

Lightweight Window Toolkit

Programmer's Guide

Motorola
6501 William Cannon Drive West
MD: OE112
Austin, TX 78735-8598

May 23, 2001



DigitalDNA is a trademark of Motorola, Inc.

All other trademarks are the property of their respective owners.


Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and  are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

Table of Contents

Introduction	9
1.1 Background	9
1.2 Scope	9
1.3 Design Goals	9
1.3.1 Small Footprint.....	9
1.3.2 Highly Efficient.....	10
1.3.3 Extensible	10
1.3.4 MIDP Compatible	10
1.3.5 Single-Layer Hierarchy	10
2 Class Hierarchy	12
2.1 ComponentScreen.....	14
2.2 Component	14
2.3 ComponentListener	14
2.4 InteractableComponent.....	14
2.5 Button.....	14
2.6 ImageLabel.....	15
2.6.1 Image.....	15
2.6.2 Text	15
2.6.3 Label Location	15
2.6.4 Alignment.....	15
2.7 Checkbox	16
2.8 CheckboxGroup	16
2.9 TextComponent.....	16
2.10 TextField	16
2.11 TextArea	16
2.12 Slider	17
3 Fundamental Component Behaviors	18
3.1 Component Management	18
3.1.1 Containership Rules	18
3.1.2 Component Indices.....	18
3.1.3 Z-Order.....	19
3.2 Component Regions.....	19
3.2.1 Region Parameters	19
3.2.2 Preferred Size	20

3.3	Component States	20
3.3.1	Visibility.....	21
3.3.2	Enabling	21
3.4	Component Layout	21
3.4.1	Layout Model.....	21
3.4.2	Offset Conventions.....	22
3.4.3	Centering.....	22
3.4.4	Left Edge.....	22
3.4.5	Right Edge.....	24
3.4.6	Top Edge.....	25
3.4.7	Bottom Edge.....	26
3.5	Validation Cycle.....	28
3.5.1	Invalidation	28
3.5.2	Changes to Preferred Width and Height.....	28
3.5.3	Validation Process.....	28
3.5.4	Layout Process	28
3.5.5	Validation Triggers	29
3.6	Focus Management	29
3.6.1	Focus Acceptance.....	29
3.6.2	Focus Eligibility	29
3.6.3	Focus Traversal	29
3.6.4	Requesting Focus	29
3.6.5	Focus Notifications	30
3.7	Key Event Handling	30
3.8	Pointer Event Handling.....	30
3.8.1	Pointer Event Targeting	30
3.9	Rendering	31
3.9.1	Component Screen Rendering.....	31
3.9.2	Component Rendering.....	31
3.9.3	Components Are Transparent.....	31
3.9.4	States That Impact Appearance.....	31
3.10	Scrolling	32
3.10.1	Enabling and Disabling Scrolling.....	32
3.10.2	Focus-Driven Scrolling	32
3.10.3	User Interface Scrolling	32
3.10.4	Programmatic Scrolling.....	32
4	Programming with LWT	33
4.1	The Basics	33
4.2	Using Components	34
4.3	Grouping Checkboxes	34
4.4	Screen Layout.....	36

4.5	Custom Components	38
4.6	Putting It All Together	42
Appendix A LWT APIs.....		44
A.1	ComponentScreen Class.....	44
A.1.1	ComponentScreen Definition and Constructor	44
A.1.2	ComponentScreen Methods	44
A.2	Component Class	46
A.2.1	Component Definition and Constructor	47
A.2.2	Component Fields	47
A.2.3	Component Methods	48
A.3	ComponentListener Interface.....	50
A.3.1	ComponentListener Interface Definition.....	50
A.3.2	ComponentListener Interface Methods	50
A.4	InteractableComponent.....	51
A.4.1	InteractableComponent Definition and Constructor.....	51
A.4.2	InteractableComponent Methods	51
A.5	Button Class	53
A.5.1	Button Class Definition and Constructors.....	53
A.5.2	Button Class Fields	53
A.5.3	Button Class Methods	53
A.6	Checkbox Class	54
A.6.1	Checkbox Class Definition and Constructors.....	54
A.6.2	Checkbox Class Fields	54
A.6.3	Checkbox Class Methods	55
A.7	CheckboxGroup Class.....	55
A.7.1	CheckboxGroup Class Definition and Constructor.....	55
A.7.2	CheckboxGroup Class Fields	56
A.7.3	CheckboxGroup Class Methods	56
A.8	ImageLabel Class.....	57
A.8.1	ImageLabel Class Definition and Constructors.....	57
A.8.2	ImageLabel Class Fields	58
A.8.3	ImageLabel Class Methods	59
A.9	Slider Class	59
A.9.1	Slider Class Definition and Constructor.....	60
A.9.2	Slider Class Fields.....	60
A.9.3	Slider Class Methods.....	60
A.10	TextComponent Class.....	61
A.10.1	TextComponent Class Definition and Constructor	61
A.10.2	TextComponent Class Fields	61
A.10.3	TextComponent Methods	62

A.11 TextArea Class	63
A.11.1 TextArea Class Definition and Constructor	63
A.11.2 TextArea Class Methods	64
A.12 TextField Class	64
A.12.1 TextField Class Definition and Constructor	64
A.12.2 TextField Class Methods	64

Table of Figures

Figure 1. Class Hierarchy	13
Figure 2. Z-Order	19
Figure 3. Region Parameters	20
Figure 4. Component Layout	22
Figure 5. Left Edge	23
Figure 6. Right Edge	25
Figure 7. Top Edge	26
Figure 8. Bottom Edge	27

Preface

When Sun created Java 2 Micro Edition (J2ME), the first specifications developed by the design committee were for use on small, mobile devices. Due to the small size of these devices (such as cell phones and PDAs), the initial specifications defined only a minimal subset of graphics operations for use by applications. The Lightweight Window Toolkit (LWT) is intended to expand the graphics capabilities of J2ME on mobile devices by defining a set of commonly-used widgets and making them available to application programmers. This manual describes the LWT APIs and how to use them.

Revision History

Version	Date	Author	Comments
0.1	19 April 2001	Roger Ritter	First Draft, based on Mark Patel's Design Spec.
0.2	23 May 2001	Roger Ritter	Second Draft
1.0	30 May 2001	Roger Ritter	Release

Who Should Use This Document

This manual is intended for application programmers who want to use the Lightweight Window Toolkit to implement GUI features needed by their application.

How This Document Is Organized

Chapter 1: Introduction to LWT

This chapter describe the scope and purpose of the Lightweight Window Toolkit

Chapter 2: Class Hierarchy

This chapter introduces the LWT classes and explains what they can do and how they are related.

Chapter 3: Fundamental Component Behaviors

This chapter explains the underlying behavior of the various components, and describes the options available when using these components.

Chapter 4: Programming With LWT

This chapter describes how to incorporate the LWT APIs and classes into an application.

Appendix A: LWT APIs

This appendix lists the LWT APIs, and provides detailed information about them.

Related Literature

For more information, see the following documents:

Lightweight Window Toolkit Design Document

LWT Javadocs

JSR-000037 Mobile Information Device Profile

JSR-000030 J2ME Connected, Limited Device Configuration

Introduction

1.1 Background

The Lightweight Window Toolkit (LWT) is an extension of the Java™ 2 Platform, Micro Edition (J2ME™) Mobile Information Device Profile (MIDP) specification. LWT addresses the limitations of the MIDP user interface APIs known as LCDUI. Specifically, LCDUI does not provide a developer with complete control over screen layouts, nor does it permit using custom components or extending existing components. LWT solves these problems. Designed to enhance user interface capabilities, LWT is a key enabler for the development of full-featured applications on mobile devices. It is especially valuable for delivering a rich user experience on more capable mobile devices that would otherwise be constrained by LCDUI's limited capabilities.

1.2 Scope

This document describes the LWT class hierarchy and the behaviors of those classes. It also explains the details of the LWT APIs, although final authority for the APIs rests in the LWT javadoc documentation.

1.3 Design Goals

LWT is designed to:

- Have a small footprint
- Be very efficient
- Be extensible
- Be MIDP compatible
- Have a single-layer hierarchy

1.3.1 Small Footprint

Since J2ME target devices have limited code storage capacity, the LWT package is implemented in less than 30k of byte code. Further, since RAM is typically in short supply

on J2ME devices, LWT programming is supported only on devices which have the LWT classes included in their J2ME implementation (in ROM or Flash memory). Loading the LWT class files into RAM as part of the application that uses them is not supported by Motorola.

1.3.2 Highly Efficient

J2ME target devices also have limited processing power in addition to limited memory. LWT is designed to minimize RAM and processor usage. Furthermore, the LWT APIs are designed to run without the creation of any garbage attributable to the LWT package once the application has been initialized.

1.3.3 Extensible

In order to fulfill its primary purpose, LWT is very extensible. Developers can create subclasses and override methods to create custom classes and extensions to LWT.

1.3.4 MIDP Compatible

LWT is a completely modular package that a manufacturer can add to any MIDP-compliant device without any API modifications in either package. Although LWT is completely MIDP-compliant, manufacturers may choose to implement some of the APIs in native code rather than Java for performance reasons. This is transparent to the application developer, who simply uses the LWT APIs without worrying about their implementation.

1.3.5 Single-Layer Hierarchy

With multi-layer hierarchies, such as those created with the Abstract Window Toolkit (AWT), the parent window may contain several containers, which may in turn contain additional containers, which may also contain additional containers. Thus, several containers must process an input event or repaint request before it reaches its ultimate target component. Not only is this process slow, but it can also generate significant garbage.

