SERIES 2™

Bananas UI Toolkit
Developer's Guide

Release 1.3

How to use the Bananas widget set in your HME applications

Contents

4 Customization

Overview

1

This chapter includes:

What Is the Bananas UI Toolkit?

The Bananas User Interface Toolkit helps you create applications that are compatible with the TiVo® look-and-feel and that follow the general behavior of the TiVo user interface (UI). The Bananas Toolkit includes a set of standard widgets—software components that plug into your application—including a button with highlights, text, a list with scrollbar and highlights, and an onscreen keyboard. Bananas tools fall into two general categories:

- *An application framework*—screens, transitions, layering of objects, focus management
- *The widget set*—button, text, list, and keyboard

Although the Bananas Toolkit does not emulate the TiVo UI exactly, it aids you in creating applications that "extend" the TiVo experience for the user. The toolkit also decreases coding time by providing you with basic, commonly used widgets so you can focus your programming efforts on features unique to your application.

What Does the Toolkit Contain?

The Bananas UI Toolkit is an extension of the Home Media Engine (HME) Software Development Kit. You can download both developer kits, with Java® APIs, from this website:

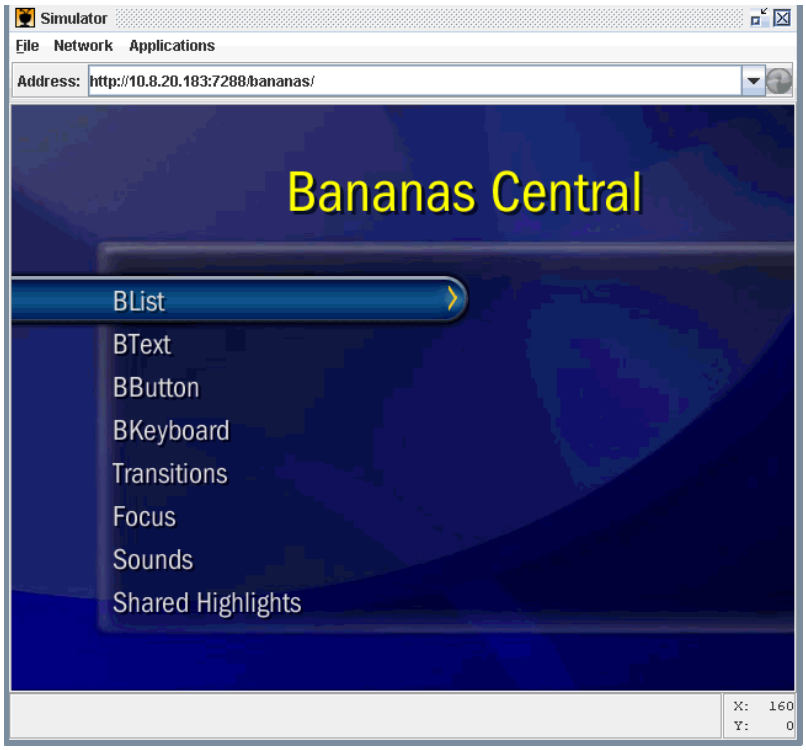
http://sourceforge.net/projects/tivohme/

This toolkit contains the following

- *Source code*
- *Documentation*—In addition to this manual, see the Java class documentation and the Release Notes for the toolkit.

- *Sample application*—The sample application (Figure 1-1) demonstrates use of all Bananas widgets and also demonstrates playing sounds, specifying screen transitions, managing focus within a screen, and sharing highlights among widgets. This manual discusses many aspects of the sample code in detail, but you'll probably also want to explore its source code on your own as well.

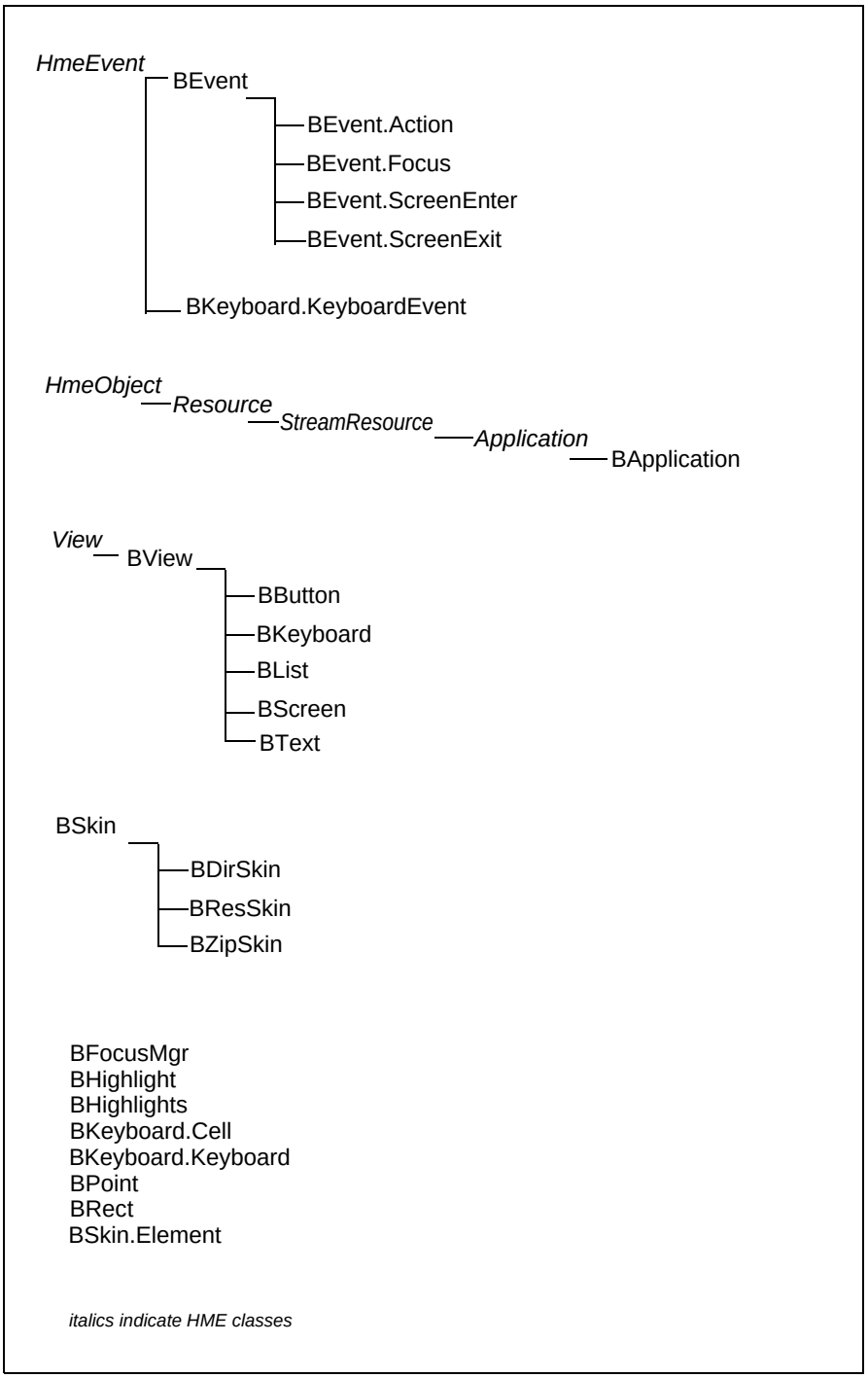
Figure 1-1. Home screen for Bananas Central sample application



