From Technologies to Solutions

LWUIT 1.1 for Java ME Developers

Create great user interfaces for mobile devices

Biswajit Sarkar

PACKT PUBLISHING

LWUIT 1.1 for Java ME Developers

Create great user interfaces for mobile devices

Biswajit Sarkar

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

LWUIT 1.1 for Java ME Developers

First published: August 2009

Production Reference: 1120809

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847197-40-5

www.packtpub.com

Cover Image by Parag Kadam (paragvkadam@gmail.com)

Credits

Author
Biswajit Sarkar

Reviewers
Lucas Hasik
Valentin Crettaz

Acquisition Editor
Douglas Paterson

Development Editor
Dhiraj Chandiramani

Technical Editor
Shadab Khan

Copy Editor
Leonard D'Silva

Indexer
Monica Ajmera

Editorial Team Leader
Akshara Aware

Project Team Leader
Priya Mukherjee

Project Coordinator
Zainab Bagasrawala

Proofreader
Claire Lane

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

About the Author

Biswajit Sarkar is an electrical engineer with a specialization in Programmable Industrial Automation. He has had extensive experience across the entire spectrum of Industrial Automation—from hardware and firmware designing for general and special purpose Programmable Controllers, to marketing and project management. He also leads a team of a young and highly talented group of engineers engaged in product (both hardware and software) development. He has been associated with a wide variety of automation projects, including controls for special purpose machines, blast furnace charge control, large air pollution control systems, controls for cogeneration plants in sugar factories, supervisory control for small hydel plants, turbine governors, and substation automation including associated SCADA.

Currently Biswajit consults on Industrial Automation and Java ME based applications. He has written extensively for Java.net on Java Native Interface, Java ME and LWUIT. He has taught courses on mathematics and analytical reasoning at a number of leading institutes in India. Biswajit has also taught a specially designed course on Java for MS and Ph.D. students as well as post doctoral fellows at the Center for Coastal Physical Oceanography, Old Dominion University, Norfolk, Virginia (USA).

Biswajit, originally from Calcutta, now lives in Nashik, India with his wife.

This book would never have seen daylight had it not been for the excellent support that I received from the editorial team at Packt Publishing. I must express my grateful appreciation of the roles played by Douglas Paterson at the critical formative stage of the book, and, later by Dhiraj Chandiramani. Lata Basantani and Zainab Bagasrawala made sure that the project remained on schedule, while Shadab Khan and his team deftly guided the completion process. I am grateful for the comments of the reviewers that helped me maintain clarity of thought, and ensured the technical integrity of the book.

On the personal front, first and foremost, I am indebted to Dada who equipped me with the ability to undertake such an activity. The encouragement and unstinting support I received from my wife Jyoti were a great source of strength and helped me survive those difficult times when I was nearly swamped by my various commitments and the temptation to give up was great. Isaac, my son-in-law, has always encouraged me to write and was a great confidence booster. Finally, I must acknowledge the sacrifices made by my grandchildren Anunita and Ian who spent many unhappy days and evenings without my participation in their games.

About the Reviewers

Lukas Hasik is Java enthusiast that likes to break the limits. However, he will always remember that real life happens out of the wires and chips.

Lukas works for SUN Microsystems from fall 2000. He used to be part of the NetBeans team, where he led a Quality Assurance team for NetBeans Mobility and NetBeans Core & Platform. Lukas has moved to the Compute Cloud group in 2009 and leads the QA team. He spoke at several conferences on topics about Java, Tools, and Testing.

I'd like to thank my employer for the extra time that I spent on airplanes, at airports, and in hotels during business trips. Those are the moments that I used for reviewing this book, and thanks to my wife Kamila for her patience during the nights of insomnia.

Valentin Crettaz holds a master degree in Information and Computer Science from the Swiss Federal Institute of Technology in Lausanne, Switzerland (EPFL). After he finished studying in 2000, Valentin worked as a software engineer with SRI International (Menlo Park, USA) and as a principal engineer in the Software Engineering Laboratory at EPFL. In 2002, as a good patriot, he came back to Switzerland to co-found a start-up called Condris Technologies, a company that provides IT development and consulting services and specializes in the creation of innovative next-generation software architecture solutions as well as secure wireless telecommunication infrastructures.

From 2004 to 2008, Valentin served as a senior IT consultant in one of the largest private banks in Switzerland, where he worked on next generation e-banking platforms.

Starting in 2008, Valentin joined Goomzee Corporation as Chief Software Guru. Goomzee is a Montana-based company that provides solutions for connecting buyers and sellers in any market vertical through mobile interactions.

Valentin also owns a small consultancy business called Consulthys, a new venture that strongly focuses on leveraging Web 2.0 technologies in order to reduce the cultural gap between IT and business people.

Table of Contents

Preface

The Lightweight Toolkit (LWUIT) is designed to help developers to create highly attractive User Interfaces for MIDP 2.0 and CLDC 1.1 compliant small devices like mobile phones. This toolkit supports a number of interesting widgets and features like theming, animations, transitions, and logging. LWUIT also addresses the issue of fragmentation by making it possible to implement screens with a device independent look and feel.

This book covers the widgets and functionalities of the library in detail, demonstrating their use with a large number of examples and a profusion of screenshots. A number of structural and architectural issues are discussed to help you gain insight into the inner workings of the library.

LWUIT is an evolving library and we are bound to see modifications and additions to its current repertoire. The knowledge you gain from this book will help you significantly in understanding these changes and in remaining up-to-date. The Lightweight Toolkit Library is an external API that is not an integral part of the Java platform and has to be bundled with an application meant for a physical device. One implication of this is that any application you write based on a given version (like version 1.1) will not become obsolete and will work on future devices too.

This book will equip you with the knowledge and skills required to create applications that will impress users with visual sophistication.

What this book covers

Chapter 1 tells you what LWUIT is all about and, broadly, how it operates. Starting with an overview of LWUIT which present the widgets and the functional features, this chapter goes on to discuss the basic architecture of LWUIT and ends with introductions to the two classes that are its foundations—LWUITImplementation and Display.

Chapter 2 lists the items that you will need to download and tells you where to find them. It prepares you for trying out the examples in the book and for creating your own applications by building a demo project. Next, you get to know the Component class, the component rendering process, and the Graphics class. Finally, this chapter lays the foundation for using Style and Animation with components.

Chapter 3 deals with the Container class, which is designed to be the holder of components. There are a number of descendants of Container—the Form, the Dialog, the Calendar and the TabbedPane. These classes also are discussed in detail with examples to show how they can be used in applications.

Chapter 4 covers Labels and the three components that are its descendants. These are the Button, the CheckBox and the RadioButton. RadioButtons exhibit special properties when they work with the ButtonGroup class and this aspect is demonstrated through an example. This chapter also takes a look at the Border class, which is used in the examples.

Chapter 5 demonstrates how flexible a List, and its subclass ComboBox, can be. This flexibility is shown through the examples that use custom renderers to enhance the appearance and functionality of lists and combo boxes.

Chapter 6 explores TextArea and TextField—the two classes that enable users to enter, display and edit text. A text field has the interesting property of in-place editing and this is treated in detail in this chapter.

Chapter 7 takes you through the various layout managers that arrange components on containers. There are six layout managers and the examples show the different ways in which these classes place components. The root of these six classes is the Layout class, which too is studied here.

Chapter 8 shows how custom components can be built. Building such a component involves not only visual aspects but also issues like styling, event handling and event generation. All of these topics are dealt with in this chapter through the examples.

Chapter 9 demonstrates how LWUIT handles various non-code elements that may be required by an application. Images, Fonts, and Animation Resources are examples of such elements. Resource files are used to package these elements and the Resources class provides the methods for extracting them from a resource file. The LWUIT bundle contains LWUIT Designer, which is a very convenient utility for creating resource files. This chapter examines how resource files are built and used.

Chapter 10 is about Themes. Themes are used to establish visual coherence through all the screens of an application. The LWUIT Designer is the tool that displays, edits and builds the themes that define how your applications will look. In this chapter, you will learn about themes, their usage and how they can be created.

Chapter 11 shows off two fascinating functionalities of LWUIT—Animations and Transitions. Animations involve repeated rendering on a component while Transitions determine the way in which a form is moved out of or brought into display. In this chapter, you will study these two features and see how to use them in actual applications. You will also see how to develop a custom transition which demonstrates the process of such customization.

Chapter 12 shows you how the Painter interface can be used to customize the appearance of a component's background. This chapter also explains how a transparent or translucent layer (like a glass pane) can be placed over a form to implement interesting visual effects.

Chapter 13 covers two useful utilities that come with the LWUIT library. These are the Effects and the Log classes. The Effects class simulates the reflection of an image and appends the reflection to the original image. The Log class enables you to monitor at runtime the inner workings of the classes that you write. This can be a very effective debugging tool. This chapter demonstrates the use of Effects and Log classes. It also examines the structure of Log class through an example that builds its subclass to provide additional capabilities.

What you need for this book

The following are required for this book:

The LWUIT bundle—this can be downloaded from `https://lwuit.dev.java.net/servlets/ProjectDocumentList`

A JDK. If you do not have one installed on your computer, you can get the latest version at `http://java.sun.com/javase/downloads/index.jsp`

The Sprint Wireless Toolkit 3.3.2 which is available at `http://developer.sprint.com/site/global/develop/technologies/java_me/p_java_me.jsp`

Who this book is for

This book is for Java ME developers who want to create compelling user interfaces for Java ME applications, and want to use LWUIT to make this happen.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code will be set as follows:

```
public class DemoForm extends MIDlet
{
   public void startApp()
   {
      //initialize the LWUIT Display
      //and register this MIDlet
      Display.init(this);
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be shown in bold:

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

   //act on the command
   public void actionPerformed(ActionEvent ae)
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "clicking the **Next** button moves you to the next screen".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to `feedback@packtpub.com`, and mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for the book

Visit `http://www.packtpub.com/files/code/7405_Code.zip` to directly download the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration, and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Introduction to LWUIT

The **Lightweight User Interface Toolkit** (**LWUIT**) is a UI library for the Java ME platform. It enables a developer to create visually attractive and functionally sophisticated user interfaces that look and behave the same on all Java ME enabled devices compatible with MIDP 2.0 and CLDC 1.1. In this library, there are enhancements to graphical components that are part of the `javax.microedition.lcdui` package, brand new widgets like the TabbedPane and Dialog, and there is also support for new features like animation and transition. The Swing like architecture of LWUIT permits customization of the appearance of an application. And, best of all, it makes sure that our applications will look just the same, regardless of the platform they are deployed on.

In this chapter, we shall cover the following:

- An overview of LWUIT
- A look at the basic architecture
- An introduction to Implementation and Display classes

By the time you get through this chapter, you will know what LWUIT is all about, and broadly, how it operates.

Why we need the LWUIT

Java ME allows us to write applications that are, generally speaking, portable across a wide range of small devices that support the platform. While the basic functionalities usually work well on all supported devices, the area that does pose problems for developers is the User Interface. Native implementations of `javax.microedition.lcdui`—the primary API for UIs in Java ME, differ so widely from one device to another, that maintaining a device-independent and uniform look-and-feel is virtually impossible.

Non-uniform look-and-feel is not the only reason why developers have been waiting for something better to turn up. The `javax.microedition.lcdui` package does not support components and capabilities that can fully satisfy present day user expectations.

This is why the arrival of LWUIT is so exciting. LWUIT offers a wide range of **Widgets** for building UIs. While some of these widgets are also available under **lcdui**, there are a number of new ones too. These additions enable application developers to design UIs that can come very close to their desktop counterparts in terms of visual sophistication. Even the components that are also offered by lcdui have been functionally enhanced. LWUIT is not just about new components. The API supports a whole range of new functionalities (Theming, Transitions, and more).

LWUIT overview

Our overview of LWUIT will discuss the following aspects:

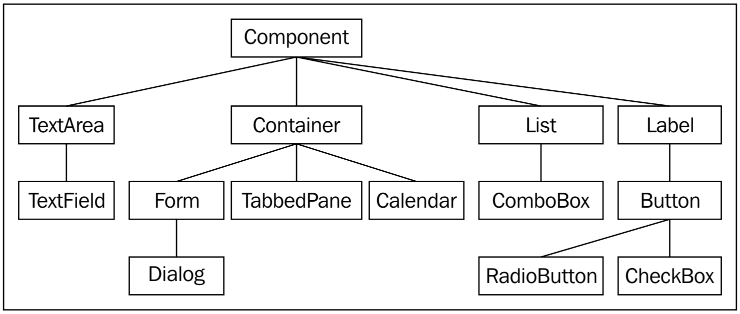
- Widgets
- Infrastructural items like Resource, UIManager, and so on
- Functionalities like theme and transition

Widgets

A **Component** is an object that has a visible avatar. Generally speaking, a component can also sense and react to customer inputs. A **Widget** is a specific type of component with its own distinctive look and feel. A button is a widget and so is a combo box.

During our exploration of LWUIT, we shall keep using the terms Widget, Component (with a capital C) and component. To make sure that we avoid any confusion, let us define what we mean by these terms. The LWUIT library has a `Component` class that is the superclass for all widgets and embodies their essential qualities. The word Component (yes, with a capital C) will be used to refer to the class, while component (with a lowercase C) will mean any instance of the `Component` class. This principle of distinguishing between *Component* and *component* will be applicable to all classes and their instances. So the word *Label* refers to the class, and the word *label* refers to a specific object of that class.

The widgets are the visible faces of LWUIT. So before we delve into the inner details of the library, let us check out the widgets. The following figure is the widget family tree showing all major widgets.



Container and Form

Among the widgets, the Container is the basic 'holder' which can contain other components, including other containers. This nesting ability of containers allows complex and elaborate UIs to be built-up. The arrangement of components within a container is taken care of by a layout manager.

Form is a container with a TitleBar at the top, on which the title of the form can be written, and a MenuBar at the bottom for commands and menu. The space between the two bars is for the content pane, which holds the components that are to be placed on the form.

The container branch of the family tree also has `Dialog`, `TabbedPane`, and `Calendar`. All of these Widgets will be introduced here, and dealt with in detail in Chapter 3.

The previous screenshot shows a form and its constituent areas. The title of the form appears on the TitleBar, and the border at the bottom with the **Exit** command is the MenuBar. We can also see where the content pane goes.

The TabbedPane

A **TabbedPane** lets a number of component groups share the same space. Each group of components has a tab associated with it, and the user sees only the group corresponding to the tab that has been selected.

The default placement of tabs is at the top of the pane. However, it is possible to position them at the left, right or at the bottom of the pane. The following screenshot shows a tabbed pane with the tabs at the top, and with the first tab selected.

Calendar

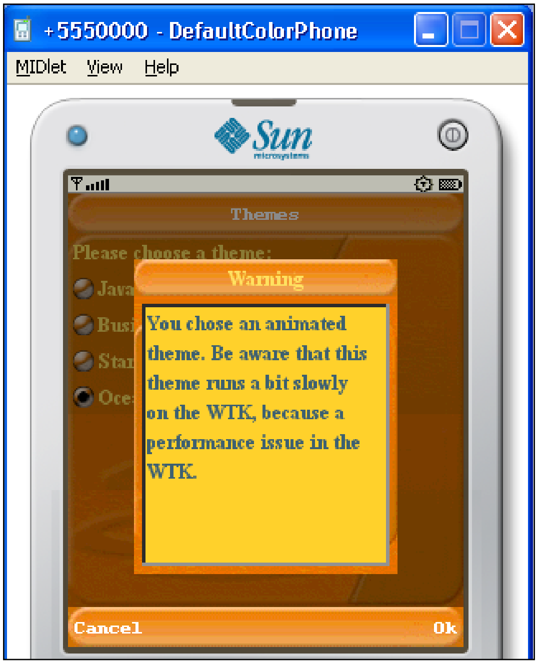
This widget shows date information and supports 'scrolling' through dates and months. The following screenshot shows a calendar:

Dialog

A **Dialog** is a component that is usually displayed against a tinted background and covers a part of the screen. By default, a dialog is modal, that is, it blocks the calling thread until it is closed by calling the `dispose ()` method. Dialogs can be one of five types:

- Alarm
- Confirmation
- Error
- Info
- Warning

Dialogs can be used to convey information to the user at runtime, and also to collect user feedback. The type of the dialog would normally determine the sound to play when the dialog is displayed. It is possible to add icons to a dialog to graphically indicate the dialog type. The following screenshot shows a simple warning dialog with the **Ok** and **Cancel** commands:



Label and Button

A **Label** displays images and text that cannot be selected. A label does not interact with the user. A number of alignment possibilities are supported for positioning the label, and also for positioning the text with respect to the image on the label. `Label` and its descendants (`Button`, `RadioButton`, and `CheckBox`) are covered in Chapter 4.

A **Button** is primarily intended for capturing an input from a user. In a way, we can think of a button as a label that can sense user action—specifically, a click.

The `Button` class extends `Label`. A button has states, and it generates an action event when selected and clicked on. Like a label, a button can have a descriptive text or an image or both. The three states of a button are as follows:

- Rollover—this is normally equivalent to the button being selected or receiving focus.

- Pressed—when the button has been clicked on or pressed.
- Default—when the button is neither selected nor clicked on.

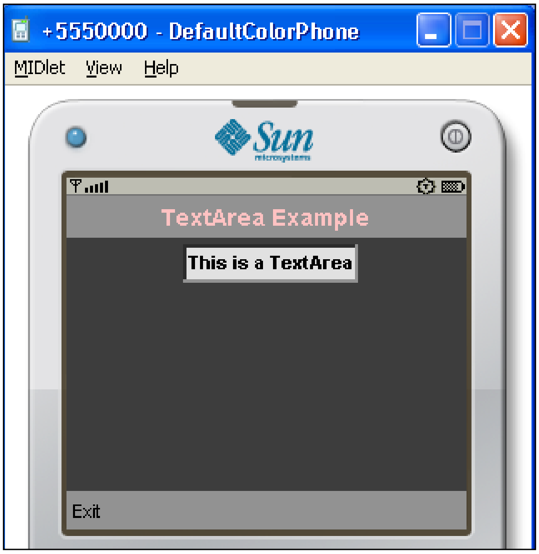
A button has two subclasses—`RadioButton` and `CheckBox`. A radio button has all of the functionalities of a button. In addition, a radio button can be added to a `ButtonGroup`, which allows it to maintain its selection state exclusively within the group. Within a button group, only one radio button can be in the pressed state at a time. If another radio button is clicked on, then the one that was in the pressed state earlier will get deselected. Also note that once a radio button is in the pressed state, clicking on it does not change the state.

A check box is similar to a radio button in the sense that it can remember its state, if it has been selected by being clicked on. However, repetitive clicking on a check box will toggle its state between selected and deselected. Another difference is that a check box cannot be a part of button group.

A number of labels, buttons, radio buttons, and check boxes are shown in the following screenshot. The screenshot shows that the first two buttons are in the *default* state, while the third is in the *rollover* state. Both the check boxes are shown *selected,* but only one radio button in each group is *selected*.

TextArea and TextField

`TextArea` is a component that displays text that may be editable. The editing is performed using the native system editor, which often opens a new screen. A text area uses the concept of *constraints* just like **TextField** of `javax.microedition.lcdui`. The constraints allow a text area to be tailored to accept a particular type of input like a password or an email address. We shall study this in detail when we discuss `TextArea` in Chapter 6. The following screenshot shows a form with a text area:



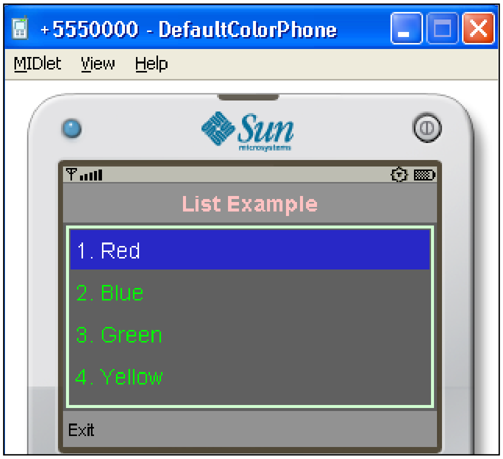
A `TextField` (this one is a part of LWUIT) extends `TextArea`, and provides a single line text display. It differs significantly from `TextArea` in the way that it supports text editing. In Chapter 6, we shall study `TextField` in detail.

List

A **List** is a very widely used component that presents a number of information items to the user. The items are presented in a single column, but each item may contain multiple lines of text and images.

The data structure of a list is represented by the `ListModel` interface. Therefore, a list is not tied to any specific data structure, and can display information from any object that implements this interface. Similarly, the rendering of a list is performed by any class that implements the `ListCellRenderer` interface, thus making it possible to create the kind of look the application developer wants. The library includes default implementations of the above interfaces that make it easy to instantiate a list.

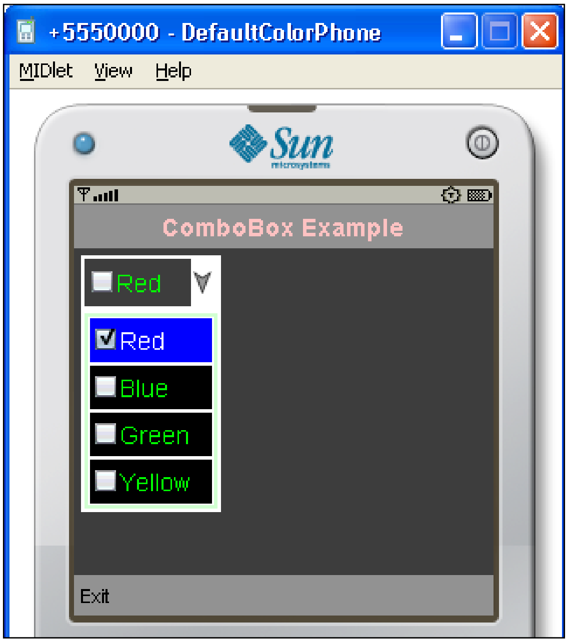
`List` and its subclass `ComboBox` will be discussed in Chapter 5. The following screenshot shows a simple list with four items:



Compare this with the list shown in the following screenshot and its significantly different appearance. The application behind the following screenshot implements its own renderer to achieve a very polished and attractive look. We shall see how to write custom renderers in Chapter 5.

ComboBox

The **ComboBox** is a space saving component that drops down a list when its button is clicked on. The user can then select one of the items on that list. As `ComboBox` extends `List`, you can use custom models and renderers for combo boxes too. The following screenshot shows a combo box with a four item drop-down list:



The underlying support elements

LWUIT incorporates a number of 'behind the scenes' support elements that make it possible for the functional features to operate. We shall now take a look at these elements, and see how they work.

Resource

Applications often require non-code items. For example, an image used as an icon for a label by an application, is such an item. LWUIT allows such items to be packaged in bundles.

A resource bundle is essentially a binary file that can be loaded into a device. An application can have more than one resource file, and each such file can contain resource elements of the same or different types.

The following types of resource elements are supported:

- Image resources
- Animation resources
- Bitmap fonts
- Localization bundles
- Themes

Individual resource elements can be extracted by a set of appropriate methods in the `Resource` class. Resource files can be built by using the **LWUIT Designer**, which is an utility that is a part of the LWUIT bundle. In Chapter 9, we shall study the `Resource` class and resource files, and we shall also learn how to use the LWUIT Designer.

Layout managers

LWUIT brings the power of sophisticated **layout managers** to the design of UIs for small devices. These managers are responsible for arranging widgets on a container.

The managers supported are:

- `BorderLayout`
- `BoxLayout`
- `FlowLayout`
- `GridLayout`
- `GroupLayout`

Layout managers will be covered in Chapter 7.

Style

LWUIT provides a very convenient way of defining the appearance of a component at a single point. Each component has a `Style` object associated with it. The attributes related to the look of the component can be set within this object. The attributes are:

- Colors
- Fonts
- Images
- Margin
- Padding
- Transparency

When a component is generated, it constructs a default style object. The values of the attributes of this object can be changed at runtime to modify the appearance of a component. We will talk about styles in greater detail in Chapter 3. We will also use style objects to specify the appearance of a widget in many other chapters.

Painter

The **Painter** interface, discussed in Chapter 12, allows you to draw on the background of a component. A painter can be used to draw a specific pattern on the background of one, or a group of components. The LWUIT package provides two classes that implement this interface:

- `BackgroundPainter`—draws the background of a component based on its style
- `PainterChain`—creates a 'chain' of painters that can produce a layer effect, having each painter draw just one element

A painter can also be used as a transparent or translucent layer called a **glass pane** on top of a form.

UIManager

An interesting feature of LWUIT is that it manages the appearance of an entire application from a single point. The `UIManager` is the class that co-ordinates the visual aspects of an application. This is a singleton to ensure that there is only one instance of `UIManager` per application. The methods of this class enable it to impose the same look on all components and to localize the UI on the fly. As `UIManager` is the class that controls the rendering of components, there will be many examples in this book that will demonstrate the use of this class.

LookAndFeel

LookAndFeel is the interface that takes care of all rendering for an application. Therefore, it is possible to completely customize the appearance of an application by implementing the appropriate methods of this interface. The concrete implementation of `LookAndFeel` that comes with the LWUIT package is `DefaultLookAndFeel` and this class controls the actual rendering of the default look of the application. We can also plug in a custom implementation of `LookAndFeel`, by using the `setLookAndFeel` method of `UIManager`. We shall use `DefaultLookAndFeel` in many of our examples in this book.

Functionalities

A really interesting aspect of LWUIT is that it goes beyond just components, and offers features that enable us to produce very sophisticated UIs. In this section, we introduce these features.

Animations and transitions

The LWUIT library supports animation at different levels, and also the implementation of different modes of transition from one form to the next one to be displayed. In the context of LWUIT, animation essentially allows objects to paint succeeding frames at timed intervals. Transition refers to the way in which a form is brought into, or taken out of the display screen. **SlideTransition**, for instance, makes the new form slide in pushing the old one out.

The basic implementations of such transitions are achieved through the mechanisms provided by the **Animation** interface and the classes `Motion`, `Transition`, `CommonTransitions`, and `Transitions3D`, which are in the `com.sun.lwuit.animations` package.

The `Animation` interface defines objects that can be animated. All components are inherently capable of performing animation tasks, as all of them implement `Animation`.

`Transition` is an abstract class that represents animation from one form into another. Its two concrete subclasses are:

- `CommonTransitions`—has methods to implement the two common types of transitions which are `Slide` and `Fade`.
- `Transition3D`—performs transitions (`Cube`, `FlyIn`, and `Rotation`) that require device support for 3D Graphics. Therefore, they do not work properly on every device.

`Motion` is a class that implements motion equations. The types of motion that are built-in are `Linear`, `Spline`, and `Friction`.

In Chapter 11, we shall go into the details of these two functionalities, and create a custom transition.

Themes

A **Theme** defines a uniform look for all visual components in an application. A theme file is essentially a list of key-value pairs that define the relevant attributes. We have seen that a style object specifies how a particular component instance will look. A theme can be thought of as a common style for the components that appear in it as "keys". For example, if you want all buttons in your application to have a green background, then you can do it by defining an appropriate "key-value" pair in your theme. We shall learn how to apply themes in Chapter 10.

Logging

This is a debugging aid that allows you to log information at runtime. The `com.sun.lwuit.util.Log` class provides methods for logging information into a *log* file (created in the `root` directory) using the `FileConnection` API, for example, and also for displaying logged information on a form. You can use the information saved in the log file through the `FileConnection` API.

The use of logging will be studied in Chapter 13.

The Basic architecture

The creators of LWUIT have said that this library has been inspired by Swing. Naturally, the architecture of LWUIT has adopted a number of features from Swing, modified to fit the constraints imposed by the small device environment in which it is meant to operate.

Like Swing, LWUIT renders all the visual entities itself, and does not derive them from native equivalents. This ensures device independence and visual uniformity. Two other features that are of interest to us are the modified **Model-View-Controller** (**MVC**) model and the **Event Dispatch Thread** (**EDT**).

The modified MVC model separates the logic (and data) that specifies the behavior of a component from the part that takes care of its appearance and user interface. For example, a list will have a part that contains the items to be shown (the model), and another that defines its look and its interaction with the user like scrolling (the combined view and controller). The two parts will interact with each other through predefined rules. In the context of LWUIT, this means that the look and feel of a list can be modified by plugging in a new combination of view and controller. The new list will look, and perhaps interact with the user differently from the "standard" list that comes with the library. However, its operation will be correct, as long as the rules of interaction between the two parts of the list are adhered to. In essence, this is the concept of the **pluggable look and feel** (**PLAF**) capability that LWUIT offers us. Through PLAF we can customize the visual aspects of a component quite easily.

The EDT is a thread that handles all interaction between the LWUIT objects in an application. Since the interactions take place over a single channel, they occur serially. This helps to avoid logical race conditions, and the highly frustrating bugs that they can produce.

In the next section, we shall see how EDT handles inter-component communication, and how critical it is to the proper operation of LWUIT-based user interfaces.

LWUITImplementation—the foundation of LWUIT

The `LWUITImplementation` is an abstract class that extends `javax.microedition.lcdui.game.GameCanvas`, and performs a number of critical functions. The `GameCanvasImplementation` class extends `LWUITImplementation`, and implements the abstract methods of its superclass. To obtain an instance of `LWUITImplementation`, the `createImplementation` method of the `ImplementationFactory` class needs to be called. This method returns an instance of `GameCanvasImplementation` that works as the default LWUIT implementation. This activity of creating an LWUIT implementation object is performed by the `Display` class, as explained in the next section, and is transparent to the application developer. Although, a developer who wants to use LWUIT to create a user interface would have no need to have access to this class, a broad understanding of the role played by the `LWUITImplementation` class helps in more effective utilization of the library.

The key issue about the structure of LWUIT is that it is built on top of `javax.microedition.lcdui`—the UI package that comes with MIDP 2.0. This means LWUIT needs to use the infrastructure provided by lcdui to render everything. Specifically, lcdui has to provide a surface on which LWUIT can draw. So the starting point for an LWUIT application is an instance of `LWUITImplementation` (which is actually a game canvas object) on which all components are drawn. Regardless of the number of forms and widgets in your application, there is just one instance of `game canvas`, which is used for rendering the various visual entities. Therefore, when you see a form replace another on the screen, remember that the new form has been drawn on the same object on which the earlier form had been rendered.

In addition to being the common rendering surface, `LWUITImplementation` performs two very important tasks—it listens for all commands directed to the LWUIT environment, and also acts as EDT, the main thread for input events, painting, and animation. To get an idea of how this works, let's consider a simple example. The following screenshot shows three buttons with the top-most button having focus:

If the scroll-down key is pressed now, then an event is sent to the underlying game canvas, that is, the LWUIT implementation object. The LWUIT implementation object does not directly act on the event, but puts it in a queue for the EDT to handle.

The EDT takes events sequentially from the queue, and sends them to the form that is currently on display. The form keeps track of the component within it that has focus. It also knows the details of components contained by it, and the order in which focus will move from component to component. When the form finds that it has received a scroll-down event, it transfers focus to the next button. If the *select* key is pressed, then the form would know the event is meant for the button that has focus, and would retransmit the event to the appropriate button. The button will redraw itself as its state has changed, and call the registered listener(s). Finally, when the event reaches the listener for the button, the required action is taken.

The EDT also handles painting and animation activities in a similar way. Therefore, the EDT can be thought of as the lifeline of an LWUIT application, and care must be taken not to block it. Developers writing code for animation, painting, and for handling events have to understand that if the EDT ever gets blocked, then the user interface will become unresponsive.

The Display class

This is the class that manages the painting and event handling with `LWUITImplementation` as a partner. `Display` is used to show forms and also to start the EDT. Like `LWUITImplementation`, there is just one instance of `Display` for an LWUIT application. However, unlike `LWUITImplementation`, it can be accessed, and used by us.

Before any form is shown, we must register the current MIDlet by calling the static method `Display.init(MIDlet midlet)`. The `init` method instantiates `LWUITImplementation`, as well as a new Thread—the EDT and starts it off. It also invokes the `init(MIDlet midlet)` method of `LWUITImplementation` to get the underlying instance of the `javax.microedition.lcdui.Display` class. Obviously, nothing much can be done with the LWUIT API until the `MIDlet` is registered with the `Display` class of LWUIT.

> The fact that there are two `Display` classes—one in the LWUIT library and the other in the `javax.microedition.lcdui` package—can be a source of confusion. In the rest of this book, the word **Display** by itself will refer to the class in LWUIT. We will not need to talk about the other `Display` class very often, but when we do, it will be referred to as the `javax.microedition.lcdui.Display` class.

Summary

LWUIT is a powerful UI library for the Java ME platform. In this introductory chapter, we have laid the basic foundation through a bird's-eye view of LWUIT. We have:

- Identified the constituents like the various widgets, support functions, and new capabilities like animation and transition
- Understood the architectural similarities between Swing and LWUIT
- Checked out the implications of the architectural features
- Been introduced to `LWUITImplementation` and `Display` classes

2
Components

In the last chapter, we saw that the root of all widgets is the `Component` class. In this chapter, we shall study the `Component` class, and also gear up for hands-on work with LWUIT. The main topics that we shall cover are:

- Downloading the LWUIT bundle
- An introduction to a development tool—the Sprint Wireless Toolkit
- Building the "Hello LWUIT!" demo using the development tool
- The `Component` class, its constructor, and the kinds of methods in the class
- The component rendering process
- The `Graphics` class
- Support for Animation and Style in `Component` class

The LWUIT bundle

One of the things that will be very essential for our exploration of the LWUIT API is the LWUIT bundle. The following is the URL of the page that lists all LWUIT documents and files for download: `https://lwuit.dev.java.net/servlets/ProjectDocumentList`. The bundle that is required for this book is `LWUIT_20081222`, which corresponds to LWUIT 1.1.

As we set out to explore LWUIT, the first thing to do is download this bundle. The documentation directory has a **Developer's Guide**, as well as the usual Javadoc files. The Developer's Guide is an extremely valuable resource for all those who want to work with LWUIT. The Javadoc documentation is the official reference for all packages in the library, and I'm sure that you will frequently refer to them for the most authentic (although sometimes cryptic) information. The package also contains the demo application and the **LWUIT Designer**, which we shall discuss in Chapter 9.

Unzip the downloaded file into a folder of your choice. Let's say you have downloaded the version dated December 22, 2008 and unzipped it into your `C:\` drive. The root directory for the LWUIT bundle would then be `C:\LWUIT_20081222`. We will refer to this directory as `LWUIT_HOME`.

Getting equipped

Creating Java ME applications is best done through one of the development tools that are freely available. The applications in this book have been developed on Sprint Wireless Toolkit 3.3.2 (referred to as **SWTK** from now on), and I would suggest that you install this toolkit on your computer. I chose the SWTK because, at the time of writing, this was the toolkit that LWUIT worked very well with—even better than with the Sun Java Wireless Toolkit 2.5 for CLDC.

The Java ME platform SDK had just been announced, but it was still evolving at the time of writing. Another attractive feature was the fairly wide variety of real life device emulators to run the applications.

The SWTK is a free download from:

`http://developer.sprint.com/site/global/develop/technologies/java_me/p_java_me.jsp`

The SWTK will need a JDK, which provides the overall Java environment. If you don't have one already on your computer, then you can get the latest version at:

`http://java.sun.com/javase/downloads/index.jsp`

If you don't have the JDK already, then download and install it before installing the SWTK. The SWTK automatically locates the JDK at the time of installation. This makes it unnecessary to worry about such things as path/classpath settings.

Once the SWTK is installed we are ready to go. The installation directory of the toolkit will be called `SWTK_HOME`. In my computer, for instance, the toolkit is installed in `D:\` drive, and `SWTK_HOME` would refer to the `D:\SPRINT_WTK_332` directory.

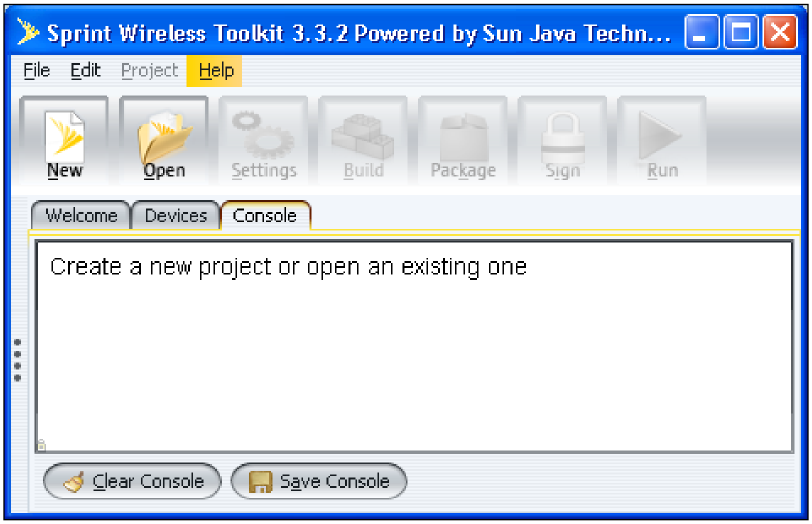
Now that the necessary tools and tackles are in place, we will build and test a simple application to familiarize ourselves with the development process.

Hello LWUIT!

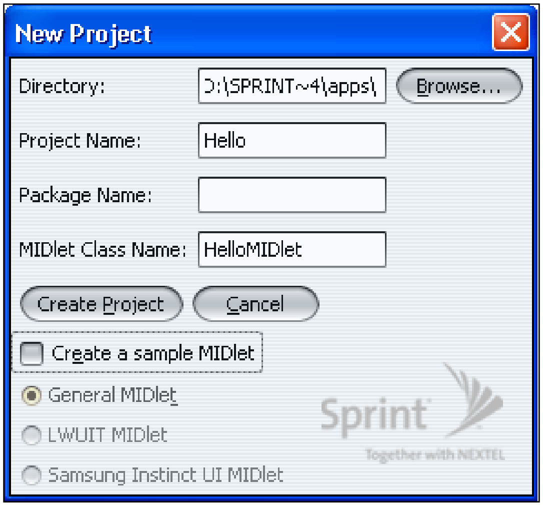
Our first sample application honors a longstanding tradition of writing a message starting with "Hello". It also has a simple animation showing an expanding circle. One alphabet of the message (**Hello LWUIT!**) appears on the screen at the end of each animation cycle. A **Replay** command allows the animation to be repeated.

Creating the project

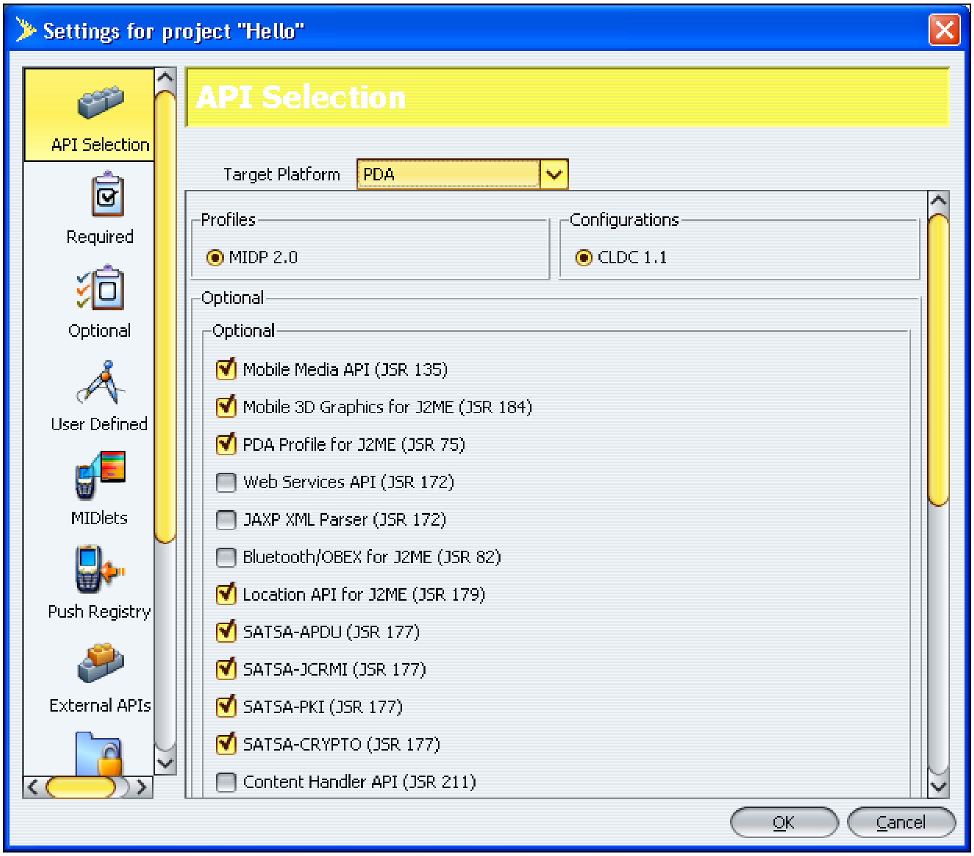
The first step for creating the project is, of course, to launch the SWTK via the **Start** menu by selecting **All Programs | Sprint Wireless Toolkit 3.3.2—Powered by Sun Java Technology | KToolbar**. When the toolkit opens, click on the **Console** tab, and this is what you will see:



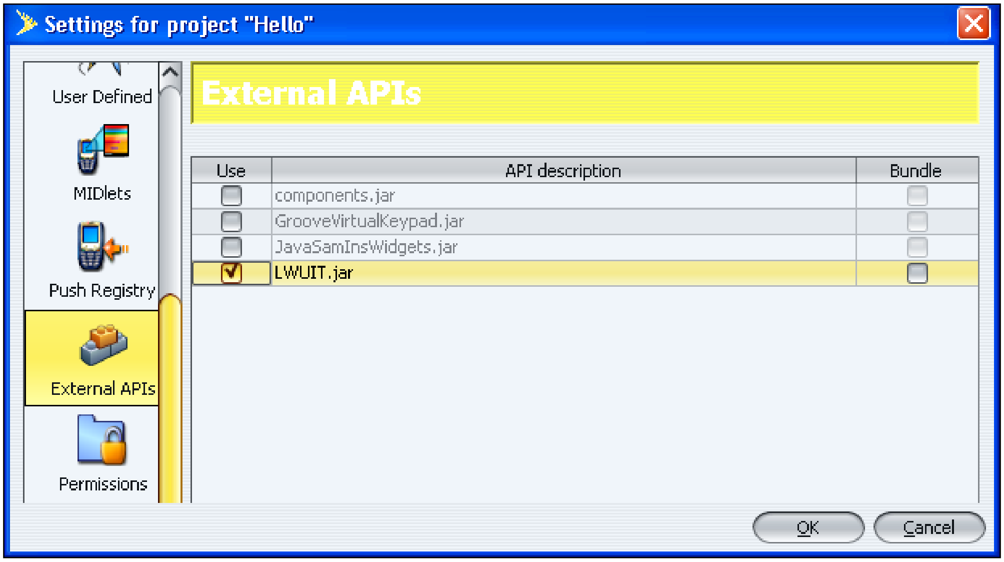
Click on the **New** icon. This will bring up the dialog shown in the following screenshot:



Don't make any changes in the **Directory** field. Fill in the **Project Name** and the **MIDlet Class Name** as shown. There are no *packages* specified in the code so the **Package Name** field has to be left blank. Now uncheck the **Create a sample MIDlet** box, as we plan to use our own MIDlet. Click on the **Create Project** button. The next screen that you will see is:



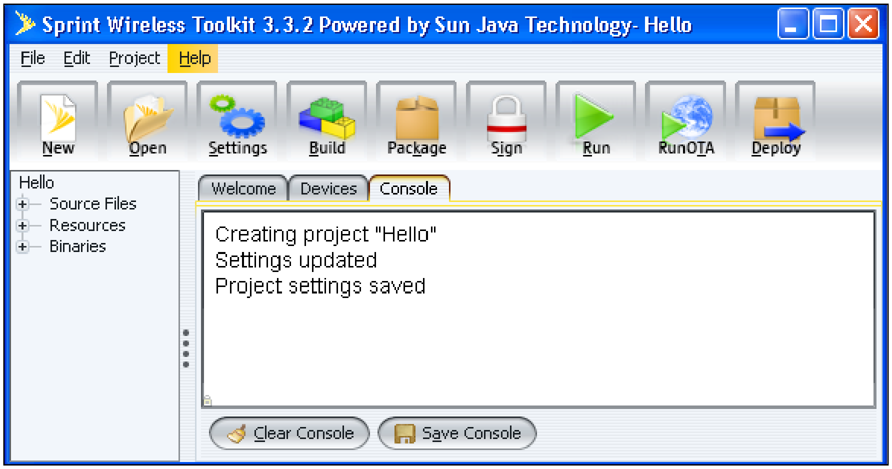
The **Target Platform** selected here is **PDA**, but this is really not very critical. Just make sure that the selected platform supports MIDP 2.0 and CLDC1.1, or else LWUIT applications will not compile. For this application at least, other API selections are not going to make any difference, and we will not make any changes. Now, click on the **External APIs** icon on the left panel to see the following screen.



Check if the **LWUIT.jar** file is listed. If it is, then select the file. If you find the file is missing, then you will have to load it manually. The JAR file can be found in the `LWUIT_HOME` directory. This has to be copied into the `SWTK_HOME\lib\ext` directory.

The box on the right column (titled **Bundle**) has been left unchecked. This is alright as long as the application runs only on the emulator. To run it on a real phone, this box has to be checked. Other aspects of MIDlet deployment have been explained in the next section.

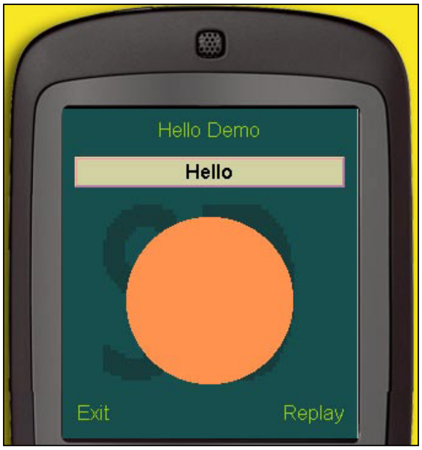
With **LWUIT.jar** selected, click on the **OK** button. This will take you to the screen confirming that the project has been created.



What we now have is an empty project. In order to make it a functional one, we need to load the source code of the project. All three code files (`HelloMIDlet.java`, `HelloForm.java`, and `HelloLabel.java` listed in the next section) can be typed or copied into any text editor. The code can also be downloaded from the download site of the book. The files must then be saved into the `src` folder for the project with a `.java` extension. In this case, the folder concerned is `SWTK_HOME\apps\Hello\src` folder. The `sdsym2.png` image file has to be downloaded from the download site of this book, and copied into the `SWTK_HOME\apps\Hello\res` folder.

The project can be built by clicking on the **Build** button. Once you get the **Build Complete** message on the console, you can run the demo by clicking on the **Run** button.

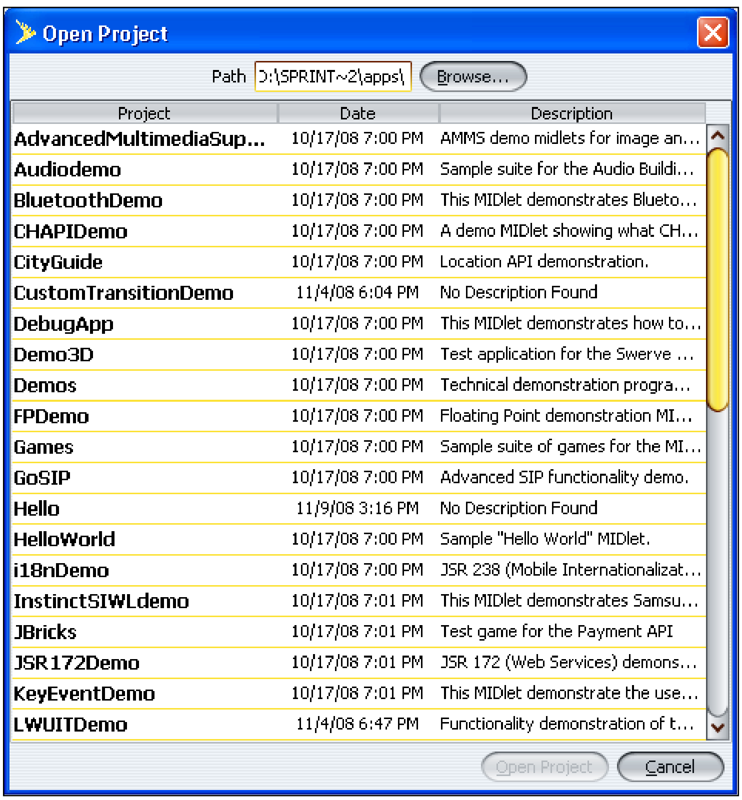
The following figure shows the application in mid-animation:



The final display will look like the following screenshot:

You can replay the animation, or exit the application by using the appropriate command. Note that a command can be selected by clicking on it, as the PDA emulator (HTC Touch) shown in the screenshot supports pointer action. For this device, the *F1* key acts as the left soft key ('*Exit*' here), and the *F2* key acts as the right soft key. You can also use these keys for selecting a command. If you select the **Replay** option, then remember that this command is effective only when the animation has been completed. Clicking on it while the animation is still going on will have no effect.

Once a project has been built, it can be seen on the **Project** list (as shown in the following screenshot) that is displayed when the **Open** icon is clicked on, and can be launched from there.



The code

The MIDlet code comes first:

```
import com.sun.lwuit.Command;
import com.sun.lwuit.Display;
import com.sun.lwuit.Font;
import com.sun.lwuit.Image;
import com.sun.lwuit.Label;
import com.sun.lwuit.events.ActionEvent;
import com.sun.lwuit.events.ActionListener;
import com.sun.lwuit.layouts.BorderLayout;
import com.sun.lwuit.plaf.Style;
import com.sun.lwuit.plaf.Border;
import javax.microedition.midlet.MIDlet;

public class HelloMIDlet extends MIDlet implements ActionListener
{
   private Label textLabel;//label to display the string
   private HelloLabel animLabel;//label for drawing animation
   private HelloForm helloForm;//form to hold the labels
   private boolean animationStopped;//indicates that animation has
                                    //been stopped

   public void startApp()
   {
            //initialise the LWUIT Display
      //and register this MIDlet
            Display.init(this);

      //create a HelloForm
      helloForm = new HelloForm(this, "Hello LWUIT!");

      //set form background image
      helloForm.getContentPane().getStyle()
     .setBgTransparency((byte)0);
      try
      {
          helloForm.getStyle().setBgImage
         (Image.createImage("/sdsym2.png"));
      }
      catch(java.io.IOException ioe)
      {
      }

      //font for title and menu bars
      Font font = Font.createSystemFont
                 (Font.FACE_SYSTEM,Font.STYLE_PLAIN,Font.SIZE_LARGE);

      //set title bar background and foreground
```

```
    helloForm.getTitleStyle().setBgTransparency(0);
    helloForm.getTitleStyle().setFgColor(0x99cc00);
    helloForm.getTitleStyle().setFont(font);

    //set menu bar background and foreground
    Style s = new Style();
    s.setBgTransparency(25);
    s.setBgColor(0x663366);
    s.setFgColor(0x99cc00);
    s.setFont(font);
    helloForm.setSoftButtonStyle(s);

    //Create animLabel and set attributes
    animLabel = new HelloLabel();
    animLabel.getStyle().setBgColor(0xd5d5d5);
    animLabel.getStyle().setBgTransparency(0);

    //add label to form
    helloForm.addComponent(BorderLayout.CENTER, animLabel);

    //create "Exit" and "Replay" commands and add them to form
    helloForm.addCommand(new Command("Exit", 0));
    helloForm.addCommand(new Command("Replay", 1));

    //set this form as the listener for the commands
    helloForm.setCommandListener(this);

    //show this form
           helloForm.show();

    //initialize animLabel and register it for animation
    animLabel.initialize();
    helloForm.registerAnimated(animLabel);
}

public void actionPerformed(ActionEvent ae)
{
        Command cmd = ae.getCommand();
        switch (cmd.getId())
    {
        //'Exit' command
        case 0:
            notifyDestroyed();
            break;

        //'Replay' command
        case 1:
            //restart animation only if it has been stopped
            if(animationStopped)
            {
```

```
            //re-initialize and resume animation
            animationStopped = false;
            helloForm.resetIndex();
             animLabel.initialize();
         restartAnimation();
             helloForm.registerAnimated(animLabel);
         }
      }
   }

   public void pauseApp()
   {
   }

   public void destroyApp(boolean unconditional)
   {
   }

   //restart animation
   public void restartAnimation()
   {
      animLabel.resumeAnimation();
   }

   //stop animation callback
   public void stopAnimation()
   {
      //deregister animLabel to stop getting animation callbacks
      helloForm.deregisterAnimated(animLabel);

      //set flag to show that animation has been stopped
      animationStopped = true;
   }
}
```

The MIDlet initializes the display instance, and creates the form. It also creates the two labels—the `textLabel` to display the message and the `animLabel` to show the animation, and adds them to the form.

The visual styles for the form and the labels are set within `HelloMIDlet`, and the commands are created and added to the form. Note that the MIDlet is the listener for both the commands, and its `actionPerformed` method takes the appropriate action for command execution.

The other tasks that are handled by the MIDlet relate to updating the message on `textLabel`, to stopping the animation, and to restarting it as required.

The form used in this demo is `HelloForm`, which is a subclass of Form. The code for this class is:

```
import com.sun.lwuit.Font;

import com.sun.lwuit.Form;
import com.sun.lwuit.Label;
import com.sun.lwuit.layouts.BorderLayout;
import com.sun.lwuit.plaf.Border;
import javax.microedition.midlet.MIDlet;

public class HelloForm extends Form
{
   private String helloString = "Hello!";//string to display
   private int index = -1;//index to access characters from string
   private HelloMIDlet midlet;
   private Label textLabel;

   //create new instance
   public HelloForm(HelloMIDlet m, String helloText)
   {
     super("Hello Demo");
      midlet = m;
      helloText = helloText.trim();
      if(!(helloText.equals("")))
      {
         helloString = helloText;
      }

      setLayout(new BorderLayout());

      //font for writing on textLabel
      Font font = Font.createSystemFont(Font.FACE_SYSTEM,
                        Font.STYLE_BOLD,Font.SIZE_LARGE);

      // Create textLabel and set attributes
      textLabel = new Label(" ");
      textLabel.setAlignment(Label.CENTER);
      textLabel.getStyle().setBorder(Border.createBevelRaised());
      textLabel.getStyle().setBgColor(0xcccc99);
      textLabel.getStyle().setFgColor(0x000000);
      textLabel.getStyle().setFont(font);

      addComponent(BorderLayout.NORTH, textLabel);
   }

   //reset the index to start a new cycle
   //and erase text
   public void resetIndex()
   {
```

```
         index = -1;
      textLabel.setText("");
      }
      public void updateText()
      {
      if(index == helloString.length() - 2)
         {
            //only one more character left
            //display entire message
            textLabel.setText(helloString);
            //and stop animation
            midlet.stopAnimation();
         }
         else
         {
            //whole string not yet written
            //update text and restart animation on anim_label
            textLabel.setText(getUpdatedText());
            midlet.restartAnimation();
         }
      }
      //called to get the next substring of helloString
      //if the next character is a space then the substring
      //keeps expanding until a non-space character is found
      private String getUpdatedText()
      {
         index++;
         //if index points to space character
         //recurse until non-space character is found
         if(helloString.charAt(index) == ' ')
         {
            return getUpdatedText();      }
         return helloString.substring(0, index+1);
      }
   }
```

The `HelloForm` class is responsible for synchronizing the printing of the message with the animation. Once the entire message has been shown, it asks the MIDlet to stop the animation.

While `textLabel` is instantiated from `Label`, `animLabel` is a customized version. The code for this class appears below:

