Advanced Java Application Development for the BlackBerry Smartphone

BlackBerry Academic Program
Prior to subscribing for, installing or using any Third Party Products and Services it is your responsibility to ensure that your airtime service provider has agreed to support all of their features. Some airtime service providers may not offer Internet browsing functionality with a subscription to BlackBerry® Internet Service. Check with your service provider for availability, roaming arrangements, service plans and features. Installation or use of Third Party Products and Services with RIM’s products and services may require one or more patent, trademark, copyright or other licenses in order to avoid infringement or violation of third party rights. You are solely responsible for determining whether to use, Third Party Products and Services and if any third party licenses are required to do so. If required you are responsible for acquiring them. You should not install or use Third Party Products and Services until all necessary licenses have been acquired. Any Third Party Products and Services that are provided with RIM’s products and services are provided as a convenience to you and are provided “AS IS” with no express or implied conditions, endorsements, guarantees, representations or warranties of any kind by RIM and RIM assumes no liability whatsoever, in relation thereto. Your use of Third Party Products and Services shall be governed by and subject to you agreeing to the terms of separate licenses and other agreements applicable thereto with third parties, except to the extent expressly covered by a license or other agreement with RIM.

Certain features outlined in this documentation require a minimum version of BlackBerry® Enterprise Server, BlackBerry® Desktop Software, BlackBerry® Device Software and/or BlackBerry Handheld Software and may require additional development or Third Party Products and Services for access to corporate applications.

Certain products mentioned in this documentation include software developed by the Apache Software Foundation (http://www.apache.org/) and/or licensed pursuant to Apache License, Version 2.0 (http://www.apache.org/licenses/). For more information, see the NOTICE.txt file included with the software. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

The terms of use of any RIM product or service are set out in a separate license or other agreement with RIM applicable thereto. NOTHING IN THIS DOCUMENTATION IS INTENDED TO SUPERSEDE ANY EXPRESS WRITTEN AGREEMENTS OR WARRANTIES PROVIDED BY RIM FOR PORTIONS OF ANY RIM PRODUCT OR SERVICE OTHER THAN THIS DOCUMENTATION.
Chapter 4
Advanced BlackBerry user interface development

Objectives

- Discuss the BlackBerry® UI components
- Describe how to create custom Managers
- Describe how to create custom Fields
- Describe how to draw directly on the screen
- Describe how to use SVG, citing specific BlackBerry examples

This chapter outlines the structure of the BlackBerry smartphone UI components, and describes how they work together. This chapter describes how you can customize Managers and Fields to create a dynamic and easy to use interface for the smartphone user, while taking into consideration their unique requirements. This chapter also describes how to create and use SVG to enhance the BlackBerry smartphone UI experience.

Design principles for BlackBerry smartphones

Applications designed for BlackBerry smartphones must provide a balance between the best possible user experience and a long battery life. When you design your BlackBerry smartphone application, consider that BlackBerry smartphones differ from computers in many ways, and have the following limitations:

- have a small screen size
  - display a limited number of characters
  - display only one screen at a time
- have limited processor speeds
- have limited available memory
BlackBerry smartphone users use applications differently than they use applications on a computer. On BlackBerry smartphones, users expect to find information quickly. For example, a CRM system can provide a massive amount of information, but users require only a small amount of that information at one time.

Design your applications for BlackBerry smartphones to be as consistent as possible with the design of other BlackBerry smartphone applications.

Consider the following guidelines:

- Use or extend existing UI components where possible so that your application can inherit the default behavior of the component.
- Follow the standard navigation model as closely as possible so that BlackBerry smartphone users can make full use of the keyboard, trackball, and touchscreen.
- Make all actions available from the menu. Verify that the actions available in the menu are relevant to users’ current context.

When you design your application, also consider the following guidelines:

- Stay focused on the users’ immediate task. Simplify data selection and presentation to display only the information that users need.
- Display information in a way that makes effective use of the small screen.

Before you design your application, consider using the core applications on the BlackBerry smartphone or the BlackBerry Smartphone Simulator to learn more about the navigation model and best practices for designing your application UI.
BlackBerry user interface

A graphical UI manages the relationship between the application and its user, while providing as much information as possible to the user in an easy to read manner using available screen space. An application that is easy to use lets a user perform tasks in an efficient manner, whereas a poor design can easily frustrate that person and prevent them from using the application. A graphical UI must feel intuitive and familiar to a user. BlackBerry APIs can help BlackBerry application developers create familiar graphical UIs by providing many common screen elements for use and reuse in an application.

Topics within this section include the following:
- Screens
- Managers
- Fields

Use BlackBerry UI APIs to create BlackBerry Java Application UIs. The BlackBerry UI APIs are a library of UI components that are designed to provide default layouts and behaviors that are consistent with the core BlackBerry smartphone applications.

- Screen components provide a standard screen layout, a default menu, and standard behavior when the BlackBerry smartphone user presses the Escape key or clicks the trackwheel or trackball.
- Field components provide standard UI elements for date selection, options, check boxes, lists, text fields and labels, and progress bar controls.
- Layout managers allow you to arrange components on a BlackBerry smartphone screen in standard ways, such as horizontally, vertically, or in a left-to-right flow.

You can use the BlackBerry UI APIs to create UIs that include tables, grids, and other specialized features. The BlackBerry® Java® Development Environment uses a standard Java® event model to receive and respond to specific types of events. Applications can receive and respond to BlackBerry smartphone user events, such as when the BlackBerry smartphone user clicks the trackwheel, clicks the trackball, or types on the keyboard, and to system events, such as global alerts, real-time clock changes, and USB port connections.
This illustration shows the BlackBerry GUI is a three-level hierarchical structure in its most basic form.

![Diagram of BlackBerry GUI structure]

**Figure 5.1 The basic BlackBerry GUI three-level hierarchical structure**

The **Screen class** (net.rim.device.api.ui.Screen) is the starting point for the UI. Only one screen is displayed on a BlackBerry smartphone at a time. Screens appear on the smartphone by pushing and popping them from the display stack. The screen at the top of the display stack is the one shown to the user. A screen can only exist once in the display stack, but you can push or pop the display stack when necessary. Common subclasses of the Screen class are contained in the net.rim.device.api.ui.container package.

The **Manager class** (net.rim.device.api.ui.Manager) manages the layout and interaction between field objects added to it. A Manager is responsible for the location and layout of its fields, as well as scrolling and focus change. A screen must have at least one manager to manage its fields, even if the default constructor provides the manager. Common subclasses of Manager are located in the net.rim.device.api.ui.container package.

The **Field class** provides key functionality for all field components. A field represents a region that a manager contains and that the manager uses to display output or handle input. The content varies depending on the field type and restrictions placed on the field. The net.rim.device.api.ui.component package contains a library of prebuilt UI components and controls for constructing BlackBerry applications.
Screens

A BlackBerry application requires at least one screen to display information. This illustration shows the various extensions of the Screen class available in the BlackBerry APIs.

There are five types of screens.

<table>
<thead>
<tr>
<th>Screen type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FullScreen</td>
<td>A screen that contains a single VerticalFieldManager as its delegate manager.</td>
</tr>
<tr>
<td>MainScreen</td>
<td>Extends FullScreen and adds some additional functionality common to many BlackBerry smartphone applications. MainScreen also allows for a title area at the top of the screen followed by a SeparatorField. You can scroll through the main area of the screen because this screen contains a VerticalFieldManager as its delegate manager that is inherited from FullScreen. MainScreen also contains an implementation of the makeMenu method to handle menu creation.</td>
</tr>
<tr>
<td>PopupScreen</td>
<td>A screen that provides Dialog or Status screen features.</td>
</tr>
<tr>
<td>Dialog</td>
<td>A screen that provides the behavior of a dialog box that is used to present information to a user and accept input. The alert, ask, inform, and doModal methods of the Dialog class display the Dialog and block (wait) for user input. The show method of the Dialog class does not block, therefore program execution continues after the Dialog screen appears.</td>
</tr>
</tbody>
</table>

*Figure 5.2 Screen class extensions available in the BlackBerry API set*
One of the most commonly used screen classes for a BlackBerry application is MainScreen, which provides functionality to easily add a title and menu functions. The following code sample demonstrates use of these methods:

```java
//Instantiate MainScreen
MainScreen theScreen = new MainScreen();

//Instantiate a LabelField
LabelField theTitle = new LabelField("My application!");

//Set the title of theScreen
theScreen.setTitle(theTitle);

//Instantiate a MenuItem
MenuItem clickMe = new MenuItem("Click Me!", 40, 40) {
    //The run command will be called when the user
    //clicks on the MenuItem
    public void run()
    {
        Dialog.alert("You clicked me!");
    }
};

//Add clickMe to theScreen
theScreen.addMenuItem(clickMe);

//Push the screen onto the display stack.
pushScreen(theScreen);
```

<table>
<thead>
<tr>
<th>Screen type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>A screen that provides the features of a dialog box but does not accept user input or block. The Status screen is designed to show that an action has taken place. Status screens are set to display for a fixed time or until the user clicks the track-wheel, or presses the Escape key or Space key.</td>
</tr>
</tbody>
</table>
A screen that has been pushed to the top of the stack is under the control of the main event thread. This means that only the event thread can make changes to the screen. An example of this is adding a new LabelField to the screen.

```java
thescreen.add(new LabelField("Hello there.");
```

You can run the preceding add method from the main event thread. An `IllegalStateException` is thrown if a thread other than the main event thread attempts to execute this line of code. The solution is to place the screen change in the screen event queue or obtain the event lock.

You can update a screen from outside of the main event thread by calling the `invokeLater()` or `invokeAndWait()` methods respectively. The main difference between `invokeLater()` and `invokeAndWait()` is that `invokeAndWait()` blocks until the code within it is executed, while `invokeLater()` does not block. Do not use either to perform lengthy operations.

If you call `invokeLater()` repeatedly in rapid succession the event queue can overflow and result in an exception. Only use these methods when necessary and only for the least amount of time to update the screen. Write calculations so that they are performed outside of these methods. When you write code on the event thread, do not allow it to block or execute for long periods of time because the system cannot dispatch messages. The event queue can overflow and cause an exception to be thrown.

The following code sample illustrates the use of `invokeLater()`:

```java
UiApplication.getUiApplication().invokeLater(new Runnable() {

    public void run()
    {
        //Perform screen changes here.

        //Calling invalidate() on your screen forces the paint
        //method to be called.
        screenName.invalidate();
    }
});
```
Managers

The next level beneath Screen in the BlackBerry UI hierarchy is the Manager class. Managers are responsible for vertical or horizontal scrolling, or both, and the positioning and layout of fields. Fields are added to managers, which place the fields in an appropriate area on the BlackBerry smartphone screen. The BlackBerry API contains a set of managers that extend from the Manager class and provide a layout mechanism for common screen designs.

