and not as RTL since the "true" problem isn't the reversal of the writing/UI but rather the mixing of RTL and LTR together. For example, numbers are always written from left to right (just like in English) so in an RTL language the direction is from right to left and once we reach a number or English text embedded in the middle of the sentence (such as a name) the direction switches for a duration and is later restored.

LWUIT supports BiDi with the following components:

- BiDi algorithm - allows converting between logical to visual representation for rendering
- Global RTL flag- default flag for the entire application indicating the UI should flow from right to left
- Individual RTL flag - flag indicating that the specific component/container should be presented as an RTL/LTR component (for example, for displaying English elements within an RTL UI).
- RTL text field input
- RTL text field input
- RTL bitmap font rendering

Most of LWUIT's RTL support is under the hood. The LookAndFeel global RTL flag can be enabled using:

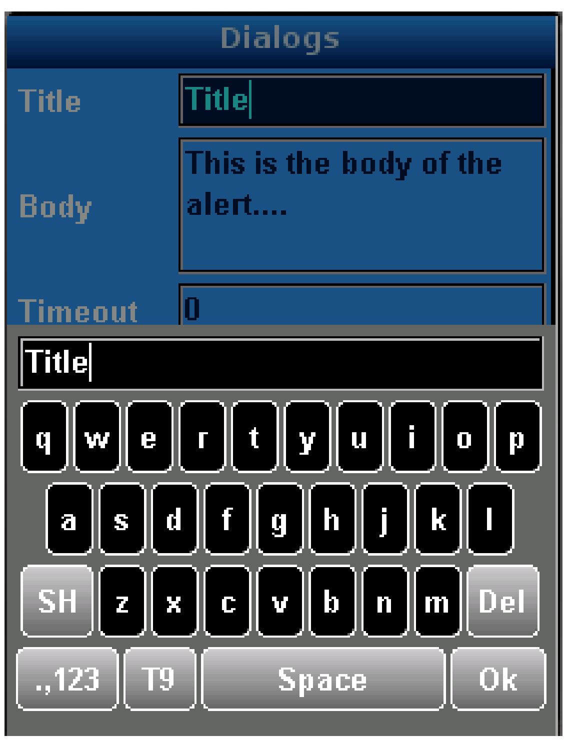
```
UIManager.getInstance().getLookAndFeel().setRTL(true)
```

(Notice that setting the RTL to true implicitly activates the BiDi algorithm).Once RTL is activated all positions in LWUIT become reversed and the UI becomes a mirror of itself. For example, a softkey placed on the left moves to the right, padding on the left becomes padding on the right, the scroll moves to the left, etcetera.

This applies to the layout managers (except for group layout) and most components. BiDi is mostly seamless in LWUIT but a developer still needs to be aware that his UI might be mirrored for these cases.

2.16 Virtual Keyboard

LWUIT supports a lightweight Virtual Keyboard which is implemented with LWUIT’s own components. The LWUIT virtual keyboard is used seamlessly where access to the native virtual keyboard isn't possible (for example, MIDP).



2.17 Customizing the Virtual Keyboard

Since the Virtual Keyboard is a pure LWUIT component it can be customized in various ways.

2.17.1 Changing the Virtual Keyboard Look and Feel

All Virtual Keyboard items can be customized from the resource editor. The associated UI IDs are as follows:

Table 2–1 Virtual Keyboard User Interface IDs

`VKB`	Customizes the Virtual Keyboard body.
`VKBtooltip`	Customizes the popup tooltip.
`VKBButton`	Customizes a regular button on the virtual keyboard (usually a char or a string).
`VKBSpecialButton`	Customizes the special buttons such as: 'Space', 'SH', etcetera.
`VKBTextInput`	Customizes the textfield on the virtual keyboard.

2.17.2 Adding a Language

The following example demonstrates how to add an input mode that supports Hebrew.

2.17.3 Adding an Input Mode for Hebrew

1. Create an array of String arrays in which each array represents a button's column.

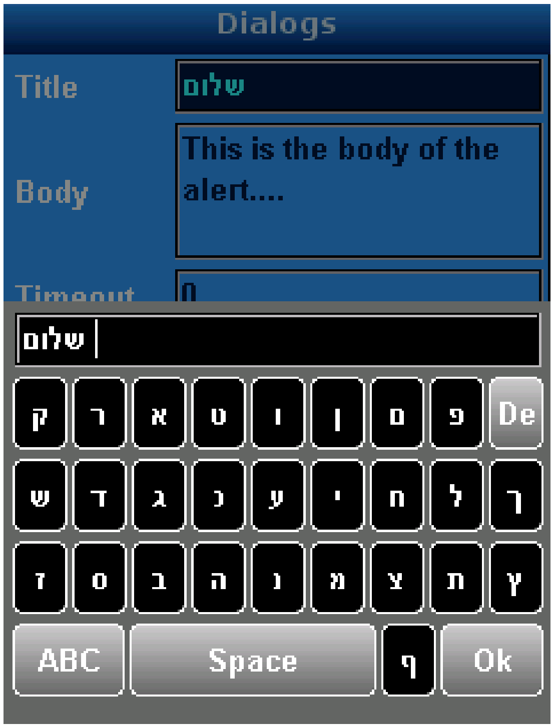
```
private static final String[][] DEFAULT_HEBREW = new String[][]{
   {"\u05e7", "\u05e8", "\u05d0", "\u05d8", "\u05d5", "\u05df", "\u05dd",
    "\u05e4", "$Delete$"},
   {"\u05e9", "\u05d3", "\u05d2", "\u05db", "\u05e2", "\u05d9", "\u05d7",
    "\u05dc", "\u05da"},
   {"\u05d6", "\u05e1", "\u05d1", "\u05d4", "\u05e0", "\u05de", "\u05e6",
    "\u05ea", "\u05e5"},
   {"$Mode$", "$Space$", "\u05E3", "$OK$"}
 };
```

2. Now extend the VirtualKeyboard and make sure the new language mode is added when the VirtualKeyboard is initialized.

```
public static class HebrewK extends VirtualKeyboard {
   public HebrewK() {
      addInputMode("05d005d105d2", DEFAULT_HEBREW);
      setInputModeOrder(new String[]{"05d005d105d2", QWERTY_MODE,
            NUMBERS_SYMBOLS_MODE, NUMBERS_MODE, SYMBOLS_MODE
         });
   }
}
```

3. When calling the virtual keyboard, specify the HebrewK class to make it the default:

```
VKBImplementationFactory.init(HebrewK.class);
```



2.17.4 Binding a Virtual Keyboard to a TextField

In some cases a TexField should accept only numbers, therefore launching the regular VirtualKeyboard is a mistake. What we need to do is to create a “numbers only” VirtualKeyboard and launch it for a specific TextField.

```
TextField txt = new TextField();
txt.setConstraint(TextField.NUMERIC);
txt.setInputModeOrder(new String[]{"123"});
txt.setInputMode("123");

VirtualKeyboard vkb = new VirtualKeyboard();
vkb.setInputModeOrder(new String[]{VirtualKeyboard.NUMBERS_MODE});

VirtualKeyboard.bindVirtualKeyboard(txt, vkb);
```

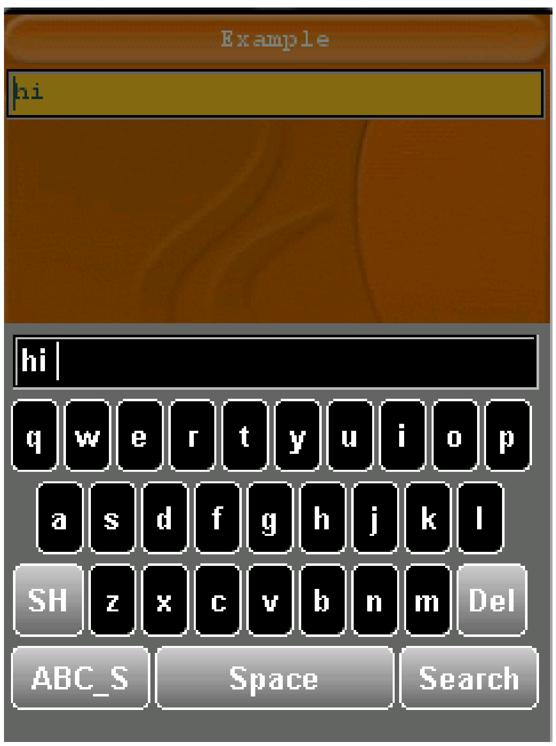
2.17.5 Adding Your Own Button to a TextField

There are several use cases where you would want to place your own buttons on a specific Virtual Keyboard. For example if you are asking the user to insert input for a search field you might want a "search" command instead of the regular "ok" command that when pressed automatically invokes a submit action to the network.To accomplish that you must create a new virtual keyboard, declare your own input buttons, and to add your own special button to be part of the virtual keyboard. (By default the Virtual Keyboard understands only the following special keys: "Shift", "Delete", "T9", "Mode", "Space", and "OK").

The following example declares a new input with a new special button "Search".

```
String[][] SEARCH_QWERTY = new String[][]{
   {"q", "w", "e", "r", "t", "y", "u", "i", "o", "p"},
```

```
   {"a", "s", "d", "f", "g", "h", "j", "k", "l"},
   {"$Shift$", "z", "x", "c", "v", "b", "n", "m", "$Delete$"},
   {"$Mode$", "$Space$", "$Search$"}
};

VirtualKeyboard vkb = new VirtualKeyboard();
//add the new input mode
   vkb.addInputMode("ABC_S", SEARCH_QWERTY);
   vkb.setInputModeOrder(new String[]{"ABC_S"});
//add the new special button to the vkb
   vkb.addSpecialButton("Search", new Command("Search") {

   public void actionPerformed(ActionEvent evt) {
      //search logic...
   }
});

//bind the vkb to the textfield
VirtualKeyboard.bindVirtualKeyboard(txt, vkb);
f.addComponent(txt);
```



Using Lists

Because screen size is limited, lists are the most common basic UI widget on devices. A List presents the user with a group of items displayed in a single column. The set of elements is rendered using a ListCellRenderer and is extracted using the ListModel. Swing's Model/View/Controller architecture (MVC) makes it possible for a list to represent many UI concepts ranging from a carousel to a To-Do checklist. A list component is relatively simple. It invokes the model in order to extract the displayed or selected information and invokes the cell renderer to show it to the user. The list class itself is completely decoupled from everything, so you can extract its content from any source (for example, the network, storage etcetera) and display the information in any form (for example, Checkboxes, Strings, Icons, and so forth).

3.1 Initializing a List

You can create a list in one of four ways:

`List()`	Creates a new instance of List with an empty default model.
`List(ListModel model)`	Creates a new instance of List with the given model.
`List(Object[] items)`	Creates a new instance of List with an array of Objects that are placed into the list model.
`List(Vector items)`	Creates a new instance of List where a set of items are placed into the list model.

3.2 Creating a Model

There are two ways to create a list model:

`ListModel`	Implement the list model interface (use a general purpose implementation of the list model interface derived from the DefaultListModel)
`DefaultListModel`	Everything is taken care of for you.

3.2.1 ListModel

Represents the data structure of the list, thus allowing a list to represent any potential data source by referencing different implementations of this interface. For example, a list model can be implemented in such a way that it retrieves data directly from storage (although caching is recommended). It is the responsibility of the list to notify observers (specifically the view List of any changes to its state (items removed, added, or changed, and so forth) thus the data is updated on the view.

3.2.2 DefaultListModel

The following code demonstrates using the DefaultListModel class with a vector of elements.

```
// Create a set of items
String[] items = { "Red", "Blue", "Green", "Yellow" };
```

```
// Initialize a default list model with “item” inside
DefaultListModel myListModel = new DefaultListModel(items);

// Creating a List with “myListModel”
```

3.3 List Cell Renderer

A list uses an object called a cell renderer to display each of its items. The default cell renderer knows how to display strings and icons and it displays Objects by invoking toString. If you want to change the way the default renderer display icons or strings, or if you want behavior different than what is provided by toString, you can implement a custom cell renderer. You can create a list renderer using ListCellRenderer or DefaultListCellRenderer:

- Section 3.3.1, "ListCellRenderer"
- Section 3.3.2, "DefaultListCellRenderer"

3.3.1 ListCellRenderer

ListCellRenderer is a "rubber stamp" tool that allows you to extract a renderer instance (often the same component instance for all invocations) that is initialized to the value of the current item. The renderer instance is used to paint the list and is discarded when the list is complete.

An instance of a renderer can be developed as follows:

```
public class MyYesNoRenderer extends Label implements ListCellRenderer {
   public Component getListCellRendererComponent(List list,
                      Object value, int index, boolean isSelected) {
      if( ((Boolean)value).booleanValue() ) {
        setText("Yes");
      } else {
           setText("No");
      }
        return this;
   }

   public Component getListFocusComponent(List list) {
      Label label = new label("");
      label.getStyle().setBgTransparency(100);

      return label;
   }
}
```

It is best that the component whose values are manipulated does not support features such as `repaint()`. This is accomplished by overriding repaint in the subclass with an empty implementation. This is advised for performance reasons, otherwise every change made to the component might trigger a repaint that wouldn't do anything but still cost in terms of processing.

3.3.2 DefaultListCellRenderer

The DefaultListCellRender is the default implementation of the renderer based on a Label and the ListCellRenderer interface.

`getListCellRendererComponent()`	Returns a component instance that is already set to renderer "value". While it is not a requirement, many renderers often derive from a component (such as a label) and return "this".
`getListFocusComponent()`	Returns a component instance that paints the list focus item. When the selection moves, this component is drawn above the list items. It's best to give some level of transparency (see code example in Section 3.3.1, "ListCellRenderer"). Once the focused item reaches the cell location then this Component is drawn under the selected item. **Note:** To emulate this animation, call `List.setSmoothScrolling(true)`. This method is optional an implementation can choose to return null

3.4 Adding Items to and Removing Items From a List

You can add items to a list in one of two ways. The first way is to create a ListModel and add it to the list, either when initiating a List or using the method `setModel(ListModel model)`. To remove an item or all items from a List, use `removeItem(int index)` or `removeAll()` methods. For example:

```
// Adding to a list using above DefaultListModel snipped code or:
....
myListModel.addItem("New Item");

// Removing is done by
....
myListModel.removeItem(index);
// or
myListModel.removeAll();
```

3.5 List Events

Two types of events are supported here, ActionEvent and SelectionsListener in addition to `addFocusListener(FocusListener l)` that is inherited from Component. ActionEvent binds a listener to the user selection action, and the SelectionListener is bound to the List model selection listener. The listener bindings mean you can track changes in values inside the Model.

3.5.1 Fixed Selection Feature

The fixed selection feature supports a dynamic versus static item movement in a List. In a Java SE environment the list items are typically static and the selection indicator travels up and down the list, highlighting the currently selected item. The Lightweight UI Toolkit introduces a new animation feature that lets the selection be static while the items move dynamically up and down. To indicate the fixed selection type, use `setFixedSelection(int fixedSelection)` where `fixedSelection` can be one of the following:

`FIXED_NONE`	Behave as the normal (Java SE) List behaves. List items are static and the selection indicator travels up and down the list, highlighting the currently selected item.
`FIXED_TRAIL`	The last visible item in the list is static and list items move up and down.
`FIXED_LEAD`	The first item in the list is static and list items move up and down.
`FIXED_CENTER`	The middle item in the list is static and list items are move up and down.

3.6 Tickers in List

Because list items are essentially rendered as a rubber stamp they can't be treated as typical LWUIT components. Things such as binding event listeners to the components in the list won't work since the list reuses the same component to draw all the entries.

Features such as tickering an individual cell are often requested and the solution isn't trivial because what we need to do is essentially "ticker the List" not the renderer.

The sample below tickers a renderer by registering itself as an animation in the parent form and calling the list's repaint method to ticker. Notice that it has a separate entry for the selected list item otherwise the entire content of the list would constantly ticker.

Example 3–1 Tickering a Renderer

```
class TickerRenderer extends DefaultListCellRenderer {
   private DefaultListCellRenderer selectedRenderer = new
            DefaultListCellRenderer(false);
   private List parentList;   public TickerRenderer()
      super(false);
   }

   public boolean animate() {
      if(parentList != null && parentList.getComponentForm() != null) {
         if(selectedRenderer.isTickerRunning()) {
            if(selectedRenderer.animate()) {
               parentList.repaint();
            }
         }
      }
      return super.animate()
  }
   public Component getListCellRendererComponent(List list, Object value, int
                      index, boolean isSelected) {
       if(isSelected) {          selectedRenderer.getListCellRendererComponent(list, value, index,
                                                             isSelected);
          // sometimes the list asks for a dummy selected value for size
          // calculations and this might break the tickering state
          if(index == list.getSelectedIndex()) {
              if(selectedRenderer.shouldTickerStart()) {
                  if(!selectedRenderer.isTickerRunning()) {
                      parentList = list;
                      list.getComponentForm().registerAnimated(this);
                    selectedRenderer.startTicker(UIManager.getInstance().
                                     getLookAndFeel().getTickerSpeed(), true);
               }           } else {
               if(selectedRenderer.isTickerRunning()) {
                 selectedRenderer.stopTicker();
               }
             }
          }
          return selectedRenderer;
      } else {
      return super.getListCellRendererComponent(list,value,index,
isSelected);
      }
   }
}
```

3.7 Generic List Cell Renderer

GenericListCellRenderer is a renderer designed to be as simple to use as a Component-Container hierarchy. This single class supports most of the common use cases. To GenericListCellRenderer assumes the model contains only `Hashtable` objects. Since `Hashtable`s can contain arbitrary data, the list model is still quite generic and allows application-specific data storage. Furthermore, a `Hashtable` can still be derived and extended to provide domain-specific business logic.

3.7.1 Mapping Components to Hashtable Entries

The GenericListCellRenderer accepts two Container instances and maps them to individual `Hashtable` entries within the model by finding the appropriate components within the given Container hierarchy.

Components are mapped to the Hashtable entries based on the name property of the component (`getName` , `setName`) and the key value within the Hashtable.

Assume a model that contains a Hashtable entry like this:

```
"Foo": "Bar"
"X": "Y"
"Not": "Applicable"
"Number": Integer(1)
```

In this model a renderer loops over the component hierarchy in the Container searching for components whose name matches Foo, X, Not and Number and assigns the appropriate value to them. Notice that if you use image objects as values they are assigned to labels as expected. You can't assign both an image and text to a single label because a key takes only one object. Two labels can be used quite easily in this case.

Even better, the renderer supports list tickering when appropriate, and if a CheckBox appears within the renderer it seamlessly toggles a boolean flag within the Hashtable.

If a value is missing from the Hashtable it is treated as empty and the component is reset. This is an issue if you hardcode an image or text within the renderer and you don't want it replaced. To ensure a component is preserved, use the `setName` property to append `Fixed` to the name. For example: Given an `address`, specify:

```
address.setName("addressFixed");
```

Naming a component within the renderer with $*number* will automatically set it as a counter component for the offset of the component within the list. For example:

```
c.mycomponent("Idate$1");
```

Styling the GenericListCellRenderer is slightly different. The renderer uses the UIID of the Container passed to the generic list cell renderer and the background focus uses that same UIID with the word "Focus" appended. For example:

```
c.setUIID("ListRendererFocused");
```

3.7.2 Focus for Tickering and Fisheye

It is important to notice that the generic list cell renderer will grant focus to the child components of the selected entry if they are focusable, thus changing the style of said entries. For example, a Container might have a child label that has one style when the parent Container is unselected and another when it is selected (focused). This can be easily achieved by defining the label as focusable. Notice that the component will never receive direct focus since it is still a part of a renderer.

Because the generic list cell renderer accepts two or four instances of a Container, the renderer can treat the selected entry differently which is very important for tickering.

It might not be practical to seamlessly clone the Container instances for the renderer's needs, so LWUIT expects the developer to provide two separate instances. This is essential for tickering; there must be separate instances, even if they are identical.

The renderer also supports a fisheye effect in which the selected entry is actually different from the unselected entry in its structure. This behavior supports a pinstripe effect where odd and even rows can have different styles. For example to get a pinstripe effect, provide 4 instances of the Containers and selected and unselected values for odd and even rows.

3.7.3 Hashtable Sample

The best way to learn about the generic list cell renderer and the Hashtable model is by playing with them in the GUI builder, however they can be used in code without any dependency on the GUI builder.