```
import com.sun.lwuit.Container;

import com.sun.lwuit.Graphics;
import com.sun.lwuit.Label;

public class HelloLabel extends Label
{
   //decides which circle is to be drawn
   private int state;

   //time when previous repaint was done
   private long prevTime;

   private boolean done;
   private boolean initialized;

   //nominal interval between two successive repaints
   private final int interval = 150;

   //width of the label
   private int width;

   //height of the label
   private int height;

   //radius of first circle
   private int rad1 = 10;

   //radii of other three circles
   private int rad2, rad3, rad4;

   //top left corners of bounding rectangles - smallest to largest
   private int x1, y1, x2, y2, x3, y3, x4, y4;

   //create a new HelloLabel
   public HelloLabel()
   {
      super();
   }

   //if this object is registered for animation
   //then this method will be called once for every frame
   public boolean animate()
   {
      //painting parameters not set up
      //so don't repaint
      if(!initialized)
      {
         return false;
```

```
    }
    //get current time
    long currentTime = System.currentTimeMillis();
    //check if it's 'interval' milliseconds or more since last
    //repaint also see if all circles have been drawn
    if (!done && (currentTime - prevTime> interval))
    {
       //it's more than 'interval' milliseconds
       //so save current time for next check
       prevTime = currentTime;
       //increment state to draw next larger circle
       state++;
       //check if all circles drawn
       if (state == 5)
       {
          //all finished so set done flag
          done = true;
          //and ask for string display to be updated
          ((HelloForm)getComponentForm()).updateText();
       }
       //request repaint
       return true;
    }
    //either too soon for next repaint
    //or all circles drawn
    //no repaint to be done
    return false;
 }
 //reinitialize to start animation for next (non-space) character
 public void resumeAnimation()
 {
    state = 0;
    done = false;
 }
 //will be called whenever 'animate' method returns true
 public void paint(Graphics g)
 {
    //save the present color
    int color = g.getColor();
    //set color for drawing circles
    g.setColor(0xff8040);
```

```
        //act as per state value
        switch(state)
        {
            //draw smallest circle
            case 1:
                //translate to draw relative to label origin
                g.translate(getX(), getY());

                //paint the circle
                g.fillArc(x1, y1, 2*rad1, 2*rad1, 0, 360);

                //restore original co-ordinate settings
                g.translate(-getX(), -getY());
                break;

            //draw next larger circle
            case 2:
                g.translate(getX(), getY());
                g.fillArc(x2, y2, 2*rad2, 2*rad2, 0, 360);
                g.translate(-getX(), -getY());
                break;

            //draw next larger cirle
            case 3:
                g.translate(getX(), getY());
                g.fillArc(x3, y3, 2*rad3, 2*rad3, 0, 360);
                g.translate(-getX(), -getY());
                break;

            //draw largest circle
            case 4:
                g.translate(getX(), getY());
                g.fillArc(x4, y4, 2*rad4, 2*rad4, 0, 360);
                g.translate(-getX(), -getY());
        }

        //restore color
        g.setColor(color);
    }

    public void initialize()
    {
        //get dimensions of label
        width = getWidth();
        height = getHeight();

        //size of largest circle to be determined by
        //the shorter of the two dimensions
        int side = width < height? width : height;
```

```
        //find the center of the circle
        int centerX = width / 2;
        int centerY = height/2;

        //radius of largest circle
        //5 less than than half the shorter side
        rad4 = side/2 - 5;

        //difference between successive radii
        int radStep = (rad4 - rad1)/3;

        //radii of second and third circles
        rad2 = rad1 + radStep;
        rad3 = rad2 + radStep;

        //top left corners of the four bounding rectangles
        x1 = centerX - rad1;
        y1 = centerY - rad1;

        x2 = centerX - rad2;
        y2 = centerY - rad2;

        x3 = centerX - rad3;
        y3 = centerY - rad3;

        x4 = centerX - rad4;
        y4 = centerY - rad4;

        initialized = true;
      }
   }
```

`HelloLabel` takes care of the animation by drawing successively larger circles with a minimum interval of 100 milliseconds between two consecutive renderings.

The code for the three classes listed above will be discussed in detail in Chapter 11.

Deploying an application

When we use the **Build** button on the SWTK to compile an application, the required class files are generated. This allows the application to be executed on the SWTK. However, to deploy an application into an actual device, the class files cannot be used. What you need is a JAD file and a JAR file. To generate these files, use the **Package** button. The two files will be created and placed in the `bin` folder of the application. To load these files onto a phone, you will need the connecting cable and the software recommended by the device manufacturer. Usually, both of these come with the handset. In case you do not have the necessary hardware and software, you can get them from third-party vendors too. Handsets that support Bluetooth or infrared interfaces can load programs through these connections.

A second way of loading applications onto a phone is through the **Over-the-Air Provisioning (OTA)** function. This allows you to download an application from a remote server over the internet. On the internet, you can find many excellent tutorials on MIDlet deployment using OTA. You can look up the following article for more details:

"*Deploy MIDlets on J2ME-enabled devices*" by Soma Ghosh at `http://www.ibm.com/developerworks/edu/wi-dw-wi-devmid-i.html`.

Finally, don't forget to check the **Bundle** box on the **External APIs** screen. This will make sure that the LWUIT library is bundled with your application.

The Component class

We have already established a nodding acquaintance with the widgets. It is now time to get to know them more intimately starting with `Component`—the root. As long as a developer is working with just the widgets that come with the library, the `Component` class does not have to be accessed directly. It is only when we want to create our own components that we have to extend the `Component` class, and override the methods that would define the look-and-feel and the behavior of the custom component. However, even when we use only the built-in widgets, it is useful to have an understanding of what the `Component` class does, as it is the foundation for all widgets in the LWUIT library.

The only constructor of `Component` class is `protected Component()`. Therefore we cannot instantiate a component. However, we can subclass `Component` if we want.

The `Component` class contains methods that define the underlying functionalities that are common to all widgets. There are a number of methods to provide support for visual aspects of components. This is only natural, as appearance is a highly important factor for a widget. There are methods for handling user inputs and for the actual rendering of components. In the following sections, we shall list out the more important methods, and throughout this book, we will try out code using examples to illustrate the use of these methods.

Methods to handle size and location

One of the major issues involved in drawing a Widget is its size. This is an important factor for desktop versions too, but is much more critical for small devices like mobile phones. This is because the dimensions of the available display area vary widely, and the display screens are small. The following methods allow access to a component's size:

- `public void setSize(Dimension d)`
- `public void setPreferredSize(Dimension d)`
- `public Dimension getPreferredSize()`

The last two methods are meant for the use of developers. The `setPreferredSize()` method does not guarantee that the specified dimension will be adhered to. The final decision, in this regard, rests with the layout manager. The first method is used by the applicable layout manager to set the actual size that will be used in a given situation. This method should not be used by an application.

When the `getPreferredSize()` method is invoked, it may, if required, invoke another method to calculate the preferred size, based on the contents of the component. This is the `protected Dimension calcPreferredSize()` method. Applications can use this method to return preferred dimensions for a component, especially when a custom component is being created.

There are also methods to access individual dimensions of a component's size. These are:

- `public void setWidth(int width)`
- `public void setHeight(int height)`
- `public int getWidth()`
- `public int getHeight()`
- `public int getPreferredW()`
- `public int getPreferredH()`

Here again, the first two methods should not be used by developers, and are meant for layout managers only.

Another important consideration for laying out widgets is the location. So we have methods for setting the coordinates of the top-left corner of a widget, but these too are not to be used directly in an application. There is an interesting method—`protected Rectangle getBounds()`, that returns the bounds of a widget as a rectangle. A rectangle has four elements: X and Y coordinates of the top-left corner, width, and height. Calling this method gives us the location and the size of a widget in a single step.

Methods for event handling

A component also needs to have the ability to respond to user inputs. In a mobile device, the user may communicate with a widget, either through a keyboard or a pointer. LWUIT supports both modes, and the `Component` class has the following methods to handle key and pointer actions:

- `public void pointerDragged(int x, int y)`
- `public void pointerPressed(int x, int y)`
- `public void pointerReleased(int x, int y)`
- `public void keyPressed(int keycode)`
- `public void keyReleased(int keycode)`
- `public void keyRepeated(int keycode)`

The methods are very aptly named, and it is easy to understand their functions. The `keyRepeated` method needs clarification, as it works only for *Up*, *Down*, *Left*, and *Right* keys. By default this method just calls the `keyPressed` and `keyReleased` methods when any of the four keys listed above is held down. A subclass can override this method to provide any other functionality. The parameters that are passed when one of the first three methods is invoked represent the coordinates of the point at which the pointer action took place. Similarly, the parameter for the last three methods is the code for the key that was pressed.

Methods for rendering

Size and location data provide the basis for actual rendering of the components. There are a host of methods that perform various tasks related to drawing a component. A look at the list of these methods gives us our first idea about the intricacies involved in giving shape to a widget:

- `public void paint(Graphics g)`
- `protected void paintBorder(Graphics g)`
- `protected void paintBackground(Graphics g)`
- `public void paintBackgrounds(Graphics g)`
- `public void paintComponent(Graphics g)`
- `public void paintComponent(Graphics g, boolean background)`
- `protected void paintScrollbars(Graphics g)`
- `protected void paintScrollbarX(Graphics g)`
- `protected void paintScrollbarY(Graphics g)`

An understanding of the rendering process of a component in the LWUIT environment helps us to visualize how a widget gets built up on the screen. At this point, we take a brief detour to explore the rendering pipeline of a widget, and later in this chapter, we will familiarize ourselves with the all important `Graphics` class , which is the foundation for all painting activities.

The painting process

The painting of a component starts with clearing the background. This is done by erasing whatever was earlier drawn on that space. This is the background painting step which allows us to paint different backgrounds for widgets. If no specific background (like an image) is specified, then this step ends up with a background that is the same color as the container on which it is being drawn.

The next step is to draw the component itself. The paint method of the component is invoked for this. The usual practice is to delegate the actual painting to an instance of `LookAndFeel` via the `UIManager`. Let us suppose we have created our own widget—`OurOwnWidget`, and we want to paint it. We shall override the paint method to pass on the job of painting to the current `LookAndFeel` object. This is the code we shall write:

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

Obviously, the `LookAndFeel` object must implement the `drawOurOwnWidget` method.

`DefaultLookAndFeel`, the concrete subclass of `LookAndFeel` that comes with LWUIT, contains methods for drawing all standard widgets. For example, it has the method `drawButton` for buttons, `drawComboBox` for combo boxes and so on. This is the key to the **Pluggable Look And Feel** feature of LWUIT for customizing the look of a widget.

This customization can actually be done in two ways, as we noted earlier while introducing `LookAndFeel` in Chapter 1. One way is to override the appropriate draw method in `DefaultLookAndeel`. The other way is to plug a completely new subclass of `LookAndFeel` into `UIManager`.

The second approach is not really a very practical one, as it will mean writing our own draw methods for all the widgets. The preferable approach would be to extend `DefaultLookAndFeel`, and override an existing method, or add a new one, as required. In this case, we would extend `DefaultLookAndFeel`, and add a method to render `OurOwnWidget`—`public void drawOurOwnWidget(Graphics g, OurOwnWidget oow)`.

Then the new version of `DefaultLookAndFeel (MyLookAndFeel)` can be installed in the following way:

```
UIManager.getInstance().setLookAndFeel(new MyLookAndFeel());
```

There is only one instance of `UIManager` per application, as we saw in Chapter 1. We cannot create an instance of `UIManager`. The only way to get a reference to this object is to invoke the static method `UIManager.getInstance()`. We can then use the `setLookAndFeel(LookAndFeel plaf)` method to install the desired instance of `LookAndFeel`.

Instead of using one of the approaches outlined above, we can obviously let the rendering be done by the component itself. If we are planning to distribute our component, this would be the preferable technique as the component will be self-sufficient, and the user of the component will not have to plug in a new `LookAndFeel`.

The final step is to paint any border that the component might have. Painting of the border is done by the `paintBorder(Graphics g)` method of `Component` class, which in turn, calls the `paint(Graphics g, Component c)` method of the `Border` clss. We shall see how borders are handled for widgets in Chapter 4.

Note that all these steps are executed automatically by LWUIT, and the relevant methods are invoked in proper sequence. However, the order of the activities described may get modified, as some borders take over the responsibility of background painting.

Miscellaneous methods

In addition to the methods that are listed above, the `Component` class contains many others that support a widget's functionalities. For example, consider the `initComponent()` method. This method can be used to set up attributes and to initialize variables or states to make a component ready to go.

The `Component` class supports a unique identifier for each component. This identifier is used to apply a style to a component. The identifier can be accessed using the `protected String getUIID()` method. All subclasses of the `Component` class must override this method, and return the identifier that is used for setting a style to that component.

As we work our way through the examples in this book, we shall become familiar with the methods described here and their applications.

Animation support for components

The `Component` class implements the **Animation** interface, making all components capable of being animated. The method that provides the basic support for animation is `boolean animate()`. This method is called once for every frame, and if it returns true, then a repaint is performed. The paint method then ensures that a new frame is drawn to visually implement the animation. The obvious advantage of this approach is that a repaint is not asked for unless it is really required, thus minimizing painting operations, which in turn, optimizes processor utilization. The `HelloLabel` class uses this method to request a repaint at proper times.

Handling Style

The `Style` class holds the attributes required to determine the look of a widget. When a widget is created, a `style` object is automatically generated, and this ensures that every widget has a `style` object associated with it. The two methods that allow a component to access its `style` object are:

- `public Style getStyle()`—gets the `style` object for this component
- `public void setStyle(Style style)`—sets a new `style` object for this component

`HelloMIDlet` makes extensive use of these methods to specify the appearances of the form and the labels.

The Graphics class

In order to draw a widget, painting methods use an instance of the platform graphics context, which is abstracted by the `Graphics` class. This class cannot be instantiated, and the only way to obtain an instance is through a paint callback, or by using the `getGraphics()` method of a **mutable image**. Incidentally, a mutable image (as opposed to an **immutable image**) is an image that can be modified.

The `Graphics` class provides the tools for drawing patterns and images. For instance, if you want to draw a line between two points, then you would use the `drawLine(int x1, int y1, int x2, int y2)` method of this class. This method draws a straight line between two points whose coordinates are (x1, y1) and (x2, y2) respectively. There are similar methods for drawing a wide range of geometrical shapes, either in the form of an outline, or filled with a color. There are also methods for drawing images and textual strings. There are appropriate accessor methods for the colors, fonts, and other attributes that support the rendering process.

Two very important functions of the `Graphics` class are to set a clip region and to translate coordinates.

A clip region defines a part of a pattern that is to be drawn. Let us say that we are dealing with an image that is 500 x 500 pixels in size. If we want to draw just a 50 x 50 pixel portion of the image, then we can use the `setClip(int x, int y, int width, int height)` method of the `Graphics` class to select a rectangular part with its top-left corner at coordinates (x, y) and with the specified width and height.

The translate capability shifts the position for drawing a pattern. This can be useful when successive frames draw a figure at different positions. Assume that a figure is to be moved to the right of the screen, with a displacement of dx pixels per frame. In this case, we can use the translate method like this: `g.translate(dx, 0)` where `g` is the graphics context for drawing, and *dx* is the required displacement to the right. The second parameter is zero, as we do not want any vertical movement. Incidentally, you can see this method being used in the code listing for `HelloLabel` class above.

Summary

The main topics of this chapter were an introduction to building an application with the SWTK and also to the `Component` class. In addition, we studied some other topics too. The following is a list of what we have studied in this chapter:

- What to download, and their sources
- How to set up, build, and run a project using the SWTK
- The `Component` class
- Different types of methods in the `Component` class
- The painting process of a component
- The `Graphics` class
- Animation support in the `Component` class
- How the `Component` class handles `Style`

3
The Container Family

A **Container** is a component that is designed to hold other components. There are several components that are members of the container family. These are the Form, the Dialog, the Calendar and the TabbedPane. As a container is itself a component, we can add one container into another. If we have two groups of components, then we can add them to two separate containers, and add these two containers to a third one. We can then add the third container to fourth, the fourth to fifth and so on. This nesting ability is very useful as it allows us to develop very intricate arrangements for components on a form.

Although our primary focus in this chapter is on `Container` and its descendants, we shall also spend some time on the `Command` and the `Font` classes, as they are very important aspects of widget look-and-feel. So our agenda will be:

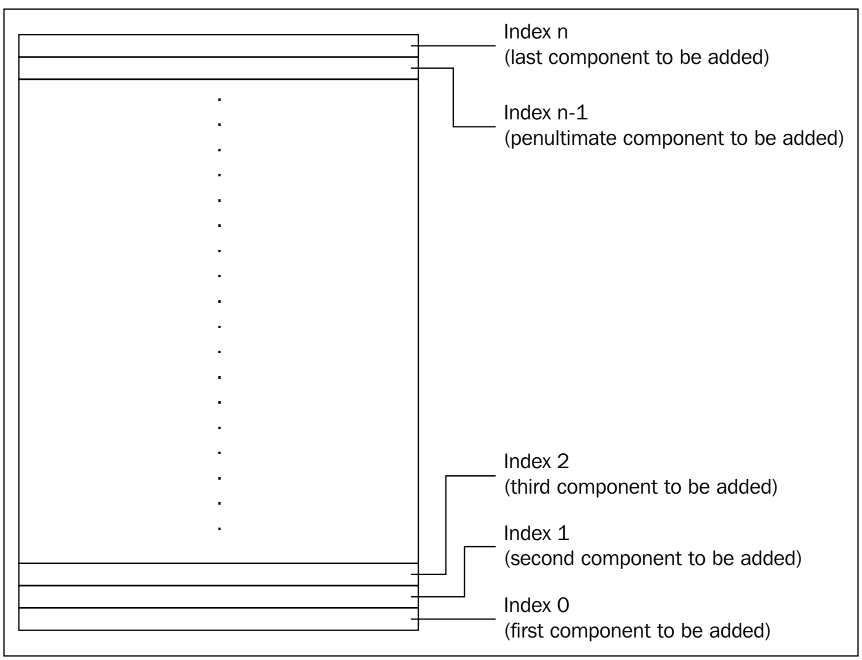
- Learn about the `Command` and the `Font` classes.
- Learn about the `Container` class.
- Learn how to create a form and use its various capabilities. As we study the different aspects of a form, we shall progressively apply them on a demo application.
- Familiarize ourselves with handling `Style` and `Command` through the example above.
- Learn about Dialogs and how to use them by building a demo.
- Learn about the `Calendar` class and build a simple calendar demo.
- Learn about TabbedPanes and build a demo.

LWUIT is an evolving library, and we have to keep up with the changes that will be announced from time to time. At the time of writing, one of the changes expected to be incorporated in the next code drop involves `Style`. In this chapter, we will get a preview of these impending modifications.

The Container

The primary function of a container is to hold components. Since a container itself is a component, it allows nesting of containers. It also allows layout managers to arrange components in specific ways.

The `Container` class maintains a list of all components that it holds in a `java.util.Vector` object. The index of a component in the vector defines its position in the stacking order within the container. Unless specified, a component to be added to a container occupies the last position in the vector. It is possible to insert a component into a given position by using a method that allows us to specify an index. This is discussed further in the section on *Calendar.* The following figure shows how components are stacked within a container:



Creating a Container

There are two constructors for creating a container. They are:

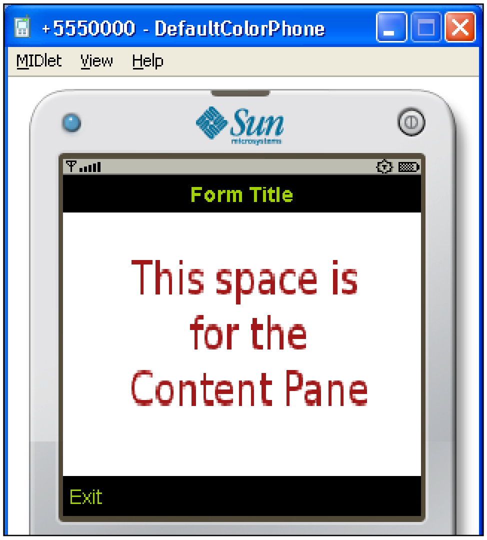
- `public Container(Layout layout)`—creates a container with the specified layout manager.
- `public Container()`—creates a container with `FlowLayout` as the layout manager. A layout manager is responsible for positioning components within a container. Layout managers will be discussed in Chapter 7.

The methods of the Container class

The methods of `Container` class are naturally oriented towards adding or removing components and towards managing the components within it. As we build and analyze the demo applications in this book (starting from the next section), we shall become familiar with the use of methods. So, without any further ado, let's roll up our sleeves and get the action going.

The form

A form is a top level container with a title bar and a menu bar. The contents of the form are placed between the two bars. Reproduced below for reference, we see the screenshot of a form. It shows the place where the title of the form appears (TitleBar) and the place for commands (MenuBar) where the **Exit** command has been placed. Also shown is the space for the **Content Pane**, which actually holds the entire set of components that are added to the form.



Creating a form

The `Form` class has two constructors:

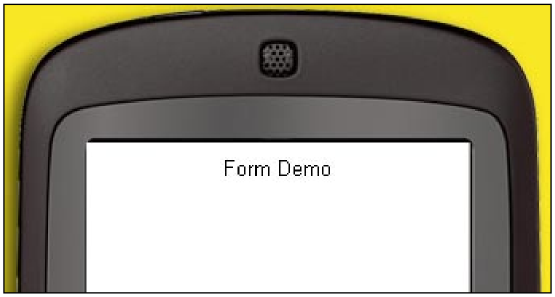
- `public Form()`–creates a form without any title. If you use this constructor, then you can later set the title. However, if no title is specified at all, then the form created will not have a TitleBar. The default layout manager of a form is `BorderLayout`.
- `public Form(String title)`– creates a form with the specified title.

We will create a form and try out the topics that we discuss as we go along. We will use the second constructor for our form. The code for the MIDlet is:

```
import com.sun.lwuit.Display;
import com.sun.lwuit.Form;
import javax.microedition.midlet.MIDlet;
public class DemoForm extends MIDlet
{
   public void startApp()
   {
      //initialize the LWUIT Display
      //and register this MIDlet
      Display.init(this);
      //create a new form
      Form demoForm = new Form("Form Demo");
      //show the form
      demoForm.show();
   }

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

At this stage, the code is very simple. All the action takes place in the `startApp()` method. The first thing to be done is to initialize the display and register the MIDlet. Remember that invoking the `init` method before doing anything else is essential for all LWUIT applications. We then create the form, and in the last line of code, we call the `show()` method to display the form.

We now have a form with just a title, as the following screenshot shows. It does not even have an **Exit** command. So, to exit from the demo you will have to close the window. Next, let us add a command to the form so that we have an appropriate way of closing the application.



Handling commands

Now it is time to add an **Exit** command to our form. To add commands to a form and to enable it to handle them, we need to do the following:

- Create the commands.
- Add the commands to the form.
- Add a listener for the command. In this case, the MIDlet will be our listener. So the MIDlet will have to implement the `ActionListener` interface.
- Write the `actionPerformed(ActionEvent ae)` method—the only method of `ActionListener`.

Before we actually install a command on this form, let's familiarize ourselves with the `Command` class.

The Command class

A command represents an action that can be taken by a user and can be placed on a soft button or in a menu. In the first screenshot of this chapter, the **Exit** command has been placed on the left soft button. Had there been a second command, it would have been placed on the right soft button, which is situated at the bottom-right corner of the screen. When there are more than two commands, the first command to be added to the screen goes on the left soft button and the rest are placed in a menu. The word **Menu** would then appear on the right soft button. There is an optional three button mode, which we will try out later in this chapter, when we add commands to a demo form.

Creating a command

A command has three attributes:

- A String that represents the name of the command.
- An image used as an icon for the command. This is an optional item. If you use this, then make sure that you get the size of the image right. For example, if it is a very large image, then the menu bar will be disproportionately high.
- An ID. This too is optional. It provides a convenient way of writing the `actionPerformed(ActionEvent ae)` method, as we shall see later in this chapter.

We can create a command with only the first attribute, or with one or both of the optional ones. The constructors are:

- `public Command(String command)`
- `public Command(.String command, Image icon)`
- `public Command(String command, int id)`
- `public Command(String command, Image icon, int id)`

Methods of Command class

The methods of this class that will be used frequently are:

Method	Parameters	Returns
`String getCommandName()`		the command name
`Image getIcon()`		the image (icon) representing the command
`int getId()`		the command `id`

Installing a command

The code listing below shows the revised DemoForm.

```
import com.sun.lwuit.Display;
import com.sun.lwuit.Command;
import com.sun.lwuit.Form;
import com.sun.lwuit.events.ActionEvent;
import com.sun.lwuit.events.ActionListener;
import javax.microedition.midlet.MIDlet;

public class FormDemoMIDlet extends MIDlet implements ActionListener

{
   public void startApp()
   {
      //initialize the LWUIT Display
      //and register this MIDlet
      Display.init(this);

      //create a new form
      Form demoForm = new Form("Form Demo");

      //create and add 'Exit' command to the form
      //the command id is 0
      demoForm.addCommand(new Command("Exit", 0));

      //this MIDlet is the listener for the form's command
```

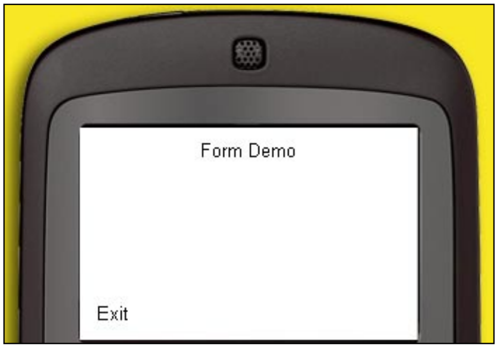
```
        demoForm.setCommandListener(this);

        //show the form
        demoForm.show();
    }
    public void pauseApp()
    {
    }
    public void destroyApp(boolean unconditional)
    {
    }
    //act on the command
    public void actionPerformed(ActionEvent ae)
    {
        Command cmd = ae.getCommand();
        switch (cmd.getId())
        {
            //'Exit' command
            case 0:
                    notifyDestroyed();
        }
    }
}
```

We can see from the highlighted code that all the necessary steps for adding a command have been implemented. The `actionPerformed(ActionEvent ae)` method shows how the command `id` simplifies the structure of the method. As the `id` is an `int`, a `switch` statement can very conveniently identify a command and take proper action. An alternative approach would be to get and check the name of the command like this:

```
//get command name
String cmdName = cmd.getCommandName();
//if name is 'Exit' the close app
if(cmdName.equals("Exit"))
{
    notifyDestroyed();
}
```

I like to work with command `id`, as `case 0:` is shorter to type than an `if` statement. However, we must keep in mind that `id` is optional, and the default value is `0`. Using the approach based on `id` will introduce a bug if `id` is not specified in the constructor, as all commands with the default `id` will initiate the action corresponding to `case 0`. On the other hand, the command name is a mandatory parameter. So the approach based on command name works properly, regardless of the constructor used to create commands, as long as names are not duplicated.

The following screenshot shows the demo form with an **Exit** command that can be used for closing the emulator:



If you are building your application for a device that supports three soft buttons, then you can use the third soft button. This is done either by programmatically setting the `thirdSoftButton` flag in the `Display` class to `true`, or by adding an `isThirdButtonSupported` user-defined property to the project, and setting its value to `true`.

For the first approach, add `Display.getInstance().setThirdSoftButton(true)` just after the `Display.init` method call. Now, when you add commands, the first command will be added to the center soft button, the second one to the left and the third one to the right. If there are more than three commands, then all the commands from the third onwards will be added to a menu, and the command **Menu** will appear on the right soft button.

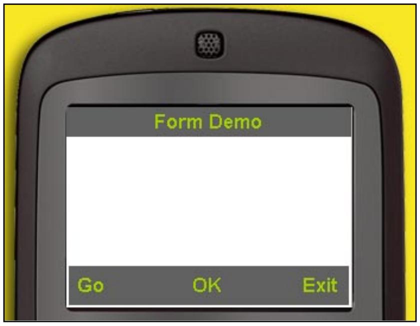
If you want to set the option for the third soft button through the project settings, then click on the **Settings** button on the SWTK, and then select the **User Defined** icon. Click on the **Add** button to get the following dialog:

Enter **isThirdSoftButtonSupported** as the **Property Name** and **true** as the **Property Value**. Now this property will be incorporated in the **Java Application Descriptor** (**JAD** file) for the project.

Assuming that the third soft button mode has been set, let us add a couple of dummy commands:

```
//create and add two dummy commands
demoForm.addCommand(new Command("OK"));
demoForm.addCommand(new Command("Go"));

//create and add 'Exit' command to the form
//the command id is 0
demoForm.addCommand(new Command("Exit", 0));
```

The following screenshot shows that the commands have been added as expected:

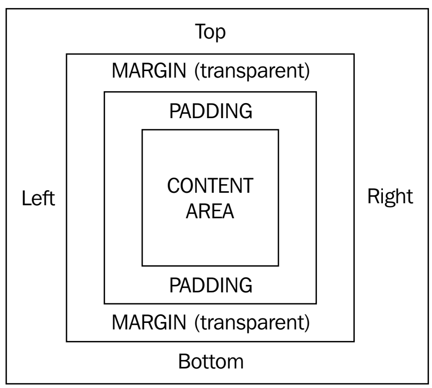


Managing the form's appearance

There are a number of attributes that determine the look of a component. These are:

- Background and foreground colors—each component has four color attributes: two each for background and foreground. A component is considered selected when it is ready for activation. When a button receives focus, for example, it is in the selected state, and it can be activated by being clicked on. A component can have a background color for the selected state and another for the unselected one. Similarly, the foreground color (usually the color used for the text on the component) can be individually defined for the two states.
- Fonts for writing text on it—text can be rendered using the standard font styles, as supported by the platform, as well as bitmap fonts. The font for each component can be set individually.

- Background transparency—the transparency of a component's background can be set to vary from fully opaque (the default setting) to fully transparent. The integer value 0 corresponds to full transparency and 255 to complete opacity.
- Background image—by default, the background of a component does not have any image. However, this setting can be used to specify an image to be used as the background.
- Background painters—background painters can be used to customize the background of one or of a group of components. We shall learn more about background painters in Chapter 12.
- Margin and padding—the visual layout of a component defines the margin and padding. This is depicted in the following figure. LWUIT allows *margin* and *padding* for each of the four directions (top, bottom, left, and right) to be set individually.



When we created the demo form, a `Style` object was automatically created and set for this form. The default values of all the attributes mentioned above for the form are held within this `Style` object. They are responsible for the form's appearance. Obviously, these default values are not very attractive, and we must modify them to improve the way our form looks.

From a visual perspective, a form has three distinct parts:

- TitleBar
- ContentPane
- MenuBar

Each of these parts have a style, and the attributes of all the three styles will have to be set with our selected values. Let's start with the title bar and see how to handle its style. The first attribute that we shall change is the background color of the title bar.

Setting the TitleBar's looks

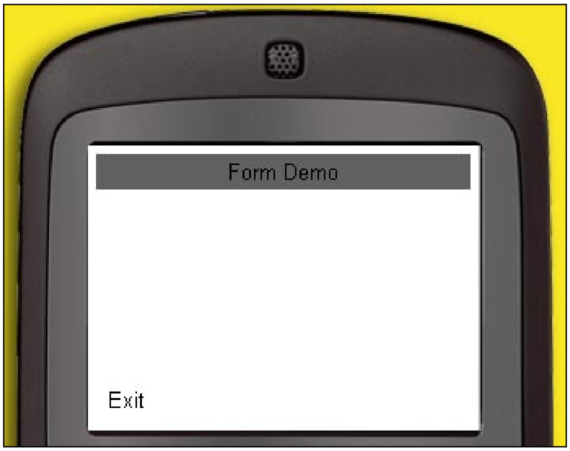
A component's style object is not directly available, and we have to use the `getStyle()` method of a component to get a reference to its style. However, the title bar is a part of the form, and the `Form` class provides the `getTitleStyle()` (and a `setTitleStyle()`) method, which has been used here. The `Style` class has methods to access the attributes. Once we have the reference to a `style` object, we can use its accessor methods to get or set attribute values. To change the background color of the title bar, we must add the following line to `demoForm`:

```
//set background color for the title bar
demoForm.getTitleStyle().setBgColor(0x555555);
```

The parameter for setting the background color is in the standard RGB format, and it is expressed in hexadecimal `int` form. The color can also be expressed as an equivalent decimal `int`, which would be `5592405` for this color.

We could have used an alternate approach for setting the title bar style attributes. The `getTitleComponent()` method of `Form` returns the component (a label actually) that forms the title. We can then set the style of the title bar directly. So the above line of code can be replaced by `demoForm.getTitleComponent().getStyle().setBgColor(0x555555)` with the same result.

Now, let's see what the form looks like.



We have a nice dark title bar, but the title itself has almost disappeared. In order to make it visible again, we need to change the font color. We will take this opportunity to change the font style entirely by installing a new font. But first, here is a short introduction to the `Font` class.

The Font class

The `Font` class is an abstraction of fonts available on the platform, as well as of those non-native fonts that are bundled with the application.

Creating a Font

A `Font` is an abstract class and cannot be directly instantiated. To get a font other than the default native font, you have to use a static method of the class. If you are going to use a font derived from the native font (system font as it is called), then the method to use is `createSystemFont(int face, int style, int size)`. The parameters to the method can take the following values:

- face—one of `Font.FACE_SYSTEM`, `Font.FACE_PROPORTIONAL`, `Font.FACE_MONOSPACE`
- style—one of `Font.STYLE_PLAIN`, `Font.STYLE_ITALIC`, `Font.STYLE_BOLD`
- size – one of `Font.SIZE_SMALL`, `Font.SIZE_MEDIUM`, `Font.SIZE_LARGE`

There are methods for creating new bitmap fonts too, but these are not very convenient to use. It is far easier to use the LWUIT Designer that comes with the LWUIT library. In Chapter 9, we shall learn how to create and edit bitmap fonts using the LWUIT Designer.

The methods of the Font class

In addition to the methods for creating new fonts, this class provides a number of methods that make it very convenient to work with fonts. The ones that we shall need to use often are:

Method	Parameters	Returns
`int charsWidth(char[] ch, int offset, int length)`	`ch`—array of characters `offset`—the starting index `length`—number of characters	The sum of the widths of characters from the given array for the instance font starting at offset and for the number of characters given by length.
`abstract int charWidth(char ch)`	`ch`—the specified character	The width of the specified character when rendered alone.
`int stringWidth(String str)`	`str`—the given String	The width of the given String for this font instance.
`int substringWidth(String str, int offset, int length)`	`str`—the given String `offset`—the starting index `length`—number of characters	The width of a substring of the given string for the instance font starting at offset and for the number of characters given by length.
`abstract int getHeight()`		Returns the total height of a character for the instance font.
`static Font getDefaultFont()`		Returns the global font instance that will be used by default.
`static void setDefaultFont(Font f)`	`f`—the font to be set as default.	Sets the global font instance that will be used by default.
`int getFace()`		Returns the type of font face for the instance font.
`int getStyle()`		Returns the style for the instance font.
`int getSize()`		Returns the size for the instance font.

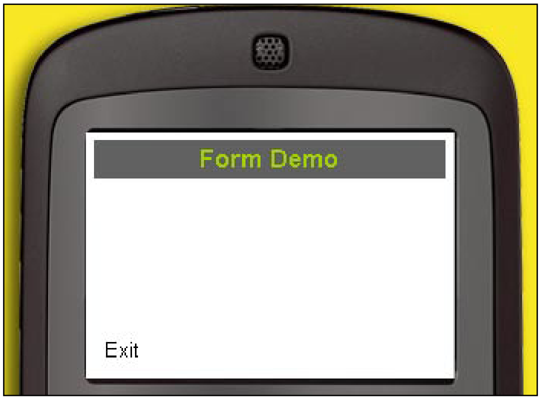
The `Font` class also provides methods that can be used only in the context of bitmap fonts. We shall discuss these methods in Chapter 9.

Installing a new font

The changes to the MIDlet code for installing a new font for the title are:

```
//create a font
Font font = Font.createSystemFont(Font.FACE_PROPORTIONAL,
                        Font.STYLE_BOLD,Font.SIZE_LARGE);
//set text color for title
demoForm.getTitleStyle().setFgColor(0x99cc00);
//set font style for title
demoForm.getTitleStyle().setFont(font);
```

And the changed look is as shown in the following screenshot:



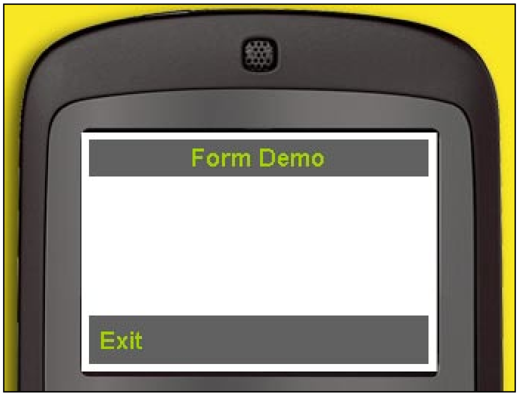
Setting the MenuBar's looks

After modifying the appearance of the title bar, let's do the same to the menu bar. This time though, we shall use a different approach. We shall create a brand new style object, with the attributes that we want, and set it for the menu bar. The point to be kept in mind is that the method to be used now is `setSoftButtonStyle(Style s)` method of the `Form` class and not the (once used but now deprecated) `setMenuStyle(Style s)` method.

```
//create a new style object
Style menuStyle = new Style();
//set the background color -- the same as for title bar
menuStyle.setBgColor(0x555555);
//set the text color for soft button
menuStyle.setFgColor(0x99cc00);
//set font style for soft button
menuStyle.setFont(font);
//now install the style for soft button
demoForm.setSoftButtonStyle(menuStyle);
```

The constructor used here for instantiating `Style` does not take any parameters. The `Style` class has other constructors too, which we shall use in some of the examples later.

We can see the result of setting the style to a soft button below.



Setting the Form's Looks

Now, let's set a background color for the rest of the form. By this time, we know exactly what to do—we have to write a line of code like this:

```
//set a background color for the form
demoForm.getStyle().setBgColor(0x656974);
```

When we do that, the result is as shown in the following screenshot:

There are other things that we can do with the way a form (and any other component) looks and behaves, and in the rest of the book, we shall progressively try them out. For example, we shall see how to use borders in Chapter 4. Just keep in mind that the technique for setting style attributes is the same for all components. Another point worth noting is that even if a particular attribute is not applicable to a component, trying to set it is generally not going to cause any problem. So you can go ahead and set the foreground selection color for the title bar. Although this attribute is not relevant to the title bar, as it does not receive focus, it is not going to do any harm.

The Dialog

The `Dialog` class extends `Form`, and it is used to present information to the user, as well as to collect the user's response to it. While a dialog is very similar to a form, there are some differences as well. One very significant difference in the way these two widgets work is that Dialog is modal by default. **Modality** means that the thread calling a dialog will block until the `dispose()` method of the dialog is called. Another implication of modality is that the user's response can be assumed to be available in the line of code, right after the `show()` method is called.

A dialog is also different from the form in size. The dialog occupies only a part of the screen. The following screenshot shows a dialog with an **Exit** command. In this case, there is no form in the background, and the dialog is the only component in the demo. This is very unlikely in a real application, but then this one is only to show you how a dialog works.

Creating a Dialog

The `Dialog` class has the same kind of constructors as `Form`:

- `public Dialog()`—creates a dialog without any title. If you use this constructor, then you can later set the title. However, if no title is specified at all, then the dialog created will not have a TitleBar.
- `public Dialog(String title)`—creates a dialog with the specified title.

The methods of the Dialog class

When we look at the documentation of Dialog, the first thing that strikes us is the large number of *show* related methods. There are twelve static methods that enable us to create and show a dialog with a single statement. An example is the `public static Command show(String title, Component body, Command[] cmds)`. This creates and shows a modal dialog with the given title, the component as its body, and with the commands specified in the command array. When the dialog is closed, the command pressed is returned. We shall use many of these variants in our demos.

Dialog also has six non-static show methods

Method	Parameters	Returns/Does
`void show()`		Shows a modal dialog at the center of the screen.
`void showModeless()`		Similar to the method above, but shows a modeless (non-modal) dialog.
`Command showPacked(String position, boolean modal)`	`position`—one of the constraints defined in the `BorderLayout` class, and it determines the position of the dialog on the screen. `modal`—whether the dialog should be modal or modeless.	Shows a dialog and returns the command used to close the dialog. The dialog shown will be sized to match its contents.

Method	Parameters	Returns/Does
`Command show(int top, int bottom, int left, int right, boolean includeTitle)`	`top`—space in pixels between the top of the screen and the dialog. `bottom`—space in pixels between the bottom of the screen and the dialog. `left`—space in pixels between the left of the screen and the dialog. `right`—space in pixels between the right of the screen and the dialog. `includeTitle`—whether the title should hang at the top of the screen or be attached to the dialog.	Shows a modal dialog with the specified margins on four sides. If the last parameter is false, then the title appears "hanging" on the top of the screen, and is detached from the dialog body. If it is `true`, then the title appears on the body as usual. Returns the command that is pressed to close the dialog.
`Command show(int top, int bottom, int left, int right, boolean includeTitle, boolean modal)`	`top`—space in pixels between the top of the screen and the dialog. `bottom`—space in pixels between the bottom of the screen and the dialog. `left`—space in pixels between the left of the screen and the dialog. `right`—space in pixels between the right of the screen and the dialog. `includeTitle`—whether the title should hang at the top of the screen or be attached to the dialog. `modal`—whether the dialog should be modal.	Shows a dialog with the specified margins on four sides. If the last parameter is `false`, then the title appears 'hanging' on the top of the screen, and is detached from the dialog body. If it is `true`, then the title appears on the body as usual. If the last parameter is `true`, then the dialog is modal. Returns the command pressed to close the dialog.
`Command showDialog()`		Shows a modal dialog, and returns the command that is pressed to close the dialog.

- To close the dialog, the `dispose()` method has to be called. An interesting feature of the `Dialog` class is its `AutoDispose` function. With this function enabled, the execution of a command on the dialog will cause the dialog to be disposed. By default, the function is enabled. The disposal related methods are:
    - `public void dispose()`—closes the dialog, and returns to the previous form. This action releases the calling thread.
    - `public void setAutoDispose(boolean autoDispose)`—enables or disables the `AutoDispose` function.
    - `public boolean isAutoDispose()`—shows whether the `AutoDispose` function is enabled or not.
    - `public void setTimeout(long time)`—sets the time (in milliseconds), after which the dialog is automatically disposed.

The last method is useful for displaying an alert that does not require any user interaction.

Displaying a dialog

The following listing creates and shows a dialog:

```
import com.sun.lwuit.Display;
import com.sun.lwuit.Dialog;
import com.sun.lwuit.Command;
import com.sun.lwuit.Font;
import com.sun.lwuit.events.ActionEvent;
import com.sun.lwuit.events.ActionListener;
import com.sun.lwuit.plaf.Style;
import javax.microedition.midlet.MIDlet;
public class DemoDialog extends MIDlet implements ActionListener
{
  private Dialog demoDialog;
  public void startApp()
  {
    //init the LWUIT Display
       Display.init(this);
    //create a new dialog with the given title
    demoDialog = new Dialog("Dialog Demo");
    //create a font
    Font f = Font.createSystemFont(Font.FACE_SYSTEM,
                   Font.STYLE_BOLD,Font.SIZE_LARGE);
    //set background color for the dialog
    demoDialog.getContentPane().getStyle().setBgColor(0xff8040);
    //set colors and font for dialog titlebar
```

```
        demoDialog.getTitleStyle().setBgColor(0xa43500);
        demoDialog.getTitleStyle().setFgColor(0xffffff);
        demoDialog.getTitleStyle().setFont(Font.createSystemFont(
             Font.FACE_SYSTEM,Font.STYLE_BOLD,Font.SIZE_MEDIUM));
        //create a new style and set it for dialog menu bar
        Style s = new Style();
        s.setBgColor(0xa43500);
        s.setFgColor(0xffffff);
        s.setFont(f);
        demoDialog.setSoftButtonStyle(s);
        //let us not close the dialog when its command is executed
        demoDialog.setAutoDispose(false);
        //add the <Exit> command to close the dialog and the MIDlet
        demoDialog.addCommand(new Command("Exit"));
        //make this MIDlet the listener for the command
        demoDialog.setCommandListener(this);
        //show the dialog
        demoDialog.show();
       }

      public void pauseApp()
      {
      }
      public void destroyApp(boolean unconditional)
      {
      }
      //act on the command
      public void actionPerformed(ActionEvent ae)
      {
        //there is only one command
        //so no need to check what command is selected
        demoDialog.dispose();
          notifyDestroyed();
      }
    }
```

The dialog is created with a title. We then set style attributes just as we did with the form earlier. The only difference that needs to be noted is that, for a dialog, direct style attributes such as the background color should be applied to the content pane, rather than to the dialog itself. The auto dispose function has been disabled so that we can also close the application when we react to the command in the MIDlet. The rest of the code is similar to the demo application of Form. Another thing to be noted is that the `actionPerformed(ActionEvent ae)` method shows a different way of acting on a command, when the form has only one command to listen to.

The Calendar

The `Calendar` class represents a widget for displaying dates. `Calendar`, like `Form`, is also a subclass of `Container` and inherits its characteristics.

The default display format shows one month at a time, with the current date highlighted. This view of the month is provided by the `MonthView` class, which can be accessed only from within the `com.sun.lwuit` package. This means we cannot use `MonthView`, nor can we see it in the API documentation.

Creating a Calendar

A `calendar` object can be instantiated by using one of the two constructors of the `Calendar` class. The constructors are:

- `public Calendar()`—the `calendar` object created will be initialized to the current date and time. `BorderLayout` is the default layout manager for `Calendar`.
- `public Calendar(long time)`—the parameter is the time in milliseconds since the Epoch, that is, January 1, 1970 00:00:00.000 GMT (Gregorian). The `calendar` object created will be initialized to the date and time corresponding to the `time` parameter.

When we create a calendar, a `MonthView` object is created to display a tabular view of the month.

Methods of Calendar class

This class provides just a few methods that are specific to its functioning. The methods that allow us to get and set the selected date are:

- `public Date getDate()`—returns the currently selected date in the specified format. For example, a return value could be `Sat Nov 22 13:26:36 GMT -08:00 2008`.
- `public void setDate(Date d)`—sets the given date in the view.
- `public long getSelectedDay()`—returns the currently selected date in the specified format-time since the Epoch. The return value corresponding to the above example would be `1227389196351`.

Recall that the `Calendar` uses `MonthView` to provide the default view. However, `MonthView` cannot be referred to directly, as access to it is restricted. The `Calendar` class has two methods that make it possible to manipulate the `style` object associated with the month view instance. They are:

- `public Style getMonthViewStyle()`
- `public void setMonthViewStyle()`

The `Calendar` class can fire events when date selection is changed. These events can be received by listeners, which can be added or removed by using the following methods:

- `public void addActionListener(ActionListener l)`
- `public void addDataChangeListener(DataChangedListener l)`
- `public void removeActionListener(ActionListener l)`
- `public void removeDataChangeListener(DataChangedListener l)`

`ActionEvent` is fired whenever a date selection is changed, and encapsulates the source of the event which, in this case, is the `calendar` object. The `DataChangedListener` is a callback interface that receives information about the type of change.

Using a Calendar

We shall now create and display a calendar. The following screenshot shows the calendar:

If you click on the **Show Date** command, a dialog will open showing the selected date, which is **December 29, 2008** in this screenshot. The time when the calendar was created is also shown.

The part of the code that sets up a form is familiar to us. The only new issue here is that the content pane has been made translucent. This makes the form background appear gray, although the background color is set to black.

```
//set background attributes for the form
f.getStyle().setBgColor(0x000000);
f.getContentPane().getStyle().setBgTransparency(127);
```

The `Calendar` is a composite object containing several components, of which `MonthView` is one. So we create a separate `style` object, and use the `setMonthViewStyle(Style s)` method of `Calendar` to set this style. The code related to creating a calendar and setting its style is quite short, as shown below:

```
//create a calendar object
//and set its background color
calendar = new Calendar();
calendar.getStyle().setBgColor(0x4d587d);
//set font and transparency for the month view
font = Font.createSystemFont(Font.FACE_SYSTEM,
           Font.STYLE_PLAIN,Font.SIZE_MEDIUM);
Style style = new Style(0xffffff, 0x000000, 0xff0000, 0x000000,
              font, (byte)128);
calendar.setMonthViewStyle(style);
//add the calendar to the form
f.addComponent(calendar);
```

To set the color of the *month* and *year* texts, we create a style with desired foreground color. We then use the method `setComponentStyle(String id, Style style)` of `UIManager` to set the style for the labels that display the *month* and the *year*:

```
//create a style to set the foreground color
//and install it for all labels
Style lStyle = new Style();
lStyle.setFgColor(0xff0000);
UIManager.getInstance().setComponentStyle("Label", lStyle);
```

This is a very useful method as it sets a style for all components with a given `id`, except for those whose applicable style attribute has been manually set.

The code for the dialog is also simple. Although `TextArea` will be covered in Chapter 6, the code for creating the text areas and for setting their styles is not difficult to follow, because of the similarity with other components that we have already learnt about.

```
//create two text areas for displaying details of selected date
TextArea ta1 = new TextArea("Selected Date:");
ta1.getStyle().setBgTransparency(0);
ta1.getStyle().setFgSelectionColor(0x000000);
ta1.getStyle().setFont(f);
ta1.setEditable(false);

TextArea ta2 = new TextArea(3, 18);
ta2.setText(calendar.getDate().toString());
ta2.getStyle().setBgTransparency(0);
ta2.getStyle().setFont(Font.createSystemFont(Font.
   FACE_SYSTEM,Font.STYLE_PLAIN,Font.SIZE_LARGE));
ta2.getStyle().setFgSelectionColor(0x000000);
ta2.setEditable(false);
```

The method used for showing the dialog illustrates a "hanging" title:

```
//show the dialog
date.show(80, 20, 10, 30, false);
```

This method gives us some control over where we would want to place a dialog and which part of the underlying form to leave uncovered. We can see this in the screenshot of the **Selection** dialog for this example.

The method used to add `calendar` to the form is common to all the containers:

```
//add the calendar to the form
f.addComponent(calendar);
```

The general form of this method is `container.addComponent(Component cmp)`. So the same method has been used to add the text areas to the dialog. There are two more methods for adding components to a container. They are:

- `public void addComponent(int index, Component cmp)`—the component is added at the position specified by the value of `index`. The `index` primarily determines the stacking order of the components that are added to the container.
- `public void addComponent(String constraints, Component cmp)`—the component is added subject to the constraints. There is a layout manager that uses constraints to determine the location of a component. Obviously, this method is to be used only with the layout manager that requires this parameter. There will be more on constraints in Chapter 7.

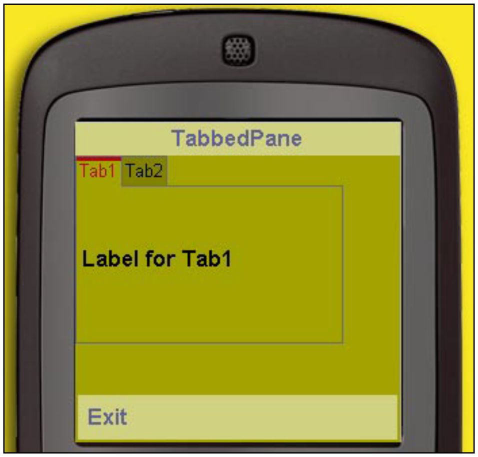
The TabbedPane

The `TabbedPane` widget has a number of sections or panes with one pane being visible at a time. Each pane has a tab which is like a flag. When a tab is clicked on, the corresponding pane becomes visible. A tab can have a title or an icon or both.

When tabs are added to a `TabbedPane`, the first to be added has an index of 0, the second has an index of 1 and so on. If there are N tabs, then the last one has an index of N-1. Note that adding a tab automatically adds the corresponding pane.

The panes can hold other components. Naturally, this means that a `TabbedPane` extends the `Container class`.

The given screenshot shows a tabbed pane with two tabs. The first tab is shown selected, and it contains a label with text on its pane.



We can select the second tab either by clicking on it (for the PDA emulator) or by using the *right* navigation key. The second tab looks like the following screenshot:

The calendar that was created earlier has been added to this tab. What we have here is a good example of *nested* containers. The calendar itself is a container with a number of containers. This has been added to a tabbed pane. Finally, the tabbed pane is added to a form.

There is one point about the operation of navigation keys that can be seen here. You can change the date selection by using the navigation keys. Once you do that, these keys become effective only for the calendar, which now has focus. In order to use them for changing tabs once more, you need to click on the center button which we shall call the **Select** key from now on. When the **Select** key is clicked on, the **up** navigation key will take the focus back to the tab title. The **left** and **right** navigation keys will now change focus from one tab to the next.

The general rule is that hitting the **Select** key toggles the applicability of navigation keys between the form and a component with focus contained in the form.

A new command can be seen to have appeared when the second tab is selected, and this is the command for showing the selected date dialog that we have already seen.

Creating a TabbedPane

In the previous two screenshots, we see the tabs on the top edge. It is possible to place the tabs on any of the other three edges. There are two ways to determine tab placement—by using the appropriate constructor or by using the method provided in this class for selecting the position of tabs. By default, tabs are placed at the top.

There are two constructors for this class. They are:

- `public TabbedPane()`—creates a `TabbedPane` object without any tabs and with `BorderLayout`. The tabs are placed at the top.
- `public TabbedPane(int tabPlacement)`—creates an empty tab but with the tabs placed as specified by the parameter. The tabPlacement parameter can either be `Component.TOP`, `Component.BOTTOM`, `Component.LEFT` or `Component.RIGHT`.

Methods of TabbedPane class

The methods that manage the manipulation of tabs are obviously the most significant. These are:

- `public void addTab(String title, Component component)`—adds a tab with the given title and component.. If the title is null, then the tab title will be blank.
- `public void addTab(String title, Image icon, Component component)`—adds a tab with the given title, icon, and component. Either the icon or the title can be null. It is also permissible for both of them to be null. However, then the tab title will be blank.
- `public int getSelectedIndex()`—gets the index of the selected tab.
- `public Component getTabComponentAt(int index)`—gets the component at the specified index.
- `public int getTabCount()`—gets the number of tabs in the TabbedPane.
- `public int getTabPlacement()`—gets the placement that has been specified for the tabs.
- `public int indexOfComponent(Component component)`—gets the index of the specified component.
- `public void insertTab(String title, Image icon, Component component, int index)`—inserts a tab at the specified index. Again, either the title or the icon or both can be null.
- `public int removeTabAt(int index)`—removes the tab at the specified index.

- `public void setSelectedIndex(int index)`—selects the tab specified by the index, as long as the index value is greater than zero and less than the number of tabs.
- `public void setTabPlacement(int tabPlacement)`—this is the method that sets the placement of tabs.
- `public void setTabTitle( String title)`—sets the title text of the tab.

A tabbed pane can inform a registered listener object that a tab selection has changed. The following method registers a listener:

- `public void addTabsListener(SelectionListener listener)`—registers a listener who will be informed when a tab selection changes.

In order to qualify as a listener, an object must implement the `SelectionListener` interface. This interface has only one method which is `voidSelectionChanged(int oldSelected, int newSelected)`.