Bananas Class Tree

Figure 1-2 shows the Bananas class tree. Some classes are derived from HME classes. For background information, see the *HME SDK Developer's Guide* and the Java class documentation for HME.

Figure 1-2. Class tree for the Bananas UI Toolkit



About This Guide

This guide is intended for developers with some Java experience. Before you begin programming with this toolkit, it is recommended that you become familiar with the HME Software Development Kit and the underlying HME Protocol.

Key Concepts

2

This chapter includes:

This chapter discusses some of the important concepts underlying use of the Bananas UI Toolkit, including the following:

- How screens are pushed onto and popped off the stack
- Layers within a screen and within the application
- Entering and exiting screens
- Styles for transitioning from one screen to the next
- Sending events to the widget (or view) that has the focus
- Playing custom sounds

BApplication and the Screen Stack

The core of your application is the *BApplication* class, which is subclassed from the HME *Application* class. *BApplication* contains a stack of screens (of class *BScreen*). You construct the user interface for your application by creating screens and pushing them onto the stack. Each *BScreen* contains one screenful of information. A *BScreen* has the same dimensions as the application.

The top screen on the stack is visible. Whenever a new screen is pushed onto the stack or a screen is popped off the stack, the old screen is hidden and the new screen is shown. The screen being hidden is said to be “exited,” and the new screen that is visible is said to be “entered.”

User Interface Layers

BApplication has three layered views that are used to organize your interface:

- *above:* contains views that appear “above all screens”
- *normal:* contains the screens
- *below:* contains views that appear “below” all screens, such as a background image.

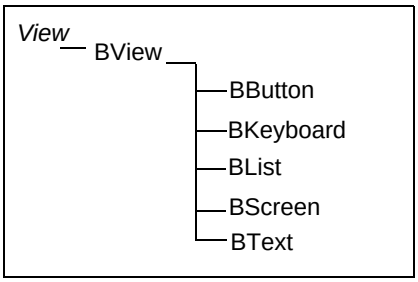
It is important to place screens in the *normal* layer of *BApplication* because *BApplication* performs sliding transitions (see “Transitions,” later in this section) by positioning the old and new screens inside *normal* and then using an animation call to *normal.setTranslation()* to slide the screens around. If you want your application to have a background image that’s shared across all screens, put it into the *below* layer of *BApplication*.

Views and Widgets

As shown in Figure 2-1, *BView*, derived from the HME *View* class, is the base class for widgets and screens. A *screen* contains one or more *views*.

BView is a bounding box associated with a resource. A *BView* object is capable of receiving focus, and it can be highlighted.

Figure 2-1. Class tree for *BView*



Screen Layers

A screen has three layers:

- *above:* contains views that are “above” the widgets. This layer contains whispering arrows and other decorations. (*Whispering arrows* are the arrows that provide a hint to the user about which arrow buttons are active at a particular spot in the UI.)
- *normal:* contains widgets such as buttons, text, lists, and keyboards
- *below:* contains backgrounds as well as highlights such as the background bar used in lists to show focus

You can supply a different background for every screen, or you can specify one global background for *BApplication*. In cases where both an individual screen and the application have backgrounds, the screen background obscures the application’s background.

Transitions

Each screen has an associated *transition* that describes its behavior when it is pushed and popped. Possible transitions are

- TRANSITION_LEFT: exited screen slides left to make room for new screen to slide in
- TRANSITION_FADE: cross fade between the exited and entered screens
- TRANSITION_NONE: exited screen is simply replaced by entered screen

See the Bananas sample application for a demonstration of screen transitions.

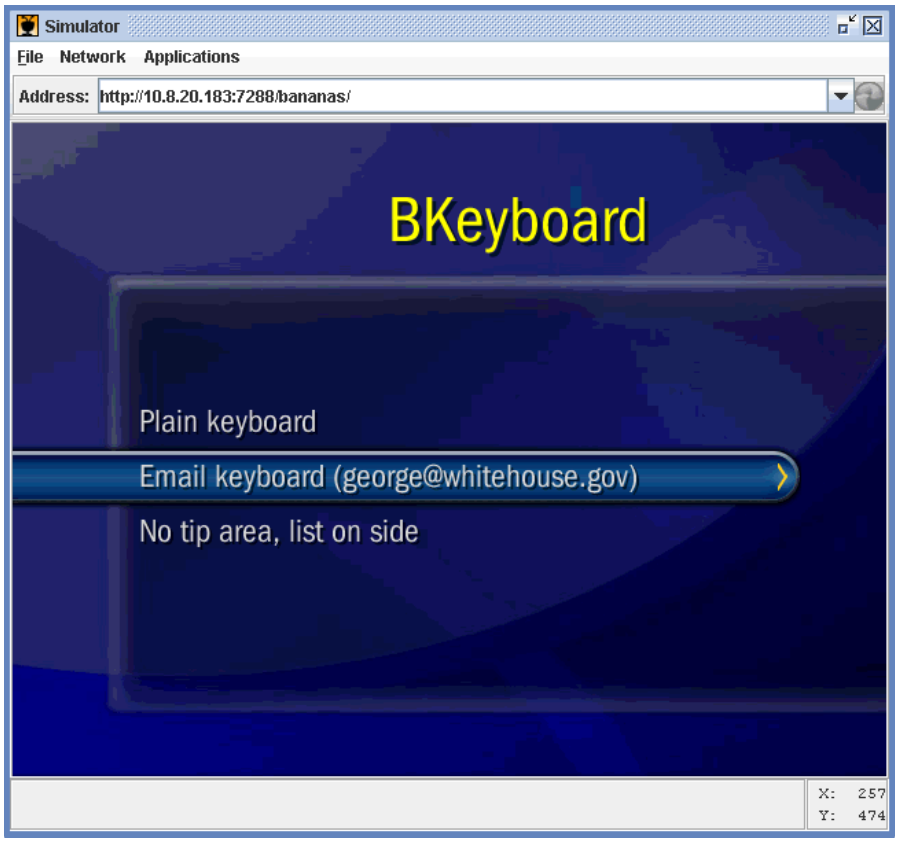
Methods for Entering and Exiting Screens