To avoid this problem, LWT restricts the developer to a single-layer hierarchy. The multiple layers in AWT are typically required to create complex screen layouts. For example, different groups of components may require different layouts, so each group must be placed in its own container. LWT provides a more controllable layout scheme that eliminates the need for



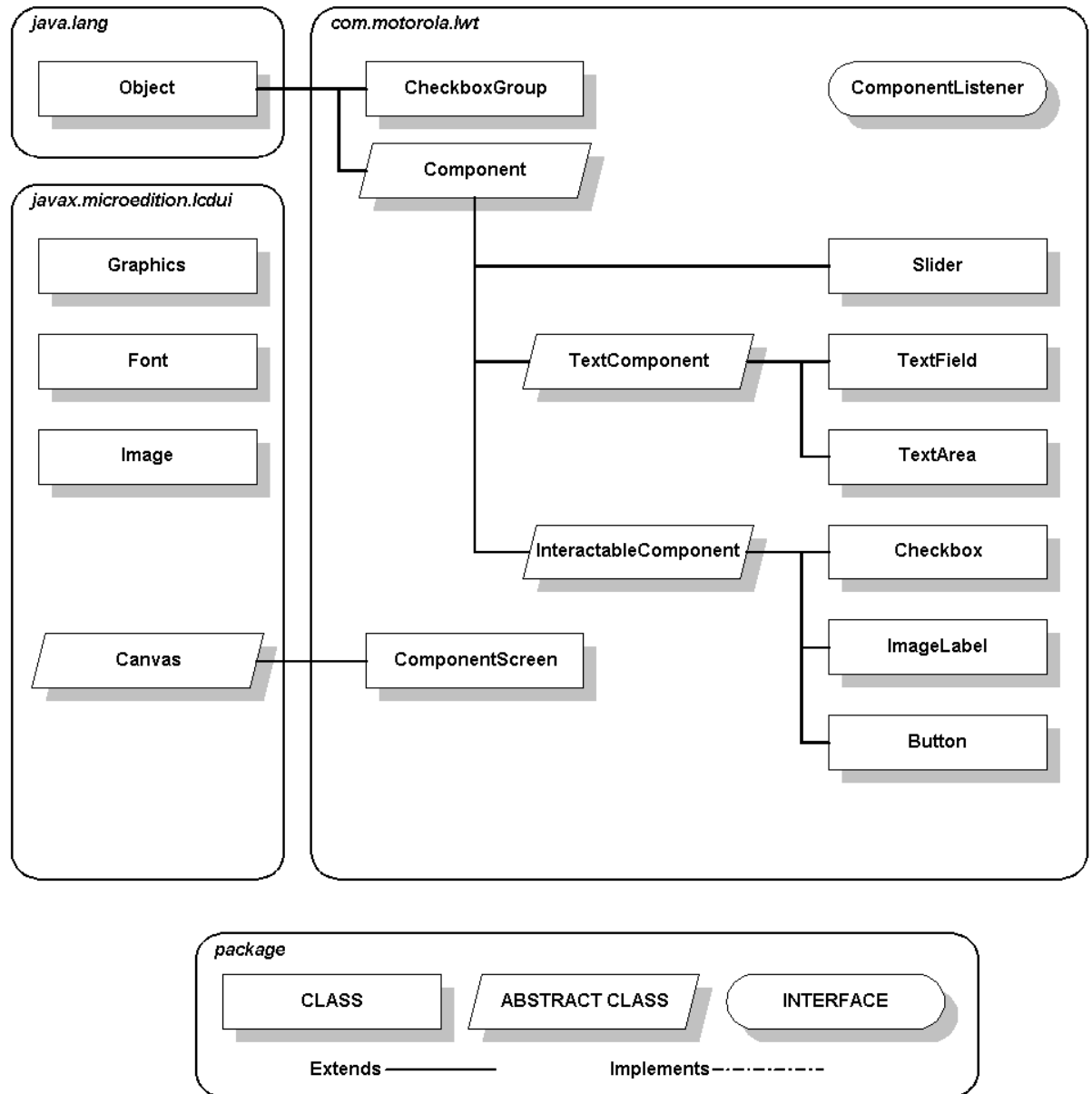
nested containers. Furthermore, the smaller screen size of a mobile device also reduces the need for multi-layer hierarchies.

2 *Class Hierarchy*

To minimize footprint, LWT employs a relatively simple class hierarchy. Due to its highly extensible nature, LWT can be limited to a modest set of components without being restrictive; developers can create special components as needed. LCDUI classes, such as **Graphics**, **Font**, and **Image**, are used wherever possible to further minimize footprint.

For more information on the classes, their capabilities, and their uses, please consult the LWT Design Document.

Figure 1. Class Hierarchy



2.1 ComponentScreen

The **ComponentScreen** class is the top-level container in an LWT user interface. As a subclass of LCDUI's **Canvas**, it can be interchanged with other LCDUI screens such as **Canvas**, **Form**, and **Alert**.

ComponentScreen inherits several methods from **Canvas** that provide the mechanisms for handling input events and repainting; thus, the interface to LCDUI is accomplished using the published APIs, and LWT can be integrated with any MIDP-compliant implementation.

2.2 Component

Component is the abstract base class of all LWT user interface entities that can be added to a **ComponentScreen**.

2.3 ComponentListener

The **ComponentListener** interface is implemented by any class that receives events from a **Component**. The **ComponentListener** is notified of an event by calling its `processComponentEvent` method with the source **Component** reference and an integer identifying the event type.

2.4 InteractableComponent

InteractableComponent is the abstract base class of the components that a user can 'press' and 'release'. Such components include buttons, checkboxes, and icons. This class serves to reduce code size and complexity of its subclasses by providing the basic interaction functionality. An **InteractableComponent** is actuated by tapping and releasing within its bounds, or by pressing and releasing the **Enter** key when the component has focus.

2.5 Button

Button is a basic button that a user can actuate. A button can display text to convey its meaning. The text font is the only customizable attribute of the **Button** class.

2.6 ImageLabel

An **ImageLabel** is a general-purpose component that can display an image and/or a text label; it can be an interactive or a read-only component.

2.6.1 Image

If an image is displayed, a developer can use either a single image or multiple images to reflect a component's different states (such as pressed, disabled, etc.).

2.6.2 Text

If text is displayed, a developer can specify the text, its color, and its font.

2.6.3 Label Location

If both text and an image are displayed, the location of the text relative to the image can be specified as:

- Above: The label is displayed centered above the image
- Below: The label is displayed centered below the image
- Left: The label is displayed centered to the left of the image
- Right: The label is displayed centered to the right of the image
- Centered: The label is displayed centered on the image

2.6.4 Alignment

Regardless of what is displayed (text, image, or both), the collective alignment of the image and/or text within the bounds of the **ImageLabel** may be specified as:

- North: The image and label both are displayed in the top half of the ImageLabel
- South: The image and label both are displayed in the bottom half of the ImageLabel
- East: The image and label both are displayed in the right half of the ImageLabel
- West: The image and label both are displayed in the left half of the ImageLabel
- Centered: The image and label both are displayed centered on the ImageLabel

2.7 Checkbox

Used as is, **Checkbox** provides a single, independent Boolean choice. A **Checkbox** displays text to convey its meaning. For example, a series of **Checkboxes** can allow a user to select toppings for a burger.

2.8 CheckboxGroup

A **CheckboxGroup** is a non-UI object that manages one or more **Checkboxes**. It can be configured to enforce multiple selection or exclusive selection rules, and can be used to query the current values of the checkboxes. When used in conjunction with a **CheckboxGroup**, **Checkboxes** can provide a list of exclusive choices in the form of radio buttons or a list. To continue the above example, a user can select how a burger is cooked using several **Checkboxes** and a **CheckboxGroup**, either as a series of radio buttons or a list.

Checkboxes can be added to or removed from a **CheckboxGroup** as needed. **Checkboxes** must still be added to the **ComponentScreen**; adding them to the **CheckboxGroup** only impacts their behavior.

2.9 TextComponent

TextComponent is the abstract base class for components that can display and edit text. It provides common functionality such as text manipulation, constraints, and input event handling.

2.10 TextField

TextField is a single-line **TextComponent** designed to display and edit text. **TextField** supports horizontal scrolling only.

2.11 TextArea

TextArea is a multi-line **TextComponent** designed for displaying and editing text. **TextArea** supports vertical scrolling only.

2.12 Slider

The **Slider** is a gauge-type component that provides a graphical representation of a numeric value. A **Slider** can be read-only or adjustable. A read-only **Slider** might be used to indicate memory usage or battery level; an adjustable **Slider** might be used to adjust a volume level.

The current value of a **Slider** represents the current setting or level of the **Slider**, which can be between 0 and the *maximum value*, inclusive. The *maximum value* of a **Slider** may be set programmatically to any non-negative integer value.

A **Slider** does not include a label. A developer can add labels and icons to the **ComponentScreen** to indicate meanings, endpoint values, etc.

3 *Fundamental Component Behaviors*

At the heart of LWT are the **ComponentScreen** and **Component** classes; together, they provide the bulk of LWT's basic functionality. This section describes the fundamental behaviors exhibited by **ComponentScreen** and **Component**.

There are compelling reasons for describing these behaviors and their mechanisms in detail. First, it allows developers to fully exploit the APIs and minimize redundant code. Second, it enables developers to customize behavior without the risk of side effects.

3.1 Component Management

3.1.1 Containership Rules

Components can be added and removed from a **ComponentScreen**. A **ComponentScreen** cannot be added to another **ComponentScreen**, and a **Component** cannot be added to another **Component**.

A **Component** can have only one parent **ComponentScreen** at a time, and it can be added to a given **ComponentScreen** only once. Whenever a **Component** is added to a **ComponentScreen**, it is first removed from the current parent if one exists, thereby ensuring that these two rules are enforced.

3.1.2 Component Indices

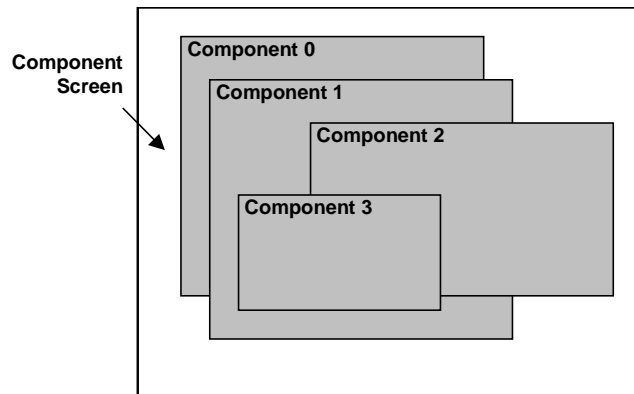
A **ComponentScreen** maintains an ordered list of its child components and assigns each one a unique index. The index of a component indicates its position in the list where 0 is the first component, and the highest index is the last component. Indices are always consecutive, so the index for a given component may change if other components are added or removed from the same **ComponentScreen**. For example, if a component is added in the middle of the list, the indices of the subsequent components will be incremented to account for the inserted component.

A component may be inserted at a specific valid index, or it may be simply appended at the end of the list and automatically assigned the next index. A component's index is significant since it implies Z-order and dictates the order in which layout and focus traversal are performed.

3.1.3 Z-Order

The component with the highest index is considered to be closest to the user, as shown below.

Figure 2. Z-Order



3.2 Component Regions

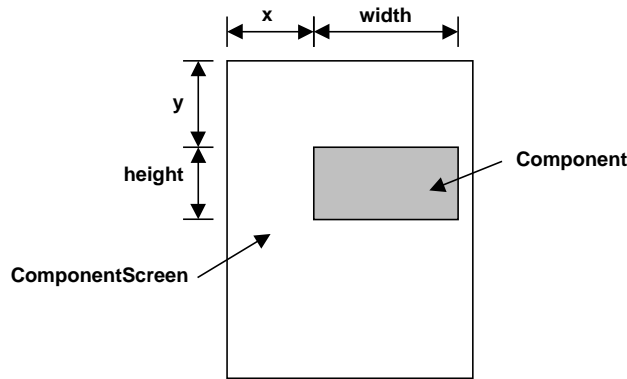
Each component occupies a rectangular region of its parent **ComponentScreen**. A component receives pointer events that occur within its rectangular region, and is responsible for rendering the pixels within its region. A component may render itself as an ellipse, a triangle, a cloud, etc., but its bounding region is always rectangular.

3.2.1 Region Parameters

The region is fully described by the location of the upper-left corner of the component and by the component's width and height. The location of the upper-left corner is relative to the **ComponentScreen**'s origin and is based on the MIDP coordinate system. Width and height are expressed in terms of pixels.