A TabbedPane in action

Let us now create the TabbedPane that we have already seen above. The code for creating the form and the calendar are known to us and so is the method `showDateDialog()`. Therefore, we will now concentrate on the part of the code that deals with the tabbed pane.

The first thing to do is instantiate a tabbed pane, and set some of its style attributes:

```
//create a tabbed pane
//and set its background and foreground colors
TabbedPane tab = new TabbedPane();
tab.getStyle().setBgColor(0x999900);//bg color of pane body
tab.getStyle().setFgColor(0x888899);//border color
tab.getStyle().setFgSelectionColor(0xff5555);
                 //tab title selection color
```

Next, we add two tabs. The first tab contains a label and the second tab, as we have already seen, contains a calendar widget. The label used here can be created within the `addTab` method:

```
//add a tab to the tabbed pane
tab.addTab("Tab1", new Label("Label for Tab1"));
```

The calendar for the second tab is created in the same way as the calendar demo that we have already studied. It is then added to the tabbed pane:

```
//add a second tab to the tabbed pane
tab.addTab("Tab2", calendar);
```

Note the use of the `getTabComponentAt(int index)` method to get the components that have been added to the panes. We make the backgrounds of these components translucent so that the color of the tabbed pane comes through:

```
    //set tab body tranaparencies
    //first tab body
  tab.getTabComponentAt(0).getStyle().setBgTransparency(50);
    //second tab body  tab.getTabComponentAt(1).getStyle().
                                    setBgTransparency(50);
```

The tabs, which are part of a pane on which the title appears, actually form a list on which each title is a button. On the tabbed pane, this list is the component at index 1. So we call the `getComponentAt(int index)` method (different from the `getTabComponentAt(int index)` method used earlier) to get a reference to this list. Here the index refers to the components on the tabbed pane as a whole and not to those added to the panes. We then use the list reference to set the color of the text in the unselected state of the list:

```
//get the component that is the strip that holds the tab titles
List list = (List)tab.getComponentAt(1);

//set tab title unselected color
list.getStyle().setFgColor(0);
```

However, we cannot set the background color of the tabs in the same way. If we did just that, then the tabbedpane would appear as shown in the following screenshot:

To change the colors of the tabs, we create a style with desired background color. Once again we use the same method `setComponentStyle(String id, Style style)` of `UIManager` to set the style for all labels, including the one that forms the top of the button:

```
//create style for the title part of the tab
Style titleButtonStyle = new Style();
//set colors and font for that style
titleButtonStyle.setBgColor(0x999900);
titleButtonStyle.setFgColor(0x000000);
titleButtonStyle.setFont(lFont);
//set the style for label components
UIManager.getInstance().setComponentStyle("Label", titleButtonStyle);
```

This code appears right at the top of the MIDlet, after initializing `Display` and creating the fonts.

The next statement makes the `DemoTab` MIDlet the listener for changes in tab selection:

```
//make this MIDlet listener for tab change
tab.addTabsListener(this);
```

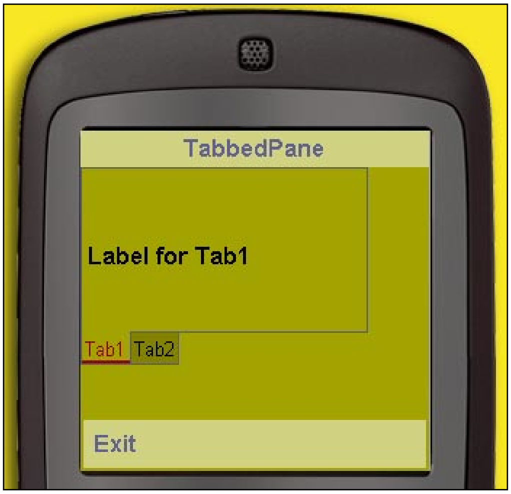
The method through which the MIDlet is actually informed about these changes is the `public void selectionChanged(int oldTab, int newTab)` method, as specified in the `SelectionListener` interface. Within this method, we check whether the newly selected tab is at index 1 (the second tab). If it is, then the `ShowDate` command is added. Otherwise, the command is removed:

```
/add or remove command when tab is changed
public void selectionChanged(int oldTab, int newTab)
{
  //print tab change message on console
  System.out.println("Selection changed from Tab" + (oldTab+1) +
        " to Tab" + (newTab+1));
  //create command for showing dialog giving details of
                                     the selected date
  Command showCommand = new Command("Show Date", 1);
  //is second tab selected?
  if(newTab == 1)
  {
    //yes, add the command to the form
    f.addCommand(showCommand);
  }
  else
  {
    //no remove the command
    f.removeCommand(showCommand);
  }
}
```

Finally, let us look at the following statements that appear just above the statement that shows the form—`f.show()`. These three commented out statements can change the position of the tabs.

```
//set tabs at the bottom
//tab.setTabPlacement(Component.BOTTOM);
//set tabs at left
//tab.setTabPlacement(Component.LEFT);
//set tabs at right
//tab.setTabPlacement(Component.RIGHT);
```

With all three statements commented out, the default positioning of the tabs comes into effect, and the tabs are placed at the top. However, if one of them is uncommented, then the tabs will be placed accordingly. For example, if the `tab.setTabPlacement(Component.BOTTOM)` statement is uncommented, then the tabs will be placed along the bottom edge of the tabbed pane, as shown in the following screenshot:



Style for the future

Earlier in this section, we saw how to use the various methods of `Style` to specify a component's appearance by setting attributes in the style instance associated with it. Under the new scheme of things, a component will have two styles associated with it and not just one style as at present—one style for the selected (focused) state and the other for the unselected state. For example, at present we use the `setBgColor` method for setting the background color for the unselected state and the `setBgSelectionColor` method to set the background color for the selected state. With the new version, there will not be a `setBgSelectionColor` method, and we will need to use the `setBgColor` method for the two distinct style instances.

The unselected style would be the default version that is created when a component is instantiated. The selected style will have to be set explicitly. The two styles can be accessed through appropriate setters and getters—`setSelectedStyle`, `getSelectedStyle`, `setUnSelectedStyle`, and `getUnselectedStyle`. Calling the `getSelectedStyle` method will create and install a new selected style, if the existing selected style is null.

To maintain compatibility with existing code (such as the example codes in this book), the new `Style` class will have a static boolean variable—`defaultStyleCompatibilityMode`, which will be true by default. When this flag is `true`, the existing `accessor` methods for the style object of a component will behave in the following ways:

- `getStyle` will return the proper style, depending on the state of the component at runtime
- `setStyle` will install the given style as the new unselected style

The existing methods (such as `setBgSelectionColor`) for accessing attributes in the focused state will automatically set and return corresponding attributes of selected style.

The mechanisms mentioned above will ensure that the code written for handling the style objects in accordance with the original release of LWUIT will not break. However, the provision for compatibility mode is likely to be withdrawn in future, once the need for backward compatibility ceases to be relevant.

Summary

This chapter has shown us how to use various types of containers, and how to create nested structure for UIs that can be as intricate as we want them to be. By this time, you should be perfectly at home with the container family of classes. The knowledge gained in this chapter will be very useful to us as we go on to study the other LWUIT widgets, and see how they can be assembled to form impressive screens for our applications.

4
The Label Family

The `Label` family of components has four members:

- `Label`
- `Button`
- `CheckBox`
- `RadioButton`

All of the above have distinct visual and behavioral characteristics. In addition to these four, there is one more member of this family that does not appear by itself on a screen and is not a component. This is the **ButtonGroup**, which is a supplementary object that works with a set of radio buttons.

In this chapter, we shall study `Label` and its descendants and build some demos. We shall also explore the `Border` class a bit, as it furnishes an important embellishment for components.

The road map for this chapter is:

- Learn about the `Border` class.
- Learn about the `Label` class, and check out the major functionalities through an example.
- Learn about the `Button` class. Build a demo application to see how buttons work.
- Learn about the `RadioButton` and the `ButtonGroup` classes. See how they work together by building a simple application.
- Learn about the `CheckBox` class and put it through its paces using an example.

The Border class

This is the class that provides an optional border around a component. The border is drawn in the padding region of components.

Although the `Border` class has a constructor for creating an empty border, we shall use the factory methods to create borders with predetermined characteristics. These factory methods are, by far, the majority of the methods in the arsenal of `Border`. The types of borders that can be created are:

- `BevelLowered` and `BevelRaised`—supports options for specifying colors.
- `EtchedLowered` and `EtchedRaised`—supports options for specifying colors.
- `Image`—the border is constructed as a tiled pattern of the given set of images.
- `Line`—this is a simple line border with a specified width. If a color is furnished, then this color is applied to the border. Otherwise the foreground color is used.
- `Round`—this is a border with the corners rounded. The degree of rounding can be specified, and so can the color.

The `Border` class allows us to define *focused* and *pressed* versions of a border instance. We can use a focused border to indicate that a component has received focus. Similarly, the pressed border can show that a component, like a button, has been pressed. The `createPressedVersion()` method derives a pressed version that reverses the effects of the border instance for use with buttons. This reversal of border effects produces an appearance of a button being pressed.

For certain types of borders, the task of painting a component's background is taken over by the `Border` class. This is done, because, for such borders, the area to be filled by a background painter cannot be easily determined by the normal background painter. On the other hand, the area for background painting is known to the border instance, as it has the parameters for creating the border.

We will see most of the borders in action in the examples of this chapter and in the rest of this book.

The Label

The `Label` class provides a space for text, icon, or both. It does not directly react to user actions, and it does not receive focus by default either.

However, it is a very widely used widget, and it provides the basis for building many other widgets. Let us recall the examples that we worked with in Chapter 3 using composite widgets like the calendar. We have seen that these composite widgets use labels within themselves to display textual information. As we work our way through LWUIT, we shall see many more examples of labels being used as a foundation for more complex components.

The LabelDemo example

The following screenshot shows a number of labels with text, or an image or both:

Once we have had a brief introduction to the `Label` class, we shall examine in detail the code that shows the above set of labels.

Creating a Label

There are three constructors for creating a Label. These are:

Constructor	Parameters	Description
`Label()`		Constructs a new Label without any text or icon.
`Label(String text)`	`text`—the string to be used as text.	Constructs a new Label with the given string as text. By default the text is left justified.
`Label(Image icon)`	`icon`—the image to be used as icon.	Constructs a new Label with the given image.

In our sample application, we have used all three constructors to create labels.

Methods of the Label class

The methods of this class are essential for handling the text and the icon that go on labels. We shall see many of the methods, especially the interesting ones, in the example here. As labels are used frequently in all our demos, we shall soon become familiar with the methods of the `Label` class.

The LabelDemo application

The only class for this application is the `DemoLabel` MIDlet.

The `startApp` method has all the code that handles the labels and the form that contains the labels. We shall discuss this in some detail. The `actionPerformed` method is for responding to the **Exit** command. This method is quite familiar to us and does not really need to be analyzed.

The first part of the code that deals with the form is virtually the same as the `DemoForm` MIDlet in Chapter 3. The appearances of the title and the menu bars are a little different. We shall return to this aspect later after discussing the labels.

After the form is set up, we create and install a style for labels using a method we studied in the previous chapter, while building the tabbed pane demo.

The first label on the form is created using the constructor that takes a `string` as its parameter.

```
//create label with short text
Label tLabel = new Label("Label that has just text");
//create label with long text
//Label tLabel = new Label("Label that has just text. Text,
                         only text and nothing but text.");
```

The string used in this case is short enough to fit on the label, as the screenshot shows. However, if the string is too long for the label, there is a way to handle the situation. The `Label` class has a `boolean` variable `endsWith3Points`, which is `true` by default. When the text for the label is too long to accommodate, it is truncated, and three dots ("...") are added to the end of the text. If you create a label with a long text by commenting out the code for short text and uncommenting the next line of code you will get the following screen:

You can see that the string has been truncated and the dots have been added. This function can be turned off or on by using the `public void setEndsWith3Points( boolean endswith3points)` method.

The next statement creates a border for `tLabel`. The border created here is of the `EtchedRaised` type.

```
tLabel.getStyle().setBorder(Border.createEtchedRaised());
```

The second label is `imLabel` and it holds only an image. It is created with the image as a parameter for the constructor. If the image cannot be loaded for some reason, then an `IOException` is thrown. In the `catch` block, we create a label with a message that tells us about the problem.

Next comes the border for `imlabel`. This is a `LineBorder` with the given color and thickness in pixels.

```
Label imLabel;

try
{
   imLabel = new Label(Image.createImage("/sdsym.png"));
}
catch(java.io.IOException ioe)
{
   imLabel = new Label("Image could not be loaded");
}

imLabel.getStyle().setBorder(Border.createLineBorder(7,
                                                0xfbe909));
```

The third label is `bothLabel`, and it has text, as well as an icon. This time, we will create an empty label and then add text to it. Finally, we add the icon.

```
Label bothLabel = new Label();
bothLabel.setText("Text and icon");
boolean noImage = false;

try
{
   bothLabel.setIcon(Image.createImage("/sdsym4.png"));

   //align text at top of label
   //bothLabel.setVerticalAlignment(Label.TOP);

   //position text at left of icon
   //bothLabel.setTextPosition(Label.LEFT);
}
catch(java.io.IOException ioe)
{
   bothLabel.setText(bothLabel.getText()
    + ". But image could not be loaded.");
   noImage = true;
}
```

A label that has both text and icon can have the text on any of the four sides of the icon (top, bottom, left or right). The default position for the text is on the right, as we see here. The `public void setTextPosition(int textPosition)` method sets the desired position for the text.

When the text is on the left-hand or right-hand side of the icon, it can be aligned along the top, center or bottom of the label. The method to be used is

```
public void setVerticalAlignment(int valign).
```

In the above code snippet from the DemoLabel MIDlet, we have two commented out statements for setting text position and alignment. If the first statement is uncommented, then the text will be aligned along the top of the label. Uncommenting the second statement will position the text on the left of the icon. If both are uncommented, what we get is shown in the following screenshot:

If the image for bothLabel cannot be accessed, an IOException will be thrown. Within the catch block, we add a message to the existing text, which is retrieved by calling the public String getText() method. Here we also set a flag to indicate that the icon could not be set.

The border for bothLabel is a RoundBorder. The createRoundBorder method takes three arguments. The first two define the diameters of the arcs at the corners—the horizontal and the vertical respectively. The third is an optional one that specifies the color. This last parameter may be left out. In that case, the foreground color of the component will be used.

```
bothLabel.getStyle().setBorder(Border.createRoundBorder(12,
                                                         5, 0xff0000));
```

After bothLabel is added to the form, the noImage flag is checked. If it is true, then the text on bothLabel is made to scroll (ticker), as we know that we have got a fairly long text here. The public void startTicker(long delay, boolean rightToLeft) method has to be called only after a label has been added to a form. This is why we have just set a flag within the catch block. The first parameter of the method specifies the time (in milliseconds) between two successive shifts during scrolling, and the second specifies the direction of scrolling, true being the value that denotes right-to-left scrolling. Just as there is a method for starting text scrolling, there is one for stopping it too—public void stopTicker().

```
if(noImage)
{
   bothLabel.startTicker(100, true);
}
```

To see the text ticker in action, change the name of the image for bothLabel from sdsym4.png to, say, sdsym40.png. If you recompile and run the application, then you will see how the ticker works.

Now we return to the issue of title and menu bar styles. The foreground and background colors have been set in their respective styles. Both title bar and menu bar have now been provided with borders. The border for title bar is BevelRaised and that for the menu bar is BevelLowered.

```
createSystemFont(Font.FACE_PROPORTIONAL,Font.STYLE_BOLD,Font.SIZE_
LARGE);
      demoForm.getTitleStyle().setFgColor(0xffffff);
      demoForm.getTitleStyle().setFont(font);
      demoForm.getTitleStyle().setBgColor(0xff8040);

      Style menuStyle = new Style();
      menuStyle.setBgColor(0xff8040);
      menuStyle.setFgColor(0xffffff);
      menuStyle.setFont(font);
      demoForm.setSoftButtonStyle(menuStyle);
      .
      .
      .
      .
      demoForm.getTitleStyle().setBorder(Border.createBevelRaised());
      demoForm.getSoftButtonStyle().
        setBorder(Border.createBevelLowered());
```

The Button class

The `Button` extends `Label`. Therefore, it inherits the characteristics of a label. In addition, `Button` has distinct capabilities of its own like these:

- It is able to sense and respond to user actions
- It can receive focus
- It has internal states—Default, Rollover, and Pressed

Like labels, buttons too are widely used, not only as standalone widgets, but also to build up other more complex components. Whenever we need to create an entity to display text, icon, or both and to be able to respond to key or pointer actions, buttons are very likely to be used. As we have seen in Chapter 3, each individual tab of a tabbed pane is actually a button.

Creating a Button

The `Button` class has five constructors, of which three are just like those of the `Label` class. The other two are a bit different. The constructors are:

Constructor	Parameters	Description
`Button()`		Creates a button without any text or icon.
`Button(String text)`	`text`—the string to be used as text.	Creates a button with the given text.
`Button(Image icon)`	`icon`—the image to be used as icon.	Creates a button with the given image.
`Button(String text, Image icon)`	`text`—the string to be used as text. `icon`—the image to be used as icon.	Creates a button with the given text and image. By default, the text is on the right of the icon and is centrally aligned.
`Button(Command cmd)`	`cmd`—the command to be bound to the button.	Creates a button with the given command bound to it.

The last constructor has a command associated with it. But this does not mean that this command will be encapsulated in the `ActionEvent` fired, when the button is pressed. Pressing the button fires an event that has an object (representing the element that triggered the event) associated with it, but not any command. If we call the `getCommand()` method on this event what we shall get is a null reference. The method to be used here is the `public Object getSource()`. In order to get the command that was bound to the button in the constructor, we need some additional coding, as we shall see when we examine the demo code.

The methods of Button class

The `Button` class inherits the methods of `Label`. In addition to these, the `Button` class has methods that enable it to sense key and pointer actions. These methods are:

Method	Parameters	Description
`void keyPressed (int keycode)`	`keycode`—code for the key that has been pressed.	Invoked by the key pressed event, if this button is focused.
`void keyReleased (int keycode)`	`keycode`—code for the key that has been released.	Invoked by the key released event, if this button is focused.
`void pointerPressed (int x, int y)`	`x`—x coordinate of the point at which the pointer has been pressed. `y`—y coordinate of the point at which the pointer has been pressed.	Invoked by the pointer pressed event, if this button is focused.
`void pointerReleased (int x, int y)`	`x`—x coordinate of the point at which the pointer has been released. `y`—y coordinate of the point at which the pointer has been released.	Invoked by the pointer released event, if this button is focused.

There are two methods that are very likely to be quite useful for building buttons that use icons. These are:

Method	Parameters	Description
`void setPressedIcon (Image pressedIcon)`	`pressedIcon`—image to be used as the icon when the button is pressed.	Sets the image to be used as icon when the button is pressed.
`void setRolloverIcon (Image rolloverIcon)`	`rolloverIcon`—image to be used as the icon when the button is in the rollover (focused) state.	Sets the image to be used as icon when the button is in the rollover (focused) state.

A button, as we know, has three states. When a button does not have focus, it is in the *default* state. A focused button is said to be in the *rollover* state. When clicked on, or when the pointer is pressed on it, the button's state is *pressed*. Various changes take place in the appearance of a button as its state changes, as the example will show.

A button fires an event when it is clicked on. To be able to receive this event, an object must register itself as a listener, by using the `addActionListener(ActionListener l)` method. To qualify as a listener, an object must be an instance of a class that implements the `ActionListener` interface. The listener can react to a click from `myButton` like this:

```
public void actionPerformed(ActionEvent ae)
{
   if(ae.getSource == myButton)
   {
      //take necessary action
   }
}
```

The DemoButton example

This example is visually very similar to the `DemoLabel` example, which we saw earlier in this chapter. The following screenshot shows the application as it looks when you open it:

While the similarities with `DemoLabel` are quite apparent, there are a number of differences too, which we shall study one by one with reference to the code.

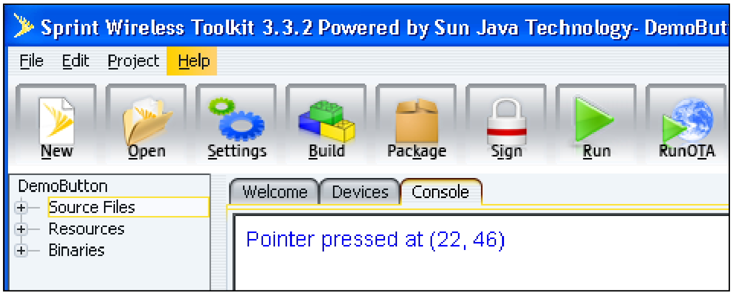
The new aspect here is the `CloseCommand` class for creating the command that is bound to one of the buttons. We shall go through the differences between the behavior and the appearance of `DemoButton` and of `DemoLabel`, and we will refer to the relevant part of the code.

The first difference is the background, text, and border colors of the first button (`tbutton`). When the application is opened, this is the button that gets focus, and the colors show the rollover state of the button. The background color has been set in `buttonStyle` (the common style that applies to all buttons in this example) through the statement `buttonStyle.setBgSelectionColor(0x555555)`. However, the color of the border and the text are default colors that have not been set in the code.

Another difference that is not visibly obvious is that a button can respond to user input in the form of both key and pointer actions. As the PDA platform on which we are running this application also supports pointer actions, let us see what happens when the pointer is pressed on `tButton`. To simulate pointer press, place the cursor over the button (the cursor should have become a "**+**") and click on it. The piece of code that responds to this action is:

```
Button tButton = new Button("Button that has just text")
{
   public void pointerPressed(int x, int y)
   {
      System.out.println("Pointer pressed at (" + x + ",
                                              " + y + ")");
   }
};
```

Here we create `tButton` and override the `pointerPressed` method to print a message on the console showing the coordinates of the point where the pointer was pressed. The result is as shown in the following screenshot:



The third difference for `tButton` is the way in which it shows a text that is too long to display fully. To see this difference, comment out the code for creating `tButton` with short text, and uncomment the statement to create `tButton` with long text. Now, if you compile and run the example, then you will find that the text in `tbutton` is scrolling on focus, although there is no code to start tickering. This is a default feature for all components. The reason we do not see this happening with a label is that a label is not focusable. When the focus moves away from `tButton`, the text will be shown ending in three dots, as in the case of a label.

The button on the right of the screen is `cmdButton`, which is a button with a command bound to it. When this button is pressed, the `actionPerformed` method of the associated command is invoked in addition to that of any other listener that may have been installed. The event that is passed as a parameter does not have a `command` object, because it has been generated by a button and not a command. Consequently, the `getCommand()` method cannot be used, as it would return null. If this event had to be processed by `DemoButton`, then its `actionPerformed` method would have to be modified to check for a source too. If we wanted to retain the existing structure and process events based only on command ids, then we would need to make sure that the command bound to the button generates an event. This new event, which would obviously encapsulate a `command` object, would then need to be fired at `DemoButton`. The sequence of activities would look like this: button (fires event) >> associated command (generates and fires new event) >> `DemoButton`.

In order to achieve this, we first write a new class (`CloseCommand`) that extends `Command`, and give it a method for setting a listener for the event it will fire.

```
class CloseCommand extends Command
 {
   //the listener for this command
   private ActionListener closeListener = null;

   //create command with given text and id
   public CloseCommand()
   {
      super("Close", 1);
   }

   //set the listener
   public void setListener(ActionListener listener)
   {
      closeListener = listener;
   }

   public void actionPerformed(ActionEvent ae)
   {
      //print message on console
```

```
                System.out.println("Close");
                if(closeListener != null)
                {
                   //create a new event
                   ActionEvent e = new ActionEvent(this);
                   //call the target method
                   closeListener.actionPerformed(e);
                }
             }
          }
```

The `actionPerformed` method of this class will be called only when the **Close** button is pressed, and this is why we do not need to check for the source. Therefore, we directly print a message on the console. Then, if a listener has been set, we can create a new event with the command, and call the `actionPerformed` method of the listener.

Within the MIDlet, we create an instance of `CloseCommand`, and call it `closeCommand`. Next, `DemoButton` is set as the listener for `closeCommand`. Finally, the `cmdButton` is instantiated with `closeCommand` as the bound command:

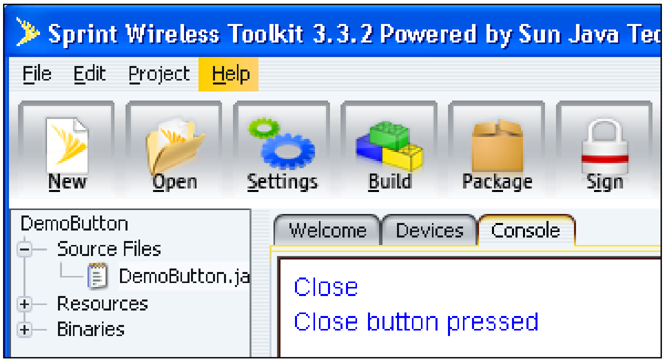
```
//create command for the next button
CloseCommand closeCommand = new CloseCommand();
//make this MIDlet the listener for closeCommand
closeCommand.setListener(this);
//create button with command
Button cmdButton = new Button(closeCommand);
//set a border for cmdButton
cmdButton.getStyle().setBorder(Border.createEtchedLowered());
//put a little space between this button
//and the one on its left
cmdButton.getStyle().setMargin(Label.LEFT, 10);
```

The highlighted statement above sets a 10 pixel margin on the left of `cmdButton` to provide a reasonable spacing.

The `actionPerformed` method of the MIDlet can now retain its original structure and can process the call based on the command id. In this case, all it does is print a message on the console.

```
//command asociated with 'Close' button
 case 1:
   System.out.println("Close button pressed");
```

If `cmdButton` is clicked on, then you shall see the following messages printed out on the console:



As you would expect, the first message is **Close**, since the `actionPerformed` method of `closeCommand` is called first. The **Close button pressed** message is printed after that by the `actionPerformed` method of `DemoButton`.

We turn our attention now to `imButton`, which is the one with only an icon. As long as this button does not have focus, it looks identical to the corresponding label in `LabelDemo`. The difference surfaces when the button gains focus is shown in the following screenshot:

The border is now different, and when the button is clicked on, we get yet another border:

The two new borders are focused and pressed versions that are set for the border instance used with `imButton`. The highlighted statements in the code snippet below create and set the appropriate borders for use, when the button gains focus and when it is pressed:

```
Border imBorder = Border.createLineBorder(7, 0xfbe909);
imBorder.setFocusedInstance(Border.createLineBorder(7,
                                                0x00ff00));

imBorder.setPressedInstance(Border.createLineBorder(7,
                                                0x0000ff));

imButton.getStyle().setBorder(imBorder);
```

The fourth button demonstrates how the icon can change, depending on the state of the button. The statement that sets the icon for the rollover state is `bothButton.setRolloverIcon(Image.createImage("/sdsym1.png"))`. The effect of this statement is seen in the following screenshot:

The icon for the pressed state is set by the statement—`bothButton.setPressedIcon(Image.createImage("/sdsym3.png"))`. Now, if you press this button, then you will see the following:

Note that the two versions of border were set in the `border` object, while the icons for the two states had to be set in the `button` object.

The CheckBox

The `CheckBox` is a subclass of `Button`. The name of this widget is pretty descriptive. It looks like a label or a button with a *Box* that gets *Checked* when selected. A check box has a memory. Therefore, it can remember its state. Also, repeated clicking on it will toggle the state of a check box. The following screenshot shows a couple of check boxes, one with only a text and the other with an icon as well as a text:

In this screenshot, both the check boxes are in the selected state. The text in the upper check box ends with three dots as the lower check box is the one that has focus. The text on the focused check box is actually tickering. However, this is not discernible in the screenshot.

Another point to note here (which also applies to the other members of the `Label` family) is that the size of the widget is automatically adjusted to accommodate the largest element that goes into it. This explains why the lower check box is much larger than the upper one.

The code for creating this screen is almost the same as the corresponding codes for labels or buttons, and we shall not discuss it here. After familiarizing ourselves with the constructors and methods of `CheckBox`, we shall spend some time on another example to see the check boxes in action.

Creating a CheckBox

The four `CheckBox` constructors are similar to those of `Label` and `Button`:

Constructor	Parameters	Description
`CheckBox()`		Creates a check box without any text or icon.
`CheckBox(String text)`	`text`—the string to be used as text.	Creates a check box with the given text.
`CheckBox(Image icon)`	`icon`—the image to be used as icon.	Creates a check box with the given image.
`CheckBox(String text, Image icon)`	`text`—the string to be used as text `icon`—the image to be used as icon.	Creates a check box with the given text and image. By default, the text is on the right of the icon and is centrally aligned.

The second and the fourth constructors have been used to make the screenshot, which was shown earlier.

Methods of the CheckBox class

As a descendant of the `Label` class and the `Button` class, the `CheckBox` class inherits methods of these two classes. There are two methods that support the check box functionality and these methods are:

Method	Parameters	Description
`boolean isSelected()`		Returns `true` if the check box is selected and false otherwise.
`void setSelected (boolean selected)`	`selected`—value to which the check box state is to be set.	Sets the check box state as per the given boolean variable.

The "Languages Known" example

The following example screen simulates something that is a very common online entity—one page of a set of several pages for creating a biodata:

There are six check boxes, and you can select any number of them. Once you have made your selection, you can click on the **Confirm** command, and then the following dialog will open:

Clicking on the **Back** command will take you back to the previous screen. In a real life situation, executing the **OK** command would have taken you to the next screen. Here it takes you back to the previous screen, but it also clears all selected check boxes.

The code for `DemoCheckBox`, which is the MIDlet for this example, is really very straightforward. We first set up the array of strings that will form the texts of the check boxes, as well as the preferred dimension of each check box. The check boxes are all very similar, and there is no point in setting them up individually. So we create all six of them inside a loop, set the preferred size, specify a margin, and then we add the check boxes to the form.

```
private final int cbQty = 6;//number of checkboxes
private CheckBox[] checkboxes = new CheckBox[cbQty];
.
.
.
   String[] langs = {"Chinese", "Spanish", "English",
                    "Portuguese", "Hindi", "Bengali"};
   Dimension d1 = new Dimension(100, 25);
   for(int i = 0; i < cbQty; i++)
   {
      checkboxes[i] = new CheckBox(langs[i]);
      checkboxes[i].setPreferredSize(d1);
      if(i % 2 == 0)
      {
          checkboxes[i].getStyle().setMargin(Label.RIGHT, 15);
      }
      demoForm.addComponent(checkboxes[i]);
   }
```

When the **Confirm** command is executed, the `showDialog` method is invoked. Within this method, we create labels (once more in a loop) to show the languages selected. Note that we do not create all six labels to start with. The `isSelected` method is called on each check box. If the check box has been selected, then the method returns `true` and a label is created with the corresponding text. This means that we create only as many labels as we actually need.

```
for(int i = 0; i < cbQty; i++)
{
  if(checkboxes[i].isSelected())
  {
     Labels l = new Label(checkboxes[i].getText());
     l.setPreferredSize(dim);
```

```
            l.getStyle().setFgColor(0x555555);
            l.getStyle().setFont(f);
            l.getStyle().setMargin(Label.BOTTOM, 5);

            d.addComponent(l);
        }
    }
```

The method used to show the dialog (`showDialog`) returns the command that was executed to close the dialog. If this was the **OK** command, then the `reset` variable is set to `true`, indicating that the selections have to be cleared.

```
    Command cmd = d.showDialog();

    if(cmd.getCommandName().equals("OK"))
    {
        reset = true;
    }
```

Back in the `actionPerformed` method, `reset` is checked when `showDialog` returns. If `reset` is `true`, the state of each check box is tested. The `setSelected()` method is called on each selected check box so that its state can be cleared.

```
    case 1:
        showDialog();

        //on retun from showDialog method
        //check whether reset is true
        if(reset)
        {
            reset = false;

            for(int i = 0; i < cbQty; i++)
            {
                if(checkboxes[i].isSelected())
                {
                    checkboxes[i].setSelected(false);
                }
            }
        }
```

Here we see modality at work—execution of the MIDlet code stops until the dialog is closed. This guarantees the clearing of the check boxes at the correct time.

The RadioButton and ButtonGroup

The `RadioButton` class is functionally similar to `CheckBox`: they both extend `Button`, and they both can remember their selection status. The special capability of `RadioButton` is that, while working with `ButtonGroup`, it supports exclusive selection within a given set. Also, once a radio button is selected, repeated clicking on it has no effect.

Since a radio button gains its special functions only when paired with a button group, let's study the `ButtonGroup` class before getting into the details of a radio button.

The ButtonGroup class

The `ButtonGroup` class is an `Object` that acts like a logical container, and it gives a radio button its special property to ensure that, within a button group, only one radio button can be in the selected state at a time. Clicking on another radio button within the group will select that button, and it will clear the one that was originally selected.

The `ButtonGroup` is a *logical* container because it has no visible presence and none of the attributes like style that go with visibility.

The `ButtonGroup` has only one constructor that does not take any parameters. This constructor creates an empty button group. The methods of this class permit radio buttons to be added and removed from a button group. There are also two methods that allow access to radio buttons within a button group. One of these methods is `public int getButtonCount()`, which returns the number of radio buttons in a group. The other is `public RadioButton getRadioButton(int index)`, which returns the radio button that is specified by the given index within the group.

The other methods offer support for detecting and modifying the states of radio buttons within a button group:

Method	**Parameters**	**Description**
`void clearSelection()`		Sets all radio buttons of the button group to the cleared (unselected) state.
`void setSelected (int index)`	`index`—index value to specify a radio button.	Sets the specified radio button to the selected state.
`void setSelected (radioButton rb)`	`rb`—specifies a radio button.	Sets the specified radio button to the selected state.

Method	Parameters	Description
`int getSelectedIndex()`		Returns the index of the selected radio button within the button group. If none is selected, then `-1` is returned.
`boolean isSelected()`		Returns true if any radio button in the button group is selected and `false` otherwise.

The `RadioButton` class is also very simple and has a familiar structure, as we shall see now.

Creating a RadioButton

The four constructors of `RadioButton` are identical in form to the ones of `CheckBox`:

Constructor	Parameters	Description
`RadioButton()`		Creates a radio button without any text or an icon.
`RadioButton (String text)`	`text`—the string to be used as text.	Creates a radio button with the given text.
`RadioButton(Image icon)`	`icon`—the image to be used as the icon.	Creates a radio button with the given image.
`RadioButton(String text, Image icon)`	`text`— the string to be used as text. `icon`—the image to be used as the icon.	Creates a radio button with the given text and image. By default, the text is on the right of the icon and is centrally aligned.

In the example that follows, we shall meet only the second form of constructors, as we are already familiar with all the others and their usage.

Methods of the RadioButton class

`RadioButton` has just two methods that control its individual usage:

Method	Parameters	Description
`boolean isSelected()`		Returns `true` if the radio button is selected and `false` otherwise.
`void setSelected (boolean selected)`	`selected`—value to which the radio button state is to be set.	Sets the radio button state as per the given boolean variable.

These methods have not been used in the `DemoRadioButton` application, because they function just like their counterparts in the `CheckBox` class, and we have already seen how to work with them. In the following example, we focus on the methods of `ButtonGroup`.

The "Reservation" Example

This example simulates what might be a part of an online flight reservation system. The opening screen has two groups of radio buttons—one for indicating meal preference and the other for seat preference. The groups are initialized so that in the first group, all radio buttons are in the cleared state, and in the second, the **None** radio button is preselected. This is what the screen looks like:

Selections can be made by clicking on a radio button. Needless to say, only one selection per group is permitted. Let us say that the following selections have been made:

Now executing the **Confirm** command opens the familiar dialog:

Again, the **Back** command is for going back to the form without any changes in selections. The **OK** command also takes us back to the form, but with the initialization of the first screen.

The differences between this MIDlet and `DemoCheckBox` are related to the fact that radio buttons are members of button groups, and they expose their functionalities through the methods of the group they belong to.

As we look through the code, we notice that it is not enough to add radio buttons to a button group. They have to be added to a container too. This is one implication of `ButtonGroup` being just a grouping mechanism and not a true `Container`. The container is what makes the radio buttons visible. For the same reason, we cannot add a border to a `ButtonGroup`. Had we not wanted to put an individual border around each button group, we could have added the radio buttons directly to the form.

```
   private final int mealNums = 4;
private final int seatNums = 4;
.
.
private RadioButton[] mealPrefs = new RadioButton[mealNums];
private RadioButton[] seatPrefs = new RadioButton[seatNums];    .
.

   Container meals = new Container();
   Container seats = new Container();

   for(int i = 0; i < mealNums; i++)
   {
      mealPrefs[i] = new RadioButton(mealTexts[i]);
      mealPrefs[i].setPreferredSize(d1);

      if(i % 2 == 0)
      {
          mealPrefs[i].getStyle().setMargin(Label.RIGHT, 15);
      }

      mealGroup.add(mealPrefs[i]);
      meals.addComponent(mealPrefs[i]);
   }

   for(int i = 0; i < seatNums; i++)
   {
      seatPrefs[i] = new RadioButton(seatTexts[i]);
      seatPrefs[i].setPreferredSize(d1);

      if(i % 2 == 0)
      {
```

```
            seatPrefs[i].getStyle().setMargin(Label.RIGHT, 15);
        }
        seatGroup.add(seatPrefs[i]);
        seats.addComponent(seatPrefs[i]);
    }
```

Initialization of the radio button states needs to be done only for the **Seating Preference** group. The **Meal Preference** group does not need any explicit initialization, as the default initialization for a button group sets all its radio buttons to the cleared state.

```
//initialize setting for seat preference group
//either of the two following statements can be used
//seatGroup.setSelected(3);
seatGroup.setSelected(seatPrefs[3]);
```

The method used to set the **None** preference as selected is the `setSelected` method, which has two forms. Either form can be used here. The indices of radio buttons within a group are determined by the order in which they are added to the group. In this example, **Meal Preference Vegetarian** has an index of 0 in its group, as it is the first one to be added. Similarly **Seating Preference None** has an index of 3 in its group, as it is the fourth one to be added to its group.

In the `showDialog` method, we get the index of the selection in the **Meal Preference** group by calling the `getSelectedIndex` method. This group is initialized so that all radio buttons are cleared, and it is quite possible for a user to forget to make a selection. If this happens, then the value returned by `getSelectedIndex` is `-1`, and the label text is set to `No selection`. Otherwise the returned index is used to get the corresponding text from the array that holds the strings for the **Meal Preference** radio buttons.

```
if(mealSelected != -1)
{
mPrefLabel.setText(mealTexts[mealGroup.getSelectedIndex()]);
}
else
{
mPrefLabel.setText("No selection");
}
```

It is important to test for `No selection` because trying to get the text from an array with an index of `-1` would throw an exception. Another way of getting the text of the selected radio button is by using the little more complex statement: `mPrefLabel.setText(mealGroup.getRadioButton(mealGroup.getSelectedIndex()).getText());`.

Executing the **OK** command causes the selections to be reinitialized. This action takes place in the `actionPerformed` method. Here, the `clearSelection` method is invoked for the **Meal preference** group to clear all the radio buttons. The next step calls the `setSelected` method to put the **None** radio button in the selected state.

```
case 1:
      showDialog();
      if(reset)
      {
         reset = false;
         mealGroup.clearSelection();
         //set 'None' selected for seat prference
         //either of the two following statements can be used
             seatGroup.setSelected(3);
         //seatGroup.setSelected(seatPrefs[3]);
      }
```

Summary

`Label`, `Button`, `CheckBox`, `RadioButton`, `ButtonGroup`, and `Border`—in this chapter, we learnt how all these classes work, and we analyzed a number of demo applications that showed us the effects of using the different constructors and methods of these classes.

In the last two examples, there were several instances of widgets being sized and aligned manually. This approach works quite well in producing neatly arranged displays for a particular device. On a different device, the screen may not produce the desired result, and the resulting look may turn out to be quite unattractive. Layout managers in LWUIT provide a solution to this problem. In Chapter 7, we shall see how a judicious use of layout managers can produce screens that look good irrespective of the device they run on.

5
List and ComboBox

There are some components that are not only used by themselves, but also form elements of foundation for other components. **Label** is one such component and **List** is another. In Chapter 3, we saw that the tabs of a tabbed pane form a list, and in this chapter, we shall see that the **ComboBox** class is built around an underlying list.

We shall build several examples with lists and combo boxes in this chapter. In the process, we shall learn different ways of creating these widgets. We shall also learn how to make them look and behave in a variety of ways.

The list

At the most fundamental level, a list is just a collection of items. Usually this collection is shown as a linear display, and the items are text strings very often. However, the LWUIT implementation of a list is extremely flexible and allows an item to be any object. The rendering too can be completely customized. This is possible because the list component is decoupled from its data as well as its visual representation in accordance with the Swing-like MVC-oriented architecture of LWUIT.

The data, that is, the collection of items making up a list, can be held in many forms. For example, when we are creating a list of names, we can set up an array of strings to hold the information. LWUIT also defines a model—the **ListModel** interface that specifies the functionality of a general purpose holder of data for a list. The **DefaultListModel** is a vector-based implementation of this interface.

Like the data container, the UI for a list can be implemented in a wide variety of ways. The behavior of the renderer is defined by the **ListCellRenderer** interface. The default implementation of this interface that comes with the library is the **DefaultListCellRenderer**. It is not mandatory to use this default implementation, and we can plug in a custom renderer in its place. However, the substitute must implement the `ListCellRenderer` interface.

To bring out the versatility of a list widget, we shall build three different lists:

- The first example will be based on the default look and feel and will demonstrate the basic functionalities.
- The second will use a custom renderer to create a more interesting list—a list of elements that are objects specially created for this demo. Moreover, a custom renderer will be used to draw the list.
- The first two lists are built using stateless elements. For the third example, we shall create a 'to do' list with check boxes as the underlying elements, and we will see how to use stateful objects in a list.

Creating a List

The first requirement for instantiating a list is to decide where we shall place the data for the list. There are two alternatives—an array or a vector. Whichever one we chose, it will have to be incorporated into the `DefaultListModel`, either directly or through the constructors of the `List` class.

There are four constructors of the `List` class. These are:

Constructor	Parameters	Description
`List()`		Creates a new instance of `List` with an empty default list model.
`List(ListModel model)`	`model`—the model instance	Creates a new instance of `List` with the given list model.
`List(Object[] items)`	`items`—the array containing items to be placed into the model	Creates a new instance of `List` with a default list model containing the given array.
`List(Vector items)`	`items`—the vector containing items to be placed into the model	Creates a new instance of `List` with a default list model containing the given vector.

The methods of the List class

We have seen that components have essentially two kinds of methods—those that handle common actions like responding to key or pointer presses and those that provide support for component-specific visual aspects and functionalities. Accordingly, `List` has methods that allow us to retrieve or set the selected item (or its index) on a list. There are also methods that determine the visibility and positioning of the selected item on a list.

In order to manipulate the visual aspects, there are methods that set the gaps between successive items on a list and determine its orientation—vertical by default, or horizontal. However, a developer can exercise significant control over the appearance of a list by customizing the renderer.

In the examples that follow, we shall encounter the previously methods mentioned, as well as plug-in renderers to produce different visual effects.

Setting up a basic list

The list we shall build in this section is going to look like this:

To create this list, we will start by defining an array to hold the strings that form the list. We then use this array to get an instance of `DefaultListModel` Finally, we set up our list.

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

// Initialize a default list model with "item"

DefaultListModel simpleListModel = new DefaultListModel(items);

// Create a List with "simpleListModel"
List simpleList = new List(simpleListModel);
```

Alternatively, we could have omitted the creation of `simpleListModel` and used the array directly to instantiate `simpleList`. The result would have been the same.

The next step is to install a `ListCellRenderer`, which in this case, is an instance of `DefaultListCellRenderer`.

```
//create a ListCellRenderer and install
DefaultListCellRenderer dlcr = new DefaultListCellRenderer();
simpleList.setListCellRenderer(dlcr);
```

The point to note here is that a `DefaultListCellRenderer` instance is automatically installed for a list at the time of its creation, and it is not really necessary to explicitly install one. This step has been included here just to show how a renderer is installed, but in actual practice we need not do this unless a custom renderer is to be used.

When the preferred size of a list is determined before rendering, it helps to have some criterion for the calculation. This can be done by using the `setRenderingPrototype(Object renderingPrototype)` method. If we do not use this facility, then the size may not be adequate to accommodate all the items properly. This can be seen in the following screenshot:

Note that the list is not wide enough for the last two strings. To ensure proper size calculation, we write the following statement:

```
//set prototype to ensure adequate width
simpleList.setRenderingPrototype("WWWW");
```

The width of the list will now be determined by the width of the string "`WWWW`" using the instance front.

To set the background transparency of the list, we use the familiar statement:

```
//set transparency for list
simpleList.getStyle().setBgTransparency(64);
```

The other visual attributes of `simpleList` cannot be set through its style object. This is because the actual rendering of the list is performed on the basis of the component furnished by the `ListCellRenderer`. When a list is painted, the `public Component getListCellRendererComponent` method of the associated `ListCellRenderer` is called for each element of the list and the returned component is painted. Therefore, the style object that is used to paint the list elements is the one for the component obtained from the `ListCellRenderer`. The `DefaultListCellRenderer` instance that we have used here is itself a component (it extends `Label`), and if we set the attributes for its `style` object, `simpleList` will have the desired look.

```
//create and set style for the default list cell renderer
Style lStyle = new Style();
lStyle.setFgColor(0);
lStyle.setFgSelectionColor(0xffffff);
lStyle.setBgSelectionColor(0x0000ff);
lStyle.setFont(Font.createSystemFont(Font.FACE_SYSTEM,Font.
                             STYLE_PLAIN,Font.SIZE_LARGE));
dlcr.setStyle(lStyle);
```

Since `DefaultListCellRenderer` extends `Label`, we can also have set `lStyle` for all labels through the `UIManager.getInstance().setComponentStyle` method. The effect on the appearance of the list would be the same, as long as this is done before creating the `DefaultListCellRenderer`.

When we scroll through a list, the way focus moves from one element to the next can be smooth or abrupt. By default, this movement is smooth, but we can disable smooth scrolling if we want. This is as follows:

```
//disable smooth scrolling
simpleList.setSmoothScrolling(false);
```

The lists we see in various applications are usually arranged vertically. The `List` class has a method which allows us to create horizontal lists:

```
//make a horizontal list
simpleList.setOrientation(List.HORIZONTAL);
```

Let's now go back for a moment and look at the items defined for the list:

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

We see that although the strings do not have any numbers, the list is a numbered one. By default, lists that use the `DefaultListCellRenderer` will have a serial number for each item. This numbering can be suppressed as follows:

```
//do not number list entries
dlcr.setShowNumbers(false);
```

The result is seen in the following screenshot:

A list by default is *non-cyclic*, that is, scrolling through the list stops at the upper and lower boundaries. It is possible to make a list *cyclic* so that scrolling beyond the last item will select the first one. Similarly, scrolling up from the first item will select the last one. To do this we use the `setFixedSelection` method:

```
//make the list cyclic
simpleList.setFixedSelection(List.FIXED_NONE_CYCLIC);
```

The method mentioned above can be used with several other parameters, which determine the positioning of the selected items. We shall try out some of them in the next example.

A list with custom rendering

The `ListCellRenderer` for the list in the previous section was the `DefaultListCellRenderer`, and the constituent items were strings. The elements for the list that we build in this section will be objects specially made for this list, and the renderer will be a customized one to convert the item into a component. Each element of this list will be an instance of the `Content` class that we are going to write. The custom renderer used for drawing the list elements will be named `AlphaListRenderer`. The resulting list will look like the following screenshots:

Each element has two parts (the large alphabet that is actually an image and a text). To create a long list, six items have been used repetitively. When an item is selected, either through the *Select* key or through a pointer press, a selection dialog is shown.

The MIDlet for this demo creates and styles a form in the usual way. There are three points worth noting here. The first is that we have disabled the scrollbar of the form as a whole. This is because the list is a long one, and it is going to have its own scrollbar. The second is that we have set styles to the scrollbars and scrollthumbs (the slider) to give them the kind of look we want. The third point to take note of is that we obtain the width of the form as applicable for the device the application is running on by calling the `getWidth` method on the form instance:

```
//create and set a style for scrollbar
Style scrollStyle = new Style();
scrollStyle.setFgColor(0x5555ff);
scrollStyle.setBgColor(0xdddd99);
UIManager.getInstance().setComponentStyle("Scroll",
                                          scrollStyle);
//create and set a style for scrollthumb
Style scrollThumbStyle = new Style();
scrollThumbStyle.setMargin(Component.LEFT, 1);
scrollThumbStyle.setMargin(Component.R3IGHT, 1);
UIManager.getInstance().setComponentStyle("ScrollThumb",
                                          scrollThumbStyle);
//create a new form
Form demoForm = new Form("Alpha List Demo");
//no scrollbar for the form
demoForm.setScrollable(false);
//get width of the form
int width = demoForm.getWidth();
```

Let's now see what has been done to set up this list.

```
try
{
   letters[0] = Image.createImage("/a.png");
   letters[1] = Image.createImage("/b.png");
   letters[2] = Image.createImage("/c.png");
   letters[3] = Image.createImage("/d.png");
   letters[4] = Image.createImage("/e.png");
   letters[5] = Image.createImage("/f.png");
}
catch(java.io.IOException ioe)
{
}
//create a list using the images and texts
alphaList = getList(letters, descriptions);
```

This code snippet shows that six images have been created from the files in the resource folder, and they have been loaded into an image array. The images are created by calling the static method `createImage(String path)` of the `Image` class—one of the several methods for creating images from a variety of sources.

The actual work of creating a list is done within the `getList` method. So let's take a look at it next.

```
//returns a list created from an image array and a text array
private List getList(Image[] images, String[] texts)
{
   int length = images.length;
   String common = " is for ";
   //the list will have 24 elements
  for(int i = 0; i < 24; i++)
   {
      //repetitively use 6 images and strings
      int index = i % length;
      //Content is the object used as list element
      //create and load 24 Content objects
     contents[i] = new Content(images[index], common+texts[index]);
   }
   return new List(contents);
}
```

This list is a collection of `Content` objects, and each such object has an image and a text. This method creates an array of `Content` objects (contents) and invokes the `List` constructor, passing `contents` as a parameter. The `Content` class has a constructor to initialize its image and the text variables. It also has getter methods for these variables, as we see in the listing that follows:

```
class Content
{
   private Image letter;
   private String text;
   public Content(Image i, String s)
   {
      letter = i;
      text = s;
   }
   public Image getIcon()
   {
      return letter;
```

```
    }
    public String getText()
    {
       return text;
    }
}
```

Once the list is returned by the `getList` method, a renderer is installed, and a preferred size for the list is hinted, taking into account the width of the form.

```
        //get an AlphaListRenderer instance and install
        AlphaListRenderer renderer = new AlphaListRenderer();
        alphaList.setListCellRenderer(renderer);
        //hint for alphaList size
        alphaList.setPreferredSize(new Dimension(width-2,
                              alphaList.getPreferredH()));
```

As we know, the renderer is a custom one, and we shall see below how it converts a `content` object into a label for rendering the list.

```
class AlphaListRenderer extends Label implements ListCellRenderer
{
   //create new AlphaListRenderer
   public AlphaListRenderer()
   {
      super();
   }
   public Component getListCellRendererComponent(List list,
            Object value, int index, boolean isSelected)
   {
      //cast the value object into a Content
      Content entry = (Content)value;
      //get the icon of the Content and set it for this label
      setIcon(entry.getIcon());
      //get the text of the Content and set it for this label
      setText(entry.getText());
      //set transparency
      getStyle().setBgTransparency((byte)128);
      //set background and foreground colors
      //depending on whether the item is selected or not
      if(isSelected)
      {
         getStyle().setBgColor(0x0000ff);
```

```
            getStyle().setFgColor(0xffffff);
        }
        else
        {
            getStyle().setBgColor(0xff0000);
        }
        return this;
    }
    //initialize for drawing focus
    public Component getListFocusComponent(List list)
    {
        setText("");
        setIcon(null);
        getStyle().setBgColor(0x0000ff);
        getStyle().setBgTransparency(80);
        return this;
    }
```

The `ListCellRenderer` interface defines two methods that are called for getting the components to be used for painting the list. The first method is called once for each element, and it returns a component initialized with values appropriate to the element concerned. The parameter *value* represents the element to be drawn. In this case, value is a content object. The image and text for this object are retrieved, and a label is created and returned. The painting method draws this label, discards it, and gets a reinitialized label to draw the next item on the list. The `isSelected` parameter is `true` for the item that has focus, and it is used here to set background and foreground attributes to the selected and unselected components.

The second method—`getListFocusComponent` is used for focus transition. It is a good idea to experiment with different values with the variables set by this method to see how the transition is affected.

When an item on the list is selected by a pointer press or by clicking on the *Select* key, an `ActionEvent` is fired. So we add an `ActionListener` so that the `ActionEvent` can be acted on.

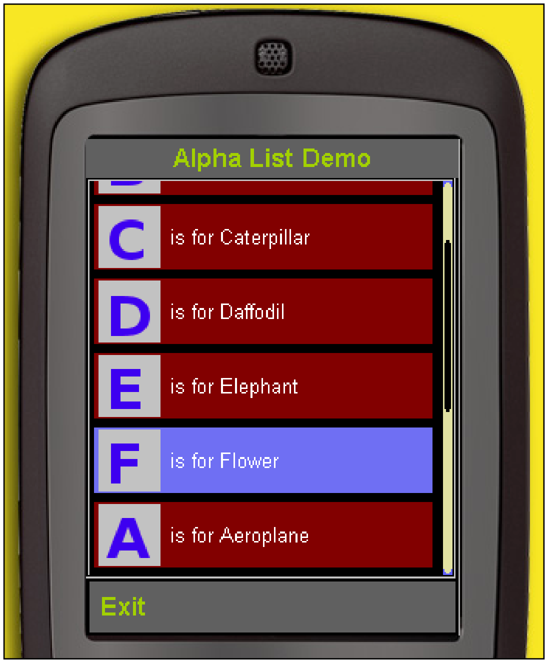
```
//add a listener to sense key/pointer action
alphaList.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent ae)
        {
            showDialog();
        }
    });
```

We saw in the last example how to use the `setFixedSelection` method to make a list cyclic. The same method can be used with other parameters to fix the position of the selected item. Consider the following statement:

```
//fixes the position of the selected item
alphaList.setFixedSelection(List.FIXED_CENTER);
```

This displays the selected item always at the center of the screen, as we can see in the next screenshot:

Another parameter that can be used with the method mentioned above is `FIXED_NONE_ONE_ELEMENT_MARGIN_FROM_EDGE`. This parameter will fix the position of the selected element one position away from the edge (bottom edge while scrolling down and top edge while scrolling up), until the last or first element is reached. In the following screenshot, we see that the selected item is shown one position above the bottom edge.



The ToDoList

The `ListCellRenderer` instance used with a list returns a component that is initialized for drawing an element. The component is then discarded, and even when the same element has to be redrawn, a reinitialized version of the required component is obtained. This means that the same component is recycled, and the individual state information is not retained.

This was fine for the two previous examples, because the elements were labels and did not have any internal states anyway. But what happens when we use elements that have internal states which need to be shown? How do we convey state information through components that are, in effect, stateless? We will deal with this in our ToDoList example.

The screenshot that follows depicts the **ToDoList** with the first two entries ticked off.

A task can be checked by a pointer press or by selecting an item and clicking on the *Select* key. A checked task can be unchecked in the same way.

The elements in this list are instances of the `Task` class, as shown below:

```
class Task
{
   private boolean done;
   private String todo;
   protected Task(String text)
   {
      todo = text;
   }
   protected boolean isDone()
   {
      return done;
   }
   protected void setDone(boolean value)
   {
      done = value;
   }
   protected String getText()
   {
      return todo;
   }
}
```

There are two variables associated with a task—`done` and `todo`. The variable `done` shows us whether a task is completed. Our first job is to set this variable to *true* or *false* when we click on the corresponding element. The *Select* key action or pointer press can be sensed by adding an `ActionListener` to the list (named `toDoList` here) so that we can receive the corresponding event:

```
//make this MIDlet the ActionListener for the List
toDoList.addActionListener(this);
```

The designated listener for `toDoList` is the MIDlet itself. Within the `actionPerformed` method, we check if there is a command associated with the event. If there is no event, then we see if the source of the event is the list. If so, then we get the index of the selected item and use it to toggle the `done` variable of the corresponding `Task` object. A second way of doing this would be to get the `Task` object for the selected item directly by using the `getSelectedItem()` method and then toggle its `done` variable.