<table>
<thead>
<tr>
<th>Manager type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HorizontalFieldManager</td>
<td>Lays out fields from left to right in a single row. This manager can provide horizontal scrolling for fields that do not fit on the BlackBerry smartphone screen as well as vertical scrolling for fields that are taller than the screen.</td>
</tr>
<tr>
<td>VerticalFieldManager</td>
<td>Lays out fields in a single vertical column. This manager can provide vertical scrolling for fields that do not fit on the BlackBerry smartphone screen as well as horizontal scrolling for fields that are wider than screen.</td>
</tr>
<tr>
<td>FlowFieldManager</td>
<td>Lays out fields in a horizontal then vertical flow. Fields are positioned from left to right, and any fields that do not fit within the allotted horizontal space are placed on the following line, starting from the left. The FlowFieldManager also supports horizontal and vertical scrolling.</td>
</tr>
<tr>
<td>DialogFieldManager</td>
<td>Handles an icon, a message, and a special area which can hold a list of user-specified custom fields. A VerticalFieldManager is used to layout the fields in the user area. It lays out its icon in the top left corner, and its message label in the top left corner.</td>
</tr>
</tbody>
</table>
| GridFieldManager          | Created with a fixed number of rows and columns. You can add fields to the grid either in the next empty cell with the `#add()` method or insert at a specific row and column location using the `#insert()` method.  
You can specify row and column heights and widths as fixed sizes or they can be automatically sized with the available space divided evenly among the rows and columns.  
The grid scrolls vertically if the heights of the rows exceed the visible height of the grid, and scroll horizontally if the widths of the columns exceed the visible width of the grid. |
You can arrange components on a screen using BlackBerry API layout managers. The following four classes extend the Manager class to provide predefined layout managers:

- VerticalFieldManager
- HorizontalFieldManager
- FlowFieldManager
- DialogFieldManager

1. Import the following classes:

```java
net.rim.device.api.ui.Manager
net.rim.device.api.ui.container.HorizontalFieldManager
net.rim.device.api.ui.component.ButtonField
```

2. Create an instance of a `HorizontalFieldManager`.

```java
HorizontalFieldManager_fieldManagerBottom = new HorizontalFieldManager();
```

3. Invoke the `add()` method to add the `HorizontalFieldManager` to a screen.

```java
myScreen.add(_fieldManagerBottom);Create an instance of a ButtonField. ButtonField mySubmitButton = new ButtonField("Submit");
```

4. Add the `ButtonField` to the `HorizontalFieldManager`.

```java
_fieldManagerBottom.add(mySubmitButton);
```

Managers can contain other managers, which is known as nesting. You can nest managers to create enhanced layout styles, such as a column or table layout. Refer to Appendix A for a sample of code that illustrates the use of nested managers.

Managers can contain other managers, which is known as nesting. You can nest managers to create enhanced layout styles, such as a column or table layout. Refer to Appendix A for a sample of code that illustrates the use of nested managers.

You can use managers to divide the screen into three zones: top, middle, and bottom. You can divide the screen in as many parts as you want in your application, or not, by using vertical and horizontal managers. These managers align all the added components into a single column or row depending on their orientation.
The following code sample shows you how to use the top and bottom zones as horizontal managers and a middle zone as a vertical manager.

After you create these managers, you must add them in the order you want to display them on the screen.

```java
HorizontalFieldManager _fieldManagerTop;
VerticalFieldManager _fieldManagerMiddle;
HorizontalFieldManager _fieldManagerBottom;

_fieldManagerTop = new HorizontalFieldManager();
_fieldManagerMiddle = new VerticalFieldManager();
_fieldManagerBottom = new HorizontalFieldManager();
add(_fieldManagerTop);
add(new SeparatorField());
add(_fieldManagerMiddle);
add(new SeparatorField());
add(_fieldManagerBottom);
add(new SeparatorField());
```

The screen itself has a vertical manager built in.
### Fields

A field represents a region contained by a manager. All fields are derived from the base Field class. The BlackBerry API contains many specialized fields.

<table>
<thead>
<tr>
<th>Field type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ActiveAutoTextEditField</strong></td>
<td>A field that uses a supplied set of string patterns to scan through simple text and pick out active regions. These active regions are identified with underlining and are designed to supply additional menu items if a user clicks them. This field supports use of the BlackBerry smartphone AutoText dictionary and abides by any AutoText replacement rules. When you enter a word that is in the AutoText database for the current locale, this field replaces that word, including any effects of macro expansion. Press the Backspace key when the cursor is on the most recent autotext expansion to reverse all effects of the AutoText substitution.</td>
</tr>
<tr>
<td><strong>ActiveRichTextField</strong></td>
<td>A field that uses a supplied set of string patterns to scan through a simple text string and pick out active regions. These active regions are identified with underlining and supply additional menu items if the user clicks them. This field also supports the formatting of a RichTextField.</td>
</tr>
<tr>
<td><strong>AutoTextEditField</strong></td>
<td>This field supports the use of the AutoText dictionary on the BlackBerry smartphone and abides by any AutoText replacement rules. When you enter a word that is in the autotext database for the current locale, this field replaces it, including any effects of macro expansion. Press the Backspace key when the cursor is on the most recent autotext expansion to reverse all effects of the autotext substitution.</td>
</tr>
<tr>
<td><strong>BasicEditField</strong></td>
<td>An editable text field that can contain a label. The label appears to the left of the editable area. BasicEditField does not support any text formatting options.</td>
</tr>
<tr>
<td><strong>BitmapField</strong></td>
<td>A field that displays a bitmap.</td>
</tr>
<tr>
<td><strong>ButtonField</strong></td>
<td>A clickable button that can contain a text label.</td>
</tr>
<tr>
<td><strong>CheckboxField</strong></td>
<td>This field consists of a text label and a check box. The check box appears to the right of the text label.</td>
</tr>
<tr>
<td><strong>Field type</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ChoiceField</td>
<td>This field consists of a text label and a choice selector/drop-down list. When you click the choice selector or the Change Option menu item, the choice selector expands to display a list of choices. You can also press the space bar or hold the Alt key while you roll the trackwheel to change choices. Text in the drop-down list is truncated if it does not fit within the available width. ChoiceField is an abstract class and cannot be instantiated directly.</td>
</tr>
<tr>
<td>DateField</td>
<td>Displays a text label followed by a formatted date value.</td>
</tr>
<tr>
<td>EditField</td>
<td>An extension of BasicEditField. The user can use EditField to enter special characters by holding a key while rolling the trackwheel. This method is common for locales that use characters with accents.</td>
</tr>
<tr>
<td>EmailAddressEditField</td>
<td>An extension of EditField. EmailAddressField is designed to only allow characters to be entered that are valid for an email address.</td>
</tr>
<tr>
<td>GaugeField</td>
<td>A horizontal bar that can be used for either numerical selection or as a progress indicator.</td>
</tr>
<tr>
<td>LabelField</td>
<td>A label of unformatted text.</td>
</tr>
<tr>
<td>ListField</td>
<td>Displays rows of selectable items. Lists support prefix searching by default where pressing a character selects the next item starting with the character. ListFields must register a ListFieldCallback object to handle repaint tasks. When this field must display a particular item in its list, it invokes the appropriate methods on the registered callback object.</td>
</tr>
<tr>
<td>MapField</td>
<td>Renders a map using the mapping service provided by BlackBerry Maps.</td>
</tr>
<tr>
<td>NullField</td>
<td>A field of no size that can provide specialized focus changing.</td>
</tr>
<tr>
<td>NumericChoiceField</td>
<td>An extension of ChoiceField that supports the selection of a range of numbers. Keep the number of values in this list to fewer than 20.</td>
</tr>
<tr>
<td>ObjectChoiceField</td>
<td>An extension of the ChoiceField that supports a list of objects. All contained objects must support the toString() method.</td>
</tr>
<tr>
<td>Field type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ObjectListField</td>
<td>An extension of ListField that displays a list of objects. All contained objects must support the toString() method.</td>
</tr>
<tr>
<td>PasswordEditField</td>
<td>An extension of an EditField where entered data appears as a mask of asterisks (*) equal to the number of characters the user enters.</td>
</tr>
<tr>
<td>PictureScrollField</td>
<td>A slider component that draws a row of images which can be scrolled from side-to-side using the track-ball or touch gestures. The images slide horizontally to align the focus image in a vertically centered position. The images decelerate as they approach their new position to give an animated effect. There are also several configurable effects to highlight the focus image.</td>
</tr>
<tr>
<td>RadioButtonField</td>
<td>This field consists of a text label and a radio-button control. The radio-button control appears to the right of a text label. These fields can be grouped using a RadioButtonGroup where only one field may be selected at a time.</td>
</tr>
<tr>
<td>RichTextField</td>
<td>A read-only text field to show text in a variety of fonts and formatting styles (such as bold, italics, or underlining).</td>
</tr>
<tr>
<td>SeparatorField</td>
<td>A field that displays a horizontal line across its width.</td>
</tr>
<tr>
<td>SpinBoxField</td>
<td>A user interface component for selecting a single item from a list where the user can spin through the various choices. This component is useful when the candidates for selection are sequential e.g. months, numbers, or a list in alphabetical order. This particular class is abstract. There are subclasses that implement different types of spin boxes, for example, spin boxes for text or bitmaps. The respective classes should actually be instantiated. If none of the provided implementations solve your use cases, you can extend this class to render the spin box to your liking.</td>
</tr>
<tr>
<td>TreeField</td>
<td>A field that displays a tree structure.</td>
</tr>
<tr>
<td>TextSpinBoxField</td>
<td>A SpinBoxField that renders text. Users of this class construct an instance with an array of objects to display. Each object toString method is used to retrieve the string to render.</td>
</tr>
</tbody>
</table>
You must add a field to a manager for the user to see the field. To add a field to a manager, call the add method of a manager, and then pass the mention to the field you want to add. You can only add fields to one manager at a time but you can remove and add fields again during their lifespan. The following sample code illustrates how to add a `LabelField` to a `VerticalFieldManager`.

```java
LabelField myField = new LabelField("Hello world!");
VerticalFieldManager myManager = new VerticalFieldManager();
myManager.add(myField);
```

By making use of screens, managers, and fields, you can create an application with a familiar look and feel. Extend any of these classes to add a custom look or function to your application. Use the graphics class methods and draw directly to a screen, manager, or field to achieve even further customization.
1. Which of the following are valid screen types?
   A. Drop-down
   B. PupupScreen
   C. MainScreen
   D. Split

2. User interface titles and menu functions are typically added to which screen class?

3. Beyond the use of the field class, which method can you use to achieve additional screen customization?
   A. icon class
   B. button class
   C. graphics class
   D. screen class

4. A BlackBerry smartphone user can expect a standard screen layout, a default menu, and behavior from screen components when they press the Escape key or click the trackwheel or trackball. True or false?