Example 3–2 List With Checkboxes

Here is a simple sample for a list with checkboxes that get updated automatically:

```
List list = new List(createGenericListCellRendererModelData());
list.setRenderer(new GenericListCellRenderer(createGenericRendererContainer(),
createGenericRendererContainer()));
private Container createGenericRendererContainer() {
   Container c = new Container(new BorderLayout());
   c.setUIID("ListRenderer");
   Label name = new Label();
   name.setFocusable(true);
   name.setName("Name");
   c.addComponent(BorderLayout.CENTER, name);
   Label surname = new Label();
   surname.setFocusable(true);
   surname.setName("Surname");
   c.addComponent(BorderLayout.SOUTH, surname);
   CheckBox selected = new CheckBox();
   selected.setName("Selected");
   selected.setFocusable(true);
   c.addComponent(BorderLayout.WEST, selected);
   return c;
}
private Hashtable[] createGenericListCellRendererModelData() {
   Hashtable[] data = new Hashtable[5];
   data[0] = new Hashtable();
   data[0].put("Name", "Shai");
   data[0].put("Surname", "Almog");
   data[0].put("Selected", Boolean.TRUE);
   data[1] = new Hashtable();
   data[1].put("Name", "Chen");
   data[1].put("Surname", "Fishbein");
   data[1].put("Selected", Boolean.TRUE);
   data[2] = new Hashtable();
   data[2].put("Name", "Ofir");
   data[2].put("Surname", "Leitner");
   data[3] = new Hashtable();
   data[3].put("Name", "Yaniv");
   data[3].put("Surname", "Vakarat");
```

```
    data[4] = new Hashtable();
    data[4].put("Name", "Meirav");
    data[4].put("Surname", "Nachmanovitch");
    return data;
}
```

Table and Tree

Unlike the list that uses the render approach to create exceptionally large lists without much of an overhead, the tree and table are more "stateful" components and use a more conventional approach of nesting components.

To create a table instance a developer needs to first instantiate a model with the data and then create a table as follows:

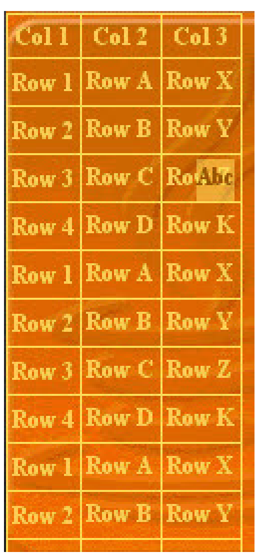
4.1 Table

A table is an editable grid component with variable sizes for its entries. Entries can be editable or not. Just like the list, the table has a model (`TableModel`) and a default model implementation (`DefaultTableModel`).

To create a table instance a developer needs to first instantiate a model with the data and then create a table as follows:

```
TableModel model = new DefaultTableModel(new String[] {
   "Col 1", "Col 2", "Col 3"}, new Object[][] {
  {"Row 1", "Row A", "Row X"},  {"Row 2", "Row B", "Row Y"},
  {"Row 3", "Row C", "Row Z"},
  {"Row 4", "Row D", "Row K"},
});
Table table = new Table(model);
```

Figure 4–1 Sample Table



A cell can be made editable by overriding the `isCellEditable` method of the model as follows:

```
public boolean isCellEditable(int row, int col) {
```

```
    return col != 0;
}
```

The table component contains a few more elaborate features such as the ability to span columns and rows and determine their width or height as percentage of available space. A table can be made to scroll on the X axis as well by setting it to `setScrollableX(true)`, in which case it can "grow" beyond the screen size.

To control the "rendering", the way in which a table creates the cells within it one needs to derive the table itself and override the method `createCell` as such:

```
Table table = new Table(model) {
    protected Component createCell(Object value, int row, int column, boolean
editable) {
        // custom code for creating a table cell
        ...
    }
};
```

Notice that components created using `createCell` are "live" for the duration of the table's existence and are able to receive events and animate. They also occupy resources for the duration of the table’s existence.

4.2 Tree

The LWUIT tree is remarkably similar to the table in its design. It however represents a hierarchical view of data such as a filesystem. In that sense a tree is must be provided with a model to represent the underlying data. It is assumed that the underlying data is already "hierarchic" in its nature, such as a corporate structure or a file system.

The tree model exists as an interface for this reason alone. Building it as a class doesn't make sense for the common use case of a domain specific data model. To create a tree model one must implement the two methods in the interface: `getChildren` and `isLeaf`.

`getChildren` is the "heavy lifter" within the interface. It has one argument for the parent node and returns the children of this node as a vector. This method is called with a null argument for its parent representing the "root" of the tree (which isn't displayed). From that point forward all calls to the method will be with objects returned via this method (which are not leafs).

`isLeaf` is trivial. It just indicates whether the object within a tree is a leaf node that has no children and can't be expanded.

Figure 4–2 Tree Sample

For example, the Tree would invoke `getChildren(null)` and receive back the String's "X", "Y' and "Z" within the return vector. It would then call `isLeaf("X")`, `isLeaf("Y")`, `isLeaf("Z")` and render the tree appropriately (as parent nodes or as leafs based on the response to `isLeaf`).

If the user clicks the "X" node and it is not a leaf the tree expands to contain (in addition to the existing nodes) the response for `getChildren("X")` as subnodes of "X".

Most of the code below relates to the model. It would be more domain specific for any specific case.

```
class Node {
    Object[] children;
    String value;

    public Node(String value, Object[] children) {
        this.children = children;
        this.value = value;
    }

    public String toString() {
        return value;
    }
}
TreeModel model = new TreeModel() {
    Node[] sillyTree =  {
        new Node("X", new Node[] {
           new Node("Child 1", new Node[] {
           }),
           new Node("Child 2", new Node[] {
           }),
           new Node("Child 3", new Node[] {
           }),
       }),
```

```
        new Node("Y", new Node[] {
            new Node("A", new Node[] {
            })
        }),
        new Node("Z", new Node[] {
            new Node("A", new Node[] {
            }),
        }),
    };

    public Vector getChildren(Object parent) {
        Node n = (Node)parent;
        Object[] nodes;
        if(parent == null) {
            nodes = sillyTree;
        } else {
            nodes = n.children;
        }
        Vector v = new Vector();
        for(int iter = 0 ; iter < nodes.length ; iter++) {
            v.addElement(nodes[iter]);
        }
        return v;
    }

    public boolean isLeaf(Object node) {
        Node n = (Node)node;
        return n.children == null || n.children.length == 0;
    }
};

Form treeForm = new Form("Tree");
treeForm.setLayout(new BorderLayout());
treeForm.addComponent(BorderLayout.CENTER, new Tree(model));
treeForm.show();
```

4.3 Customizing the Tree

The tree has special static methods to determine icons appropriate for expanded or folded folder and leaf nodes: `setFolderOpenIcon(Image)`, `setFolderIcon(Image),setNodeIcon(Image).`

Besides that, one can derive the tree component and override the `createNodeComponent` method to customize the returned component in any desired way.

Using Dialogs

A Dialog is a form that occupies a part of the screen as a top level component. By default dialogs always appear as a modal entity to the user. Modality indicates that a dialog blocks the calling thread even if the calling thread is the *Event Dispatcher Thread* (EDT). Dialogs allow us to prompt users for information and rely on the information being returned as a response after the dialog `show` method. Each Dialog has a body that is located in the center of the dialog. The Body can contain a component, so you can use your own customer component or pre-built container.

> **Note:** A modal dialog does not release the block until a dispose method is called. For example, calling `show()` from another form does not release the block.

5.1 Dialog Types

For better user experience, dialogs have five types of alerts. The alert type indicates a sound to play or an icon to display if none is explicitly set:

- ALARM
- CONFIRMATION
- ERROR
- INFO
- WARNING

By default the alerts are set to play the device alert sounds.

Icons are not currently provided by default, but you can manually add them to customized dialogs. Icons can be used to indicate the alert state, similar to JDialog icons in Swing. See http://java.sun.com/docs/books/tutorial/uiswing/components/dialog.html (`http://java.sun.com/docs/books/tutorial/uiswing/components/dialog.html`).

5.2 Creating a Dialog

To create and show a dialog you can do the following:

- Create and show the dialog using one of the static show methods.
- Use `new Dialog()` and invoke its `show()` method. The static methods are only helpers.

The arguments to all of the show methods are standardized, though the number of arguments for each method varies. The static show methods provide support for laying out standard dialogs, providing icons, specifying the dialog title and text, and customizing the button text.

The following list describes each argument. To see the exact list of arguments for a particular method, see the Dialog API in the API documentation located in *install-dir*`/docs/api/lwuit`.

- `String` *title*

The title of the dialog.

- `Component` *body*

  Component placed in the center of the dialog. This component can be a container that contains other components.

- `String` *text*

  The text displayed in the dialog which can be used instead of Body.

- `Command[]` *cmds*

  Array of commands that are added to the dialog. Any click on any command disposes of the dialog. Examples of commands are OK and Cancel.

- `int` *type*

  The type of the alert can be one of TYPE_WARNING, TYPE_INFO, TYPE_ERROR, TYPE_CONFIRMATION or TYPE_ALARM to indicate the sound to play or an icon to display.

- `Image` *icon*

  The icon to display in the dialog.

- `long` *timeout*

  A timeout in milliseconds, after which the dialog closes and null is returned. If time-out value is 0, the dialog remains open indefinitely, until its dispose method is invoked.

- `Transition` *transition*

  The transition installed when the dialog enters and leaves the screen. For more information see Section 13.3, "Transition".

- `String` *okText*

  The text to appear in the command dismissing the dialog.

- `String` *cancelText*

  Optionally null for a text to appear in the cancel command for canceling the dialog.

- `int` *top*

  Inset in pixels between the top of the screen and the form.

- `int` *bottom*

  Inset in pixels between the bottom of the screen and the form.

- `int` *left*

  Inset in pixels between the left of the screen and the form.

- `int` *right*

  Inset in pixels between the right of the screen and the form.

- `boolean` *includeTitle*

  Whether the title should hang in the top of the screen or be glued onto the dialog content pane.

5.2.1 Return Types of Show Methods

You can use one of three convenient return value show methods: void, Command, or boolean.

- Command returns the command object the user clicked. See the Command API in the API documentation found in *install-dir*`/docs/api/lwuit`.
- The boolean value of true is returned when the OK command is pressed or if `cancelText` is null (meaning there is no cancel command text visible). It is false otherwise.

5.2.2 Non-Static Show Methods

The dialog API provides two non-static methods to create two more types of dialogs.

The first method takes no arguments and produces a dialog without any commands. The only way to close such a dialog is to invoke the `dispose()` method on the dialog. Since the dialog is blocking, meaning once the dialog is displayed its calling thread can not proceed until it is closed, the call to `dispose` must be made from a different thread. To do this, schedule the call to `dispose` with a timer thread. Note that the timer thread must be started before the dialog is displayed. This approach is referred to as an auto-closing dialog.

The second dialog type has five parameters. The first four are the four wing insets (top, bottom, left, and right) and the fifth parameter determines whether to include the Dialog title assigned through the dialog constructor (see Figure 5–1).

```
// Call show with inset parameters

dialog.show(90, 90, 10, 10, true);
```

5.2.3 Using the `dispose()` Method

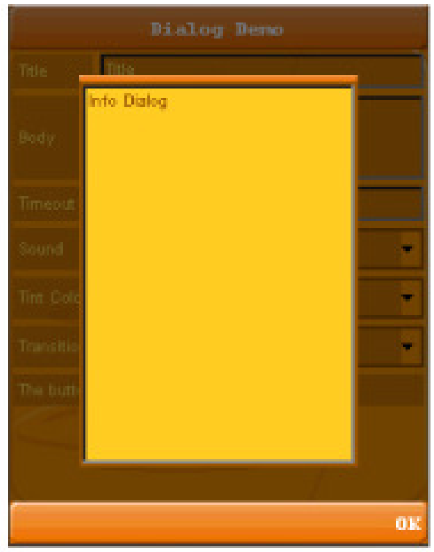
The dispose methods closes the current dialog and returns to the parent form. When `show()` is used without arguments, one way to close the dialog is to set a timer to call `dispose` just before calling the `show` method (otherwise the dispose method is never performed).

5.2.4 Getting the User's Input from a Dialog

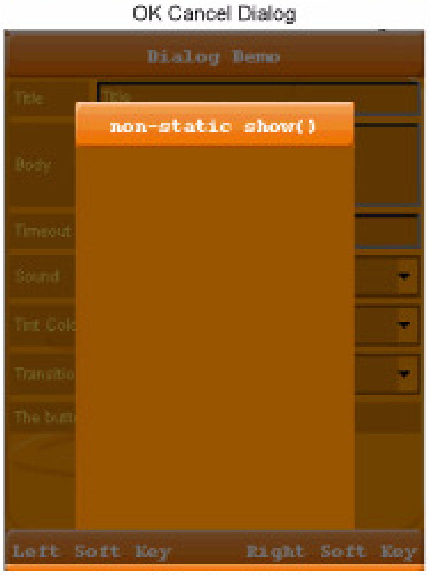
As mentioned in Section 5.2.2, "Non-Static Show Methods", return value types can be either Command or a boolean value. For example, if a user has a dialog with two commands, Approve and Decline, the user clicks and the selected command is returned. For the boolean return type, a true or false value indicates whether the user clicked the OK command.

Figure 5–1 Typical Dialogs

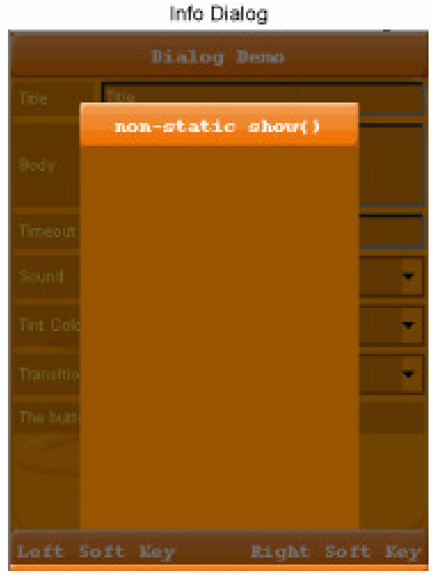
OK Cancel Dialog



Info Dialog



Non-static `show()` Dialog



Dialog with Insets

Using Layout Managers

This chapter shows you how to use the layout managers provided by the Lightweight UI Toolkit library. It also gives an example of writing a custom layout manager. For each layout manager, this chapter supplies sample code demonstrating how to use the layout manager and a general illustration.

In Lightweight UI Toolkit you can find the following layout managers:

- BorderLayout
- BoxLayout
- FlowLayout
- GridLayout
- GroupLayout
- Coordinate Layout
- Table Layout

6.1 BorderLayout

A BorderLayout object has five areas. These areas are specified by the BorderLayout constants:

- Center
- East
- North
- South
- West

When adding a component to a container, specify the component's location (for example, `BorderLayout.CENTER`) as one of the arguments to the addComponent method. If this component is missing from a container, controlled by a BorderLayout object, make sure that the component's location was specified and that no other component was placed in the same location.

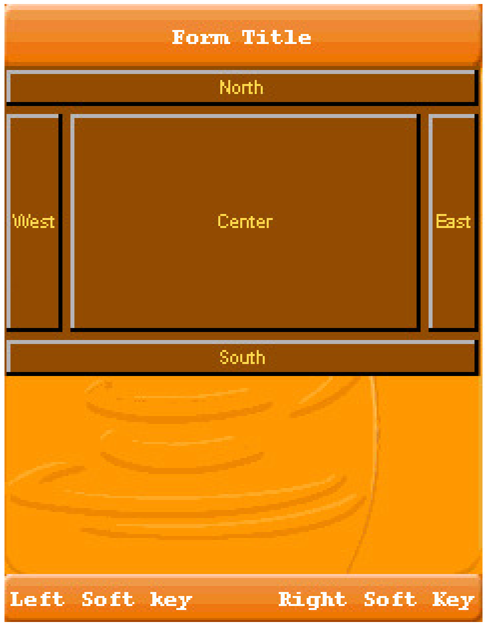
```
addComponent(BorderLayout.CENTER, component) // preferred
```

or

```
addComponent(“Center”, component) // valid but error prone
```

The center area gets as much of the available space as possible. The other areas expand only as much as necessary to fit the components that have been added to it. Often a container uses only one or two of the areas of the BorderLayout object — just the center, or the center and the bottom.

Figure 6–1 BorderLayoutLocations



6.2 BoxLayout

The BoxLayout class puts components either on top of each other or in a row – your choice.

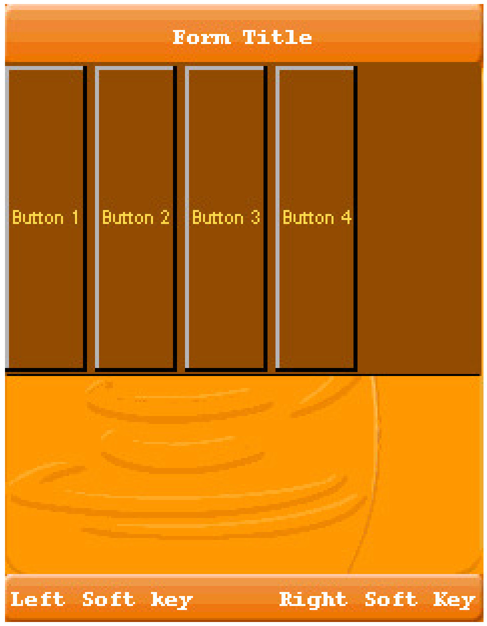
6.2.1 X_AXIS

To lay out components in a row, use `BoxLayout.X_AXIS` as the axis indication.

```
BoxLayout boxLayout = new BoxLayout(BoxLayout.X_AXIS);
```

In this layout, the box layout manager honors the component width of each layout component to fill the width of the container, and the height is determined by the container height. Any extra space appears at the right side of the container, as shown in Figure 6–2.

Figure 6–2 BoxLayout.X_AXIS Components in a Row



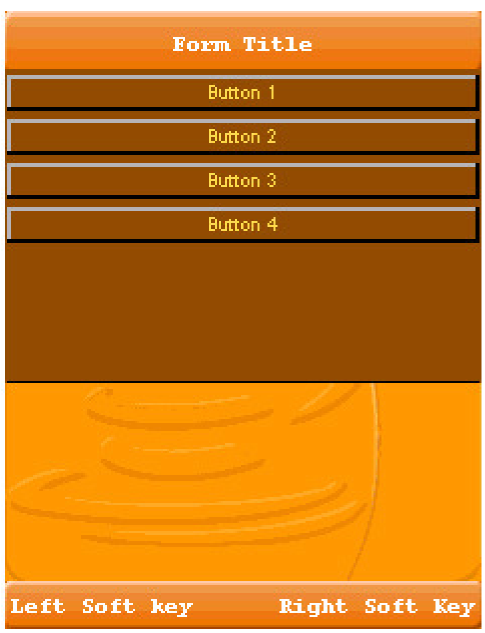
6.2.2 Y_AXIS

To lay out components in a column, use `BoxLayout.Y_AXIS` as the axis indication.

```
BoxLayout boxLayout = new BoxLayout(BoxLayout.Y_AXIS);
```

In this layout, the box layout manager honors the component height of each layout component to fill the height of the container, and the width is determined by the container width. Any extra space appears at the bottom of the container, as shown in Figure 6–3.

Figure 6–3 BoxLayout_Y_Axis Components in a Row



6.3 FlowLayout

The FlowLayout class provides a very simple layout manager that is the default layout manager for Container objects.

The FlowLayout class puts components in a row, sized at their preferred size. If the horizontal space in the container is too small to put all the components in one row, the FlowLayout class uses multiple rows. To align the row to the left, right, or center, use a FlowLayout constructor that takes an alignment argument.

The code snippet below creates a FlowLayout object and the components it manages.

```
FlowLayout exampleLayout = new FlowLayout();

...

container.setLayout(exampleLayout);

container.addComponent(new Button("Button 1"));
container.addComponent(new Button("Button 2"));
container.addComponent(new Button("Button 3"));
container.addComponent(new Button("Button 4"));
```

Figure 6–4 FlowLayout Default Alignment

When constructing a FlowLayout manager you can select either the Left, Right, or Center option to set up the component's orientation. The default alignment is Left. The following code snippet applies the Right component orientation to the above exampleLayout.

```
FlowLayout exampleLayout = new FlowLayout(Component.RIGHT);
```

Figure 6–5 FlowLayout With Right Alignment

6.4 GridLayout

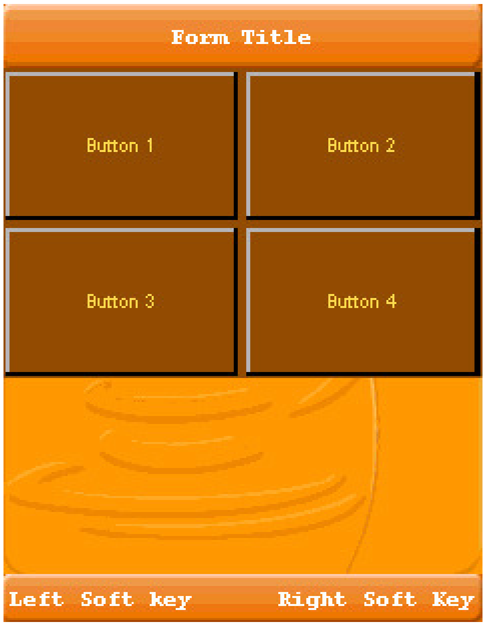
A GridLayout object places components in a grid of cells. Each component takes all the available space within its cell, and each cell is exactly the same size.

The code snippet below creates the GridLayout object and the components it manages.

```
GridLayout exampleLayout = new GridLayout(0,2);

...
container.setLayout(exampleLayout);

container.addComponent(new Button("Button 1"));
container.addComponent(new Button("Button 2"));
container.addComponent(new Button("Button 3"));
container.addComponent(new Button("Button 4"));
```

In this example the constructor of the GridLayout class creates an instance that has two columns and as many rows as necessary.