The *BScreen* class provides *handleEnter()* and *handleExit()* methods that are called when a screen is entered and exited. You can override these methods to implement custom behavior for your screens, such

as starting and stopping any helper threads associated with the screen. If the user calls *push* or *pop* with an *arg*, the *arg* is passed in to *handleEnter()*.

The *isReturn* parameter indicates whether your screen was just popped (=TRUE) or pushed (=FALSE). This mechanism can be used to create screens that accept arguments and return results. For example, in the Bananas sample (*KeyboardScreen.java*), when the user types an e-mail address using the e-mail keyboard screen and then returns to the keyboard menu screen, the text the user typed is passed back to the keyboard menu screen and shown in parentheses on the Email keyboard row (Figure 2-2).

Figure 2-2. Example of passing user data from one screen to the next



Use the *handleExit()* method to stop any threads your screen has created and to perform general cleanup.

Focus

Within each screen, one view contains the current *focus*. When a screen is pushed onto the stack, key events are first sent to the view that contains the default focus. For example, if your screen contains only a list widget, you probably want to call

```
screen.setFocusDefault(list);
```

to ensure that the list will have focus when your screen is pushed.

When a new screen is pushed onto the stack, the old screen's focus is not modified. When the covering screen is popped, the old screen will have the same focus as before. This behavior provides continuity for the user.

If the currently focused view does not handle the event, it is sent to the view's parent.

Changing Focus

Use *screen.setFocus()* to change the screen's focus at any time. When the focus changes, events are sent to the views that are losing and gaining focus. These events propagate up the parent hierarchy, as with all HME events, until *handleFocus()* returns TRUE.

Focus Movement

The user presses arrow keys on the TiVo remote control to navigate within and between screens. Bananas facilitates this behavior. *BScreen* contains default behavior for handling *left, right, up,* and *down* actions. When the screen handles these actions, it attempts to automatically move the focus in the desired direction. Bananas offers two mechanisms for changing focus:

- The system handles focus movement by default if you place whispering arrows to indicate the directions in which the user is allowed to move.
- Your application can monitor key events and explicitly set the focus to the new view you want to move to.

See *"Actions for Whispering Arrows"* and *"Actions"* to learn about actions and how they are generated.

Focus Manager

The Focus Manager (*BFocusManager*) handles focus movement in a fashion similar to that of the TiVo UI. The Focus Manager will try to move the focus in response to the user pressing an arrow key. It finds the view that is closest to the ray pointing from the center of the source view in the direction of the arrow key pressed.

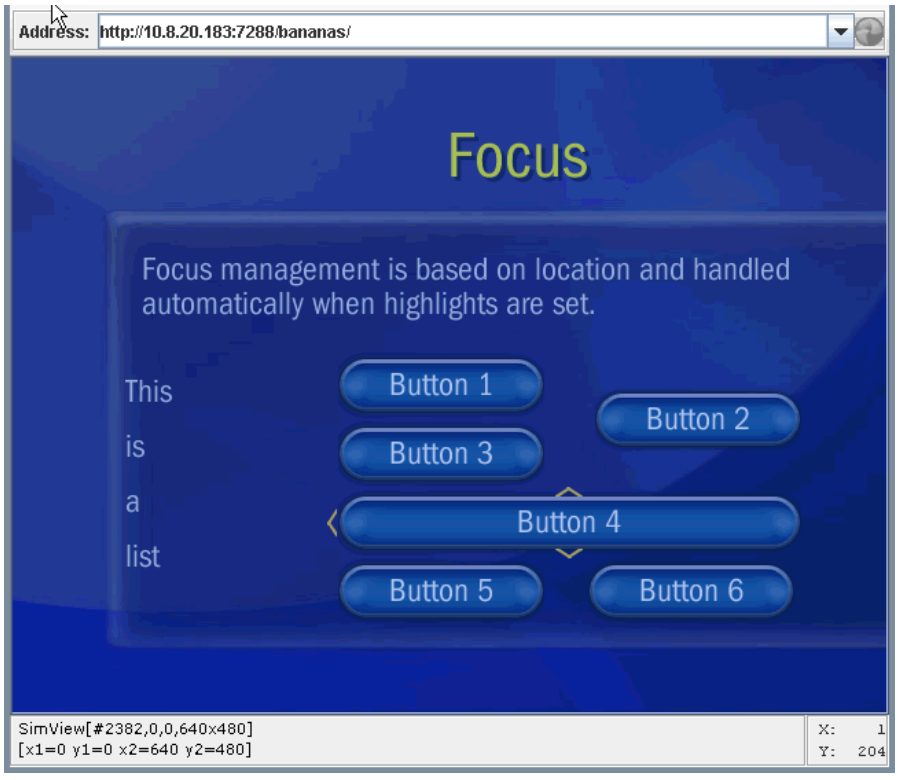
To be a candidate for receiving focus, a view must

- Be visible on the screen
- Be focusable (*BView.setFocusable* is TRUE)

See the Bananas sample application for a demonstration of focus management (Figure 2-3).

Be sure to call *view.setFocusable()* if you want a view to be eligible for auto-focus management. For example, *BButton* calls *setFocusable(true)* in its constructor so that buttons are always candidates for focus. In contrast, *BText* is not focusable by default. *BList* itself is not focusable but contains rows that are focusable.

Figure 2-3. Example of focus management built into the toolkit



Highlights

A *highlight* is a view displayed at a position relative to the currently focused widget. There are three types of Bananas highlights, described in Table 2-1.