5. Why is good Graphical User Interface design important?

6. Which APIs can you use to create UIs?

7. _______________ provide a standard screen layout, a default menu, and standard behavior when the BlackBerry smartphone user presses the Escape key or clicks the trackwheel or trackball.

8. _______________ provide standard UI elements for date selection, options, check boxes, lists, text fields and labels, and progress bar controls.
9. _______________ provide an application with the ability to arrange components on a BlackBerry smartphone screen in standard ways, such as horizontally, vertically, or in a left-to-right flow.

10. What type of input can an application capture from the user?

11. In what order do you use the following classes when creating a UI?
   A. Field
   B. Manager
   C. Screen

12. What is the minimum number of screens that are required to display information?

13. How many screen types are there?

14. Managers can contain other managers. This is called _____________.

15. How many parts can you divide your screen into?
Answers

1. B and C

2. One of the most commonly used screen classes for a BlackBerry application is MainScreen, which provides functionality to easily add a title and menu functions.

3. C

4. True

5. An application that is easy to use lets a user perform tasks in an efficient manner, whereas a poor design can easily frustrate that person and prevent them from using the application.

6. You can use standard MIDP APIs and BlackBerry UI APIs to create BlackBerry Java Application UIs.

7. Screen components

8. Field components

9. Layout managers

10. Trackwheel movement and clicks, trackball movement and clicks, touchpad movement and clicks, screen touches and clicks, typing on the keyboard, and shaking and moving the device (for BlackBerry smartphones with an accelerometer).

11. C, B, A

12. 1

13. 5

14. Nesting

15. You can divide the screen in as many parts as you want in your application, or not, by using vertical and horizontal managers.
Creating custom managers

You can extend the Manager class to allow custom behavior. The Manager class provides fundamental functionality for all field manager and manager objects used to contain and layout fields. Manager subclasses handle specific layout types. You can implement your own manager by extending the Manager class to implement a sublayout that is more controlled.

You can use custom managers to position fields in a unique manner or add a graphical effect to the screens in your application. The code sample in Appendix B is an example of a custom manager that works with a custom field. You can use the custom field to set its $x$ and $y$ coordinates used to position the field in the manager.
1. Which of the following managers do you use to layout a field in a horizontal then vertical flow?
   A. HorizontalFieldManager
   B. FlowFieldManager
   C. DialogFieldManager
   D. VerticalFieldManager

2. You can use no more than two managers to divide the screen into no more than three areas. True or false?

3. When you create managers to define screen zones (such as a vertical or horizontal screen manager), you can add the managers in any order, regardless of the order in which you want the zones to appear. True or false?

4. The Manager class provides fundamental functionality for all field manager and manager objects used to contain and layout fields. True or false?
Answers

1. B

2. False. You can nest managers within managers, and you can divide a screen into as many areas as you want.

3. False. You must add managers in the order in which you want the zones to appear.

4. True
Custom fields

You can extend the Field class to allow custom behavior by using screen placement, visual changes, or other forms of custom functionality. The code sample in Appendix C is a sample class that extends the Field class. This code sample works with the sample CustomManager class, shown in the sample code in Appendix B.

Creating a custom field

You can only add custom context menu items and custom layouts to a custom field. The following procedure illustrates how to create a custom field:

1. Import the following classes:
   - net.rim.device.api.ui.Field
   - java.lang.String
   - net.rim.device.api.ui.Font
   - java.lang.Math
   - net.rim.device.api.ui.Graphics

2. Import the net.rim.device.api.ui.DrawStyle interface.

3. Extend the Field class, or one of its subclasses, implementing the DrawStyle interface to specify the characteristics of the custom field, and turn on drawing styles.

   public class CustomButtonField extends Field implements DrawStyle {

   public static final int RECTANGLE = 1;
   public static final int TRIANGLE = 2;
   public static final int OCTAGON = 3;
   private String _label;
   private int _shape;
   private Font _font;
   private int _labelHeight;
   private int _labelWidth;

   }

4. Implement constructors to define a label, shape, and style of the custom button.

   public CustomButtonField(String label) {
     this(label, RECTANGLE, 0);
   }

}
public CustomButtonField(String label, int shape) {
    this(label, shape, 0);
}

public CustomButtonField(String label, long style) {
    this(label, RECTANGLE, style);
}

public CustomButtonField(String label, int shape, long style) {
    super(style);
    _label = label;
    _shape = shape;
    _font = getFont();
    _labelHeight = _font.getHeight();
    _labelWidth = _font.getAdvance(_label);
}

5. Implement layout() to specify the arrangement of field data.

6. Perform the most complex calculations in layout() instead of in paint(). The manager of the field invokes layout() to determine how the field arranges its contents in the available space. Invoke Math.min() to return the smaller of the specified width and height, and the preferred width and height of the field. Invoke Field.setExtent(int, int) to set the required dimensions for the field.

protected void layout(int width, int height) {
    _font = getFont();
    _labelHeight = _font.getHeight();
    _labelWidth = _font.getAdvance(_label);
    width = Math.min( width, getPreferredWidth() );
    height = Math.min( height, getPreferredHeight() );
    setExtent( width, height );
}
7. Implement `getPreferredSize()` using the relative dimensions of the field label to ensure that the label does not exceed the dimensions of the component.

8. Use a switch block to determine the preferred width based on the shape of the custom field. For each type of shape, use an if statement to compare dimensions and determine the preferred width for the custom field.

```java
public int getPreferredSize() {
    switch (_shape) {
        case TRIANGLE:
            if (_labelWidth < _labelHeight) {
                return _labelHeight << 2;
            } else {
                return _labelWidth << 1;
            }
        case OCTAGON:
            if (_labelWidth < _labelHeight) {
                return _labelHeight + 4;
            } else {
                return _labelWidth + 8;
            }
        case RECTANGLE: default:
            return _labelWidth + 8;
    }
}
```

9. Implement `getPreferredHeight()` using the relative dimensions of the field label to determine the preferred height.

10. Use a switch block to determine the preferred height based on the shape of the custom field. For each type of shape, use an if statement to compare dimensions and determine the preferred height for the custom field.

```java
public int getPreferredHeight() {
    switch (_shape) {
        case TRIANGLE:
            if (_labelWidth < _labelHeight) {
                return _labelHeight << 2;
            } else {
                return _labelWidth << 1;
            }
        case OCTAGON:
            if (_labelWidth < _labelHeight) {
                return _labelHeight + 4;
            } else {
                return _labelWidth + 8;
            }
        case RECTANGLE: default:
            return _labelWidth + 8;
    }
}
```
switch(_shape) {
    case TRIANGLE:
        if (_labelWidth < _labelHeight) {
            return _labelHeight << 1;
        } else {
            return _labelWidth;
        }
    case RECTANGLE:
        return _labelHeight + 4;
    case OCTAGON:
        return getPreferredWidth();
}
return 0;
}

11. Implement `paint()`.

12. The manager of a field invokes `paint()` to redraw the field when an area of the field is marked as invalid.

13. Use a switch block to repaint a custom field based on the shape of the custom field. For a field that has a triangle or octagon shape, use the width of the field to calculate the horizontal and vertical position of the start point and end point of a line.

14. Invoke `graphics.drawLine()`, and then use the start and end points to draw the lines that define the custom field. For a field that has a rectangular shape, invoke `graphics.drawRect()` and use the width and height of the field to draw the custom field.

15. Invoke `graphics.drawText()`, and then use the width of the field to draw a string of text to an area of the field.

protected void paint(Graphics graphics) {
    int textX, textY, maxWidth;
    int w = getWidth();
    switch(_shape) {
case TRIANGLE:
int h = (w>>1);
int m = (w>>1)-1;
graphics.drawLine(0, h-1, m, 0);
graphics.drawLine(m, 0, w-1, h-1);
graphics.drawLine(0, h-1, w-1, h-1);
textWidth = Math.min(_labelWidth,h);
textX = (w - textWidth) >> 1;
textY = h >> 1;
break;

case OCTAGON:
int x = 5*w/17;
int x2 = w-x-1;
int x3 = w-1;
graphics.drawLine(0, x, 0, x2);
graphics.drawLine(x3, x, x3, x2);
graphics.drawLine(x, 0, x2, 0);
graphics.drawLine(x, x3, x2, x3);
graphics.drawLine(0, x, x, 0);
graphics.drawLine(0, x2, x, x3);
graphics.drawLine(x2, 0, x3, x);
graphics.drawLine(x2, x3, x3, x2);
textWidth = Math.min(_labelWidth, w - 6);
textX = (w-textWidth) >> 1;
textY = (w-_labelHeight) >> 1;
break;
case RECTANGLE: default:
  graphics.drawRect(0, 0, w, getHeight());
  textX = 4;
  textY = 2;
  textWidth = w - 6;
  break;
}
  graphics.drawText(_label, textX, textY, (int)(getStyle() & DrawStyle.ELLIPSIS | DrawStyle.HALIGN_MASK ), textWidth );
}

16. Implement the Field set() and get() methods.

17. Implement the Field.getLabel(), Field.getShape(), Field.setLabel(String label), and Field.setShape(int shape) methods to return the instance variables of the custom field.

public String getLabel() {
  return _label;
}

public int getShape() {
  return _shape;
}

public void setLabel(String label) {
  _label = label;
  _labelWidth = _font.getAdvance(_label);
  updateLayout();
}

public void setShape(int shape) {
  _shape = shape;
}
1. You must create a custom field if you want to add custom context menus to your application. True or false?

2. Field components provide which of the following standard UI elements?
   A. Options
   B. Labels
   C. Check boxes
   D. Dialog boxes

3. Name two field types that support text formatting options (such as bold, italic, and underline).

4. Extending the Field class allows you to create custom behavior by using __________.
Answers

1. True

2. A, B, and C

3. RichTextField and ActiveRichTextField

4. screen placement, visual changes, or other forms of custom functionality
Drawing directly on the screen

There are two main areas where you can typically use the Graphics class: at the field level and the screen level. You can draw at the field level to create a unique field such as a custom looking button or a graph. You can draw at the screen level to have full control of the screen and what is presented to the BlackBerry smartphone user. Typically you draw at the screen level to make use of animation or multimedia content in an application, such as games and graphic demos.

Graphics class

Whether you create a custom screen or a custom field, you perform the majority of the work of the Graphics class in the `paint` method of the Field class (remember that the Screen class extends the Field class). A BlackBerry smartphone invokes the `paint` method when a field, or part of the field, must be redrawn.