Figure 6–6 GridLayout With Two Columns



6.5 GroupLayout

GroupLayout is a layout manager that was developed for GUI builders such as Matisse, the Java SE GUI builder delivered with the NetBeans IDE. Although the layout manager was originally designed to suit GUI builder needs, it also works well for manual coding. To get more information you can refer to the GroupLayout API (http://java.sun.com/javase/6/docs/api/javax/swing/GroupLayout.html (`http://java.sun.com/javase/6/docs/api/javax/swing/GroupLayout.html`)) or review the Swing GroupLayout tutorial at:

http://java.sun.com/docs/books/tutorial/uiswing/layout/group.html (`http://java.sun.com/docs/books/tutorial/uiswing/layout/group.html`)

6.6 Coordinate Layout

Unlike other layout managers coordinate layout assigns a component an absolute position in relation to the space available within the UI. The coordinate layout allows developers to position components within an X/Y location, however, it doesn't guarantee the position won't change and doesn't determine absolute positions.

Instead coordinate layout accepts positions as "relative" and calculates the actual position based on available space. This is essential since the available size for a container might change at runtime based on font size, screen rotation, etcetera.

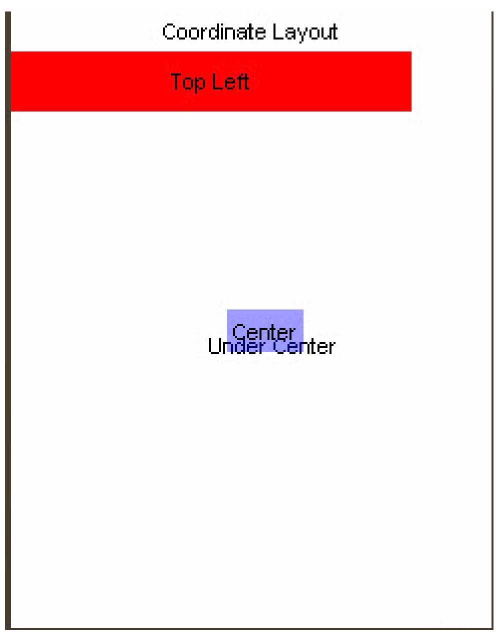
For example, a coordinate layout for 200x200 will show a 20x20 component placed in the 90x90 position exactly in the center, regardless of the actual size of the container. If the container is laid out to a larger size, for example, 190x300 the component in the center would still be centered.

Unlike the other standard layouts in LWUIT the coordinate layout allows positioning components on top of one another to achieve z-ordering. The z-ordering is determined by the order in which the components are placed into the parent container.The last component added is the one on top.

```
Display.init(this);
final Form mainForm = new Form("Coordinate Layout");
mainForm.setLayout(new CoordinateLayout(200, 200));

Label centeredLabel = new Label("Center");
centeredLabel.setX(90);
centeredLabel.setY(90);
centeredLabel.getUnselectedStyle().setBgTransparency(100);
centeredLabel.getUnselectedStyle().setBgColor(0xff);

Label underCenter = new Label("Under Center");
underCenter.setX(80);
underCenter.setY(95);

Label top = new Label("Top Left");
top.setAlignment(Component.CENTER);
top.setX(0);
top.setY(0);
top.setPreferredW(200);
top.setPreferredH(30);
top.getUnselectedStyle().setBgColor(0xff0000);

mainForm.addComponent(underCenter);
mainForm.addComponent(centeredLabel);
mainForm.addComponent(top);

mainForm.show();
```

This code produces Figure 6–7:

Figure 6–7 Coordinate Layout Sample



There are several interesting things we can glean even from this simple example:

- Coordinate layout must be hard-coded. The coordinates are implicitly scaled by LWUIT so there is no need to use logic, such as `getWidth/Height,` to calculate positions.

- Elements are sized based on their preferred size, yet positioned based on their X and Y coordinates. Their dimensions determined via `setWidth` and `getHeight` are ignored.
- Unlike the X and Y coordinates that are relative to layout dimensions, the preferred size is absolute in pixels and should be calculated based on content dimensions. This works as expected as long as you don't change the preferred size on your own.
- Alignment and other LWUIT related positioning logic should work as you would expect.

6.7 Table Layout

The table layout is a part of the table component discussed later, however it is quite useful on its own. It is largely inspired by the HTML table tag and also influenced by AWT's GridBagLayout.

The table layout is a constraint based layout (similar to the border layout). Other layout managers expect components to be added on their own. For example:

```
container.addComponent(component);
```

The table layout container expects something like this:

```
container.addComponent(tableConstraint, component);
```

Notice that this syntax is optional. If the constraint is omitted, the component is placed in the next available cell.

The table layout will automatically size components to the largest preferred size in the row or column until you run out of space. If the table is not horizontally scrollable this happens when the edge of the parent container is reached (near the edge of the screen), and additional components are "crammed together". Notice that all cells in the table layout are always sized to fit the entire cell. To change a cell’s alignment or margin, use the Component or Style methods.

The constraint argument is an instance of TableLayout.Constraint that can be used only once. Reusing the instance will cause an exception.

A constraint can specify the absolute row/column where the entry should fit as well as spanning between cell boundaries.

Figure 6–8 Table Layout Sample

In Figure 6–8, the "First" cell is spanned vertically while the "Spanning" cell is spanned horizontally. This is immensely useful in creating elaborate UIs.

Constraints can also specify a height/width for a column/row that will override the default. This size is indicated in percentage of the total table layout size. In the code below you can see that the "First" label is sized to 50% width while the "Fourth" label is sized to 20% height.

```
final Form mainForm = new Form("Table Layout");
TableLayout layout = new TableLayout(4, 3);
mainForm.setLayout(layout);
TableLayout.Constraint constraint = layout.createConstraint();
constraint.setVerticalSpan(2);
constraint.setWidthPercentage(50);
mainForm.addComponent(constraint, new Label("First"));
mainForm.addComponent(new Label("Second"));
mainForm.addComponent(new Label("Third"));

constraint = layout.createConstraint();
constraint.setHeightPercentage(20);
mainForm.addComponent(constraint, new Label("Fourth"));
mainForm.addComponent(new Label("Fifth"));
constraint = layout.createConstraint();
constraint.setHorizontalSpan(3);
Label span = new Label("Spanning");
span.getStyle().setBorder(Border.createLineBorder(2));
span.setAlignment(Component.CENTER);
mainForm.addComponent(constraint, span);
mainForm.show();
```

Using the Style Object

The Style object sets colors, fonts, transparency, margin, padding, images, and borders to define the style for a given component. Each Component contains a selected Style Object and allows Style modification at runtime using `component.getSelectedStyle()` and `component.getUnselectedStyle()`. The style is also used in Theming (Chapter 8). When a Theme is changed, the Style objects are updated automatically.

7.1 Color

Each Component has two adjustable colors:

Foreground color	The component foreground color that usually refers to the component text color. For example, for a Button it's the text color.
Background color	The component background color.

The color specification is RGB. There is no alpha channel within the color (the background transparency is separate).

Valid values are integers ranging from 0x000000 to 0xffffff (black to white respectively) or a decimal number.

7.2 Font

Fonts are set with the Font object (see the Font API in the API documentation located in *install-dir*`/docs/api/lwuit`. Lightweight UI Toolkit supports both for Bitmap fonts and for system fonts, similar to common MIDP fonts. Fonts are discussed in Chapter 9.

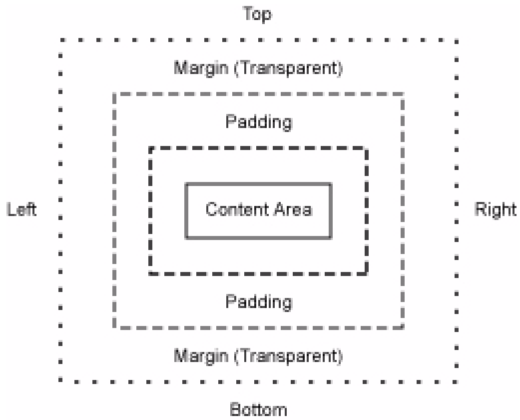
7.3 Transparency

Lightweight UI Toolkit style supports background component transparency, to add flexibility and appeal to the UI. To set a component transparency level, call setBgTransparency and specify an integer or a byte. The integer value must range between 0 to 255, where 255 (the default) is opaque.

7.4 Margin and Padding

Margin and Padding are inspired by the CSS Box Model. Each component has a main content area (for example, text or icon) and optional surrounding padding and margin areas. The size of each area is specified by four integers that represent the top, bottom, left and right space (similar to component Insets terminology in SWING). The following diagram shows the placement of the areas in relation to the component content area:

Figure 7–1 Padding and Margin Relationships



Padding and margins can be set as follows:

```
// Setting padding with positive values
setPadding(int top, int bottom, int left, int right)

// orientation can be Component.TOP, BOTTOM, LEFT or RIGHT
setPadding(int orientation, int gap)

// Setting margin with positive values
setMargin(int top, int bottom, int left, int right)

// orientation can be Component.TOP, BOTTOM, LEFT or RIGHT
setMargin(int orientation, int gap)
```

7.5 Images

In Style, Images refer to a component background image. By default components do not have a background image, so the bgImage parameter is null by default. For more details about images, please refer to Chapter 9.

7.6 Borders

The Style object supports defining custom rendering of a border. There are several default built-in border types (see the Javadoc™ of the Border class). Borders can either replace the background painter (as is the case with round borders and sometimes with image borders) or they can be rendered after the component itself is rendered. A custom border can be built by deriving the Border class and overriding the appropriate methods.

A border is rendered into the padding area of the component so it is important that the component padding is large enough to contain the border drawing.

7.6.1 Bevel

The bevel border type presents a simple 3D style border that can appear lowered or raised, providing simple depth perception for actions such as button presses.

Figure 7–2 Bevel Lowered Border

Figure 7–3 Bevel Raised Border

7.6.2 Etched

The etched border type provides a look similar to an engraved line. Like the bevel border, it too can appear raised or lowered.

Figure 7–4 Etched Lowered Border

Figure 7–5 Etched Raised Border

7.6.3 Line

The line border just draws a rectangle around the component with the option of defining the thickness and color of the rectangle.

7.6.4 Round

The round border draws a rounded rectangle and optionally fills the background appropriately.

PERFORMANCE WARNING:

The round border might be very expensive! A round border is cheap for a completely opaque solid color. However, when using features such as gradients, images or alpha channel the round border effect is calculated on the fly! This is computationally expensive. We recommend trying to achieve these same effects with image borders which are cheaper.

The round border supports defining its color (or using the theme color) and defining the size of the arcs rounding the border.

Figure 7–6 Round Border

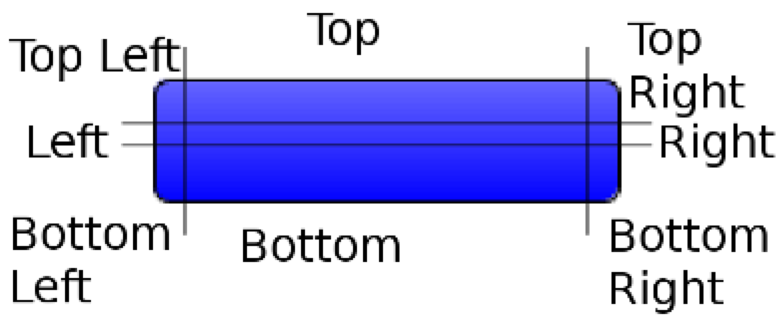
7.6.5 Image

The image border option will only appear when images exist in the resource file, you can read more on creating image borders in the image border wizard. Image borders come in 4 flavors:

7.6.5.1 9 Part

Uses 9 or 8 images to represent the border for the component. The structure is as shown in Figure 7–7:

Figure 7–7 9-Part Image Border



Notice that the center image is optional. The top, bottom, left and right images are tiled while the corners (Top Left, Top Right, Bottom Left, and Bottom Right) are kept in place. The center image (if defined) is also tiled. This allows LWUIT to resize the component without any scaling, degradation, performance cost, or memory overhead.

Remember that drawing an image is an expensive operation, so images in the image border shouldn't be too small. For example, a common designer mistake is to produce a single pixel image for tiling. LWUIT seamlessly crops tiled images, so you should make an effort to make images a reasonable size (when in doubt use something in the area of 80-100 pixels).

7.6.5.2 3 Part

The 3 Part is somewhat unique to LWUIT and relies on MIDP's fast rotation drawing. It assumes perfectly rectangular images and draws the top left image rotated to produce all corners and does the same for the top image (center is again optional and used as usual). Thus the 3 part border can produce some attractive results in a smaller size.

7.6.5.3 Horizontal/Vertical

The image border is highly biased to symmetric shapes that can be enlarged to all directions. However, some shapes (such as the iPhone's angular back button) cannot be cut into a 9-patch image without causing artifacts.

The Horizontal and Vertical Image borders accept 3 images each and only grow on one axis. We don't recommend using them freely, even when text (which can vary wildly in size) isn't used, one often needs to align to text which requires resizing.

7.7 Style Listener

The Style listener gives you the ability to track changes in a certain component style object. For example you might want to monitor changes in the background color of a component, and react accordingly.

The following code shows how to add a listener and track any style property change to the Font.

```
myComponent.getStyle().addStyleListener(new StyleListener() {
   public void styleChanged(String propertyName, Style source) {
      if (propertyName.equals(Style.FONT)) {
         System.out.println("Font of myComponent got changed.");
      }
   }
});
```

7.8 Painters

Painters in Style refers to the component's background drawing. The Painter draws itself and then the component draws itself on top. For more information please refer to Chapter 10.

To set a painter, use the `setBgPainter` method. For example to set myPainter as the component background painter, write:

```
mycomponent.getStyle().setBgPainter(myPainter);
```

Theming

The Lightweight UI Toolkit library supports pluggable themes similar to CSS and somewhat simpler than Swing's pluggable Look And Feel.

8.1 Basic Theming

Every LWUIT component has a style associated with it (see Chapter 7). This style can be manipulated manually and can be customized using a set of definitions for a specific component type. For example, in order to make the backgrounds for all the buttons red you can use the following theme:

```
Button.bgColor=ff0000
```

This theme sets the background in the style object within the button object to red. A theme can be packaged into a resource file (see Chapter 9) and it can be loaded or switched in runtime. In order to update a theme after switching you must refresh the root component (the Form/Dialog containing our component tree) using the `refreshTheme` method to update all styles.

Note: Manually modified style elements are not updated when switching a theme.

For example, if you have a button whose background is customized to blue, and you load or refresh a theme with a different background color for buttons, the new theme affects all button instances except for the one you have modified manually.

This allows you to determine styles for specific components yet still be able to use themes for the general look of the application without worrying about how they affect your changes.

A theme file is very similar in spirit to CSS, yet it is much simpler and it is structured like a Java properties file. A theme file is comprised of key value pairs. The key acts in a similar way to a CSS selector that indicates the component or attribute affected by the theme value. For example:

- `Button.font` – font for all buttons
- `font` – default application font applied to all components where no default is defined

The key element is comprised of an optional unique identifier ID for the component (the UIID) and a required attribute type. Unlike CSS, themes do not support elements such as hierarchy or more complex selectors.

Component UIIDs correspond to the component class name by convention. For example.: `Button`, `Label`, `CheckBox`, `RadioButton`, `Form`, etcetera.

The supported attributes and their value syntax are illustrated in Table 8–1:

Table 8–1 Attributes

Attribute	Value
bgGradient	Determines the values for the gradient of the image. Accepts source/destination color as well as X/Y of the center of a radial gradient.
`bgColor`	Hexadecimal number representing the background color for the component in an unselected widget. For example, blue would be: `ff`
bgImage	Name of an image from within the resource that should be used as the background for this component. The image referenced must exist within the resource using the same name mentioned here. See the resources chapter for further details about resources and theme files.
bgType	Allows determining the type of the background whether it is an image, color, or gradient. Valid values are: BACKGROUND_IMAGE_SCALED BACKGROUND_IMAGE_TILE_BOTH BACKGROUND_IMAGE_TILE_VERTICAL_ALIGN_LEFT BACKGROUND_IMAGE_TILE_VERTICAL_ALIGN_CENTER BACKGROUND_IMAGE_TILE_VERTICAL_ALIGN_RIGHT BACKGROUND_IMAGE_TILE_HORIZONTAL_ALIGN_TOP BACKGROUND_IMAGE_TILE_HORIZONTAL_ALIGN_CENTER BACKGROUND_IMAGE_TILE_HORIZONTAL_ALIGN_BOTTOM BACKGROUND_IMAGE_ALIGNED_TOP BACKGROUND_IMAGE_ALIGNED_BOTTOM BACKGROUND_IMAGE_ALIGNED_LEFT BACKGROUND_IMAGE_ALIGNED_RIGHT BACKGROUND_IMAGE_ALIGNED_TOP_LEFT BACKGROUND_IMAGE_ALIGNED_TOP_RIGHT BACKGROUND_IMAGE_ALIGNED_BOTTOM_LEFT BACKGROUND_IMAGE_ALIGNED_BOTTOM_RIGHT BACKGROUND_IMAGE_ALIGNED_CENTER BACKGROUND_GRADIENT_LINEAR_HORIZONTAL BACKGROUND_GRADIENT_LINEAR_VERTICAL BACKGROUND_GRADIENT_RADIAL
`fgColor`	Hexadecimal number representing the foreground color for the component usually used to draw the font in an unselected widget. For example, red would be: `ff0000`
`font`	The name of the bitmap or system font from the build XML file.
`margin`	The amount of margin for the component defined as 4 comma-separated integer values representing top, bottom, left, and right. For example, `1, 2, 3, 4` results in 1 pixel margin top, 2 pixels margin bottom, 3 pixels margin left and 4 pixels margin right.
`padding`	Padding is identical to margin in terms of format but it updates the padding property of the component. To understand padding versus margin further please refer to the box model explanation in Section 7.4, "Margin and Padding".

Table 8–1 (Cont.) Attributes

Attribute	Value
`transparency`	A number between 0 and 255 representing the opacity of a component's background. 0 means the background of the component doesn't draw at all (fully transparent) while 255 represents a completely opaque background. Notice that this value currently has no effect on background images (although this behavior might change in a future release).

To install a theme you must load it from the Resources class (see Chapter 9), from which you receive the already parsed hashtable containing the selectors (keys) and their appropriate values. You then submit this class to the UI manager's `setThemeProps` method in order to update the current theme. It is a good practice to call `refreshTheme` on all components in memory (even those that are not visible) otherwise behavior is unpredictable.

8.2 Look and Feel

While a theme is remarkably powerful and relatively simple, it doesn't allow the deep type of customization some applications require. Developers would often like the ability to control the drawing of all widgets from a single location, relieving them of the need to subclass widgets and manipulate their paint behavior.

LWUIT delegates all drawing to a single abstract base class called `LookAndFeel`, an instance of which may be obtained from the `UIManager`. This class has a concrete subclass which provides the default LWUIT look called `DefaultLookAndFeel`. Both `LookAndFeel` and `DefaultLookAndFeel` may be subclassed in order to extend/replace their functionality.

The look and feel class has methods for determining the boundaries (preferred size) of component types and for painting all components. In addition it has some special methods that allow you to bind special logic to components and manually draw widgets such as scroll bars. It is the responsibility of the Look and Feel developer to properly use the Style objects delivered by the theme. If you replace the look and feel class, you must make sure to extract values appropriately from component styles of the theming functionality or LWUIT can break.

For further details about the look and feel classes, please consult the API documentation.

Resources

LWUIT permits the following resource elements:

- Image Resources
- Dynamic Fonts
- Localization (L10N) bundles
- Themes
- GUI / UI Builder

Resources can be delivered as a bundle (a binary file that can be loaded and used on the device). A bundle can combine several different resource types within a single file, thereby easing distribution and improving compression. LWUIT supports two methods for creating a resource bundle: a set of Ant tasks, or the graphical Resource Editor utility (see Section 9.2, "The LWUIT Resource Editor").

9.1 Resource Elements

The following sections detail the five resource types and the ways in which they relate to the resource bundle mechanism.

9.1.1 Building a Bundle

A resource bundle can be built using Ant during the standard application build process. Resource files convert existing files into bundles as necessary. An application can have any number of resource files.

A resource file it is loaded fully into memory (due to Java ME IO constraints), so you should group resources based on the needs of the application flow. This allows the application to load only the necessary resources for a given form or use case and leaves memory free for additional resources needed by other forms or use cases.

9.1.1.1 Creating a Resource

To create a resource, use code similar to the following example in your build file:

```
<taskdef
  classpath="editor.jar"
  classname="com.sun.lwuit.tools.resourcebuilder.LWUITTask"
  name="build" />
<build dest="src/myresourceFile .res">
  <image file="images/myImage.png" name=”imageName” />
</build>
```

You can add several additional types of resources to the build tag. These optional resource tags are explained in the remainder of this chapter.

9.1.1.2 Loading a Resource

To load a resource into your application, use code similar to this:

```
Resources res = Resources.open(”/myresourceFile.res”);
```

```
Image i = res.getImage("imageName");
```

9.1.2 Image Resources

There are several types of images in LWUIT, most of which can be stored either individually in the Java archive (JAR™) or packaged as part of a resource bundle.