Table 2-1 Highlight Objects in Bananas Toolkit

Highlight	Function	Drawn in This Layer
whispering arrows	Indicate the direction in which the user can change focus. Whispering arrows are associated with buttons and are visible only when that view has focus. In Figure 2-3, Button 4 has three whispering arrows.	above
bars	Indicate that a view has focus. Bars are always the same height but can be stretched to any width.	below
page hints	Indicate whether there is more information on the previous or next screen. They imply that the user can hit channel up/down to scroll the page.	above

Highlights are drawn either above or below the view being highlighted. They are associated with the underlying view but are not direct children of it. This makes it possible to place highlights outside the highlighted view. Because highlights are defined independently of the objects they're highlighting, they can be shared by multiple views. For example, all the rows in a list can share the same set of highlights.

Highlights are laid out by assigning them a position. This position is always relative to the view being highlighted, even though the highlight is not a child of the view. For example, a highlight at 0,0 will always draw in the upper left corner of the highlighted view. A highlight at -100,0 will draw to the left of the highlighted view.

Some highlights are associated with the focus. For example, in a list widget, the arrows and bar indicate that a specific row has focus, and when the focus moves, the highlights move too.

The *BHighlight* class is used to create a specific highlight on a widget (for example, one whispering arrow). One widget can have multiple *BHighlight* objects associated with it.

The Bananas Toolkit also contains a low-level framework for managing highlights (the *BHighlights* class). Individual widgets sometimes contain helper methods that facilitate highlight creation.

There are three different ways to add highlights to a view. In order of complexity, they are as follows:

- *Widget-specific:* Some widgets contain helpers for adding highlights. An example of a helper method for adding highlights is *BList.setBarAndArrows()*. This is the easiest way to add highlights:

```
list.setBarAndArrows(BAR_DEFAULT, BAR_DEFAULT, "left",
                     "right");
```

- *Highlight-specific:* The *BHighlights* class also contains helper methods for adding and laying out highlights. An example is the *BHighlights.setWhisperingArrow()* method. This approach provides flexibility but is more difficult than using built-in highlights:

```
//add left arrow to a view; place it in left/top corner
BHighlights h = view.getHighlights();
h.setWhisperingArrow("left", A_LEFT, A_TOP, "left");
```

- *Custom:* For custom highlights, you manually create a *BHighlight* instance and lay it out from scratch. Although complicated, this method offers great flexibility.

Actions for Whispering Arrows

Whispering arrows usually contain an action. When the user presses an arrow key on a view that contains a whispering arrow highlight, that action is fired automatically. For example:

```
BHighlights h = view.getHighlights();

// left arrow should fire "pop" action; right arrow should
// move focus to the right

h.setWhisperingArrow("left", A_LEFT -10, A_CENTER, "pop");
h.setWhisperingArrow("right", A_RIGHT +10, A_CENTER, "right");
```

Sounds

In addition to handling the screen stack, *BApplication* handles the playing of default sounds. It attempts to handle generic sounds for events such as pushing and popping screens and changing focus. Most of the time, the default sound is the sound you'll want to use. If you do not want your application to play a certain sound, however, you can specify that it play a NULL sound for a given action. Make the following call to suppress the sound:

```
view.getBApp.play (NULL);
```

Actions

The Bananas sample program (*SoundsScreen.java*) shows how to play custom sounds by overriding the *handleAction()* method. An action is a type of event and is generated by Bananas or by your application, usually in response to a keypress. As do other events, actions propagate up through the parent hierarchy until they are handled (the *handleAction()* method then returns TRUE).

Custom Sounds

The following sample code shows overriding the sounds played when the user hits one of the four arrow buttons on the remote control.

```
// Play a custom sound when an action is selcted.

public boolean handleAction(BView view, Object action)
{
// Override the default sound for a given action.

   if (action.equals(H_RIGHT)) {
       getBApp().play("speedup3.snd");
   }  else if (action.equals(H_LEFT)) {
       getBApp().play("slowdown1.snd");
   }  else if (action.equals(H_UP)) {
      getBApp().play("thumbsup.snd");
   }  else if (action.equals(H_DOWN)) {
       getBApp().play("thumbsdown.snd");
   }

   return super.handleAction(view, action);
}
```

Default Application Behavior

When Bananas dispatches a key event, the following set of rules is followed to determine what sound is played:

1. In response to a key event, was *BApplication.play()* called? (This function can be called either with a custom sound or with NULL.) If yes, be silent.
   Note: In order for *BApplication* to detect that a sound was played, your application must call *BApplication.play()* directly.
2. Did the focus change or did the screen change? If so, play a sound according to Table 2-2.
3. If none of the above occurred, play *bonk*.

Table 2-2 Default Sounds

Key	Sound
LEFT	Focus change: *updown* Screen change: *pageup*
RIGHT	Focus change: *updown* Screen change: *pagedown*
UP/DOWN	*updown*
SELECT	*select*
THUMBSUP/ DOWN	*thumbsup/down*
CHANNELUP/ DOWN	*pageup/down*

Bananas Widgets

3

This chapter includes:

This chapter describes the four widgets contained in the Bananas User Interface Toolkit

- Button
- Text
- List
- Keyboard

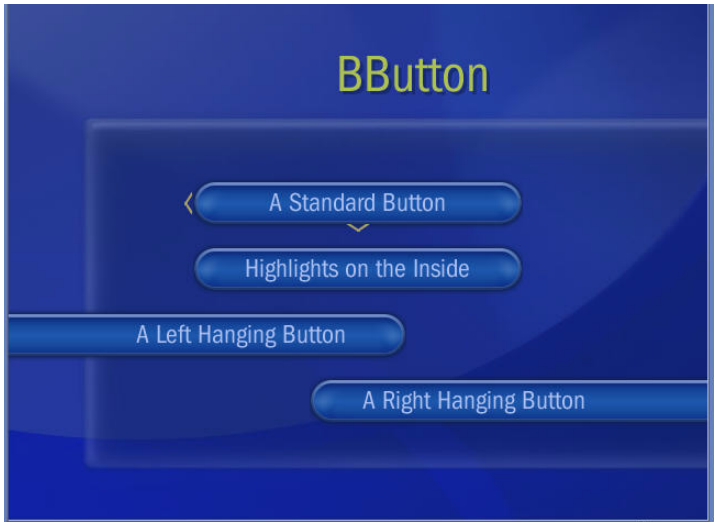
Button

The *BButton* widget provides focus management and event handling for a button. The *BButton* widget consists of these elements:

- Bar (background layer that appears behind the button)
- Highlights (whispering arrows that that provide a hint about where the user can move to next)

This class provides a view with highlights; typically you will also add text to it using either *BText* or *setResource()*. The Bananas sample code shows creating different types of buttons (*ButtonsScreen.java*).