A developer can query the bounds of a component by calling `getX()`, `getY()`, `getWidth()`, and `getHeight()` on the component.

Figure 3. Region Parameters



3.2.2 Preferred Size

Each **Component** subclass must implement the methods `getPreferredWidth` and `getPreferredHeight`. Together, these two methods specify the ideal dimensions of a given component instance. Even for the same class, different instances may specify different preferred sizes to reflect the length of a text label, size of an image, etc.

The `preferredWidthChanged()` method must be called whenever the preferred width of the component changes. Similarly, the `preferredHeightChanged()` method must be called whenever the preferred height of the component changes. For standard LWT components, these methods are automatically called when a relevant parameter is changed; for custom components, it is a developer's responsibility to call these methods whenever a change is made that impacts the preferred width or height of the component.

3.3 Component States

Each **Component** instance carries state information. Subclasses may introduce additional state information as needed. The pre-defined states are:

- Visibility
- Enabling

3.3.1 Visibility

A visible component is shown to a user, whereas an invisible component is not. Components may be hidden to conceal portions of the user interface that are not relevant, thereby simplifying the user interface. By default, all components are initially visible.

3.3.2 Enabling

Enabling indicates whether or not a component is currently available to a user. Enabling and disabling is useful for conveying the availability of certain features that may be temporarily unavailable based on the current context. For example, a View button should be disabled if the corresponding list contains no items. By default, all components are initially enabled.

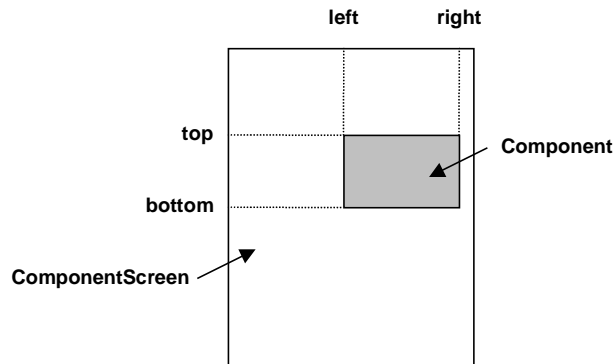
3.4 Component Layout

To meet LWT's design goals, the layout model is designed to provide a developer with complete control over component placement and size. Although this approach provides the greatest flexibility, it can result in fairly large applications, especially if the application must automatically adjust its layout to account for different display and component sizes. Therefore, the LWT layout model also incorporates several features that enable the creation of adaptable complex layouts with very little code; furthermore, the execution of these layouts is inherently efficient.

3.4.1 Layout Model

A component's region is specified in terms of its left, right, top, and bottom edges.

Figure 4. Component Layout



A developer can independently specify the location of each edge using one of several schemes. An accompanying value controls the location of the edge according to the scheme selected.

3.4.2 Offset Conventions

For all schemes that use an offset, the offset values extend down and to the right. That is, a horizontal offset extends to the right for positive values and to the left for negative values. Similarly, a vertical offset extends down for positive values and up for negative values.

3.4.3 Centering

Whenever components are centered, the location is obtained by truncating the mean of the two endpoints. This convention permits the use of a bit shift rather than a more complex division operation to determine the center. Mathematically, the center point between A and B is defined as $(A + B) \gg 1$.

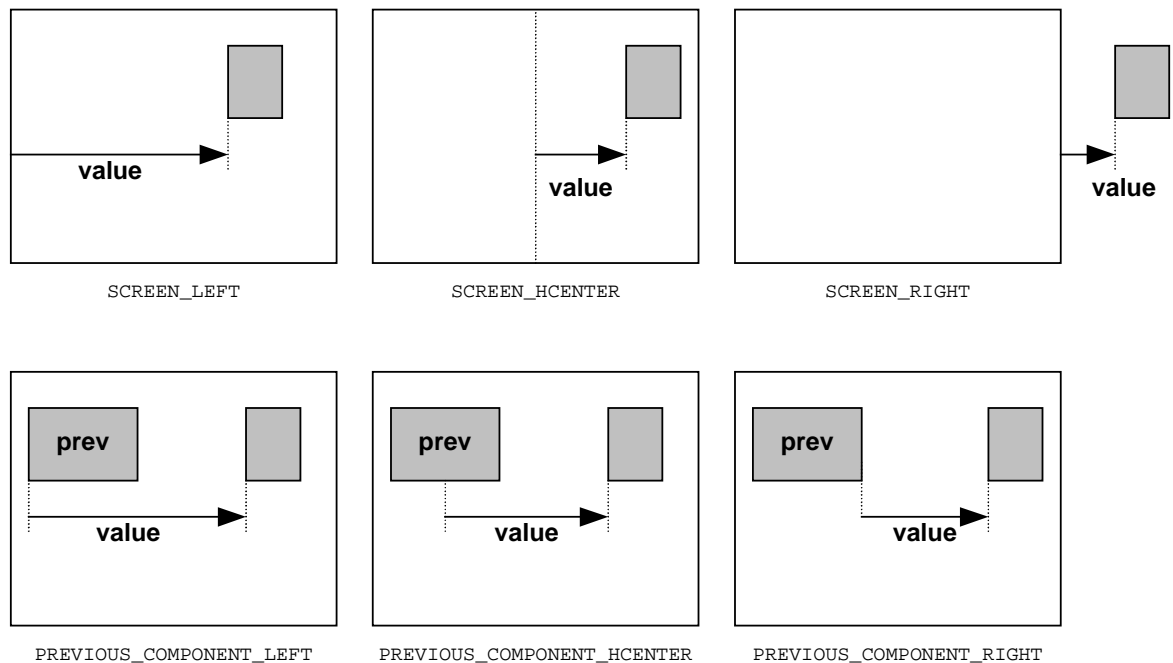
3.4.4 Left Edge

The following schemes may be used for specifying the location of a component's left edge:

Scheme	Behavior
SCREEN_LEFT (default)	The accompanying value describes the edge's offset from the left edge of the screen
SCREEN_HCENTER	The accompanying value describes the edge's offset from the center of the screen
SCREEN_RIGHT	The accompanying value describes the edge's offset from the right edge of the screen
PREVIOUS_CMponent_LEFT *	The accompanying value describes the edge's offset from the left edge of the previous component
PREVIOUS_COMPONENT_HCENTER *	The accompanying value describes the edge's offset from the center of the previous component
PREVIOUS_COMPONENT_RIGHT *	The accompanying value describes the edge's offset from the right edge of the previous component

* Interpreted as SCREEN_LEFT if there is no previous component.

Figure 5. Left Edge



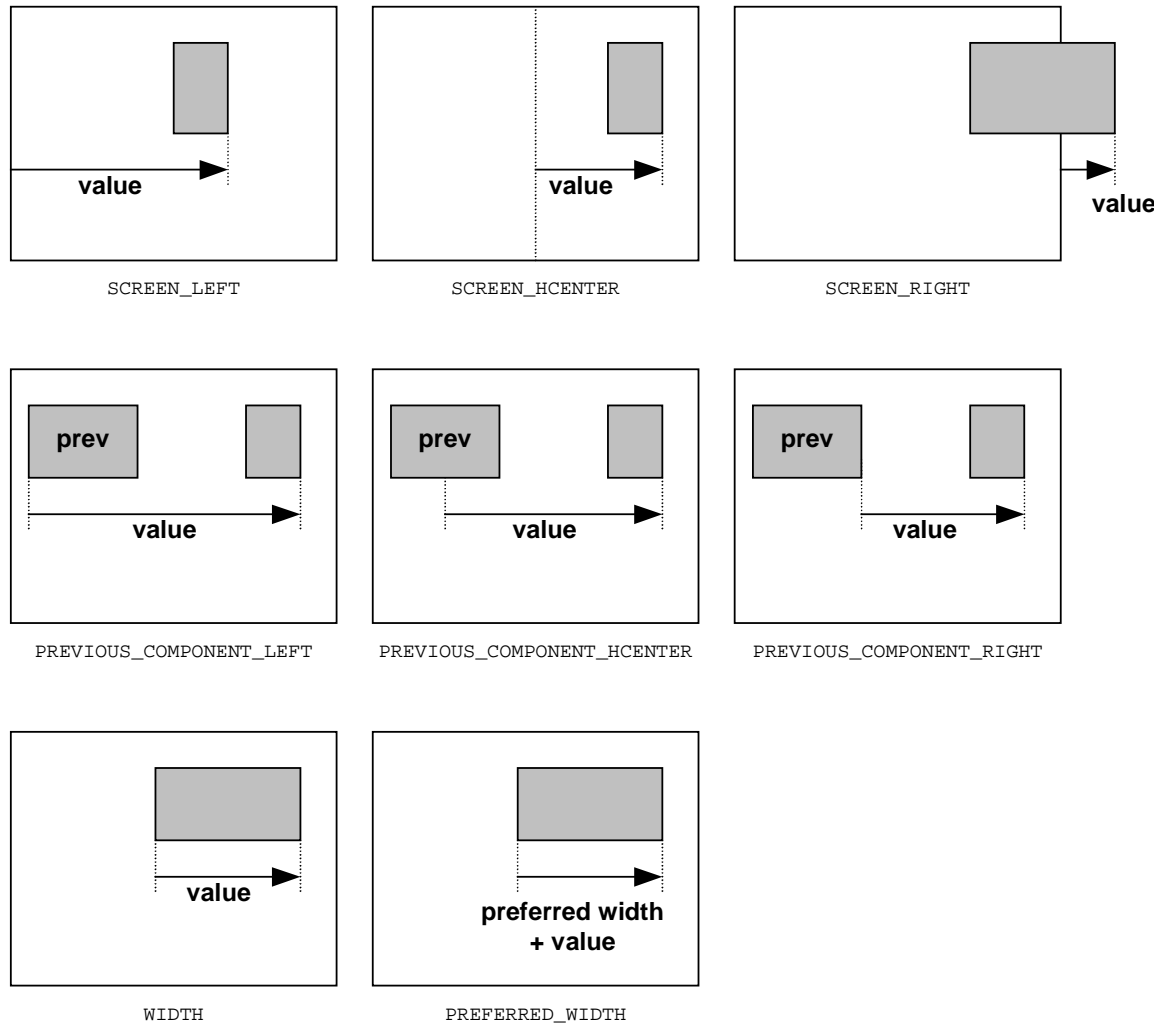
3.4.5 Right Edge

The following schemes may be used for specifying the location of a component's right edge. Developers should use `PREFERRED_WIDTH` wherever feasible to maximize application portability across different devices.

Scheme	Behavior
<code>SCREEN_LEFT</code>	The accompanying value describes the edge's offset from the left edge of the screen
<code>SCREEN_HCENTER</code>	The accompanying value describes the edge's offset from the center of the screen
<code>SCREEN_RIGHT</code>	The accompanying value describes the edge's offset from the right edge of the screen
<code>PREVIOUS_COMPONENT_LEFT *</code>	The accompanying value describes the edge's offset from the left edge of the previous component
<code>PREVIOUS_COMPONENT_HCENTER *</code>	The accompanying value describes the edge's offset from the center of the previous component
<code>PREVIOUS_COMPONENT_RIGHT *</code>	The accompanying value describes the edge's offset from the right edge of the previous component
<code>WIDTH</code>	The right edge is located such that the component's width is equal the accompanying value
<code>PREFERRED_WIDTH (default)</code>	The right edge is located such that the component's width is equal to its preferred width plus the accompanying value

* The component is set to its preferred width if there is no previous component.