```
//act on the command
public void actionPerformed(ActionEvent ae)
{
   Command cmd = ae.getCommand();
   if(cmd != null)
   {
           switch (cmd.getId())
      {
         //'Exit' command
         case 0:
                  notifyDestroyed();
      }
      return;
   }
   Object src = ae.getSource();
   //see if event fired by toDoList
   if(src == toDoList)
   {
      //get the index number of the selected item
      int index = ((List)src).getSelectedIndex();
      //toggle done variable of corresponding Task
      tasks[index].setDone(!tasks[index].isDone());
      //get the selected Task object
      //Task t = ((List)src).getSelectedItem();
      //toggle done variable of the Task
      //t.setDone(!t.isDone());
   }
}
```

Now that we have recorded the user action within the applicable `Task` object, we need to use that information to initialize the check box returned by the `getListCellRendererComponent` method of `ToDoListRenderer` class—the custom renderer used for `toDoList`. This is done quite easily when initializing the check box, as shown by the highlighted statement below.

```
public Component getListCellRendererComponent(List list,
            Object value, int index, boolean isSelected)
{
   //cast the value object into a Task
   Task task = (Task)value;

   //get the text of the Task and set it for this check box
   setText(task.getText());

   //set state of this check box as per 'done' variable of Task
   setSelected(task.isDone());

   //set transparency
   getStyle().setBgTransparency((byte)200);

   //set background and foreground colors
   //depending on whether the item is selected or not
   if(isSelected)
   {
      getStyle().setBgColor(0x0000ff);
      getStyle().setFgColor(0xffffff);
   }
else
   {
      getStyle().setBgColor(0xffcce0);
      getStyle().setFgColor(0);
   }

   return this;
}
```

Now, we have a list in which each element is a check box. The state of each check box can be set to `selected` or `cleared` and then saved, so that it can be displayed properly on the element.

Control for one visual aspect has been added in this example. The gap between two successive items on the list has been set to zero through the following statement:

```
//set gap between items
toDoList.setItemGap(0);
```

The ComboBox

A `ComboBox` represents a combination of a list and a button. Normally, only the selected item is visible. Clicking on the button makes the entire list visible in a (usually) drop-down format, from which an item can be selected. The `ComboBox` class extends `List`, and the flexible rendering model of a list is applicable to it. A combo box is good for optimum use of screen real estate, because only one item of the underlying list needs to be drawn on the screen, except when a selection is being made.

Creating a ComboBox

A `ComboBox` has four constructors which are just like those of `List`. Whenever we want to instantiate a combo box, we first need to define, as an array or a vector, the set of items that will form the list for the combo box instance. This set of items can then be used either directly or via a `ListModel` to formulate a combo box. An alternate approach is to create a combo box with an empty `DefaultListModel`, and then set the model through the `setModel(ListModel m)` method.

The four constructors are:

Constructor	Parameters	Description
`ComboBox()`		Creates a new instance of `ComboBox` with an empty default list model.
`ComboBox(ListModel model)`	`model`—the model instance	Creates a new instance of `ComboBox` with the given list model.
`ComboBox(java.lang.Object[] items)`	`items`—the array containing items to be placed into the model	Creates a new instance of `ComboBox` with a default list model containing the given array.
`ComboBox(jav.lang.Vector items)`	`items`—the vector containing items to be placed into the model	Creates a new instance of ComboBox with a default list model containing the given vector.

The methods of the ComboBox class

The methods of the `ComboBox` class too are similar to the ones of `List`. In the following sections, we shall construct two combo boxes. The first one will use the default rendering classes of the library, while the second one will plug in a custom renderer. These examples will show us how to use the various methods of the `ComboBox` class.

A combo box with the default renderer

A combo box with the default look and feel would appear as shown in the following screenshot.

The combo box shown above looks the way it normally would. If we click on it when it has focus or press the pointer on it, then the underlying list will drop down, as shown in the following screenshot:

As we look through the listing of the `DemoComboBox` MIDlet, we see that it is very similar to that of the basic list that we had set up earlier in this chapter. There are two things though that we have done differently here.

The first is the way in which we have gone about creating the combo box. Here we have not put the items in a vector or an array. Instead, we have used a constructor without any parameters and have then used the `addItem` method to insert the items into the combo box.

```
//create a combo box
ComboBox conCombo = new ComboBox();
//add items to combo box
conCombo.addItem("Africa");
conCombo.addItem("Asia");
conCombo.addItem("Australia");
conCombo.addItem("Europe");
conCombo.addItem("North America");
conCombo.addItem("South America");
```

The second difference is that we have not explicitly created a `DefaultListCellRenderer`. This is in accordance with what we learnt while discussing the code for the basic list example. To set style attributes, we get the `DefaultListCellRenderer` instance and work with its style object.

```
//get the DefaultListCellRenderer instance and set its style
DefaultListCellRenderer dlcr =
(DefaultListCellRenderer)conCombo.getRenderer();
dlcr.getStyle().setFgColor(0xff0000);
dlcr.getStyle().setBgSelectionColor(0x4d80d0);
dlcr.getStyle().setFgSelectionColor(0x00ff00);
```

A combo box with a custom renderer

We can plug in a custom renderer for a combo box just as we did for lists. In this section, we shall build an example with two combo boxes that use radio buttons as rendering components. We can see how this demo application looks in the following screenshot:

The combo boxes are created just as in the previous example, except for the fact that an anonymous array is passed to the constructor for the second combo box.

```
//create second combo box
ComboBox combo2 = new ComboBox(new String [] {"Car",
                  "Truck", "Locomotive", "Tanker"});
```

The renderer returns a radio button for each element, just as a check box was returned in the `ToDoList` example. Note that the same renderer instance has been used for both combo boxes. This is because every time when a rendering component is asked for, initialization is done afresh, thus preventing any crosstalk. This reuse minimizes memory requirement as the same radio button instance is used over and over again without new radio buttons being created.

A rendering prototype has been used only for the first combo box, and we can see how the selected element for the second one has been truncated.

As we scroll through the components on a form, the component that receives focus invokes a focus event, and adding a `FocusListener` allows a callback to be received by the designated object. A form differs a little in this respect, as it sends a focus event for every focus change within it.

For the second combo box, a `FocusListener` has been added, which displays a dialog to inform us that `combo2` has gained focus. A static method of the `Dialog` class has been used here to keep the dialog displayed for 2500 milliseconds.

```
//add listener to second combo box
//so that we know about focus change
combo2.addFocusListener(new FocusListener()
   {
      //show dialog for 2500 millis informing focus change
      public void focusGained(Component cmp)
      {
         Dialog.show(null, new Label("combo2 gained focus"),
                        null, Dialog.TYPE_INFO, null, 2500);
      }

      public void focusLost(Component cmp)
      {
      }
   });
```

The approach shown here for showing the dialog is very handy when simple messages need to be displayed as an alert. The type of dialog specified here is `TYPE_INFO`. Several other types are defined in the `Dialog` class. The type is an indication of the type of tone to be played, or the icon if none is specified.

The following screenshot shows the dialog being displayed:

We could also have sensed and responded to a loss of focus by taking some action within the `focusLost` method. Although we are not doing anything here when `combo2` loses focus, an empty implementation of the `focusLost` method is necessary. Otherwise there will be a compilation error.

Summary

The five examples of this chapter have shown us how flexible a `List`, and its subclass `ComboBox`, can be. The foundation of this flexibility lies in the mechanism for plugging in the custom look and feel code. Some of our examples have exploited this mechanism to enhance the appearance and functionality of lists and combo boxes. In this chapter, we have learnt:

- Different ways of creating lists and combo boxes.
- How to determine the basic behavior of lists and combo boxes. For example, a list can be either cyclic or non-cyclic.
- How custom tailored objects can be used as list elements.
- About the structure of renderers and how to use them.
- How listeners can be used to track user actions on lists and combo boxes.

The examples have been designed to demonstrate that there are different ways of handling these widgets, and more importantly, to show the philosophy behind their designs so that we can extrapolate from what has been done here and achieve greater functional complexity, as well as visual sophistication.

6 TextArea and TextField

Display and editing of text are probably the most commonly used features of software applications. As small devices like mobile phones grow in terms of computational capabilities, applications running on these platforms become more complex. They are now able to address a wide variety of functional requirements. Consequently, components for handling text are highly important building blocks in libraries like LWUIT.

Consider an email client running on mobile phones. The first action on opening such an application is likely to be to log in. Subsequently, you would possibly read a mail from the **Inbox** or compose a new message. Both of these activities will require text handling. If the UI for this application is built around LWUIT, then it is very likely that the log in screen would have two **TextFields**, one for the username and the other for the password—and the mail would be displayed on an uneditable **TextArea**. Similarly, for writing a mail, an editable TextArea would be used.

In this chapter, we shall spend some time studying these two classes. The primary objective for both `TextArea` and its subclass `TextField` is to support functions that relate to attractive display and convenient (and powerful) editing of textual matter. A very powerful feature of a text field is the ability to edit on the widget itself (*in place* or *in situ* editing) without using a native text box. This ensures a device-independent text entry and editing environment.

As usual, our emphasis will be on the usage of the components. We shall build an example for each widget—one step at a time with detailed explanation of what is being done. Thus, each example will provide us with exposure to constructors and methods for effective use of text areas and text fields. Now, let the action begin!

The TextArea

A text area provides a space for displaying text that can vary in length. When a single-line text area is created, it does not grow beyond one line. A multiline text area has the ability to grow as the content grows. A text area is editable by default. Later in this section, we shall build a text area and experiment with its capabilities.

Creating a TextArea

The `TextArea` class has the following constructors:

Constructor	Parameters	Description
`TextArea()`		Creates a single-line empty text area that will not grow.
`TextArea(int rows, int columns)`	`rows`—number of rows `columns`—number of columns	Creates an empty text area with the given number of rows and columns.
`TextArea(int rows, int columns, int constraints)`	`rows`—number of rows `columns`—number of columns `constraints`—one of the constraints described below	Creates an empty text area with the given number of rows, columns, and constraints.
`TextArea(String text)`	`text`—the initial content to be displayed. If text is null, then the empty String – "" – will be displayed	Creates a single-line text area with the given text that will not grow.
`TextArea(String text, int maxSize)`	`text`—the initial content to be displayed. If text is null, then the empty String – "" – will be displayed `maxSize`—maximum number of characters the text area can hold	Creates a single-line text area with the given text and the specified maximum size.
`TextArea(String text, int rows, int columns)`	`text`—the initial content to be displayed. If text is null, then the empty String – "" – will be displayed `rows`—number of rows `columns`—number of columns	Creates a text area with the given text, number of rows, and number of columns.

Constructor	Parameters	Description
`TextArea(String text, int rows, int columns, int constraints)`	`text` — the initial content to be displayed. If text is null, then the empty String – "" – will be displayed `rows` — number of rows `columns` — number of columns `constraints` — one of the constraints described below	Creates a text area with the given text, number of rows, number of columns and constraints.

A text area can be created to hold a predetermined type of string through the use of a **constraint**. A constraint can also be used to indicate the behavior of a text area or the way it handles its content. The following constraints are available.

Constraint	Description
`ANY`	Allows any type of input. This is the default value.
`EMAILADDR`	Allows entry of email addresses.
`NUMERIC`	Allows entry of only integer values.
`PHONENUMBER`	Allows entry of phone numbers.
`URL`	Allows entry of a URL.
`DECIMAL`	Allows entry of numeric values with optional fractions.
`PASSWORD`	Indicates that the input should be obscured if possible.
`UNEDITABLE`	Indicates that content cannot be edited.
`SENSITIVE`	Indicates that the content is sensitive and should not be saved in an accelerated input scheme.
`NON_PREDICTIVE`	Indicates that the content is made up of words unlikely to be found in dictionaries.
`INITIAL_CAPS_SENTENCE`	A hint to the implementation that, while editing, the first letter of each sentence should be capitalized.
`INITIAL_CAPS_WORD`	A hint to the implementation that, while editing, the first letter of each word should be capitalized.

It is permitted to combine any of the first six constraints with any of the rest by using the `bitwise or` function. For instance, we can use the combination `ANY|PASSWORD` so that any character can be used in a password, which shall be in a form that is not human readable.

The constraints are actually *hints* and *indications*. Whether a constraint is actually implemented depends upon the platform.

The methods of the TextArea class

Like the other widgets, `TextArea` has methods that support the basic functionalities of a component. In addition to these, it also has methods that are tailored for processing and presentation of text data. In the next section, we shall try out most of these methods on a demo application.

Putting TextArea class through its paces

Our application is implemented through the `DemoTextArea` MIDlet, which first sets up a form. This part of the MIDlet works in the same way as all the examples in the preceding chapters. Applying style attributes to the text area also proceeds along familiar lines. We then set up a blank text area with 5 rows, 10 columns, and permit the entry of all types of input.

```
TextArea demoText = new TextArea(5, 10, TextArea.ANY);
```

By default, a text area can hold a maximum of 124 characters. If we try to enter more than 124 characters, then the input will not be accepted. It is possible to increase this limit in two ways. Both these ways are shown with commented out statements in our example.

```
//TextArea.setDefaultMaxSize(500);
.
.
.
//demoText.setMaxSize(500);
```

The first statement calls a static method to set the maximum size for all text areas to be subsequently instantiated, while the second one sets an enhanced limit only for the referenced instance.

The text area that we have created looks like this:

We could have also used a constructor that does not specify rows and columns.

```
TextArea demoText = new TextArea();
```

In that case, the text area would appear as shown in the following screenshot:

The *Select* key needs to be clicked on to enter text into the text area. This will open a `javax.microedition.lcdui.TextBox` for editing. By default, a text area is editable. Editability can be disabled by calling the `setEditable` method as follows:

```
demoText.setEditable(false);
```

Text can be entered using the keyboard of the actual physical device on which the application is running. Clicking on the **Cancel** command will abort editing, while the **Menu** command provides an option for saving the entered text so that it can be displayed in the text area.

Although we can specify the size of a text box (in terms of rows and columns) while creating it, a text box can be subsequently resized. The following statements, when uncommented, will alter the number of rows and columns.

```
//demoText.setRows(2);
//demoText.setColumns(5);
```

The resulting text box will look like this:

When the content grows beyond a single line, the text is rendered with a default gap of two pixels between successive lines.

This gap between the two lines can be changed by using the following statement:

```
demoText.setRowsGap(20);
```

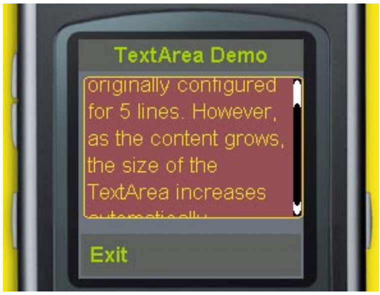
This increases the interline gap to 20 pixels, as we can see in the following screenshot:

A text area has the flexibility to grow beyond the rows and columns originally set by the constructor. This is a default property, and we can see the increase in size as we enter input beyond the original capacity. However, this elasticity is subject to the limit set by the maximum size defined for the instance.

This automatic increase in size can be disabled by using the statement below:

```
demoText.setGrowByContent(false);
```

The fact that the text area will not grow with increasing content does not mean that input beyond the original setting for the number of rows will be restricted. What will happen is the dimensions of the text area will remain fixed at the original value, and a scrollbar will be added when required.

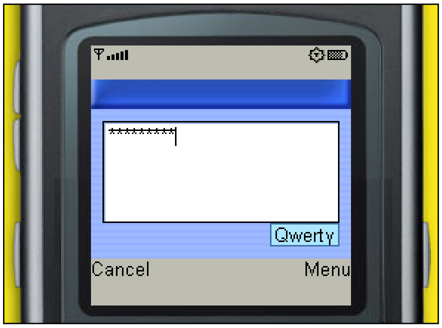


A text area, in its basic form, is unable to handle formatting characters like `"\\t"`, the tab character. Furthermore, certain font settings may cause problems with some specific characters. The `TextArea` class has two methods that allow developers to make provisions for handling such cases. The first method is `setUnsupportedChars(String unsupportedChars)`, which defines a string of problematic characters that can be removed while rendering the text area content. The other method is the protected method `preprocess(String text)`, which can be used in a subclass of `TextArea` to replace such characters as `"\\t"` with a predefined number of spaces to implement tab setting.

Earlier we had seen that the `TextArea` class provides a set of *constraints* that allow us to specify the nature of the input that can be accepted by the text area instance—subject to native platform support. One of the common types of input that an application usually handles is a password. A password may be allowed to include any character. However, the actual entry is required to be obfuscated by substituting a symbol (usually '`*`') for the characters entered. To implement this capability in our text area, we need to use the statement shown:

```
demoText.setConstraint(TextArea.ANY|TextArea.PASSWORD);
```

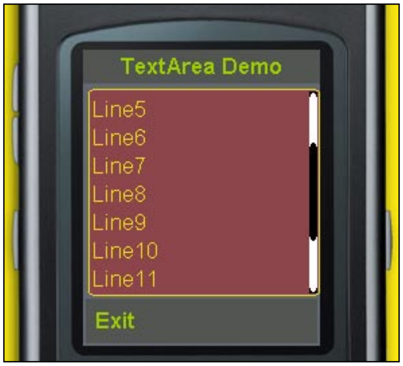
This will make sure that the password is hidden during editing as shown in the following screenshot:



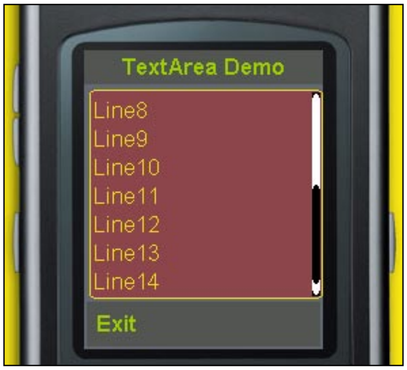
When the password has been entered and the main screen is displayed, the password continues to remain hidden.

An interesting feature of `TextArea` is that we can control the number of lines for scrolling. The following statements will set a multiline text, and specify that scrolling should move the text by three lines.

```
//set a multiline text
demoText.setText("Line1\nLine2\nLine3\nLine4\nLine5\nLine6\nLine7\
nLine8\nLine9\nLine10\nLine11\nLine12\nLine13\nLine14\nLine15");
//set number of lines for scrolling
demoText.setLinesToScroll(3);
```



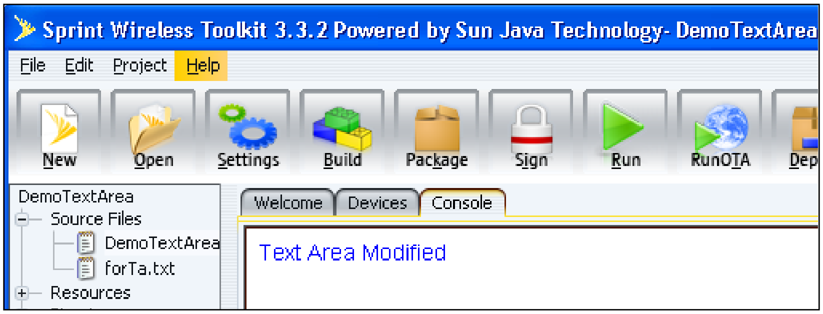
The previous screenshot shows that **Line5** is the first line on the text area. Now, if we scroll down by pressing the *Down* button once, then we can expect **Line8** to take the top-most position. The next screenshot shows that our objective has been achieved:



A text area conveys that it has been modified by firing an `ActionEvent`. In the following snippet, we add an `ActionListener` to print a message on the console:

```
demoText.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent ae)
    {
        System.out.println("Text Area Modified");
    }
});
```

Now, if we edit the text, the message will be printed when the editing is completed, and the text area comes back on the screen.



The `TextArea` class provides a multiline space for text display. In order to create a single-line text display that can allow *in situ* editing without invoking a text box, we can use an instance of `TextField`, as we shall see in the next section.

The TextField class

The `TextField` class is an inelastic single-line version of the `TextArea` class that can be edited optionally without opening a separate screen. While it never grows beyond a single line, it can accept long texts. The limitation lies in the fact that only one line will be displayed, and the line width will be limited by the available screen area.

Creating a TextField

We can choose from a set of four constructors for creating an instance of the `TextField` class.

Constructor	Parameters	Description
`TextField()`		Creates an empty text field.
`TextField(int columns)`	`columns`—the number of columns	Creates an empty text field with the given number of columns.
`TextField(String text)`	`text`—initial text to be displayed	Creates a text field with the given initial text.
`TextField(String text, int columns)`	`text`—initial text to be displayed `columns`—the number of columns	Creates a text field with the given initial text and the number of columns.

When we compare the list of constructors mentioned in the given table with that for `TextArea`, we can see that there is no mention of *constraints* here. This is because constraints are not fully implemented for this class, and the only constraint that works for a text field is `TextArea.PASSWORD`.

The methods of the TextField class

A significant difference between `TextField` and `TextArea` is that the former supports "in place" editing, that is, editing without opening a native text box. So we find a wide range of methods that deal with the various aspects of *in situ* editing here. In addition to these, the `TextField` class has methods for styling and for specifying other behavioral aspects like the corresponding methods of `TextArea`. The `TextField` class also inherits a number of methods from its superclass `TextArea`.

We shall now build a `TextField` example, just as we did for `TextArea`, and see how this widget can be configured using the constructors and methods of the class.

Checking out TextField

The first part of the MIDlet—`DemoTextField`—initializes `Display` and creates the form. This part is the same as in our other examples. The difference begins when we add more commands to the form than we have been doing so far.

```
//commands for changing text field behaviour
Command REPLACE_CMD = new Command("Overwrite");
Command INSERT_CMD = new Command("Insert");
.
.
.
demoForm.addCommand(new Command("Exit"));
demoForm.addCommand(new Command("Resize"));
demoForm.addCommand(REPLACE_CMD);
```

As the form now has three commands and we are using it in the two soft button mode, the last two commands will be shown through a menu. Commands on a menu are implemented as a list. The styling for this can be done through the corresponding renderer, which is the `MenuCellRenderer` in this case. So our next step is to install the `MenuCellRenderer` and to style it after setting the base (background) color for the menu.

```
demoForm.getMenuStyle().setBgColor(0x555555);
DefaultListCellRenderer dlcr = new DefaultListCellRenderer();
demoForm.setMenuCellRenderer(dlcr);
Style mStyle = new Style();
mStyle.setFgColor(0x99cc00);
```

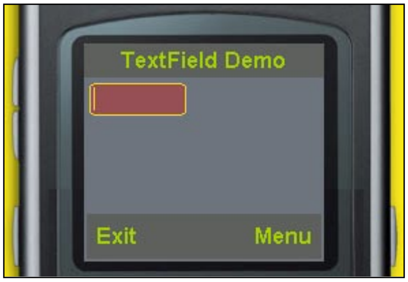
```
mStyle.setFgSelectionColor(0xffffff);
mStyle.setBgSelectionColor(0x0000ff);
mStyle.setFont(Font.createSystemFont(Font.FACE_SYSTEM,Font.
                          STYLE_PLAIN,Font.SIZE_MEDIUM));
dlcr.setStyle(mStyle);
Style sbStyle = new Style();
sbStyle.setBgColor(0x555555);
sbStyle.setFgColor(0x99cc00);
sbStyle.setFont(Font.createSystemFont(Font.FACE_PROPORTIONAL,
                          Font.STYLE_BOLD,Font.SIZE_MEDIUM));
UIManager.getInstance().setComponentStyle(
                   "SoftButton", sbStyle);
```

The above snippet also shows that the style for all soft buttons has been set. This determines the styling for the soft buttons associated with the menu. However, the soft buttons for the form have been styled separately, and they are not affected by the styling done here.

Having taken care of the menu, we now turn our attention to creating a text field. Before invoking a text field constructor, we shall install a style for the text fields.

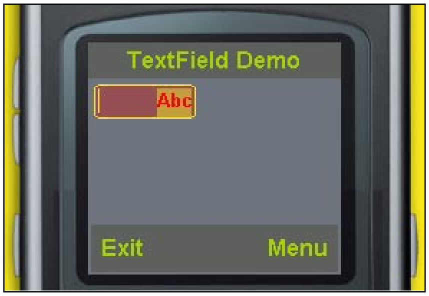
```
Style txtStyle = new Style();
txtStyle.setFgColor(0xe8dd21);
txtStyle.setFgSelectionColor(0xe8dd21);
txtStyle.setBgColor(0xff0000);
txtStyle.setBgSelectionColor(0xff0000);
txtStyle.setBgTransparency(80);
txtStyle.setFont(Font.createSystemFont(Font.FACE_PROPORTIONAL,
                          Font.STYLE_BOLD,Font.SIZE_MEDIUM));
txtStyle.setBorder(Border.createRoundBorder(10, 7, 0xe8dd21));
UIManager.getInstance().setComponentStyle(
                   "TextField", txtStyle);
demoText = new TextField(5);
```

This results in a text field without any initial content and with a width of 5 columns. We can see this text field in the following screenshot. Note the cursor shown as a line at the left edge of the text field.

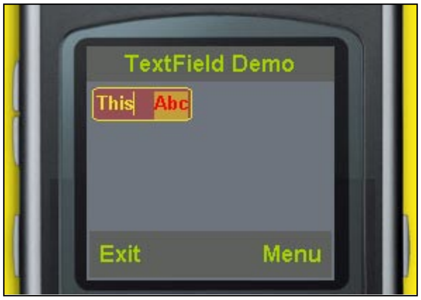


A click on the **Menu** soft button causes the menu to pop up. The styling of the menu can be seen clearly in the following screenshot:

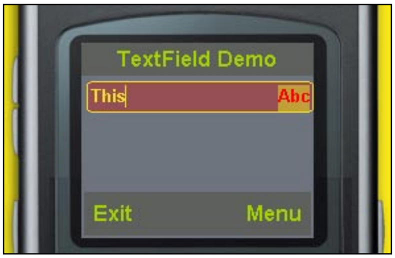
It is not necessary to use the menu to enter text. All we need to do is click on the *Select* key. This sets up the text field for editing in place without opening a native text box. The input mode is shown on the right edge. In this case, the mode is the normal text entry mode in which the first letter of a sentence is capitalized.



Text entry is now possible through a numeric keypad, as we do on a phone; press key '2' once to enter 'a' and twice in rapid succession to enter 'b' and so on.



If we continue to enter text, then it will be accepted, but only a small part of it will be displayed. However, we can resize the text field if we want by selecting the **Resize** command on the menu.



The resizing code is in the `actionPerformed` method of the MIDlet:

```
public void actionPerformed(ActionEvent ae)
{
   Command cmd = ae.getCommand();
   if(cmd.getCommandName().equals("Exit"))
   {
            notifyDestroyed();
   }
   else
   {
      if(cmd.getCommandName().equals("Overwrite"))
      {
         demoText.setOverwriteMode(true);
         demoForm.removeCommand(REPLACE_CMD);
         demoForm.addCommand(INSERT_CMD);
      }
      else
      {
         if(cmd.getCommandName().equals("Insert"))
         {
            demoText.setOverwriteMode(false);
            demoForm.removeCommand(INSERT_CMD);
            demoForm.addCommand(REPLACE_CMD);
         }
         else
         {
            if(cmd.getCommandName().equals("Resize"))
            {
               if(demoText.getColumns() < 20)
               {
```

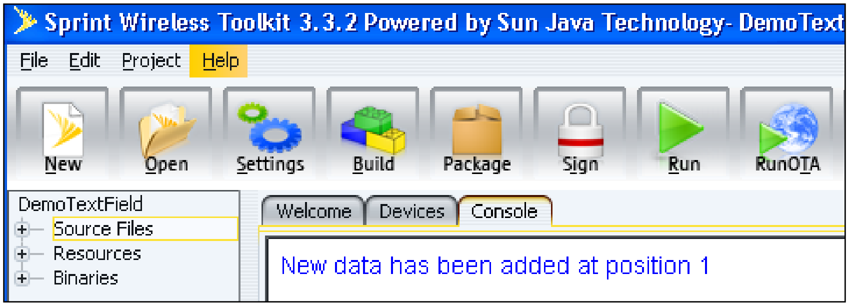
```
                    demoText.setColumns(20);
                  }
                else
                  {
                  demoText.setColumns(5);
                  }
                }
              }
            }
          }
          .
          .
        }
```

The `setColumns` method is used to change the size of `demoText` on the fly. The code uses the `getColumns` method to access and check the present size of the text field. If the size is less than 20 columns, then it is increased to 20. Otherwise, it is reduced to 5. The **Resize** command thus effectively toggles the size of `demoText` between 5 and 20 columns.

The other commands that are handled in the `actionPerformed` method (other than the ubiquitous **Exit** command) are **Overwrite** and **Insert**. These commands control the mode of text entry—in the *Insert* mode, new characters are inserted at the cursor position, whereas in the *Overwrite* mode, the character at the cursor position is replaced by the one entered. The code for handling these commands ensures that only one of these two commands appears on the menu at a particular time. So, while the *Insert* mode is active, the **Overwrite** command is on the menu and vice versa. The default mode is *Insert*.

The other input mode, as we have already noted, is the one that is shown on the right edge of the text field while editing. During text entry, we can select other modes (numeric, all caps or all lower case) by pressing the # key. Incidentally, the mode control function of this key can be shifted to any other key by calling the `setDefaultChangeInputModeKey(int keycode)` method, which sets the key specified by the parameter *keycode* as the one that will change this input mode.

The `TextField` class allows us to track changes in its content through a `DataChangedListener`. The `addDataChangeListener` method can be called to add a `DataChangedListener` as we have done here. This listener has to implement the `dataChanged` method, which gets two parameters—`type` and `index`, both of which are `int`s. The parameter *type* indicates what kind of change has taken place. The possible values are `ADDED`, `CHANGED`, and `REMOVED`, and the corresponding `int` values are 1, 2, and 0. The other parameter—*index* is the position where the change has taken place. Our new `DataChangedListener` prints out a message when a new character is entered.



Let us now take a look at the menu as it appears during editing. You will see that two new commands have been added. These commands are **Clear** and **Edit**.

These commands are added by LWUIT when in-place editing is initiated. The actual name of the command that appears as **Edit** here is **T9**, and it opens a native text box. We have used the static method `setT9Text` to change this name to **Edit**. As this is a static method, it effects the change for all text fields that are instantiated after this method is invoked. In our MIDlet, the statement for renaming the **T9** command is placed just before `demoText` is created.

When these two commands are added, the original commands of the form are removed by default. To prevent the form commands in our example from being removed, the following statement has been used:

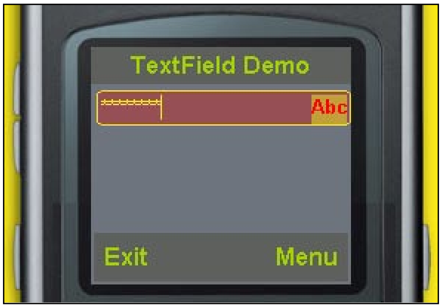
```
demoText.setReplaceMenu(false);
```

Note that the handling of commands here is different from that in our other examples. Here we act on the basis of a command's name rather than its ID. One reason for doing this is to emphasize the convenience (and neatness) of using the ID-based approach. But there is another reason too. When a command is created using only its name, the ID defaults to `0`. The IDs of both commands added for editing (**Clear** and **Edit**) have this default value, and this would cause a problem since our **Exit** command also has an ID of `0`. Of course, we could have solved this problem in other ways too. For example, we could have used non zero IDs for our commands. However, I wanted to use this opportunity to show how neat the ID based approach can be, especially when a significant number of commands have to be acted on.

The `TextArea` class uses a number of constraints as an indication for entering special types of data such as a password. The `TextField` class, as we know, ignores all such fields except `TextField.PASSWORD`. As in a text area, the following line of code hides a password in a text field:

```
demoText.setConstraint(TextArea.PASSWORD);
```

A password entered would now look like this.



`TextField`, unlike `TextArea`, displays a blinking cursor to mark the current text entry position. We can shift the cursor programmatically, and we can also change the blink on and off times. To set the blink times, uncomment the following lines in the MIDlet. You will see that the default blink on time (800 milliseconds) and off time (200 milliseconds) have changed, and the cursor now stays visible for 2 seconds and invisible for 2 seconds.

```
//demoText.setCursorBlinkTimeOn(2000);
//demoText.setCursorBlinkTimeOff(2000);
```

We can shift the cursor position quite easily. Uncomment the statements shown below (the first one in the `startApp` method and the rest in the `actionPerformed` method). A new command, **Home**, will now be added to the menu. Enter some text, and then select the **Home** command. The cursor will move to the extreme left position.

```
//demoForm.addCommand(new Command("Home"));
.
.
.
/*if(cmd.getCommandName().equals("Home"))
{
   demoText.setCursorPosition(0);
}*/
```

During in-place editing, the interval for automatic commit of an entry is 1 second. If the same key press is repeated within this time, then it is considered to be a multiple stroke, and the character to be entered is determined accordingly. On the other hand, if no key press occurs within this time, then the character determined by the last press is committed. The default value of 1 second for the commit timeout can be changed through the `setCommitTimeout(int commitTimeout)` method, where the `commitTimeout` parameter is the desired new value in milliseconds.

Summary

In this chapter, we have seen how to use text areas and text fields. These two widgets provide very flexible but easy to use UIs for the display, entry, and editing of text. A text area supports an operating region with adjustable dimensions that can be larger than that available with a text field and with input constraints that permit us to tailor the component instance for specific types of input. This makes a text area suitable for showing and writing comparatively large and varied text contents. On the other hand, a text field is a component that is very convenient for short texts. A special feature of text fields is that they can be edited in-place through a simple but highly versatile interface. Together, these two components provide attractive text handling capabilities with a uniform look and feel over a wide range of devices.

7
Arranging Widgets with Layout Managers

Arranging components on a screen can be a tricky business. For instance, any manual arrangement that depends on a particular display dimension to look good is very likely to disappoint on a device with a different screen size. LWUIT provides a number of classes that automate the process of laying out components on a container. These *layout managers* produce screen arrangements that are device independent to a considerable extent. However, each class has its own characteristics and needs to be used while keeping its capabilities in mind.

The root class for the layout classes is **Layout**, and it has six subclasses that perform component arrangement, each in accordance with its operating algorithm. The six layout classes are:

- `BorderLayout`
- `BoxLayout`
- `CoordinateLayout`
- `FlowLayout`
- `GridLayout`
- `GroupLayout`

There is also a supporting class `LayoutStyle` that takes care of the gaps between two adjacent components and between a component and an adjoining edge of its parent container.

In this chapter, we shall study all these classes, and as usual, we will analyze sample code to see how they can be used.

The demos for this chapter are derived from the `DemoLabel` MIDlet. This MIDlet has been modified to include the code for all six types of layout managers in clearly demarcated sections. The sections are commented out, and the relevant portion will have to be uncommented to try out an example of a specific type of layout. Just make sure that the sections for other layout types have all been commented out.

Layout class

This is an abstract class that embodies the quintessential qualities of all the layout managers. All the six layout classes are subclasses of `Layout`.

The default and only constructor of this class is `Layout()`. However, we cannot directly create an instance of this class, as it is abstract. To use a specific type of layout, we have to instantiate an object of the corresponding class.

The methods of `Layout` provide essential support for laying out elements on a container. The following table lists these methods:

Method	Parameters	Description
`void addLayoutComponent(Object value, Component comp, Container c)`	`value`—optional metadata information like alignment orientation. `comp`—component to be added. `c`—parent container.	Provides an option allowing users to furnish hints on object positioning.
`Object getComponentCostraint(Component comp)`	`comp`—component whose constraint is to be returned.	Returns the optional constraint of the specified component such as the compass direction for a component specified in a border layout.
`Dimension getPreferredSize(Container parent)`	`parent`—the parent container	Returns the preferred size of the container. This is an abstract method that must be implemented by concrete subclasses.
`boolean isOverlapSupported()`		Returns `true` if components are allowed to overlap.
`void layoutContainer(Container parent)`	`parent`—the parent container for which layout is to be done.	Arranges elements for the container. This is an abstract method that must be implemented by concrete subclasses.

Method	Parameters	Description
`void removeLayoutComponent (Component comp)`	`comp`—the component to be removed.	Removes the specified component from the layout. This method will work only if the layout manager maintains references of the components laid out by it.

The LayoutStyle class

The `LayoutStyle` class is used for specifying the spacing between two components or between a component and an edge of the parent container. The spacing can depend upon whether the components are logically grouped together or not. The grouping relationship is used for accessing the preferred gap between components. The field `LayoutStyle.RELATED` indicates a logical grouping, while `LayoutStyle.UNRELATED` indicates that two components under reference are not related.

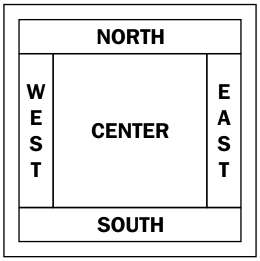
`LayoutStyle` does have a default constructor, but the usual way of getting an instance of this class is to use the static method `getSharedInstance()`, which returns the appropriate layout style instance.

The two other methods of this class provided for general use are tabulated below:

Method	Parameters	Description
`int getContainerGap( Component component, int position, Container parent)`	`component`—the component to be positioned. `position`—the position of the component relative to the container. `parent`—the parent container.	Returns the space between the component and the container edge. The specified position must be one of the layout constraints defined in `GroupLayout`.
`int getPreferredGap( Component component1, Component component2, int type, int position, Container parent)`	`component1`—the reference component for placing component2. `component2`—the component being placed. `type`—the positional reletionship as specified by RELATED, UNRELATED or INDENT. `position`—the desired position of component2 relative to component1. `parent`—the parent container.	Returns the spacing to be used between the two components. The specified position must be one of the layout constraints defined in `GroupLayout`.

BorderLayout

`BorderLayout` provides five positions for placing widgets on a screen—NORTH, SOUTH, EAST, WEST, and CENTER. The figure below shows how these positions are defined.



`BorderLayout` inserts components in the following order.

Position	Insertion Order	Remarks
NORTH	First	Expands laterally to occupy all available horizontal space. Height determined by content.
SOUTH	Second	Expands laterally to occupy all available horizontal space. Height determined by content.
EAST	Third	Expands to occupy all available vertical space. Width determined by content.
WEST	Fourth	Expands to occupy all available vertical space. Width determined by content.
CENTER	Last	Expands to occupy all remaining space.

In order to see how a border layout is used, refer to the code for the `DemoLayout` MIDlet. The relevant section for border layout is given below:

```
/*****Start of BorderLayout statements*****/

 BorderLayout testLayout = new BorderLayout();
   demoForm.setLayout(testLayout);
   demoForm.setTitle("BorderLayout");

   //demoForm.setScrollableX(true);

   //tLabel.setFocusable(true);
   //imLabel.setFocusable(true);
   //bothLabel.setFocusable(true);
```

```
        demoForm.addComponent(BorderLayout.NORTH, tLabel);
        //demoForm.addComponent(BorderLayout.EAST, tLabel);

        demoForm.addComponent(BorderLayout.SOUTH, imLabel);
        //demoForm.addComponent(BorderLayout.WEST, imLabel);

        demoForm.addComponent(BorderLayout.CENTER, bothLabel);

    /*****End of BorderLayout statements*****/
```

To create an instance of `BorderLayout`, we use the default (and only) constructor of the class, and install it on `demoForm` using the `setLayout(Layout layout)` method. At this point, let us recollect that the default layout manager for `Form` is the `BorderLayout` (refer to Chapter 3). One can therefore legitimately ask why is it necessary to install a brand new instance of `BorderLayout`, if a form already has a default border layout. The answer is that components are not added directly to a form but to its content pane, and the default layout manager for a content pane is `FlowLayout`. The `setLayout` method actually installs a layout manager for the content pane.

By default, a form is not scroll enabled along the x-axis. Calling the `setScrollableX` method on a form enables the horizontal scroll bar. The next three statements explicitly override a default characteristic of labels and make them capable of receiving focus. We do all this to see how screens are laid out when there are more widgets than the layout manager can accommodate within the width of the device. These statements have been commented out in the code listing. We shall uncomment them later to check out their effect on the screen layout.

Let us now add three labels in the NORTH, SOUTH, and CENTER positions. The method that has to be used is `addComponent(Object constraints, Component cmp)`, where the value of the desired position is passed as `constraints` and can be one of NORTH, SOUTH, EAST, WEST or CENTER. The first combination we try is:

Component	**Position**
`tLabel`	`NORTH`
`imLabel`	`SOUTH`
`bothLabel`	`CENTER`

The screen looks like this:

As expected, `tLabel` was added first. Its height was determined by the font size of its text, and the width was increased to take up all the horizontal space on the form. The same holds true for `imLabel`, which was added next. Finally, `bothLabel` was added, and all the remaining space was allocated to it.

Next, we shall change the positioning, as specified by the statements shown below.

```
//demoForm.addComponent(BorderLayout.NORTH, tLabel);
demoForm.addComponent(BorderLayout.EAST, tLabel);

//demoForm.addComponent(BorderLayout.SOUTH, imLabel);
demoForm.addComponent(BorderLayout.WEST, imLabel);

demoForm.addComponent(BorderLayout.CENTER, bothLabel);
```

The result is shown in the following screenshot.

We see here that `bothLabel` has practically disappeared. The sequence of component insertion by `BorderLayout` explains this. The first component to be handled this time was `tLabel`. Its width was determined by the width of the string, and its height was set so that all the available vertical space could be occupied. The next component to be sized was `imLabel`, and the horizontal space allocated to it was calculated on the basis of the image size. By the time the last component—`bothLabel`—could be taken up, the remaining space along the x-axis was only sufficient to render the borders.

The solution to this problem is to enable the horizontal scrollbar and to make the labels focusable. We can now see that the scrollbar has been added for the x-axis. Also, the directional navigation keys can be used to move focus from one label to another, thereby scrolling the screen to the left or to the right.

Before moving on, let's try one more combination of positions.

```
demoForm.addComponent(BorderLayout.NORTH, tLabel);
//demoForm.addComponent(BorderLayout.EAST, tLabel);

//demoForm.addComponent(BorderLayout.SOUTH, imLabel);
demoForm.addComponent(BorderLayout.WEST, imLabel);

demoForm.addComponent(BorderLayout.CENTER, bothLabel);
```

This time `tLabel` will be added first and can be expected to take up all the horizontal space along its x-axis. `imLabel` is added next and will take up the vertical space left after sizing `tLabel`. Finally, `bothLabel` will get the remaining space. The following screenshot shows that this is indeed what happens:

When components on a screen are arranged without due consideration for an appropriate layout manager, the result can be disappointing if the application runs on devices with different screen dimensions. In order to appreciate this, let us see what happens when we run the `DemoButton` example (using `FlowLayout`) that we encountered in Chapter 4 on a device with a smaller screen than that of the device used earlier.

The resulting display is quite different from what we saw in the original example and is definitely not what we would want. The consistency of a layout can improve significantly when we use `BorderLayout`. The next screenshot shows the result achieved with `BorderLayout` for a device with a comparatively large screen.

On a device with a smaller screen, the arrangement remains essentially the same with a vertical scrollbar automatically added to accommodate all the buttons within a relatively smaller area.

One limitation of `BorderLayout` is that it can handle a maximum of five components (one in each position). It also changes dimensional proportions to use up the available space, and this can distort the appearance of a widget. The choice of a layout manager is therefore dependent on the type of components that are to be laid out. One notable example of an appropriate use of `BorderLayout` is the form itself, which adds the title bar in the NORTH position, the menu bar in the SOUTH position, and the content pane in the CENTER position. Now we understand why the default layout manager of a form is `BorderLayout` and not `FlowLayout`, as in the case of a container.

BoxLayout

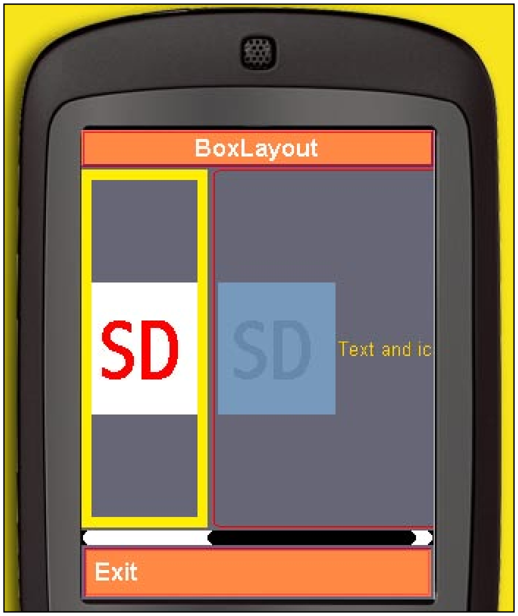
This layout manager places widgets along a row or a column in accordance with the specified orientation. The code for using `BoxLayout` in our sample MIDlet is given below.

```
/*****Start of BoxLayout statements*****/

 BoxLayout testLayout = new BoxLayout(BoxLayout.X_AXIS);
   //BoxLayout testLayout = new BoxLayout(BoxLayout.Y_AXIS);

   demoForm.setLayout(testLayout);
   demoForm.setTitle("BoxLayout");

   //demoForm.setScrollableX(true);

   tLabel.setFocusable(true);
   imLabel.setFocusable(true);
   bothLabel.setFocusable(true);

   demoForm.addComponent(tLabel);
   demoForm.addComponent(imLabel);
   demoForm.addComponent(bothLabel);

/*****End of BoxLayout statements*****/
```

`BoxLayout` has one constructor, which takes an orientation field as its only parameter. The orientation is horizontal when the parameter is `X_AXIS` and vertical when the parameter is `Y_AXIS`.

We shall first work with horizontal orientation for the box layout instance that we create and shall keep the horizontal scrollbar disabled. The resulting layout is shown in the following screenshot:

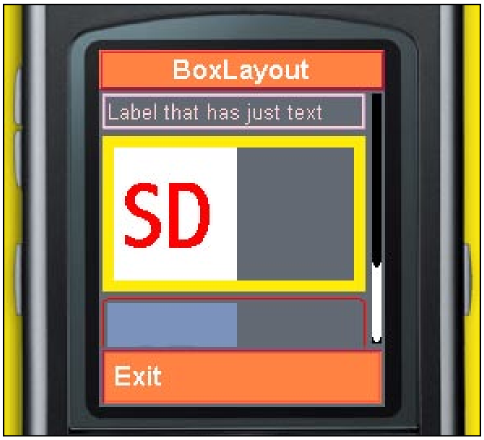
`BoxLayout` tries to use up the available space just as `BorderLayout` does. As a result, the vertical dimensions of `tLabel` and `imLabel` have been increased. Unlike `BorderLayout` however, `BoxLayout` places elements in the order in which they are added to the form. So, `tLabel` is the first to be positioned, followed by `imLabel`. By the time it is the turn of `bothLabel`, all the horizontal space has been used up, and all we see of that component is the border. The solution lies in enabling the horizontal scrollbar again. Uncommenting the relevant statement produces the following arrangement:



Now a horizontal scrollbar gets added, and `bothLabel` becomes visible.

If we change the orientation to vertical by uncommenting the second statement and by commenting out the first statement, then the labels will be laid out from top to bottom instead of from left to right, as was done for horizontal orientation.

All three labels are visible this time, but they have been expanded sideways to fill up the horizontal space. `BoxLayout`, like `BorderLayout`, preserves the spatial relationship in a device independent manner, as shown in the following screenshot:



CoordinateLayout

The `CoordinateLayout` is actually a kind of supplementary class that requires absolute placement coordinates to be pre-specified by some other class. The following snippet shows the code that we will use to see how `coordinate layout` works.

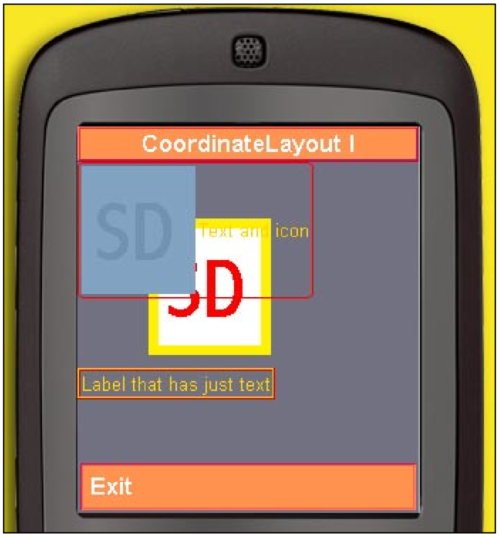
```
/**Start of CoordinateLayout statements - first set**/
 CoordinateLayout testLayout = new CoordinateLayout(demoForm.
                          getWidth(), demoForm.getHeight());
   demoForm.setLayout(testLayout);
   demoForm.setTitle("CoordinateLayout I");
   //tLabel.setY(180);
   //imLabel.setX(50);
   //imLabel.setY(50);
   demoForm.addComponent(tLabel);
   demoForm.addComponent(imLabel);
   demoForm.addComponent(bothLabel);
/**End of CoordinateLayout statements - first set**/
```

The constructor for `CoordinateLayout` takes a particular width and a particular height as parameters, which are used as the reference dimensions for defining the aspect ratio of the container. In this case, we have passed the dimensions of the form, but any other reference could have been used. The layout manager will use the ratio of the actual dimensions of the parent container and the reference dimensions to position the components.

If we run the code as it stands, we will get the following layout for the labels.

Note that all the labels have been positioned with the top left corners at coordinate (0, 0), relative to the content pane. This happens because the coordinates for positioning these components have not been specified, and the default values have been used by the layout manager.

Running the code once more after uncommenting the three commented out lines will set some values for positioning the labels, and the result will look like this:

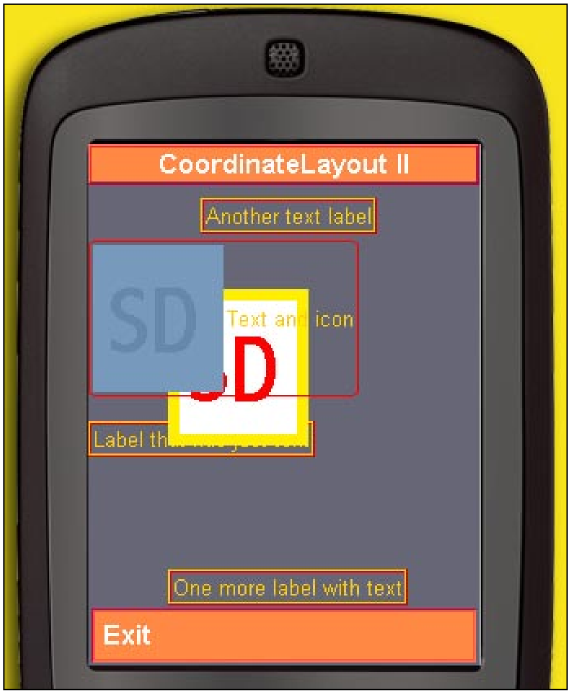


Although the labels have now been positioned at different locations, this image does not tell us anything about the actual role played by the `CoordinateLayout` class. In order to see how the `CoordinateLayout` class modifies the position of a component depending upon the container dimensions relative to the reference dimensions used in the constructor, let us use the following code.

```
/**Start of CoordinateLayout statements - second set**/

//new coordinate layout instance
CoordinateLayout testLayout = new CoordinateLayout(
        demoForm.getWidth(), demoForm.getHeight());

//set a new BorderLayout as the layout manager for demoForm
demoForm.setLayout(new BorderLayout());
demoForm.setTitle("CoordinateLayout II");

Container c1 = new Container();
Container c2 = new Container();
Container c3 = new Container();

Label tLabel2 = new Label("Another text label");
tLabel2.getStyle().setBorder(Border.createEtchedRaised());
c1.addComponent(tLabel2);

//set testLayout as the layout manager for this container
c2.setLayout(testLayout);

tLabel.setY(180);
imLabel.setX(50);
imLabel.setY(50);

c2.addComponent(tLabel);
c2.addComponent(imLabel);
c2.addComponent(bothLabel);

Label tLabel3 = new Label("One more label with text");
tLabel3.getStyle().setBorder(Border.createEtchedRaised());
c3.addComponent(tLabel3);

demoForm.addComponent(BorderLayout.NORTH, c1);
demoForm.addComponent(BorderLayout.CENTER, c2);
demoForm.addComponent(BorderLayout.SOUTH, c3);

/**End of CoordinateLayout statements - second set**/
```

This time we do not install a coordinate layout on `demoForm`. Instead, we set a new border layout. We then go on to create three containers (`c1`, `c2`, and `c3`) and two new labels. The coordinate layout instance we created earlier is installed on `c2`, and the three original labels (`tLabel`, `imLabel`, and `bothLabel`) are added to `c2`, while the one new label is added to `c1` and the other to `c3`. Finally, the three containers are added to `demoForm`, as shown in the above code listing.

Now that we have added three containers to `demoForm`, the space for the container holding `tLabel`, `imLabel`, and `bothLabel` (`c2`) is smaller than that of the whole form. Therefore, `CoordinateLayout` proportionately reduces the relative spacing of these three labels leading to a different degree of overlap, as we can see in the next screenshot.



FlowLayout

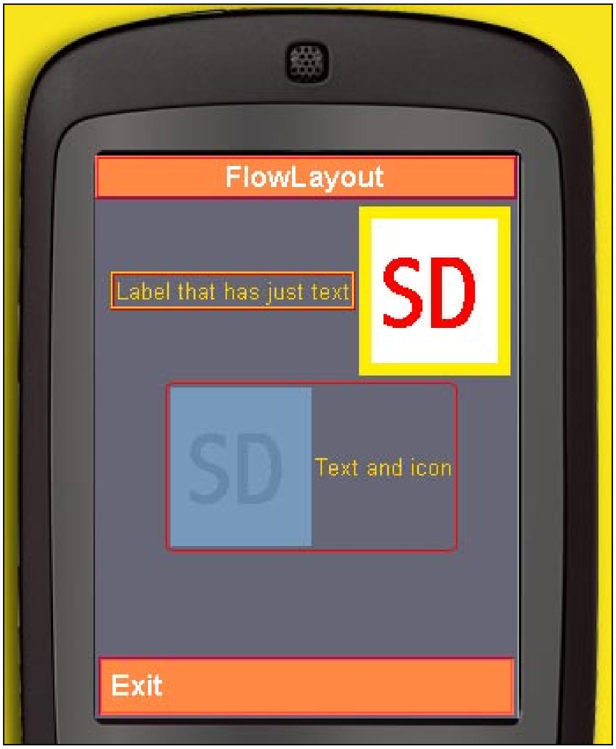
`FlowLayout` arranges elements sequentially along a row from left to right. If there is not enough space for a widget on the current row, a new row is started below the first one. By default, the positioning is left justified. However, it is possible to create a flow layout instance that performs right or center justified placement. The following code is for trying out the right and center justified configurations of `FlowLayout`. We shall leave the default version out of our example here, as we have seen many instances of it in the demo examples of the preceding chapters.