You can override the `paint` method to create a custom field, or to modify an existing field. One of the simplest things you can do by overriding the `paint` method is to change the color used. For example you can change the text color of a `RichTextField` to green by doing the following:

```java
//Green - The format for color is 0x00RRGGBB.
Long myColor = 0x00008800;

RichTextField colorChange =
    new RichTextField("The quick brown fox jumps over the lazy dog.")
{
    public void paint(Graphics graphics)
    {
        //Change the color of the text in
        //the RichTextField to green.
        graphics.setColor(myColor);
        //call super.paint() to carry out the
        //default painting of the field
        super.paint(graphics);
    }
}
```
You can also change values used within the `paint` method between calls to `paint` to create animation. Continuing from the preceding code sample, you can do the following:

```java
//Change the color to red.
myColor = 0x00FF0000;
colorChange.invalidate();
```

You can force a field, or part of a field, to be redrawn by invoking one of its `invalidate` methods. If you call the `invalidate()` method, you mark the entire field as invalid by instructing the entire field to be redrawn. If you call `invalidate(int x, int y, int width, int height)`, you mark a certain region of a field as invalid. By marking a certain region of a field as invalid, you can increase the speed of the painting because only a certain area of a field has changed. Only the region specified is redrawn, which can provide a substantial performance increase for large fields, or fields that are redrawn in rapid succession such as when performing animation.

**Drawing graphics within graphics**

You can also create graphics within graphics. Vary the approach you take to create your graphics based on their usage pattern. You can successfully override the `paint` method for a field or screen that changes often, such as during an animation, but this approach is less desirable when you are creating an image that rarely changes but is reused throughout an application. The following code sample is an example of how you can create a dynamic bitmap you can use to populate a `BitmapField`. The following code sample shows a bitmap that contains a large purple square and a green circle:

```java
//Instantiate a bitmap 100x100 pixels in size.
Bitmap myBitmap = new Bitmap(100, 100);

Graphics myGraphics = new Graphics(myBitmap);

//Change the colour to purple.
myGraphics.setColor(0x00550055);

//Draw a filled rectangle starting at 0,0
//that is 100 pixels square.
```
myGgraphics.fillRect(0, 0, 100, 100);

//Change the colour to green.
myGraphics.setColor(0x00004400);

//Draw a filled circle that has a radius of 25.
myGraphics.fillArc(50, 50, 25, 25, 0, 360);

//Instantiate a BitmapField with the
//created bitmap.
BitmapField myBitmapField = new BitmapField(myBitmap);

You can use this technique to store the bitmap and display it again in other areas at a later time. If the image is fairly detailed and requires multiple calls to several of the graphics methods, you can save some processing time by creating the image once and reusing it. It is faster to reuse an image than to recreate an image every time because you use graphics methods from within the paint method and reuse the existing bitmap image.

Although ideal for images that remain fairly constant, this approach is not optimal for highly dynamic content.

**Animation**

Animation used by a game or multimedia application can push any device to its limits. Whether you are using a desktop PC, laptop or a mobile device such as a BlackBerry smartphone you want to ensure that you use the platform effectively. This section describes using the Graphics class and other BlackBerry APIs to create and optimize an animated sequence.

The following graphics illustrates a complete animation that uses the Graphics class and other BlackBerry APIs.

The following graphics shows the basis of the animation.
The calculations and drawing of the animation take place in the AnimationScreen class. AnimationScreen extends FullScreen and contains a thread called AnimationThread. AnimationThread controls what is drawn in each frame, performs all of the manipulation calculations and calls invalidate to repaint the screen.

There are three images included in the application. One image contains a single brick used to build the house.

Another holds the roof.

A third contains the sun, cloud, and each frame of the runner (there are six runner images in total). The sun, cloud, and runner are all of the same height, which allows you to easily place them side by side in a rectangular image.

Including these elements in one file allows for a size reduction and reduces the overhead (image headers, meta-data, for example.) that exist if all eight elements are in their own image and helps reduce the application size. When everything is complete the total size of the application including images is only 17.6 kb. The images are defined as members of the AnimationScreen class.
private final static Bitmap _imageItems = Bitmap.getBitmapResource("imageItems.png");
private final static Bitmap _brick = Bitmap.getBitmapResource("brick.png");
private final static Bitmap _roof = Bitmap.getBitmapResource("roof.png");

The paint method of FullScreen — which is inherited from the Screen, then Field class — is overridden to allow you to draw directly to the screen. The following sections look at some of the Graphics methods that are used to draw the scene.

First, in the paint method, obtain the screen dimensions. You can use the dimensions to scale aspects of the animation and make use of all available screen real estate. The animation is designed to not leave any blank areas along the top, bottom, or sides of the screen on high resolution BlackBerry smartphones, but to still fit on the screen on BlackBerry devices that have a smaller screen resolution.

int width = Graphics.getScreenWidth();
int height = Graphics.getScreenHeight();

Four integers are defined to be used in counters, for loops and as holders for calculations for x and y values used in the application.

int count, count2;
int tempX, tempY;

Storing these integers in a defined variable can be more efficient than calculating them inline, as the inline calculation creates a temporary int variable for each calculation. This can result in the creation of a lot of garbage that can add up quickly in a loop, and even faster in a loop inside the paint method that is called repeatedly during the animation. In the following sample code, you can also save two arithmetic operations by storing the result in a variable. The following code samples show both the recommended and improper methods.

Note:
The following code sample is not part of this application
//Inline calculations (BAD!):
graphics.fillRect(0, width - 10, width, 10);
graphics.fillRect(0, width - 10, width, 20);
graphics.fillRect(0, width - 10, width, 30);
...

//Use defined variables (GOOD!):
tempY = width - 10;
graphics.fillRect(0, tempY, width, 10);
graphics.fillRect(0, tempY, width, 20);
graphics.fillRect(0, tempY, width, 30);
...

Returning to the animation, the first element you draw is the grass. Set the drawing colour to green and use the fillrect method to draw a painted rectangle at the bottom of the screen. The rectangle is 10 pixels high and the width of the screen, and is placed at the bottom of the screen.

//Set the colour to green
graphics.setColor(0x00008800);

//Draw the grass.
tempY = height - 10;
graphics.fillRect(0, tempY, width, 10);

The next element to draw is the sun. The sun is placed at the same coordinates regardless of the screen resolution. The sun is part of the imageItems bitmap, which also contains the cloud and all six runner frames. The sun is 43 pixels wide, 40 pixels tall, and starts 170 pixels from the left of the image. Use the drawBitmap method and only draw the portion of the imageItems bitmap that contains the sun.

graphics.drawBitmap(20, 10, 37, 40, _imageItems, 170, 0);
Next you draw the house. The first item to draw is the house wall. Because you are using an image that contains a single brick, you must draw it multiple times to make up the entire house wall. You can do this using two loops that repeatedly call the `drawBitmap` method. Prepopulate the count variable and decrement in each loop. Comparing to 0 is faster than comparing to another value, and predecrementing count (``count--``) is also faster than postdecrementing (``count--``).

```java
for(count = 8; count > 0; --count)
{
    for (count2 = 5; count2 > 0; --count2)
    {
        //The x coordinate.
        tempX = 220 - count2 * 20;

        //The y coordinate.
        tempY = height -10 - count * 10;

        //Draw a brick.
        graphics.drawBitmap(tempX, tempY, 20, 10, _brick, 0, 0);
    }
}
```

Draw the remainder of the house (roof, window and door) next. Draw the roof using `drawBitmap`, the door using `fillRect` and `fillArc`, and the window using `fillRect`. The clouds and runner are also drawn using `drawBitmap`.

The lightning is not shown on every frame. Before starting to draw the lightning, check to ensure the `drawLightning` variable is true (this value is changed in the `AnimationThread` that controls the animation sequence). There are multiple calls used to make up the bolt of lightning, but the only new one is the `drawLine` method.

```java
graphics.drawLine(100, 30, 110, 45);
```

The `drawLine` method accepts four ints. The first two are the x and y coordinates for the start point of the line, and the second two are the x and y coordinates for the end point of the line.
The paint method is now drawing what is required and the AnimationThread is set up to control the sequence.
1. Which method is most commonly used to create graphics in the Graphics class?

2. Which method do you use when something must be redrawn?

3. How do you force a field to redraw?

4. How do you draw over a bitmap without changing the content of the bitmap?

5. You can override the paint method for a field or screen that changes, such as during an animation. When is using this approach ideal?

6. How does combining multiple images into one file help reduce file size?
Answers

1. Paint method

2. Invalidate method

3. By invoking one of its invalidate methods.

4. Create a BitmapField and pass the Bitmap object into the BitmapField constructor

5. This approach is ideal for images that remain fairly constant.

6. Including multiple elements in one file, when possible, allows for a size reduction of the file and reduces the overhead (image headers, meta-data, for example.) that exist if all the separate elements are in their own image. This helps reduce the application size.
SVG

SVG is a text-based XML language that is useful for describing, two-dimensional graphical content. SVG was developed by the W3C®, the nonprofit, open-standards consortium that created HTML and XML.

SVG are used to develop the following types of content for mobile smartphones:
- interactive themes
- mobile web sites
- splash screens
- graphics
- animations

Bitmap web graphics, such as JPG and GIF images, maintain a specified size despite the display area. In contrast, you can scale SVG to fit any size screen. This feature makes SVG suitable for deployment on wireless handhelds and ideal for content deployment on BlackBerry smartphones. Different BlackBerry smartphones have different screen resolutions; using SVG, you can create content that is automatically scaled based on the display size, with no loss in image quality or in the legibility of text.

The PME file format is a binary representation of your SVG content that plays on BlackBerry smartphones. The PMB file format combines images, audio, and PME files into a single file to minimize the number of connections made over the wireless network.

These file formats enable you to do the following:
- animate primitives (such as rectangles and polygons), color, and images (PNG, GIF)
- create hotspots (similar to links in HTML, but more flexible)
- trigger events (for example, clicking a hotspot to change a color or play an animation)

You can create flash content, and then use the Plazmic Composer to convert the Adobe® Flash® SWF file into supported SVG. You can then compile the SVG into a PME file.

You can create SVG content by writing code manually in a text editor or designing content with a graphics editing tool, such as Plazmic Composer, Adobe® Illustrator, or Microsoft® Visio. Use these tools to export your content as SVG.
Chapter 4

SVG creation with text editors

SVG is a text-based XML language, so you can write code in a text editor to create SVG.