Figure 6. Right Edge



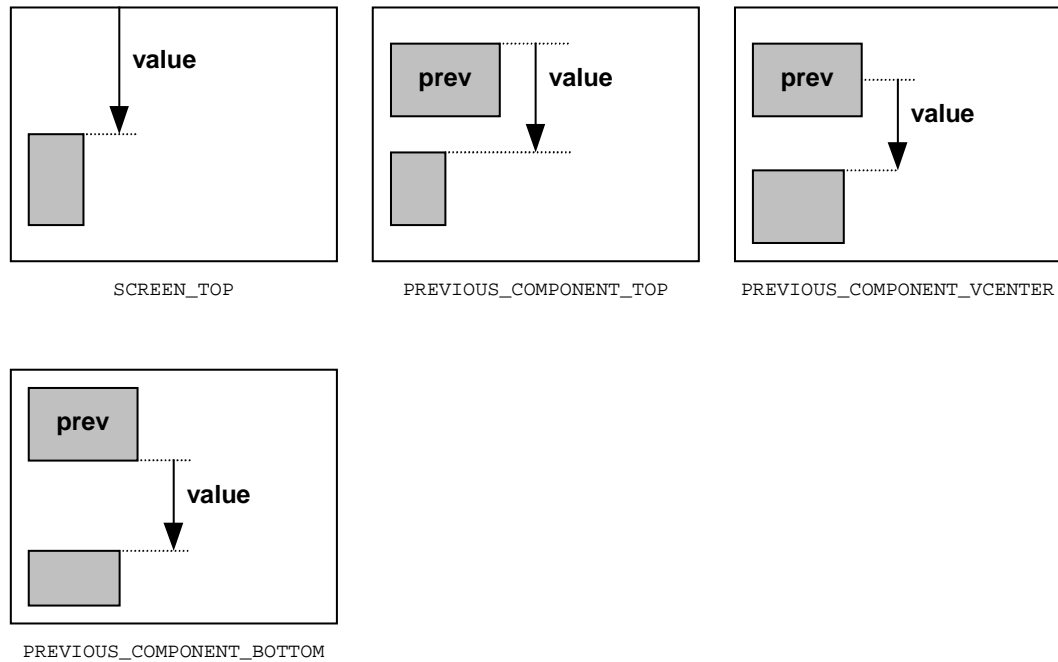
3.4.6 Top Edge

The following schemes may be used for specifying the location of a component's top edge:

Scheme	Behavior
SCREEN_TOP	The accompanying value describes the edge's offset from the top edge of the screen
PREVIOUS_COMPONENT_TOP *	The accompanying value describes the edge's offset from the top edge of the previous component
PREVIOUS_COMPONENT_VCENTER *	The accompanying value describes the edge's offset from the center of the previous component
PREVIOUS_COMPONENT_BOTTOM * (default)	The accompanying value describes the edge's offset from the bottom edge of the previous component

* Interpreted as SCREEN_TOP if there is no previous component.

Figure 7. Top Edge



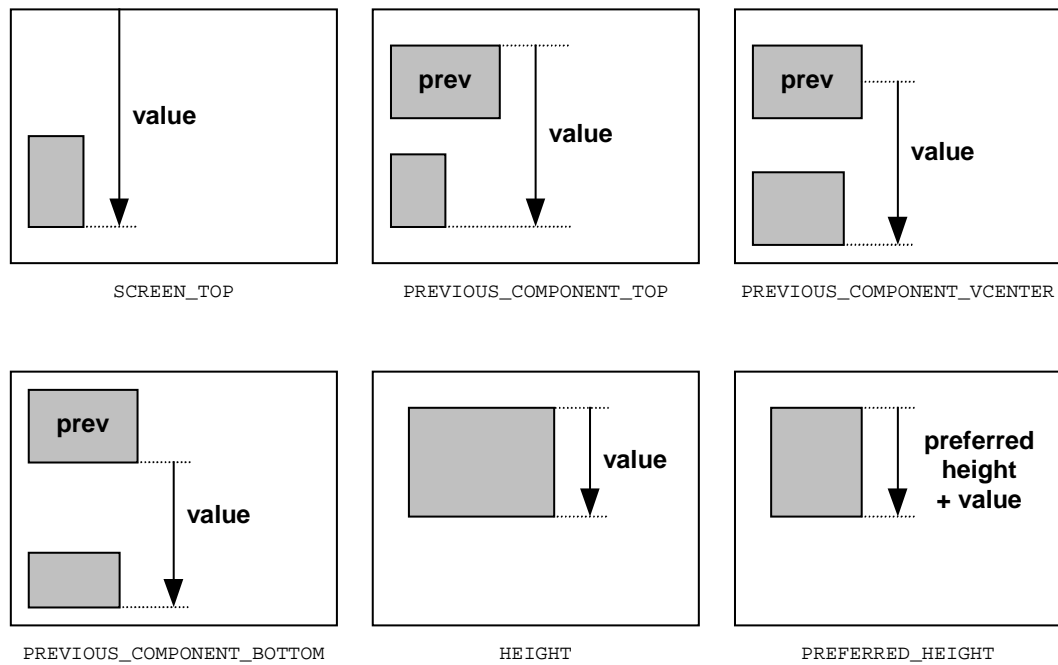
3.4.7 Bottom Edge

The following schemes may be used for specifying the location of a component's bottom edge. Developers should use `PREFERRED_HEIGHT` wherever feasible to maximize application portability across different devices.

Scheme	Behavior
SCREEN_TOP	The accompanying value describes the edge's offset from the top edge of the screen
PREVIOUS_COMPONENT_TOP *	The accompanying value describes the edge's offset from the top edge of the previous component
PREVIOUS_COMPONENT_VCENTER *	The accompanying value describes the edge's offset from the center of the previous component
PREVIOUS_COMPONENT_BOTTOM *	The accompanying value describes the edge's offset from the bottom edge of the previous component
HEIGHT	The bottom edge is located such that the component's height is equal the accompanying value
PREFERRED_HEIGHT (default)	The bottom edge is located such that the component's height is equal to its preferred height plus the accompanying value

* The component is set to its preferred height if there is no previous component.

Figure 8. Bottom Edge



3.5 Validation Cycle

To minimize redundant layout computations, the **ComponentScreen** tracks the state of its layout and computes its layout only when necessary. This mechanism effectively consolidates requests for layout computation and defers the layout process until updated component bounds are actually needed.

3.5.1 Invalidation

A **ComponentScreen** becomes *invalid* when a change is made that could potentially alter the layout of its components. Such changes include adding or removing components and changing the edge specifications or visibility of a child component. When such a change is made, the affected **ComponentScreen** automatically becomes invalid. A **ComponentScreen** can be programmatically made invalid by calling `invalidate()`.

3.5.2 Changes to Preferred Width and Height

For some components, the preferred size is dependent on component-specific attributes such as label width, image size, font, etc. In such cases, a change to one of these attributes may result in a change to the preferred width or height. When such a change occurs, the component must call `preferredWidthChanged()` or `preferredHeightChanged()`, respectively. These methods invalidate the parent if the preferred dimension is currently being used for the component's layout; otherwise the change is irrelevant and invalidation is not required.

3.5.3 Validation Process

An invalid **ComponentScreen** becomes *valid* by ensuring that the layout of its children is up to date. The process of validation involves checking whether or not the **ComponentScreen** is invalid; if so, the `doLayout()` method is called and the **ComponentScreen** then becomes valid.

3.5.4 Layout Process

The `doLayout()` method computes the location of left, right, top, and bottom edges for each component based on their schemes and accompanying values. The components are processed in ascending index order. Invisible components are ignored by the layout process; their edges are not computed and they never become the previous component.

3.5.5 Validation Triggers

Validation automatically occurs prior to any operation that relies on accurate component layout information, specifically rendering and pointer event dispatching. A developer can programmatically force validation to occur by calling `validate()`.

3.6 Focus Management

Each **ComponentScreen** instance keeps track of its current *focus owner*. The *focus owner* is the component within the **ComponentScreen** that receives key events. By default, the focus owner is null indicating that no component is currently receiving key events; in this case, the **ComponentScreen** continues to receive key events but does not dispatch them to a component. The focus owner may also become null if the current focus owner is removed or is no longer eligible to maintain focus.

3.6.1 Focus Acceptance

A component may indicate whether or not it is interested in ever becoming the focus owner by setting the Boolean field `acceptsKeyFocus` to the appropriate value. If this field is set to `true`, the component may gain focus; if `false`, the component will never gain focus.

3.6.2 Focus Eligibility

In order to be eligible to gain focus, the component must be visible, enabled, and have its `acceptsKeyFocus` field set to `true`.

3.6.3 Focus Traversal

The user may traverse focus by the appropriate keys on the device. Focus traversal occurs in component index order and skips any components that are not eligible to receive focus. Focus traversal wraps from the last component to the first component and vice versa.

Focus traversal can also be triggered programmatically by calling `focusNext()`, `focusPrevious()`, `focusFirst()`, and `focusLast()`.

3.6.4 Requesting Focus

A component may programmatically request focus by calling `requestFocus()`.

3.6.5 Focus Notifications

Whenever a component gains key focus, its `gainedFocus` method is called. Similarly, its `lostFocus` method is called whenever it loses focus. A component can query whether or not it has focus by calling `hasFocus()`.

3.7 Key Event Handling

Key events are dispatched to the current **ComponentScreen** through the three methods defined in LCDUI's **Canvas**. The default implementations of these methods in **ComponentScreen** check if there is a current focus owner and dispatch the event to that component, if any. Subclasses may override these methods to implement custom key event handling.

Key events are dispatched to the component through these three methods: `keyPressed`, `keyRepeated`, and `keyReleased`. These methods return a Boolean to indicate if the component consumed the key event, thereby allowing **ComponentScreen** subclasses to implement default behaviors for unconsumed key events.

3.8 Pointer Event Handling

Pointer events are dispatched to the current **ComponentScreen** by means of the three methods defined in LCDUI's **Canvas**. The default implementations of these methods in **ComponentScreen** dispatch these events to the appropriate *target* component. Subclasses may override these methods to implement custom pointer event handling.

3.8.1 Pointer Event Targeting

A component becomes the target when a pointer-pressed event occurs within its bounds. The search for the component is performed in descending index order to implement the correct Z-order; in the event that components overlap, the component with the highest index (i.e., closest to the user) becomes the target. Invisible and disabled components are ignored when searching for the target.

Once it becomes the target, a component continues to be the target until the pointer is released. Therefore, a component may receive pointer-drag and release events outside of its bounds.