```
/*****Start of FlowLayout statements*****/

FlowLayout testLayout = new FlowLayout(Component.CENTER);
//FlowLayout testLayout = new FlowLayout(Component.RIGHT);

demoForm.setLayout(testLayout);
demoForm.setTitle("FlowLayout");
```

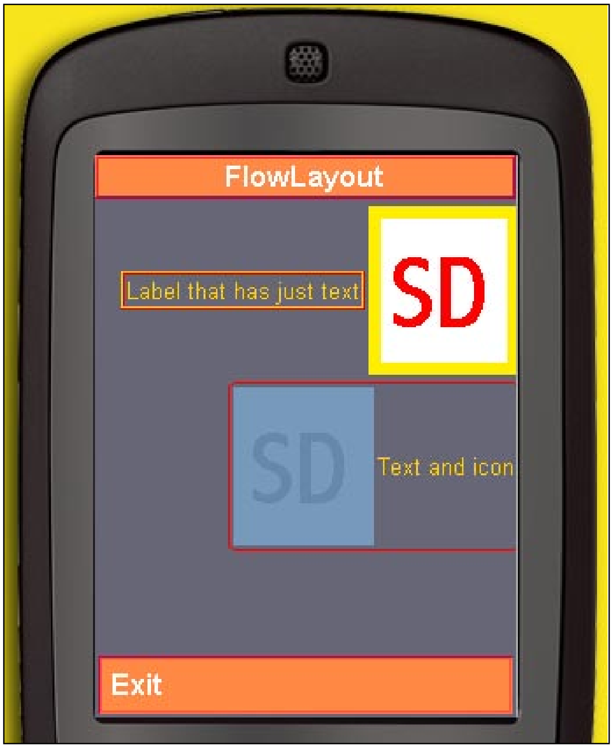
```
        demoForm.addComponent(tLabel);
        demoForm.addComponent(imLabel);
        demoForm.addComponent(bothLabel);
        /*****End of FlowLayout statements*****/
```

The constructor used here takes one parameter to specify the orientation, which can be either `LEFT`, `RIGHT` or `CENTER`. There is a default constructor too—`FlowLayout()`—which creates a left aligned instance.

The center justified version works as shown in the following screenshot:



You can now comment out the first line, and uncomment the second for a right justified display as shown in the following screenshot:



GridLayout

`GridLayout` divides the space available for widget placement into a number of equally sized cells arranged in the form of a grid. The number of rows and columns are specified through the constructor while instantiating `GridLayout`.

The `GridLayout` class has only one constructor, which is of the form `GridLayout(int rows, int columns)`. In the current version, the number of rows can be zero, in which case, the appropriate number is derived from the number of components in the container and the number of columns. As a matter of fact, the `layoutContainer` method of this class checks to see if the specified number of rows and columns are sufficient to accommodate all components in the container, and if required, sets the number of rows to the proper value. However, if the number of columns is zero, then a divide-by-zero occurs. In view of this, the next version will require that both the parameters (number of rows and number of columns) have to be greater than zero.

The code for testing `GridLayout` is as follows.

```
/*****Start of GridLayout statements*****/

GridLayout testLayout = new GridLayout(3, 1);
//GridLayout testLayout = new GridLayout(1, 3);
//GridLayout testLayout = new GridLayout(1, 2);

demoForm.setLayout(testLayout);
demoForm.setTitle("GridLayout");

demoForm.setScrollableX(true);

tLabel.setFocusable(true);
imLabel.setFocusable(true);
bothLabel.setFocusable(true);

demoForm.addComponent(tLabel);
demoForm.addComponent(imLabel);
demoForm.addComponent(bothLabel);

/*****End of GridLayout statements*****/
```

This code creates a grid layout with three rows and one column, which produces a vertical arrangement of the three labels. Note that a vertical scrollbar is automatically added, and it is not necessary to enable this behavior explicitly, as in the case of the horizontal scrollbar.

Changing the constructor to specify one row and three columns by uncommenting the second line leads to a single line horizontal arrangement. As we have enabled the horizontal scrollbar, all three labels are adequately sized. In all cases, you can see that the cells are equally sized, and the dimensions are based on the largest element.

When the third statement is used to create a **GridLayout** instance, three labels cannot be accommodated with the specified grid size. What happens then is that the number of rows is increased in order to hold all the labels.

`GridLayout` is extremely useful when elements of same size are to be displayed. One obvious use case is menu layout. The following screenshot shows the menu of the `LWUIT Demo` application demonstrating an effective use of `GridLayout`.

GroupLayout

`GroupLayout` arranges components by organizing them into groups. A group may contain components as well as other groups. This nesting ability provides an extremely flexible layout environment that can produce very attractive screen arrangements.

The two types of groups supported by `GroupLayout` are `Parallel Group` and `Sequential Group`. A parallel group arranges its child elements in the same space on top of each other, while a sequential group arranges child elements one after the other. A special case of parallel group functionality is the ability to align elements along their baseline.

`GroupLayout` determines the position of an element along the horizontal axis through a horizontal group. The horizontal group is responsible for sizing and positioning along the horizontal axis only. Similarly, a vertical group, responsible for sizing and positioning along the vertical axis only, is used to vertically position an element. A group layout instance must have a vertical and a horizontal group, and each widget must be included in both groups.

Let's see how this concept works in practice by considering an example. The section of code in the `DemoLayout` MIDlet that deals with `GroupLayout` is shown below.

```
/*****Start of GroupLayout statements*****/
 GroupLayout testLayout = new GroupLayout(demoForm);
//testLayout.linkSize(new Component [] {tLabel, imLabel,
                     bothLabel}, GroupLayout.HORIZONTAL);
 demoForm.setLayout(testLayout);
 demoForm.setTitle("GroupLayout");
 demoForm.setScrollableX(true);
 tLabel.setFocusable(true);
 imLabel.setFocusable(true);
 bothLabel.setFocusable(true);
 Label imLabel2;
 try
 {
   imLabel2 = new Label(Image.createImage("/sdsym4.png"));
}
 catch(java.io.IOException ioe)
 {
   imLabel2 = new Label("Image could not be loaded");
}
 imLabel2.getStyle().setBorder(Border.createLineBorder(7,
                                            0xfbe909));
 testLayout.setAutocreateGaps(true);
 testLayout.setAutocreateContainerGaps(true);
 GroupLayout.SequentialGroup hGroup = testLayout.
                       createSequentialGroup();
 GroupLayout.SequentialGroup vGroup = testLayout.
                      createSequentialGroup();
//GroupLayout.ParallelGroup hGroup = testLayout.
       createParallelGroup(GroupLayout.CENTER);
//GroupLayout.ParallelGroup vGroup = testLayout.
                      createParallelGroup();
 vGroup.add(testLayout.createSequentialGroup().add(bothLabel).
                      add(tLabel).add(imLabel));
 hGroup.add(testLayout.createParallelGroup().add(bothLabel).
                      add(tLabel).add(imLabel));
//hGroup.add(testLayout.createParallelGroup(GroupLayout.CENTER).
                      add(bothLabel).add(tLabel).add(imLabel));
//hGroup.add(testLayout.createSequentialGroup().add(bothLabel).
                      add(tLabel).add(imLabel));
//hGroup.add(testLayout.createParallelGroup(GroupLayout.CENTER).
  add(tLabel).add(testLayout.createSequentialGroup().
  add(imLabel).add(imLabel2)));
//vGroup.add(testLayout.createSequentialGroup().add(tLabel).
  add(testLayout.createParallelGroup().add(imLabel).
```

```
         add(imLabel2)));
        testLayout.setHorizontalGroup(hGroup);
        testLayout.setVerticalGroup(vGroup);
       /*****End of GroupLayout statements*****/
```

The group layout instance that we are going to work with is created by the only constructor of the class. The single parameter of the constructor specifies the container for which the layout manager is being instantiated—for our example it is `demoForm`. After setting the layout manager for `demoForm` and initializing the scroll and focus properties as for the other examples, a fourth label (`imLabel2`) is created which we shall use a little later.

The next two statements make sure that gaps are automatically created between adjacent components (gap between components), as well as between the host container and the widgets at the edges (`containerGap`).

```
        testLayout.setAutocreateGaps(true);
        testLayout.setAutocreateContainerGaps(true);
```

Next, we create a vertical group and a horizontal group. The classes for the two types of groups—`GroupLayout.SequentialGroup` and `GroupLayout.ParallelGroup`—do not have public constructors, and we have to use the `create` methods to get instances of these classes.

```
        GroupLayout.SequentialGroup hGroup =
            testLayout.createSequentialGroup();
        GroupLayout.SequentialGroup vGroup =
            testLayout.createSequentialGroup();
```

It is time now to look at how the screen for this layout is going to appear so that the setup of `hGroup` and `vGroup` can be properly studied.

As we look at this screenshot, let's imagine that the vertical dimension has been reduced to zero so that we can consider this arrangement from the perspective of the horizontal group. Notice that all the labels are aligned along the left edge with an offset determined by the automatic setting of `containerGap`. So, if we ignore the vertical dimension, we shall be left with three *overlapping* lines of different lengths, with all of them starting from the left edge. This is the result of adding the labels to a *parallel* group along the horizontal axis.

```
hGroup.add(testLayout.createParallelGroup().add(bothLabel).
                                 add(tLabel).add(imLabel));
```

Now if we consider the vertical dimension alone, we will have three vertical lines placed one after another along the left edge. This is because we have added the labels to a *sequential* group along the vertical axis.

```
vGroup.add(testLayout.createSequentialGroup().add(bothLabel).
                                 add(tLabel).add(imLabel));
```

The combined effect of the two statements above is to arrange the three labels with the same horizontal origin but with different vertical displacements as determined by their respective heights. In other words, the parallel horizontal group sets the `x` coordinates of the top-left corners of the labels to the same value (which is `0 + containerGap`). Similarly, the sequential vertical group sets the `y` coordinates of the top-left corners to the following values:

- value for `bothLabel` = y1 = 0 + `containerGap`
- value for `tLabel` = y2 = y1 + height of `bothLabel` + gap between components
- value for `imLabel` = y3 = y2 + height of `tLabel` + gap between components

Finally, the two groups `hGroup` and `vGroup` are added to `testLayout`.

```
testLayout.setHorizontalGroup(hGroup);
testLayout.setVerticalGroup(vGroup);
```

`GroupLayout` allows us to force all components to have the same size along the horizontal axis or the vertical axis or along both axes. The following code will ensure that all three labels have the same width.

```
GroupLayout testLayout = new GroupLayout(demoForm);
 testLayout.linkSize(new Component [] {tLabel, imLabel,
     bothLabel}, GroupLayout.HORIZONTAL);
.
.
.
 vGroup.add(testLayout.createSequentialGroup().add(bothLabel).
     add(tLabel).add(imLabel));
 hGroup.add(testLayout.createParallelGroup().add(bothLabel).
    add(tLabel).add(imLabel));
```

The method used for making all the labels have the same width is `linkSize(Component [] components, int axis)`. In this case, the value of the axis is `GroupLayout.HORIZONTAL`. As the following screenshot shows, all three labels have equal width. If the value of the axis had been `GroupLayout.VERTICAL`, then the heights would have become equal, and a value of `GroupLayout.HORIZONTAL|GroupLayout.VERTICAL` would have made the labels equal in both width and height.

There are a couple of things to be noted in the code for testing `GroupLayout`:

- It is not necessary to add the labels to the form. This is taken care of by the `add` methods.
- The `add` methods can be chained.

The left alignment of the labels in the example above is the default arrangement. We can opt for other alignments by using the appropriate method for creating a group, as in the statement shown below:

```
hGroup.add(testLayout.createParallelGroup(GroupLayout.CENTER).
                    add(bothLabel).add(tLabel).add(imLabel));
```

Here we create a parallel group with a `CENTER` alignment. Using this statement creates the following arrangement:

If we use a sequential horizontal group instead of a parallel one, then we can expect the labels to be placed one after the other along the horizontal axis. The code for this looks like:

```
hGroup.add(testLayout.createSequentialGroup().add(bothLabel).
                                        add(tLabel).add(imLabel));
```

The screen will now appear with the labels no longer overlapping along the horizontal dimension, as shown in the following screenshot:

We had touched upon nested groups earlier. We shall now see how this works. The screenshot that follows shows a screen layout that requires both the vertical and the horizontal groups to have nested structures. The new label (`imLabel2`) is used here so that the effect of the `CENTER` alignment becomes clear.

Here the labels in the second row share the same vertical space but are displaced horizontally. However, the label in the first row has a different position in the vertical direction. Obviously we cannot manage with a single vertical group, as we want a combination of parallel and sequential behavior. What we need here is a parallel group containing the labels in the second row and a sequential group to hold the parallel group and the label in the first row.

Similarly, as a group, the two labels in the second row have a `CENTER` alignment with respect to the label in the first row. So, for the horizontal group as well, we need a sequential group for the two labels in the second row first, and then a parallel group with a `CENTER` alignment to hold the sequential group and the label in the first row. The code that does all this is shown below:

```
hGroup.add(testLayout.createParallelGroup(GroupLayout.CENTER).
 add(tLabel).add(testLayout.createSequentialGroup().add(imLabel).
 add(imLabel2)));

vGroup.add(testLayout.createSequentialGroup().add(tLabel).
 add(testLayout.createParallelGroup().add(imLabel).
 add(imLabel2)));
```

Although we have used only the `CENTER` alignment in our example, `GroupLayout` offers other options as well, which are listed in the table below:

Alignment	Description
`BASELINE`	Arranges components so that their bottom edges are aligned.
`LEADING`	Arranges components so that their left edges are aligned.
`TRAILING`	Arranges components so that their right edges are aligned.

The `GroupLayout` class has three nested classes for the implementation of groups. We shall now study these classes.

GroupLayout.Group

This is an abstract class that makes it possible for the other two nested classes (`GroupLayout.ParallelGroup` and `GroupLayout.SequentialGroup`) to have commonality. This is an abstract class and cannot be instantiated. We must work with its two subclasses, that is, `GroupLayout.ParallelGroup` and `GroupLayout.SequentialGroup`. This class does not have any methods of its own, and all functionality is implemented through its subclasses.

GroupLayout.ParallelGroup

This class, as we know, lays out elements on top of each other. There are no `public` constructors for this class, and we have to use one of the `create` methods of the `GroupLayout` class. All the methods are for adding elements to an instance of this class. The following table gives an overview of the methods:

Method	Parameters	Description
`GroupLayout.ParallelGroup add(Component component)`	`component` — component to be added.	Adds the specified component to the group. Returns this group.
`GroupLayout.ParallelGroup add(Component component, int min, int pref, int max)`	`component` — component to be added. `min` — minimum size of the component. `pref` — preferred size of the component. `max` — maximum size of the component.	Adds the specified component to the group. The other three parameters refer to dimensions along the axis of the group. They can be specified as absolute values. `DEFAULT_SIZE` or `PREFERRED_SIZE` are also acceptable. Returns this group.

Method	Parameters	**Description**
`GroupLayout.ParallelGroup add(GroupLayout.Group group)`	`group`—group to be added.	Adds the specified group to the group instance. Returns this group.
`GroupLayout.ParallelGroup add(int pref)`	`pref`—size of gap to be added.	Adds a gap of a given size. Returns this group.
`GroupLayout.ParallelGroup add(int alignment, Component component)`	`alignment`—alignment to be used for laying out the component. `component`—component to be added.	Adds the specified component to the group using the given alignment. Returns this group.
`GroupLayout.ParallelGroup add(int alignment, Component component, int min, int pref, int max)`	`alignment`—alignment to be used for laying out the component. `component`—component to be added. `min`—minimum size of the component. `pref`—preferred size of the component. `max`—maximum size of the component.	Adds the specified component to the group using the given alignment. The other three parameters refer to dimensions along the axis of the group. They can be specified as absolute values. `DEFAULT_SIZE` or `PREFERRED_SIZE` are also acceptable. Returns this group.
`GroupLayout.ParallelGroup add(int alignment, GroupLayout.Group group)`	`alignment`—alignment to be used for laying out the component. `group`—group to be added.	Adds the specified group to the group instance using the given alignment. Returns this group.
`add(int min, int pref, int max)`	`min`—minimum size of the gap. `pref`—preferred size of the gap. `max`—maximum size of the gap.	Adds a gap of a given size. The three parameters refer to dimensions along the axis of the group. They can be specified as absolute values. `DEFAULT_SIZE` or `PREFERRED_SIZE` are also acceptable. Returns this group.

The size parameters can be used to indicate what kind of sizing the developer prefers. For example, the parameter *max* indicates a size allocation subject to availability of space. The *min* parameter indicates the minimum size regardless of space availability. If required, a scrollbar can be added to accommodate all the components. The problem with specifying these parameters is that the resulting arrangement may not have a device independent appearance. Generally speaking, it is better not to use these parameters without compelling reasons.

A *gap* related parameter adds a fixed spacing (or padding) along the direction of the group. Again, adding a gap may not give the same look to a screen layout on all devices.

The specified alignment of a parameter can override the group alignment. Consider the following modification in the statement for creating a horizontal group with a default alignment.

```
hGroup.add(testLayout.createParallelGroup().add(bothLabel).
              add(GroupLayout.TRAILING, tLabel).add(imLabel));
```

This aligns `tLabel` along the right edge instead of the default alignment along the left edge.

GroupLayout.SequentialGroup

This class arranges its child elements one after the other. Like `GroupLayout.ParallelGroup`, this class too does not have any `public` constructor and can be instantiated only through one of the `create` methods of its enclosing class.

The methods of this class are designed to add elements to the group instance. Since the elements here do not share the same space, alignment is not a relevant factor. So we find all the methods of `GroupLayout.ParallelGroup` in this class too, except the ones that use alignment as a parameter. On the other hand, the gap between adjacent components, as well as that between a container edge and the first or last component has a greater significance in this class and can be specified. Accordingly, we see a host of methods to add gaps.

Method	**Parameters**	**Description**

`GroupLayout.`
`SequentialGroup`
`addContainerGap()`
		Adds a `DEFAULT_SIZE` gap between the edge of a container and the following or preceding component.
		Returns this sequential group.

`GroupLayout.`
`SequentialGroup`
`addContainerGap`
`(int pref, int max)`
	`pref`—preferred size of gap. Must be `DEFAULT_SIZE` or a value greater than zero.
	`max`—maximum size of gap. Must be `DEFAULT_SIZE` or `PREFERRED_SIZE` or a value greater than zero.
		Adds a `DEFAULT_SIZE` gap between the edge of a container and the following or preceding component.
		The value of `pref` cannot be greater than that of `max`.
		Returns this sequential group.

`GroupLayout.`
`SequentialGroup`
`addPreferredGap`
`(Component comp1,`
`Component comp2,`
`int type)`
	`comp1`—first component.
	`comp2`—second component.
	`type`—type of gap. Must be one of the constants defined by `LayoutStyle`.
		Adds a preferred gap between two components.
		Returns this sequential group.

`GroupLayout.`
`SequentialGroup`
`addPreferredGap`
`(Component comp1,`
`Component`
`comp2, int type,`
`boolean canGrow)`
	`comp1`—first component
	`comp2`—second component.
	`type`—type of gap. Must be one of the constants defined by `LayoutStyle`.
	`canGrow`—if `true`, then the gap will grow depending on available space.
		Adds a preferred gap between two components.
		Returns this sequential group.

`GroupLayout.`
`SequentialGroup`
`addPreferredGap`
`(int type)`
	`type`—type of gap. Must be one of the constants defined by `LayoutStyle`.	Adds an element representing the preferred gap between the nearest components, that is, during layout, the neighboring components are found, and the `min`, `pref` and `max` of this element is set based on the preferred gap between the components. If no neighboring components are found, then the `min`, `pref` and `max` are set to 0.
		Returns this sequential group.

Method	Parameters	Description
`GroupLayout.SequentialGroup addPreferredGap (int type, int pref, int max)`	`type`—type of gap. Must be one of the constants defined by `LayoutStyle`. `pref`—preferred size of a gap. Must be `DEFAULT_SIZE` or a value greater than zero. `max`—maximum size of a gap. Must be `DEFAULT_SIZE` or `PREFERRED_SIZE` or a value greater than zero.	Adds an element representing the preferred gap between the nearest components. This means that during layout, the neighboring components are found, and the `min` of this element is set based on the preferred gap between the components. If no neighboring components are found, then the `min` is set to `0`. This method allows you to specify the preferred and maximum size by way of the `pref` and `max` arguments. These can either be a value >= `0`, in which case the preferred or max is the max of the argument and the preferred gap, or `DEFAULT_VALUE`, in which case the value is the same as the preferred gap. Returns this sequential group.

There are also three `add` methods with an option to use the specified element for determining the baseline for the group instance.

Method	Parameters	Description
`GroupLayout.SequentialGroup add(boolean useAsBaseline, Component component)`	`useAsBaseline`—if `true`, then the specified component should be used to calculate the baseline of this group. `component`—the component to be added	Adds the specified component. Returns this group.

Method	Parameters	Description
`GroupLayout.SequentialGroup add(boolean useAsBaseline, Component component, int min, int pref, int max)`	`useAsBaseline`—if `true`, then the specified component should be used to calculate the baseline of this group. `component`—the component to be added. `min`—minimum size of the component to be added. `pref`—preferred size of the component to be added. `max`—maximum size of the component to be added.	Adds the specified component. The other three parameters refer to dimensions along the axis of the group. They can be specified as absolute values. `DEFAULT_SIZE` or `PREFERRED_SIZE` are also acceptable. Returns this group.
`GroupLayout.SequentialGroup add(boolean useAsBaseline, GroupLayout.Group group)`	`useAsBaseline`—if `true`, then the specified group should be used to calculate the baseline of this group. `group`—the group to be added.	Adds the specified group. Returns this group.

Summary

The layout classes provide easy-to-use approaches to component layouts that are powerful enough to come up with polished device independent screen designs. One has to take note though of the special capabilities of each layout class, and use them accordingly. For instance, `BorderLayout` is useful in applications where stretching a widget to occupy all available space along a particular axis will not cause distortions. Similarly, `GridLayout` can be a very attractive option when equally sized elements need to be arranged in the form of a matrix as in a menu.

Of all the layout classes currently available, `GroupLayout` is the most accomplished. It was originally developed to support visual design tools, and as a result, has the capability to support arbitrary arrangements. Fortunately, it is quite easy to use in manually generated code too.

Complex screen patterns can be created by using a hierarchical structure of containers, each with an appropriate layout. We can set up groups of components this way, which are laid out to reflect their logical or functional relationship with each other, and each group can use a layout manager best suited for the arrangement that we have envisaged.

Similar results can also be achieved with `GroupLayout` by using groups as elements within other groups. An additional strength of `GroupLayout` lies in its ability to use customized spacing for indentation and for functional or logical grouping through `LayoutStyle`.

Finally, LWUIT allows us to write our own layout managers. All we have to do is make sure that our custom layout class extends `Layout` and implements the abstract methods.

8
Creating a Custom Component

Sometimes we feel the need for a special application-oriented component that is not available in the LWUIT library. On such occasions, a custom component has to be created, and in this chapter, we are going to see how to do that.

At a very fundamental level, it would seem that the only thing one needs to do for building a made-to-order component is write a class that extends `Component`. However, this essential step is not enough by itself, except in trivial cases. For practical usage, explicit action is required to implement one or more of the characteristics that make the LWUIT components so flexible and versatile. Some of these characteristics are:

- The ability to be styled
- Support for responding to customer inputs like a *keypress*
- Mechanisms for informing other objects that a specified incident has taken place
- Support for plugging in different visual presentations
- Provision for working out preferred dimensions

In the following demo application, we shall build up a component that tells the current time not through the usual analog or digital presentation, but through a text string.

The making of a component

Our new component has two operational modes: real-time display and elapsed-time display. The default mode is real-time display, which displays the time of day. This can be seen in the following screenshot:

The other mode displays the elapsed time from the instant the timer is started. As shown in the following screenshot, an icon (**e**) is used to indicate that the component is operating in the elapsed-time mode.

`TimeTeller` is the underlying class here that generates the time information to be displayed but does not handle the implementation of the display. It also generates an alarm but does not explicitly take any further action.

In this example, the `TimeTeller` class works with the following interfaces and class:

- `public interface AlarmHandler`—defines the functionality of the class for handling alarms generated by TimeTeller.
- `public interface Viewer`—defines the functionality of the class for displaying the time data from TimeTeller.

- `public class TimeViewer`—a concrete class that extends `Container` and implements `Viewer` to display the time data in this example.

`AlarmHandler` has just one method, and the interface definition is:

```
public interface AlarmHandler
{
   void alarmHandled(TimeTeller tt, int hrValue, int minValue,
                                        int dayNightValue);
}
```

The `alarmHandled` method allows the implementing class to take appropriate action when the alarm goes off in `TimeTeller`.

The `Viewer` interface has methods for performing various display-related activities for `TimeTeller`. The interface definition is as follows:

```
public interface Viewer
{
   //displays the time in AM/PM or 24-hour format
   void showTime(int hour, int min, int dayNight);
   //used in elapsed time mode to display time
   void showCount(int hrCount, int minCount);
   //enables alarm mode in Viewer
   void setAlarmOn(boolean value);
   //returns true if alarm is enabled
   boolean isAlarmOn();
   //enables or disables the flasher
   //which can be used
   //to control any periodic element in the display
   void setFlasher(boolean value);
   //returns true if flasher is enabled and false otherwise
   boolean getFlasher();
   //sets styles for various elements of display in an
   //implementation dependent manner
   void setStyles(Style[] newStyles);
   //returns styles for various elements of display in an
   //implementation dependent manner
   Style[] getStyles();
   //sets elapsed time display mode if value is true
   //otherwise sets realtime display mode
   void setElapsedTimeMode(boolean value);
   //returns true if elapsed time display mode
   //has been set and false otherwise
   boolean isElapsedTimeMode();
}
```

In this example, the time data generated by `TimeTeller` is displayed as a text string. However, the display can be totally customized through the use of the `Viewer` interface, thus providing a pluggable look for `TimeTeller`. For instance, consider the `showTime` method. The `TimeViewer` class implements this method to display time in a 12-hour format. It is also possible to use the same method signature for a 24-hour format. We can implement the method so that a value of `2` for the `dayNight` argument would indicate that the display is meant for a 24-hour format, while a value of `1` (for PM) or `0` (for AM) would specify a 12-hour format. Similarly, the `flasher` variable can be used by an implementing class for controlling the movement of a *seconds* hand or the blinking of the *seconds* digits. All these methods enable us to tailor the implementing class in such a way that we can plug in the kind of display we want including the common analog or digital varieties. While an `AlarmHandler` is required only when an alarm needs to be acted on, a `Viewer` is an essential part of the package.

The TimeViewer class

We shall start our discussion on `TimeTeller` with a look at the `TimeViewer` class. The `TimeViewer` class is really a container with two labels—the `titleLabel`, which displays the text "**The time is**:" along with the mode dependent icon when applicable, and the `timeLabel` for displaying time information. The colon in the given text blinks to show that the clock is ticking. There is no icon for real-time mode.

The variables declared for `TimeViewer` are as follows:

```
private Label titleLabel;
private Label timeLabel;
private boolean flasher = true;
private final String titleString = "The time is:";
private final String titleBlinkString = "The time is";
private int hrValue;
private int minValue;
private int dayNightValue;
private int hrCount;
private int minCount;
private boolean alarmOn;
private boolean elapsedTimeMode;
private Image alarmIcon;
private Image timerIcon;
private boolean largeScreen = true;
//fonts for timeLabel
private final Font tmFont1 = Font.createSystemFont(Font.
     FACE_PROPORTIONAL,Font.STYLE_BOLD,Font.SIZE_LARGE);
private final Font tmFont2 = Font.createSystemFont(Font.
    FACE_PROPORTIONAL,Font.STYLE_BOLD,Font.SIZE_MEDIUM);
//padding values for timeLabel
private final int pad = 3;
```

The constructor of `TimeViewer` first creates a container with border layout:

```
super(new BorderLayout());
```

It then creates and initializes the two labels:

```
titleLabel = new Label(titleString);
timeLabel = new Label("");
timeLabel.setAlignment(Label.CENTER);
Style tmStyle = timeLabel.getStyle();
tmStyle.setFont(tmFont1);
tmStyle.setPadding(pad, pad, pad, pad);
int tmWidth = tmFont1.stringWidth("WWWWWWWWWWWW");
int tmHeight = tmFont1.getHeight();
tmWidth = tmWidth + 2 * pad;
tmHeight = tmHeight + 2 * pad;
timeLabel.setPreferredSize(new Dimension(tmWidth, tmHeight));
if(timeLabel.getPreferredW() > Display.getInstance().
                                    getDisplayWidth())
{
   tmStyle.setFont(tmFont2);
   tmWidth = tmFont2.stringWidth("WWWWWWWWWWWW");
   tmHeight = tmFont2.getHeight();
   tmWidth = tmWidth + 2 * pad;
   tmHeight = tmHeight + 2 * pad;
   timeLabel.setPreferredSize(new Dimension(tmWidth, tmHeight));
   largeScreen = false;
}
```

The text for `timeLabel` will keep changing, so this label is created without a text. However, this will create a problem for preferred size calculations, as the `calcPreferredSize` method of `timeLabel` is unaware of the size of the text to be displayed. The `List` class addresses this problem through the `setRenderingPrototype` method. As the `Label` class does not have such a method, it is necessary for us to provide the required sizing support. In order to do this, we first set up two final font versions and a final value for padding in the list of declared variables.

```
//fonts for timeLabel
private final Font tmFont1 = Font.createSystemFont(Font.FACE_
               PROPORTIONAL,Font.STYLE_BOLD,Font.SIZE_LARGE);
private final Font tmFont2 = Font.createSystemFont(Font.FACE_
             PROPORTIONAL,Font.STYLE_BOLD,Font.SIZE_MEDIUM);
//padding values for timeLabel
private final int pad = 3;
```

First, tmFont1 is incorporated into the **style** object for timeLabel. We then calculate the width of the label based on that of a prototype text (12 Ws) and the declared padding value. The height of timeLabel is calculated similarly from that of the font and the padding value. At this time, we check to see whether the width of timeLabel is greater than the display width, and if so, then use tmFont2 to produce a narrower timeLabel. The result of this adjustment is seen in the next two screenshots. Without the size check, the complete time data is not displayed on a relatively small screen.

When the label width is set as per the display width, the full text of timeLabel can be displayed.

The reason for doing all this is to ensure that we always have the same size for the label of a given screen. The problem is that a user can still change the font and padding in the timeLabel style, and this may make the label look disproportionate. In order to prevent this, we override the paint method where we set the proper font and the proper padding value before TimeTeller is repainted.

```
public void paint(Graphics g)
{
    if(largeScreen)
    {
        timeLabel.getStyle().setFont(tmFont1);
    }
    else
```

```
        {
            timeLabel.getStyle().setFont(tmFont2);
        }
        timeLabel.getStyle().setPadding(pad, pad, pad, pad);
        super.paint(g);
    }
```

Back in the constructor, we create the images for indicating alarm and elapsed time modes. Finally, the two labels are added to the container, and some style attributes are set for it.

```
        try
        {
            alarmIcon = Image.createImage("/alarm.png");
        }
        catch(java.io.IOException ioe)
        {
        }
        try
        {
            timerIcon = Image.createImage("/timer.png");
        }
        catch(java.io.IOException ioe)
        {
        }
        addComponent(BorderLayout.NORTH, titleLabel);
        addComponent(BorderLayout.CENTER, timeLabel);
        getStyle().setBorder(Border.createLineBorder(2, 0xfea429));
        getStyle().setBgColor(0x555555);
        getStyle().setBgTransparency((byte)255);
        getStyle().setPadding(pad, pad, pad, pad);
```

The two methods of major importance are `public void showTime(int hour, int min, int dayNight)` and `public void showCount(int hrCount, int minCount)`. The first method is meant for displaying the time of the day and has been customized for this example to handle the 12-hour format. It just converts the integers into strings, while taking care of singular and plural values, as well as uses the terms *noon* and *midnight* instead of 12 PM and 12 AM respectively.

```
    public void showTime(int hour, int min, int dayNight)
    {
        String singlePluralString = " minutes ";
        String dayNightString = " (AM) ";
        String hourString = String.valueOf(hour);
```

```
      String minString = String.valueOf(min);
      if(min <= 1)
      {
         singlePluralString = " minute ";
      }

      //0 means AM and 1 means PM
      if(dayNight == 1)
      {
         dayNightString = " (PM) ";
      }

      if(hour == 0)
      {
         if(dayNight == 0)
         {
            timeLabel.setText(minString + singlePluralString +
                                                "past midnight");
            return;
         }
         timeLabel.setText(minString + singlePluralString +
                                                "past noon");
         return;
      }
      timeLabel.setText(minString + singlePluralString +
                  "past " + hourString + dayNightString);
   }
```

The `showTime` method can also be configured to handle elapsed time display. However, the `showCount` method has been included in `TimeViewer` for convenience. This method is a stripped down version of `showTime`, as it does not have to bother about any AM/PM information.

```
   public void showCount(int hrCount, int minCount)
   {
      String singlePluralMinString = " minutes ";
      String singlePluralHrString = " hours ";

      String hourString = String.valueOf(hrCount);
      String minString = String.valueOf(minCount);

      if(minCount <= 1)
      {
         singlePluralMinString = " minute ";
      }

      if(hrCount <= 1)
      {
```

```
            singlePluralHrString = " hour ";
        }
        timeLabel.setText(hourString + singlePluralHrString +
                "and " + minString + singlePluralMinString);
    }
```

The rest of the methods are accessors for the variables that influence various display parameters. The following methods are for the alarm mode.

```
    public void setAlarmOn(boolean value)
    {
        alarmOn = value;
        if(alarmOn)
        {
            titleLabel.setIcon(alarmIcon);
        }
        else
        {
            titleLabel.setIcon(null);
        }
    }
    public boolean isAlarmOn()
    {
        return alarmOn;
    }
```

The first method modifies the value of `alarmOn` and accordingly sets or removes the icon for mode indication. The second just returns the value of `alarmOn`. The accessor methods for the `elapsedTime` also work in the same way.

```
    public void setElapsedTimeMode(boolean value)
    {
        elapsedTimeMode = value;
        if(elapsedTimeMode)
        {
            titleLabel.setIcon(timerIcon);
        }
        else
        {
            titleLabel.setIcon(null);
        }
    }
    public boolean isElapsedTimeMode()
    {
        return elapsedTimeMode;
    }
```

The `flasher` variable is intended for controlling the display of an element that periodically changes state. In this application, it is used to make the *colon* blink in the `titleLabel` text.

```
public void setFlasher(boolean value)
{
   //flasher = value;
   if(flasher != value)
   {
      flasher = value;
      if(flasher)
      {
         titleLabel.setText(titleString);
         return;
      }
      titleLabel.setText(titleBlinkString);
   }
}
public boolean getFlasher()
{
   return flasher;
}
```

Setting style attributes for a composite component involves manipulation of styles for all the constituent components. Therefore, the accessor methods for style have to be flexible enough to handle different numbers and types of `style` objects, depending on the composition of the display. This goal has been achieved by using a style array, which would have the requisite number of styles as the argument for `setStyles` method. The supporting private methods are then used to link the elements of the style array with the corresponding style objects.

```
public void setStyles(Style[] newStyles)
{
   //either or both styles may be null
   if(newStyles != null && newStyles.length == 2)
   {
      if(newStyles[0] != null)
      {
         setTitleStyle(newStyles[0]);
      }
      if(newStyles[1] != null)
      {
         setTimeStyle(newStyles[1]);
      }
   }
```

```
        else
        {
           //throw exception
           throw new IllegalArgumentException("Style array must not
                         be null and two styles must be specified");
        }
     }
     public Style[] getStyles()
     {
        Style[] viewerStyles = {getTitleStyle(), getTimeStyle()};
        return viewerStyles;
     }
     private void setTimeStyle(Style newStyle)
     {
        timeLabel.setStyle(newStyle);
     }
     private void setTitleStyle(Style newStyle)
     {
        titleLabel.setStyle(newStyle);
     }
     private Style getTimeStyle()
     {
        return timeLabel.getStyle();
     }
     private Style getTitleStyle()
     {
        return titleLabel.getStyle();
     }
```

The TimeTeller class

Now that we know how the `Viewer` interface allows us to use different types of display for `TimeTeller` and how the `TimeViewer` implements a specific display, we can proceed to the class that generates the basic information to be displayed—the `TimeTeller` class.

The `TimeTeller` class has two constructors. The first one takes no arguments and looks like this:

```
public TimeTeller()
{
   this(new TimeViewer());
}
```

The second constructor of `TimeTeller`—`public TimeTeller(Viewer viewer)`—takes a `viewer` object as an argument and can be used to install a `Viewer` other than the one provided here. This constructor does all the initialization that is required. First comes the obvious task of installing the `Viewer`. This is followed by the setting of the starting times for the blinking and garbage collection cycles, which we shall discuss a little later in our code analysis.

Even though LWUIT ensures a platform-neutral look for `TimeTeller`, there is a non-visual issue that has to be taken care of to make this component work properly across diverse devices. This involves handling the different ways in which the `Calendar` class returns time values. The same code for `TimeTeller` can show different times, depending on which device or emulator it is running on. The following list shows what time value was displayed on three different systems, although the time zone setting (Indian Standard Time—GMT + 5:30) was the same:

- On the Sprint WTK 3.3.2, the time is shown correctly
- Sun Java(TM) Wireless Toolkit 2.5 for CLDC displays GMT
- One of my phones shows time with an offset of GMT + 5:00, even though the clock setting is GMT + 5:30

This problem is taken care of in the constructor by calling the `getRawOffset` method of the `java.util.TimeZone` class. This method returns the offset (in milliseconds) with respect to the GMT that is used to return time values on the given device. This is compared with the desired offset, which is set as a final value in the variable declaration list, and the difference is used for getting the correct values of time.

```
private final int desiredOffset = 19800000;//IST
//private final int desiredOffset = -25200000;//PDT
//private final int desiredOffset = -28800000;//PST
//private final int desiredOffset = 0;//GMT
.
.
public TimeTeller(Viewer viewer)
{
    .
    .
    .
    int offset = TimeZone.getDefault().getRawOffset();
    if(offset != desiredOffset)
    {
        //calculate correction factors
        localOffset = desiredOffset - offset;
        hrOffset = localOffset/3600000;
        minOffset = (localOffset/60000)%60;
```

```
    }
    calendar = Calendar.getInstance();
    hrValue = calendar.get(Calendar.HOUR);
    minValue = calendar.get(Calendar.MINUTE);
    dayNightValue = calendar.get(Calendar.AM_PM);
    if(localOffset != 0)
    {
       if(localOffset > 0)
       {
          hrValue += hrOffset;
          minValue += minOffset;
          if(minValue >= 60)
          {
             minValue -= 60;
             hrValue++;
          }
          if(hrValue >= 12)
          {
             hrValue -= 12;
             dayNightValue = (dayNightValue + 1) % 2;
          }
       }
       else
       {
          hrValue += hrOffset;
          minValue += minOffset;
          if(minValue < 0)
          {
             minValue = 60 + minValue;
             hrValue--;
          }
          if(hrValue < 0)
          {
             hrValue = 24 + hrValue;
             hrValue = hrValue % 12;
             dayNightValue = (dayNightValue + 1) % 2;
          }
       }
    }
```

The sample code includes offsets corresponding to four time zones. While using any of them, just make sure that the other three are commented out, or else the code will not compile.

Once the corrected time values are determined, then the `updateView` method is called to initialize the viewer display. Finally, the `Thread` that acts as the time base is created and started.

```
    updateView();

    Thread t = new Thread(this);
    t.start();
```

The `updateView` method calls either the `showTime` or the `showCount` method of the installed viewer, depending on the mode setting.

```
public void updateView()
{
   if(mode == TimeTeller.REALTIME)
   {
      viewer.showTime(hrValue, minValue, dayNightValue);
   }
   else
   {
      viewer.showCount(hrCount, minCount);
   }
}
```

`TimeTeller` has methods to get and to set styles for the two labels. These are provided as convenience methods for working with the time viewer, which is the default viewer. When used with other viewers, these accessors may not be usable. Accordingly, `TimeTeller` has two empty methods that can be overridden in a subclass to provide necessary styling support. All these methods are as follows:

```
//sets style for timeLabel in TimeViewer
public void setTimeStyle(Style newStyle)
{
   Style[] styles = {null, newStyle};
   viewer.setStyles(styles);
}

//sets style for titleLabel in TimeViewer
public void setTitleStyle(Style newStyle)
{
   Style[] styles = {newStyle, null};
   viewer.setStyles(styles);
}
```

```
//gets style for timeLabel in TimeViewer
public Style getTimeStyle()
{
   Style[] styles = viewer.getStyles();
   return styles[1];
}
//gets style for titleLabel in TimeViewer
public Style getTitleStyle()
{
   Style[] styles = viewer.getStyles();
   return styles[0];
}
//empty method to be overridden for other types of Viewers
public void setStyles(Style[] styles)
{
}
/*empty method to be overridden for other types of Viewers
not to be uncommented unless body of method is inserted
public Style[] getStyles()
{
}*/
```

The default mode of `TimeTeller` is real time. So let us see how this mode works.

The Real time mode

In the real time mode, `TimeTeller` generates the time values to be displayed in its `run` method, which starts executing as soon the constructor is invoked and loops until `done` is set to `true`. This happens when the `Exit` command is executed. The thread sleeps for 100 milliseconds at the beginning of an iteration. When it wakes up, the current time is obtained.

```
try
{
   Thread.sleep(sleepTime);
}
catch(java.lang.InterruptedException ie)
{
}
long newTime = System.currentTimeMillis();
```

The signal for *blinking* is to be generated only when the clock is enabled, which happens in the real time mode and in the elapsed time mode if the `timerEnabled` variable is `true`. These conditions are checked for, and action is taken to switch the state of `blinkOn`, depending on its current state and how long it has been in the current state. The values of `blinkOnTime` and `blinkOffTime` determine how long `blinkOn` should remain in a particular state.

```
if(mode == TimeTeller.REALTIME || (mode == TimeTeller.
                        ELAPSEDTIME && timerEnabled))
{
   if(blinkOn && (newTime >= lastBlinkTime + blinkOnTime))
   {
   lastBlinkTime = newTime;
      setBlinkOn(false);
   }
   else
   {
      if(!blinkOn && (newTime >= lastBlinkTime +
                            blinkOffTime))
      {
         lastBlinkTime = newTime;
         setBlinkOn(true);
      }
   }
}
```

In real time mode, the current value of the minute is saved as `newMin`, and a correction is applied to take care of the offset. As the value of `minOffset` can be negative, we must check if `newTime` itself has become negative and take appropriate action.

```
Calendar cal = calendar.getInstance();
int newMin;
if(localOffset >= 0)
{
   newMin = (cal.get(Calendar.MINUTE) + minOffset) % 60;
}
else
{
   newMin = (cal.get(Calendar.MINUTE) + minOffset);

   if(newMin < 0)
   {
      newMin = 60 + newMin;
   }
}
```

If the value of `newMin` has changed since the last iteration, then the values of the variables to be displayed are updated. The value of minute is retrieved once again so that offset correction, if required, can be applied to all three variables.

```
if(newMin != minValue)
{
   minValue = cal.get(Calendar.MINUTE);
   hrValue = cal.get(Calendar.HOUR);
      dayNightValue = cal.get(Calendar.AM_PM);
   if(localOffset != 0)
   {
   .
   .
   .
}
```

The process of offset correction is the same as the one that was used within the constructor.

Finally, `blinkOn` is synchronized and the `showTime` method of the `Viewer` is called.

```
setBlinkOn(true);
viewer.showTime(hrValue, minValue, dayNightValue);
```

The duration for keeping `blinkOn` `true` and that for keeping it `false` can be set through the following methods:

```
public boolean setBlinkOnTime(int millis)
{
   if(millis >= 200)
   {
      blinkOnTime = millis;
      return true;
   }
   return false;
}
public boolean setBlinkOffTime(int millis)
{
   if(millis >= 200)
   {
      blinkOffTime = millis;
      return true;
   }
   return false;
}
```

The durations are set keeping in mind the timebase granularity.

Using the Alarm function

The `TimeTeller` class can also generate an alarm at a preset time in the real time mode if this function is enabled. The alarm is not handled explicitly by `TimeTeller`, but by an instance of `AlarmHandler`. The `addAlarmHandler` method adds a handler, and the `removeAlarmHandler` method removes it. Although, we have added only one handler in our example, it is possible to have multiple handlers.

In order to activate the alarm, we need to use the **Alarm On** command, as shown in the following screenshot:

Executing the **Alarm On** command calls the `actionPerformed` method of the MIDlet (`TimeTellerMIDlet`) for this example, which in turn, invokes the `changeAlarmMode` method of `TimeTeller` with `true` as the argument. As the alarm mode is to be activated, a dialog is shown to set the alarm values. Note that the `alarmOn` variable is not set to `true` at this time. This is done by the dialog if it successfully sets the time values for the alarm.

```
public boolean changeAlarmMode(boolean value)
{
   if(value)
   {
      showDialog(false);
   }
   else
   {
```

```
            setAlarmOn(false);
        }
        return isAlarmOn();
    }
```

The `showDialog` method takes a `boolean` argument. When the argument is `true`, the time fields and the radio buttons are initialized to the existing settings. This allows us to see when the alarm is to go off and to change the time if we want.

On the other hand, if this argument is `false`, then the fields for alarm time values are shown empty, and the **AM** radio button is put in the selected state so that the alarm time settings can be initialized.

The `showDialog` method, which displays the dialog is very similar to the `showDialog` methods we have already encountered in some of our earlier examples. However, there is one difference that has to be noted. Let us cast our minds back to Chapter 6 and recall *in situ* editing of text fields. The following screenshot shows a text field ready for editing:

We can see the icon for input mode on the text field, which can be stepped through by pressing the '#' key—an icon that is missing from the text fields in the dialog for setting or editing alarm time values. In our application, we only want numeric inputs, and there is no need for other modes to be used at all. Accordingly, we use the following code in `showDialog`:

```
String[] inputModeOrder = {"123"};
tf1.setInputModeOrder(inputModeOrder);
.
.
.
tf2.setInputModeOrder(inputModeOrder);
```

This code sets the numeric input mode as the only input mode for the text fields. Now that there is only one input mode, the icon is not shown during editing.

The **Cancel** command closes the dialog without doing anything. The **OK** command sets the alarm timings, and if successful, sets `alarmOn` to `true` by calling the `setAlarmOn` method. The `setAlarmOn` method calls the method of same name in `TimeViewer` to show the icon that indicates if the alarm has been activated.

The method used for setting alarm time values is `setAlarmValue`. This method checks to make sure that the values are in the acceptable range, and then saves them.

```
public void setAlarmValue(int hr, int min, int dayNight)
                        throws IllegalArgumentException
{
   if(mode == TimeTeller.REALTIME)
   {
      if(hr >= 1 && hr <= 12 && min >= 0 && min <= 59 && (
                        dayNight == 0 || dayNight == 1))
      {
         alarmHr = hr;
         //convert 1 to 12 range of hour values from the dialog
         //into 0 to 11 range
         if(alarmHr == 12)
         {
            alarmHr = 0;
            }
         alarmMin = min;
         alarmDayNight = dayNight;
         }
      else
      {
         //throw exception
         throw new IllegalArgumentException("hr must be within 1
                        and 12 -- min must be within 0 and 59");
      }
   }
}
```

There is also a companion method that returns the values for alarm setting. This method returns the existing values if the alarm has been activated. Otherwise, it will return `-1` for all the three settings.

```
public int[] getAlarmValue()
{
   int[] alVal = {alarmHr, alarmMin, alarmDayNight};
   int [] invAlVal = {-1, -1, -1};
   if(alarmOn)
   {
      return alVal;
   }
   else
   {
      return invAlVal;
   }
}
```

Referring back to the `run` method, we see that every time the minute value changes, a check is made to see if the alarm has been enabled and the time set for the alarm matches the current time. When a match occurs, the alarm is triggered.

```
if(alarmOn && hrValue == alarmHr && minValue ==
    alarmMin && dayNightValue == alarmDayNight)
{
//time to trigger alarm
callAlarmHandler();
/*Display.getInstance().callSerially(new Runnable()
{
   public void run()
      {
         callAlarmHandler();
      }
   });*/
   /*Display.getInstance().callSeriallyAndWait(
                                 new Runnable()
   {
      public void run()
      {
         callAlarmHandler();
      }
   });*/
}
```

In the previous code, the `callAlarmHandler` method has been invoked in three different ways. The first, shown uncommented, directly calls the method to take proper action. This is a straightforward approach that acts instantaneously. It is also possible for the alarm to hitch a ride on the EDT through the other two sets of statements that have been commented out.

Both these sets work similarly by queuing up the call for handling the alarm through the event propagation mechanism of LWUIT, which ensures delivery of all such events and calls in the same order in which they were placed in the queue. The only difference is that `callSerially` returns without waiting for the action in the alarm handler to be completed, while `callSeriallyAndWait` returns only after the action is completed. The point to note here is that the argument for these methods MUST be runnable and NOT a thread.

The issue to be considered here is the criteria for selecting one of the three approaches. If the call is isolated in the sense that its processing does not depend upon a sequential relationship with any other call or event, then the first option—the direct call to `AlarmHandler`—is quite acceptable. That is why, for our example, this is a good choice. Another reason for the direct call could be a need for immediate action.

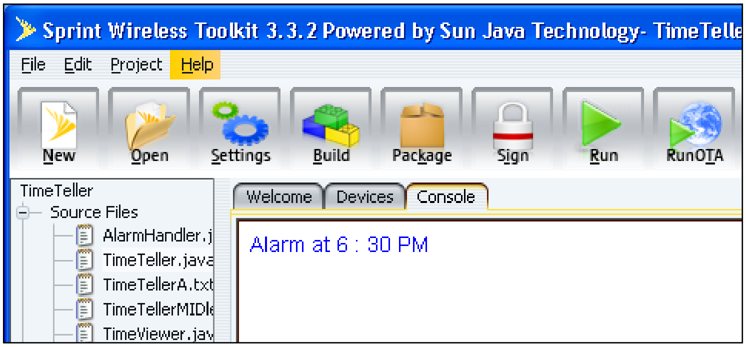
However, if multiple calls or events are involved and their processing has to be tailored as per the sequence in which they were generated, then we have to use one of the other two techniques. The question is 'which'? In situations where we can afford to have the calling thread blocked or where the subsequent action to be taken by the calling thread depends on the result of call processing, `callSeriallyAndWait` will be the logical choice. On the other hand, if it is not possible to risk blocking the calling thread, or the processing of the call has no bearing on what the calling thread does after generating the call, then `callSerially` needs to be used. Here, the `TimeTeller` thread is the time base, and we cannot accept the possibility that, depending on the nature of response to the call, the thread may get held up and so disrupt the timing action. Even if we decide to use one of the commented out versions, we have to select `callSerially`.

The discussion above offers only a general guideline. There will be occasions when the implications will not be so clear cut, and one would have to consider other approaches including, in extreme cases, restructuring the code.

The `callAlarmHandler` method calls the `alarmHandled` method of the alarm handler that has been added to the time teller instance, which is `TimeTellerMIDlet` in this case.

```
private void callAlarmHandler()
{
    if(handler != null)
    {
        handler.alarmHandled(this, hrValue, minValue, dayNightValue);
    }
}
```

The result of this chain of method invocations is that a message is printed on the console. The alarm is also disabled by calling the `setAlarmMode` method with a `false` value as an argument. This call ripples through to the time viewer, which turns off the alarm icon on the display.



The action to be taken when an alarm is triggered is entirely dependent on the alarm handler. The usual action would be to show an alert accompanied by a tone, the flashing of backlight, and possibly vibrator activation. Functions like snooze can also be implemented around this basic core.

The menu provides a command for turning the alarm off at any time.

The final action taken in real time mode is to call the garbage collector once a minute. This ensures proper memory utilization on some devices.

```
if(newTime >= lastGcTime +60000)
{
    lastGcTime = newTime;
    System.gc();
}
```

The ElapsedTime mode

The second operating mode for `TimeTeller` is the elapsed-time mode. In order to enter this mode, the **Timer** command has to be selected from the Menu.

The `actionPerformed` method of `TimeTellerMIDlet` calls the `setMode` method of `TimeTeller`, which sets proper mode, does the necessary housekeeping, and calls the `setElapsedTimeMode` method in `TimeViewer` to show the mode icon.

```
if(mode == TimeTeller.ELAPSEDTIME)
{
   //initialize time parameters, synchronize blinkOn
         and update display hrCount = minCount = 0;
   lastBlinkTime = lastUpdateTime =
       System.currentTimeMillis();
   setBlinkOn(true);
   updateView();

   //remove alarm icon from viewer if it was there
   viewer.setAlarmOn(false);

   //enable elapsed time mode and disable realtime mode
   viewer.setElapsedTimeMode(true);
}
```

The elapsed time mode can be used through three commands: **Start**, **Stop**, and **Reset**. These commands are shown in the following screenshot:

When the mode is switched to elapsed time, the timer remains disabled. This is shown by the colon on `titleLabel`, which stops blinking. The **Start** command has to be executed to commence timing. This calls the `enableTimer` method in `TimeTeller`.

```
public void enableTimer(boolean value)
{
    if(mode == TimeTeller.ELAPSEDTIME && timerEnabled != value)
    {
        timerEnabled = value;
        if(timerEnabled)
        {
            lastBlinkTime = lastUpdateTime =
                System.currentTimeMillis();
        }
        else
        {
            //timer is disabled take appropriate action if required
        }
        setBlinkOn(true);
        updateView();
    }
}
```

The **Stop** command too calls into this method with a **false** value for the argument. Stopping the timer does not reset the count to zero, and when restarted, the count gets accumulated. In order to bring the counter to zero, the **Reset** command has to be used, which results in the `resetTimer` method being called.

```
public void resetTimer()
{
   if(mode == TimeTeller.ELAPSEDTIME)
   {
      hrCount = minCount = 0;
      setBlinkOn(true);
      if(timerEnabled)
      {
         lastBlinkTime = lastUpdateTime =
               System.currentTimeMillis();
      }
      updateView();
   }
}
```

If the timer is reset while it is running, then the starting instance is reinitialized (but not in a *thread safe* manner), and elapsed time is measured from the instant of resetting.

A component often needs to sense user inputs directly. There are methods in the `Component` class to sense key and pointer (if supported by the device) events. In keeping with the standard LWUIT practice, `TimeTeller` uses the `keyReleased` method to start and stop the timer from the keyboard in elapsed time mode. The `'*'` key starts the timer, and the `'#'` key stops it.

```
public void keyReleased(int keyCode)
{
   if(keyCode == '#' && mode == TimeTeller.ELAPSEDTIME)
   {
      //stop the timer
      enableTimer(false);
   }
   if(keyCode == '*' && mode == TimeTeller.ELAPSEDTIME)
   {
      //start the timer
      enableTimer(true);
   }
}
```

Elapsed time counting is also done in the `run` method of `TimeTeller`. The minute count is incremented after every 60,000 milliseconds, and the minute and hour values are adjusted as required. As this technique of generating time base does not give very accurate results, we try to minimize the cumulative error by applying a correction factor to `lastUpdateTime`. Here too `blinkOn` is synchronized, and the new count is displayed through `TimeViewer`.