Figure 3-1. Bananas sample buttons



There are four versions of the *setBarAndArrows()* method for this class. The simplest version allows you to specify the action to associate with the left and right whispering arrows. When the user presses the LEFT ARROW or RIGHT ARROW key, the corresponding action is fired:

```
void setBarAndArrow(Object action_left,
                    Object action_right)
```

If the parameter is NULL, the whispering arrow is omitted.

The sample code uses the most complex version:

```
void setBarAndArrows(int bar_left,
                     int bar_right,
                     java.lang.Object action_left,
                     java.lang.Object action_right,
                     java.lang.Object action_up,
                     java.lang.Object action_down,
                     boolean inside)
```

The *bar_left* parameter specifies the indentation for the left side of the button, as follows:

- BAR_DEFAULT: specifies to have the bar touch the left edge of the button
- BAR_HANG: specifies to hang the button off the left side (see the sample)

Similarly, the *bar_right* parameter specifies the alignment for the right side of the button.

The *action_left*, *action_right*, *action_up* and *action_down* parameters refer to the four possible arrows you can associate with a *BButton*. (If the parameter is NULL, the arrow is omitted.) Each arrow can be associated with an action object defined by you, or it can specify a pre-defined action, such as:

- *"pop":* pops your screen
- *"push":* pushes the specified screen
- H_UP: attempts to move the focus up
- H_DOWN: attempts to move the focus down
- H_LEFT: attempts to move the focus to the left
- H_RIGHT: attempts to move the focus to the right

See *"Custom Sounds"* in Chapter 2 for an example of overriding the default behavior of actions.

The Boolean value for the *inside* parameter specifies whether to put the left and right whispering arrows inside (TRUE) or outside (FALSE) the button. The up and down whispering arrows are always outside the button and are not affected by this parameter.

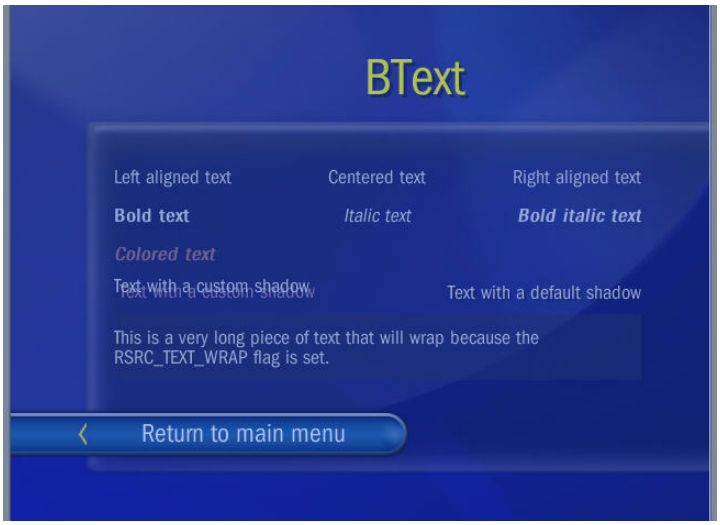
Here is the sample code to create the first button, which has highlights outside the button and a text resource that says "A Standard Button."

```
boolean highlights_inside = false;
BButton stdrdButton = new BButton(getNormal(),
                                  (getWidth()/2)-150, top,
                                  300, 30);
stdrdButton.setBarAndArrows(BAR_DEFAULT, BAR_DEFAULT, "pop",
                            null, null, H_DOWN,
                            highlights_inside);
stdrdButton.setResource(createText("default-24.font",
                        Color.white, "A Standard Button"));
setFocusDefault(stdrdButton);
```

Text

The *BText* widget provides an easy way for you to specify colors, styles, alignment, and shadows for a text object. This widget is a *BView* that displays a text resource. The *value* of the *BText* widget is the text string. The Bananas sample illustrates setting different text attributes (*TextScreen.java*).

Figure 3-2. Sample Bananas text attributes



As this figure shows, the *BText* widget contains methods for specifying:

- Font (*setFont()* method)
- Color (*setColor()* method)
- Shadow and shadow color (*setShadow()* method)

This widget also uses resource flags to set these attributes:

- Alignment (left, right, top, bottom)
- Wrapping

The following resource flags, defined in *IHmeProtocol*, are set with the *BText.setFlags()* method:

- RSRC_HALIGN_LEFT: align the text string to the left
- RSRC_HALIGN_CENTER: center the text string
- RSRC_HALIGN_RIGHT: align the text string to the right
- RSRC_VALIGN_TOP: align the text at the top of the view
- RSRC_VALIGN_BOTTOM: align the text at the bottom of the view
- RSRC_TEXT_WRAP: if the text string is too long to fit, wrap it

For example, here is the sample code to create the centered italic text:

```
BText italicText = new BText(getNormal(), border_left, top,
                             text_width, 30);
italicText.setFlags(IHmeProtocol.RSRC_HALIGN_CENTER);
italicText.setFont("default-18-italic.font");
italicText.setValue("Italic text");
```

Here is the sample code to create text with a custom shadow (custom color and offset by 5 pixels):

```
BText shadowSample2 = new BText(getNormal(), border_left, top,
                                text_width, 50);

shadowSample2.setFlags(IHmeProtocol.RSRC_HALIGN_LEFT |
                       IHmeProtocol.RSRC_VALIGN_TOP);
shadowSample2.setShadow(new Color(255, 200, 200), 5);
shadowSample2.setValue("Text with a custom shadow");
shadowSample2.setFont("default-18.font");
```

List

The *BList* widget provides built-in event handling for the UP, DOWN, CHANNEL UP, and CHANNEL DOWN keys. It also provides focus management and scrolling behavior (if the list does not fit on one screen). The *BList* widget consists of several elements:

- *Rows* (not all may be visible at one time; the list can scroll)
- *Bar* (background layer that appears behind the row that has focus)
- *Highlights* (whispering arrows that provide a hint about where the user can move to next)

Although the bar and arrows are "built-in" to this widget, you do need to call *setBarAndArrows()* in the *BList* constructor to set certain parameters for the highlights.

Creating a List

BList is an abstract class, so you always need to create your own subclass. (The *BList* class does not handle drawing of the list items.) In the constructor for your subclass, call *setBarAndArrows()* to specify the highlights. Then, override the *createRow()* method, which adds a row to the list, and the *setFocus()* method, which sets the focus on the desired view. You can add a single object, an array of objects, or a vector of objects (see *"Performance Tip,"* below).