3.9 Rendering

3.9.1 Component Screen Rendering

The **ComponentScreen** is rendered by a call to its `paint` method. By default, this method first clears the background (i.e., fills it with white pixels) and then renders its components by calling `paintComponents`. **ComponentScreen** subclasses may override the default `paint` method to implement special backgrounds or to render other artifacts on the screen.

The `paintComponents` method renders the components in ascending index order. If the components overlap, the component with the highest index is rendered last and appears to be closest to the user, thereby implementing the correct Z-order. Invisible components are not rendered.

3.9.2 Component Rendering

A component is rendered by a call to its `paint` method. The provided `Graphics` object is translated such that its origin is located at the upper-left corner of the component. Also, the clip region of the `Graphics` object is intersected with the bounds of the component.

3.9.3 Components Are Transparent

Since the **ComponentScreen** is responsible for rendering the background, **Component** does not clear the background prior to rendering. Rendering of the background by **Component** is redundant and reduces performance.

3.9.4 States That Impact Appearance

A component must render itself in a manner that conveys its current state to the user. All components must render themselves to reflect the following mutually exclusive states:

- **Normal:** Normal appearance
- **Disabled:** Should be grayed out or drawn with dotted lines instead of solid lines.
- **Focus Owner:** Normal appearance with a thick border (a component needs to support this state only if it accepts key focus)

Component subclasses may include additional states or attributes that affect their appearance; these should also be accounted for by the rendering code.

3.10 Scrolling

ComponentScreen supports vertical scrolling, but does not support horizontal scrolling.

3.10.1 Enabling and Disabling Scrolling

Scrolling is automatically enabled by the native user interface if the bottom edge of the last component extends past the bottom of the screen. In other words, scrolling support is provided when it is needed, and may be removed when it is not needed.

3.10.2 Focus-Driven Scrolling

Whenever a component receives key event focus, the screen is automatically scrolled when necessary to ensure that the component is visible to the user.

3.10.3 User Interface Scrolling

It is the responsibility of the implementation and native user interface to provide the user with the ability to control the scroll position.

3.10.4 Programmatic Scrolling

A developer can query and set the scroll position programmatically. However, a developer is not permitted to explicitly enable or disable scrolling since that functionality is implicitly provided by the device.

4 Programming with LWT

Programming with LWT is, of course, much like programming with any other Java class library. You must import the LWT package, create instances of the classes that your application uses, and then extend these as needed to provide the specific functionality desired. The classes provided by the LWT package are described briefly in appendix A, LWT APIs, and more fully in the javadoc documentation distributed with the LWT. The program LWTDemoMIDlet.java contains code samples demonstrating the use of all of the LWT components. The sample code shown below is taken from LWTDemoMIDlet.java.

4.1 The Basics

You must start by importing the LWT class files:

```
import com.motorola.lwt.*;
```

The first LWT class you should instantiate is a **ComponentScreen**, which will form the basis for laying out the LWT components to be displayed. If your application will use several screens, it may be worth creating a subclass that all of the screens can build off of. For example:

```
/**
 * Superclass for all of the demo screens, provides the next/previous
 * commands
 */
class DemoScreen extends ComponentScreen {
    public DemoScreen() {
        Command next = new Command("Next", Command.OK, 1);
        Command prev = new Command("Previous", Command.BACK, 1);
        addCommand(next);
        addCommand(prev);
    }
}
```

This class allows you to build a number of screens which have ‘Previous’ and ‘Next’ command buttons in addition to whatever components you decide to place on the individual screens.

4.2 Using Components

Once you have your **ComponentScreen** defined, you can now start adding LWT components to it. All of the components defined by LWT are used alike, so the methods shown here are applicable to all of the LWT components. This example creates a screen with three buttons on it:

```
/**
 * Demo Screen for the Button Component
 */
class ButtonScreen extends DemoScreen {
    public ButtonScreen() {
        Button b1 = new Button("Button");
        add(b1);

        Button b2 = new Button("Large Button");
        b2.setRightEdge(Component.SCREEN_RIGHT, 0);
        b2.setBottomEdge(Component.HEIGHT, b2.getPreferredHeight() * 2);
        add(b2);

        Button b3 = new Button("Disabled Button");
        b3.setEnabled(false);
        add(b3);
    }
}
```

This code also shows how to modify the defaults when creating a new component. Since buttons are automatically created as 'Enabled', button b3 has specific code to initialize it as disabled. The code for button b2 demonstrates a method for changing the default size of the button.

4.3 Grouping Checkboxes

While the pre-defined LWT components can all be created and used individually (as shown in the previous example), some components are also designed to be used together. Although individual **Checkboxes** have many uses, it's also nice to be able to combine them to provide a multiple-choice grouping. The **CheckboxGroup** class is designed to make that possible. Here's how to use it:

```
/**
```

```
* Demo Screen for the RadioButton Component (Checkbox with a
CheckboxGroup)
**/
class RadioButtonScreen extends DemoScreen {
    public RadioButtonScreen() {
        CheckboxGroup g = new CheckboxGroup(Checkbox.STYLE_RADIO_BUTTON);

        Checkbox c1 = new Checkbox("Radio Button A");
        add(c1);
        g.add(c1);

        c1 = new Checkbox("Radio Button B");
        add(c1);
        g.add(c1);
        c1 = new Checkbox("Radio Button C");
        add(c1);
        g.add(c1);

        Checkbox c2 = new Checkbox("Large Radio Button");
        c2.setRightEdge(Component.SCREEN_RIGHT, 0);
        c2.setBottomEdge(Component.HEIGHT, c2.getPreferredHeight() * 2);
        add(c2);
        g.add(c2);

        Checkbox c3 = new Checkbox("Small Radio Button");
        c3.setRightEdge(Component.SCREEN_RIGHT, 0);
        c3.setBottomEdge(Component.HEIGHT, c3.getPreferredHeight() - 8);
        add(c3);
        g.add(c3);

        Checkbox c4 = new Checkbox("Disabled Radio Button");
        c4.setEnabled(false);
        add(c4);
        g.add(c4);
    }
}
```

Note in this example that each **Component** is added not only to the **ComponentScreen**, but also to the **CheckboxGroup**. The type of **CheckboxGroup** is specified when the **CheckboxGroup** is instantiated (in this case it's a radio button group), and the

Checkboxes themselves are created and customized just as the **Buttons** in the earlier example.

4.4 Screen Layout

One of the drawbacks of the MIDP that LWT addresses is that MIDP display elements can't easily be laid out in relation to each other. LWT provides the programmer with much greater flexibility in laying out the screen since LWT components can be placed on the screen either in relation to the screen itself or in relation to other components. See section 3.4 of this document for details on the available layout methods.

This code shows some of the possibilities:

```
/**
 * Demo Screen for various layout options
 */
class LayoutScreen extends DemoScreen {

    public LayoutScreen () {

        Button a = new Button("In");
        add(a);

        Button b = new Button("A");
        b.setTopEdge(Component.PREVIOUS_COMPONENT_TOP, 0);
        b.setLeftEdge(Component.PREVIOUS_COMPONENT_RIGHT, 3);
        add(b);

        Button c = new Button("Row");
        c.setTopEdge(Component.PREVIOUS_COMPONENT_TOP, 0);
        c.setLeftEdge(Component.PREVIOUS_COMPONENT_RIGHT, 3);
        add(c);

        Checkbox d = new Checkbox("In");
        d.setLeftEdge(Component.PREVIOUS_COMPONENT_LEFT, 0);
        d.setRightEdge(Component.SCREEN_RIGHT, 0);
        d.setTopEdge(Component.PREVIOUS_COMPONENT_BOTTOM, 3);
        add(d);
    }
}
```

```
Checkbox e = new Checkbox("A");
e.setLeftEdge(Component.PREVIOUS_COMPONENT_LEFT, 0);
e.setRightEdge(Component.SCREEN_RIGHT, 0);
add(e);

Checkbox f = new Checkbox("Column");
f.setLeftEdge(Component.PREVIOUS_COMPONENT_LEFT, 0);
f.setRightEdge(Component.SCREEN_RIGHT, 0);
add(f);

Button g = new Button("Across The Screen");
g.setRightEdge(Component.SCREEN_RIGHT, 0);
add(g);

Checkbox i = new Checkbox("Side");
Button h = new Button("And");
h.setBottomEdge(Component.HEIGHT, i.getPreferredHeight() * 3);
h.setRightEdge(Component.WIDTH, h.getPreferredWidth() * 2);
add(h);

CheckboxGroup grp = new
    CheckboxGroup(Checkbox.STYLE_RADIO_BUTTON);

i.setLeftEdge(Component.PREVIOUS_COMPONENT_RIGHT, 4);
i.setTopEdge(Component.PREVIOUS_COMPONENT_TOP, 0);
add(i);
grp.add(i);

Checkbox j = new Checkbox("By");
j.setLeftEdge(Component.PREVIOUS_COMPONENT_LEFT, 0);
add(j);
grp.add(j);

Checkbox k = new Checkbox("Side");
k.setLeftEdge(Component.PREVIOUS_COMPONENT_LEFT, 0);
add(k);
grp.add(k);
    }
}
```

4.5 Custom Components

Although LWT provides the most commonly-used screen components, some applications may require additional features. The extensibility designed into LWT allows an application programmer to create these additional features when they are needed.

To create a custom feature in LWT that's not based on one of the provided components, you must extend the **Component** class. In addition to the code that actually implements your new feature, you must also provide methods for `paint()`, `getPreferredWidth()`, and `getPreferredHeight()`. These classes are defined as abstract by **Component**, and so have no default code associated with them.