```
if(mode == TimeTeller.ELAPSEDTIME && timerEnabled)
{
   //elapsed time mode and timer is enabled
   //update time every minute
   if(newTime >= lastUpdateTime + 60000)
   {
      //minimize cumulative error
      lastUpdateTime = newTime - ((newTime -
                   lastUpdateTime) - 60000);

      minCount++;

      if(minCount == 60)
      {
         minCount = 0;
         hrCount++;
      }

      setBlinkOn(true);

      viewer.showCount(hrCount, minCount);
   }
}
```

There is one method that a component should override. This is the `getUIID` method, which allows us to define a style that is specific to this component. In several examples, we have used `UIID` to set style to an entire genre of components. The following statement, for instance, sets a style (`labelStyle`) to all labels:

```
UIManager.getInstance().setComponentStyle("SoftButton",
                                          labelStyle);
```

When we discuss resource creation in Chapter 9, we shall see how to use `UIID` with **LWUIT Designer** (previously known as **Resource Editor**). For our component, this method is:

```
protected String getUIID()
{
   return "TimeTeller";
}
```

The TimeTellerMIDlet

The basic structure of the MIDlet remains the same as in our other examples. The only new feature here is that the MIDlet implements `AlarmHandler`, and so it has an `alarmHandled` method. It registers itself as the alarm handler to be able to listen to alarms raised by `tt`.

```
tt.addAlarmHandler(this);
```

The `alarmHandled` method makes sure that the alarm origin was `tt` and then prints a message with the alarm time on the console. The `if(!tt.setAlarmMode(false))` statement turns the alarm mode off and then the commands are updated.

```
public void alarmHandled(TimeTeller tt, int hrValue,
                    int minValue, int dayNightValue)
{
   if(tt.equals(this.tt))
   {
      String amPmString = " AM";
      if(dayNightValue == Calendar.PM)
      {
         amPmString = " PM";
      }
      if(!tt.changeAlarmMode(false))
      {
         demoForm.removeCommand(ALARM_OFF_CMD);
         demoForm.addCommand(ALARM_ON_CMD);
      }
      if(minValue >= 10)
      {
         System.out.println("Alarm at " + hrValue + " : " +
                                   minValue + amPmString);
      }
      else
      {
         System.out.println("Alarm at " + hrValue + " : 0" +
                                    minValue + amPmString);
      }
   }
}
```

Finally, we need to set a style to our component using its `UIID`. Similar to other components, we create a style, and set it for `TimeTeller` just before instantiating it.

```
//set style for TimeTeller
Style ttStyle = new Style();
ttStyle.setBgColor(0x555555);
ttStyle.setBgSelectionColor(0x555555);
ttStyle.setPadding(0,0,0,0);
ttStyle.setMargin(0,0,0,0);
UIManager.getInstance().setComponentStyle("TimeTeller",
                                              ttStyle);

//create a new TimeTeller instance with a Viewer
TimeViewer tv = new TimeViewer();
tt = new TimeTeller(tv);
```

Enhancements

The `TimeTeller` is meant to be a pedagogic exercise. There are many ways to enhance this component. Working on modifying `TimeTeller` to improve its usability or to add features can be a very useful tool in your learning process. We have already pointed out that providing a proper `AlarmHandler` would be a significant improvement.

If you are interested in getting your hands dirty, then here are some more suggestions:

- There is no reason for not having an alarm facility in the elapsed time mode. Adding this capability would make the component more useful as an elapsed time counter.
- The `setter` and `getter` methods for style in `TimeViewer` support only the styles for the labels. The container style cannot be accessed easily now. It would be good to be able to manipulate the container style too. Similarly, it should be possible to set attributes for the `TimeTeller` container too.
- The offsets for different time zones are hardcoded. It would be very convenient to have a user selectable list to set the desired offset.
- `TimeViewer` is the only available viewer at this time. Why not plug in some more viewers?

This list is not exhaustive, and you can definitely think of more possibilities. So let's get going and enjoy!

Summary

In this chapter, we built up a component using other components. Along the way, we saw how to incorporate several important features that characterize an LWUIT component. These were:

- Decoupling the look part of a component from its logic to permit visual customization
- Handling events such as a *keypress*
- Calling into a processing routine either directly or through the EDT when a predetermined incident takes place
- Setting `style` attributes
- Working out the preferred size

Although we put `TimeTeller` together from off-the-shelf components in the LWUIT library, that is not the only option. It is possible to create a component from scratch by rendering through a graphics object. The **Developer's Guide** that comes with the LWUIT bundle provides a good example of such a component in the chapter entitled **Authoring Components**.

There are two aspects of component building that we have not talked about here—**Animation** and **Background Painter**. These will be taken up in Chapter 11 and in Chapter 12 respectively.

9

Resources Class, Resource File and LWUIT Designer

An LWUIT application usually requires one or more elements in addition to the code itself. Images are very often needed for use as icons. Animated GIFs may be used for animations, and fonts not supported by the given platform are often useful for rendering text. All such elements required for a particular application form its **Resource** bundle. This technique is very convenient for distributing an application, as resource bundles get integrated into JAR files.

One or more of the following types of resources that are supported by LWUIT can be included in a resource bundle:

- Image Resources
- Animation Resources
- Bitmap Fonts
- Localization (L10N) bundles
- Themes

A sixth type named *Data* is also supported. However, the data type is not recommended for general use and shall not be discussed here.

A resource bundle can be created through a set of Ant tasks. The LWUIT environment also includes the **LWUIT Designer**—an extremely useful utility that provides a very convenient method of putting together resource bundles and for viewing them. The LWUIT Designer can also edit existing resource bundles. Here we shall use the LWUIT Designer to demonstrate how a resource bundle can be created.

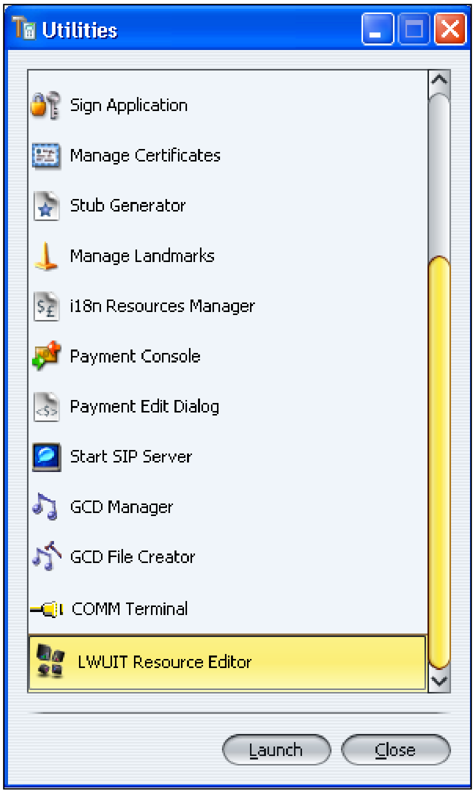
During the process of building an application, a binary resource file is generated. The **Resources** class provides the methods for extracting the resources from this file and for examining them to ascertain what they contain.

In this chapter, we shall start with a brief introduction to the LWUIT Designer and then go on to the procedures for creating each type of resource (except themes, which will be covered in Chapter 10). Next, we shall check out the `Resources` class, and finally, we shall try our hands at building and using a resource file.

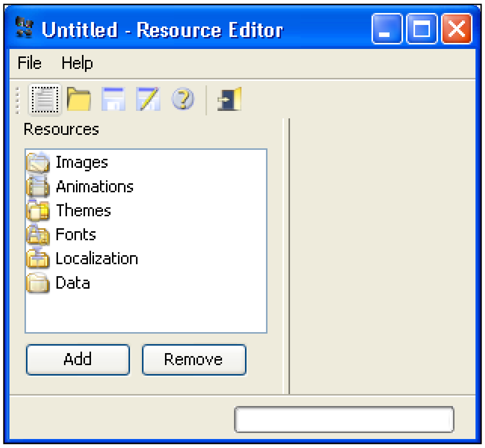
The LWUIT Designer

The original name for this utility was Resource Editor. Although, it has now been renamed, the old name continues to appear in the documentation, as well as on the title bar of the utility itself.

The LWUIT download bundle includes an LWUIT Designer, which is available in the `util` directory of the bundle. It can be launched by double-clicking on the icon. The LWUIT Designer is also integrated into the SWTK and can be accessed by selecting **File | Utilities | LWUIT Resource Editor**.

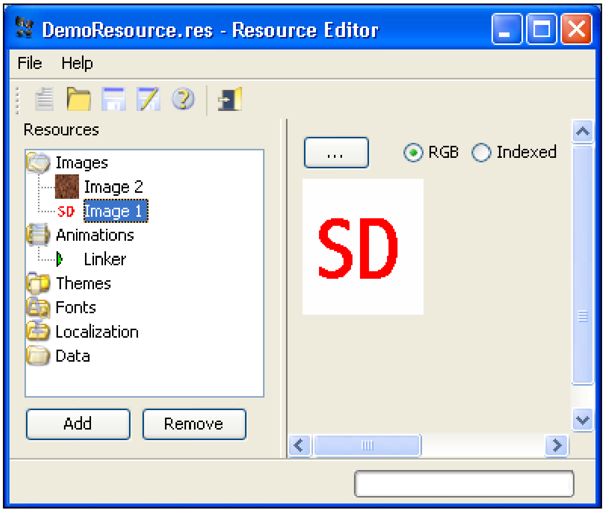


We shall use the SWTK to access the LWUIT Designer, which opens as shown in the following screenshot:



The panel on the left shows a list of resource element types that are supported by this tool. The **Add** button can be used to add elements to a selected category, and the **Remove** button will delete a selected element from a resource file.

An existing resource file can be opened by right clicking on the file and clicking on **ResourceEditor** from the list shown when **Open With** is selected. Alternately, select **File | Open** on the Resource Editor **menu**, and navigate to the desired file. The following screenshot shows that a file named `DemoResource.res` (`.res` is the extension for resource files) has been opened. The file is seen to contain two images—**Image 1** and **Image 2** and an animation—**Linker**.

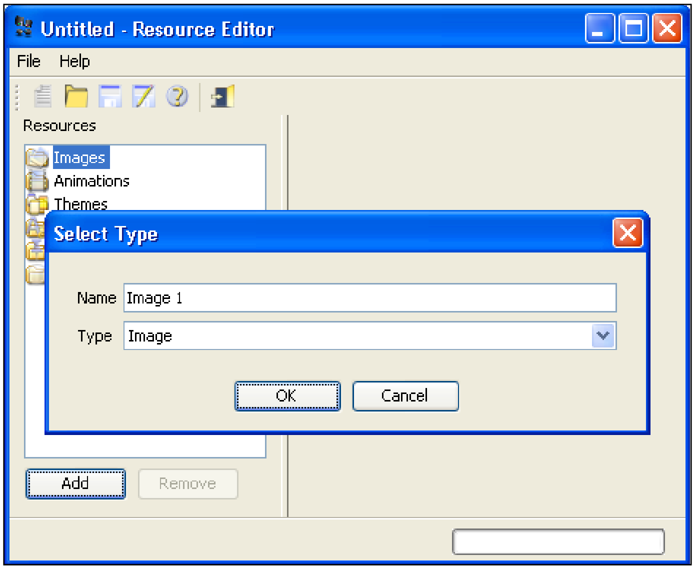


Creating a resource file

It is very easy to create a resource file using the LWUIT Designer. In the following sections, we are going to see how to build such a file containing the elements that we have been talking about. Launch the LWUIT Designer, and add the elements you need as explained below.

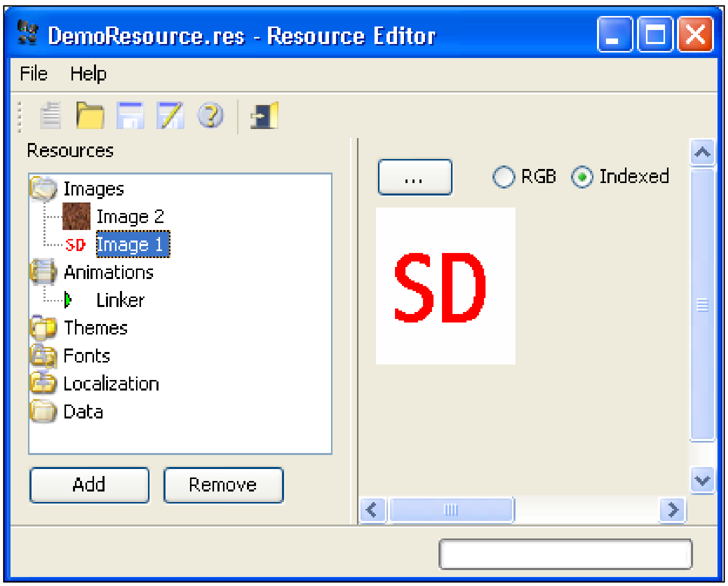
Adding an image

The first step in doing this is to create an image. Photographs taken with a phone or a digital camera are instances of images. Even photographs taken with non-digital cameras can be easily converted into images. Software like the GIMP allows screenshots and hand drawn graphics to be saved as images in standard formats like **gif**, **png** or **jpeg**. Once the image is ready to be packaged into a resource file, select **Images** on the left panel of the LWUIT Designer, and click on the **Add** button. You will see a dialog box like the following:



Enter the name for the image, and click on the **OK** button to browse and locate the target image. Once the image has been selected, it becomes visible under **Images**, as we have already seen for the file `DemoResource`.

An image can be saved either in the RGB or indexed form. An indexed image needs less memory than the RGB version. However, it is slower to render and supports a maximum of 256 colors. Modern small devices are usually fast enough so that the relatively slow rendering of indexed images is not noticeable, and the upper limit of 256 colors is quite adequate in many cases. When adding an image to a resource file, you can select either form. If **Indexed** is selected and the number of colors is found to be more than 256, then the utility offers to reduce it with a corresponding degradation of quality. Image manipulation software products provide a very effective option for indexing images before bundling them into a resource file. The screenshot below shows the radio buttons for selecting **RGB** or **Indexed** mode for the image. Also shown is a browse button (the button with three dots) that can be used to locate an image to replace the existing one.



Adding an animation

An animation for inclusion in a resource bundle is an animated image. Presently, LWUIT supports only the animated GIF type. LWUIT uses the `StaticAnimation` class internally, which is a descendant of the `Image` class, to handle an animation resource. This ancestry allows an animation to be used just like an image—as an icon, as a background, or in any other way relevant for an image. Animated GIFs can be created with standard image manipulation tools. **Linker**, which is the animation included in the `DemoResource` file, was created using the GIMP.

The procedure for adding an animation is pretty similar to that for adding an image:

1. Select **Animations**.
2. Enter the name in the dialog box.
3. Locate the file, select it.
4. And click the **Open** button on the browsing window.

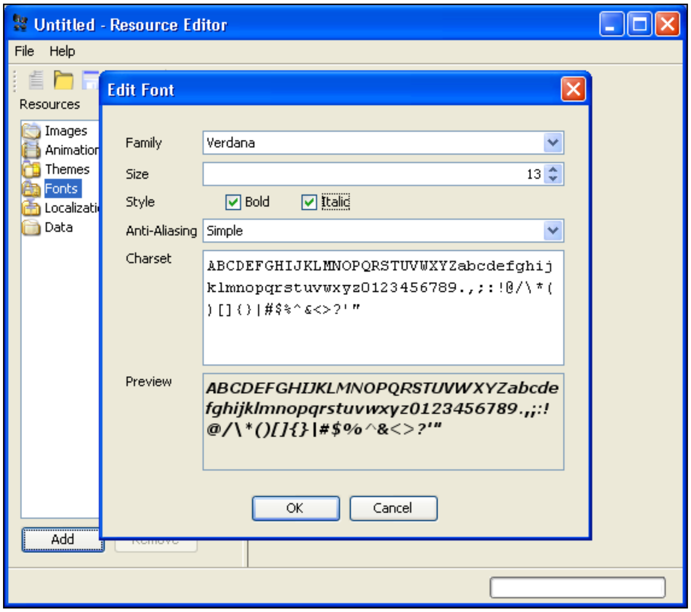
Adding a font

LWUIT supports both system fonts available on a device, as well as those supported by the development environment. The LWUIT Designer creates a bitmap font from any font installed on the platform on which it is running. This font can then be used to render text in an LWUIT based application.

The system fonts that a device offers will not necessarily be identical across diverse platforms. Bitmap fonts will have greater visual consistency and anti-aliasing support, thus making them attractive for multi-device applications.

To add a bitmap font, select the **Fonts** option on the LWUIT Designer, and click on the **Add button.** You will then be prompted to enter the name for the font. Once that has been done, the **Edit Font** dialog will open. You can now choose the font, its style and size, the kind of anti-aliasing to be applied, and the character set to be included. Incidentally, the type of anti-aliasing that may be used depends on the version of Java you are using. Under Java 5, the only options are **Off** and **Simple** (standard). Java 6 offers a number of other options, and the documentation for the `java.awt.RenderingHints` class in Java 6 offers some insight into these additional choices.

There is also a preview window, which shows what the chosen font is going to look like. In the following screenshot a bold, italic, 13 point **Verdana** with simple anti-aliasing has been selected.

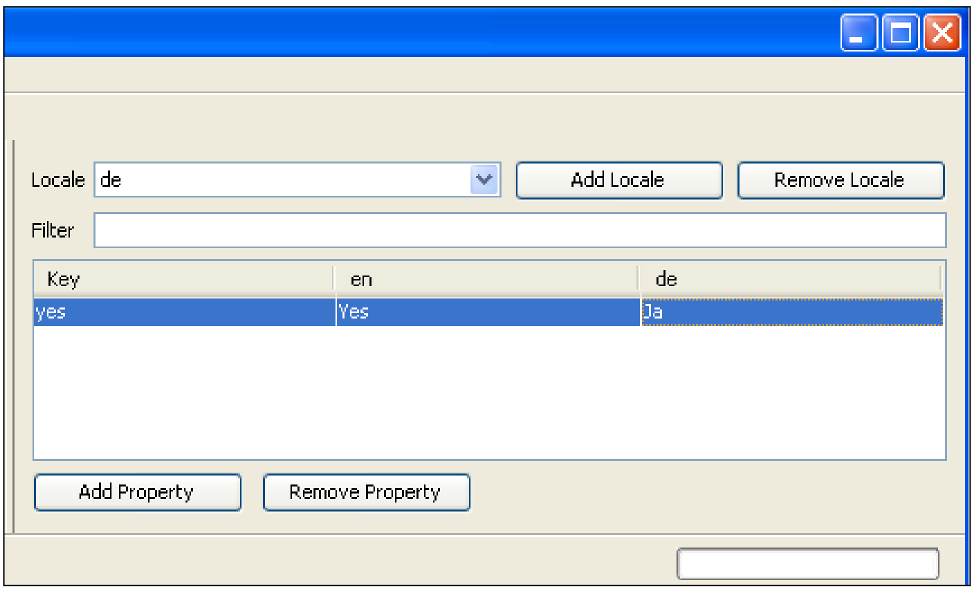


Click on the **OK** button to add the selected font to the Resource File.

Adding a localization resource

Localization enables our applications to adapt to different locales through the use of predefined *key-value* pairs. For example, texts for buttons may be made to change depending upon the country in which an application is running.

The addition of a localization resource is done along similar lines. Select **Localization** and click on the **Add** button. After entering the name of the resource, click on the **OK** button to get the following editing panel on the LWUIT Designer:



A locale can be added or removed by using the **Add Locale** or **Remove Locale** buttons respectively. A **locale,** in this context, is the name of the language for which an item (like a string) is being customized. When a locale is added, it appears to the right of **Key**. The code for languages have been standardized and can be found at `http://www.loc.gov/standards/iso639-2/php/English_list.php`. Use the **Add Property** button to add a key like the **yes** key seen here. Double-click under **en** or **de** in order to enter the corresponding value. To remove a key, use the **Remove Property** button.

The two locales shown refer to English (en) and German (de). This file can now be saved and used for localizing. We will work with a resource bundle, which will contain a number of resource elements including one for localization, and we will also see how to use such a bundle later in this chapter.

Adding a Theme

A **Theme** provides a convenient mechanism for centralized setting of styles for all components of specified types. In Chapter 10, we shall study themes in detail and that is when we shall also see how the LWUIT Designer can be used to build a theme.

Saving a resource file

The completed resource file has to be saved in the `res` folder of the application. To save the file, select **File | Save As** from the menu of LWUIT Designer, give the file a name, and save it in the appropriate folder. If an existing resource file is to be used, then make sure that it is copied into the relevant `res` folder.

The Resources class

The `Resources` class extracts resources from a binary resource file. There is no public constructor for this class. In order to get a resource object, we have to use one of the two methods shown below.

Method	Parameters	Description
`static Resources open (String resource)`	`resource`—a local reference to a resource.	Creates and returns a resource object from the local JAR resource identifier.
`static Resources open (InputStream resource)`	`resource`—the stream from which to read the resource.	Creates and returns a resource object from the given input stream.

The other methods are mostly accessors for getting lists of individual resources from a file or for getting a specific resource. There are also a number of methods for checking the type of a resource.

The methods that tell us the names of a given type of resource contained in a file are of the form `get*ResourceNames`, where the asterisk is to be replaced by the type of resource we are interested in. So, to get a list of images in a resource file, we use the `getImageResourceNames` and a string array containing the names will be returned. If there are no images in the file, then an empty array will be returned.

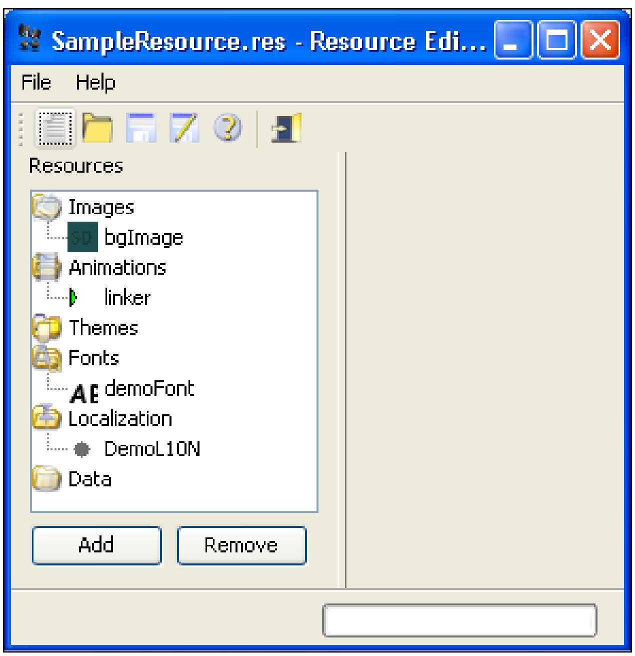
Similarly, there are methods of the form `is*(String name)` that tell us whether a given element belongs to a particular type. Here again the asterisk is a place holder for a resource type. The method `isImage(String name)` will return `true` if the element specified by the string name is indeed an image. Otherwise it will return `false`.

For each type of resource element there is a method to extract it from a resource bundle. We shall use these methods in our examples in this and the following chapters whenever resources are used.

When we open a resource file, it is loaded into the memory in its entirety. If all resource elements are packaged into a single file, then the whole file will need to be accommodated in the memory whenever the `open` method is executed. This may not be desirable for all devices from the point of view of memory usage. It is good practice to group resources into separate files, such that only the file required for current use is loaded into memory. This allows memory allocation to be optimized.

The SampleResource demo

We shall now see how the resource elements contained in a resource file can be accessed and used. The file that will be used in this demo is the `SampleResource.res`, which is shown through the LWUIT Designer in the following screenshot:



The file contains an **Image**, an **Animation**, a (bitmap) **Font,** and a **Localization** bundle. In the next screenshot, we can see how these resource elements are incorporated into the form. The line of text at the bottom looks truncated as it is tickering.

The resource file is opened, and the individual elements are extracted, as shown in the code snippet below:

```
Resources sample = null;
.
.
.
try
{
   //Load resource
   sample = Resources.open("/SampleResource.res");
}
catch(java.io.IOException ioe)
{
}
if(sample != null)
{
   //get the image for background
   Image bgImage = sample.getImage("bgImage");
   //get the animation
   Image linkImage = sample.getImage("linker");
```

```
            //get font
            Font demoFont = sample.getFont("demoFont");

            //get localization resource
            //Hashtable locHash = sample.getL10N(«DemoL10N», «fr»);
            //Hashtable locHash = sample.getL10N(«DemoL10N», «es»);
            Hashtable locHash = sample.getL10N(«DemoL10N», «en»);
            .
            .
            .
        }
```

The code here is pretty straightforward. There are two lines of code for loading the localization bundle that have been commented out. These will be discussed a little later, when we take a more detailed look at a localization use case.

Once the elements are available, we set `bgImage` as the background of the form, after making sure that an image was indeed extracted from the resource file.

```
            //set background image for demoForm
            if(bgImage != null)
            {
               demoForm.getStyle().setBgImage(bgImage);
            }
            else
            {
            //there is no image so just set the background color
               demoForm.getStyle().setBgColor(0x555555);
            }
```

The next step is to create labels using the animated image and the bitmap font. We have already seen that an animation can be treated just like an image. So we use `linkImage` as an icon to create a label.

```
            //create common label style
            Style labelStyle = new Style();
            labelStyle.setBgTransparency(0);
            labelStyle.setFont(font);
            UIManager.getInstance().setComponentStyle("Label",
                                                      labelStyle);

            //label for animated image
            Label linkLabel;
            if(linkImage != null)
            {
               linkLabel = new Label(linkImage);
               linkLabel.setText("Animated image");
```

```
}
else
{
   linkLabel = new Label("Animated image not accesible");
}
//set individual style for linkLabel
linkLabel.getStyle().setFgColor(0x0000ff);
linkLabel.getStyle().setBorder(Border.createLineBorder(3,
                                                   0x0000ff));
linkLabel.getStyle().setBgColor(0xffffff);
linkLabel.getStyle().setBgTransparency((byte)255);
```

The font extracted from the resource file is used as a style attribute for the `fontLabel`.

```
//label for bitmap font
Label fontLabel = new Label("This is a bitmap font");
//set individual style for fontLabel
fontLabel.getStyle().setFgColor(0xe8dd21);
if(demoFont == null)
{
   demoFont = Font.createSystemFont(Font.FACE_
     PROPORTIONAL,Font.STYLE_PLAIN,Font.SIZE_MEDIUM);
   fontLabel.setText("Bitmap font not accesible");
}
fontLabel.getStyle().setFont(demoFont);
```

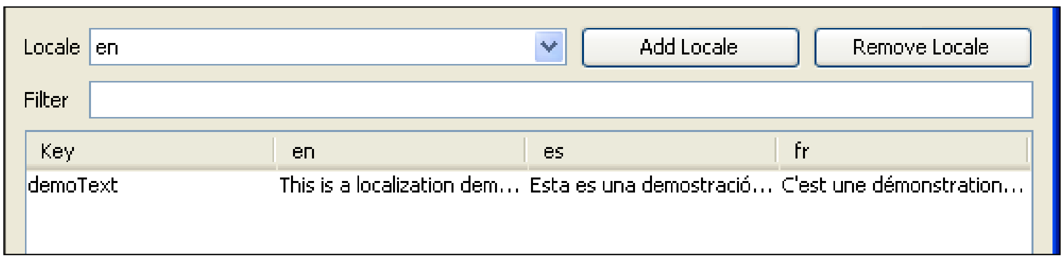
So far the resources and their corresponding labels have used the coding methodology that we have already explored in the preceding chapters. However, the use of the localization element involves techniques that we have not yet encountered. So we shall now examine the ways in which localization can be used in practice.

Localization, or **L10N**, is the basis for adapting applications for diverse languages. Incidentally, the number 10 in L10N refers to the number of characters between the first 'L' and 'N' in the word LOCALIZATION. Texts for labels and lists, titles for forms and dialogs, commands on menus and soft buttons—all of these can be translated into the language of the region where a phone is being used through the proper use of localization techniques.

Broadly speaking, there are three actions that have to be taken to implement localization:

- Establish the locale to be used
- Get the corresponding localization element if it exists
- Apply localization as required

There are two basic ways of going about this. I call them the *manual* and the *automatic* techniques. Let's look at them one-by–one, but first the localization element itself.



The above screenshot shows the `DemoL10N` element, which has only one key, that is, the *demoText*. This key has three values corresponding to the three locales for which our application is designed. They are **en** (for English), **es** (for Spanish), and **fr** (for French). Our discussion on using localization will be based on this localization resource.

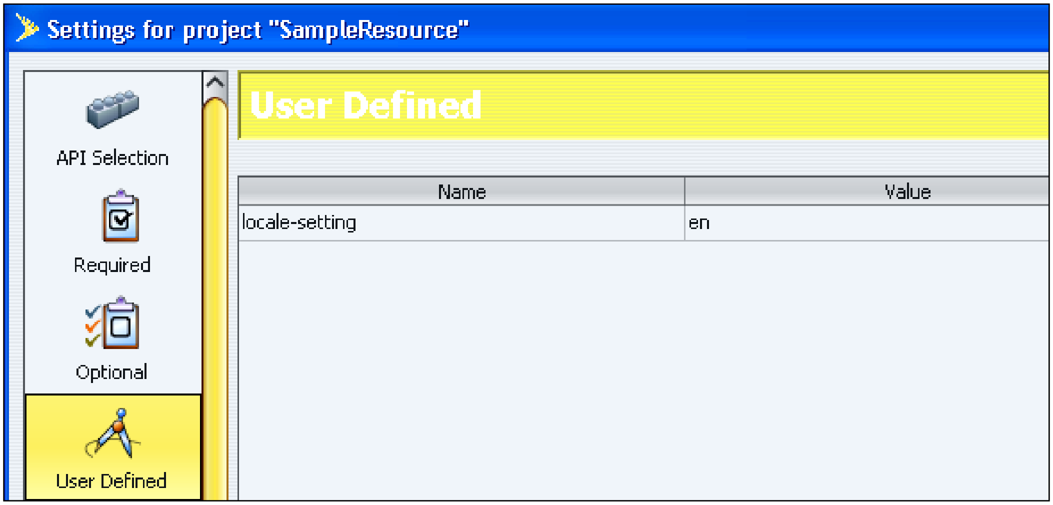
The manual approach

In this approach, all three steps require explicit and detailed action on the part of the application. The locale can be established in two ways. We can directly hardcode the loading of a localization resource, as has been done in our example. Let's recap.

```
//get localization resource
//Hashtable locHash = sample.getL10N("DemoL10N", "fr");
//Hashtable locHash = sample.getL10N("DemoL10N", "es");
Hashtable locHash = sample.getL10N("DemoL10N", "en");
```

Here we have loaded the `hashtable` corresponding to **en**. Loading one of the other hashtables would display the corresponding version.

The second way to set up a locale is to put it in the JAD file as an application property. When you click on the **Settings** tab on the SWTK, the screen that opens allows user-defined properties to be set. Let's add the **locale setting** property with a value of **en**.



The following code, which uses the `getAppProperty` method of the `MIDlet` class reads the value of the property and accordingly gets the proper `hashtable`:

```
Hashtable locHash = null;
 String appLocale = getAppProperty("locale-setting");
 locHash = sample.getL10N("DemoL10N", appLocale);
```

When we have the hashtable we want, we can use it to translate the string like this to get the string corresponding to the specified locale:

```
String locText = locHash.get("demoText");
locLabel = new Label(locText);
```

Alternately, we can set `locHash` as the global `hashtable`, and then use the `localize` method of the `UIManager` class to localize text strings as required.

```
UIManager.getInstance().setResourceBundle(locHash);
  .
  .
  .
  string locText = UIManager.getInstance().localize("demoText",
                     "This is a localization demo (English)");
  Label locLabel = new Label(locText);
```

The `localize` method looks through the installed `hashtable` and gets the string corresponding to `demoText`. If such a *key-value* pair cannot be found, then the second parameter is returned as the default. By the way, the three code sections that are shown are for information only and have not been used in the demo.

As you can see, all the three steps enumerated earlier for implementing localization have been executed here through explicit coding or parameterization and will produce the correct result only for the supported option. Whenever the application needs to be tailored for a new locale, changes have to be made in the code or in the JAD settings, and a new JAR file has to be built. This is essential, even though the localization bundle itself may already contain all necessary information for the new locale. Obviously, this approach has no adaptability and works only when it is possible to predict the environment in which the application is going to run.

The automatic approach

The ideal way for implementing localization should make an application completely self sufficient so that it can determine the applicable locale, load the proper `hashtable`, and localize all applicable strings without any need for code modification or recompilation. The automatic approach allows us to do exactly that.

In our demo application, you will find the following code sections:

```
/***Start of statements for manual localization***/

//get localization resource
//Hashtable locHash = sample.getL10N("DemoL10N", "fr");
//Hashtable locHash = sample.getL10N("DemoL10N", "es");
//Hashtable locHash = sample.getL10N("DemoL10N", "en");

/***End of statements for manual localization***/

/***Start of statements for automatic localization***/

 String locName = "DemoL10N";
 String localeString = System.getProperty(
                   "microedition.locale");
 Hashtable locHash = getL10N(sample, locName, localeString);

/***End of statements for automatic localization***/
```

Earlier we had come across the statements for manual localization. Note that all the statements in that set have now been commented out. The second set of statements finds out what is the applicable locale for the device, and loads the corresponding `hashtable`.

The static method `getProperty(String key)` of the `System` class returns a locale for a given device when invoked with `"microedition.locale"` as the parameter. The returned string can be a two letter language code or a longer one containing a country code as well. The `getL10N` method first eliminates the country code, if any, and then gets and returns the `hashtable` corresponding to the language code. If no such `hashtable` is found, then the one for English is returned as a default.

```
private Hashtable getL10N(Resources r, String locName, String l)
{
   if(l.length() > 2)
   {
      l = l.substring(0, 2);
   }
   Hashtable retRsrc = r.getL10N(locName, l);
   if(retRsrc == null)
   {
      retRsrc = r.getL10N(locName, "en");
   }
   return retRsrc;
}
```

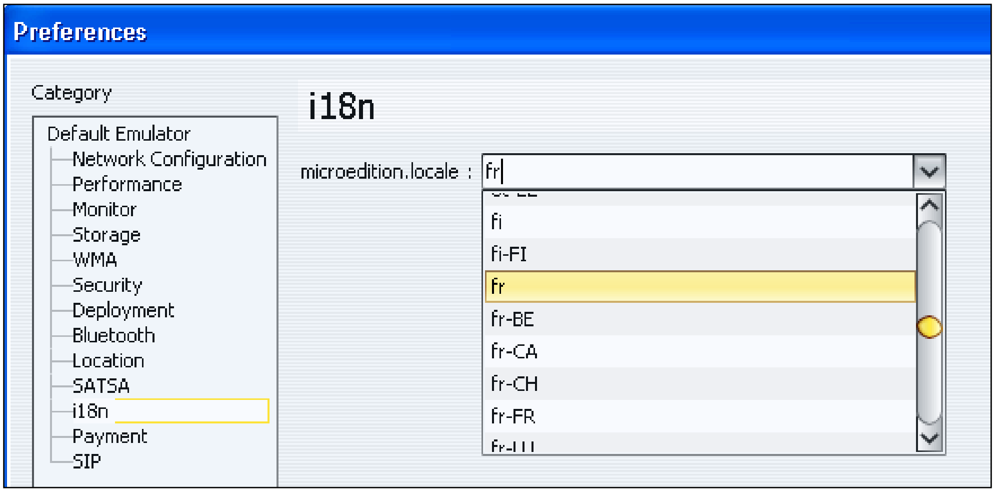
The reference returned by `getL10N` is then set as the global hashtable.

```
UIManager.getInstance().setResourceBundle(locHash);
```

The actual translation is taken care of by LWUIT through a very convenient feature. All LWUIT components have a method to localize the texts they use. This is a completely transparent process. All that has to be done to use this feature is the *key*—and not the actual text—must be used for setting up a component. The key used in the localization bundle for the `locLabel` text is `demoText`. So we use that key as the parameter in the constructor when creating `locLabel`, and the rest is taken care of by LWUIT. However, we need to ensure that a value corresponding to the key exists in the hashtable. Otherwise the key itself will be used as the text for the label.

```
locLabel = new Label("demoText");
```

To test the demo with locales other than **en**, select **Edit | Preferences | i18n** on the SWTK, and the following screen will come up:



Select the desired locale and click on the **OK** button at the bottom of the screen. The demo can now be tested for the locale that has been set. You can see on the drop-down list that the lowercase language code may be extended by an uppercase code for a country. For instance, **en-US** stands for the US version of English. The term **i18n** seen here means **internationalization**, and 18 is the number of characters between the first alphabet 'i' and the last alphabet 'n'. The following screenshot shows the demo running with the locale (French) selected above:

We can now see that under the *automatic* technique, an application only needs to make sure that the localization bundle contains all information required for the targeted locales. Thereafter, there is no need for any code modification or any for recompilation. The application will be able to operate under any locale. Also, in case an unsupported locale is encountered, it will at least work with the default values for the keys.

Summary

In this chapter, we have seen what the various types of resources supported by LWUIT are, how to use the LWUIT Designer to build resource bundles, and how the Resources class enables us to use the contents of such bundles. The LWUIT Designer is a very convenient tool, which is equivalent to Ant tasks for building resource bundles with just one difference—when a resource file is built by the LWUIT Designer, a theme element in that file cannot access bitmap fonts located in a different resource file.

We have also had a good look at localization, and the `SampleResource` demo application showed us how to use it. In the next chapter, we are going to study themes in detail, and see how the LWUIT Designer can be used to build themes.

10
Using Themes

In the preceding chapters, we have seen how to set styles for components. In an application with a large number of UI components, setting attributes for each can be a tedious task and can also lead to errors. A **Theme** allows us to set the style attributes for an entire class of components in a single place. This not only simplifies the task of setting attributes for all components of a particular type but also ensures that any newly added component will look just like all the others of the same type in the application. A theme thereby establishes a visual coherence through all the screens of an application.

In this chapter, we shall study themes and their usage in detail through the following steps:

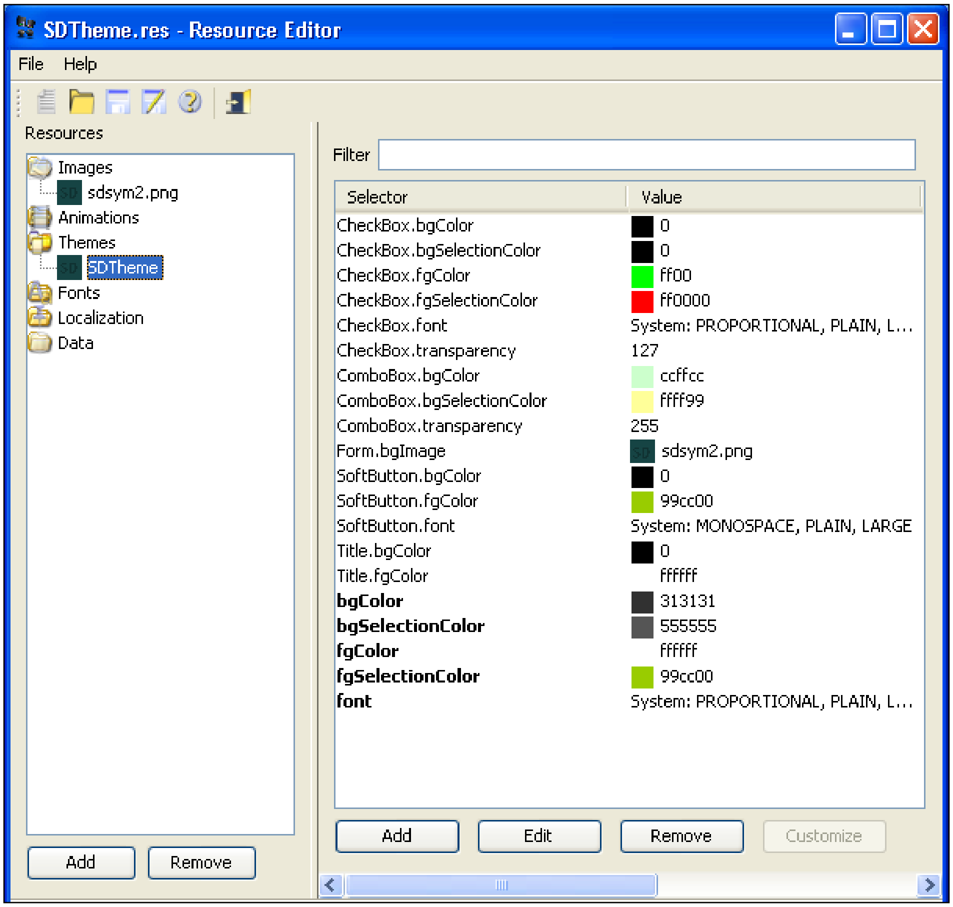
- View an existing theme using the LWUIT Designer
- Edit a theme
- Build a new theme
- Preview the new theme on LWUIT demo MIDlet
- Use the new theme in a demo MIDlet
- Use your own component in a theme

Working with theme files

A theme file is conceptually similar to CSS while its implementation is like that of a Java properties file. Essentially a theme is a list of *key-value* pairs with an attribute being a *key* and its value being the second part of the *key-value* pair An entry in the list may be `Form.bgColor= 555555`. This entry specifies that the background color of all forms in the application will be (hex) 555555 in the RGB format. The list is implemented as a hashtable.

Viewing a theme file

A theme is packaged into a resource file that can also hold, as we have already seen, other items like images, animations, bitmap fonts, and so on. The fact that a theme is an element in a resource bundle means it can be created, viewed, and edited using the LWUIT Designer. The following screenshot shows a theme file viewed through the LWUIT Designer:

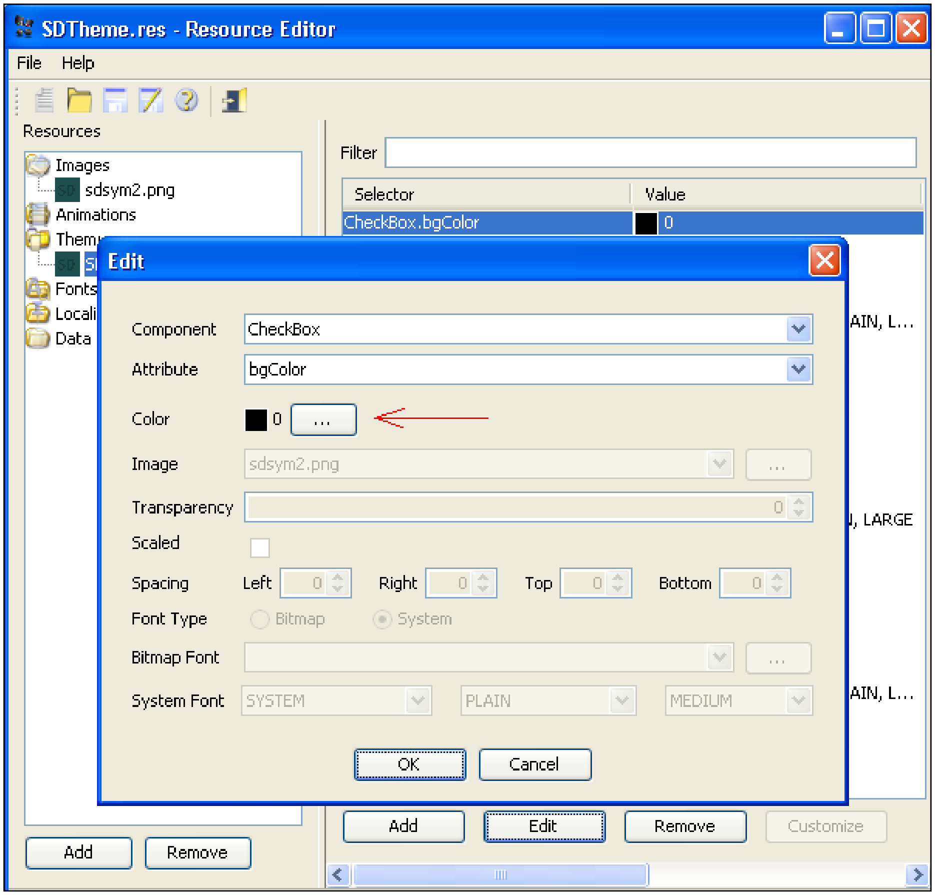


The first point to note is that there are five entries at the bottom, which appear in bold letters. All such entries are the defaults. To take an example, the only component-specific font setting in the theme shown above is for the soft button. The font for the form title, as well as that for the strings in other components is not defined. These strings will be rendered with the default font.

A theme file can contain images, animations, and fonts—both bitmap and system—as values. Depending on the type of key, values can be numbers, filenames or descriptions along with thumbnails where applicable.

Editing a theme file

In order to modify an entry in the theme file, select the row, and click on the **Edit** button. The dialog for edit will open, as shown in the following screenshot:



Clicking on the browse button (the button with three dots and marked by the arrow) will open a color chooser from which the value of the selected color will be directly entered into the edit dialog. The edit dialog has fields corresponding to various keys, and depending on the one selected for editing, the relevant field will be enabled. Once a value is edited, click on the **OK** button to enter the new value into the theme file. In order to abort editing, click on the **Cancel** button.

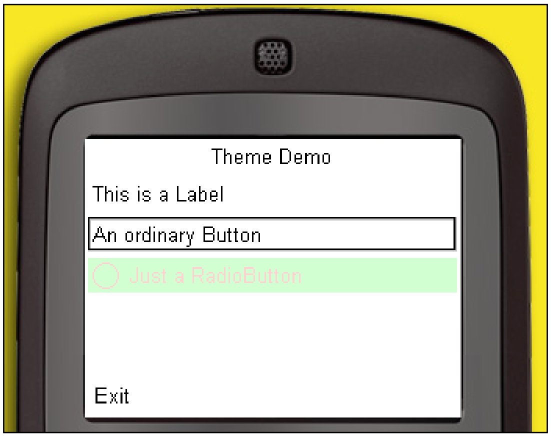
Populating a theme

We shall now proceed to build a new theme file and see how it affects the appearance of a screen. The application used here is `DemoTheme`, and the code snippet below shows that we have set up a form with a label, a button, and a radio button.