Performance Tip

If you know the full set of row objects ahead of time, add the objects as an array or vector rather than adding them one at a time. This technique is especially important with large lists because the default behavior turns painting off and on for each call to *list.add(object)*. The following code illustrates the preferred method for adding rows to a list:

```
BList list = new BList(...);
Vector objVec = getObjectVector();
list.add(objVec);
```

The following technique should be used only when the elements are not known at list creation time. It requires considerable processing overhead:

```
//NOT recommended, especially for large lists

BList list = new BList(...);
ArrayList objList = getObjectArray();
Iterator it = objList.iterator();

while (it.hasNext())
{
   Object obj = it.next();
   ...
   list.add(obj);
   ...
}
```

Sample Code

The Bananas Toolkit sample code (*ListsScreen.java*) illustrates how to override *BList* to create three different list styles:

- *Standard list:* The first list is a simple standard list that displays an icon (*star.png*) and a string.
- *Icon list:* A list with an icon that moves to the focused row.
- *Right-aligned list:* A list that hangs from the right.

Figure 3-3. Bananas sample lists

Example 1: Standard List

The first list is a standard list that has an icon (a star) at the beginning of each row. It includes a call to *setBarAndArrows()* in its constructor:

```
static class StandardList extends BList
{
// Constructor

   public StandardList(BView parent, int x, int y, int width,
                       int height, int rowHeight)
   {
      super(parent, x, y, width, height, rowHeight);
      setBarAndArrows(BAR_HANG, BAR_DEFAULT,
                      "pop", H_RIGHT);
   }
.
.
.
}
```

The *setBarAndArrows()* method has the following syntax:

```
void  setBarAndArrows(int bar_left,
                      int bar_right,
                      java.lang.Object action_left,
                      java.lang.Object action_right)
```

The *bar_left* parameter specifies the indentation for the left side of the scroll bar, as follows:

- BAR_DEFAULT: specifies to indent the bar a default amount on the left side
- BAR_HANG: specifies to hang the bar off the left side (see List 1 in the sample)

Similarly, the *bar_right* parameter specifies the indentation for the right side of the scroll bar, as follows:

- BAR_DEFAULT: specifies to indent the bar a default amount on the right side
- BAR_HANG: specifies to hang the bar off the right side (see List 3 in the sample)

The last two parameters in this method (*action_left* and *action_right*) are strings that indicate the actions associated with the left and right highlight arrows. Specifying NULL for one of these parameters omits the highlight arrow. As shown in these examples, possible values are:

- H_LEFT: specifies to move the highlight to the left
- H_RIGHT: specifies to move the highlight to the right
- *"pop":* specifies to pop this screen

When List 1 has focus, pressing LEFT ARROW pops the screen; pressing RIGHT ARROW moves the focus to the right.

The *createRow()* method for this list creates a row with two views: an icon view and a text view:

```
protected void createRow(BView parent, int index)
{
   BView icon = new BView(parent, 20, 0, 34,
                          parent.getHeight());
   icon.setResource("star.png");
   BText text = new BText(parent, 60, 0,
                          parent.getWidth()-70,
                          parent.getHeight());
   text.setFlags(RSRC_HALIGN_LEFT);
   text.setShadow(true);

   // Set the value of the row to be the text that
   // was passed in through add()

   text.setValue(get(index));
}
```

The text in each row is left-aligned (RSRC_HALIGN_LEFT) and has a shadow.

Example 2: List with Icon Showing Focus

The second list overrides the *handleFocus()* method to add an icon to the focused row. It changes the location of *iconView* (the musical note) depending on the focus. Whenever the focused row changes, the *handleFocus()* method is called twice, once for the row that loses focus (to remove the icon) and once for the row that gains the focus (to add the icon). This list is longer than one screen and illustrates the built-in *BList* scrolling mechanism. Call *BHighlights.setPageHint()* to add the Page Up/Page Down icons to scrolling lists.

Here is the *handleFocus()* method for List 2:

```
public boolean handleFocus(boolean isGained, BView gained,
                           BView lost)
{

// We really only care when we gain focus and when we do, we
// set the icon to be on the row that just gained focus.
// If we gain focus and the previous focus was not this list,
// then we jump the icon directly to the new focused item.
// Otherwise, we animate it at the same speed as the bar so
// that it looks like it is part of the bar.

   if (isGained && gained.getParent() == this) {
      if (lost.getParent() == this) {
         iconView.setLocation(30, gained.getY()+3,
                              this.getResource("*100"));
      } else {

      // gaining focus from another widget

      iconView.setLocation(30, gained.getY()+3);
      }
   }
   iconView.setVisible(true);

   return super.handleFocus(isGained, gained, lost);
}
```

Example 3: Right-Aligned List

The third list overrides *createRow()* to make the rows right-aligned. Note that it has only a left highlight arrow, so it handles only the LEFT ARROW key press.

Keyboard

The *BKeyboard* widget provides an onscreen keyboard that allows the user to spell out words on the television using the TiVo remote control. Your application can monitor the keyboard and respond to changes as the user enters characters in the text entry box above the keyboard. This widget thus provides basic keyboard functionality for your application without requiring additional hardware in the user's living room.

Types of Standard Keyboards

There are two standard types of Bananas keyboards: the plain keyboard (Figure 3-4) and the e-mail keyboard (Figure 3-5). The e-mail keyboard has special characters used to enter e-mail addresses, such as the @ character, the suffixes *.com*, *.org*, and *.net*, and several punctuation marks (*underscore* and *period*).

Figure 3-4. Plain version of standard keyboard

Both the plain and e-mail keyboards have three versions: lowercase, uppercase, and symbol. The standard keyboards are named as follows:

STANDARD_KEYBOARD_LOWERCASE
STANDARD_KEYBOARD_UPPERCASE
STANDARD_KEYBOARD_SYMBOL
STANDARD_KEYBOARD_EMAIL_LOWERCASE
STANDARD_KEYBOARD_EMAIL_UPPERCASE
STANDARD_KEYBOARD_EMAIL_SYMBOL

Figure 3-4 shows the lowercase version of the plain keyboard. Figure 3-5 shows the uppercase version of the e-mail keyboard. Figure 3-6 shows the symbol version of the plain keyboard.