This code shows how to build two custom components – an animator, and a greyscale selector. First, we set up the custom component screen.

```
/**
 * Demo Screen for custom components
 */
class CustomComponentScreen extends DemoScreen {

    public CustomComponentScreen() {

        add(new GrayPicker());

        Animation a = new Animation();
        a.setTopEdge(Component.PREVIOUS_COMPONENT_BOTTOM, 10);
        add(a);
    }
}
```

Here, we define the greyscale selector.

```
/**
 * A simple picker component that allows the user to select a gray level
 */
class GrayPicker extends Component {

    public GrayPicker() {
        acceptsKeyFocus = true;
    }
}
```

```
public int getPreferredWidth() {
    return 20 * count;
}

public int getPreferredHeight() {
    return 20;
}

int selection = 0;
int count = 4;

public void paint(Graphics g) {

    int w = getWidth();
    int h = getHeight();

    if (hasFocus()) {
        g.drawRect(0, 0, w - 1, h - 1);
    }

    w /= count;

    for (int n=0; n < count; n++) {
        g.setGrayScale(255 * n / (count-1));

        g.fillRect(4, 4, w-8, h-8);

        if (selection == n) {
            g.setGrayScale(0);
            g.drawRect(0, 0, w-1, h-1);
            g.drawRect(1, 1, w-3, h-3);
            g.drawRect(2, 2, w-5, h-5);
        }
        g.translate(w, 0);
    }
}

public boolean keyPressed(int keyCode) {

    int action = getParent().getGameAction(keyCode);
```

```
        if (action == Canvas.LEFT) {
            if (selection > 0) {
                selection--;
                repaint();
            }
            return true;
        }
        else if (action == Canvas.RIGHT) {
            if (selection < count - 1) {
                selection++;
                repaint();
            }
            return true;
        }
        return false;
    }

    public void pointerPressed(int x, int y) {
        requestFocus();
        int s = x / (getWidth() / count);
        selection = s;
        repaint();
    }
}
```

This section defines a threaded animator class.

```
class Animation extends Component implements Runnable {

    Image img;
    int frameHeight = 0;
    int frameWidth = 0;
    int frameIndex = 0;

    public Animation () {
        try
        {
            img = Image.createImage("/com/mot/demo/doggy.png");
            frameHeight = img.getHeight() / 3;
            frameWidth = img.getWidth();
        }
    }
}
```



```
        new Thread(this).start();
    }
    catch (Throwable e)
    {
    }
}

public int getPreferredWidth() {
    return frameWidth;
}

public int getPreferredHeight() {
    return frameHeight;
}

public void paint(Graphics g) {
    g.translate((getWidth() - frameWidth) / 2,
               (getHeight() - frameHeight) / 2);

    g.clipRect(0, 0, frameWidth, frameHeight);

    g.drawImage(img, 0, - frameHeight * frameIndex, Graphics.TOP +
               Graphics.LEFT);

    synchronized (this) {
        notify();
    }
}

public void run() {
    while (true) {

        try {
            synchronized (this) {
                frameIndex = (frameIndex + 1) % 3;
                repaint();
                wait();
            }
        }
    }
}
```

```
        Thread.sleep(150);
    }
    catch (Throwable e)
    {
    }
}
}
```

4.6 Putting It All Together

The sections above described how to define and use various LWT components, but don't really show how a MIDlet might actually use them. This code shows how to do it. To see it in all its glory, look at the LWTDemoMIDlet.java program. Here is just the MIDlet snippet:

```
public class LWTDemoMIDlet extends MIDlet implements CommandListener
{
    // List of screens to show in the appropriate order
    // Additional screens can be simply added to this list, no other
    // changes are needed :-)
    DemoScreen[] screens = {
        new ButtonScreen(),
        new CheckboxScreen(),
        new RadiobuttonScreen(),
        new TextFieldScreen(),
        new LayoutScreen(),
        new CustomComponentScreen()
    };

    public LWTDemoMIDlet() {
        // Set this midlet as the command listener for the demo screens
        for (int n=screens.length - 1; n >= 0; n--) {
            screens[n].setCommandListener(this); /* changed this */
        }
    }

    protected void startApp()
    throws MIDletStateChangeException {
        // Show the first screen
        Display.getDisplay(this).setCurrent(screens[0]);
    }
}
```

```
protected void pauseApp()
{
}

protected void destroyApp(boolean unconditional)
throws MIDletStateException
{
}

public void commandAction (Command c, Displayable d) {

    // Find the screen that generated the command action
    for (int n=screens.length - 1; n >= 0; n--) {
        if (screens[n] == d) {
            // Found it, check which command was triggered
            if (c.getCommandType() == Command.BACK) {
                // Go to the previous screen
                Displayable d1 = d.getCurrentScreen().getPreviousScreen();
                d.setCurrentScreen(d1);
            }
            else if (c.getCommandType() == Command.OK) {
                // Go to the next screen if there is one
                Displayable d1 = d.getCurrentScreen().getNextScreen();
                d.setCurrentScreen(d1);
            }
            return;
        }
    }
}
```

Appendix A LWT APIs

LWT contains methods which allow an application programmer to instantiate and/or extend all of the LWT classes, and to use them in a program. Each of these classes and components is briefly described in this appendix. For a full description, refer to the LWT javadoc documentation.

A.1 ComponentScreen Class

The **ComponentScreen** class extends the **Canvas** class, and forms the basis for all LWT screen layouts. An application defines a user interface screen by creating a **ComponentScreen** and then adding the desired LWT components to it. With this class, a developer can create a screen from an arbitrary mix of components, including special component subclasses. It also provides the developer with complete control over the screen layout.

A.1.1 ComponentScreen Definition and Constructor

The **ComponentScreen** class is defined by:

```
public class ComponentScreen extends javax.microedition.lcdui.Canvas
```

Its only constructor is:

```
public ComponentScreen()
```

A.1.2 ComponentScreen Methods

In addition to the methods described in this section, a **ComponentScreen** inherits many methods from the `javax.microedition.lcdui.Canvas`, `javax.microedition.lcdui.Displayable`, and `java.lang.Object` classes. These inherited methods are listed in the LWT API documentation.

The methods specifically defined by **ComponentScreen** are:

- `void add (Component w)` – adds a **Component** to the **ComponentScreen**
- `protected void doLayout ()` – recomputes the layout of the **Components** according to the edge specifications for each component

- `Component GetComponent (int index)` – gets the **Component** at the specified index
- `int GetComponentCount ()` – gets the number of **Components** currently contained in this screen
- `Component getFocusOwner ()` – gets the **Component** that currently has key event focus
- `int getScrollOffset()` - gets the current vertical scroll offset
- `int getWidth()` - gets the width of the **ComponentScreen**
- `void insert(Component comp, int index)` - inserts a **Component** to the screen at the specified index
- `void invalidate()` - invalidates this **ComponentScreen**, indicating that its component layouts need to be recalculated
- `protected void keyPressed(int keyCode)` - called when a key is pressed
- `protected void keyReleased(int keyCode)` - called when a key is released
- `protected void keyRepeated(int keyCode)` - called when a key is repeated (held down)
- `void paint(javax.microedition.lcdui.Graphics g)` - renders the screen
- `protected void paintComponents(javax.microedition.lcdui.Graphics g)` - renders the **Components**
- `protected void pointerDragged(int x, int y)` - called when the pointer is dragged
- `protected void pointerPressed(int x, int y)` - called when the pointer is pressed
- `protected void pointerReleased(int x, int y)` - called when the pointer is released
- `void remove(Component comp)` - removes the specified **Component** from the screen
- `void remove(int index)` - removes the **Component** at the specified index from the screen
- `void removeAll()` - removes all **Components** from this screen

- `void scrollTo(Component comp)` – scrolls to the specified **Component**. This method insures that the screen's scroll position is adjusted to show as much as possible of the specified **Component**.
- `void setFocusFirst()` - moves key event focus to the first **Component** that accepts focus
- `void setFocusLast()` - moves key event focus to the last **Component** that accepts focus
- `void setFocusNext()` - moves key event focus to the next **Component** that accepts focus
- `void setFocusPrevious()` - moves key event focus to the previous **Component** that accepts focus
- `void setScrollOffset(int offset)` - sets the vertical scroll offset
- `protected void showNotify()` - called when the **ComponentScreen** is shown
- `void validate()` - validates this **ComponentScreen**

Detailed information about using these methods is available in the LWT API documentation.

A.2 Component Class

The **Component** class is the abstract base class from which the various LWT components are descended. Each subclass descended from the **Component** class must implement methods to render the **Component**, provide preferred width and height, and optionally, to handle events.

A **Component** is a single user interface object that occupies a rectangular region of its parent **ComponentScreen**. A **Component** can belong only to a single screen. If a program attempts to add a **Component** to a second screen, it will first be removed from its current screen (which could be the screen that it is being added to) before it is added to the second screen.

The location and size of a **Component** is determined by specifying where the left, right, top and bottom edges are located. The location of each edge can be specified using one of several schemes, including offsets relative to the preceding **Component**, as defined in Section 3.4 of this document. By default, a **Component** will be set to its preferred size, and will be aligned with the left edge of the screen directly beneath the preceding **Component**.

A **Component** must provide a paint method to render itself. The **Graphics** object passed to the **Component**'s `paint()` method is translated so that the origin is located at the upper left corner of the **Component**.

If desired, a **Component** can respond to key and pointer events by overriding the appropriate methods (`keyPressed()`, `pointerDragged()`, etc.) In order to receive key event focus, a **Component** must have `acceptsKeyFocus` set to `true`, and it must be visible and enabled. Provided the parent screen is shown, the **Component** with key event focus will receive all key events.

A.2.1 Component Definition and Constructor

The **Component** class is defined by:

```
public abstract class Component extends java.lang.Object
```

Its only constructor is:

```
public Component()
```

A.2.2 Component Fields

These constants define the values used by the programmer to define where the **Component** code should place and size the **Component** on the screen, as well as to define its ability to interact with the user. These constants are typically passed to some of the methods defined below to indicate from where the **Component** location value is to be measured.

- `protected boolean acceptsKeyFocus` - indicates if this **Component** accepts key focus (that is, it uses key events)
- `static int HEIGHT` - the bottom edge is located such that the **Component**'s height is equal to the accompanying value
- `static int PREFERRED_HEIGHT` - the bottom edge is located such that the **Component**'s height is equal to its preferred height plus the accompanying value
- `static int PREFERRED_WIDTH` - the right edge is located such that the **Component**'s width is equal to its preferred width plus the accompanying value
- `static int PREVIOUS_COMPONENT_BOTTOM` - the value describes the edge's offset from the bottom edge of the previous **Component**
- `static int PREVIOUS_COMPONENT_HCENTER` - the value describes the edge's offset from the horizontal center of the previous **Component**

- `static int PREVIOUS_COMPONENT_LEFT` - the value describes the edge's offset from the left edge of the previous **Component**
- `static int PREVIOUS_COMPONENT_RIGHT` - the value describes the edge's offset from the right edge of the previous **Component**
- `static int PREVIOUS_COMPONENT_TOP` - the value describes the edge's offset from the top edge of the previous **Component**
- `static int PREVIOUS_COMPONENT_VCENTER` - the value describes the edge's offset from the vertical center of the previous **Component**
- `static int SCREEN_HCENTER` - the value describes the edge's offset from the center of the screen
- `static int SCREEN_LEFT` - the value describes the edge's offset from the left edge of the screen
- `static int SCREEN_RIGHT` - the value describes the edge's offset from the right edge of the screen
- `static int SCREEN_TOP` - the value describes the edge's offset from the top edge of the screen
- `static int WIDTH` - the right edge is located such that the **Component's** width is equal to the accompanying value

A.2.3 Component Methods

In addition to the methods defined directly by the **Component** class, this class inherits several methods from the **java.lang.Object** class. See the LWT API documentation for details on these inherited methods. The methods defined by the **Component** class are:

- `boolean acceptsFocus()` - checks if this **Component** currently accepts key focus
- `void gainedFocus()` - called when this **Component** gains key focus
- `int getHeight()` - gets the height of the **Component**, in pixels
- `ComponentScreen getParent()` - obtains a reference to the **Component's** parent screen
- `abstract int getPreferredHeight()` - gets the preferred height of this **Component**
- `abstract int getPreferredWidth()` - gets the preferred width of this **Component**