```
//create a new form
 Form demoForm = new Form("Theme Demo");
//demoForm.setLayout(new BorderLayout());
 demoForm.setLayout(new BoxLayout(BoxLayout.Y_AXIS));
//create and add 'Exit' command to the form
//the command id is 0
 demoForm.addCommand(new Command("Exit", 1));
//this MIDlet is the listener for the form's command
 demoForm.setCommandListener(this);
//label
 Label label = new Label("This is a Label");
//button
 Button button = new Button("An ordinary Button");
//radiobutton
 RadioButton rButton = new RadioButton("Just a RadioButton");
//timeteller -- a custom component
//TimeTeller timeTeller = new TimeTeller();
//set style for timeLabel and titleLabel(in TimeViewer)
//these parts of TimeTeller cannot be themed
//because they belong to TimeViewer which does not
//have any UIID
/*Style tStyle = new Style();
 tStyle.setBgColor(0x556b3f);
 tStyle.setFgColor(0xe8dd21);
 tStyle.setBorder(Border.createRoundBorder(5, 5));
 timeTeller.setTitleStyle(tStyle);
 Style tmStyle = timeTeller.getTimeStyle();
 tmStyle.setBgColor(0xff0000);
 tmStyle.setFgColor(0xe8dd21);
 tmStyle.setBgTransparency(80);
 tmStyle.setBorder(Border.createRoundBorder(5, 5));*/
//add the widgets to demoForm
 demoForm.addComponent(label);
 demoForm.addComponent(button);
 demoForm.addComponent(rButton);
//demoForm.addComponent(timeTeller);
//show the form
 demoForm.show();
```

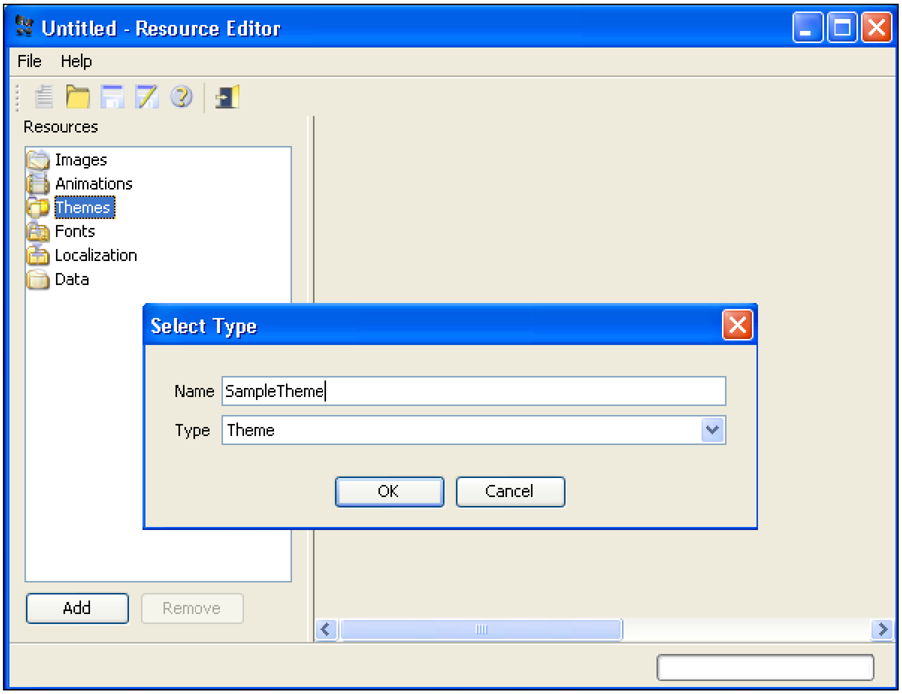
The statements for `TimeTeller` have been commented out. They will have to be uncommented to produce the screenshots in the section dealing with setting a theme for a custom component.

The basic structure of the code is the same as that in the examples that we have come across so far, but with one difference—we do not have any statement for style setting this time around. That is because we intend to use *theming* to control the look of the form and the components on it. If we compile and run the code in its present form, then we get the following (expected) look.

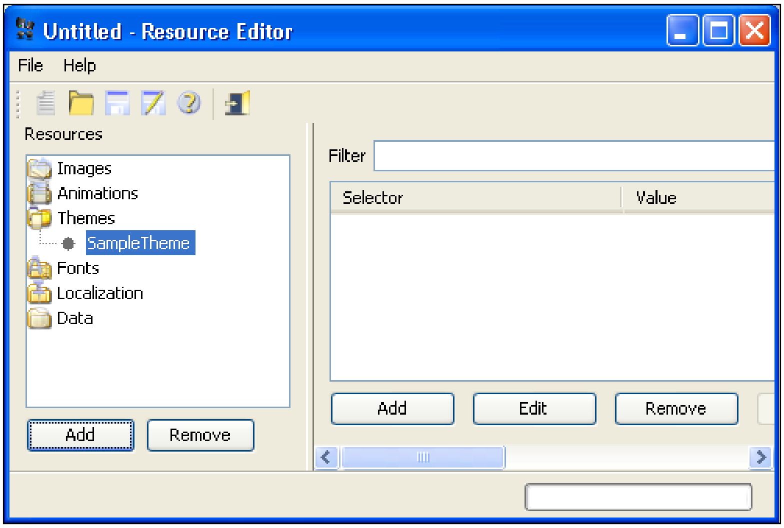


All the components have now been rendered with default attributes. In order to change the way the form looks, we are going to build a theme file—`SampleTheme`—that will contain the attributes required. We start by opening the LWUIT Designer through the SWTK. Had a resource file been present in the `res` folder of the project, we could have opened it in the LWUIT Designer by double-clicking on that file in the SWTK screen. In this case, as there is no such file, we launch the LWUIT Designer through the SWTK menu.

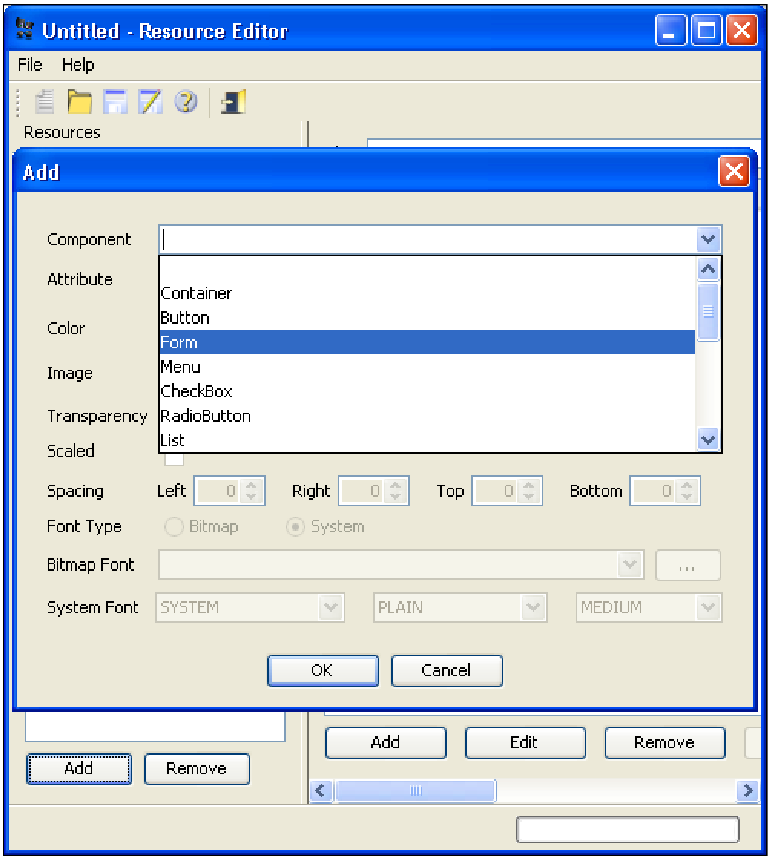
The following screenshot shows the result of selecting **Themes**, and then clicking on the **Add** button:



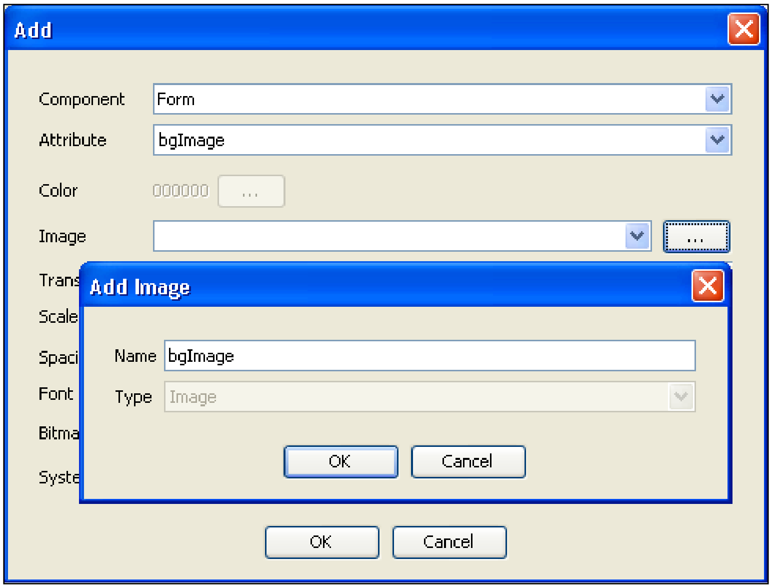
The name of the theme is typed in, as shown in the previous screenshot. Clicking on the **OK** button now creates an empty theme file, which is shown under **Themes**.



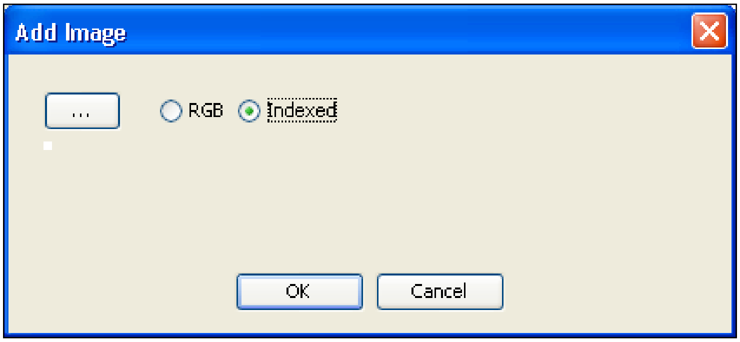
Our first target for styling will be the form including the title and menu bars. If we click on the **Add** button in the right panel, the **Add** dialog will open. We can see this dialog below with the drop-down list for the **Component** field.



Form is selected from this list. Similarly, the drop-down list for **Attribute** shows all the attributes that can be set. From this list we select **bgImage**, and we are prompted to enter the name for the image, which is **bgImage** in our case.



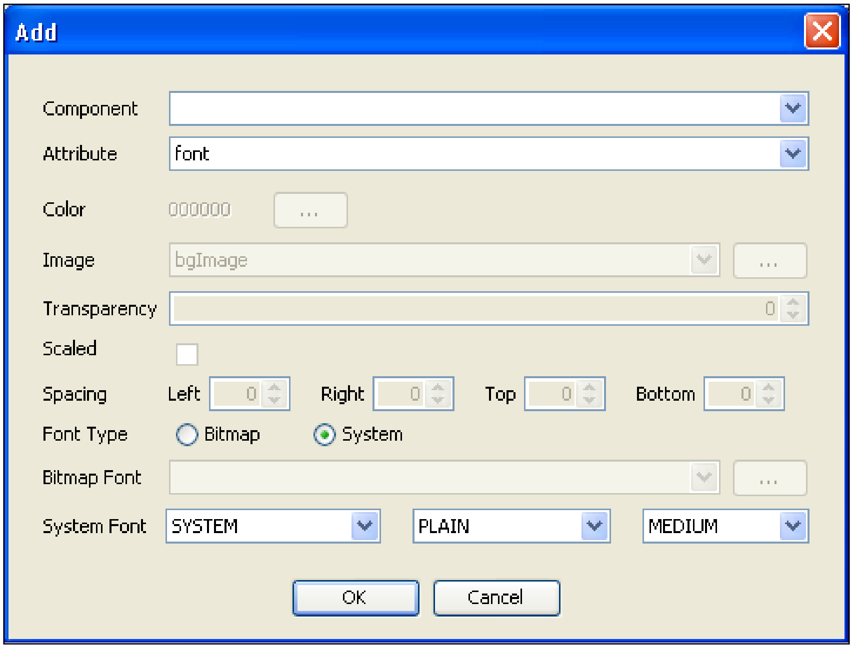
The next step is to close the **Add Image** dialog by clicking on the **OK** button. As we have not added any image to this resource file as yet, the **Image** field above is blank. In order to select an image, we have to click on the browse button on the right of the **Image** field to display the following dialog.



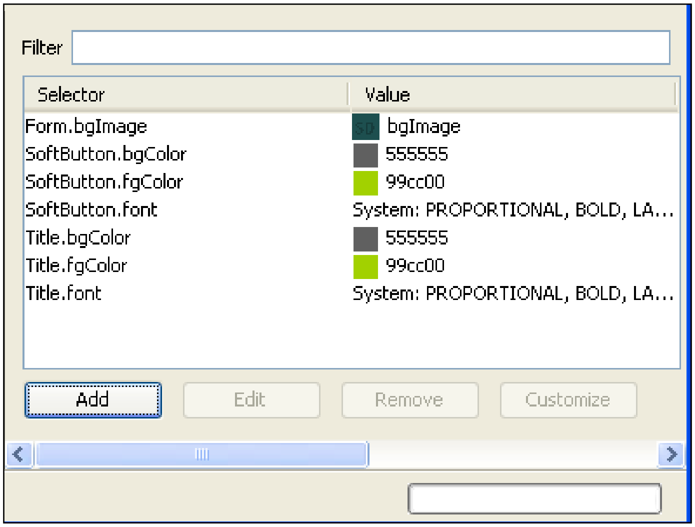
Again, the browse button has to be used to locate the desired image file. We confirm our selection through the successive dialogs to add the image as the one to be shown on the background of the form.

As we can see, the process of adding an attribute to a component is pretty intuitive. In a similar manner, we can set the various background and foreground colors for the components. When setting a color attribute, the dialog will have the familiar browse button to select the color from a color chooser dialog.

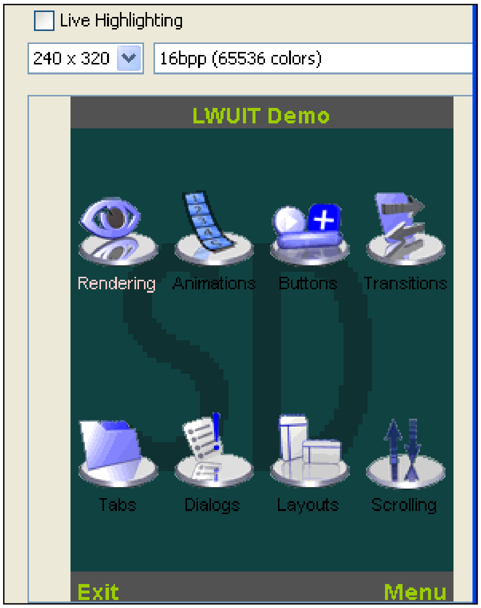
For setting a font, the dialog has two radio buttons to select **Bitmap** or **System** fonts. If **Bitmap** is selected, then the available bitmap fonts in the resource file will be shown in the corresponding field. If there are no bitmap fonts in the resource file, then the required font will have to be selected by clicking on the browse button, which will initiate the same sequence of operations that we saw in Chapter 9 for adding a bitmap font. With the **System** button selected, on the other hand, the applicable parameter fields will become active.



Once we have set the attributes for the form, its title and its menu, the theme file looks like the following screenshot:



Now, it is time to see what we have done to our form. One way of doing this is to run the application. But the LWUIT Designer provides a very convenient way of checking the result of our actions on a theme file through the preview panel. If you maximize the LWUIT Designer window, then this panel will appear on the extreme right of the window whenever a theme file is selected.

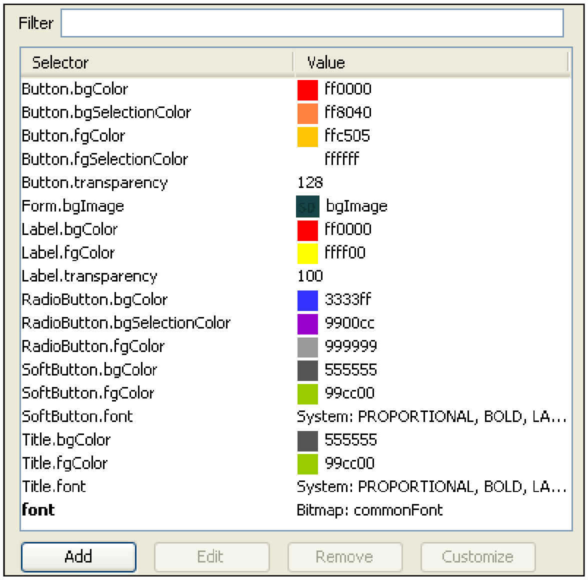


What we see on the preview panel is how the LWUITDemo application (it is a part of the LWUIT download bundle) would look with the attributes specified in the theme file that we are dealing with. This preview keeps showing the effects of style attributes as we enter them into the theme. The style settings for the form, the title and the menu do appear to have produced the intended result. To cross-check, let us save the file and run our demo so that we can see how the screen looks now. However, for the theme file to take effect, it has to be installed, and that is quite easily done as we see in the code below:

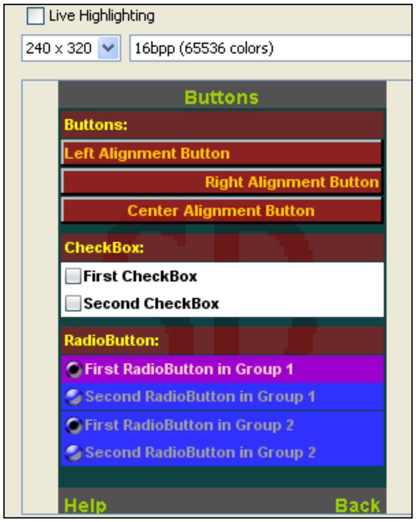
```
Resources sampleThemeResource =
                Resources.open("/SampleTheme.res");
Hashtable sampleTheme =
                sampleThemeResource.getTheme("SampleTheme");
UIManager.getInstance().setThemeProps(sampleTheme);
```

With that done, we can compile and run the code, and as we see, the form together with the title and menu, does indeed look the way we would expect it to after the preview.

We now turn our attention to the label, the button, and the radio button. The following is the complete theme file to set attributes for all the components on the form. Note that there is no font setting for the label, the button, or the radio button. Instead, there is a default font, which is effective for all these three components.



Before we save this file, let us preview the theme. The `LWUIT Demo` on the preview panel is not merely an image. It works as well. So, to preview the look of the label, the button, and the radio button, we select **Buttons** on the demo, and the following screen opens on the preview panel:



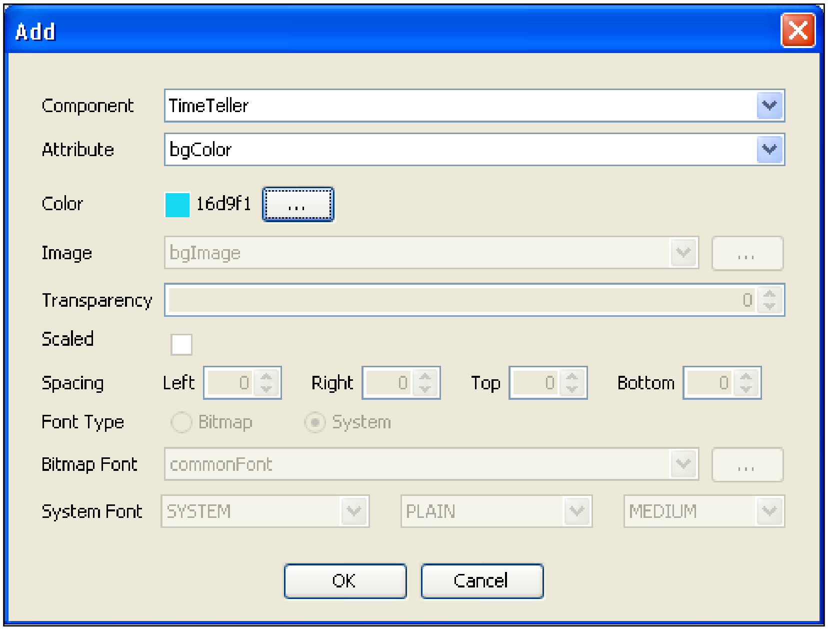
The styles appear to be acceptable. We have not set any attribute for the check boxes, and the defaults become applicable to them. However, we did set a default font, and this is effective for the check boxes too. The file can now be saved. The final result as applied to our form is shown in the following screenshot:

Theming custom components

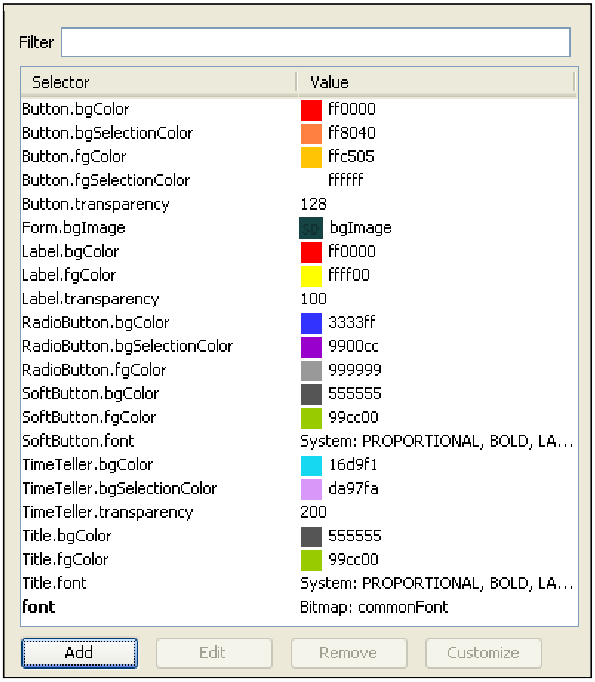
So far our theme file has specified attributes for the standard components supported by the library. It is possible to use theming for custom components too. To try this out, we shall use the `TimeTeller` component that we had created in Chapter 8. The two interfaces and the two classes that together make up that component have been put into a package named *book.newcomp* to make the code better organized. A timeteller is added to the form by uncommenting the relevant statements shown in the MIDlet code snippet listed earlier in this chapter. However, note that we have not set any style for the timeteller instance.

Without any entry for `TimeTeller` in the theme file, the screen looks similar to the following screenshot:

We can see that the two labels of the timeteller are properly styled, while the overall component background has the default color. This happens because the labels have been explicitly styled through code while creating the timeteller. If you experiment with the theme file, then you will see that it is not possible to affect the styles of the `TimeViewer` part of the component through the file. This can be explained when we consider how theming works. When the `setThemeProps` method is executed, the UIManager instance transfers values from the theme file into respective style objects by using the `UIID` of a component as the key. Obviously, if a component does not have its own `UIID`, its style cannot be set through the theme. The `TimeViewer` class has not been allocated a `UIID` and that is why it will not be affected by any entry in the theme file. `TimeTeller`, on the other hand, does have a `UIID`, and we can therefore set its attributes through the theme file. In order to do that, we click on the **Add** button to get the **Add** dialog. In the **Component** field, we type in **TimeTeller** and set **bgColor** following the usual procedure.



A click on the **OK** button enters the value in the theme. The following screenshot shows three entries for `TimeTeller`:



The result of setting these attributes can be seen in the following screenshot:

Manual styling versus theming

We know that an attribute can be set for a specific widget by using a `setter` method of the `Style` class. Let's take a concrete example. In our demo MIDlet, we have manually set background colors for the two labels of the timeteller. We have also defined a different set of background colors for labels in general through the theme. We need to understand which setting takes precedence when conflicting attributes are set in this way.

The API documentation tells us that there are two types of methods for setting attributes. For setting background colors, the methods are `setBgColor(int bgColor)` and `setBgColor(int bgColor, boolean override)`. If the first method is used to manually set the background color of a widget, then a theme file entry will not be effective for that particular component instance. However, all other instances of the same component will be styled as per the theme file, provided manual styling using the same method has not been done. In this case, we have used the `setBgColor(int bgColor)` method to set background colors for the two labels within the timeteller. Therefore, the theme file has no effect on these two labels, although it does determine the corresponding color for the other label on the form. On the other hand, when the `setBgColor(int bgColor, boolean override)` method is used and the Boolean parameter is `true`, theme settings will override any manual styling.

There is another way to allow a theme to override manually set style attributes. If we create a new style object and pass all the options in the constructor, then setting a theme file will change the attributes of such a style object.

Theming on the fly

One feature of theming that we have not talked about so far is that it is possible to change themes at runtime by using the `setThemeProps` method of `UIManager`. But when a theme is set on the fly there will be, in general, components that are not visible, and the effect of setting a theme on these components is not predictable. In order to make sure that even the components that are not visible have their styles properly updated, you should call the `refreshTheme` method using code like this:

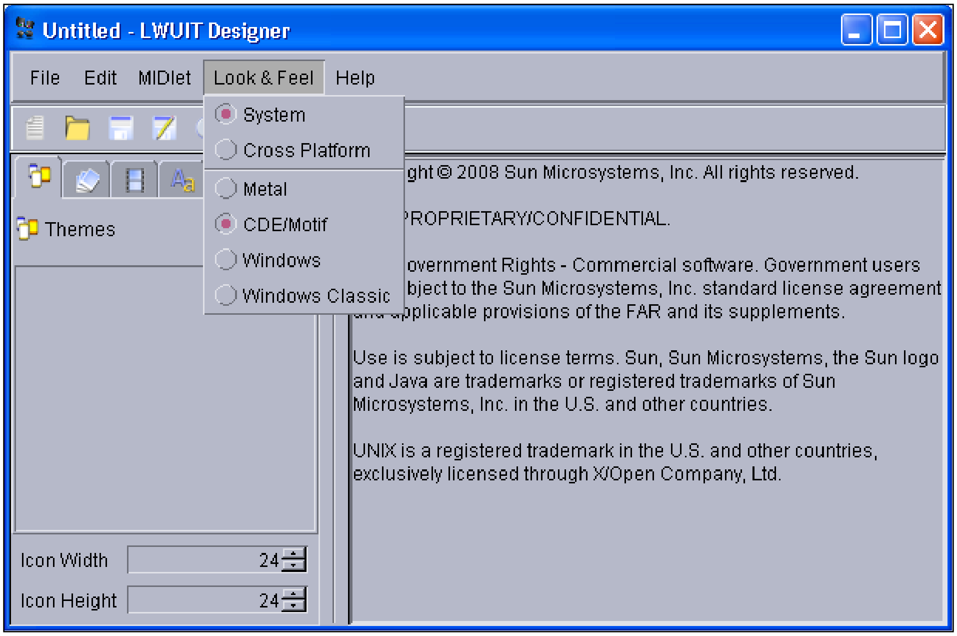
```
Display.getInstance().getCurrent().refreshTheme();
```

When a theme is installed at runtime, there may be form instances that have been created earlier and are to be displayed later. In order that the newly installed theme may take effect on these forms too, it is necessary to call `refreshTheme` on all such forms before they are shown on screen. For forms that are created after the theme is set, no such action is required, as the respective style objects will be initialized to the theme settings. In the current example, `demoForm` was instantiated after the theme was installed, and accordingly, `refreshTheme` was not invoked.

New version of the LWUIT Designer

The example in this chapter has been developed on the SWTK and the LWUIT Designer (Resource Editor) that comes with it. This is very convenient, as the resource bundle can be opened for viewing and editing from the SWTK console. The LWUIT download bundle also includes an **LWUIT Designer**, which offers some additional capabilities. In this section, we shall examine this version and see what is different about it.

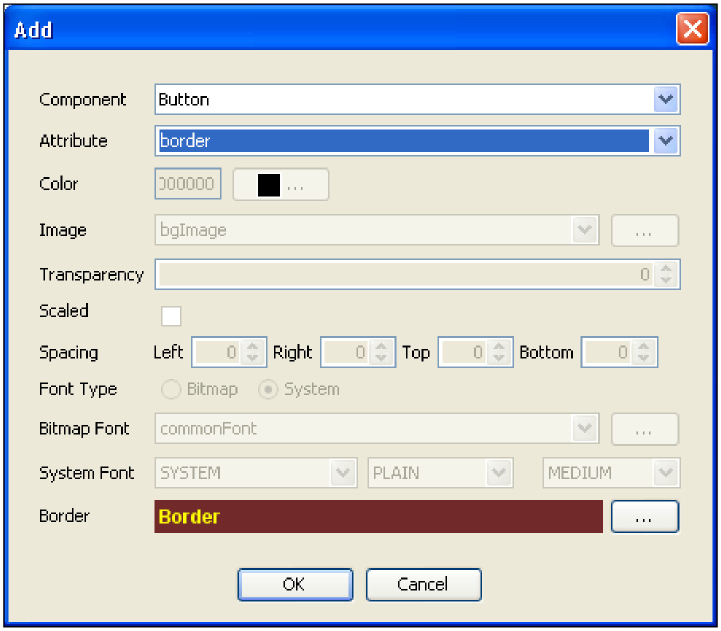
The first impression we get from the next screenshot is that of a totally different look. We then realize that it also has the new name on the title bar.



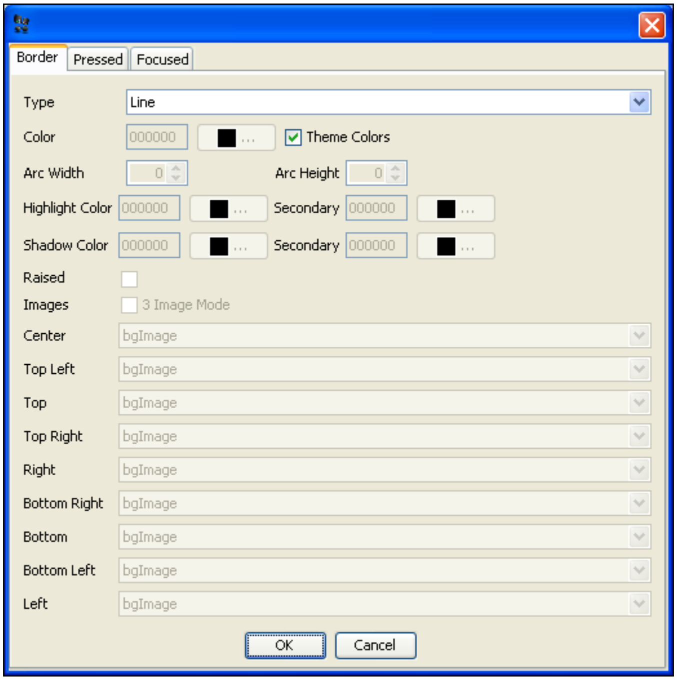
However, the really significant difference is that this version of the **LWUIT Designer** supports a much wider variety of components and also an additional attribute. Some of the additions to the list of supported components are as follows:

- ComboBoxPopup
- Command
- DialogBody
- ScrollThumb

The additional attribute that has been included is **border**. We shall use this edition of LWUIT Designer to add a border to the button in our demo. We will click on the **Add** button on the panel that shows the theme listing. We then select **Button** from the list of components and **border** from the drop-down attribute list on the **Add** dialog.



The browse button next to the **Border** field opens a dialog for specifying the kind of border we want.



Here we have selected a line border and have checked the box that will use the color scheme from the theme for the border. The resulting border for the button (**An ordinary Button**) is shown in the following screenshot:

Word of caution

When you create or modify a theme file using the new versions of LWUIT Designer, you may not be able to open the file through the version that comes with the SWTK. However, you will still be able to run the application on the SWTK, and the new or modified theme will work properly.

Summary

In this chapter, we had a detailed discussion on themes—what they are, how to create them, and how to use them in applications. The steps we followed during our exploration of this topic are:

- We saw how to use the LWUIT Designer for viewing and editing an existing theme
- We built up a new theme from scratch
- As we built the theme, we previewed its effects
- The new theme was installed in our MIDlet, and the proper way to ensure that a newly installed theme works on all existing forms was studied
- A custom component was themed
- We saw how to make theming and manual styling work together without contentions
- A second version of LWUIT Designer was also tried out

Theming is one of the radically new functionalities available on the LWUIT as compared to the `javax.microedition.lcdui` package and allows the use of application-specific look and feel that is also independent of the native platform. So, complete familiarity with theming is very important for effective application development using LWUIT.

11
Adding Animations and Transitions

Animations and Transitions are very important when it comes to creating captivating user interfaces. Animation involves repeated rendering at a frequency determined by the `Display` class. Transitions are used to determine the way in which a form or a component is moved out of display and a new one is brought in. The concept is very similar to that used in slide presentations to effect transition from one slide to another.

In this chapter, we will study these two features and see how to use them in actual applications. We also develop a custom transition, which demonstrates the process of such customization fully.

There are four demo applications in this chapter. The first demo is the "Hello" application from Chapter 2 that will be an aid to understanding how components can use animation.

The LWUIT has a number of interesting built-in transitions. The second demo—DemoTransition—will be used to illustrate the application of the transitions that are available in the library.

Transitions are not only applicable to forms and dialogs but to any other component as well. The third demo shows how transition works when applied to a label.

The basic philosophy underlying LWUIT is that a high degree of customization is supported. This allows the creation of our own transitions, and the BlindsTransitionDemo application will show us how to go about performing this task.

Animations

It was possible to animate images on the Java ME platform even before LWUIT appeared on the scene. There are a number of features in the `javax.microedition.lcdui.game` package that support animation that is particularly suitable for implementing games. This is the kind of animation that moves layers of images around the screen and also relative to each other. In addition, the JSR 226 API provides support for SVG-based animation. LWUIT handles animation a little differently providing support for a capability that is broad-based and simple to use. In the context of LWUIT, animation essentially allows objects to paint succeeding frames at timed intervals. Animation support in LWUIT also makes it easy for transitions to be implemented.

In order to be capable of being animated, an object has to implement the **Animation** interface which has two methods:

- `animate`—this is a callback method invoked once for every frame.
- `paint`—This method is called if `animate` returns `true`. If the object being animated is a component, then it will have a `paint` method and that is the one to be called as the signatures of the two methods are identical.

The `Component` class implements Animation. This makes every widget animation-capable. However, in order to actually carry out animation, an object must also be registered for animation by calling the `registerAnimated` method on its parent form. The `animate` method of a registered object is called once for every frame of animation as determined by the `Display` class. Only if `animate` returns `true`, then the `paint` method of the object is called, and a repaint takes place. In order to stop animation, the object must be deregistered by calling the `deregisterAnimated` method on the parent form.

In addition to the approach outlined above, there is another way to display animation using animated images. In Chapter 9, this method was used to show an animated icon on a label.

In order to see how an animation works by implementing the `Animation` interface, let us analyze `Hello` MIDlet—the application we have already come across in Chapter 2.

The Hello MIDlet

This demo, to quickly refresh our memories, draws an expanding circle, and as the animation comes to an end, writes "**H**" on a label. Then the expanding circle animation restarts, and "**He**" is written on the label after the largest circle is drawn. This goes on until the string "**Hello LWUIT!**" is written, whereupon the animation comes to a final stop. There is a **Replay** command for starting the cycle all over again. The next screenshot shows the animation just before the last character ('**!**') is written.

The demo has the following three classes:

- `HelloMIDlet`—this is the MIDlet that sets up the screen for display. It also starts, stops, and restarts the animation as required.
- `HelloForm`—this is a subclass of `Form` with a few additional functionalities. `HelloForm` has a built-in label for writing a string one character at a time. The trigger for writing each character has to come from an external source.
- `HelloLabel`—extends `Label` and controls the animation for drawing the expanding circle.

As we are mainly interested in the animation activity, let's see what the `HelloLabel` class does. The `animate` method in this class controls the animation by checking the time at every invocation, and if the elapsed time since the previous repaint exceeds the value of the `int` variable `interval` (set to 150 milliseconds in this example), then the time of repaint is updated. Also, the variable `state`, which determines the size of the circle to be drawn, is incremented. If `state` equals `5`, then all the circles have been drawn and the `done` flag is set to ensure that `animate` returns `false` the next time around. Further, the form that contains the `HelloLabel` instance is asked to update the string ("**Hello LWUIT**!") that is being displayed by invoking the `updateText` method. Note that `animate` returns `true` even though the value of the state may have reached `5`. This is done to erase the largest circle after the last character in the display string has been written. The code for this method is given below.

```
public boolean animate()
{
   if(!initialized)
   {
      return false;
   }
   long currentTime = System.currentTimeMillis();
   if (!done && (currentTime - prevTime> interval))
   {
      prevTime = currentTime;
      state++;
      if (state == 5)
      {
         done = true;
         ((HelloForm)getComponentForm()).updateText();
      }
      return true;
   }
   return false;
}
```

Whenever `animate` returns `true`, the `paint` method of `HelloLabel` is called. Within `paint`, a switch statement draws the circle as determined by the variable `state`. The following code paints the first (smallest) circle.

```
case 1:
   g.translate(getX(), getY());
   g.fillArc(x1, y1, 2*rad1, 2*rad1, 0, 360);
   g.translate(-getX(), -getY());
   break;
```

The point to note here is that the circle's coordinates specified in the call to the `draw` method of the `Graphics` class are relative to top-left corner of the `HelloLabel` object. In order to position the circles properly on the screen, the coordinates of the `graphics` object being used for rendering must be mapped into the coordinate system of the container. This is done by calling the `translate` method before drawing the circle. Also, whenever such coordinate translations are done, it is a good practice to restore original values before returning from the `paint` method. Accordingly, there is another call to `translate` after the circle is rendered. The code for drawing the other circles is very similar.

The other methods of `HelloLabel` perform necessary housekeeping tasks to support the animation. The `initialize` method gets the size of the label and accordingly calculates the radii of the second, third, and the fourth circles. The radius of the first circle is hardcoded to the value `10`. This method also sets `initialized` to `true` so that the `animate` method will not prematurely abort repaint.

```
public void initialize()
{
   width = getWidth();
   height = getHeight();

   int side = width < height? width : height;

   //find the center of the circle
   int centerX = width / 2;
   int centerY = height/2;

   //radius of largest circle
   rad4 = side/2 - 5;

   //difference between successive radii
   int radStep = (rad4 - rad1)/3;

   //radii of second and third circles
   rad2 = rad1 + radStep;
   rad3 = rad2 + radStep;

   //top left corners of the four bounding rectangles
   x1 = centerX - rad1;
   y1 = centerY - rad1;

   x2 = centerX - rad2;
   y2 = centerY - rad2;

   x3 = centerX - rad3;
   y3 = centerY - rad3;

   x4 = centerX - rad4;
   y4 = centerY - rad4;

   initialized = true;
}
```

The only other method in `HelloLabel` is `resumeAnimation`, which initializes the relevant variables so that animation may be restarted.

```
public void resumeAnimation()
{
   state = 0;
   done = false;
}
```

The class `HelloForm` also performs animation by writing a message one (non space) character at a time. However, this is not an independent action, and the `animate` method is not used here. The text update is triggered by the `HelloLabel` instance. The method that is called for this is `updateText`. Here a check is made to see if there is only one more character to be displayed. In that case, the whole message is displayed, and the `stopAnimation` method of `HelloMIDlet` is called to deregister `animLabel` (the `HelloLabel` instance) from receiving animation callbacks. If two or more characters remain to be displayed, then the `getUpdatedText` method is called to get the new string to be displayed.

```
public void updateText()
{
   if(index == helloString.length() - 2)
   {
      textLabel.setText(helloString);
      midlet.stopAnimation();
   }
   else
   {
      textLabel.setText(getUpdatedText());
      midlet.restartAnimation();
   }
}
```

This method, in turn, calls `getUpdatedText` to get the new string to be displayed. Within `getUpdatedText` the variable `index`—the pointer to the next character to be drawn—is incremented. As the method opens with this incrementation, the value to which `index` is initialized for starting the message display is `-1`. This ensures that `index` will point to the first character (`0`) when `getUpdatedText` is called for the first time. If after being incremented, `index` is pointing to a space character, then the method calls itself recursively until a non space character is found. Finally, it returns a substring of the message up to the character pointed to by `index`

```
private String getUpdatedText()
{
   index++;
   //if index points to space character
   //recurse until non-space character is found
   if(helloString.charAt(index) == ' ')
```

```
        {
            return getUpdatedText();
        }
        return helloString.substring(0, index+1);
    }
```

There is an implied assumption here that the message does not end in one or more spaces. In order to make sure that this assumption is not violated, the message passed to the constructor is trimmed. If the result is an empty string, then the default message is used for the display.

```
    private String helloString = "Hello!";//string to display
    .
    .
    .
    public HelloForm(HelloMIDlet m, String helloText)
    {
        .
        .
        helloText = helloText.trim();
        if(!(helloText.equals("")))
        {
            helloString = helloText;
        }
        .
        .
    }
```

The only other method in `HelloForm` is `resetIndex`, which clears the message from `textLabel` and reinitializes `index`, thus preparing the ground for resumption of animation.

```
    public void resetIndex()
    {
        index = -1;
        textLabel.setText("");
    }
```

Finally, let's deal with the `HelloMIDlet` class. Most of the activities of this class relate to setting up the form for display. The two methods related to animation are the `restartAnimation` and the `stopAnimation` methods.

```
public void restartAnimation()
{
   animLabel.resumeAnimation();
}
public void stopAnimation()
{
   helloForm.deregisterAnimated(animLabel);
   animationStopped = true;
}
```

The first method just calls into the `resumeAnimation` method of `animLabel`, which as we have already seen, does the required reinitialization for resumption of the animation cycle. The second stops animation callback to `animLabel` and sets the `animationStopped` flag to `true` so that the `Replay` command can be effective, as shown below in the code snippet from the `actionPerformed` method.

```
//'Replay' command
case 1:
   if(animationStopped)
   {
      animationStopped = false;
      helloForm.resetIndex();
      animLabel.initialize();
      restartAnimation();
      helloForm.registerAnimated(animLabel);
   }
```

The action here is simple—it calls all the required methods to reinitialize the two objects and start the animation all over again.

The `Hello` application shows how calls to animate can be used for implementing visual effects. The animation callback can also be used for other repetitive tasks. Let us look back at `TimeTeller`—the component we had created in Chapter 8. The timebase used in that component was implemented through a thread. Using the callback to `animate` could have been an alternate approach to achieve a similar result.

Transition

Transition is a cool feature of LWUIT that refers to the way a form is brought into (*animate in* transition) and removed from (*animate out* transition) display. The base class for implementing transitions is **Transition**, which implements the `Animation` interface. This is why transitions work basically like animations and utilize the same callback mechanism as any other animated component. However, in a subtle way, transition functions somewhat differently from animation for components in the sense that a component animates its own rendering, while transition controls the animated rendering of forms, dialogs, and components. There is also a difference between the ways in which component animations and transitions are used. It is not necessary to register the form concerned for animation to use transitions. When a form transition occurs, the `Display` instance directly places the form into the queue for receiving animation callbacks.

The Transition class

The `Transition` class is an abstract class that embodies the characteristics of the process of moving one form (or dialog or component) out of the display and bringing the next one in. The only constructor for this class is the default one. In any case, there is no way of directly instantiating this class, as it is an abstract one.

The use of transitions is made possible by two concrete subclasses. They are:

- CommonTransitions
- Transition3D

Let's examine these classes one at a time.

CommonTransitions

The `CommonTransitions` class currently supports the following two types of transition:

- **Slide**—new form slides in while pushing the current one out
- **Fade**—new form fades in while the current one fades out

`CommonTransitions` works with a supporting class—`Motion`—that provides the models for simulating physical motion. The three types of motions that the `Motion` class contains are:

- **Friction**—represents motion that is affected by friction
- **Spline**—represents motion with initial acceleration followed by deceleration
- **Linear**—represents uniform motion

It is possible to extend **Motion** to incorporate any other type of motion that may be required for a given application. Similarly, new types of transitions can be created by extending `Transition` or `CommonTransitions`. Later in this chapter, we shall build our own transition and its supporting motion. This exercise will show us the significant inner details of the `Transition` and `Motion` classes.

The constructors of `CommonTransitions` are private, and we have to use one of the factory methods to get an instance of this class.

Method	Parameters	Description
`public static createEmpty()`		Creates a transition that does nothing.
`public static createFade (int duration)`	`duration`—determines in milliseconds how long the transition will continue.	Creates a transition that lasts for the given duration and fades out the original form, while fading in the new one.
`public static createSlide(int type, boolean forward, int duration)`	`type`—determines the direction of motion. Can be either `SLIDE_HORIZONTAL` or `SLIDE_VERTICAL`. `forward`—for horizontal movement, if `forward` is `true`, then the new form will move from left to right. For vertical movement, if forward is `true`, then the new form will move from top to bottom. `duration`—determines in milliseconds how long the transition will continue.	Creates a transition that lasts for the given duration and slides the original form out, while sliding the new one in. The orientation of slide (vertical or horizontal) will be determined by `type` and the direction of movement by `forward`.

Method	Parameters	Description
`public static createSlide(int type, boolean forward, int duration, boolean drawDialogMenu)`	`type`—determines the direction of motion. Can be either `SLIDE_HORIZONTAL` or `SLIDE_VERTICAL`. `forward`—for horizontal movement, if forward is true, then the new form will move from left to right. For vertical movement, if forward is `true`, then the new form will move from top to bottom. `duration`—determines in milliseconds how long the transition will continue. `drawDialogMenu`—if `true`, then the soft button area will be displayed during transition. Relevant only for dialogs.	Creates a transition that lasts for the given duration and slides the original form out, while sliding the new one in. The orientation of slide (vertical or horizontal) will be determined by `type` and the direction of movement by `forward`. Whether the menu will remain on screen during transition is determined by `drawDialogMenu`. This is applicable to dialogs only.

Once an instance of the kind of `Transition` that we want is available, it can be installed for incoming or outgoing transition of a form. Installing the empty transition is the same as setting transition to null. Incidentally, although I have been talking about form in the context of transitions, it is equally applicable to dialog and component.

Transition3D

The `Transition3D` class works with **M3G API (JSR 184)** to provide transition animations with 3D effect. JSR 184 support is necessary for these transitions to work properly and so the `Motion` class is not required for `Transition3D`. The 3D suite of transitions is comprised of the following:

- **Cube**—simulates a rotating cube with the current form on the face that is rotating off the screen and the new one on the face that is rotating into the screen.
- **FlyIn**—the new form flies in.
- **Rotation**—a two dimensional version of Cube in which a plane, rather than a cube, rotates.

- **StaticRotation** — primarily meant for transitions involving a dialog. Only the dialog rotates while the form remains static.
- **SwingIn** — the incoming form swings into place either from top to bottom or from bottom to top.

Like `CommonTransitions`, `Transition3D` too has to be instantiated through one of the factory methods shown in the following table:

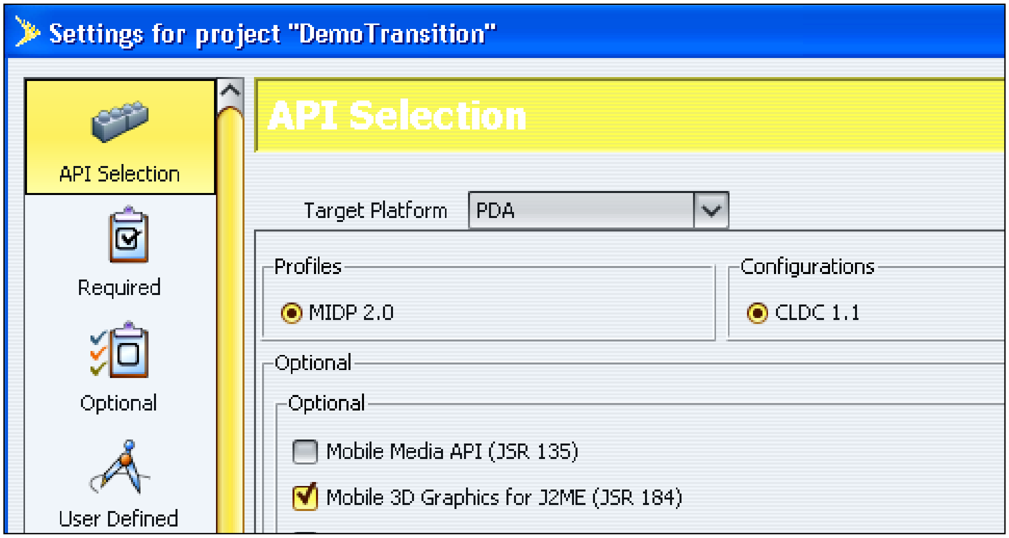
Method	Parameters	Description
`createCube(int duration, boolean rotateRight)`	`duration` — determines in milliseconds how long the transition will continue. `rotateRight` — if `true`, then the rotation will be towards right.	Creates a transition that lasts for the given duration and simulates a horizontally rotating cube with the original form on the face that is going off the screen and the new one on the face that is coming in. The direction of rotation is determined by `rotateRight`.
`createVerticalCube(int duration, boolean rotateDown)`	`duration` — determines in milliseconds how long the transition will continue. `rotateDown` — if `true`, then the rotation will be downwards.	Creates a transition that lasts for the given duration and simulates a vertically rotating cube with the original form on the face that is going off the screen and the new one on the face that is coming in. The direction of rotation is determined by `rotateDown`.
`createFlyIn(int duration)`	`duration` — determines in milliseconds how long the transition will continue.	Creates a transition that lasts for the given duration and simulates a form that flies in.
`createRotation(int duration, boolean rotateRight)`	`duration` — determines in milliseconds how long the transition will continue. `rotateRight` — if `true`, then the rotation will be towards right.	Creates a transition that lasts for the given duration and simulates a two dimensional version of Cube transition in which a plane, not a cube, rotates. The direction of rotation is determined by `rotateRight`.

Method	Parameters	Description
`createStaticRotation (int duration, boolean rotateRight)`	`duration`—determines in milliseconds how long the transition will continue. `rotateRight`—if `true`, then the rotation will be towards right.	Primarily meant for a dialog. Creates a transition that lasts for the given duration and simulates a two dimensional version of Cube transition in which a plane, not a cube, rotates. In this case, only the dialog rotates. The direction of rotation is determined by `rotateRight`.
`createSwingIn (int duration)`	`duration`—determines in milliseconds how long the transition will continue.	Creates a transition that lasts for the given duration and simulates a swinging form that comes in. The top edge appears as the hinge.
`createSwingIn (int duration, boolean topDown)`	`duration`—determines in milliseconds how long the transition will continue. `topDown` - if `true`, then the top edge will be 'hinged' and the bottom edge will swing in.	Creates a transition that lasts for the given duration and simulates a swinging form that comes in. The parameter `topDown` determines which edge will be hinged.

The `Transition3D` instance returned by a factory method can be installed to produce the desired transition effect. In the following section, we shall see how instances of `CommonTransitions` and `Transition3D` can be created and installed.

Using transitions

The DemoTransition application contains code for creating instances of all the different types of transitions we have seen above, and also for setting them to effect transitions on two forms. We have seen earlier that the 3D transitions need to be supported by JSR 184. So, before running this application we must confirm that this support is indeed available. In order to do this, click on the **Settings** tab on the SWTK and open the **API Selection** panel. Now make sure the box corresponding to **JSR 184** is selected, as shown in the following screenshot.



We are all set now to compile and run the application.

The DemoTransition application

All transitions are created in the same way by using a factory method regardless of the type. The following lines of code create two instances of horizontal slide transition, one moving from left to right and the other from right to left.

```
//horizontal slide left to right
Transition in = CommonTransitions.createSlide(
           CommonTransitions.SLIDE_HORIZONTAL, true, duration);
//horizontal slide right to left
Transition out = CommonTransitions.createSlide(
           CommonTransitions.SLIDE_HORIZONTAL, false, duration);
```

Once we have these instances, we can install them as required by using the `setTransitionInAnimator` method for incoming transitions of forms or the `setTransitionOutAnimator` method for outgoing transitions. The following code sets the transitions mentioned above as outgoing transitions for the two forms in this application.

```
demoForm1.setTransitionOutAnimator(in);
demoForm2.setTransitionOutAnimator(out);
```

The following screenshot shows `demoForm1` sliding out, which automatically causes `demoForm2` to slide in.

While setting transitions, care must be taken to ensure that conflicts are not caused. Consider the following statements.

```
demoForm1.setTransitionOutAnimator(in);
demoForm2.setTransitionInAnimator(out);
```

Here we have set a transition for the exit of `demoForm1` and a different one for the entry of `demoForm2`. As `demoForm2` enters just as `demoForm1` goes out, both transitions become effective simultaneously and this can produce an unintended result. You can try this out and see exactly what happens.

In order to avoid such contentions, one possible approach is to set either the *in* transition or the *out* transition for all forms but not both. This will make sure that there will not be any conflicting settings.

There is one exception to this rule though. When the incoming screen is a dialog the outgoing form does not go through the usual process of transition and the *out* transition for the form does not work. So, in such a situation, the *in* transition for the dialog also needs to be specified.

Transitions can be set in two ways. One approach, as shown earlier, sets a transition for individual forms. Alternately, we can set transitions for all forms at one go. The statement below shows how to do this:

```
UIManager.getInstance().getLookAndFeel().
        setDefaultFormTransitionOut(out);
```

Now all forms will use `out` as the outgoing transition. There is a set of methods in the `LookAndFeel` class that allow default transitions to be set for forms, dialogs, and menus. These are:

- `setDefaultFormTransitionIn(Transition defaultFormTransitionIn)`
- `setDefaultFormTransitionOut(Transition defaultFormTransitionOut)`
- `setDefaultDialogTransitionIn(Transition defaultDialogTransitionIn)`
- `setDefaultDialogTransitionOut(Transition defaultDialogTransitionOut)`
- `setDefaultMenuTransitionIn(Transition defaultMenuTransitionIn)`
- `setDefaultMenuTransitionOut(Transition defaultMenuTransitionOut)`

The methods mentioned above install default transitions for specified components. However, if we use the `setTransitionInAnimator` or the `setTransitionOutAnimator` method to install a different transition for a component, then this manually set transition will override the default one.

The other transitions can be instantiated and installed in the same way. A set of statements that create all the remaining types of transitions is listed below. Before using any of these, make sure that only one `in` and one `out` transition has been defined.

```
//vertical slide downwards
//Transition in = CommonTransitions.createSlide(
  CommonTransitions.SLIDE_VERTICAL, true, duration);
//vertical slide upwards
```

```
//Transition out = CommonTransitions.createSlide(
             CommonTransitions.SLIDE_VERTICAL, false, duration);
//fade
//Transition in = CommonTransitions.createFade(duration);
//Transition out = in;
/*make sure JSR 184 box is selected in project settings (SWTK)
for all 3D transitions to work*/
//cube left to right
//Transition in = Transition3D.createCube(duration, true);
//cube right to left
//Transition out = Transition3D.createCube(duration, false);
//vertical cube downwards
//Transition in = Transition3D.createVerticalCube(duration, true);
//vertical cube upwards
//Transition out = Transition3D.createVerticalCube(
                                  duration, false);
//swingin
//Transition in = Transition3D.createSwingIn(duration);
//Transition out = in;
//directional swingin hinged at top
//Transition in = Transition3D.createSwingIn(duration, true);
//directional swingin hinged at bottom
//Transition out = Transition3D.createSwingIn(duration, false);
//rotation right to left
//Transition in = Transition3D.createRotation(duration, true);
//rotation left to right
//Transition out = Transition3D.createRotation(duration, false);
//flyin
//Transition in = Transition3D.createFlyIn(duration);
//Transition out = in;
```

The only transition not shown here is the *static rotation*, as it is primarily meant for dialogs, and we are using forms here. However, it can also be tried out very easily using the same approach, as shown above.

Transition for components

Transitions are applicable not only to forms and dialogs but to other components as well. To use a transition for a component, the method to be used is `replace(Component current, Component next, Transition t)` of the `Container` class. This method replaces `current` with `next`. The component `current` must already be contained in the container on which the method is being invoked. This method returns without waiting for the transition to be completed. There is another method in the `Container` class that performs the same function but returns only after the transition is completed. This method is `replaceAndWait(Component current, Component next, Transition t)`.

In order to see component transition in action, we shall use the `DemoCompTrans` application. Within `DemoCompTrans` a label is added to a form in much the same way as in other examples. The form in this case has two commands—**Exit** and **Next**. When **Next** is executed, another label ("New Label") is used to replace the original label with a slide transition. The code within the `actionPerformed` method is as follows:

```
public void actionPerformed(ActionEvent ae)
{
   Command cmd = ae.getCommand();
   switch (cmd.getId())
   {
      //'Exit' command
      case 1:
         notifyDestroyed();
         break;
      //'Next' command
      case 2:
         //remove 'Next' command
           demoForm.removeCommand(demoForm.getCommand(0));
         //use slide transition to replace label
           demoForm.replace(currentLabel, nextLabel,
           CommonTransitions.createSlide(CommonTransitions.
           SLIDE_HORIZONTAL, true, 500));
   }
}
```

For the **Next** command, the command itself is removed before the label is replaced. When the commands were added to the form, **Exit** was added first followed by **Next**. The `addCommand` method of `Form` class puts all the commands in a vector with the new command being added at the top—that is, at index 0. So, `demoForm.getCommand(0)` returns the command that was added last. In this case, it is the `Next` command.

After the command is removed, `nextLabel` is used to replace `currentLabel` using an instance of slide transition. The following screenshot shows the transition in progress:

Authoring transitions

One of the great things about LWUIT, as we well know, is that it supports a good deal of customization. So, if you want a transition that is not available in the transition classes that come with the library, then you can easily write your own. In this section, we develop a transition to demonstrate the basic techniques for creating custom transitions.

Our transition is called **BlindsTransition**. As its name suggests, it makes the destination screen move into position through a series of gradually widening strips, much like closing blinds. The individual blinds are vertical and the closing progresses from left to right. While the transition is in progress, the menu bar remains on the screen and is affected by the transition. In order to keep the exercise simple, the class has been made non-configurable. Also, it does not cater to all conditions that may be encountered in real physical devices and in applications that use features like background painters. Notwithstanding these constraints, the example explains the issues that are fundamental to authoring transitions, based on the `CommonTransitions` philosophy.

In order to create a custom transition, we need to write a class that extends `Transition`. In this case, the class we use is **BlindsTransition**. We also need a model for the appropriate motion, and for our demo, that is the **StepMotion** class.

The BlindsTransition class

We start by looking at the `BlindsTransition` class. The `Transition` class has the following abstract methods, which have to be implemented by `BlindsTransition`:

- `public abstract Transition copy()`
- `public abstract boolean animate()`
- `public abstract void paint(Graphics g)`

In addition, there is the empty `initTransition` method that will have to be suitably implemented. Depending on the nature of the transition, this method may not be required at all. In that case, you may omit it altogether.

The `initTransition` method is a callback that is invoked at the beginning of each transition. So this is the place to initialize all parameters for starting the transition. We also instantiate a `StepMotion` object here, which will work out the successive widths of the blinds for rendering the destination screen (source screen remains stationary for this transition) for each frame.

The first task of `initTransition` is to initialize the variables required to set up the transition. One of the variables—`iBuffer`—is an image that is used for initial painting of a screen. In order to appreciate the need for `iBuffer`, we must be aware of a major difference between transition from one form to another and that from a form to a dialog or from a dialog to a form.

When a form-to-form transition occurs, both the outgoing object (source) as well as the incoming one (destination) are painted as per their visual attributes—positioned in accordance with the type of transition at work. In case of a slide transition, for example, a gradually increasing part of the screen will be occupied by the destination form and the rest by the source. But the visible parts of both forms will retain their original looks.

A transition from a form to a dialog or one from a dialog to a form works a little differently. We have seen in earlier demos that a dialog is displayed against a background which is the source in a tinted version, and when the dialog is *disposed*, the original appearance of the form is restored. In order to avoid transition between a form and its own tinted version, we adapt the following sequence for form-to-dialog transition:

- change instantaneously to the tinted look
- bring the dialog in under the installed instance of `Transition` while the tinted form remains static

Similarly, for the return transition, we do this:

- change instantaneously to the untinted look
- move the dialog out under the installed instance of `Transition`—again while the untinted form remains static

A second point to be kept in mind is that the overhead associated with the process of 'tinting' is pretty high, and it would be desirable do the tinting only once for a transition. This leads to the need for saving the tinted form for repeated use during transition without having to go through the tinting process over and over again.

In light of the issues discussed above, we can understand why the *out* transition set for form does not apply when the destination is a dialog. We can also appreciate the reason for having a static backdrop (especially the tinted one) against which the dialog transition can occur. The image `iBuffer` is meant to be this backdrop.

If the incoming object (destination) is a dialog, a tinted version of the source (the outgoing object) is painted into `iBuffer`. This image can now be used during transition, as the background on which the animation is rendered. So `initTransition` starts like this:

```
Component source = getSource();
Component destination = getDestination();
  blindWidth = 0;
  int w = source.getWidth();
  int h = source.getHeight();
  int startOffset = 0;
numOfStripes = w/stripeWidth;
if(w%stripeWidth != 0)
{
   numOfStripes++;
}


if (iBuffer == null)
{
   iBuffer = Image.createImage(w, h);
}
```

The next step is to create a `StepMotion` instance. This has to be done for every transition, as our `StepMotion` class does not support reinitialization.

```
motion = new StepMotion(startOffset, stripeWidth, duration, step);
```

The tinted version of source is then rendered into an image, as discussed above, using the graphics object obtained by calling `getGraphics` on `iBuffer`. Finally, the `StepMotion` instance is started.

```
        if(getDestination() instanceof Dialog)
        {
           doPaint(iBuffer.getGraphics(), getSource(), 0, 0, 0, 0);
        }
        motion.start();
```

Once the transition process starts, the `animate` method is called once for every frame. When `animate` for a transition returns `false`, it signals the end of the transition sequence, and it is no longer invoked until a new transition is started. Contrast this with classes that implement `Animation`. In order to stop the animation for such classes, it is necessary to call the `deregisterAnimated` method on the parent form. In our example, `animate` returns `false` if there is no motion object to implement the transition, or if transition has been completed as indicated by the `isFinished` method of the `StepMotion` instance, which returns `true` on completion. Otherwise `animate` gets the latest width of the blinds and returns `true` to show that transition should continue.

```
    public boolean animate()
    {
       if(motion == null || motion.isFinished())
       {
          //in either case terminate transition
          return false;
       }
       blindWidth = motion.getStep();
       return true;
    }
```

The `paint` method checks the value of `blindWidth`, which was returned by the `getStep` method of `StepMotion`. If `blindWidth` is equal to `-1`, then it means that the value has not been updated since the last call to `paint`, and the method returns as there is no point in repainting. Otherwise it calls the `paintBlinds` method, which does the actual work of rendering the latest state of the transition.

```
    public void paint(Graphics g)
    {
       //return if new position is not available
```

```
        if(blindWidth == -1)
        {
            return;
        }
        //otherwise paint
        paintBlinds(g);
    }
```

The `paintBlinds` method sets clip regions for actual painting to be done and calls the private `doPaint` method of this class. The first action is to get the source from which the transition is to occur. If the source does not exist, then this method returns without doing anything, as the form to be displayed next is the first one of the application and transition does not have to be performed. On the other hand, if there is a source, then the destination is also obtained and transition can proceed.

```
        Component source = getSource();
        if (source == null)
        {
           return;
        }
        Component dest = getDestination();
```

We have already discussed how dialog transition differs from transitions that involve only forms. If the source is a dialog, then the destination and the source are painted once. This painting is controlled by the Boolean variable `firstCycle`, which is initialized to `true` in the `initTransition` method. After this painting is done, `firstCycle` is set to `false` so that this step can be skipped in subsequent frames. We can create an interesting effect by not painting the destination, which will result in an *out* transition of the dialog and an apparently simultaneous *in* transition of the untinted destination over the tinted image.

```
        if(source instanceof Dialog)
        {
           if(firstCycle)
           {
              //comment out following statement
              //to get a different effect
              doPaint(g, dest, 0, 0, 0, 0);
              //paint the dialog
              doPaint(g, source, 0, 0, 0, 0);
              firstCycle = false;
           }
           .
           .
```

The real job of painting the progressively widening strips is divided into two parts. First, clips are set so that a series of strips can be rendered properly and the job of painting is then delegated to the private `doPaint` method of this class. The `setClip` method of `Graphics` class defines an overall region for painting. In this case, it is the total area of the source. The `clipRect` method then selects a part of that overall region for actual painting. Here the selected area corresponds to the height of the source and the width of the strip to be drawn. Before the `doPaint` method is called, the position of the strip is shifted by a value that equals the maximum width of the strip for every iteration of the loop shown as follows:

```
for(int i = 0; i < numOfStripes; i++)
{
   g.setClip(source.getAbsoluteX(), source.getAbsoluteY(),
      source.getWidth(), source.getHeight());
   g.clipRect((source.getAbsoluteX()+i*stripeWidth),
      source.getAbsoluteY(), blindWidth, source.getHeight());
   doPaint(g, dest, 0, 0, 0, 0);
}
return;
```

A similar approach is used for the transition when the destination is a dialog. The tinted source is painted once and the dialog is then rendered frame-by-frame as the transition proceeds. In the code snippet below, note that the clip settings are done in accordance with the dimensions of the dialog and also that the clipping parameters—`crx` and `crw`—are passed to `doPaint` so that the dialog may be painted properly.