Components of a basic SVG document

The following annotated sample describes a simple SVG document.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;?xml version=&quot;1.1&quot;?&gt;</td>
<td>This line specifies the XML declaration.</td>
</tr>
<tr>
<td>&lt;ellipse cx=&quot;100&quot; cy=&quot;80&quot; rx=&quot;80&quot; ry=&quot;60&quot; fill=&quot;white&quot;/&gt;</td>
<td>This line shows SVG content that defines an ellipse.</td>
</tr>
<tr>
<td>&lt;/svg&gt;</td>
<td>You require the &lt;/svg&gt; tag to close the SVG document fragment.</td>
</tr>
</tbody>
</table>

SVG root element

You define an SVG document using the <svg> element. An SVG document consists of elements contained within the opening <svg> tag and closing </svg> tag.

Namespaces

The SVG Transcoder uses sets from several different XML languages (for example, SVG and XLink). Each xmlns attribute in the <svg> element specifies the namespace for one of those XML languages. The SVG Transcoding Utility requires these namespaces to validate the tags in your SVG content before transcoding. You must properly identify each of the namespaces you require for your SVG document for the SVG Transcoder to convert the file into PME.

The SVG Transcoder identifies the namespaces that you reference in your SVG document by using the xmlns[:prefix] attribute of the outermost <svg> element. For more information about using namespaces, visit www.w3.org/tr/rec-xml-names.
At minimum, your SVG document must include the following namespaces:

```
xmlns="http://www.w3.org/2000/svg"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

You can use additional namespace references if your document requires more functionality.

<table>
<thead>
<tr>
<th>Feature set</th>
<th>Namespace reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyperlinks</td>
<td>xmlns:xlink=&quot;<a href="http://www.w3.org/1999/xlink">http://www.w3.org/1999/xlink</a>&quot;</td>
</tr>
<tr>
<td>Plazmic extensions</td>
<td>xmlns:pz=&quot;<a href="http://www.plazmic.com/plazmic_extents">http://www.plazmic.com/plazmic_extents</a>&quot;</td>
</tr>
<tr>
<td>XForms</td>
<td>xmlns:xforms=&quot;<a href="http://www.w3.org/2002/xforms">http://www.w3.org/2002/xforms</a>&quot;</td>
</tr>
</tbody>
</table>

**Elements**

An element includes the start tag («circle»), the end tag («/circle»), and everything in between.

The SVG language relates elements as parents and children. You can categorize SVG elements into the following basic types:

- **Container elements**, which hold graphic elements and other container elements such as child elements. BlackBerry smartphones support the following container elements: «svg» , «g», «defs», and «a».
- **Graphic elements**, which you use to draw graphics onto the target canvas. SVG supports three types of graphic objects: vector shapes, images, and text. BlackBerry smartphones support the following graphics elements:
  - **basic shapes** («circle», «ellipse», «line», «polygon», «polyline», and «rect»)
  - **text** («text», «textarea», and «tspan»)
  - **paths** («paths»)
  - **images** («image»)
- **Animation elements**, which you use to perform affine animations on graphic elements.
- **XForms elements**, which you use to add form fields to your SVG content to collect data from users.

Mobile devices render elements in the order in which they appear in an SVG file. When layering objects, place background objects at the top of the file.
Syntax

As in XML, you can validate SVG against a schema. SVG with correct syntax is well-formed. This document describes the syntax for valid and well-formed SVG.

In well-formed SVG, consider the following guidelines:

- Close your elements or express them as empty. Every opening tag (for example, `<text>`) requires a closing tag (for example, `</text>`). Express empty elements as `<text></text>` or `<text/>`.
- Nest your elements properly. Because SVG is a hierarchical language, you cannot overlap elements. To nest elements properly, you must close child elements before closing parent elements.
- Use consistent case when you name elements; elements are case sensitive. XML-aware applications (such as XML parsers that convert SVG code into SVG objects) cannot match mixed cases for starting and ending element tags. For example, the `<Name>` tag is different from the `</name>` tag.

For more information about coding SVG manually, see the SVG for BlackBerry Smartphones Reference Guide.

SVG APIs

SVG supports a number of APIs:

- SVGAnimationElement
- SVGElement
- SVGLocatableElement
- SVGMatrix
- SVGPath
- SVGPoint
- SVGRect
- SVGRGBColor
- SVGSVGEleme

SVGAnimationElement

This interface represents an Animation element, which contains methods to control the timing of animations.
SVGElemnt

This interface represents an SVG element in the document tree.

You can use this interface to read and manipulate the value of traits associated with the SVGElement API. Each trait corresponds to an attribute or property, which is parsed and understood by the element and in most cases animatable. Unlike attributes, each element has a well-defined set of traits and attempting to access undefined trait is an error. Also unlike attributes, traits are typed and their values are normalized; for instance SVG path specification is parsed and all path commands are converted to their absolute variants. It is not possible to say through the value of the trait if a path command was absolute or relative. When getting and setting trait values, you must use an accessor of the correct type or an exception is thrown.

Initial trait values come from parsing corresponding attributes. If a value is not specified, but a corresponding attribute (or property for environments where styling is supported) is inherited, the inherited value is returned as a result of the trait query method. If the value is not inherited, the default value is returned. Default values are also returned when there is no parent to inherit from. For example, when you create a new element, set a trait value to inherit, but there is no parent for inheritance. The value that is returned is always a base value (that is, no animation has been applied). This is true for both static and animated content.

Setting a trait value has the same effect as changing a corresponding attribute, but trait setters can operate on typed values. The value which is modified is always a base value. For inheritable traits, the trait value can always be set to inherit (but querying the value always returns the actual inherited value as explained previously).

There are two situations where the various trait setter methods (such as setTrait, setFloatTrait or setPathTrait methods) consider a value invalid and throw a DOMException with the INVALID_ACCESS_ERR code. The first situation is when the trait value is invalid with regards to its definition (for example, trying to set the stroke-linejoin trait to foo causes this exception). The second situation is when the trait value is invalid with regards to animations currently applied to the trait. The value is considered invalid because it puts the animation, and therefore the document, in an error state. For example, if an element has animations on its d attribute, trying to change the d attribute to a value incompatible with the animations causes the exception to happen.

SVGLocatableElement

This interface represents an SVGLocatableElement. This API is implemented by all drawable SVG elements in the document tree. Drawable elements are:

- <rect>
- <circle>
- <ellipse>
- <line>
- <path>
Chapter 4

- <use>
- <image>
- <text>
- <svg>
- <a>
- <g>

The code samples shown in Appendix D at the end of this chapter clarify the behavior of the `getBBox()` method.

**SVGMatrix**

This interface represents an SVGMatrix datatype, identified by an affine transform.

You can use this API to read and modify the values of a transform attribute according to SVG specifications. The `mTranslate`, `mMultiply`, `mScale` and `mRotate` methods in this interface mutate the `SVGMatrix` object and return a reference to the `SVGMatrix` instance itself, after performing the necessary matrix operation.

This matrix transforms source coordinates \((x, y)\) into destination coordinates \((x', y')\) by considering them to be a column vector and multiplying the coordinate vector by the matrix according to the following process:

\[
\begin{bmatrix}
  x' \\
  y'
\end{bmatrix} =
\begin{bmatrix}
  a & c & e \\
  b & d & f \\
  0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  1
\end{bmatrix} =
\begin{bmatrix}
  a.x + c.y + e \\
  b.x + d.y + f \\
  1
\end{bmatrix}
\]

**SVGPPath**

This interface represents an SVGPPath datatype used to define the path geometry. This interface corresponds with the SVG path specification or the "d" attribute.

Native implementations must support the following simplifications or canonicalization of path segments. Any simplifications must have no loss.

- Convert relative commands \((c, h, l, m, q, s, t, \text{ and } v)\) to their absolute counterparts.
- Convert horizontal and vertical lines \((H, h, V, \text{ and } v)\) to general lines \((L \text{ and } l)\).
Translate command $S$ to command $C$.

Translate command $T$ to command $Q$.

**SVGPoint**

This interface represents an SVGPoint datatype, identified by its x and y components.

**SVGRect**

This interface represents an SVGRect datatype, consisting of a minimum X, minimum Y, width and height values.

**SVGRGBColor**

This interface represents an SVGRGBColor datatype made up of red, green, and blue components. You can use this interface to read properties that store color values (getRGBColorTrait) such as fill, stroke, and color.

**SVGSVGElement**

This interface represents the `<svg>` element in an (SVG) document tree.

**User Agent Transforms**

The DOM attributes currentScale, currentRotate, and currentTranslate are combined to form user agent transformation. This is applied at the outermost level on the SVG document (that is, outside the outermost svg element) if magnification is enabled (that is, the zoomAndPan attribute is set to magnify). The values of the attributes can be modified through a user-agent specific UI. User agent transformation can be obtained by multiplying matrix.

\[
\begin{bmatrix}
\text{currentScale} & 0 & \text{currentTranslate}.x \\
0 & \text{currentScale} & \text{currentTranslate}.y \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\cos(\text{currentRotate}) & -\sin(\text{currentRotate}) & 0 \\
\sin(\text{currentRotate}) & \cos(\text{currentRotate}) & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

(That is, translate, then scale, then rotate the coordinate system.) The reference point for scale and rotate operations is the origin (0, 0).

If the application does not have the necessary privilege rights to access this (SVG) content, a SecurityException can be thrown by the underlying implementation. This is applicable to all the Tree navigation and Trait accessor methods. Features such as zooming, panning, and playing animations are not affected.
Creating 2-D and 3-D graphics by using JSR-239

You can create 2-D and 3-D graphics for a BlackBerry smartphone by using JSR-239 and BlackBerry APIs. JSR-239 contains the Java binding for OpenGL® ES. OpenGL ES is based on the desktop version of OpenGL, but has been designed for use on wireless devices.

OpenGL ES is a low-level, lightweight API for advanced embedded graphics using well-defined subset profiles of OpenGL. It provides a low-level applications programming interface (API) between software applications and hardware or software graphics engines.

Refer to Appendix E for a code sample that demonstrates how to use JSR-239 to render a multicolor 2D triangle.

Packages for JSR-239 support

The following packages support JSR-239.

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.microedition.khronos.opengles</td>
<td>This package provides the bindings for OpenGL® ES 1.0, 1.1 and extensions. It includes floating and fixed-point manipulation of values, and provides a subset of OpenGL APIs.</td>
</tr>
<tr>
<td>javax.microedition.khronos.egl</td>
<td>This package provides the bindings for EGL™ 1.0 and 1.1 for the OpenGL ES API. The rendering contexts that are supported can provide a container for the OpenGL ES rendering state. The package is designed to support Window, Pbuffer, and Pixmap rendering surfaces.</td>
</tr>
</tbody>
</table>
| net.rim.device.api.opengles | The GLUtils class in this package provides the following support:  
  - determine if the BlackBerry device supports OpenGL ES  
  - load a 3-D perspective projection matrix  
  - load a view transformation matrix  
  - define a 2-D orthographic projection matrix  
  - load a texture from a Bitmap image  
  - free memory that is currently allocated for a direct buffer |
| java.nio | This package provides the classes that enable your application to read and write high-speed block I/O. |
The SVG rendering model is a basic extension of the standard DOM 2 structure. The Document acts as a scene tree to host a variety of graphical and behavioural elements. The document uses a painters model for rendering, where nodes are rendered in document order, from parent to children through its siblings. SVG 1.1 Tiny (SVGT 1.1) is the document model supported on the BlackBerry smartphone and defined in JSR 226. The style attribute is markedly absent from SVGT 1.1 (that is, `<rect style="fill:rgb(0,0,255);stroke-width:1;" .../>`). JSR 226 gives you full programming access to the rendering model through a DOM+ set of APIs, where you can make changes to the model and see them immediately reflected visually.

The following code sample produces two overlapping images.

<!-- BlackBerry.svg -->