- `int getWidth()` - gets the width of the **Component**, in pixels
- `int getX()` - gets the x coordinate of the **Component's** left edge within the parent
- `int getY()` - gets the y coordinate of the **Component's** top edge within the parent
- `boolean hasFocus()` - checks if this **Component** currently has key focus (that is, it is receiving key events). For a given screen, no more than one **Component** can have key focus
- `protected void invalidateParent()` - invalidates this **Component's** parent screen, if any
- `boolean isEnabled()` - checks if this **Component** is currently enabled (can be interacted with by the user)
- `boolean isVisible()` - checks if this **Component** is visible (can be seen by the user)
- `protected boolean keyPressed(int keyCode)` - called when a key is pressed
- `protected boolean keyReleased(int keyCode)` - called when a key is released
- `protected boolean keyRepeated(int keyCode)` - called when a key is repeated (held down)
- `void lostFocus()` - called when this **Component** loses key focus
- `abstract void paint(javax.microedition.lcdui.Graphics g)` - renders the **Component**
- `protected void pointerDragged(int x, int y)` - called when the pointer is dragged
- `protected void pointerPressed(int x, int y)` - called when the pointer is pressed within this **Component**
- `protected void pointerReleased(int x, int y)` - called when the pointer is released
- `protected void preferredHeightChanged()` - notifies the system that the preferred height of this **Component** has changed
- `protected void preferredWidthChanged()` - notifies the system that the preferred width of this **Component** has changed
- `void repaint()` - requests a repaint for the entire **Component**
- `void repaint(int x, int y, int width, int height)` - requests a repaint for the specified portion of this **Component**

- `void requestFocus()` - requests key focus for this **Component**
- `void setBottomEdge(int scheme, int value)` - specifies the location of the **Component's** bottom edge. The scheme parameter is one of the constants defined above, specifying from where the value is to be measured.
- `void setEnabled(boolean b)` - sets this **Component** as enabled or disabled
- `void setLeftEdge(int scheme, int value)` - specifies the location of the **Component's** left edge. The scheme parameter is one of the constants defined above, specifying from where the value is to be measured.
- `void setRightEdge(int scheme, int value)` - specifies the location of the **Component's** right edge. The scheme parameter is one of the constants defined above, specifying from where the value is to be measured.
- `void setTopEdge(int scheme, int value)` - specifies the location of the **Component's** top edge. The scheme parameter is one of the constants defined above, specifying from where the value is to be measured.
- `void setVisible(boolean visible)` - shows or hides the **Component**

Detailed information about using these methods is available in the LWT API documentation.

A.3 ComponentListener Interface

The **ComponentListener** interface is implemented by any class that wishes to receive events from a **Component**. The events vary depending on the object that originates the event, but can include events such as button actuation, checkbox selection, etc.

A.3.1 ComponentListener Interface Definition

This interface is defined as

```
public interface ComponentListener
```

A.3.2 ComponentListener Interface Methods

The **ComponentListener** interface defines a single method, `ProcessComponentEvent()`, which (as expected) processes an event received by a **Component**. It is defined as

```
public void processComponentEvent(java.lang.Object source, int eventType)
```

where `source` is the object which originated the event, and `eventType` is the type of event that occurred. `EventType` is defined by the originating object's class, and is only defined to be unique within that class. A listener should call the appropriate methods in the originating object to determine information about the event that occurred.

A.4 InteractableComponent

The **InteractableComponent** class is a subclass of **Component**. This class adds functionality to allow it to interact with the user, i.e. the user can 'press' and 'release' objects created from this class. It provides the basic pointer and key event handling required by checkboxes, buttons, image labels, etc. It also provides methods for setting and getting the text and font associated with the component.

An **InteractableComponent** may be actuated by tapping and releasing within its bounds. It may also be actuated by pressing and releasing the device's 'Enter' key when it has key focus.

Button, **Checkbox**, and **ImageLabel** are all subclasses of **InteractableComponent**.

A.4.1 InteractableComponent Definition and Constructor

The **InteractableComponent** class is defined by

```
public abstract class InteractableComponent extends Component
```

Its only constructor is

```
InteractableComponent(java.lang.String label)
```

This constructs a new **InteractableComponent** with the specified label. It also sets `acceptsKeyFocus` to `true` so that this **Component** can accept focus and key events.

A.4.2 InteractableComponent Methods

Since this class extends the **Component** class, it inherits many fields and methods from that class. In addition, it defines the following methods:

- `abstract void componentActuated()` - called when the **Component** is actuated (tapped and released). Subclasses must define this method to perform the appropriate actions when they are actuated.
- `protected void dispatchComponentEvent(int event)` - dispatches the specified event to this **InteractableComponent**'s listener, if any
- `javax.microedition.lcdui.Font getFont()` - gets the **Font** associated with this label
- `java.lang.String getLabel()` - gets the label for this **Component**
- `boolean isPressed()` - checks if this **InteractableComponent** is currently pressed
- `protected boolean keyPressed(int keyCode)` - called when a key is pressed
- `protected boolean keyReleased(int keyCode)` - called when a key is released
- `abstract void paint(javax.microedition.lcdui.Graphics g)` - renders the **Component**
- `protected void pointerDragged(int x, int y)` - called when the pointer is dragged
- `protected void pointerPressed(int x, int y)` - called when the pointer is pressed
- `protected void pointerReleased(int x, int y)` - called when the pointer is released
- `void setComponentListener(ComponentListener l)` - sets this **InteractableComponent**'s listener
- `void setFont(javax.microedition.lcdui.Font font)` - sets the **Font** object for rendering the label, if any
- `void setLabel(java.lang.String label)` - sets the label for this **InteractableComponent**
- `void setPressed(boolean b)` - sets the pressed/unpressed state of this **Component**

A.5 Button Class

A **Button** is a subclass of **InteractableComponent**. As such, it can interact with the user. Optionally, it can display a label to convey its function.

A.5.1 Button Class Definition and Constructors

The **Button** class is a subclass of **InteractableComponent** and is defined by:

```
public class Button extends InteractableComponent
```

It has two constructors:

`Button()` – Construct a **Button** with no label

`Button(java.lang.String label)` – Construct a **Button** with the given text string as a label

A.5.2 Button Class Fields

The **Button** class has a single constant, which is used to send an event to a **Button's** listener, if any, when the **Button** is pressed.

```
public static int BUTTON_ACTION_EVENT
```

A.5.3 Button Class Methods

The **Button** class inherits methods from **InteractableComponent** and **Component**. In addition, it defines these methods:

- `void componentActuated()` - called when this **Button** is actuated. This implementation dispatches a `BUTTON_ACTION_EVENT` to the **Button's** listener, if any.
- `int getPreferredHeight()` - gets the preferred height for this **Button**
- `int getPreferredWidth()` - gets the preferred width for this **Button**
- `void paint(javax.microedition.lcdui.Graphics g)` - renders the **Button**

A.6 Checkbox Class

The **Checkbox** is a component that represents a boolean value.

A **Checkbox** can be used as-is to provide a single, independent choice for the user. A **Checkbox** can also be added to a **CheckboxGroup** to provide more extensive functionality.

A.6.1 Checkbox Class Definition and Constructors

The **Checkbox** class is a subclass of **InteractableComponent** and is defined by:

```
public class Checkbox extends InteractableComponent
```

It has two constructors:

`Checkbox()` - creates a new **Checkbox** with an empty (null) label

`Checkbox(java.lang.String label)` - creates a new **Checkbox** with the specified label

A.6.2 Checkbox Class Fields

These constants are used to pass events to a listener, if any, or to define the appearance of the **Checkbox**:

- `static int CHECKBOX_CHECKED_EVENT` - event indicating that this **Checkbox** was checked
- `static int CHECKBOX_UNCHECKED_EVENT` - event indicating that this **Checkbox** was unchecked
- `static int STYLE_CHECKBOX` - indicates that this **Checkbox** should look like a checkbox
- `static int STYLE_LIST_ITEM` - indicates that this **Checkbox** should look like a list item
- `static int STYLE_RADIO_BUTTON` - indicates that this **Checkbox** should look like a radiobutton

A.6.3 Checkbox Class Methods

Checkbox inherits several methods from **InteractableComponent** and **Component**. In addition, it defines the following methods:

- `void componentActuated()` - called when this **Checkbox** is actuated by the user
- `void gainedFocus()` - called when this **Checkbox** gains key focus
- `int getPreferredHeight()` - gets the preferred height of this **Checkbox**
- `int getPreferredWidth()` - gets the preferred width of this **Checkbox**
- `boolean getValue()` - gets the current value of this **Checkbox**
- `void lostFocus()` - called when this **Checkbox** loses key focus
- `void paint(javax.microedition.lcdui.Graphics g)` - renders the **Checkbox**
- `void setValue(boolean value)` - sets the current value of this **Checkbox**.

A.7 CheckboxGroup Class

The **CheckboxGroup** manages a group of **Checkboxes**. Unlike the AWT class of the same name, the use of a **CheckboxGroup** does not imply an EXCLUSIVE list. A **CheckboxGroup** can be constructed for both exclusive and multiple selection modes. In LWT, the **CheckboxGroup** serves as a single reference point for several **Checkboxes**, eliminating the need to deal with each **Checkbox** individually. The **CheckboxGroup** supports a **ComponentListener**, so the interested object can listen to just the **CheckboxGroup**, rather than having to add itself as a listener to each of the **Checkboxes** individually.

A.7.1 CheckboxGroup Class Definition and Constructor

The **CheckboxGroup** class is a subclass of **Object** and is defined by:

```
public class CheckboxGroup extends java.lang.Object
```

This class has only a single constructor:

`CheckboxGroup(int style)` - creates a new **CheckboxGroup** with the given style

A.7.2 CheckboxGroup Class Fields

The **CheckboxGroup** defines a single event which indicates that the value of one or more of the **Checkboxes** in the group has changed.

```
static int CHECKBOXGROUP_SELECTION_CHANGED
```

A.7.3 CheckboxGroup Class Methods

CheckboxGroup defines the following methods:

- `int add(Checkbox b)` - adds a **Checkbox** to this group and returns the index assigned to it
- `Checkbox getCheckbox(int index)` - gets the **Checkbox** with the specified index
- `int getCheckboxCount()` - gets the number of **Checkboxes** that belong to this **CheckboxGroup**
- `int getSelectedIndex()` - gets the index of the selected element
- `void insert(Checkbox b, int index)` - Inserts a **Checkbox** into this group at the specified index
- `boolean isSelected(int index)` - gets the value of the **Checkbox** with the specified index
- `void remove(Checkbox b)` - removes the specified **Checkbox** from this group
- `void remove(int index)` - removes the **Checkbox** with the specified index from this group
- `void setComponentListener(ComponentListener l)` - sets this **Component's** listener
- `void setSelectedFlags(boolean[] selectedArray)` - sets the values of the group's **Checkboxes** to the values of the provided array. For single-select **Checkboxgroups**, only the first true value in the array will be set. Other true values will be ignored. For multi-select **CheckboxGroups**, all true values will be set.
- `void setSelectedIndex(int index, boolean value)` - sets the selection for this **CheckboxGroup**

A.8 ImageLabel Class

ImageLabel is a **Component** that can display an image and/or a text label. The image and/or text label, referred to as the **ImageLabel**'s contents, can be collectively placed North, South, East, West, or centered within the bounds of the **ImageLabel**. See Section 2.6 for more information.