```
if(dest instanceof Dialog)
{
   if(firstCycle)
   {
      g.drawImage(iBuffer, 0, 0);
      firstCycle = false;
   }
   //render transition over the tinted source
   for(int i = 0; i < numOfStripes; i++)
   {
      int crx = dest.getAbsoluteX()+i*stripeWidth;
      int crw = blindWidth;
      g.setClip(dest.getAbsoluteX(), dest.getAbsoluteY(),
                     dest.getWidth(), dest.getHeight());
      g.clipRect(crx, dest.getAbsoluteY(), blindWidth,
                                 dest.getHeight());
            doPaint(g, dest, 0, 0, crx, crw);
   }
   return;
}
```

If the transition is from one form into another, then there is no need for any initial painting, as the source does not have to be tinted, and it already exists on the screen. All that needs to be done is set the clip and call the `doPaint` method.

```
for(int i = 0; i < numOfStripes; i++)
{
   g.setClip(dest.getAbsoluteX(), dest.getAbsoluteY(),
           source.getWidth(), source.getHeight());
   g.clipRect((dest.getAbsoluteX()+i*stripeWidth),
           dest.getAbsoluteY(), blindWidth, source.getHeight());
   doPaint(g, dest, 0, 0, 0, 0);
}
```

The `doPaint` method paints the transition on the basis of the clip settings done within `paintBlinds`. When a dialog transition has to be painted, the content pane, the title bar, and the menu bar are painted individually, and the clip settings are reinitialized before painting the last two components. In all cases, the `paintComponent` method of the `Component` class is called into for the low level actions associated with painting.

```
private void doPaint(Graphics g, Component cmp, int x, int y,
      int crx, int crw)
    {
   int cx = g.getClipX();
        int cy = g.getClipY();
        int cw = g.getClipWidth();
        int ch = g.getClipHeight();
   if(cmp instanceof Dialog)
   {
      Dialog dialog = (Dialog)cmp;
      dialog.getContentPane().paintComponent(g, false);
      if(dialog.getCommandCount() > 0)
      {
         Component menuBar = dialog.getSoftButton(0).getParent();
         g.setClip(0, 0, cmp.getWidth(), cmp.getHeight());
         if(getDestination() instanceof Dialog)
         {
            g.clipRect(crx, cy, crw, ch);
         }
         menuBar.paintComponent(g, false);
      }
      if(getDestination() instanceof Dialog)
      {
```

```
            g.setClip(0, 0, cmp.getWidth(), cmp.getHeight());
            g.clipRect(crx, cy, crw, ch);
         }
         dialog.getTitleComponent().paintComponent(g, false);
         g.setClip(cx, cy, cw, ch);
         return;
      }
      .
      .
      .
```

When a form has to be painted either by itself or as a background for a dialog, we make sure that the screen and form coordinates are properly aligned and call the `paintComponent` method.

```
g.translate(x, y);
cmp.paintComponent(g, false);
```

The `BlindsTransition` class has two more methods—`cleanup` and `copy`. The first method, called when a transition is completed, provides optional support for releasing resources and for cleaning up any garbage left behind by a transition. The second method returns a copy of the current transition object required for the `Display` class to work with.

```
//support for grabage clean up
public void cleanup()
{
   super.cleanup();
   iBuffer = null;
}

//return a functionally equivalent transition object
public Transition copy()
{
   return new BlindsTransition(duration, step, stripeWidth);
}
```

The StepMotion class

`StepMotion` is the class that represents a progressively widening strip. The constructor takes four integers as parameters:

- `sourceValue`—this is starting value of width. As the new screen starts moving in from the left edge, this will be zero.
- `destinationValue`—this is the width corresponding to the finishing position.

- `duration`—the duration (in milliseconds) of the transition.
- `steps`—the number of steps to be taken to complete the transition.

The constructor calculates the value by which the position of the destination screen has to be incremented from one step to the next. It also calculates the minimum time that has to elapse before the screen can be refreshed.

```
public StepMotion(int sourceValue, int destinationValue, int duration,
int steps)
   {
      this.destinationValue = destinationValue;
      this.duration = duration;
      this.steps = steps;

      //the size of a step
      stepSize = (destinationValue - sourceValue)/steps;

      //the time interval between two successive steps
      interval = duration/steps;
   }
```

The `isFinished` method checks if the current step number is greater than the number of steps the transition should take and the destination has been reached. If this is the case, then the transition has been done, and the method returns `true`. As we have already seen, the `animate` method of `BlindsTransition` terminates the transition when `isFinished` returns `true`.

```
public boolean isFinished()
{
   return stepNumber > steps && offset == destinationValue;
}
```

The third method is `getStep`, which increments `stepNumber` and returns the updated value of `offset` (representing the latest width) if the required time interval since the last update has expired. Otherwise it returns `-1` so that `animate` in `BlindsTransition` does not call for a repaint. The only point to note here is that calculated step size may not be a factor of the total distance (`destinationValue` - `sourceValue`) to be covered, as we are using integer math. In such a case, the `getStep` method will ensure that the missing distance is made up, as it always continues to be called until the required width has been covered. This is because `isFinished` returns `true` only when the number of steps taken (`stepNumber`) exceeds the required number of steps and the destination is reached. These extra steps do not really cause any problem, as the value of `offset` is not allowed to exceed `destinationValue`.

```
public int getStep()
{
   if(System.currentTimeMillis() >= startTime +
```

```
                                    interval*(stepNumber+1))
            {
               stepNumber++;
               offset = stepNumber*stepSize;
               if(offset > destinationValue)
               {
                  offset = destinationValue;
               }
               return offset;
            }
            else
            {
               return -1;
            }

        }
```

The MIDlet

The `BlindsTransitionDemo` MIDlet is very simple. It loads the theme file and goes on to set up two forms. A `BlindsTransition` instance is then created and installed.

```
Transition out = new BlindsTransition(duration, numOfSteps,
               Display.getInstance().getDisplayWidth()/10);
demoForm1.setTransitionOutAnimator(out);
demoForm2.setTransitionOutAnimator(out);
```

The following screenshot shows the `BlindsTransition` in progress:

You can modify the MIDlet such that the transition occurs between a form and a dialog. The resulting transition would look like this:

Summary

This chapter has shown us quite a few things. We have seen how to animate objects by implementing the `Animation` interface. We have also seen how to use the transitions that come with the LWUIT library—for forms, dialogs, and menus as well as for widgets like labels. We have also authored our own transition.

The intricacies of designing transitions that work with dialogs have been dealt with in some detail. There are some more issues that may need to be addressed while building custom transitions. The source code of LWUIT transitions provides extensive insight into these factors, especially from the perspective of handling graphics related issues and is an invaluable resource for a developer.

12
Painters

All LWUIT components have a multi-layered structure. The first layer erases a visually obsolete widget, and the subsequent layers then paint the background followed by the constituent parts of the new version. As a matter of fact, the background too can be made up of several layers, and that is not all. After a form has been fully rendered, we can place a layer above it that can be drawn upon regardless of any changes or animations that may be taking place in the form below. Such a layer—known as a **GlassPane**—is usually transparent or translucent so that the form under it remains visible.

The classes that work as a background painter or a glass pane must implement the **Painter** interface. In case more than one background painter is used, they can be formed into a chain through the **PainterChain** class so that the background can be rendered layer-by-layer. Similarly, a glass pane also can have many layers.

In this chapter, we shall familiarize ourselves with the `Painter` interface and the **PainterChain** class. We shall also learn, with the help of examples, how background painters and glass panes can be used.

The Painter interface

`Painter` defines the fundamental interface for all objects that are meant to draw backgrounds or to render on a glass pane. This interface declares only one method—`public void paint(Graphics g, Rectangle rect)`—for drawing inside the bounding rectangle (specified by `rect`) of a component. The library provides a class that implements `Painter` and is used as a default background painter for widgets and containers. This is the **BackgroundPainter** class that has (you guessed it) just the one method `paint`, which either paints the background image if one has been assigned or fills in the bounding rectangle of the component with the color set in its style.

When we want to paint a background ourselves, we can write our own class that implements `Painter`, and set it as the background painter for the relevant component. The `DemoPainter` MIDlet, discussed in the next section, shows how this is done.

The DemoPainter application

This application creates a combo box and uses a theme to set the style for the various elements that are displayed. When the application is compiled without setting a custom background painter, the combo box looks as shown in the following screenshot:

The MIDlet code has the following statement commented out in the MIDlet. When uncommented, this statement sets an instance of `ComboBgPainter` as the background painter for the combo box.

```
combobox.getStyle().setBgPainter(new ComboBgPainter(0x4b338c));
```

The recompiled application produces the following display showing the new background color:

The class responsible for drawing the background is `ComboBgPainter`, which implements `Painter`. The constructor for this class takes the color to be used for background painting as its only parameter. The `paint` method determines the coordinates of the top-left corner of the rectangle to be painted and its dimensions. The rectangle is then filled using the color that was set through the constructor.

```
class ComboBgPainter implements Painter
{
   private int bgcolor;
   public ComboBgPainter(int bgcolor)
   {
      this.bgcolor = bgcolor;
   }
   public void paint(Graphics g, Rectangle rect)
   {
      g.setColor(bgcolor);
      int x = rect.getX();
      int y = rect.getY();
      int wd = rect.getSize().getWidth();
      int ht = rect.getSize().getHeight();
      g.fillRect(x, y, wd, ht);
   }
}
```

Drawing a multi-layered background

In actual practice, there is hardly any point in using a custom painter just to paint a background color, because the `setBgColor` method of `Style` will usually do the job. Themes too can be used for setting background colors. However, painters are very useful when intricate background patterns need to be drawn, and especially if multiple layers are involved. `PainterChain`, described in the next section, is a class designed for handling such requirements.

The PainterChain class

It is possible to use more than one painter to render different layers of a background. Such a set of painters can be chained together through the `PainterChain` class. The only constructor of this class has the form `public PainterChain(Painter[] chain)` where the parameter `chain` is an array of painters. The contents of `chain` will be called sequentially during the painting of a background, starting from the element at index `0` to the last one.

There are two methods of the `PainterChain` class that provide support for adding painters to the array underlying the chain. A new painter can be added either to the top (the `prependPainter` method) or at the end (the `addPainter` method) of the array. The array itself can be accessed through the `getChain` method.

`PainterChain` implements `Painter` so that the `setBgPainter` method can be used to set a `PainterChain` as well as a lone painter, which means the `paint` method also is present here. The function of `paint` in `PainterChain` is to call the `paint` methods of the painter array elements one by one starting at index `0`.

The `DemoPainterChain` application that comes up next shows how a chain of painters can be used to draw the multiple layers of a background.

The DemoPainterChain application

The `DemoPainterChain` example uses `alphaList` (the list for `DemoList` MIDlet in Chapter 5) to show a painter chain in action. After organizing the form and the list, we set up a painter array to hold the three painters that we shall deploy.

```
Painter[] bgPainters = new Painter[3];
```

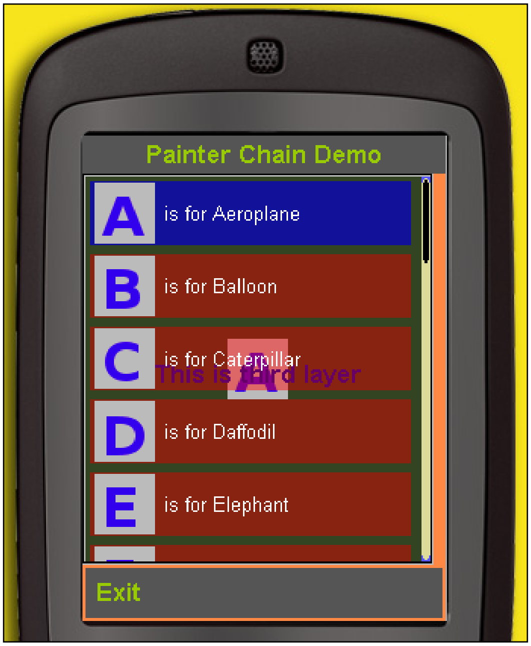
Once we have the array, we create three painters and load them into the array. The first (lowest) painter, which will fill the bounding rectangle for the list with a designated color, goes in at index `0`. The next (middle) layer, at index `1`, will draw an image at the center of the list. Finally, the topmost layer for writing a text a little below the center line of the list is inserted at index `2`.

```
bgPainters[0] = new Eraser(0x334026);

try
{
   bgPainters[1] = new ImagePainter(Image.createImage(
                                                "/a.png"));
}
catch(java.io.IOException ioe)
{
}

bgPainters[2] = new TextPainter("This is third layer");
```

Now we are ready to instantiate a `PainterChain` object, and install it as a background painter for the list.

```
PainterChain bgChain = new PainterChain(bgPainters);
alphaList.getStyle().setBgPainter(bgChain);
```

The list itself will be drawn on top of these three layers, and the background layers will be visible only because the list is translucent as determined by the transparency value `100`, set by the `AlphaListRenderer` instance used to render `alphaList`. The list now looks as shown in the following screenshot:



A close inspection of the screenshot that we have just seen will show that the layers have indeed been drawn in the same sequence as we had intended.

The three painters are very similar in structure to the `ComboBgPainter` class we came across in the previous example. The `Eraser` class here is virtually identical to `ComboBgPainter`. The other two classes work in the same way, except for the fact that `TextPainter` draws a line of text, while `ImagePainter` draws an image.

```
class TextPainter implements Painter
{
   private String text;
   TextPainter(String text)
   {
      //set the text to be written
      this.text = text;
   }
   public void paint(Graphics g, Rectangle rect)
   {
      //get the dimension
      //of background
      int wd = rect.getSize().getWidth();
      int ht = rect.getSize().getHeight();
      //create and set font for text
      Font textFont = Font.createSystemFont(
              Font.FACE_PROPORTIONAL,Font.STYLE_BOLD,Font.SIZE_LARGE);
      g.setFont(textFont);
      //set text color
      g.setColor(0x0000aa);
      //position text slightly below centerline
      int textX = wd/2 - textFont.stringWidth(text)/2;
      int textY = ht/2 - textFont.getHeight()/2 + 3;
      //write text
      g.drawString(text, textX, textY);
        }
}

class ImagePainter implements Painter
{
   private Image bImage;
   ImagePainter(Image bImage)
   {
      //set the image to be drawn
      this.bImage = bImage;
   }
   public void paint(Graphics g, Rectangle rect)
   {
      //get the dimensions
      //of background
```

```
        int wd = rect.getSize().getWidth();
        int ht = rect.getSize().getHeight();
        //position image at center
        int imageX = wd/2 - bImage.getWidth()/2;
        int imageY = ht/2 - bImage.getHeight()/2;
        //draw image
        g.drawImage(bImage, imageX, imageY);
    }
}
```

When an image is used on the background of a form, we have seen that it is scaled to occupy the entire form real estate. But if the same image is used as an icon for a label, then it is drawn in its actual size. This task of scaling the image for backgrounds is taken care of by `BackgroundPainter`, which is used as the default `bgPainter`.

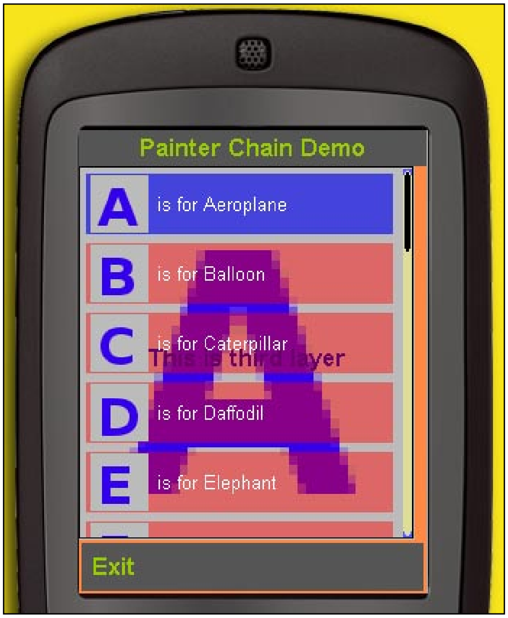
The `scaleImage` attribute of `Style` determines whether the background image of a component should be *scaled* (`scaleImage` == `true`) or *tiled* (`scaleImage` == `false`), and its default value is `true`. When required, `scaleImage` can be set to any value by calling the `setScaleImage` method of `Style`. Before drawing the background image, the default background painter calls the `isScaleImage` method of `Style`. If the returned value is `true` and the dimensions of the background image are not the same as those of the background (specified by the parameter `rect` of `paint` method), then the image is scaled to the same size as that of the rectangle to be drawn into. This image is then set as the background image so that the scaling does not have to be done over and over again.

In our case, we need not check the `scaleImage` attribute, as we have already decided that scaling is required. So all we need to do is compare the dimensions of the image with those of the background we are drawing on and scale it if required. Then we save the scaled version so that it will not be necessary to scale it the next time this method is called. In order to see the effect of scaling, we replace the `paint` method of the `ImagePainter` class with the following version. The highlighted code does the scaling.

```
public void paint(Graphics g, Rectangle rect)
{
    //get the dimensions and position
    //of background
    int imageX = rect.getX();
    int imageY = rect.getY();
    int wd = rect.getSize().getWidth();
    int ht = rect.getSize().getHeight();
    //check if image dimension different
    //from component background dimension
    if (bImage.getWidth() != wd || bImage.getHeight() != ht)
```

```
        {
            //scale image and save
            bImage = bImage.scaled(wd, ht);
        }

        //draw image
        g.drawImage(bImage, imageX, imageY);
    }
```

The new `paint` method creates the following list. Note the change in background color, which is now the same as that of the image, as it has been scaled up to cover the original background.



Using a glass pane

A glass pane is indeed like a fully transparent and distortion free glass sheet placed over a form. We can draw whatever we want on the pane, and only the part of the form under that pattern will be obscured. The rest of the form will be visible. Also, the pattern drawn on the glass pane will not be affected by any transition, animation, or change of any kind that may take place on the form below.

In the world of LWUIT, a glass pane is also a painter. However, unlike the painters we have used so far, a glass pane can only be used with a form. Let us see, with the help of the `DemoGlassPane` example, how to install a glass pane.

The DemoGlassPane application

For this application as well, the basic building block is `alphaList` on which we shall place a glass pane with text written on it. The action required is very simple as the following snippet shows:

```
try
{
   demoForm.setGlassPane(new ImagePainter(Image.createImage(
                                          "/text.png")));
}
catch(java.io.IOException ioe)
{
}
```

We just created an instance of our old friend `ImagePainter` and installed it as a glass pane on `demoForm`. The statement invoked for installing the glass pane is one of the two that can be used. We could have used the static method of `PrinterChain` shown below to get the same result.

```
PainterChain.installGlassPane(demoForm, new ImagePainter(Image.
  createImage("/text.png")));
```

The glass pane that is created by the previous code is shown in the next screenshot. As expected, we find that only the portion of `alphaList` directly below the text is obscured, but the rest of the list is clearly visible.

The fact that we have used `ImagePainter` to render text does not mean that it is mandatory for a glass pane to incorporate an image, as we could also have used a `TextPainter` object. It is just that writing diagonally positioned text is very easy when an image is used. In general, any valid painting activity can be used for glass panes.

In order to illustrate the ease with which image orientation can be changed, let us write the same string as in the previous screenshot but rotated by 90°. In order to do this, replace the statement within the `try` block in the code shown earlier for installing the glass pane with the one shown below:

```
demoForm.setGlassPane(new ImagePainter(Image.createImage(
                                   "/text.png").rotate(90)));
```

The following screenshot shows the new **Glass Pane**:

The `rotate` method used above assumes that the image to be rotated is a square one. A word of caution here, rotating images through angles that are not multiples of 90° is rather inefficient and should be avoided as far as possible. Also, such angles may not be supported on all platforms.

A GlassPane with multiple layers

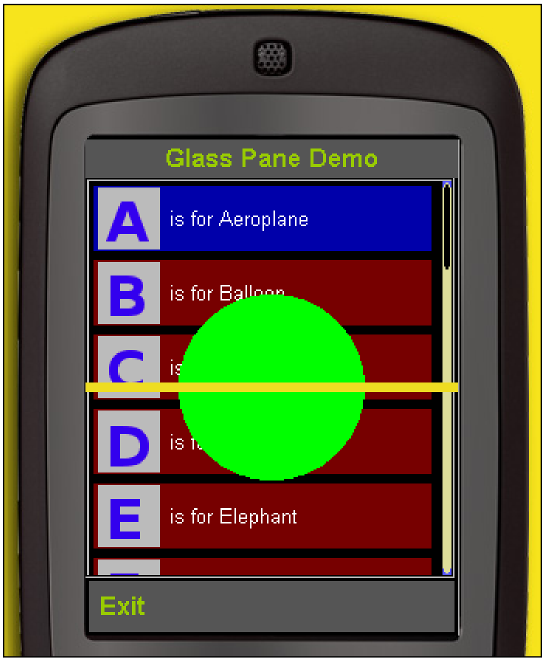
A glass pane, like a background, can have as many layers as we want. Our next example, `DemoLayeredGP`, has two layers. The first layer draws a circle at the center of the pane, and the second draws a six pixel wide band right across its middle. Obviously, the glass pane that we want has to be a painter chain, and we create this chain exactly as we had created the chain for background painters.

```
Painter[] glassPanes = new Painter[2];
glassPanes[0] = new CirclePainter();
glassPanes[1] = new BandPainter();
PainterChain gpChain = new PainterChain(glassPanes);
```

Then the painter chain is installed using the same approach as the one for a single layer glass pane in the previous example.

```
//install glass pane
//using either of the following statements

//PainterChain.installGlassPane(demoForm, gpChain);
demoForm.setGlassPane(gpChain);
```

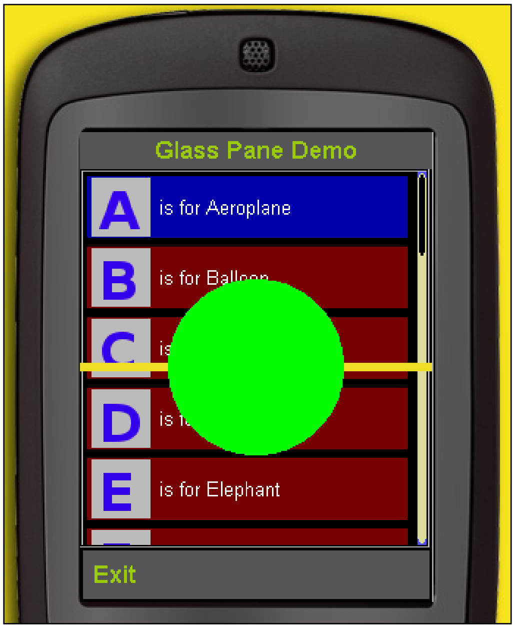
The glass pane will now look like the following screenshot with the band over the circle. Observe that both these figures remain stationary even if the list beneath is scrolled.



We can reverse the order of the two panes simply by changing the respective indices in the painter array.

```
glassPanes[1] = new CirclePainter();
glassPanes[0] = new BandPainter();
```

The circle will now be drawn over the band, as shown in the following screenshot:



Summary

Painters are essentially layers on which we can paint. Such a layer can be placed under the visual elements of a component forming its background. It can also be placed above a form as a normally transparent pane through which the form can be seen. Anything that is rendered on such a pane remains unaffected by changes (such as animations) in the form below.

Painters are convenient tools, for example, when we want custom patterns to be used as backgrounds. This is especially true when we need to support different combinations of such patterns in a range of components. Such variability can be easily programmed by arranging the constituents in layer sets that can be formed into `PainterChains`. A chain can then be selected for use, depending on the required combination.

There are many instances when a message needs to be superimposed on a UI. Consider an email client, which shows a message overlay that first says "Sending mail..." when the `Send` command is executed and then changes to "Your mail was successfully sent" upon completion of the activity. Glass panes are very useful for implementing such effects. As in the case of background painters, chains of glass panes can be used to form varied combinations of a set of texts, images, or rendered elements.

In this chapter, we have studied painters and their applications quite extensively. The four examples have demonstrated the use of both background painters and glass panes—single layered as well as chained.

13
Effects and Logging—Useful Utilities

Effects is a class in the `com.sun.lwuit.util` package intended to implement visual effects. The current implementation has just one effect, which appends a reflection of an image just below the original one. In this chapter, we shall learn about this class and go through an example that shows its use.

Logging has been around for a long time in the world of Java. We have had several logging software products like Ceki Gülcü's **log4j** (now an Apache Software Foundation project). We also have Sun's own logging API for JDK 1.4 as well as **Lumberjack** for JDKs 1.2 and 1.3. Through the **Log** class, LWUIT provides an easy to use and pluggable logging framework for Java ME applications.

Here we are going to gain considerable insight not only into the use of `Log`, but also into its structure through three sample MIDlets. The first of these will use the `Log` class to illustrate its application. The second will develop a subclass giving details of what happens under its hood. Finally, the third example will demonstrate the concept of installing a subclass.

Using Effects

The `Effects` class is intended to implement visual effects that enhance the appearance of a widget. Currently this class supports one such effect that simulates a reflected image placed below the original. The resulting combination looks as if the original image is being reflected in still water.

The Effects class

`Effects` is a static class that has two methods to return the image received as a parameter with the reflection appended.

Method	Parameters	Description
`public static Image reflectionImage (Image source)`	`source`—the image to which the reflection effect is to be added.	Returns the original image with its reflection appended to it. The height of the reflection will be half that of the source image. The transparency of the image will start from `120`.
`public static Image reflectionImage (Image source, float mirrorRatio, int alphaRatio)`	`source`—the image to which the reflection effect is to be added. `mirrorRatio`—the ratio of the height of the reflected image to that of the source image. Generally less than `1`. A `mirrorRatio` of `0.5f` will generate a reflection that is half the height of the source. `alphaRatio`—the value of transparency at the starting point of reflection.	Returns the original image with its reflection appended to it. The height of the reflection will be determined by `mirrorRatio`. The transparency of the image will start from `alphaRatio`.

Our next section shows a demo application for `Effects`.

The DemoEffects application

This application creates an image and then calls the `Label` constructor with the image returned by one of the static methods of the `Effects` class.

```
//create label with reflected image
//use either statement -- one at a time
Label effectLabel = new Label(Effects.reflectionImage(
                                        sourceImage));
//Label effectLabel = new Label(Effects.reflectionImage(
                                sourceImage, 0.7f, 80));
```

When the first statement is used, the resulting reflection is half the height of `sourceImage`, and the transparency of the image at the starting point is `120`. This image is shown in the following screenshot:

If we now comment out the first statement and uncomment the second, then we shall get a reflection that has a greater height than the one above. Also, the starting edge of the reflected image will be somewhat lighter than what we saw in the previous image, as `alphaRatio` is now 80. The following image is what we get with the second statement:

The `reflectionImage` methods need careful handling with respect to background color and `alphaRatio`. You may have to go through a few iterations to arrive at an attractive rendition. The API documentation recommends an `alphaRatio` in the range of `90` to `128` but sometimes, as in the case of this example, even `80` works quite well.

Logging with LWUIT

The `Log` class provides a framework for recording information at runtime. By default, the specified information is logged using the **Record Management System** (**RMS**). If the device supports the **File Connection API** defined by JSR 75, then the file system is used for logging. The recorded messages can be retrieved and displayed to provide insight into the behavior of a program.

There are four levels at which logging can be done: **DEBUG**, **INFO**, **WARNING,** and **ERROR**. The lowest (and default) level is *debug*, and *error* is the highest. The basic function of this stratification is to establish a threshold so that only messages allocated to a level equal to or higher than this threshold are logged. If the level for logging has been set at *debug*, all messages will be logged. Similarly, with the logging level set at *warning*, only warning and error messages will be logged. The logic for classifying information into a hierarchy of levels has to be determined by the programmer, and there is nothing in the framework to decide what kind of information should fall into each category.

The debug level, as its name suggests, would normally be used to record information required to debug code. Let us take a hypothetical method that uses five variables that go through a series of computations after initialization. Finally, a division by zero occurs, but it proves difficult to figure out how the divisor is being set to zero. The situation may be something like the following:

```
int a = ....;
int b = ....;
int c = ....;
int d = ....;
int e = ....;
.
.
a = (c - d) * e;
.
.
b = e % c;
.
.
if(b > 11)
{
   b = 11;
}
.
.
int f = ....;
.
```

```
.
b = Math.max(b, f);
.
.
d = a/b;
```

Logging can be used here to see at which statement `b` is becoming equal to zero, assuming that `c` is known to be non zero. We assign step numbers to each statement that changes the value of `b` and log the value of `b` after the statement is executed.

```
int a = ....;
int b = ....;
int c = ....;
int d = ....;
int e = ....;
.
.
a = (c - d) * e;
.
.
b = e % c;//step 1
//log: ("value after step 1 is " + b)
.
.
//step 2
if(b > 11)
{
   b = 11;
}
//log: ("value after step 2 is " + b)
.
.
int f = ....;
.
.
b = Math.max(b, f);//step 3
//log: ("value after step 3 is " + b)
.
.
try
{
   d = a/b;
}
catch(Exception e)
{
   //log error message with log level specified as Error
   //show the log if level is debug
}
```

When this code is run, the division by zero will cause the log to be displayed, and the problem area can then be identified. Once debugging is done, we can set the logging level to *error* so that a message will be logged only if an exception is thrown again in future. Note that the messages at steps `1`, `2`, and `3` need not specify the logging level, as they belong to the default (debug) level.

Messages can also be recorded to provide glimpses into other aspects of program operation. Warning messages can be generated, for instance, when an activity tries to access sensitive or potentially risky resources. A log printout can show that program behavior needs modification to avoid such actions. Similarly, logging at info level can be used perhaps to record a history of network activity such as the URLs connected to or the size of the data downloaded from a specific site.

We shall see how the `Log` class is used by analyzing an example. However, before that, here is an introduction to the class itself.

The Log class

The functionalities of this class are exposed through a set of static methods. These methods can be classified into three groups:

- Methods to access logging level: A `getter` and a `setter`.
- Methods for writing into the log file: There are two such methods—one for logging at the default level and the other for logging at the level passed as a parameter.
- Methods to retrieve the log file: Again there are two methods to perform this task—one of these methods gets the contents as a string and lets the application display it as desired, while the other displays a form with a text area showing the logged information.

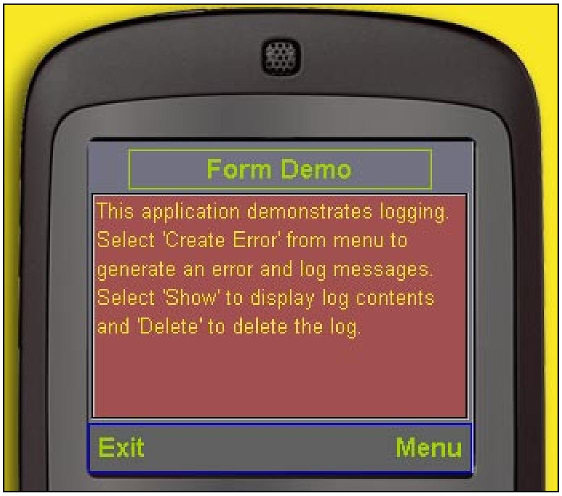
There is also a static method that allows us to install a subclass instance so that logging can be done into a different file or in accordance with a different algorithm.

In addition to the above, there are two protected methods that enable subclasses to customize logging behavior.

In the following section, `DemoLogger` shows us how the `Log` class can be used in an application.

The DemoLogger application

The opening screen of DemoLogger tells us how this application works.



This application logs messages at three levels—*debug*, *warning*, and *error*—within a method that forces a division by zero. The method that generates logs and the error is `makeError`, called when the **Create Error** command is executed . The following code snippet shows that four logging statements have been used. Two of these statements log at the *debug* level, one logs at the *warning* level, and the fourth logs at the *error* level.

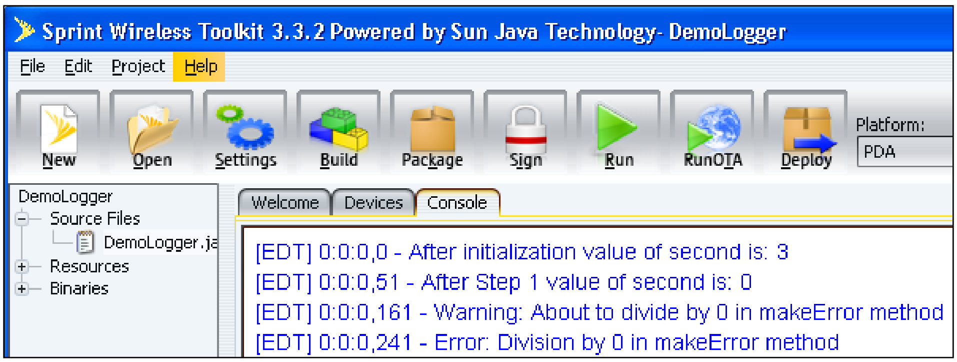
```
private void makeError()
{
   int first = 6;
   int second = 3;

   //logs using default level
   Log.p("After initialization value of second is: " +
                                     second + "\r\n");

   //Log.setLevel(Log.WARNING);
   //Log.setLevel(Log.ERROR);

   //Step 1
   second -= first/2;

   //logs using default level
   Log.p("After Step 1 value of second is: " + second + "\r\n");

   if(second == 0)
   {
      //logs using WARNING level
```

```
            Log.p("Warning: About to divide by 0 in makeError method\r\
                                                         n", Log.WARNING);
         }
         try
         {
            int c = first/second;
         }
         catch(ArithmeticException ae)
         {
            //logs using ERROR level
            Log.p("Error: Division by 0 in makeError method\r\n",
                                                         Log.ERROR);
         }
      }
```

Initially, the logging level is not explicitly set, which means that the default level remains effective. When the `makeError` method is called, two `int` variables are initialized, and the value of variable `second` is logged using the static method `Log.p`. Then, at *step 1*, `second` is modified, and the new value is again logged. Both of these values are logged at the default level. Next, an `if` statement checks `second` and logs a *warning* message if the value is zero. Finally, a message is logged at the *error* level to record the error.

The menu, as we can infer, has three commands in addition to **Exit**. The next screenshot shows the menu with the cursor on **Create Error**:

The `Log.p` method not only records the messages, but it also prints them on the console. The messages are shown below. We see that all four messages have been printed as the logging level was the default one. Before logging a message, `Log.p` appends the name of the thread, which generated the message and the time since the initiation of the MIDlet. The format of the time stamp is **hours:minutes: seconds, milliseconds**.



Executing the **Show** command invokes the `showLog` method of the MIDlet, which in turn, calls into the `Log.showLog` method. It also prints the complete log data obtained by calling the `Log.getLogContent` method, which returns the contents of the log as a string.

```
private void showLog()
{
    Log.showLog();
    System.out.println(Log.getLogContent());
}
```

The `Log.showLog` method uses the `Log.getLogContent` method to retrieve logged information and sets it as the text for a text area. This text area is then added to a form, and the form is displayed.

There are two statements that have been commented out in the code. These are:

```
//Log.setLevel(Log.WARNING);
//Log.setLevel(Log.ERROR);
```

If one of these statements is uncommented, then four error messages will not be logged. In order to generate the following screenshot, the second statement was activated.

The first logging statement was accepted because `level` was set to *error* later. Once `level` was set at the new value, only the message in the `catch` block was logged. If the **Create Error** command is executed once more, then only the last of the four messages will be logged, as `level` has already been set to *error*.

The `Log` class does not have any provision for deleting the log when a record store is used. However, for a meaningful debugging exercise, we should be able to distinguish one run from another, and this would be difficult to do from a log dump, as the time stamp is not absolute but relative to the launch of the MIDlet. Also, small devices usually have memory restrictions, and this strengthens the case for not allowing logs to grow indefinitely. Therefore, we need to make our own arrangement for deleting the log so that we can study the result of one run at a time. This is done in the `deleteLog` method of `DemoLogger`, which is called when the **Delete** command is executed. If the delete attempt fails, then a message is logged.

```
private void deleteLog()
{
   try
   {
      RecordStore.deleteRecordStore("log");
   }
   catch(Exception e)
   {
      Log.p("Error: Attempt to delete log record failed\r\n",
                                               Log.ERROR);
   }
}
```

The `Log.p` and the `Log.showLog` methods use the name "log" for the record store to be used for logging into. Both these methods call the static method `RecordStore.openRecordStore` with a `true` second parameter to make sure that a new record store is created if one with specified name cannot be found. That is why we can delete the entire record without any fear of a `RecordStoreException` being thrown.

> Note that for `DemoLogger`, we assume that the RMS will be used for logging. The example in the next section does not make any such assumption and deletes the relevant record or file as required.

Customizing Log

The `Log` class supports customization through the `install` method, which allows a subclass to be plugged in. This subclass can override some of the methods to modify logging behavior to some extent. However, there are limitations to this approach. For example, a subclass cannot access the private variable `fileWriteEnabled`, as there are no `getter` and `setter` methods for it. In order to adapt logging in a more significant way, we shall subclass `Log` but refrain from plugging it into a log object. Instead, we shall use the subclass directly to provide the desired functionalities.

There are three main objectives that we would like our custom class to meet. These are:

- We should be able to log into a file if the device supports the file connection API
- We should be able to specify the name of the file or the record store
- We should be able to delete the log (file or record)

The class that we are going to create will be called `MyLog`. This new class will have duplicates of most of the methods of `Log`. Some of these methods will be different from the corresponding ones of `Log` in order to meet the objectives mentioned above. The others will be copies of methods with the same name in the superclass. Although these methods will be the same as in the superclass, we need the duplicates because we are dealing with static methods, and we need to hide the originals so that the proper references are used. In order to understand this, consider the following method:

```
public static void p(String text, int level)
{
   instance.print(text, level);
}
```

This method in `MyLog` is an exact copy of the corresponding method in `Log`. We want to keep the method signature the same so that application code can remain unchanged even if `MyLog` is used instead of `Log`. However, the method is a static one and unless the original one is hidden, the method in `MyLog` would not be accessed. Of course, one may wonder why it is necessary to use the `MyLog` version, as the methods are identical. When `MyLog` is used instead of `Log`, `instance` in `MyLog` would be different from `instance` in `Log`, and that is why the method in `MyLog` has to be called.

We shall also duplicate the private methods of `Log`, as the original methods cannot be accessed from `MyLog`. Here we could have used different names for the methods. However, having the same method signatures can be a help if one tries to compare the original `Log` class with `MyLog`.

In order to meet our objectives, we need to understand the inner workings of the methods of the `Log` class that logs messages and retrieves them as required. Firstly, let's take a look at the methods that do the logging. There are two static methods that we can use for logging—`public static void p(String text)` and `public static void p(String text, int level)`. When the first method is called, it invokes the second with `Log.DEBUG` as the second parameter. The second method, in turn, calls a protected method `void print(String text, int level)`. We shall override the two methods but the code will remain the same as in `Log`.

```
public static void p(String text)
{
   p(text, DEBUG);
}
public static void p(String text, int level)
{
   instance.print(text, level);
}
```

Note that the `print` method that is called belongs to the `instance` object. `MyLog`, like its superclass, allows a subclass to be installed. If no subclass is installed, then `instance` is the reference to a `MyLog` object. On the other hand, if a subclass is installed, then `instance` refers to that subclass. So, the statement `instance.print(text, level)` results in a call to the `print` method of the installed subclass.

```
private static MyLog instance = new MyLog();
.
.
public static void install(MyLog newInstance)
{
   instance = newInstance;
}
```

The `print` method checks the level that is set for `instance` and proceeds to log only if the specified value of `level` for the logging operation (the second parameter passed to the method) is equal to or higher than the set value. It then appends the thread name and time stamp to the message and logs the message. If `isFileWriteEnabled` returns `true`, then a `java.io.Writer` instance is obtained and the message is logged. Otherwise the RMS is used for logging.

```
protected void print(String text, int logLevel)
{
   if(getLevel() > logLevel)
   {
      return;
   }
```

```
        text = getThreadAndTimeStamp() + " - " + text;
        System.out.println(text);
        if(isFileWriteEnabled())
        {
            try
            {
                getWriter().write(text);
            }
            catch(Throwable err)
            {
                err.printStackTrace();
                fileWriteEnabled = false;
            }
        }
        //if cannot write to file then use RMS
        else
        {
            try
            {
                RecordStore outputStore =
                    RecordStore.openRecordStore(instance.recordName, true);
                byte[] bytes = text.getBytes();
                outputStore.addRecord(bytes, 0, bytes.length);
                outputStore.closeRecordStore();
            }
            catch (RecordStoreException ex)
            {
                            ex.printStackTrace();
            }
        }
    }
```

The `Writer` instance is obtained through the `private Writer getWriter()` method, and if a writer object does not already exist, then the `protected Writer createWriter()` method is called.

```
    private Writer getWriter() throws IOException
    {
        if(output == null)
        {
            output = createWriter();
        }
        return output;
    }
```

```
protected Writer createWriter() throws IOException
{
   try
   {
      FileConnection con = (FileConnection)Connector.
         open("file:///" + FileSystemRegistry.listRoots().
         nextElement() + instance.fileName, Connector.READ_WRITE);
      if(con.exists())
      {
         con.delete();
      }
      con.create();
      return new OutputStreamWriter(con.openOutputStream());
   }
   catch(Exception err)
   {
      setFileWriteEnabled(false);

      //return a dummy writer
      return new OutputStreamWriter(new ByteArrayOutputStream());
   }
}
```

Now we are ready to meet the first two objectives stated earlier. The following methods enable us to access `fileWriteEnabled` and the name of the file or record to be used for logging:

```
public static void setFileWriteEnabled(boolean value)
{
   instance.fileWriteEnabled = value;
}

public static boolean isFileWriteEnabled()
{
   return instance.fileWriteEnabled;
}

public static void setFileName(String newName)
{
   instance.fileName = newName;
}

public static void setRecordName(String newName)
{
   instance.recordName = newName;
}
```

```
public static String getFileName()
{
   return instance.fileName;
}
public static String getRecordName()
{
   return instance.recordName;
}
```

However, we have won only the first battle. We still have to make sure that we can read from the file (or the record) we have logged into. This is easily done by writing a new method to read from the log file or from the log record, depending on the state of `fileWriteEnabled`.

```
protected String getLoggedString()
{
   try
   {
      String text = "";
      if(isFileWriteEnabled())
      {
         FileConnection con = (FileConnection) Connector.
              open("file:///" + FileSystemRegistry.listRoots().
              nextElement() + instance.fileName, Connector.READ);
         Reader r = new InputStreamReader(con.openInputStream());
         char[] buffer = new char[1024];
         int size = r.read(buffer);
         while(size > -1)
         {
            text += new String(buffer, 0, size);
            size = r.read(buffer);
         }
         r.close();
      }
      else
      {
         RecordStore store = RecordStore.openRecordStore(
                              instance.recordName, true);
         int size = store.getNumRecords();
         for(int iter = 1 ; iter <= size ; iter++)
         {
            text += new String(store.getRecord(iter));
         }
         store.closeRecordStore();
      }
```

```
            return text;
        }
        catch (Exception e)
        {
            e.printStackTrace();
            return "";
        }
    }
```

In order to preserve the original interface for reading from the log, we still have the `public static String getLogContent()` method, but all it does is call `getLoggedString`.

```
    public static String getLogContent()
    {
           return instance.getLoggedString();
    }
```

The `public static void showLog()` method remains unchanged but is hidden nonetheless, as it is a static method. This is in accordance with what has been said earlier.

All that remains now is to provide a mechanism for deleting a log file or record. The `createWriter` method we saw earlier deletes any existing log file (but not record) when the first message is logged after the application is launched. We still provide a way to delete a log file through a command or any other suitable event by including a new method `public static void deleteLog()`, which invokes `protected void deleteLogRecordOrFile()` of `instance` to do the actual work of deleting the log. This method deletes the log file or the log record, as applicable.

```
    public static void deleteLog()
    {
        instance.deleteLogRecordOrFile();
    }
    protected void deleteLogRecordOrFile()
    {
        if(instance.isFileWriteEnabled())
        {
            try
            {
                if(output != null)
                {
                    output.close();
                    output = null;
                }
```

```
                FileConnection con = (FileConnection)Connector.
                 open("file:///" + FileSystemRegistry.listRoots().
                 nextElement() + instance.fileName, Connector.READ_WRITE);
                if(con.exists())
                {
                con.delete();
                }
                con.create();
             }
             catch(Exception e)
             {
                instance.setFileWriteEnabled(false);
                instance.p("Error: Attempt to delete log file failed\r\
                        n", Log.ERROR);
             }
          }
          else
          {
             try
             {
                RecordStore.deleteRecordStore(instance.recordName);
             }
             catch(Exception e)
             {
                instance.p("Error: Attempt to delete log record failed\r\
                        n", Log.ERROR);
             }
          }
       }
```

The `MyLog` class supports the plugging in of a subclass just as its superclass does. If we plug in such a subclass, then logging will be done as determined by subclass variables such as `level` and `fileWriteEnabled`. The name of the destination for the logging operation can also be set through the `setFileName` or the `setRecordName` method. As a matter of fact, `MyLog` together with a set of subclasses can log into different log destinations depending on error classification. So you can have one file for the *default* level, one for the *warning* level, and yet another for the *error* level. You can also log into dedicated files or records for each class of your application.

The one protected method of `Log` that we did not override in `MyLog` is `getThreadAndTimeStamp`. However, if the time stamp or the thread identification format needs to be changed, it would be a simple matter to rewrite `MyLog` with the desired version of this method, and that makes us realize that we have very nearly created a new independent class for logging.

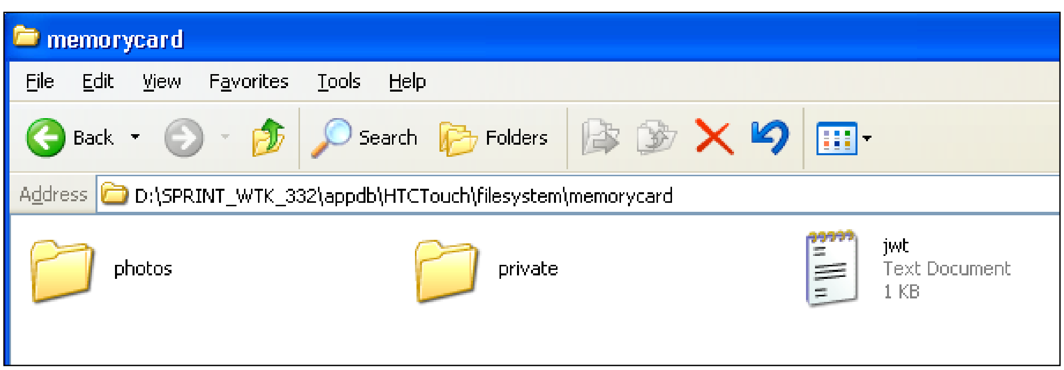
The DemoMyLog MIDlet

In order to put `MyLog` through its paces, we have the DemoMyLog MIDlet, which is derived from `DemoLogger`. The first version, which uses only `MyLog`, is a virtual copy of `DemoLogger` with `MyLog` replacing `Log` in all statements involved in logging, and with the `fileWriteEnabled` variable of `MyLog` set to `true`. The only other difference is that `DemoMyLog` does not have any method for deleting, and the `actionPerformed` method now directly calls `MyLog.deleteLog`.

```
public void startApp()
{
   Display.init(this);
   MyLog.setFileWriteEnabled(true);
   .
   .
   .
}
.
.
public void actionPerformed(ActionEvent ae)
{
   Command cmd = ae.getCommand();
   switch (cmd.getId())
   {
      .
      .
      .
      //'Delete' command
      case 4:
         MyLog.deleteLog();
   }
}
.
.
private void makeError()
{
   int first = 6;
   int second = 3;
   MyLog.p("After initialization value of second is: " +
                                    second + "\r\n");
   //MyLog.setLevel(MyLog.WARNING);
   //MyLog.setLevel(MyLog.ERROR);
   //Step 1
```

```
            second -= first/2;

            //logs using default level
            MyLog.p("After Step 1 value of second is: " + second + "\r\n");

            if(second == 0)
            {
               //logs using WARNING level
               MyLog.p("Warning: About to divide by 0 in makeError method\
                                                      r\n", MyLog.WARNING);
            }

           try
            {
               int c = first/second;
            }
            catch(ArithmeticException ae)
            {
               //logs using ERROR level
               MyLog.p("Error: Division by 0 in makeError method\r\
                                                    n", MyLog.ERROR);
            }
        }

        private void showLog()
        {
            MyLog.showLog();
            System.out.println(MyLog.getLogContent());
        }
```

If we run this MIDlet, then we would expect a file named "**jwt**" to be created, and the following screenshot shows that such a file was indeed created:

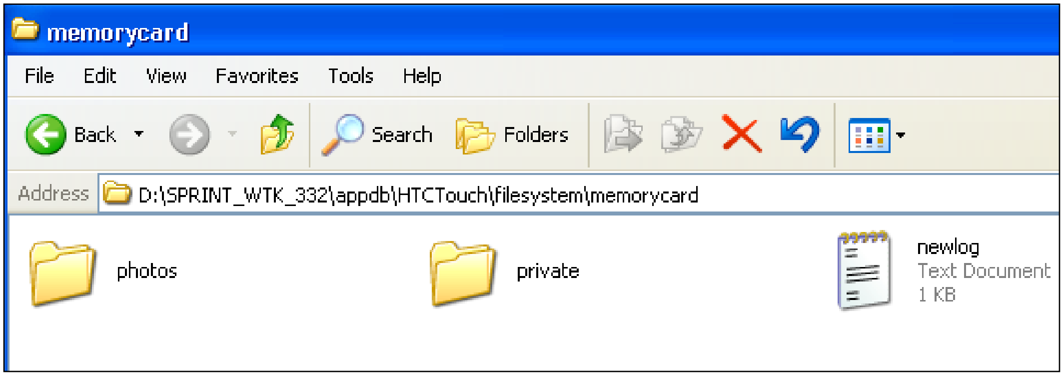


We can also check out how `MyLog` works when we plug in a subclass. `XLog` is a typical subclass of `MyLog`. It extends the protected methods and duplicates the private methods that do the logging and uses variables of appropriate `instance` to access the right variables and log file or record. Thus we can specify the logging destination for each XLog object. As we are going to plug it in, the `instance` variable of `MyLog` will refer to the right object, and the static methods will not cause the kind of conflict we had discussed in the context of the structure of `MyLog`.

There are three commented lines in the MIDlet code just after the statement that sets the `fileWriteEnabled` variable. Let us change the code shown below:

```
//MyLog.setFileWriteEnabled(true);

MyLog.install(XLog.thisLog);

MyLog.setFileWriteEnabled(System.getProperty("microedition.
     io.file.FileConnection.version") != null);

MyLog.setFileName("/newlog.log");
```

The first uncommented statement installs an `XLog` instance. The second sets the value of `fileWriteEnabled` to *true* if the platform supports the FileConnection API and to *false* if it does not. The third statement specifies the name of the file for logging (provided, of course, the platform permits file handling). The rest of the code does not need any change, as all references to `MyLog` will automatically be routed to the installed `XLog` instance. Running the modified MIDlet creates a log file with the given name.



Summary

This chapter has shown us how to use the `Effects` and the `Log` classes. `Effects`, at present, is a relatively small class, and we did not have to spend a lot of time on it. Logging was dealt with in considerable detail. We used the `Log` class to show how it can help us in figuring out what is happening inside an application at runtime. However, that is not all. We also went into its structural aspects to see how we can extend its functionalities.

Before we close the chapter, here is a word about using the `Log` class with NetBeans. Using preprocessing tags with `Log` in the NetBeans environment optimizes the source code size. For more information on this topic, please refer to the Developer's Guide.

Index

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

Thank you for buying
LWUIT 1.1
for Java ME Developers

Packt Open Source Project Royalties

When we sell a book written on an Open Source project, we pay a royalty directly to that project. Therefore by purchasing LWUIT 1.1 for Java ME Developers, Packt will have given some of the money received to the LWUIT project.

In the long term, we see ourselves and you—customers and readers of our books—as part of the Open Source ecosystem, providing sustainable revenue for the projects we publish on. Our aim at Packt is to establish publishing royalties as an essential part of the service and support a business model that sustains Open Source.

If you're working with an Open Source project that you would like us to publish on, and subsequently pay royalties to, please get in touch with us.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

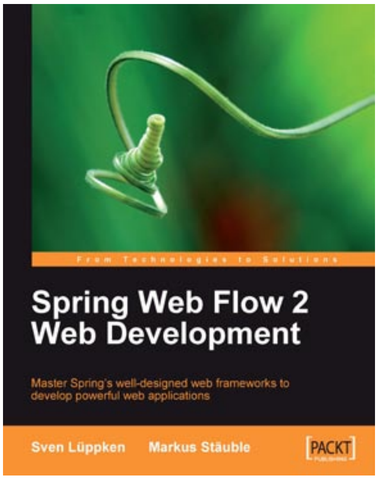
We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.PacktPub.com`.



Spring Web Flow 2 Web Development

ISBN: 978-1-847195-42-5 Paperback: 272 pages

Master Spring's well-designed web frameworks to develop powerful web applications

1. Design, develop, and test your web applications using the Spring Web Flow 2 framework
2. Enhance your web applications with progressive AJAX, Spring security integration, and Spring Faces
3. Stay up-to-date with the latest version of Spring Web Flow
4. Walk through the creation of a bug tracker web application with clear explanations

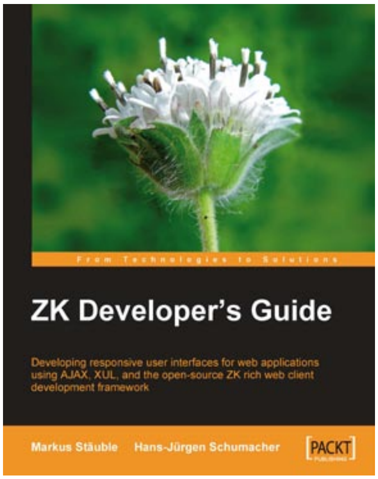
---

Google Web Toolkit GWT

ISBN: 978-1-847191-00-7 Paperback: 248 pages

A practical guide to Google Web Toolkit for creating AJAX applications with Java, fast

1. Create rich Ajax applications in the style of Gmail, Google Maps, and Google Calendar
2. Interface with Web APIs create GWT applications that consume web services
3. Completely practical with hands-on examples and complete tutorials right from the first chapter

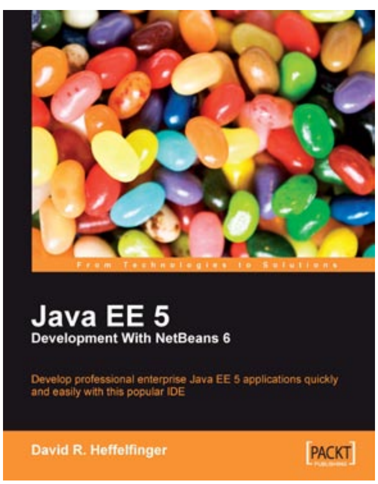


ZK Developer's Guide

ISBN: 978-1-847192-00-4 Paperback: 184 pages

Developing responsive user interfaces for web applications using Ajax, XUL, and the open source ZK rich web client development framework

1. Introducing the ZK framework
2. Installing and configuring ZK
3. Setting up, managing, and publishing a project
4. Improving navigation and optimizing result preparation
5. Internationalization with the ZK framework
6. Creating custom components



Java EE 5 Development with NetBeans 6

ISBN: 978-1-847195-46-3 Paperback: 400 pages

Develop professional enterprise Java EE applications quickly and easily with this popular IDE

1. Use features of the popular NetBeans IDE to improve Java EE development
2. Careful instructions and screenshots lead you through the options available
3. Covers the major Java EE APIs such as JSF, EJB 3 and JPA, and how to work with them in NetBeans
4. Covers the NetBeans Visual Web designer in detail

Please check **www.PacktPub.com** for information on our titles