```xml
<?xml version="1.0" encoding="UTF-8"?>

<svg version="1.1" baseProfile="tiny"
 xmlns="http://www.w3.org/2000/svg"
 width="100%" height="100%" viewBox="0 0 320 240">
  <!-- Blue Ellipse -->
  <ellipse id="e1" cx="50" cy="50" rx="100" ry="30" fill="blue"/>

  <!-- Red Recangle with rounded corners and transparency -->
  <rect id="r1"
    x="30" y="50" width="180" height="80"
```

This package provides the classes that are designed to help you enhance the arithmetic versatility and processing speed of the BlackBerry device applications that you create.
The preceding code produces the following image.

![Image of JSR 226 on BlackBerry graphic rendering]

*Figure 5.7 Overlapping graphic rendering*
Loading an SVGImage (rendering model)

The following code sample illustrates how to load an SVGImage.

```java
public void loadModel(String path)
{
    // Load the input stream.
    InputStream is = javax.microedition.io.Connector.openInputStream(path);

    // Load the render model image. Parameter value 'null' means no external resource handler.
    SVGImage image = (SVGImage)SVGImage.createImage(is, null);

    // Get the document from the image.
    Document document = image.getDocument();

    // Get the root svg element. <svg/>
    SVGSVGElem svgElement = (SVGSVGElem)document.getDocumentElement();

    ...
}
```

Creating an SVGImage (rendering model)

To make a full SVG Rendering Model from scratch, you must first create an empty model. The following sample text illustrates an empty model for this purpose.

```java
public void createModel()
{
    // Create an empty model.
```
Chapter 4

```
SVGImage image = SVGImage.createEmptyImage(null);

// Get the document from the image.
Document document = image.getDocument();

// Get the root svg element. <svg/>
SVGSVGElement svgElement = (SVGSVGElement)document.getDocumentElement();

...
```

**Setting and getting traits**

With an `<svg/>` element to work with, you can easily get and set traits on it using `getTrait/setTrait` and `getTraitNS/setTraitNS`. Additionally, there is support for scalar traits using the `getFloat_trait` or `setFloatTrait(String, float)` and arrays with `getFloatListTrait/setFloatListTrait(String, float[])`. In SVG Tiny 1.x the `<svg/>` element is the only element that supports percentage values on only the width and height (for example, `width="100%"`). When not set, the default width and height are in pixels, or 100x100. The following code sample illustrates the use of these scalar traits to set the width and height of the svg elemen.

```
public void updateElement()
{
    SVGSVGElement svgElement = (SVGSVGElement)document.getDocumentElement();

    svgElement.setFloatTrait("width", 200);
    svgElement.setFloatTrait("height", 200);

    ...
}
```
Setting and getting complex traits

For ease of use, the specification has defined a set of set/get traits methods for dealing with scalar int, floats, and complex types. There are several methods for creating, setting, and getting complex traits. Some methods include SVGRect, SVGPath, and SVGRGBColor.

The following code sample illustrates how to apply a view box transformation to the svg element:

```java
public void updateComplexTrait()
{
    // Define the <svg viewBox="10 10 200 200" for the svg node.
    SVGRect viewBox = svgElement.createSVGRect();
    viewBox.setX(10);    
    viewBox.setY(10);    
    viewBox.setWidth(200);  
    viewBox.setHeight(200);  
    svgElement.setRectTrait("viewBox", viewBox);  
    ...
}
```

Creating a node (SVGElemet)

You can create your own scalable content. There are a variety of node graphical types.

The following code sample illustrates how to create an ellipsis: using node graphical types.

```java
public void createElement()
{
    // Create an ellipse, position it and make its fill blue and set the stroke width to 4
    SVGElemet ellipse1 = (SVGElemet)document.createElementNS(SVG_NAMESPACE_URI, "ellipse");

    ellipse1.setId("e1");
```
ellipsel.setFloatTrait("cx", 50);
ellipsel.setFloatTrait("cy", 50);
ellipsel.setFloatTrait("rx", 100);
ellipsel.setFloatTrait("ry", 30);

SVGRGBColor colorBlue = svgElement.createSVGRGBColor(0, 0, 255);
ellipsel.setRGBColorTrait("fill", colorBlue);

ellipsel.setTrait("stroke-width", "4");

...

} // end of the build() method

Inserting, appending, and removing nodes

You can insert, append, and remove nodes to build dynamic content. Be aware of memory usage. Although the scene renders as efficiently as possible, it does nothing for controlling its size. Order matters. The model renders top down to give you known-to-render order. The methods used in the following code sample are the standard DOM APIs: insertBefore, appendChild, and removeChild.

The following code sample illustrates how to add nodes to the svg element created in a preceding code sample.

    public void build()
    {
        svgElement.appendChild(rect1);
        svgElement.appendChild(text1);
        svgElement.insertBefore(ellipse1, rect1);
        ...
    } // end of the build() method
Finding nodes

Every node can be uniquely identified as in DOM through the \texttt{id} attribute or \texttt{traid}. By applying an \texttt{SVGElement} with an \texttt{id} attribute, you can search for the node by its \texttt{id}. Use the standard DOM APIs to find an element by its \texttt{id} and cast it to an \texttt{SVGElement}.

The following code sample illustrates how to find the \texttt{<rect/>} node called \texttt{r1} in the preceding code sample BlackBerry \texttt{svg}:

```java
public void findById()
{
    SVGElement rect1 = (SVGElement)document.getElementById("r1");
    ...
}
```

You can traverse the DOM tree using the standard traversal APIs such as \texttt{getParentNode}, \texttt{getFirstChild}, \texttt{getLastChild}, \texttt{getNextSibling}, and \texttt{getPreviousSibling}.

The following code sample illustrates how to traverse down to the last element in the preceding.

```java
{
    // Move down 1 level and across 2 sibling. Promise me you will never do this ;)
    SVGElement t1 = (SVGElement) svgElement.getFirstChild().getNextSibling().getNextSibling();
    ...
}
```
Locating nodes

You can find elements location. Use the SVGLocatableElement interface to determine the screen bounds of an element. Group node bounds are a sum of all their children.

The following code sample illustrates how to get a node's screen bounding box rectangle:

```java
{
    // Get the screen bounds for a element.
    SVGRect screenBounds = svgElement.getScreenBBox();

    ...
}
```

Handling model events

Register yourself as an EventListener on any element that implements EventTargets. The elements that you can register yourself as an EventListtener on include the following:

- SVGElement
- SVGSVGElement
- SVGAnimationElement
- SVGLocatableElement

EventTarget exposes a typical publisher/subscriber model for receiving standard SVG events. Supported events, along with their corresponding familiar SVG markup name, include the following:

- DOMFocusIn (focusin)
- DOMFocusOut (focusout)
- DOMActivate (activate)
- click (click)

Register yourself as an EventListener on DOM events. Like any other event, add the target to the focus ring.
The following code sample illustrates how to register yourself as a listener to a DOMFocusIn event on a rect node.

    public class MySVGScreen extends MainScreen implements EventListener {
        protected SVGImage _image;

        public MySVGScreen()
        {
            SVGElement element = (SVGElement)_image.getDocument().getElementById("r1");

            // Register this class to receive focusin events
            // and 'false' to bubble the event up to its parents.
            element.addEventListener("DOMFocusIn", this, false);
        }

        void handleEvent(Event evt)
        {
            // Handler your event and determine the target which sent it.
            EventTarget target = evt.getEventTarget();
            String eventType = evt.getEventType();

            if ("DomFocusIn".equals(eventType))
            {
                // Not required but check that it was actually the focusin.
            }
        }
    }
Navigation and focus order

Animations that have begin or end conditions on any of the DOM events add their targets to the focus ring in render order. Similar to animations, registered event listeners on any DOM event do the same. JSR 226 does not support a method for controlling an alternative focus order without having to change the render order. This is supported in JSR 287. The `nav-next` and `nav-previous` attributes on all visual nodes are supported to control alternative focus order.
1. What does the acronym SVG stand for?

2. What is the minimum screen size for SVG files?

3. What file format do you use to combine images, audio, and PME files into to minimize connections?

4. Which of the following types of content for mobile devices can you create using SVG?
   A. Graphics
   B. Splash screens
   C. Animations
   D. Interactive themes

5. In the XML format for SVG content, which of the following elements contains all other elements?
   A. svg
   B. SVG
   C. xml
   D. svgImage

6. Which types of elements can you include in your SVG document?
   A. Container elements
   B. Graphic elements
   C. XForms elements
   D. All of the above

7. Where must you place background objects in your SVG file?
A. At the top of the file
B. At the bottom of the file
C. In the middle of the file

8. How can you ensure that your SVG document is well-formed? Choose three.
   A. Close all elements or express them as empty
   B. Nest elements properly
   C. Match upper and lower case
   D. Use the .xml file extension

9. Which method must you override to create or modify a custom graphic or animation?

10. After you export content to an SVG file, what is the next step?

11. What must you verify in your SVG document for the SVG Transcoding Utility can validate the SVG tags in your content before transcoding?
### Answers

1. Scalar vector graphics

2. There is no minimum screen size

3. PMB

4. A, B, C, and D

5. A

6. D

7. A

8. A, B, and C

9. Paint method

10. When you export the content, you compile the content. When the content is compiled (exported), you can play the content.

11. The SVG Transcoding Utility requires these namespaces to validate the tags in your SVG content before transcoding. You must properly identify each of the namespaces you require for your SVG document for the SVG Transcoder to convert the file into PME.
The graphical UI is the mechanism that connects the user to the applications they want to use. The graphical UI must be user-friendly and familiar, as well as dynamic and esthetically pleasing. By following standard layouts and reusing familiar components, users can easily navigate the UI on their new BlackBerry smartphone. However, BlackBerry smartphones UIs have different restrictions than laptops and computers, namely, their screen size.

When designing a UI or an application for the BlackBerry smartphone, ensure your design can be supported by the area of the BlackBerry smartphone screen. Customize elements of the UI to provide greater navigation with ease. Create and use custom managers and custom fields to allow users to easily scroll and move through content that can be too large for their BlackBerry smartphone screen.

To increase the user experience, work with SVG to create dynamic UI components and animations. You can leverage SVG and the Plazmic Content Developer Kit tools to create the following:

- interactive themes
- mobile web sites
- splash screens
- graphics
- animations

Take advantage of the flexibility of the PME and PMB formats to do the following:

- animate primitives (such as rectangles and polygons), color, and images (PNG, GIF)
- create hotspots (similar to links in HTML, but more flexible)
- trigger events (for example, to change a hotspot color or play an animation)

Use custom managers, custom fields, and SVG, and the Plazmic Content Developer Kit tools to create advanced BlackBerry smartphone UI details to create UIs that the user can navigate easily, quickly, and with the best experience.
Review Questions

1. What does the screen class do?

2. Explain how the scalability of SVGs is important for BlackBerry smartphone UIs.

3. Describe how screens are made visible to the interface user.

4. What does the Manager class do?

5. What does the Field class do?

6. What is SVG?

7. Describe how custom managers can enhance a standard UI layout.

8. The PME file format is a binary representation of your SVG content that plays on BlackBerry smartphones. The PMB format combines images, audio, and PME files into a single file to minimize the number of connections made over the wireless network. These file formats enable you to do which of the following:
   A. Create hotspots
   B. Trigger events
   C. Animate graphics

9. What two methods can you use to result in a PME file?

10. What is the SVG Transcoding Utility?
Appendix A: Sample code of nested managers

The following code sample uses nested managers to divide a BlackBerry smartphone screen into two even columns:

```java
//Create an instance of MainScreen.
MainScreen theScreen = new MainScreen();

//Create a HorizontalFieldManager to hold the two VerticalFieldManagers.
HorizontalFieldManager backGround = new HorizontalFieldManager();
VerticalFieldManager leftColumn = new VerticalFieldManager();
VerticalFieldManager rightColumn = new VerticalFieldManager();

//The data to be displayed.
String[] names = {
    "Homer",
    "Marge",
    "Bart",
    "Lisa",
    "Maggie"};
int ages[] = {36, 34, 10, 8, 3};

//Arrays of fields to display the names
//and ages.
LabelField displayNames[] = new LabelField[5];
LabelField displayAges[] = new LabelField[5];
```
int count, width, spaceSize;
StringBuffer sBuffer = new StringBuffer();

//Add the column titles.
LabelField leftTitle = new LabelField("Name");
leftColumn.add(leftTitle);

LabelField rightTitle = new LabelField("Age");
rightColumn.add(rightTitle);

//Determine half of the screen width.
width = Graphics.getScreenWidth() / 2;

//Get the default font.
Font font = Font.getDefault();

//Determine the size of a space in the default font.
spaceSize = font.getAdvance(' ');

//Build up a string that contains enough spaces that it fills half of the screen. This will be used to set the column width.
for(count = 0; count <= width; count += spaceSize)
{
    sBuffer.append(' ');
}
} //Add spacers to move to the next line and set
//the column width.
leftColumn.add(new LabelField(sBuffer.toString()));
rightColumn.add(new LabelField(sBuffer.toString()));

//Add some data to the screen.
//Populate and add the name and age fields.
for (count = 0; count < 5; count++)
{
    displayNames[count] = new LabelField(names[count]);
    displayAges[count] = new LabelField(String.valueOf(ages[count]));

    //Add the name LabelField to the left column.
    leftColumn.add(displayNames[count]);

    //Add the age LabelField to the right column.
    rightColumn.add(displayAges[count]);
}

//Add the two vertical columns to the
//horizontal field manager.
backGround.add(leftColumn);
backGround.add(rightColumn);
//Add the horizontal field manager to
//the screen.

theScreen.add(backGround);

Managers are an extension of the Field class and can contain other managers. This is known as nesting. You can nest managers to create an enhanced layout style, such as a column or table layout.

add(_fieldManagerBottom);

The preceding code sample uses SeparatorField to draw a line between the sections. That is not required, however, it does enhance the look of the screen.

This illustration shows the output of the preceding sample code appearing as two columns of text.

![Figure 5.8 Code result as shown on the screen of a BlackBerry smartphone](image-url)
Appendix B: Sample code for creating a custom manager

The following custom manager code sample makes use of these $x$ and $y$ coordinates to determine where to draw the field:

```java
public class CustomManager extends Manager {
  public CustomManager() {
    //Disable scrolling in this manager.
    super(Manager.NO_HORIZONTAL_SCROLL |
          Manager.NO_VERTICAL_SCROLL);
  }

  //Override sublayout.
  protected void sublayout(int width, int height) {
    CustomField field;

    //Get the total number of fields within
    //this manager.
    //Note that this manager only supports the
    //CustomField described within this chapter.
    int numberOfFields = getFieldCount();

    for (int i = 0; i < numberOfFields; i++) {
      //Get the field.
    }
  }
}
```
field = (CustomField)getField(i);

//Obtain the custom x and y coordinates for
//the field and set the position for
//the field.
setPositionChild(field, field.getXCoord(), field.getYCoord());

//Layout the field.
layoutChild(field, width, height);

} //Set the manager's extent
setExtent(width, height);

}

public int getPreferredWidth()
{
    //This manager is designed to utilize the
    //entire screen width.
    return Graphics.Display.getWidth();
}

public int getPreferredHeight()
{
    //This manager is designed to utilize the
    //entire screen height.
    return Graphics.Display.getHeight();
}
}
Appendix C: Creating a custom field sample code

This field creates a nonfocusable circle that you can place anywhere on the screen. This field adds additional methods to get and set the x and y coordinates of where the field is shown in the manager.

```java
public class CustomField extends Field implements DrawStyle {

    private int fieldWidth, fieldHeight, xCoord, yCoord;

    public CustomField(int xVal, int yVal) {
        //Create a non focusable field.
        super(Field.NON_FOCUSABLE);

        //Set the x and y coordinates.
        xCoord = xVal;
        yCoord = yVal;

        //Set the field to be one quarter the screen width and height.
        fieldHeight = Display.getWidth()/4;
        fieldWidth = Display.getWidth()/4;
    }

    //Set the x coordinate of the field.
    public void setXCoord(int xVal) {
        //If the x coordinate is different the the the
//current x coordinate
//set the x coordinate to the new value and
//call the invalidate method of this field.
//Calling invalidate will force the field
//to be repainted.
if (xVal != xCoord)
{
    xCoord = xVal;
    invalidate();
}

//Get the x coordinate of the field.
public int getXCoord()
{
    return xCoord;
}

//Set the y coordinate of the field.
public void setYCoord(int yVal)
{
    //If the y coordinate is different the
    //current y coordinate then set the y
    //coordinate to the new value and call the
    //invalidate method of this field.
    //Calling invalidate will force the field
    //to be repainted.
    if (yVal != yCoord)
    {

yCoord = yVal;
invalidate();
}
return;
}

// Get the y coordinate of the field.
public int getYCoord()
{
    return yCoord;
}

// Retrieves the preferred width of the button.
public int getPreferredWidth()
{
    return fieldWidth;
}

// Retrieves the preferred height of
// the button.
public int getPreferredHeight()
{
    return fieldHeight;
}

// Lays out this field's contents.
// This field's manager invokes this method
// during the layout process to instruct this
// field to arrange its contents, given an
// amount of available space.
public void layout(int width, int height)
{
    setExtent(getPreferredSize(),
              getPreferredHeight());
}

//Repaints the button.
//The field's manager invokes this method
//during the repainting process to instruct
//this field to repaint itself.
public void paint(Graphics graphics)
{
    //Draw a circle the size of the field.
    graphics.fillArc
        (fieldWidth / 2, 0, fieldWidth, fieldHeight, 0, 360);
}
Appendix D: SVGLocatableElement API code samples

The following code samples each show an SVG fragment followed by a set of bounding box values, which have the following format:

[elementId] : {x, y, width, height} | {null}

where x, y, width, and height define the values of the SVGRect object returned from a getBBox call on the element with the specified ID. There are a few cases where the bounding box is null (see example 6).

Example 1: Simple groups and bounds

This following code sample shows the values returned by the getBBox method for various simple basic shapes and groups. In particular, this code sample shows that the transform, on an element does not change the value of its user space bounding box.

```xml
<g id="group1" transform="translate(10, 20)" fill="red">
  <rect id="rect1" transform="scale(2)" x="10" y="10" width="50" height="50"/>
  <rect id="rect2" x="10" y="10" width="100" height="100"/>
  <g id="group2" transform="translate(10, 20)">
    <rect id="rect3" x="0" y="10" width="150" height="50"/>
    <circle id="circle1" cx="20" cy="20" r="100"/>
  </g>
</g>
```

[group1] : {-70.0, -60.0, 230.0, 200.0}
[rect1] : {10.0, 10.0, 50.0, 50.0}
[rect2] : {10.0, 10.0, 100.0, 100.0}
[group2] : {-80.0, -80.0, 230.0, 200.0}
[rect3] : {0.0, 10.0, 150.0, 50.0}
Example 2: Bounding box on zero width or height rectangle

The following code sample illustrates that the bounding box on elements is based on the geometry coordinates of the element. For example, the bounding box on a zero-width rectangle is defined, even though the rectangle is not rendered.

```xml
<g id="group1" transform="translate(10, 20)" fill="red">
  <rect id="rect2" x="10" y="10" width="400" height="0"/>
  <g id="group2" transform="translate(10, 20)" >
    <rect id="rect3" x="0" y="10" width="150" height="50"/>
  </g>
</g>
```

[group1] : {10.0, 10.0, 400.0, 70.0}
[rect2] : {10.0, 10.0, 400.0, 0.0}
[group2] : {0.0, 10.0, 150.0, 50.0}
[rect3] : {0.0, 10.0, 150.0, 50.0}
Example 3: Bounding Box on zero radius ellipses

The following code sample also illustrates how bounding boxes are based on the geometry of the element. Here, the bounding box of an ellipse with a zero x-axis radius is still defined, even though the ellipse is not rendered.

```xml
<svg id="mySVG" width="10" height="20">
    <g id="group1" transform="translate(10, 20)" fill="red">
        <rect id="rect1" x="10" y="10" width="100" height="100"/>
        <ellipse id="ellipse1" cx="20" cy="20" rx="0" ry="70"/>
    </g>
</svg>

[mySVG] : {20.0, -30.0, 100.0, 160.0}
[group1] : {10.0, -50.0, 100.0, 160.0}
[rect1] : {10.0, 10.0, 100.0, 100.0}
[ellipse1] : {20.0, -50.0, 0.0, 140.0}
```