Figure 3-5. e-mail version of standard keyboard

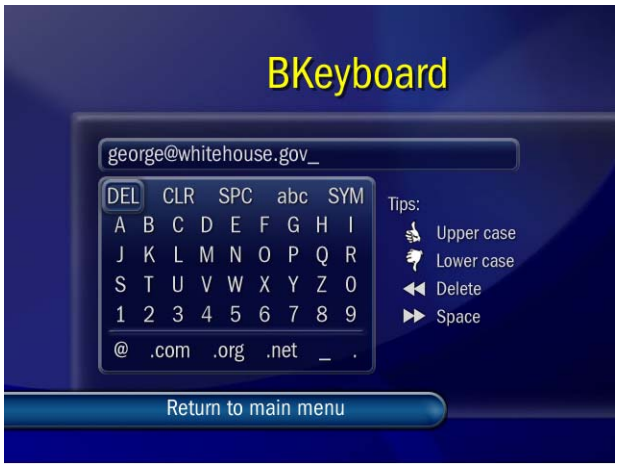
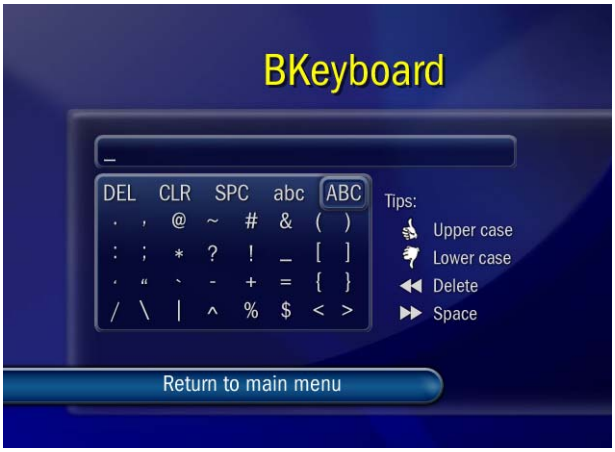


Figure 3-6. Symbol version of standard keyboard



Standard Keyboard Behaviors and Options

As the user selects characters on the keyboard widget, they appear in the text entry box above the keyboard. After the user has entered one or more characters, the CLR button is visible. If the user presses CLR, that button toggles to UNDO (so that the user can retrieve the deleted text if desired).

Also built in to the standard keyboards is an optional *tips area*, which provides hints to the user on standard behavior for buttons on the remote. Both Figure 3-4 and Figure 3-5 include the Tips, which prompt the user to

- Switch to the uppercase version of this keyboard by pressing Thumbs Up on the remote
- Switch to the lowercase version of this keyboard by pressing Thumbs Down on the remote.
- Press the Back button to delete a character
- Press the Forward button to add a space

The number keys and the CLEAR key on the remote can be used in place of the corresponding keyboard keys to enter and clear characters.

The width of the text entry box is customizable. The keyboard itself and Tips area are of fixed size and layout.

Sample Code

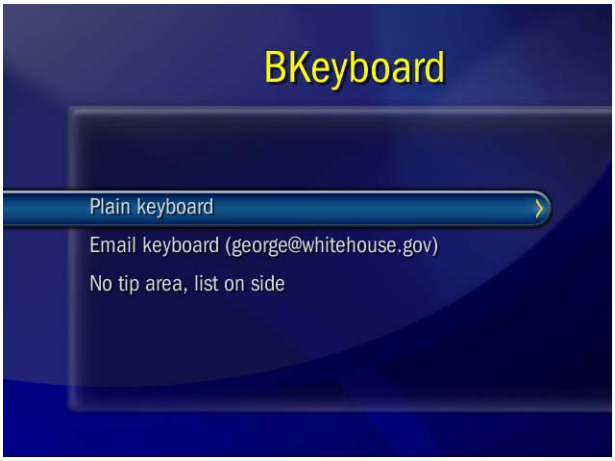
The following section discusses how the Bananas sample code (*KeyboardScreen.java*) creates three different keyboard widgets.

Creating a Keyboard Widget

The Bananas Toolkit sample creates three keyboards (see Figure 3-7):

- A plain keyboard (lowercase, with Tips enabled)
- An e-mail keyboard (lowercase, with Tips enabled)
- A plain keyboard with a scrolling word list that updates as the user enters text (no Tips)

Figure 3-7. Menu choices for BKeyboard sample



The *BKeyboard* class has three constructors. Typically, you will use one of the first two constructors listed here. Use the following simple constructor to create a plain standard lowercase keyboard, with the Tips area visible:

```
public BKeyboard(BView parent,   //parent view
                 int x, //x position relative to parent
                 int y, //y position relative to parent
                 int width,      //width of widget
                 int height)     //height of widget
```

This second constructor allows you to specify which standard keyboard to use (either PLAIN_KEYBOARD or EMAIL_KEYBOARD) and to choose whether to display the Tips area or not:

```
public BKeyboard(BView parent,
                 int x,
                 int y,
                 int width,
                 int height,
                 int KeyboardType, //one of two types
                 boolean tips)    //include Tips area?
```

The third constructor allows you to specify any keyboard object, to choose whether to display the Tips area, to specify a custom width for the text entry box, and to choose whether to display the keyboard itself.

```
public BKeyboard(BView parent,
                 int x,
                 int y,
                 int width,
                 int height,
                 Keyboard keyboard,
                 boolean tips,
                 int textEntryWidth,
                 boolean visible)
```

Plain Keyboard

The first keyboard in the *BKeyboard* sample code (Figure 3-4) uses the simplest constructor to create a default keyboard. (This constructor creates the plain standard lowercase keyboard, includes the Tips area, and creates a text entry box of standard width.) Before you construct the keyboard, you need to call the helper function *getKeyboardSize()* to obtain the size of the keyboard, which is passed in to the constructor's *width* and *height* parameters.

This code shows how the plain keyboard is created:

```
Point p = BKeyboard.getKeyboardSize(null,
                 BKeyboard.INPUT_WIDTH_SAME_AS_WIDGET,
                 true);

kb = new BKeyboard(getNormal(), 100, 140, p.x, p.y);
```

e-Mail Keyboard

The second keyboard in the *BKeyboard* sample code uses the constructor that allows you to pass in a keyboard type and to specify whether Tips are visible. This example creates a keyboard of type EMAIL_KEYBOARD. As described above, you first need to call the helper function *getKeyboardSize()* to obtain the size of the keyboard, which is passed in to the constructor's *width* and *height* parameters.