The relative layout of the contents can be controlled by specifying the location of the text label relative to the image. The text label can be placed either above the image, below the image, to the left of the image, to the right of the image, or in the center of the image.

An **ImageLabel** may be either interactable or non-interactable. An interactable **ImageLabel** may be actuated by the user and its state can be normal, disabled or pressed. Separate images may be provided for each of these states; the normal image is used by default if a specific image for is not provided for a given state.

A non-interactable **ImageLabel** cannot be actuated by the user; its state can be either normal or disabled.

The separation between the text and the image when the LABEL_RIGHT or LABEL_LEFT position is selected is 3 pixels. The separation between the text and the image when the LABEL_ABOVE or LABEL_BELOW position is selected is 2 pixels. In the absence of either text or image, the separation will be zero

A.8.1 ImageLabel Class Definition and Constructors

The **ImageLabel** class is a subclass of **InteractableComponent** and is defined by:

```
public class ImageLabel extends InteractableComponent
```

This class has two constructors:

```
ImageLabel( javax.microedition.lcdui.Image normal,  
            javax.microedition.lcdui.Image disabled,  
            javax.microedition.lcdui.Image pressed, java.lang.String  
            label ) - constructs a new interactable ImageLabel with the specified images for the  
three states (normal, disabled and pressed) and the specified label.
```

`ImageLabel(javax.microedition.lcdui.Image normal, javax.microedition.lcdui.Image disabled, java.lang.String label)` - constructs a new non-interactable **ImageLabel** with the specified images for the two states (normal and disabled).

A.8.2 ImageLabel Class Fields

The fields defined for the **ImageLabel** class are mainly used to indicate the relative positions of the image and text items within the object. One event is defined for use by any listeners.

- `static int ALIGN_CENTER` - the image and label should be horizontally and vertically centered within the **ImageLabel**
- `static int ALIGN_EAST` - The image and label should be vertically centered and aligned with the right edge of the **ImageLabel**
- `static int ALIGN_NORTH` - The image and label should be horizontally centered and aligned with the top edge of the **ImageLabel**
- `static int ALIGN_SOUTH` - The image and label should be horizontally centered and aligned with the bottom edge of the **ImageLabel**
- `static int ALIGN_WEST` - The image and label should be vertically centered and aligned with the left edge of the **ImageLabel**
- `static int HORIZONTAL_GAP` - the horizontal gap between image and the text
- `static int IMAGE_LABEL_ACTION_EVENT` - event indicating that the **ImageLabel** was actuated by the user (for interactable **ImageLabels** only)
- `static int LABEL_ABOVE` - the label, if any, should be placed above the image and horizontally centered relative to the image
- `static int LABEL_BELOW` - the label, if any, should be placed below the image and horizontally centered relative to the image
- `static int LABEL_CENTER` - the label, if any, should be centered on the image
- `static int LABEL_LEFT` - the label, if any, should be placed to the left of the image and vertically centered relative to the image
- `static int LABEL_RIGHT` - the label, if any, should be placed to the right of the image and vertically centered relative to the image
- `static int TRANSPARENT` - transparent background color
- `static int VERTICAL_GAP` - the vertical gap between image and text

A.8.3 ImageLabel Class Methods

The **ImageLabel** class inherits many methods from **InteractableComponent** and **Component**. In addition, it defines the following methods:

- `void componentActuated()` - called when this **ImageLabel** is actuated by the user
- `int getBackgroundColor()` - gets the current background color
- `int getForegroundColor()` - gets the current foreground color
- `int getPreferredHeight()` - gets the preferred height of the **ImageLabel**
- `int getPreferredWidth()` - gets the preferred width of the **ImageLabel**
- `void paint(javax.microedition.lcdui.Graphics g)` - paints this **ImageLabel**
- `void setAlignment(int alignment)` - sets the desired alignment for this **ImageLabel**
- `void setBackgroundColor(int color)` - sets the background color
- `void setDisabledImage(javax.microedition.lcdui.Image i)` - sets the image for the disabled state
- `void setForegroundColor(int color)` - sets the foreground color
- `void setLabelLocation(int location)` - sets the location of the label, if any, relative to the **ImageLabel**'s image
- `void setNormalImage(javax.microedition.lcdui.Image i)` - sets the image for the normal state
- `void setPressedImage(javax.microedition.lcdui.Image i)` - sets the image for the pressed state

A.9 Slider Class

The **Slider** component represents a variable (and possibly adjustable) numeric value.

The **Slider** can either be a read-only device to display a value, or it can be an interactable device that allows the user to view and adjust a value. Only horizontal **Sliders** are supported since the vertical direction keys are reserved for changing key focus.

A **Slider's** value can range from 0 to its maximum value, inclusive. The maximum value must be at least 0. There is no upper limit on the maximum value; however, the resolution of a **Slider** will be reduced if its maximum value exceeds its width.

A.9.1 Slider Class Definition and Constructor

The **Slider** class is a subclass of **Component**, and is defined by:

```
public class Slider extends Component
```

The constructor for a **Slider** is:

```
Slider(boolean interactive, int maxValue, int value)
```

A.9.2 Slider Class Fields

The **Slider** class defines two fields which are passed as events to any listeners that may be active on the class. These fields are:

- `static int SLIDER_DRAGGED` - event indicating that the **Slider's** value has been changed and that it is still being interacted with by the user. Since this event may happen repeatedly and quickly, the code that deals with it should execute quickly.
- `static int SLIDER_SET` - event indicating that the **Slider's** value has been changed and that the user is no longer interacting with it.

A.9.3 Slider Class Methods

The **Slider** class inherits many methods from **Component**. In addition, it defines the following methods:

- `int getMaxValue()` - gets the maximum value of the **Slider**
- `int getPreferredHeight()` - gets the preferred height of the **Slider**
- `int getPreferredWidth()` - gets the preferred width of the **Slider**
- `int getValue()` - gets the current value of the **Slider**
- `protected boolean keyPressed(int keyCode)` - called when a key is pressed
- `protected boolean keyReleased(int keyCode)` - called when a key is released

- `protected boolean keyRepeated(int keyCode)` - called when a key is repeated (held down)
- `void paint(javax.microedition.lcdui.Graphics g)` - renders the **Component**
- `protected void pointerDragged(int x, int y)` - called when the pointer is dragged
- `protected void pointerPressed(int x, int y)` - called when the pointer is pressed
- `protected void pointerReleased(int x, int y)` - called when the pointer is released
- `void setComponentListener(ComponentListener l)` - sets this **Slider's** listener
- `void setMaxValue(int maxValue)` - sets the maximum value of the **Slider**
- `void setValue(int value)` - sets the current value of the **Slider**

A.10 TextComponent Class

The **TextComponent** class is the base class for **TextArea** and **TextField**. It provides common functionality such as text manipulation, font control, length limiting, constraints, justification and echo character support. The native implementation of **TextComponents** may include support for selection, cut/copy/paste, handwriting recognition, keypad prediction, etc.; however, these features are not exposed in the API since they are not guaranteed to be supported by all devices.

A.10.1 TextComponent Class Definition and Constructor

The **TextComponent** class is a subclass of **Component**, and is defined by:

```
public abstract class TextComponent extends Component
```

There is no constructor for this class. Use a **TextArea** or **TextField** class to define a text handling object.

A.10.2 TextComponent Class Fields

The **TextComponent** class defines the following constants:

- `static int JUSTIFY_CENTER` - constant for center justification
- `static int JUSTIFY_LEFT` - constant for left justification
- `static int JUSTIFY_RIGHT` - constant for right justification
- `static int NO_LIMIT` - constant for no length limit

A.10.3 TextComponent Methods

TextComponent inherits several methods from **Component**. In addition, it defines these methods:

- `void appendChar(char c)` - appends the specified character at the end of the current text
- `void appendText(java.lang.String text)` - appends the specified text at the end of the current text
- `int getConstraint()` - gets the text entry constraint for this **TextComponent**
- `char getEchoChar()` - obtains the echo character used by this **TextComponent**, or 0 if the actual characters are displayed.
- `javax.microedition.lcdui.Font getFont()` - gets the font currently used by this **TextComponent**
- `int getLengthLimit()` - gets the length limit
- `int getPreferredHeight()` - gets the preferred height of this **TextComponent**
- `int getPreferredWidth()` - gets the preferred width of this **TextComponent**
- `java.lang.String getText()` - obtains the text contained in this **TextComponent**
- `void insertChar(char c, int index)` - inserts the specified character at the specified index in the current text
- `void insertText(java.lang.String newText, int index)` - inserts the specified text at the specified index in the current text
- `boolean isEditable()` - checks whether or not the contents of this **TextComponent** may be edited by the user
- `boolean keyPressed(int keyCode)` - called when a key is pressed

- `boolean keyRepeated(int keyCode)` - called when a key is repeated (held down)
- `void paint(javax.microedition.lcdui.Graphics g)` - renders the **Component**
- `void pointerPressed(int x, int y)` - called when the pointer is pressed within this **Component**
- `void setConstraint(int constraint)` - sets the text entry constraint for the contents of this **TextComponent**
- `void setEchoChar(char c)` - sets the echo character to be displayed by this **TextComponent**. Use 0 to display the actual characters typed.
- `void setEditable(boolean editable)` - sets whether or not the **TextComponent** can be edited by the user
- `void setFont(javax.microedition.lcdui.Font newFont)` - sets the current font of the **TextComponent**
- `void setJustification(int justification)` - sets the justification of this **TextComponent**
- `void setLengthLimit(int maxChars)` - sets the length limit
- `void setText(java.lang.String newText)` - sets the contents of the **TextComponent** to the specified **String**
- `protected void textChanged(int start, int end, boolean user)` - called whenever the contents of this **TextComponent** are changed, either by the user or programmatically

A.11 TextArea Class

The **TextArea** class is a multi-line **TextComponent** that scrolls vertically as needed.

Scrolling is automatically provided by the platform. A scrollbar (or other similar mechanism) is provided by the native UI as needed so that the user can adjust the scroll offset.

A.11.1 TextArea Class Definition and Constructor

This class is defined by

```
public class TextArea extends TextComponent
```

The only constructor is:

```
TextArea(java.lang.String text, int rows, int columns)
```

which constructs a new **TextArea** with the given text, number of rows, and number of columns. The number of rows and columns is used only to establish the preferred height and width for layout purposes; it does not restrict the length of text that can be entered into the **TextArea**.

A.11.2 TextArea Class Methods

The **TextArea** class inherits all of its methods from the **TextComponent** and **Component** classes. There are no additional methods defined in the **TextArea** class.

A.12 TextField Class

The **TextField** class defines a single line **TextComponent** that scrolls horizontally as needed.

A.12.1 TextField Class Definition and Constructor

This class is defined by

```
public class TextField extends TextComponent
```

The only constructor is:

```
TextField(java.lang.String text, int columns)
```

which constructs a new **TextField** with the given text and number of columns. The number of columns is used only to establish the preferred width for layout purposes; it does not restrict the length of text that can be entered into the **TextField**.

A.12.2 TextField Class Methods

The **TextField** class inherits all of its methods from the **TextComponent** and **Component** classes. There are no additional methods defined in the **TextArea** class.