To load an image stored in the JAR file, use the following code:

```
Image image = Image.createImage("/images/duke.png");
```

The Image tag supports the following attributes:

`name`	The name of the resource (defaults to the name of the file name).
`file`	The file that would be used for the image (required)

Once loaded, the image is ready to be used as a background image of a component or even as an icon for a component that can contain an image.

9.1.3 Fonts

The LWUIT library supports bitmap fonts, system fonts, and loadable fonts. System fonts use basic native fonts and are based on the common MIDP fonts. For more detailed information please see the Font API in the API documentation located in *install-dir*`/docs/api/lwuit`.

Bitmap fonts generate fonts on the desktop as image files. These image can be used to draw desktop quality fonts on a device without requiring specific support from the device.

Loadable fonts support specifying a font as a name or even as a TrueType font file, if the underlying operating system supports such fonts, the font object would be created.

All fonts can be used in a theme file and are represented using the Font class in LWUIT.

9.1.3.1 System Font

Three basic parameters define a system font:

`Face`	Valid values are `FACE_SYSTEM`, `FACE_PROPORTIONAL` and `FACE_MONOSPACE`.
`Style`	Valid values are `STYLE_PLAIN`, `STYLE_ITALIC`, `STYLE_BOLD`.
`Size`	Valid values are `SIZE_SMALL`, `SIZE_MEDIUM`, `SIZE_LARGE`.

To create a system font, use the following code:

```
Font.createSystemFont(Font.FACE_SYSTEM,
                      Font.STYLE_BOLD,
                      Font.SIZE_MEDIUM);
```

To create a bold italic font style use code similar to the following:

```
Font.createSystemFont(Font.FACE_SYSTEM,
                      Font.STYLE_BOLD | Font.STYLE_ITALIC,
                      Font.SIZE_MEDIUM);
```

9.1.3.2 Dynamic Fonts

Different platforms have different font support. For example, phones usually only support system and bitmap fonts while TVs usually support TrueType fonts but don't work well with bitmap fonts. LWUIT has support for defining fonts in resources that allow a resource to adapt for different devices. To support portability LWUIT allows specifying a loadable font if such a font is supported by the underlying system and allows bundling bitmaps for increased portability. As a fallback a system font is always defined, thus if the native font isn't supported or a developer isn't interested in using a bitmap font the system font fallback can always be used. It is possible to define such a font using the Ant task with the following syntax:

```
<build dest="src/myresourceFile.res">
     <font logicalName=”SansSerif” name=”myFont” size=”20” />
</build>
```

The following attributes are supported for the font Ant task:

`name`	Name of the font to load from the resource file (optional: defaults to logical name or file name).
`charset`	Defaults to the English alphabet, numbers and common signs. Should contain a list of all characters that are supported by a font. For example, if a font is always used for uppercase letters then it would save space to define the charset as: "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
`src`	Font file in the case of using a file. Defaults to TrueType font. size floating point size of the font.
`bold`	Defaults to False. Indicates whether the font should be bold.
`trueType`	Defaults to True, relevant only when src is used. If set to False, type 1 fonts are assumed.
`antiAliasing`	Defaults to True. If false, fonts are aliased.
`logicalName`	The logical name of the font as specified by java.awt.Font in Java SE: Dialog, DialogInput, Monospaced, Serif, or SansSerif.
`createBitmap`	Defaults to True. If false no bitmap version of the font is created.

9.1.4 Localization (L10N)

Resource bundles support localization resources, allowing the developer to store key-value pairs within the resource file. The localization bundles use the format of Java property files, which only support USASCII characters. To enter characters in a different script, either use a special editor (such as NetBeans) or use the native2ascii JDK tool with the Ant task to convert the file.

To create a resource bundle use the following code

```
<build dest="src/myresourceFile.res">
   <l10n name="localize">
      <locale name="en" file="l10n/localize.properties" />
       <locale name="iw" file="l10n/localize_iw_IL.properties" />
   </l10n>
</build>
```

To load the localization resource use the following syntax:

```
Hashtable h = bundle.getL10N("localize", "en");
```

The hashtable contains the key value pairs of the resources within the bundle allowing for easy localization. LWUIT supports automatic localization through the `UIManager.setResourceBundle(Hashtable)` method. This installs a global resource bundle which is "checked" whenever a localizable resource is created, thus allowing for the replacement of the entire UI language without querying the resource bundle manually.

9.1.5 Themes

This section discusses how themes work as resources. See Chapter 7 and Chapter 8 to both of these chapters in-depth discussions of styles and theming in LWUIT.

A theme can be defined using a key value properties file containing selectors and values. A selector can be defined as an attribute value, optionally with a component name prepended to it for narrowing the selection further.

The value of an entry in the theme depends on the type of the entry, some entries such as bgImage expect an image object and some entries such as Font expect a font definition. Most entries just expect numbers. For example, this is a typical snippet from a theme:

```
sel#fgColor= 0017ff
font= systemSmall
Form.bgImage=myBackground
Form.font=Serif
SoftButton.bgColor= ff
SoftButton.fgColor= ffffff
```

To add this theme into a resource, add the following:

```
<build dest="src/myresourceFile .res">
   <font logicalName="Serif" bold="true" />
   <font createBitmap="false" name="systemSmall"
         system="FACE_SYSTEM ; STYLE_PLAIN; SIZE_SMALL" />
   <image file="images/background.png" name="myBackground" />
   <theme file="themes/myTheme.conf" name="myTheme" />
</build>
```

This theme can then be installed as follows:

```
UIManager.getInstance().setThemeProps(res.getTheme(myTheme));
```

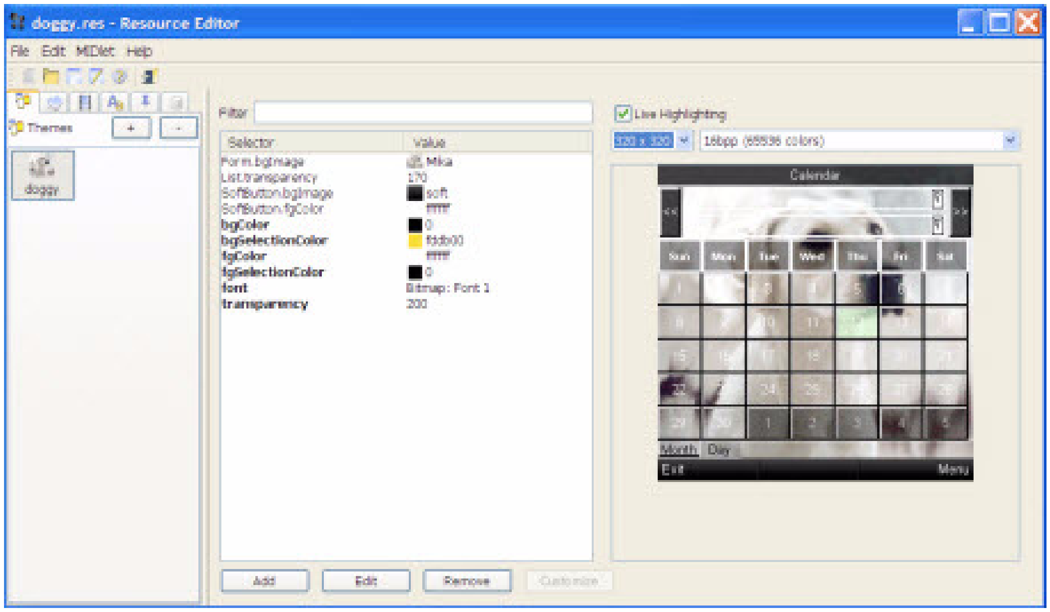
9.2 The LWUIT Resource Editor

The Resource Editor is a standalone GUI tool that allows UI experts, developers, and translators to open, create, and edit resource packages for LWUIT. The Resource Editor was designed for visual work and provides "live" preview of all UI changes, enabling rapid UI customization.

The resource editor and the Ant tasks accomplish similar things, with some limitations in the Ant task feature set. The Ant task is designed for features that make more sense as developer tasks, while the Resource Editor is a tool aimed at designers doing visual work.

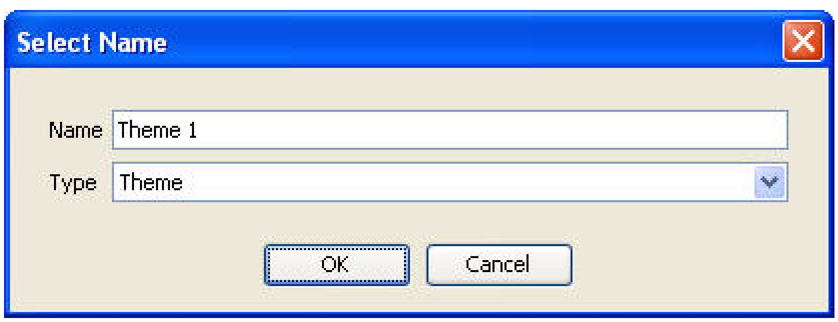
The Resource Editor supports the resource types described in Section 9.1, "Resource Elements".

Figure 9–1 Editing the Default LWUIT Look and Feel



To use the tool, launch the Resource Editor application from your LWUIT distribution.

- Use File > Open to load an existing resource (`.res`) file.
- To add a resource, click the + button in the tab representing the element type you wish to add and specify a name for the resource. Specify a name for the resource. The new resource is added under the appropriate tab.



- To create a new theme, select the Theme node, then click the + button. Note that a resource bundle can contain more than one theme.

> **Note:** The live preview is displayed for themes only and represents the behavior of the theme alone. It doesn't contain the other resources in the file that do not relate to the theme.

9.2.1 Images and Animations

The LWUIT Resource Editor supports the following image types:

- RGB: Standard JPG/PNG formats. Indexed PNGs also work very well (and are highly recommended) with this image type.
- SVG: SVG Tiny images are supported. LWUIT can optionally seamlessly generate PNG images for the SVG files when a device doesn't support SVG.

- Multi-Image: One can add several images based on the DPI of the device (one of several predefined family ranges). Irrelevant images are skipped when loading the resource file .

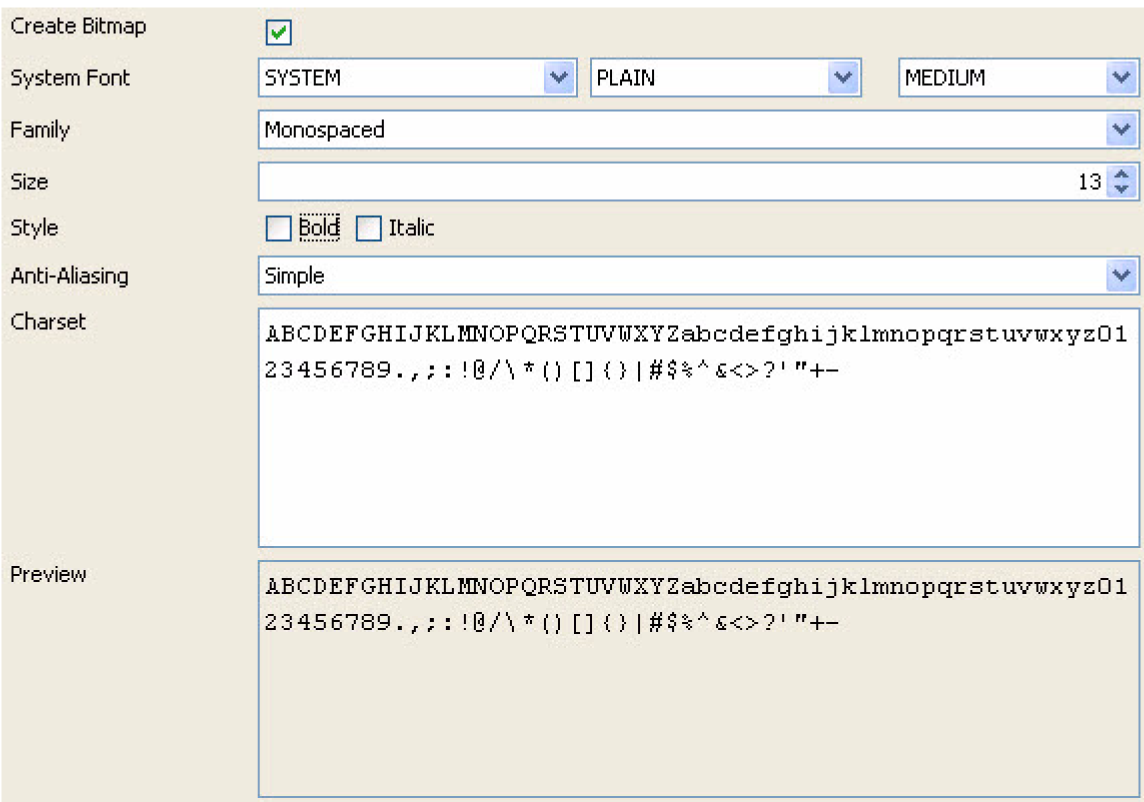
  Multi-images are ideal for icons or small artifacts that are hard to scale properly. They are not meant to replace things such as 9-image borders and so forth since adapting them to every resolution or to device rotation isn't practical.

- Timeline: A timeline is a set of images that can be moved rotated, scaled, and blended to provide interesting animation effects.Timelines allow rudimentary animation and enable GIF importing using the resource editor.

9.2.2 Fonts

The Resource Editor can use device specific fonts or create bitmap fonts for the devices from any font installed in your desktop operating system. Figure 9–2 shows the font editing dialog that appears when adding a new font to the resource file.

Figure 9–2 Font Editing View



Note: Using the Resource Editor does not grant you permission to use the fonts commercially in any way. Licensing the right to use a particular font within a mobile application is strictly your responsibility!

Make sure to specify the characters you need from the font (defaults to upper and lower case English with numbers and symbols). Notice that the more characters you pick in the character set, the more RAM the font will consume on the device. Anti-aliasing is built in to the bitmap font. When running under Java 5 the Resource

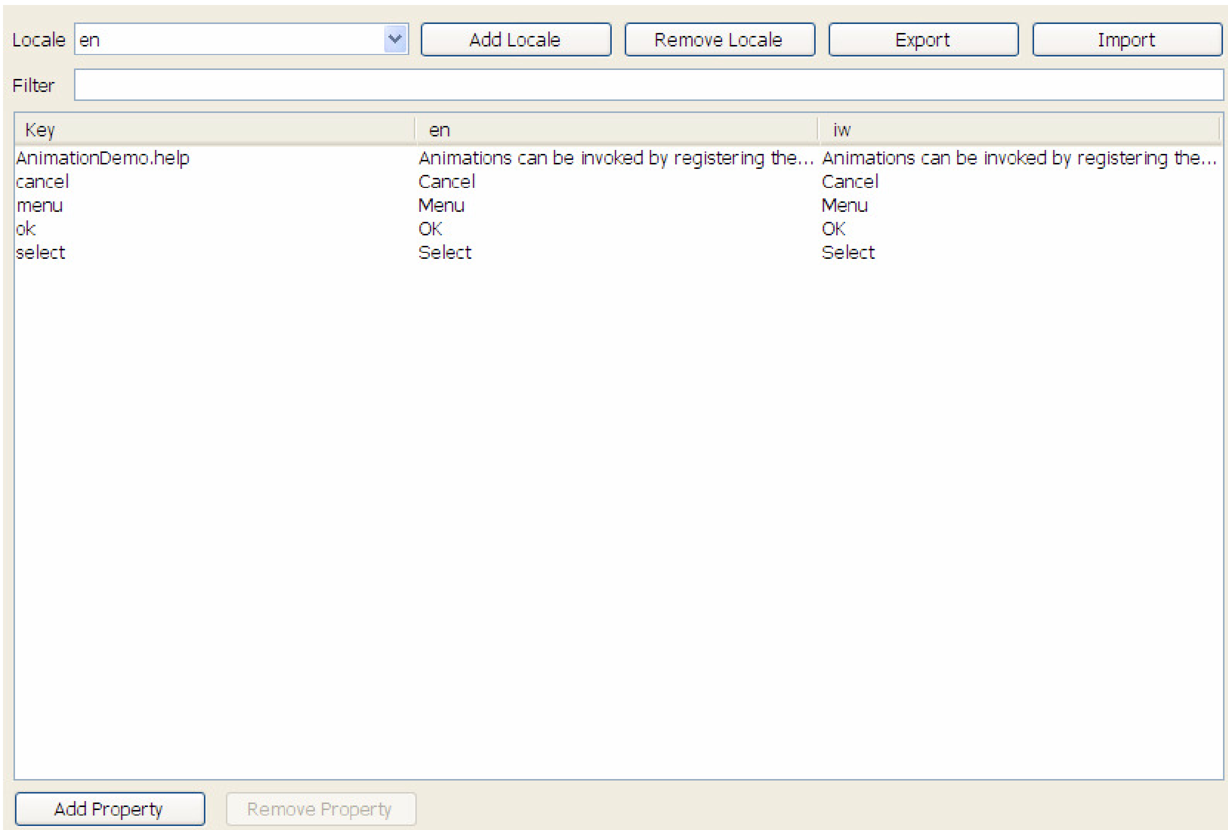
Editor has two anti-aliasing options: Off indicates no anti-aliasing in the bitmap font, and Simple indicates standard anti-aliasing.

9.2.3 Localization

A localization resource can be edited by assigning key/value pairs to use within the application. A key can be mapped to a resource name in any locale.

The editor allows you to add or remove locales listed in the combo box above and appropriately edit the locale entries in the table below. To add or remove a locale property use the buttons on the bottom of the screen.

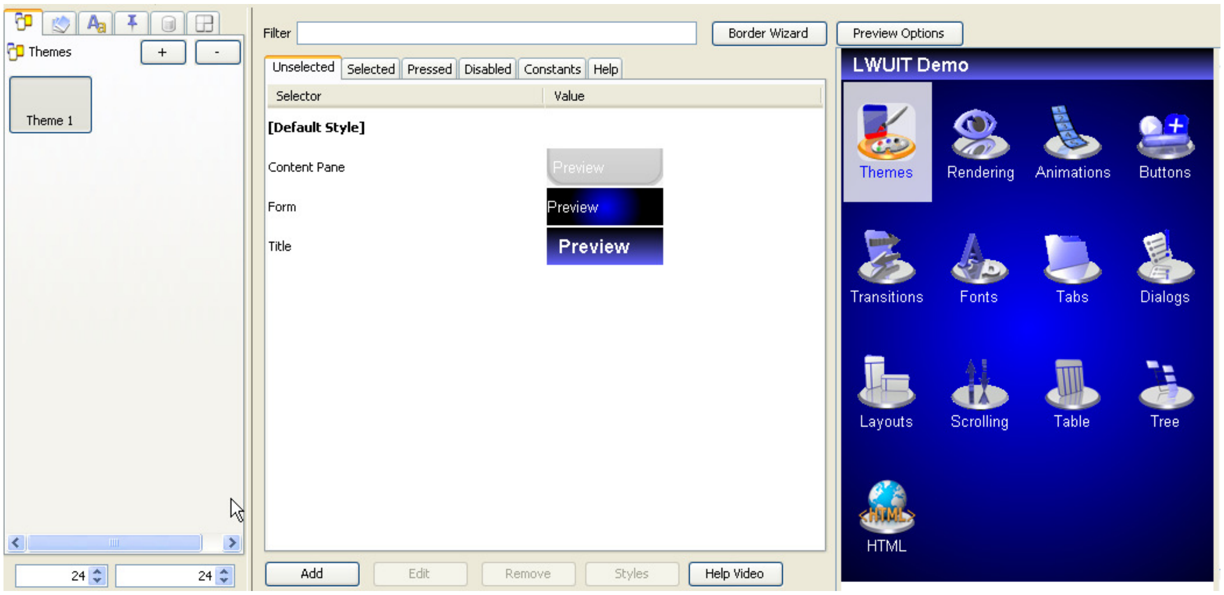
Figure 9–3 Localization and Internationalization View



9.2.4 Themes

To modify a theme resource, set the selectors and the theme resources to appropriate values to produce an attractive UI. When creating a new theme you see a UI containing the table of selectors and resources (for more in depth details of themes for developers, see Chapter 8).

Figure 9–4 Blank Theme View Without Any Styles



To modify the theme, choose a selector on the left side and click the Edit button. You can add new selectors using the Add button in the theme. To modify an existing selector, select it in the table and double click or press the Edit button.

9.2.4.1 Example: Adding a New Theme

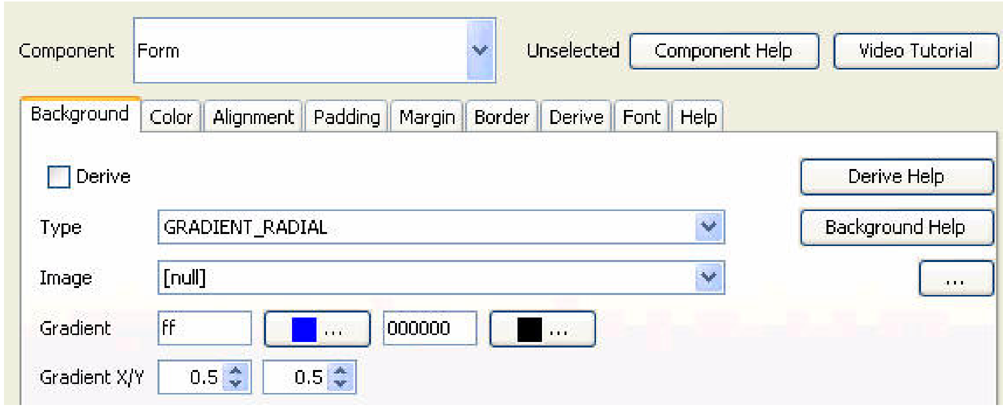
This section describes how to add a new theme using the Resource Editor.