This code shows how the e-mail keyboard is created:

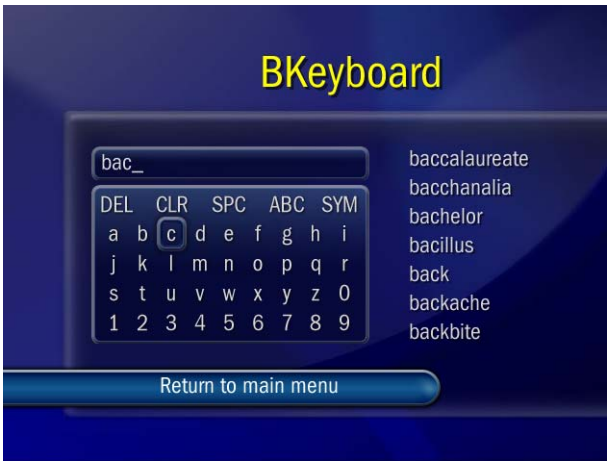
```
Point p = BKeyboard.getKeyboardSize(
                                BKeyboard.EMAIL_KEYBOARD,
                                true);

kb = new BKeyboard(getNormal(), 100, 140, p.x, p.y,
                   BKeyboard.EMAIL_KEYBOARD, true);
```

Plain Keyboard with Scrolling Text Area

The third keyboard in the *BKeyboard* sample code uses the most detailed constructor, which allows you to pass in any keyboard object, to specify the width of the text entry box, and to specify whether Tips are visible. (See Figure 3-8).

Figure 3-8. Keyboard with scrolling text area



This constructor requires use of this form of the *getKeyboardSize()* function:

```
static public Point getKeyboardSize(Keyboard keyboard,
                                    boolean tips, int textEntryWidth)
```

This code shows how the third keyboard is created:

```
Point p =
BKeyboard.getKeyboardSize(BKeyboard.getStandardKeyboard
                    (BKeyboard.STANDARD_KEYBOARD_LOWERCASE),
                    BKeyboard.INPUT_WIDTH_SAME_AS_WIDGET,
                    false);

kb = new BKeyboard(getNormal(), 100, 150, p.x, p.y,
                    BKeyboard.getStandardKeyboard
                    (BKeyboard.STANDARD_KEYBOARD_LOWERCASE),
                    false,
                    BKeyboard.INPUT_WIDTH_SAME_AS_WIDGET,
                    true);
```

Handling Events

In the Bananas Toolkit sample, the third keyboard monitors and responds to keyboard events. The *handleEvent()* method in the

sample monitors the characters typed into the text entry box and updates the scrolling word list at the right accordingly:

```
//handle a change in the value of the BKeyboard by
//updating the related text field

public boolean handleEvent(HmeEvent event) {
   if (event instanceof KeyboardEvent) {
      update(((KeyboardEvent)event).getValue());
   }
   return super.handleEvent(event);
}
```

In this example, the *getValue()* method returns a string that is the current value displayed in the text entry box of the *BKeyboard* widget.

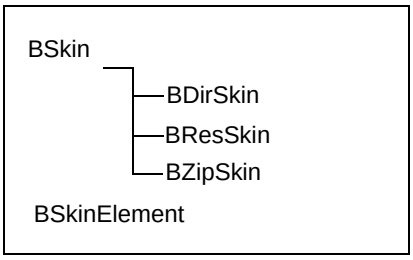
Customization

4

This chapter includes:

This chapter discusses how to customize the look and feel of your application by creating your own skins. The Bananas Toolkit includes classes that allow you to easily create a custom graphical appearance, or *skin*, for your application. Figure 4-1 shows the class tree for the Bananas skin classes.

Figure 4-1. Bananas Toolkit classes used for skinning your application



Sample Code

The *SkinSample.java* sample code illustrates how to add a custom skin to a Bananas application. This sample includes three different skins, each packaged using a different technique. You can edit the launcher file to use one of the three skins (*skin-charcoal*, *skin-rainbow*, and *skin-steel*). *SkinSample.java* is derived from the basic Bananas sample program (*BananasSample.java*) and simply adds the skinning information. (Note that the sample uses a factory because it switches skins for demonstration purposes. Your application will probably not need to use a factory for skinning.)

Packaging the Images

Using the *BSkin* subclasses, you create the images for your skin and package them in one of three ways:

- Place the image files in a directory (use the *BDirSkin* class)
- Include the image files as a resource packaged with the application (*BResSkin* class)
- Place the image files in a directory and then zip up the directory into a single zip file (*BZipSkin* class)

The easiest way to test new skins is to use the *BDirSkin* class. Once you're satisfied with the results, you can use one of the other two packaging techniques.

Standard Elements

Using one of the *BSkin* subclasses, you can create your own graphics for the elements shown in Table 4-1.

Table 4-1 Standard Bananas skin elements

Element	Size in Pixels
bar.png	640 x 48
down.png	20 x 7
left.png	8 x 20
pagedown.png	14 x 26
pageup.png	14 x 26
right.png	8 x 20
up.png	20 x 7

If you want to use different sizes for the skin elements listed in Table 4-1, you'll need to create your own *BSkin* subclass (not described here; use the standard *BSkin* subclasses as models for your work).

Loading Application-Specific Images for a Skin

You can also use the *BSkin* subclasses (*BDirSkin, BResSkin, BZipSkin*) to add application-specific user interface elements, such as icons, backgrounds, or any other PNG resource that you want to bundle in a skin package. This facility makes it easy to provide multiple presentations for a single application.

Here is the sample code that loads a background image directly from the skin into the *below* layer of the application:

```
BSkin skin = getSkin();
getBelow().setResource(skin.get
                       ("background-main").getResource());
push(new MainMenuScreen(this), TRANSITION_NONE);
```

You can use this same technique to load other application-specific elements from your skin.

Basic Steps

The basic steps for adding a custom skin to your application are as follows:

1. Create the image files for the skin elements, naming them appropriately.
2. Package the images using one of the three techniques (directory, resource, or zip file).
3. Construct the skin class and pass in your elements. The three constructors are as follows:

```
BDirSkin(BApplication app, java.lang.String skinDir)
BResSkin(BApplication app, java.lang.String resPath)
BZipSkin(BApplication app, java.lang.String zipFile)
```

4. In your application's *init()* method, call *BApplication.setSkin()* and specify the skin that contains your resources.
5. Load other application-specific images, such as a background, into the skin.

Index

Index