1. Select the theme tab on the left, and click the + button to open the New Theme window. From the Template combo box, select Blank.
2. The UI area on the right shows the Unselected tab by default. Click the Add button at the bottom of the UI area.

The Add window opens.

Select Form from the Component combo box at the top of the Add window.

- In the Background tab uncheck Derive. From the Type dropdown, choose GRADIENT_RADIAL.
- Type ff into the left gradient color (to select a blue to black radial gradient).
- Type 2 into the gradient Size spinner to double the size of the gradient.



Click OK to save your changes.

3. Click the Add button again and select Title from the Component combo box.

   - Select the Background tab.

     Uncheck Derive.

     In the Type combo box select GRADIENT_LINEAR_VERTICAL.

     In the Gradient input field on the right, type 6363ff to select a light blue color.

   - Select the Color Tab.

     Uncheck the Derive Foreground checkbox

     Type ffffff into theForeground field.

   - Select the Alignment tab.

     Uncheck the Derive checkbox and make sure Left is selected in the Alignment combo box.

   - Select the Padding tab.

     Uncheck the Derive checkbox

     Type 10 in the Left and Right fields, and type 5 in the Top and Bottom fields.

   - Select the Font tab.

     Uncheck the Derive Font check box.

     Select BOLD in the center combo box and LARGE in the combo box on the right.

   These steps will create a left-aligned spaced large title that uses white bold text on a vertical gradient background.

4. In the UI area Unselected tab, double-click the **[Default Style]** selector to open the Edit window.

   - Select the Color tab.
   - Uncheck Derive Foreground and type in ffffff to make the default text color white.
   - Uncheck Derive Transparency and type in 0 to make components transparent by default.

5. In the UI area choose the Selected tab and double-click the **[Default Style]** entry.

   a. Select the Color tab.

- Uncheck Derive Foreground.
- Type in ff to make the foreground blue
- Uncheck Derive Transparency

  Type in 200 to make the selection color blend nicely. Click OK.

6. Click the Border Wizard button on the top of the UI area.

   Make these changed on the Create Image tab:

   - In the Width and Height spinners, type 200.
   - In the Thickness spinner, type 4.
   - In the Arc Width and Arc Height fields, type 30.
   - In Color A and Color D fields, type in ffffff.

     In Color B and Color C fields, type in 9f9f9f.

     This creates a reverse border gradient which strengthens a sense of depth.
   - In the opacity spinner type 130 to make the image translucent.

   Select the Cut Image tab.

   - Type 100 in the Top spinner, 90 in the Bottom spinner.
   - Type 25 in the Left and Right spinners.

   Select the Apply To tab.

   - In the Component field, type Content Pane.
   - In the Style combo box, choose Unselected.
   - Click the Add button on the Right. This populates the Applies To area.
   - Click the Generate button at the bottom, and close the wizard.

7. Click the Unselected tab.

   Double-click the ContentPane style entry.

   Select the Padding tab.

   - Uncheck Derive.
   - Input 5 in Top, Bottom, Left and Right spinners.

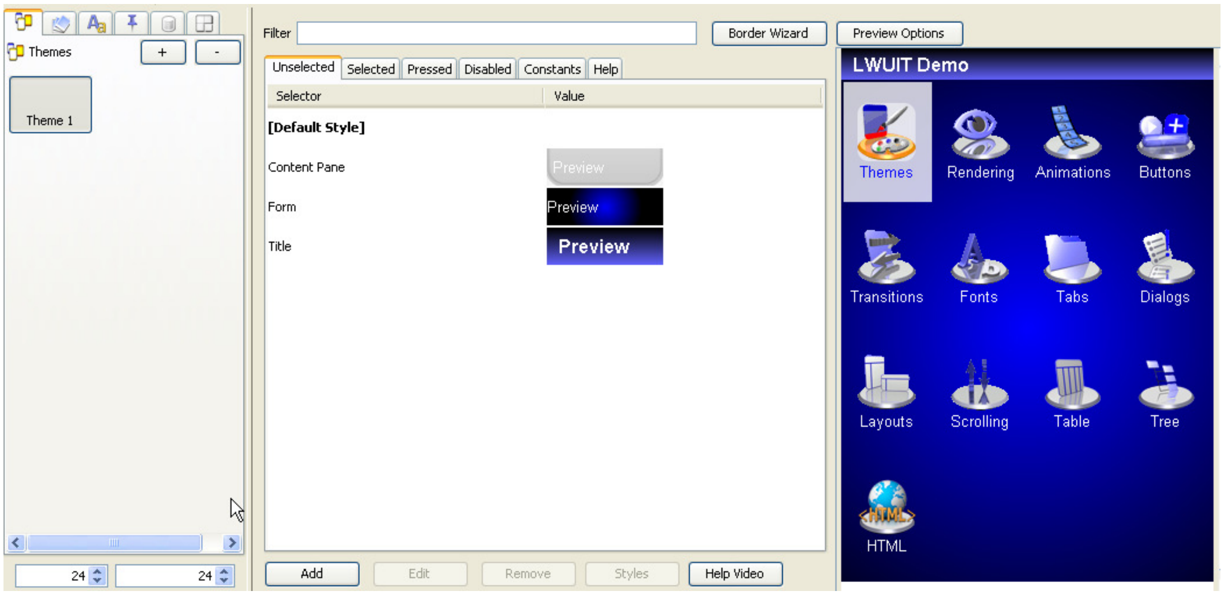
   Select the Margin tab.

   - Uncheck Derive.
   - Input 14 in Top, Bottom, Left and Right spinners.

   This will provide some spacing around the border to accentuate its effect.

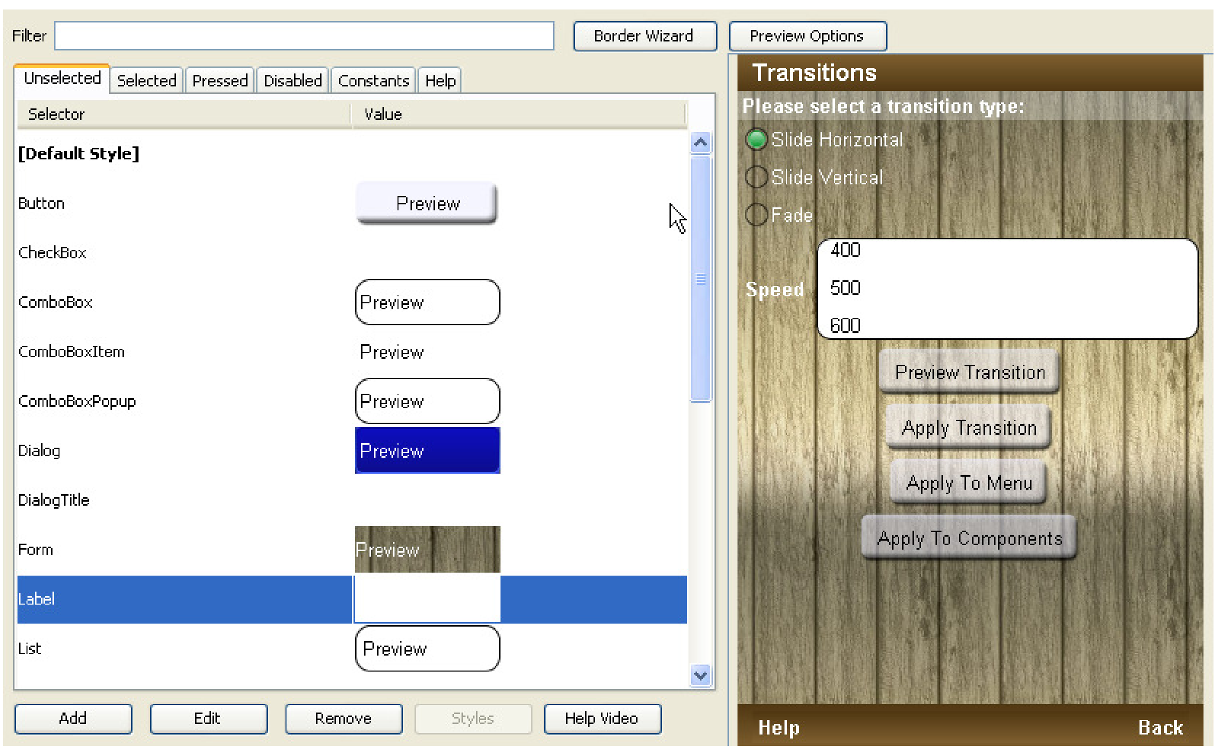
Figure 9–5 shows the final result.

Figure 9–5 New Theme



To gain a deeper understanding of themes add a theme from a template and review its settings. In the Themes tab on the left, click + to add a new theme. From the template combo box, choose Wood and click OK. For example, view the Transitions.

Figure 9–6 Wood Theme Default



You can gain deeper understanding of the selector concepts from Chapter 7 and Chapter 8.

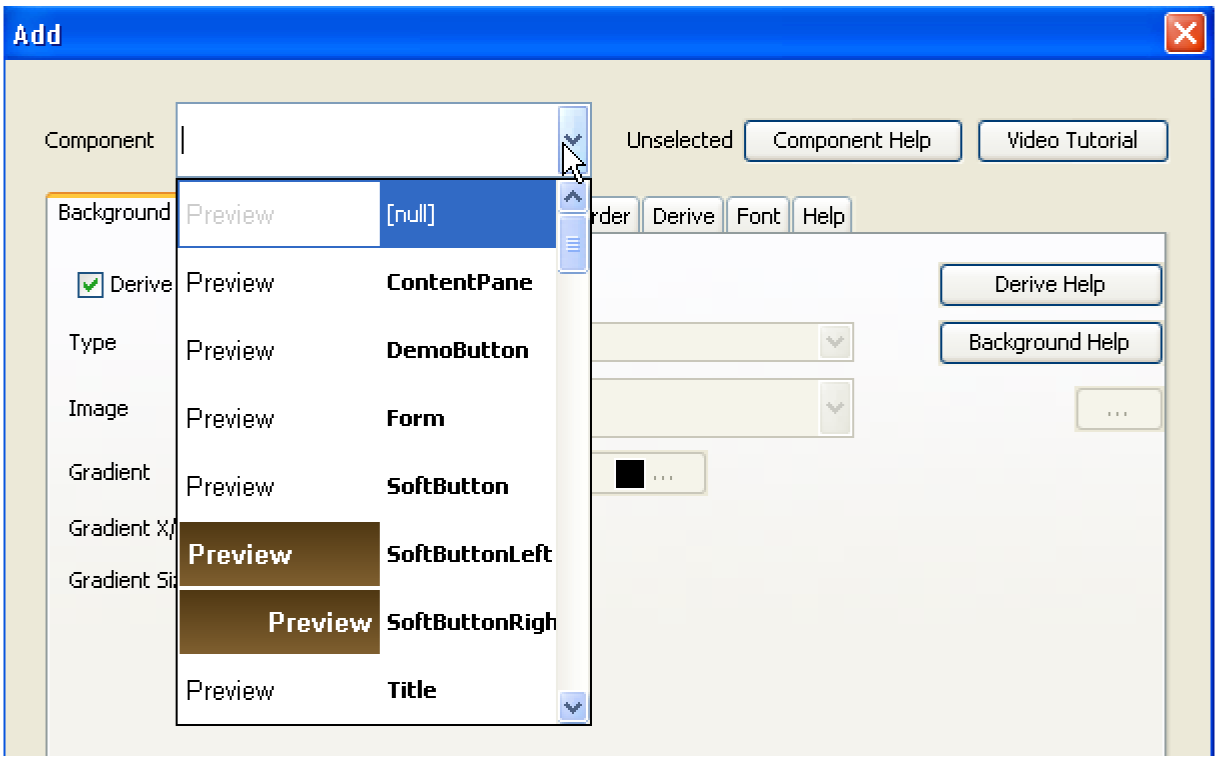
9.2.4.2 Modifying Theme Entries

A theme entry is comprised of a UIID and the attributes modified for the specific UIID. The hardest part in building a theme is understanding the component names for the entries within the UI.

When pointing at the UI preview on the right, a tooltip pops up indicating the UIID of the component you are pointing at, however, this might not work as expected for list renderers (who are no longer there) or for components that are underneath other components (for example. a container that has components on top of it).

When adding a new entry (using the Add button at the bottom of the screen) the combo box at the top highlights in bold the UIIDs that are present in the current screen. This allows discovery via trial and error.

Figure 9–7 Component UIIDs



Notice that every attribute within this dialog has a Derive check box associated with it. The default behavior of adding a new theme entry is to derive from the base style. You need to explicitly indicate that you are interested in modifying a specific attribute. This allows the theme to remain efficient by reducing the amount of "noise" within the theme and also allows inheritance to work properly.

LWUIT style inheritance is built in stages:

- LWUIT has constant "sensible defaults" for some component behaviors (for example, Buttons have a border style by default).
- Every style type has the **[Default Style]** global scope where you can define your own defaults for components (although this won't replace LWUIT's builtin defaults such as the button borders).
- Individual styles can use the Derive tab to define explicit inheritance hierarchy from a specific style UIID to reuse definitions made for one component. This is

very useful when you have multiple style types because the Selected, Pressed and Disabled component styles can derive from the Unselected style of a component.

9.2.4.3 Data

Data is generally designed for developers and shouldn't be used by designers.

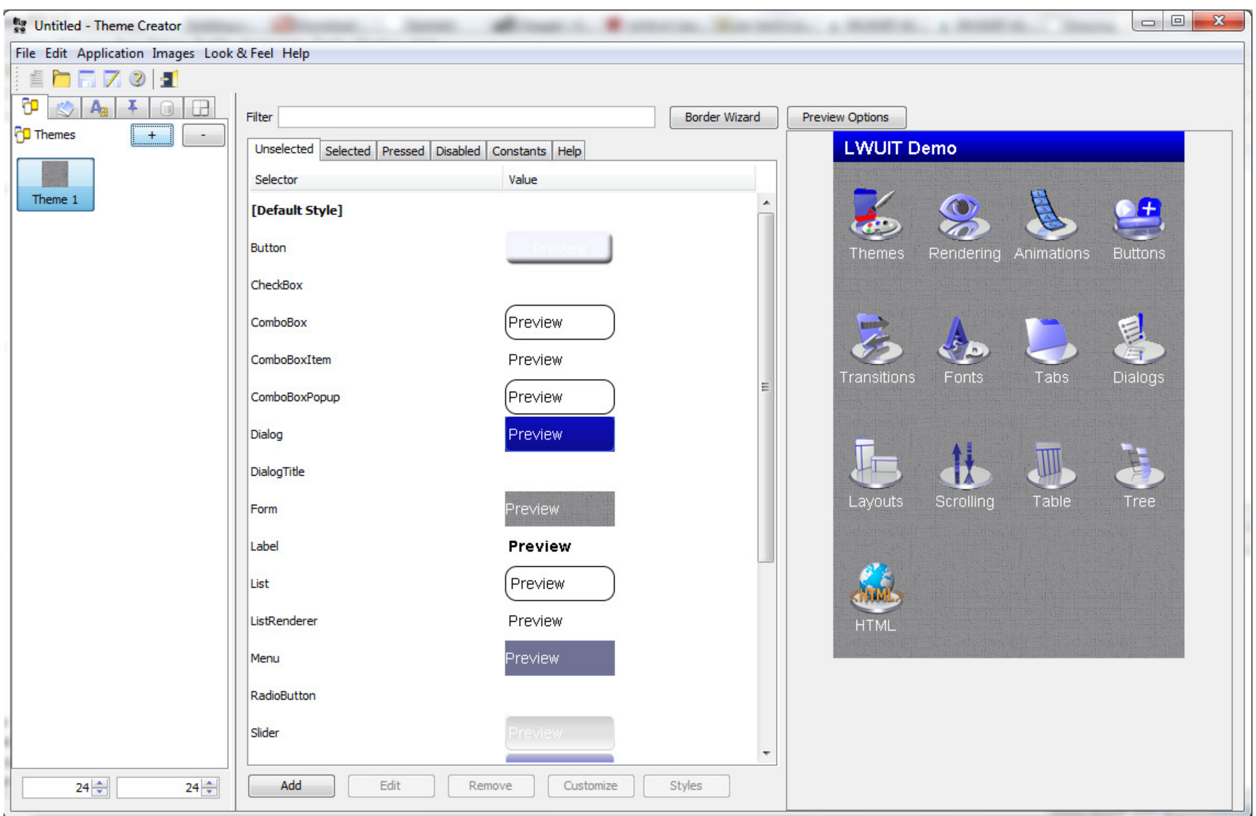
An arbitrary file can be placed within this section and it can be accessed by developers in runtime. This section has no effect on the rest of the functionality even if the data file is an image or font.

9.2.4.4 Customizing the Preview

The preview showing the LWUIT Demo allows for easy customization of a MIDlet. This capability is not limited to the LWUIT Demo. The Resource Editor supports plugging in your own MIDlet so you can test your theme on the fly.

You can install your own MIDlet into the Resource Editor preview panel. In the Resource Editor menu bar, select Application > Pick Application MIDlet and select your MIDlet's JAR file.

Figure 9–8 LWUIT Browser Demo With



There are, however, several restrictions and limitations in this feature:

Since the MIDlet is executed in Java SE it can't leverage `javax.microedition` APIs. While the APIs are present they are implemented in stub form. For example, if you use RMS, GCF, and so forth, they will return null for all queries and do nothing in all operations. Additionally, invoking features such as theming won't work.

If there is a failure in the MIDlet the Resource Editor will silently load the LWUIT Demo in the preview and use it instead. To debug the failure, execute the Resource

Editor from command line using `java -jar ResourceEditor.jar`. When entering the theme option you can see the stack trace of the exception that caused the failure.

It is critical that you DO NOT obfuscate the MIDlet meant for the Resource Editor's preview feature since the Resource Editor must replace the LWUIT instance used within the MIDlet with its own!

Some developers use a preprocessor to create a custom version of their MIDlet for use within this feature. This allows them to avoid functionality that won't work properly in the tool and still give designers the ability to view the entire application for customization.

9.2.4.5 Known Issues

There is currently a known issue in some operating systems which causes the Resource Editor to fail in some cases when using the Aero theme. This issue stems from Java SE's look and feel implementation and the only workaround is to change the application look and feel using the Look And Feel menu option.

Painters

Painter is an interface that can be used to draw on a component background. The Painter draws itself and then the component draws itself on top within the restrictions of the component bounds. One of the biggest advantages of using a painter is that you can write arbitrary code to draw the component background. An example of such code might be a gradient background for a component, or tiling (using an image to tile the component background). Using a generic painter allows you to reuse background painters for various components.

Note: To view the painter drawing, a component must have some level of transparency.

To clarify these points, assume you want to make a painter that draws a diagonal line in the background of a component. This kind of painting is vectoring since you are specifying the absolute coordinates and dimensions of a component. You can reuse the painter for other components.

10.1 Using Painter

The Painter code might look like the following example:

```
Painter diagonalPainter = new Painter() {

   public void paint(Graphics g, Rectangle rect) {
      g.drawLine(rect.getX(),
                 rect.getY(),
                 rect.getX() + rect.getSize().getWidth(),
                 rect.getY() + rect.getSize().getHeight());
   }
};
```

To use the diagonalPainter you created, use it as the component background painter:

```
myComponent.getStyle().setBgPainter(diagonalPainter);
```

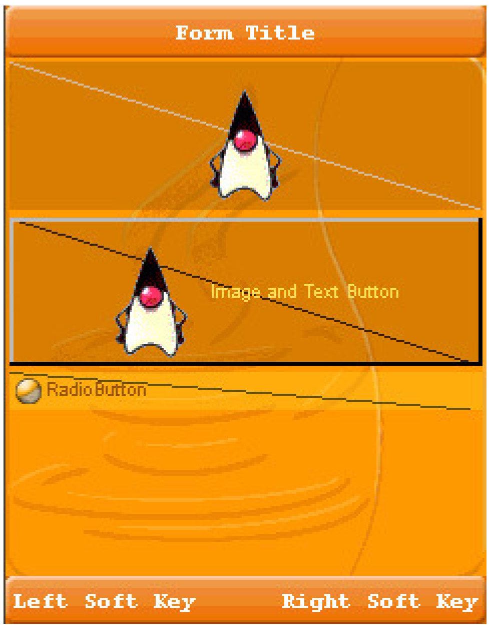
Let's create a Label, Button and a RadioButton and set their background painter with the above diagonalPainter.

```
....

Label myLabel  = new Label(Image.createImage("/images/duke.png"));
myLabel.setAlignment(Component.CENTER);
myLabel.getStyle().setBgTransparency(100);
myLabel.getStyle().setBgPainter(diagonalPainter);

....
Button myButton  = new Button("Image and Text Button");
myButton.setIcon(Image.createImage("/images/duke.png"));
myButton.setAlignment(Component.CENTER);
myButton.getStyle().setBgTransparency(100);
myButton.getStyle().setBgPainter(diagonalPainter);
```

```
....
RadioButton myRadioButton = new RadioButton("RadioButton");
myRadioButton.getStyle().setBgTransparency(100);
myRadioButton.getStyle().setBgPainter(diagonalPainter);

....
```

The three components are shown in Figure 10–1.

Figure 10–1 Label, Button, and RadioButton With diagonalPainter in Background



As a result, you see a diagonal line that is painted in the components' background (behind the Duke images and text).

10.2 Painter Chain

Sometimes a single painter is not enough to represent complex drawing logic necessary for an application's needs. The painter chain allows you to bind together several painters and present them as one. This can be used to separate responsibilities. For example, one painter can draw a background image while another painter can highlight validation errors.

To create a painter chain just use:

```
PainterChain chain = new PainterChain(new Painter[]{painter1, painter2});
```

The painter chain is very useful with the glass pane.

10.3 Glass Pane

The glass pane is a painter that is drawn on top of the form. The form cannot paint over the glass panel! This allows creating very unique visual effects for an application and allows a developer to implement functionality such as validation errors, or special effects such as fading tooltips.A glass pane can be installed using a painter chain to prevent a new glasspane from overriding the already installed glass pane.

To install a glass pane invoke:

```
Painter glassPane = ...;
myForm.setGlassPane(glassPane);
```

Use this code to install a glass pane without overriding an existing glass pane (this method works correctly even if a glass pane is not installed):

```
Painter glassPane = ...;
PainterChain.installGlassPane(myForm, glassPane);
```

LWUIT Implementation

The LWUIT implementation is the foundation of LWUIT and its portability. It is a single huge class representing a hardware abstraction layer (HAL) that contains all the platform-specific code within LWUIT.

WARNING:

The LWUIT implementation is a mechanism for the use of LWUIT developers and "deep hacking." It won't maintain compatibility between versions since it is not generally exposed for developers.

11.1 LWUIT Class

The underlying implementation is often replaced implicitly by using things like the CDC port of LWUIT, which is mostly an implementation class that delegates its calls to the appropriate CDC APIs rather than MIDP's APIs.

Developers should be aware that the LWUIT implementation can be replaced. That is, a developer relying on MIDP API's such as Canvas might run into errors when running on different platforms.

LWUIT ships with an SVGImplementation that can be installed by invoking:

```
SVGImplementationFactory.init();
```

Notice that this method must be invoked before `Display.init()` is invoked! The implementation cannot be replaced at runtime.

The SVGImplementation allows LWUIT to treat SVG image files as if they were standard LWUIT images.

LWUIT also features a VKBImplementation that allows binding a virtual keyboard for touch devices. There are several 3rd-party and LWUIT team implementations mostly designed for porting LWUIT to various platforms.

HTMLComponent

The HTMLComponent class allows rendering of HTML documents that conform to the XHTML Mobile Profile 1.0 (XHTML-MP 1.0) standard.

XHTML-MP 1.0 is a subset of XHTML adapted for mobile. The standard supports most of the basic elements such as Images, Fonts, Lists, Tables, Forms, and even WCSS (a subset of CSS2 for wireless). It does not support Javascript or frames, and it does not support all CSS2 tags or attributes.

This chapter discusses HTMLComponent use cases, interfaces, and implementation details. To learn more about HTMLComponent check out the LWUITBrowser application from the LWUIT SVN repository and examine the code. LWUITBrowser uses most of HTMLComponent's capabilities.

12.1 HTMLComponent Use Cases

HTMLComponent can be used to render local or remote documents. It extends Container and as such it can be added to any Form.

HTMLComponent uses an internal parser to parse the given HTML documents. The parser is not 100% strict and can tolerate some errors in the document, however, some errors may be too fatal for the parser. It is very important to stick to the XHTML-MP1 standard. You must close all open tags in the correct hierarchical order.

12.1.1 Rendering Rich Text

The most simple use case of HTMLComponent is rendering rich text:

```
HTMLComponent htmlC = new HTMLComponent(null);
htmlC.setBodyText("Hello <b>bold text</b>");
```

The only parameter the constructor expects is a class implementing the DocumentRequestHandler interface. This interface defines how links and external resources (such as images, CSS files) in the document are fetched.

Since the example does not use links, we can specify null instead of the document handler. In this case, if links or external resources are specified in the document body they are disabled or ignored.

setBodyText accepts a string containing any text with XHTML-MP 1.0 tags. The text is wrapped with the HTML and BODY tags and passed on for parsing.

If the text is encoded, you can specify the encoding as follows:

```
setBodyText(String htmlText,String encoding)
```

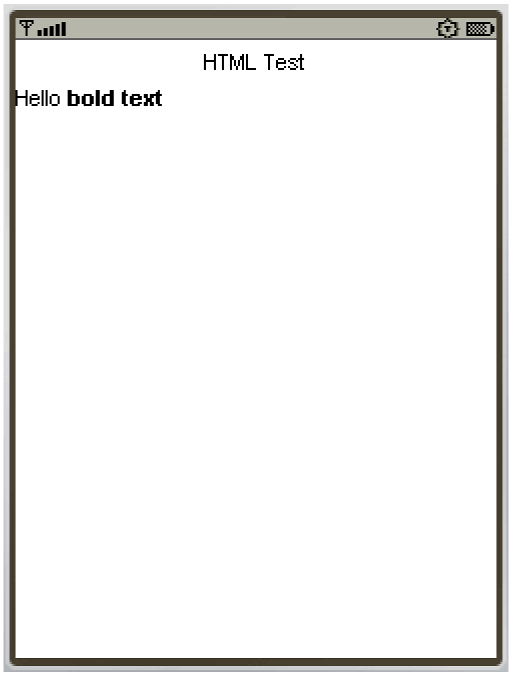
If you have a full HTML file and not just the body text, the following can be used:

```
setHTML(String htmlText,String encoding,String title,boolean isFullHTML)
```

To make the HTMLComponent visible add it to a form and display that form. For example:

```
Form form = new Form("HTML Test");
form.setLayout(new BorderLayout());
form.addComponent(BorderLayout.CENTER,htmlC);
form.show();
```

Figure 12–1 Rich Text Rendered Using HTMLComponent



12.1.2 Reading HTML and Enabling External Resources

The most common use case for HTMLComponent is reading HTML files from either a local or remote source, while enabling external resources such as images and CSS files, and allowing the user to follow links.

To support this use case you must first implement a `DocumentRequestHandler` interface that contains a single method:

```
InputStream resourceRequested(DocumentInfo docInfo)
```

This method is called by `HTMLComponent` (and other internal classes in the html package) to obtain the `InputStream` of the specified document. Requested documents are HTML files (followed links), referenced CSS files, and referenced images.

The requested document information is stored in a `DocumentInfo` object, which is populated automatically by `HTMLComponent`. The `DocumentInfo` values can be used to determine the document's path, file name, type, etcetera.

This example does not implement a `DocumentRequestHandler`. It uses the `HttpRequestHandler` (a ready-made implementation that can be found in the `LWUITBrowser` application) instead. LWUITBrowser be checked out from the LWUIT SVN under `MIDP/applications`.

HttpRequestHandler implementation supports fetching HTML documents via both HTTP and from a JAR file. It supports cookies, encoding, error handling and caching via the Storage class (also available in LWUITBrowser).

12.2 HTMLCallback

During the lifecycle of HTMLComponent there are many events that the developer can respond to. Developers should implement the HTMLCallback interface and set it to the HTMLComponent.

The html package provides a default implementation of the HTMLCallback named DefaultHTMLCallback. This implementation doesn't do too much, but it does demonstrate how to implement the interface methods without harming HTMLComponent tasks (as there are several potential pitfalls). The methods are parsingError, pageStatusChanged, titleUpdated, linkClicked, getLinkProperties and Auto Complete.

12.2.1 parsingError

This method is called whenever the internal parser encounters an error during the document's parsing. This can occur while processing the main HTML document or its referenced CSS files.

You must return a boolean value denoting whether to continue the document processing despite the error (true) or to stop processing (false).

Detailed information on the error can be found in the parameters the method passes, especially errorId which holds the error code (one of the ERROR_* constants).

12.2.2 pageStatusChanged

This method notifies detects changes in the page loading lifecycle. pageStatusChanged can help you display status information to the user or to delay to certain statuses for certain flows.

A new HTMLComponent starts as STATUS_NONE. Shortly after a page URL is set it becomes STATUS_REQUESTED. After a successful connection to the input stream it changes to STATUS_CONNECTED. When the page is displayed (and this can be before images have been completely loaded) the status changes to STATUS_DISPLAYED and finally after all resources have been fully loaded the status becomes STATUS_COMPLETED.

If an error is encountered during the page loading, for example an unrecoverable parsing error, then the status is STATUS_ERROR. If the page loading was cancelled the status becomes STATUS_CANCELLED.

12.2.3 titleUpdated

A useful event that is called after the document's title has been extracted from the TITLE tag of the HTML document.

12.2.4 linkClicked

Called whenever a link is clicked to allow alternative or additional handling. Usually when a link is clicked the link is simply followed through, but in some cases you might want to take additional actions. For example, some updates to the UI outside the HTMLComponent.

The return value should be true if the regular link processing should proceed, and false if it should not.

12.2.5 getLinkProperties

This method is used to support Visited and Forbidden links.

- **Visited Links**: Most browsers to mark visited links in different colors. HTMLComponent does not have any info on which links have been visited before, but getLinkProperties can help hook it to any implementation that tracks links, returning LINK_VISITED for visited links. (See LWUITBrowser for an example.)
- **Forbidden Links**: Sometimes you may want to disallow the use of some links. A common use case may be restricting the user from accessing links outside a defined domain. Another may be blocking content types that HTMLComponent can not render. When getLinkProperties is called, the implementation can look at the URL and determine whether it returns LINK_REGULAR which enables the link, or LINK_FORBIDDEN which disables it.

12.2.6 Auto Complete

The fieldSubmitted and getAutoComplete methods support an auto complete implementation.

fieldSubmitted is called whenever a field in an HTML form is submitted. In return, the implementation should return the actual field value to send to the form. This can be used to perform some content filtering if needed. When none is needed, the value should be returned as is. However, you get the chance to store the field value along with its name, the form URL etc.

Data collected with fieldSubmitted can be used to populate form fields with getAutoComplete, which is called while constructing forms to obtain values for the various fields. Returning null simply means that users must fill out the form themselves. You can also supply another value that is appropriate for the form's specific field,. For example from a repository of stored values as recorded by fieldSubmitted.

12.3 Fonts

When rendering HTML, the HTMLComponent uses the following font facilities described in the following sections:

- Default Font
- System Fonts in HTMLComponent
- Bitmap Fonts
- Font Tags

12.3.1 Default Font

The default font used is the system font with FACE_SYSTEM, STYLE_PLAIN and SIZE_MEDIUM. This can be changed using the setDefaultFont method that accepts a font key (see Section 12.3.3, "Bitmap Fonts") and the font itself.

12.3.2 System Fonts in HTMLComponent

`HTMLComponent` automatically uses all available system fonts. For example if the `<b>` tag is encountered while rendering text with the default system font, the text in the tag is rendered with a system font with the style `STYLE_BOLD`. Same goes for the `<big>` tag which causes text to be rendered with a system font that has a size of `SIZE_LARGE`.

Note that not all system fonts, faces, styles, and sizes are available on all handsets. In fact it is very rare that a device has the full range of fonts representing all possible combinations of those properties. When `HTMLComponent` attempts to use unavailable fonts they are rendered according to fonts the device actually supports.

12.3.3 Bitmap Fonts

To enable `HTMLComponent` to use bitmap fonts, introduce them with the `addFont` method. This method accepts a String identifying the font (Font Key), and a LWUIT Font object. Usually this would be a bitmap font loaded from one of the resource files.

The font keys is an important concept in `HTMLComponent`. A font key identifies the font properties such as its family, style and size. While these properties are known for system fonts, they are unknown for bitmap fonts – and providing them to `HTMLComponent` allows them to be used correctly while rendering documents.

For example, to add a bold Arial font with a size of 20 pixels, one should use:

```
Font font = Font.getBitmapFont("myarialfont");
addFont("arial.12.bold", font);
```

The format of the font key is the family, size and style(s) delimited with the period sign. Order is irrelevant (i.e. `arial.12.bold` is the same as `12.bold.arial`).

Note that the name of the font may be different in the resource files than the font key (In our example it is called "`myarialfont`"), though it is a good practice to name the font according to the font key.

Let's say that we add the following fonts as well:

```
// fonts = …
// filling the fonts array with fonts from the resource file
setDefaultFont("arial.12", fonts[0]);
// Specifying "plain" as a style is optional
addFont("arial.12.bold.italic", fonts[1]);
addFont("timesnewroman.10", fonts[2]);
addFont("arial.14", fonts[3]);
```

And now we load the following HTML:

```
<html>
   <body>
      Default font
      <Bold font <i>Bold and Italic</i> </b>
      <big> Big font </big>
      <small> Small font </small>
   </body>
</html>
```

By specifying the font keys we allow `HTMLComponent` to know which font to assign when encountering font related tags (and also CSS attributes).

In the example above the words "Default font" are displayed in `arial.12 font`, and the rest of the text is displayed according to the tags. However, the `"Small font"` text is displayed in the default font, because even though there is a smaller font (`"timesnewroman.10"`) it is not of the same family. The font matching algorithm gives more weight to the family than the size, and in fact is configured to match only fonts from the same family. Font matching is done sometimes under less then ideal scenarios. While HTML documents may be rich in fonts, the mobile client can offer a limited number of system and bitmap fonts. You should try to match the content with the available fonts in the application.

Also note that system fonts are always matched with other system fonts and bitmap fonts only with other bitmap fonts.

12.3.4 Font Tags

HTML defines several tags that cause (among other thing) a font change when rendered. The font selected to render these tags can be defined in a similar way to adding bitmap fonts. All you need to do is add the desired tag name to the font key. For example:

```
addFont("arial.20.bold.h1", myheaderfont);
```

Now text inside the <h1> tag is rendered with the specified font. Note that the font is added to the font pool and can be also used, for example, when the component seeks a matching bold and big font. Technically, you can prevent the component from using this font by adding it with a font key of just `"h1"`, but of course this is not recommended.The tags that have associated fonts are: `H1`, `H2`, `H3`, `H4`, `H5`, `H6`, `EM`, `STRONG`, `DFN`, `CODE`, `SAMP`, `KBD`, `VAR`, `CITE` and `PRE`.

By default these tags are assigned with the following system fonts:

- `EM`, `DFN`, `VAR`, `CITE`: system, italic, medium
- `CODE`, `SMAP`, `KBD`: monospace, plain, medium
- `STRONG`, `H3`: system, bold, medium
- `H1`:system, bold, large
- `H2`:system, italic, large

Note that while usually there is no reason to add a system font (as they are all automatically used), there is a use case for defining a tag-related font as follows:

```
Font sysFont=Font.createSystemFont(Font.FACE_SYSTEM,
     Font.STYLE_BOLD, Font.SIZE_SMALL);
addFont("h4", sysFont);
```

Note that here it is totally unnecessary to provide any other font properties in the font key because system fonts are supported without explicit addition. But denoting the font key as `"h4"` makes the component render text inside the H4 tag with the specified font.

Small-caps Font

One special case worth noting is small-caps fonts. In CSS one can define the `font-variant` property to the `small-caps` value. In this case the text should be displayed all in caps, with large capital letters depicting “regular” capital letters, and small caps depicting regular text.

System and bitmap fonts do not have this effect. If you have system or bitmap fonts in the documents the application renders, add the font to a resource file. In the resource editor, select a font that behaves like a small caps font) and it should be named as a "small-caps" family:

```
addFont("small-caps.14.bold", smallcapsFont);
```

If no small-caps fonts are added, the `font-variant: small-caps` CSS directive is ignored if encountered.

12.4 Styles in HTMLComponent

HTMLComponent renders most of the HTML tags as regular LWUIT components and as such uses the defined styles for these components. For example, form buttons render as LWUIT's Button, and as such any style that is applied in the theme to Button is expressed in buttons inside the HTML document.

There are however some custom components with the following UIIDs:

- **HTMLLink**: Used for links in the document.
- **HTMLHR**: Used for the HTML `hr` tag (Horizontal separator)
- **HTMLFieldSet**: Used to render the HTML `fieldset` tag
- **HTMLOptgroup**: Used to render the title of an option group inside a ComboBox (option groups are defined by the `optgroup` html tag)
- **HTMLOptgroupItem**: Used to render a ComboBox single item that is a part of an option group.
- **HTMLMultiComboBoxItem**: Used to render an item in a multiple choice ComboBox. LWUIT ComboBox

One can define the style of these components in the theme by using the above UIIDs. The `LWUITBrowser` application contains a theme that includes standard definitions for these UIIDs and can be used as a starting point.

Page Styling

Pages rendered with `HTMLComponent` are rendered on an internal container. This means that setting styles to the `HTMLComponent` itself won't necessarily affect the page style.

To change the style of this internal container, one can use the `setPageStyle` method that accepts a Style object.

12.5 Character Entities

Some characters are represented by character entities (which can be compared to Java escape sequences) either because the characters are reserved or because the character matching key in the keyboard.

A character entity is represented either by its Unicode numeric value or by a verbal symbol. For instance the character > (greater than) which is reserved for HTML tags, is represented either by `>` (its Unicode value) or by `>` (gt is the symbol assigned to this character).

HTMLComponent translates any numeric value and display the according character (of course depending on its availability in the font used). As for symbols, it supports all the standard ISO 8859-1 symbols (up until Unicode value of 255) and does not recognize symbols with Unicode value greater than 255 except 2 very common symbols - `euro` and `bull` (bullet).

If you need support for upper symbols, they can be added using the static methods `addCharEntity` and `addCharEntitiesRange`. For example:

```
HTMLComponent.addCharEntity("spades",9824);
```

12.6 HTMLComponent Settings

There are various settings you can control (or relay to the user's control) with `HTMLComponent`:

- **Image loading**: Can be turned on/off using `setShowImages(boolean)`. The default is true (showing images). When this is set to false, referenced images are not loaded nor are they displayed.
- **CSS loading**: Can be turned on/off using `setIgnoreCSS(boolean)`. The default is false (CSS are loaded). When this is set to true, all CSS directives are ignored including inline CSS, embedded CSS and external CSS files.
- **CSS media types**: CSS references can specify which media types they are suitable for. For example an HTML document can have 2 separate CSS files, one for use with the "handheld" media type and the other with the "screen" media type. By default `HTMLComponent` accepts CSS files and segments that are defined as "handheld" or "all" (or if the media type is unspecified). To modify the supported media types one can use the `setCSSSupportedMediaTypes` method.
- **Max Threads**: The number of threads used by `HTMLComponent` to load external referenced images and CSS files can be set with `setMaxThreads(int)`. The default is 2.

12.7 CSS Support

`HTMLComponent` supports WCSS which is a subset of CSS 2.0. It supports inline CSS directives, embedded CSS segments, and external CSS files. Following are the supported attributes in `HTMLComponent`:

Fully supported CSS properties:

Background	background-color, background-image, background-repeat, background-attachment, background-position-x, background-position-y
Border	border-*-width, border-*-style, border-*-color
Fonts	font-family, font-size, font-style, font-weight, font-variant
Lists	list-style-image, list-style-position, list-style-type
Margins	margin-*, padding-*
Text	text-align, text-indent, text-transform
Misc	color, height, width, visibility
WAP	-wap-access-key, -wap-input-format, -wap-input-required
Shorthand properties	All shorthand properties are fully supported
	* represents `top`, `left`, `bottom`, `right`

Partially supported properties:

display	Supported: none, marquee Unsupported: block, inline , list-item
white-space	Supported: normal, nowrap Unsupported: pre
vertical-align	Works only within tables

Unsupported properties:

`clear`, `float`

Known issues:

- `width` or `height` work for simple elements, but may be problematic with complex elements (for example tables).
- `font-family` accepts the first mentioned font and ignores all fallback fonts, since finding a matching font is very time consuming, and also since in the ME environment usually there aren't that many fonts anyway.
- `text-decoration` is irrelevant: since the only mandatory WCSS decoration value is 'none' which is usually used to remove underlines from links - since we don't have underlines it has no meaning.
- text-transform may have issues when overriding a parent which has a different transform.
- Some properties are ignored if associated with a pseudo-class (such as `a:focus` or `hover`) - and that's because while LWUIT does have separate styles for selected, unselected and pressed states - these styles include properties such as padding, margins, colors, background, font - but for example not alignment or visibility which affect the component in all of its states.

12.8 Implementing a DocumentRequestHandler

In the first example we have used a ready-made `DocumentRequestHandler` implementation. In this section we will create our own simple implementation that reads from files stored in the JAR.

Our implementation will accept URLs with the `file://` protocol only, and fetch them from the JAR:

```
import com.sun.lwuit.html.DocumentInfo;
import com.sun.lwuit.html.DocumentRequestHandler;
import java.io.ByteArrayInputStream;
import java.io.InputStream;

class FileRequestHandler implements DocumentRequestHandler {

   public InputStream resourceRequested(DocumentInfo docInfo) {
      // Get the full URL from the docInfo
      String url=docInfo.getUrl();

      if (!url.startsWith("file://")) { // We support only files
         return getErrorStream("This handler handles files only.")
      }

      if (docInfo.isPostRequest()) { // We don't support POST
         return getErrorStream("GET requests only please!");
      }
      url=url.substring(7); // Cut the file://
      return getClass().getResourceAsStream(url);
      }
   // Utility method to get a stream out of a string
   private InputStream getErrorStream(String err) {
      err="<html><body>"+err+"</body></html>";
      ByteArrayInputStream bais =
         new ByteArrayInputStream(err.getBytes());
      return bais;
   }
}
```

As we can see the implementation is quite simple. It uses the `getResourceAsStream` method to obtain an `InputStream` of the file and send it over, but before that it queries the passed `DocumentInfo` object to get some information on the requested page. This object is explained in detail in the next section.

12.9 DocumentInfo

The `DocumentInfo` is an object that is passed from the `HTMLComponent` to the `DocumentRequestHandler`, and can be used by the latter to obtain information about the document such as its location, type, encoding etcetera, and also to hint back to the `HTMLComponent` about attributes it found about the document.

When a `setPage` is called on an HTMLComponent, it results in a call to the `DocumentRequestHandler`'s `resourceRequested` method, with a populated `DocumentInfo` object. This method is also called when links are clicked or referenced images and CSS files are needed. The remainder of this section discusses some useful `DocumentInfo` getters/setters that a `DocumentRequestHandler` implementation should consider:

- `getUrl`
- `getEncoding` and `setEncoding`
- `getParams`
- `getExpectedContentType` and `setExpectedContentType`
- `getFullUrl` or `getBaseUrl`

12.9.1 `getUrl`

This method returns the absolute URL of the requested document. The absolute URL is automatically calculated internally according to the page on which the link was clicked on. Implementations can learn about the document protocol (file, http etc.) and about the document's domain and act accordingly. For example, it is possible to allow only certain protocols or domains, or to use custom protocol strings etc.

12.9.2 `getEncoding` and `setEncoding`

`getEncoding` and `setEncoding` are quite important when reading documents that can have different encodings.Encoding information of HTML and CSS documents can appear in multiple places. For example when posting a form, its `FORM` tag can have an ENCTYPE property that specifies the form's encoding. This is one situation in which the encoding in the provided `DocumentInfo` is different than the default (which is ISO-8859-1), and thus has to be queried to set encoding headers appropriately. On the other direction, when requesting a document, the encoding can be specified by the response headers (`charset` in the content-type header) – and then in order for `HTMLComponent` to be able to read the document properly, the encoding type must be set using `setEncoding`. Note that encoding can be set in other ways as well such as BOM (Byte Order Mark), and it is the responsibility of the `DocumentRequestHandler` to figure it out and relay that info to `HTMLComponent` via the `DocumentInfo` object.

12.9.3 `getParams`

`getParams` returns the request parameters. It can be used for example to screen parameters before sending to the server (And it has a matching setter as well)

12.9.4 `getExpectedContentType` and `setExpectedContentType`

The expected content type is what the `HTMLComponent` expects to find when requesting the resource in question. This would be an HTML document (`TYPE_HTML`) when setting a page or clicking links, an image (TYPE_IMAGE) for image references and a CSS file (TYPE_CSS) for CSS references. Queering the expected content type can help processing, for example we will check encoding only for HTML and CSS, but not for images. Another reason may be that we want to cache images and not HTML documents and so on.

12.9.5 `getFullUrl` or `getBaseUrl`

Other more informative methods include `getFullUrl` returning a string composed of the absolute URL plus the parameters of the request (if any, and only if this was a GET request). Another one is `getBaseUrl` returning the document base URL.

Using Transitions and Animations

The Lightweight UI Toolkit library implements transitions using animation.

13.1 Animation

Animation is an interface that allows any object to react to events and draw an animation at a fixed interval. All animation methods are executed on the EDT. For simplicity's sake, all Components can be animated, however, no animation appears unless it is explicitly registered into the parent form. To stop animation callbacks the animation must be explicitly removed from the form (notice that this differs from removing the component from the form)! In Lightweight UI Toolkit there are few transitions that have been extended from Animation. See Section 13.3, "Transition".

13.2 Motion

The Motion class abstracts the idea of motion over time, from one point to another. Motion can be subclassed to implement any motion equation for appropriate physics effects. This class relies on the `System.currentTimeMillis()` method to provide transitions between coordinates. Examples for such movement equations can be; parabolic, spline, or even linear motion. Default implementation provides a simple physical algorithm giving the feel of acceleration and deceleration. In this implementation all animation elements (Transition, Scrolling, and so forth) use the same motion implementation, so they all have smooth movement.

13.3 Transition

Currently a transition refers to the transition between two Forms as animate In and Out transition animation. All transitions use a physical animation curve calculation to simulate acceleration and deceleration while pacing a motion based on the amount of time specified. There are three types of transitions:

`Slide`	Exiting form by sliding out of the screen while the new form slides in.
`Fade`	Components fade into and out of the screen at a predefined speed.

13.3.1 Slide Transition

To create a slide transition, that reacts while exiting the first form, use:

```
CommonTransitions.createSlide(int type, boolean forward, int
speed)
```

`type`	Type can be either `SLIDE_HORIZONTAL` or `SLIDE_VERTICAL`, indicating the movement direction of the forms.
`forward`	Forward is a boolean value representing the directions of switching forms. For example for a horizontal type, true means horizontal movement to the right. For a vertical type, true means movement towards the bottom.
`speed`	Speed is an integer representing the speed of changing components in milliseconds.

For example:

```
// Create a horizontal transition that moves to the right
// and exposes the next form

myForm.setTransitionOutAnimator(CommonTransitions.createSlide(
   CommonTransitions.SLIDE_HORIZONTAL, true, 1000));
```

Figure 13–1 shows four snapshots of the horizontal transition from a menu to a radio button list.

Figure 13–1 Slide Transition from Form to Theme Menu

13.3.2 Fade Transition

Fade transition creates a fade-in effect when switching to the next form. To create this transition use:

```
CommonTransitions.createFade(int speed)
```

In the above code `speed` is an integer representing the speed of changing components, in milliseconds.

For example:

```
// Create a fade effect with speed of 400 millisecond,
// when entering myform

themeForm.setTransitionInAnimator(CommonTransitions.createFade(400)
);
```

Figure 13–2 Fade Transition From Form to Theme Menu

Authoring Components

LWUIT is designed to be as extensible and modular as possible. A developer can replace or extend almost every component within LWUIT (as of this writing none of the LWUIT components are defined as final). In the spirit of Swing, a third-party developer can write an LWUIT component from scratch by implementing painting and event handling.

Furthermore, thanks to the composite pattern used by LWUIT (and Swing with AWT), small custom and preexisting components can be combined to form a single component.

The composite approach is mentioned in Chapter 2. This chapter focuses on writing a component from scratch and plugging it into the LWUIT features such as the theme engine, painters, etcetera. This chapter discusses direct derivation from the Component, but you can derive from any existing LWUIT component to produce similar results. For example, ComboBox derives from List, Button from Label, CheckBox from Button, Dialog from Form, and so forth.

14.1 Painting

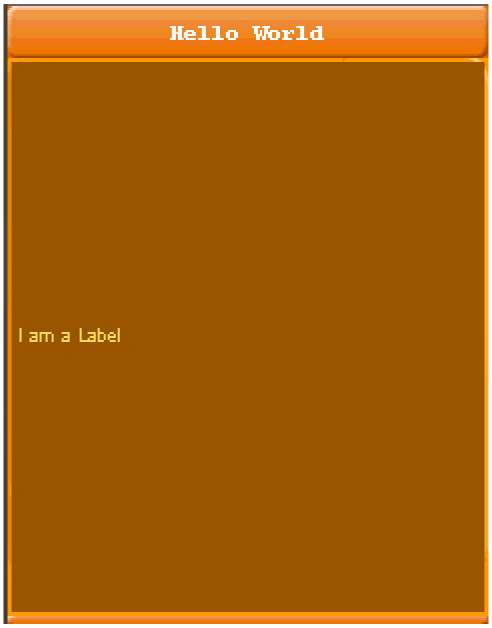
Writing a custom component should be immediately familiar to Swing/AWT developers. The following example derives from Component and overrides paint in order to draw on the screen:

```
public class MyComponent extends Component {
   public void paint(Graphics g) {
         g.setColor(0xffffff);
         g.fillRect(getX(), getY(), getWidth(), getHeight());
         g.setColor(0);
         g.drawString("Hello World", getX(), getY());
   }
}
```

This component writes `Hello World` in black text on a white background. To show it we can use the following code, resulting in Figure 14–1. As mentioned earlier, you can also derive from an appropriate subclass of Component; overriding `paint` is optional.

```
Form testForm = new Form();
testForm.setLayout(new BorderLayout());
testForm.addComponent(BorderLayout.CENTER, new MyComponent());
testForm.show();
```

Figure 14–1 Hello World



Notice several interesting things that might not be obvious in the example:

- Setting the color ignores the alpha component of the color. All colors are presumed to be opaque RGB colors.
- The rectangle is filled and the text is drawn in the X coordinate of the component. Unlike Swing, which "translates" for every component coordinate, LWUIT only translates to the parent container's coordinates, so it is necessary to draw in the right X/Y position (rather than 0,0) because the component position might not be the same as the parent's. For example, to draw a point a the top left of the component, you must draw it from `getX()` and `getY()`.

14.2 Sizing In Layout

In most cases the example above won't work properly because the layout manager doesn't "know" how much space to allocate. To fix this you must define a preferred size.

A preferred size is the size which the component requests from the layout manager. It might take more (or less) but the size should be sufficient to support rendering. The preferred size is calculated based on images used and font sizes used. The component developer (or look and feel author) is responsible for calculating the proper size.

The `calcPreferredSize()` method is invoked when laying out the component initially (and later when changing themes). It allows you to determine the size you want for the component as follows:

```
protected Dimension calcPreferredSize() {
   Font fnt = Font.getDefaultFont();
   int width = fnt.stringWidth("99999-9999")
    int height = fnt.getHeight();
    return new Dimension(width, height);
}
```

Unlike Swing/AWT, LWUIT doesn't have minimum or maximum size methods, thus your job as a component developer is simpler. Components grow based on the layout manager choices rather than component developer choices

This example uses a hardcoded text for sizing rather than the input string, so the component won't constantly resize itself in the layout as the user inputs characters.

After making these changes you no longer need to use the border layout to place the component and it now occupies the correct size, so you can show the component using the following code (default layout if FlowLayout):

```
Form testForm = new Form();
testForm.addComponent(new MyComponent());
testForm.show();
```

14.3 Event Handling

So far the component doesn't have any interactivity or react to user events. To improve the component, we can build a simple input area that accepts only numeric values (for simplicity's sake we do not support cursor navigation).

Event handling in LWUIT is very similar to MIDP event handling (which is designed for small devices) in which we receive the calls from the platform in methods of the subclass. To accept user key presses, override the appropriate key released method as follows:

```
public void keyReleased(int keyCode) {
   if(keyCode >= '0' && keyCode <= '9') {
       char c = (char)keyCode;
 inputString += c;
 repaint();
 }
}
```

Note, it is an LWUIT convention to place actions in the key released event rather than the key press event (except for special cases). This is important from a UI perspective, because navigation triggered by a key press event might send the key release event to a new form, causing odd behavior.

14.4 Focus

If you run the event handing code above, you can see the event never actually occurs. This is because the component must accept focus in order to handle events. By default, components are not focusable and you must activate focus support as follows:

```
setFocusable(true);
```

Once activated, focus works as you would expect and the behavior is correct. It makes sense to detect focus within the `paint(Graphics)` method (or `paintBorder`) and draw the component differently in order to visually indicate to the user that focus has reached the given component.

14.5 The Painting Pipeline

This section discuss painting the component with regard to styles and focus. To understand styling and proper painting process it’s necessary to understand the basics of how painting occurs in LWUIT.

Painting operations are performed in order by the rendering pipeline, and all painting is performed in order on the event dispatch thread (EDT):

1. First the background is painted using the appropriate painter (see the background painters section). This makes sure the background is properly “cleared” to draw.
2. The `paint` method is invoked with the coordinates translated to its parent container.
3. The `paintBorder` method is invoked with the same translation.
4. Both `paint` and `paintBorder` delegate their work to the `LookAndFeel` and `Border` classes respectively to decouple the drawing code. For example, Button's paint method looks something like this:

```
public void paint(Graphics g) {
 UIManager.getInstance().getLookAndFeel().drawButton(g, this);
}
```

Paint border from component defaults to a reasonable value as well:

```
Border b = getBorder();
if(b != null){
  g.setColor(getStyle().getFgColor());
  b.paint(g, this);
}
```

14.6 Styling

In the beginning we painted the component using simple drawing methods, completely disregarding the style. While this is perfectly legal it fails to take advantage of LWUIT's theming functionality.

The “right way” to paint in LWUIT regards the Style object and ideally delegates work to the `LookAndFeel` class. Notice that you can subclass `DefaultLookAndFeel` and add any method you want, such as `paintMyComponent()`. This allows you to implement component painting “correctly” within the look and feel. However, for custom-made components this might not be the best approach since it blocks other third parties from using your components if they have already created a look and feel of their own.

For simplicity, this example does all the painting within the component itself.

To paint the input component correctly, implement the `paint` method as follows:

```
public void paint(Graphics g) {
   UIManager.getInstance().getLookAndFeel().setFG(g, this);
   Style style = getStyle();
   g.drawString(inputString,
     getX() + style.getPadding(LEFT),
     getY() + style.getPadding(TOP));
}
```

There are several things of interest in the code above:

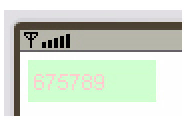
- `setFG` sets the foreground color and font based on the state of the component (enabled, hasFocus).
- Style padding positions the text. Notice it ignores the margins, which are already in the translated coordinates of the paint (margins work without any change in the code).
- There’s no need to paint the background, draw a border or check for focus. These things are all handled implicitly by LWUIT!

This isn't enough though, the implementation of `calcPreferredSize` must take all of these things into account, including the possibility of user installed fonts.

```
protected Dimension calcPreferredSize() {
   Style style = getStyle();
   Font fnt = style.getFont();
   int width = fnt.stringWidth(inputString);
   int height = fnt.getHeight();
   height += style.getPadding(Component.TOP) +
style.getPadding(Component.BOTTOM);
   width += style.getPadding(Component.LEFT) +
style.getPadding(Component.RIGHT);
   return new Dimension(width, height);
}
```

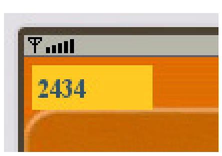
With these two things in order our component is functional and works with the existing theme!

Figure 14–2 Original Component Theme



If we change the theme to the Java theme from the UI demo, the same code produces Figure 14–3.

Figure 14–3 New Theme



However, there is one last thing for styles to work correctly. Currently the component uses the default color scheme and font and doesn't allow the designer to specify a style specific to this component. To allow this functionality you must allow the component to be identified in the theme editor, even in obfuscated code and in case of subclasses. To do this, override `getUIID()` and return the name you want for the component:

```
public String getUIID() {
 return “NumericInput”;
}
```

This allows a designer to specify `NumericInput` within the Resource Editor's theme builder (in the Component combo box) in order to customize this component. Note, currently the Resource Editor doesn't support previews for custom-built components.

14.7 Background

Up until now we've assumed that LWUIT takes care of the background handling for us. However, it is important to understand how this works, otherwise performance might be impacted.

The background of a component is managed by a Painter (see the API documentation for `Painter` for further details). A Painter can draw any arbitrary graphics in the background and can be translucent or opaque. LWUIT creates painters implicitly based on background image or color in the style. Furthermore you can customize them either by creating your own special painter or by manipulating the style.

Since a painter can be translucent or transparent LWUIT recurses to the top-most component, starts drawing its painter, then recurses down the paint hierarchy until the background is properly drawn. If your component is completely opaque (a square that draws all of its data) this extra work is redundant. To improve performance, define background transparency (in the style) to be 255 (0xff). This indicates your background is opaque.

Painters are designed for general pluggability. They work with your customized component without any effort on your part.

14.8 Animating The Component

We briefly discussed the animation framework in Section 13.1, "Animation". However, with a custom component the features are far more powerful.

First you must register the component as interested in animation. You cannot perform this registration during construction since there is no parent form at this stage. The component has an `initComponent` method that is guaranteed to invoke before the component is visible to the user and after the parent form is available.

```
protected void initComponent() {
   getComponentForm().registerAnimated(this);
}
```

The code above registers the animation, essentially triggering the animate method. The animate method can change the state of the component and optionally trigger a repaint when it returns true.

It is relatively easily to implement a “blinking cursor“ using the animate method:

```
private boolean drawCursor = true;
private long time = System.currentTimeMillis();
public boolean animate() {
   boolean ani = super.animate();
   long currentTime = System.currentTimeMillis();
   if(drawCursor) {
      if((currentTime - time) > 800) {
         time = currentTime;
         drawCursor = false;
         return true;
      }
   } else {
```

```
        if((currentTime - time) > 200) {
            time = currentTime;
            drawCursor = true;
            return true;
        }
    }
    return ani;
}
```

Notice that all this code really does is change the `drawCursor` state in which case it returns true, indicating the need for a repaint. Now implementing a cursor within our paint method requires only the following lines:

```
public void paint(Graphics g) {
    UIManager.getInstance().getLookAndFeel().setFG(g, this);
    Style style = getStyle();
    g.drawString(inputString, getX() + style.getPadding(LEFT),
                              getY() + style.getPadding(TOP));
    if(drawCursor) {      int w = style.getFont().stringWidth(inputString);
        int cursorX = getX() + style.getPadding(LEFT) + w;
        int cursorY = getY() + style.getPadding(TOP);
        int cursorY = getY() + style.getPadding(TOP);
    }
}
```

14.9 The Custom Component

Example 14–1 shows the MIDlet Code with a theme.

Example 14–2 shows the component code.

Example 14–1 "Col 1", "Col 2", "Col 3"}, new Object[][] {

```
import java.io.IOException;
import javax.microedition.midlet.MIDlet;
import com.sun.lwuit.Display;
import com.sun.lwuit.Form;
import com.sun.lwuit.plaf.UIManager;
import com.sun.lwuit.util.Resources;

public class LWUITMIDlet extends MIDlet {

    private boolean started;
    protected void startApp() {
        try {
            Display.init(this);
            Resources r1 = Resources.open("/javaTheme.res");
            UIManager.getInstance().setThemeProps(r1.getTheme("javaTheme"));

            // distinguish between start and resume from pause

            if (!started) {
                started = true;
                Form testForm = new Form();
                testForm.addComponent(new MyComponent());
                testForm.show();
            }
```

```
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
    protected void pauseApp() {
    }
    protected void destroyApp(boolean arg0) {
    }
}
```

Example 14–2 Component Code

```
import com.sun.lwuit.Component;
import com.sun.lwuit.Font;
import com.sun.lwuit.Graphics;
import com.sun.lwuit.geom.Dimension;
import com.sun.lwuit.plaf.Style;
import com.sun.lwuit.plaf.UIManager;

public class MyComponent extends Component {

    private boolean drawCursor = true;
    private long time = System.currentTimeMillis();
    private String inputString = "";

    public MyComponent() {
        setFocusable(true);
    }

    public void paint(Graphics g) {

        UIManager.getInstance().getLookAndFeel().setFG(g, this);
        Style style = getStyle();
        g.drawString(inputString, getX() + style.getPadding(LEFT), getY() +
           style.getPadding(TOP));
        if (drawCursor) {
            int w = style.getFont().stringWidth(inputString);
            int cursorX = getX() + style.getPadding(LEFT) + w;
            int cursorY = getY() + style.getPadding(TOP);
            g.drawLine(cursorX, cursorY, cursorX, cursorY +
               style.getFont().getHeight());
}}

    protected Dimension calcPreferredSize() {
        Style style = getStyle();
        Font fnt = style.getFont();
        int width = fnt.stringWidth("99999-9999");
        int height = fnt.getHeight();
        height += style.getPadding(Component.TOP) +
           style.getPadding(Component.BOTTOM);
width += style.getPadding(Component.LEFT) +return new Dimension(width, height);
           style.getPadding(Component.RIGHT);}

    public String getUIID() {
        return "NumericInput";
    }

    public void keyReleased(int keyCode) {

        if (keyCode >= '0' && keyCode <= '9') {
            char c = (char) keyCode;
            inputString += c;
            repaint();
        }
    }

    protected void initComponent() {
        getComponentForm().registerAnimated(this);
    }
```

```
    public boolean animate() {

        boolean ani = super.animate();
        long currentTime = System.currentTimeMillis();
        if (drawCursor) {
            if ((currentTime - time) > 800) {
                time = currentTime;
                drawCursor = false;
                return true;
            }
        } else {
            if ((currentTime - time) > 200) {
                time = currentTime;
                drawCursor = true;
                return true;
            }
        }
        return ani;
    }
}
```

Portability and Performance

While portability is one of LWUIT's best attributes, it is also one of the hardest features to grasp. LWUIT is portable as a library and it also enables application porting in such a way that binary code or source can be compatible across different Java ME profiles.

15.1 Introduction

Much has been said in the past about Java device fragmentation (write once debug everywhere). To understand LWUIT's portability you must first understand the original problems and the solutions LWUIT provides for each problem:

- Low quality or buggy implementations of the specification

  This problem was far more severe with older (prior to CLDC 1.1) devices that LWUIT does not support. Thanks to modern TCKs, the virtual machine (VM) in modern devices is compatible, and furthermore the UI layer on which LWUIT is based is very narrow and relatively robust across devices.

- Low power, low memory devices

  Again with newer CLDC 1.1 devices this is not as much of a problem as it used to be, but there are still concerns. See Chapter 2 for a discussion on increasing performance and reducing memory overhead (sometimes trading off one for the other).

- Varying screen resolutions

  LWUIT ships with a very fast low memory overhead scaling algorithm that doesn't lose transparency information. For extreme cases where the algorithm is not enough, LWUIT supports pluggable themes, allowing the UI to be customized with images more fitting to the resolution of the phone.

- Varying input methods

  LWUIT detects soft buttons automatically, and navigation is already portable. LWUIT supports touch screens seamlessly out of the box. Text input works with the device native text box, ensuring proper input.

- Over-The-Air (OTA) code size limitations

  This problem is solving itself, given relaxed carrier restrictions and increasing JAR file size allocations. LWUIT fully supports obfuscation and works properly with obfuscators that remove redundant code.

- Non-UI related pitfalls (networking issues, RMS incompatibility, etcetera)

  LWUIT currently focuses only on UI related issues, so you must find your own solution for the many minor issues related to these problems. For most common use cases failure occurs because the device expects the "right thing". For example, networking is problematic on some devices due to a connection that was never closed, and so forth.

15.2 Performance

Performance is a huge problem in portability. Problems in performance often creep on an application only to appear later in its life cycle. Performance is often a trade-off, mostly of memory for CPU or visa versa. The easiest way to improve performance is to reduce functionality.

Since LWUIT has pluggable theming you can substitute a simple theme without changing code. This makes it easier to see whether the problem is in the UI itself.

The following subsections discuss the specifics of memory and responsiveness. One thing to keep in mind is that performance and memory use on an emulator is no indication of device performance and memory overhead.

15.2.1 Memory

This section discussions factors that impact memory and speed.

15.2.1.1 Encoded Images

Memory is problematic, especially when programming small devices. When using LWUIT you must understand how memory directly relates to resolution and bit depth.

Assume you have two devices, a 16-bit color (65536 colors) device with 128x128 resolution that has 2 megabytes of memory, and a 24-bit color device (1.6 million colors) with a 320x240 resolution and 3 megabytes of memory. Which device provides more memory for a LWUIT application? The answer is not so simple.

Assume both devices have a background image set and scaled, so they need enough RAM to hold the uncompressed image in memory.

The smaller device needs 32,768 bytes just for a background buffer of the screen. The larger device requires 307,200 bytes for the same buffer!

Because screen buffers are needed both for the current form, the current transition (twice), and the MIDP implementation, the amount of memory the larger device consumes is staggering! How did we reach these numbers?

The simple formula is:

screen width * screen height * bytes per pixel = memory

Therefore:

16 bit: 128 * 128 * 2 = 32,768

24 bit: 320 * 240 * 4 = 307,200

Notice that in the 24-bit device 24 bits are counted as an integer because there is no 24-bit primitive and implementations treat 24-bit color as 32-bit color.

So getting back to the two devices: In the worst case scenario four buffers are immediately consumed, and the remaining RAM compares as follows:

16 bit: 2,097,152 – 32,768 * 4 = 1,966,125

24 bit: 3,145,728 – 307,200 * 4 = 1,916,928

It turns out the 24-bit device has more RAM to begin with but doesn't have as much RAM to work with!

Note:

All of these calculations don't take into account the additional memory overhead required for LWUIT and your application.

Thankfully, LWUIT offers a partial solution in the form of encoded images, which allow the device to cleanup unnecessary bitmap data from memory when it is scarce.

Encoded images work by using a weak/soft reference to a keep the encoded version of an image. For example, a PNG or JPEG is usually compressed at a very high ratio producing a much smaller byte size than the ones mentioned above (typically a 240x320 image can be 4-5kb or even less!). The EncodedImage keeps in memory the actual JPEG or PNG data, when image information such as pixels, dimensions etc. is needed the native Image object is dynamically created and maintained in a weak/soft reference for caching.

This allows the garbage collection to remove the image from memory when additional memory is needed, however its potentially expensive since an image might be created multiple times. It is also expensive to scale an encoded image since scaling is far more expensive for these cases.

The encoded image is the default image type returned when loading an image through a resource file.

Using encoded images, a UI-heavy application can be run on a 2 megabyte 320x240 24-bit color device. Note that using tiled images or a solid color to fill the background is much “cheaper” than the savings reachable using encoded images.

15.2.2 Speed

UI speed is often a user perception rather than a “real” performance issue. Slow performance happens, but a developer’s opinion of performance may not match an end-user’s perception. The best way to measure the speed of a UI is to give devices to a focus group of objective people and ask them how the UI “feels”.

That said, the following subsections you can monitor the event dispatch thread and LWUIT performance.

15.2.2.1 Event Dispatch Thread (EDT)

Performance often suffers because of slow paints. This often occurs when the EDT is being used without being released. It’s important not to “hold” the EDT and release it immediately when performing long running tasks. For further details on releasing the EDT see Display methods `callSerially`, `callSeriallyAndWait`, and `invokeAndBlock`.

The EDT might be blocked due to unrelated work on a different thread. Bad thread scheduling on devices causes this problem, in part because many hardware devices ignore thread priorities.

On some devices networking can cause a visible stall in the UI, a problem for which there is no “real” solution. The workaround for such cases is logical rather than technical. In this case a standard progress indicator stalls during a networking operation. It might work better to use a progress indicator heuristic that moves slower or does not move at all so the user is less likely to notice the interruption in the display.

15.2.2.2 LWUIT Performance

Different transition types have different performance overheads on devices. Play with the transition selection and possibly disable transitions if necessary.

Indexed images carry a performance overhead. It shouldn't be excessive, but when using many animations or indexed images you can expect a slower repaint cycle, especially on devices without a JIT or fast CPU.

Light mode often trades speed for memory overhead. If there is plenty of memory and low performance, explicitly turning off light mode (after `Display.init()`) might impact speed.

15.3 Device Bugs And Limitations

This section describes the device bugs and limitations the LWUIT development team found while in the process of creating demos and applications. While this is not an exhaustive list, you can apply these principles if you encounter device issues of your own.

15.3.1 Bugs

The LWUIT development team encountered several device bugs and limitations (but not nearly as many as were expected). The first rule of bug investigation is:

It is not a VM bug.

Often developers blame the VM for bugs. Despite many rumors, the development team hasn't found a CLDC 1.1 VM with a significant bug (they reproduced crashes, but they were all due to bad API implementations).

The VM and GC seem to work pretty flawlessly, which means several things should work. You should be able to rely on proper exception handling and proper class loading behavior. This essentially allows you to use Java technology for exception handling and class loading to work with multiple devices, instead of the "problematic" preprocessor statements used in the past.

The preprocessor approach was essential in the past when targeting all phones (even seriously broken VMs) with code size requirements that were very low. Today's market has changed considerably, both in the quality of the common devices and in the space or OTA code size available for a typical application.

The advantages of avoiding preprocessor are mostly in code maintenance (refactoring, compiler checks, etcetera), simplicity in reusing object oriented paradigms, and easier deployment (one JAR file for all or most devices).

Rather than beat around the bush, here are a few examples of actual device behaviors:

- A device throws an exception in a certain condition when using an API. This happens with several devices that fail in `drawRGB`. The solution is to catch the exception and activate a flag to invoke a different code path designed for that device class only.
- Some devices have a bug with API *X* or with a specific usage of API *X*. Avoid that API or usage if possible. For example, many devices have a problem with `flushGraphics(int, int, int, int)`, but all devices tested worked perfectly with `flushGraphics()`.

As you can see, you can rely on Java working properly and throwing exceptions, making it possible to implement workarounds on the fly.

15.3.2 Limitations

The rules for dealing with device limitations are very similar to the rules for dealing with device bugs. If a missing API is invoked in code, it throws an exception because it doesn't exist. You can catch that exception and activate a flag disabling the functionality related to the feature. For example, your application might offer a location based feature based on JSR 179. You can wrap the calls related to JSR 179 code in try/catch and disable the functionality if a Throwable is thrown by the code (for example, `NoSuchMethodError` or `ClassNotFoundException`).

An example of this approach exists in the M3G class from LWUIT which is designed to run on devices that do not support JSR 184. The Log class is also designed in a similar way. It can utilize the `FileConnector` when the API is available in order to log to the device file system rather than RMS.

Limitations are often related to appearance, number of colors, device speed, device resolution, and so forth. These can be worked around using a multitude of themes and picking the right default theme upon startup. Use the methods in Display to check general device capabilities, then enable or disable some features.

For example, some devices support only three alpha levels (0%, 50%, 100%). This causes anti-aliased fonts to look horrible on those devices especially when using white over black color schemes. Devices like these can be easily detected using `Display.numAlphaLevels()` and such themes can be disabled on these devices (or simply excluded from the default state). Similar properties such as `numColors` are available on display.

Speed and memory constraints are much harder to detect on the fly. `TotalMemory` is often incorrect on devices and speed is notoriously hard to detect. True memory heap can be detected by allocating byte arrays until an `OutOfMemoryError` is thrown. While the VM is not guaranteed to be stable after an OOM it generally recovers nicely. Store the amount of memory in RMS to avoid the need to repeat this exercise.

The best solution is to allow your users as much configurability as possible (to themes, animations, transitions, etcetera) thus giving them the choice to tailor the application to their device needs.

15.4 Resolution Independence

One of the biggest problems in Java ME programming is the selection of device resolutions. This problem is aggravated by lack of scaling support and the small selection of devices with SVG device. A bigger problem than multiple resolutions is the problem of varying aspect ratios, even changing in runtime on the same device! (For example some slider devices change resolution and aspect ratio on the fly.)

LWUIT solves the lack of scaling support by including a fast low overhead scaling algorithm that keeps the image's alpha channel intact. Scaling on devices is far from ideal for some image types. It is recommended that designers avoid "fine details" in images that are destined for scaling.

Since images and themes can be stored in resource bundles, such bundles can be conditionally used to support different resolutions. This solution is not practical on a grand scale with a single JAR file strategy, however, for some basic resolution and important images this is a very practical solution, especially when dynamically downloading resources from a server.

15.5 Input

This section describes input methods that LWUIT supports.

15.5.1 Soft Buttons

Soft buttons for common devices in the market are detected automatically by LWUIT. If LWUIT fails to detect a specific device a developer can still set the key code for the soft keys using setter methods in Display.

LWUIT supports 3 SoftButton navigation common in newer phones from Sony Ericsson and Nokia. The 3 SoftButton mode can be activated via the Display class. In this mode the center “fire” key acts as a soft button.

15.5.2 Back Button

Some devices, most commonly older Sony Ericsson devices, have a special hardcoded back button device. You can assign a command as a “back command” using the form method for determining the back command. This ensures that only one command at any given time is deemed as a back command. The back command can also be configured using the Display methods. Currently the back button is only supported on Sony Ericsson devices.

15.5.3 Touch Screen Devices

Touch screens are supported out of the box, however, designing a UI for finger operation is very different from designing a UI for general use. Finger operations expect everything to be accessible via taps (not keys).

A touch interface expects widgets to be big enough to fit the size of a human finger. This is somewhat counter-intuitive because normally you might want to cram as much UI detail as possible into a single screen to avoid scrolling.

Component sizes can be easily customized globally using the theme. Simply set the default padding attribute to a large enough value (e.g. 5, 5, 5, 5) and all widgets “grow” to suit finger navigation. It is also a good practice to use buttons for touch devices and avoid menus where possible.

The only problem is that currently there is no standard code that allows you to detect touch screen devices on the fly. However such information can be easily placed in the Java application descriptor (JAD) file for the application to query.

15.6 Specific Device Issues

This list is rather limited since the development team doesn't have much to say about most devices. Most of the common CLDC 1.1 devices just work out of the box without much of a hassle. This section describes behaviors that might catch developers off guard. This is by no means an exhaustive or definitive list.

15.6.1 Motorola

The RAZR family doesn't support different levels of translucency -only 50% translucency is supported. This causes anti-aliased fonts to look bad on these devices.

15.6.2 Create a `.cod` File

1. Create a new project in JDE and name it appropriately. Select project type: "Empty Midlet project".
2. Right click on the project and choose the "add file to project" option and choose the JAR file from your projects /dist directory.
3. Right click on the project and choose "properties".
4. In the "Application" tab insert the name of your main MIDlet class.
5. Build and run the project.

15.6.3 Nokia S40

Generally series 40 devices work well. Some “high end” S40 devices only contain 2mb of memory yet have 24-bit color 320x240 resolutions. These devices have 3mb installed but only 2mb is accessible to Java applications.

The Nokia S40 emulator provides a very good approximation of the devices.

15.6.4 Sony Ericsson

Sony Ericsson makes good Java devices that are indexed with memory and have 16-bit color for even better memory.

The Back button, as discussed in Section 15.5.2, "Back Button" exists in SE until JP-8, at which point a new interface based on three soft keys was introduced.

Native Networking Sony Ericsson threads in SE tend to freeze the GUI. The devices in JP-7 and newer completely ignore thread priorities as well.

15.6.5 General Portability Tip

Test on manufacturers emulators. While development is easier using the Java ME SDK, the Sprint Plugin for Java ME SDK, or the Sprint Wireless Toolkit, there is no substitute for occasional emulator testing. An emulator provides more accurate memory readings especially related to images and buffers.

LWUIT Mini FAQ

This appendix addresses common questions about LWUIT.

Performance on the Java ME SDK or the Wireless Toolkit is very slow, what is the problem?

There are documented issues of slow performance due to Hyperthreading.

- Slow loopback in the network interface (often caused by miss-configured networking) also impacts performance because the toolkit uses network calls to perform all drawing.
- Sprint WirelessToolkit versions 3.2 and higher do not have these performance issues because they feature a different architecture.

How does painting in LWUIT differ from Swing/AWT?

Generally both are very much alike. There are, however, some minor key differences that might "bite" an unsuspecting Swing/AWT developer:

- LWUIT clears the background – when drawing the component LWUIT makes sure to first clear the background for the component using the painters for its parents if necessary.
- LWUIT translates to parent component coordinates – A Swing component always starts at 0, 0. This is because Graphics.translate is invoked with the X and Y coordinates of the component. In LWUIT this is done only for parent containers, which is why the components in LWUIT must be drawn in their X and Y location. The problem with this approach is that drawing in 0,0 often works for the first component in the container and fail for subsequent components.
- LWUIT doesn't make a distinction between paintContent or paintChildren – All calls in LWUIT lead to paint and paintBorder. There is no separate call for painting the children of a container.

Scrolling isn't working like I expect. What went wrong?

There are several common possibilities.

- You nested a scrollable component within another scrollable component (this is technically legal but might look odd). By default the form is scrollable so just try invoking setScrollableY(false) on the form.
- Scrolling behaves based on the amount of space allocated by the layout manager. Some layout managers do everything to prevent scrolling (such as grid layout) while the box layout tries to increase size as much as possible. Read the documentation for your layout manager of choice.
- For group layout components (generated by the UI builder) you must make sure to mark components to grow and verify that they indeed do so in preview mode. You must size the container to match the size of the component boundaries, otherwise the container size is hardcoded.

What is a painter? Why not just use an image?

The idea behind a painter is simple, provide flexibility to the developer and allow the developer to define rendering suitable for his needs on any device. While images provide decent flexibility for artists' ideas, painters provide limitless flexibility:

- A developer can use a relatively low overhead gradient painter to get a very compelling look without a single additional image file. Furthermore, the gradient adapts nicely to any screen resolution.
- In high-end devices that support SVG, etcetera, painters can use SVG to render and scale vector graphics rather than scale raster graphics. This increases the application UI fidelity.

Is LWUIT identical across all platforms?

Yes and No.

The basic core API is the same on most tested platforms and is binary compatible, allowing MIDP applications to run on Java SE (for example, in the Resource Editor actual MIDlet code is running in Java SE).

The catch is in several details:

- Some components aren't available in other platforms: M3G, Media (sometimes available), and SVG.
- Rendering might seem different on other platforms due to platform issues. For example, in some platforms LWUIT takes advantage of anti-aliasing. System fonts look completely different between platforms and bitmap fonts look odd in some platforms that don't properly support alpha channels.
- Different platforms have different performance characteristics.

For more details on these issues check out the portability chapter.

Does LWUIT support 3 SoftButton devices?

Yes, 3 SoftButton mode is implemented in display. However, since there is no reliable way to detect 3 SoftButton phones this features can be activated either programmatically or through a JAD file attribute.

A device doesn't seem to work with LWUIT. What should I do?

Is it a MIDP 2.0/CLDC 1.1 device? If it is then please mail lwuit@sun.com with the following additional details:

- Does LWUIT hello world work on the device?
- Does the LWUIT UIDemo work on the device?
- What is the resolution+color depth of the device, and how much memory is allocated for Java?

I want my application to look "native" on the device. Is there any way to accomplish that?

While LWUIT is designed to do the exact opposite (support your own look and feel) a native look and feel can be partially achieved if you implement a theme or look and feel that resembles the native look.

This won't work very well on most devices since there is no way to detect if the user switched the default theme.

Downloadable themes are probably a good approach for a strong user community.

The UI for my touch screen phone seems too small for my fingers. How do I make the UI more appropriate for such a device?

Use a global padding setting in the theme to increase the size of all widgets to a point where they are big enough for a finger tip.

Why am I getting memory errors in LWUIT? Why is LWUIT is consuming a lot of memory in my memory monitor?

Check that your application doesn't hold onto pointers for components. Because a component references its parent component, holding onto a single button can keep an entire form with all its content in memory... LWUIT allocates and discards frequently to allow for a small memory footprint. This causes the graph of free memory to fluctuate but the alternative of caching would be worse for highly constrained devices. Check out the LWUIT blog for more information on the subject of tracking and identifying memory issues.

Why won't my list/text area scroll? Why does my list/text area jump around?

You need to disable the scrolling for the form using myForm.setScrollable(false) and you should place the list in the center of a border layout. For deeper understanding of why this is required, read the next question about scrolling.

How do I change the appearance of the list? Remove the numbers from the side etcetera? Can I nest containers in the list?

Listis designed for a very large number of element and fast traversal. You can use its cell renderer facility to customize it any way you want as explained here. How the list can scale and grow is explained here and additionally here.

My application freezes or stalls. How do I fix this?

99% of the problems of this type are related to Event Dispatch Thread (EDT) violations.

What is the Event Dispatch Thread (EDT)?

The EDT broadcasts all the events in LWUIT. It is also responsible for drawing all elements on the screen.The EDT thread is responsible for drawing all screen elements, if it is blocked by a long running operation elements won't update and key/pointer events won't be received. The solution is to use threads for such long running tasks, however interaction with LWUIT can only be performed on the EDT. To return into the EDT you can use Display.callSerially and callSeriallyAndWait. A different option is to use invokeAndBlock.

I'm not opening any threads, why am I having problems?

A typical application always uses at least two threads, lifecycle and the EDT. The lifecycle thread is the callback thread used for the application. For example, in MIDP the startApp method is invoked from the MIDP thread which is different from the EDT.

Does anything work from a separate thread?

There are no guarantees, but repaint() should generally work from every thread and show() should as well.

How do I reposition/resize a dialog?

Use a Dialog instance and a version of show which accepts 4 integer values to position the dialog. You can use the set the default dialog position to simplify dialog positioning.

How do I show Video?

Use MMAPI to create a Player object, then use the VideoComponent class.

Can I create my own components?

Everything in LWUIT is fully extensible.You can derive from any component and extend it. It is demonstrated in the Chapter 14 and it is discussed extensively in the blog at http://codename-1.blogspot.com/.

I'm trying to display an animated gif. Why isn't it working?

Animated gifs can be shown in MIDP using the MMAPI and VideoComponent (see How do I show Video?). LWUIT has special support for StaticAnimation which is a LWUIT specific format very similar to the animated gif. Both the Resource Editor and the Ant task accept animated GIF files to create static animations.

I'm having issues on a Windows Mobile device?

Windows mobile VMs vary significantly in their quality. If the VM is giving you problems try the Phone ME open source VM port for Windows mobile devices.

How do I create resource (.res) files?

Use the Resource Editor or the Ant task.

What is the difference between the Resource Editor and the Ant task?

The difference is mainly in the use case, the ant tool is designed mostly for developer related resources (locales, application images, etcetera). The Resource Editor is targeted for use by graphic designers.

Index

W