Example 4: Viewports do not clip bounding boxes

The following code sample illustrates that no matter what the viewport is on the root SVG element, the bounding boxes, based on the geometry, are still defined. Even though the root svg has a zero width, the bounding boxes for the root itself and its children is precisely defined.

```xml
<svg id="mySVG" width="0" height="50">
    <g id="group1" transform="translate(10, 20)" fill="red">
        <rect id="rect1" x="10" y="10" width="50" height="50"/>
        <g id="group2" transform="translate(10, 20)">
            <rect id="rect2" x="0" y="10" width="150" height="0"/>
            <circle id="circle1" cx="20" cy="20" r="500"/>
        </g>
    </g>
</svg>
```

```xml
[group1] : {10.0, -50.0, 100.0, 160.0}
```
Example 5: getBBox on <use>

The following code sample illustrates that the bounding box for a <use> element accounts for the x and y attributes defined on the element, just like the x and y attributes impact the bounding box computation on a <rect> or on an <image> element.

```xml
<svg>
  <defs>
    <rect id="myRect" x="0" y="0" width="60" height="40" />
  </defs>
  <use id="myUse" xlink:href="#myRect" x="-30" y="-20" />
</svg>
```

[myRect] : {0.0, 0.0, 60.0, 40.0}

[myUse] : {-30.0, -20.0, 60.0, 40.0}
Example 6: Empty group

The following code sample illustrates that the bounding box for an empty group is null. By the same token, the bounding box of a `<path>` with an empty SVGPath (that is, one with no path commands, which can happen after creating a new `<path>` element with a `Document.createElementNS` call) is also null.

```xml
<g id="emptyG" />
```

```json
[emptyG] : {null}
```

Example 7: Impact of display='none' and visibility='hidden'

The following code sample illustrates how the bounding box of children with display='none' are not accounted for in the computation of the parent bounding box. This reflects the definition of the display property and its impact on rendering and bounding box computation. The sample also shows that elements with a 'hidden' visibility still contribute to the parent bounding box computation.

```xml
<g id="g1">
  <g id="g1.1.display.none" display="none">
    <rect id="rect1" x="10" y="10" width="40" height="40"/>
  </g>
  <g/>
  <rect id="rect2.visibility.hidden" visibility="hidden"
    x="30" y="60" width="10" height="20"/>
</g>
```

```json
[g1] : {30.0, 60.0, 10.0, 20.0}
[g1.1.display.none] : {10.0, 10.0, 40.0, 40.0}
[rect1] : {10.0, 10.0, 40.0, 40.0}
[rec2.visibility.hidden] : {30.0, 60.0, 10.0, 20.0}
```
Example 8: Concatenating bounding boxes in the container user space

The following code sample illustrates how the concatenation and computation of bounding boxes for container elements happens in the container user space.

```xml
<g id="g1">
  <line id="line1" x2="100" y2="100" transform="rotate(-45)"/>
</g>

[g1] : {0.0, 0.0, 141.42136, 0}
[line1] : {0.0, 0.0, 100.0, 100.0}
```

Example 9: No influence of stroke-width

The following code sample illustrates that stroking has no impact on the computation of bounding boxes.

```xml
<g>
  <line id="thickLine" stroke-width="10" x2="100" y2="0"/>
</g>

[thickLine] : {0.0, 0.0, 100.0, 0.0}
```
Example 10: No influence of viewBox

The following code sample illustrates that viewBox has no impact on the computation of bounding boxes.

```
<svg id="rootSvg" width="500" height="300" viewBox="0 0 200 100" >
  <rect x="-100" y="-200" width="500" height="100" />
</svg>
```

[rootSVG] : {-100, -200, 500, 100}
Appendix E: Code sample: Rendering a multicolor 2-D triangle

You can use the following code sample to demonstrate use of JSR-239 by rendering a multi color 2D triangle.

```java
import java.nio.*;
import javax.microedition.khronos.egl.*;

import javax.microedition.khronos.opengles.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.system.*;
import net.rim.device.api.opengles.*;

public final class OpenGLTest extends UiApplication {
    public OpenGLTest() {
        pushScreen(new OpenGLTestScreen());
    }

    public static void main(String[] args) {
        new OpenGLTest().enterEventDispatcher();
    }
}
```
class OpenGLTestScreen extends FullScreen implements Runnable
{
    private EGL11 _egl;
    private EGLDisplay _eglDisplay;
    private EGLConfig _eglConfig;
    private EGLSurface _eglSurface;
    private EGLContext _eglContext;
    private GL10 _gl;

    private Bitmap _offscreenBitmap;
    private Graphics _offscreenGraphics;

    private FloatBuffer _vertexArray;
    private FloatBuffer _colorArray;

    private boolean _running;
    private boolean _paused;

    OpenGLTestScreen()
    {
        super(FullScreen.DEFAULT_MENU | FullScreen.DEFAULT_CLOSE);
    }
}
private void initialize()
{

    // Get EGL interface
    _egl = (EGL11)EGLContext.getEGL();

    // Get the EGL display
    _eglDisplay = _egl.eglGetDisplay(EGL11.EGL_DEFAULT_DISPLAY);

    // Initialize the display for EGL, null since we don't really need the version
    _egl.eglInitialize(_eglDisplay, null);

    // Choose an EGL config
    EGLConfig[] configs = new EGLConfig[1];
    int[] numConfigs = new int[1];
    int[] attrs =
    {
        EGL11.EGL_RED_SIZE, 5,
        EGL11.EGL_GREEN_SIZE, 6,
        EGL11.EGL_BLUE_SIZE, 5,
        EGL11.EGL_NONE
    };
    _egl.eglChooseConfig(_eglDisplay, attrs, configs, 1, numConfigs);
    _eglConfig = configs[0];
// Create an EGL window surface
_eglSurface = _egl.eglCreateWindowSurface
(_eglDisplay, _eglConfig, this, null);

// Create an EGL context
createEGLContext();

// Specify vertices and colors for a triangle
float[] vertices =
{
    -0.5f, -0.5f, 0.0f,
    0.0f, 0.5f, 0.0f,
    0.5f, -0.5f, 0.0f
};
float[] colors =
{
    0.0f, 1.0f, 0.0f, 1.0f,
    1.0f, 0.0f, 0.0f, 1.0f,
    0.0f, 0.0f, 1.0f, 1.0f
};

_vertexArray = ByteBuffer.allocateDirect(3 * 3 * 4).asFloatBuffer();
_vertexArray.put(vertices);
_vertexArray.rewind();

_colorArray = ByteBuffer.allocateDirect(4 * 3 * 4).asFloatBuffer();
_colorArray.put(colors);
_colorArray.rewind();
}

private void createEGLContext()
{

    // Create an EGL context
    _eglContext = _egl.eglCreateContext
        (_eglDisplay, _eglConfig, EGL10.EGL_NO_CONTEXT, null);

    // Get the GL interface for our new context
    _gl = (GL10)_eglContext.getGL();

    // Make our new context current
    _egl.eglMakeCurrent
        (_eglDisplay, _eglSurface, _eglSurface, _eglContext);

}

private void destroyEGL()
{

    _egl.eglMakeCurrent(_eglDisplay, EGL10.EGL_NO_SURFACE,
                        EGL10.EGL_NO_SURFACE, EGL10.EGL_NO_CONTEXT);
    _egl.eglDestroyContext(_eglDisplay, _eglContext);
    _egl.eglDestroySurface(_eglDisplay, _eglSurface);
}
private void handleContextLost()
{

    // Destroy our EGL context
    _egl.eglMakeCurrent(_eglDisplay, EGL10.EGL_NO_SURFACE,
                        EGL10.EGL_NO_SURFACE, EGL10.EGL_NO_CONTEXT);
    _egl.eglDestroyContext(_eglDisplay, _eglContext);
    _eglContext = EGL10.EGL_NO_CONTEXT;

    // Re-create our EGL context
    createEGLContext();
}

/**
 * Main render loop.
 */
public void run()
{
    initialize();

    while (_running)
    {
        // Idle If we are in the background
        if (_paused)
        {
            synchronized (this)
            {
            
```
try {
    wait();
}
catch (InterruptedException x) { }
}

updateBackBuffer();

renderFrame();

// Throttle cpu usage
try {
    Thread.sleep(20);
}
catch (InterruptedException x) { }
}

destroyEGL();
private void renderFrame()
{

    // Make our context and surface current and check for 
    // EGL_CONTEXT_LOST
    if (!_egl.eglMakeCurrent(_eglDisplay, _eglSurface,
                            _eglSurface, _eglContext))
    {
        if (_egl.eglGetError() == EGL11.EGL_CONTEXT_LOST)
            handleContextLost();
    }

    // Signal that we are about to begin OpenGL rendering
    _egl.eglWaitNative(EGL10.EGL_CORE_NATIVE_ENGINE,
                       _offscreenGraphics);
    render(_gl);

    // Signal that OpenGL ES rendering is complete
    _egl.eglWaitGL();

    // Swap the window surface to the display
    _egl.eglSwapBuffers(_eglDisplay, _eglSurface);
}

private void render(GL10 gl)
{
    // Set our GL viewport
gl.glViewport(0, 0, getWidth(), getHeight());

// Clear the surface

gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);

// Set our projection matrix

gl.glMatrixMode(GL10.GL_PROJECTION);

gl.glLoadIdentity();

GLUtils.gluPerspective(gl, 45.0f, (float)getWidth()/(float)getHeight(), 0.15f, 10.0f);

// Draw our triangle

gl.glMatrixMode(GL10.GL_MODELVIEW);

gl.glLoadIdentity();

gl.glTranslatef(0.0f, 0.0f, -3.0f);

gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);

gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

gl.glVertexPointer(3, GL10.GL_FLOAT, 0, _vertexArray);

gl.glColorPointer(4, GL10.GL_FLOAT, 0, _colorArray);

gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);

}/**
 * Called by the UI system to paint the screen.
 */
protected void paint(Graphics g)
{
    if (_offscreenBitmap != null)
    {
        g.drawBitmap(0, 0, _offscreenBitmap.getWidth(),
                     _offscreenBitmap.getHeight(), _offscreenBitmap, 0, 0);
    }
}

/**
 * Called when the visibility of our screen changes.
 *
 * @param visible true if our screen is being made visible,
 * false if it's being hidden
 */
protected void onVisibilityChange(boolean visible)
{
    if (visible)
    {
        resume();
    }
    else
    {
        pause();
    }
}

/**
* Called when the screen is closing.
* /

public void close()
{
    _running = false;
    synchronized (this) { notifyAll(); }

    super.close();
}

/**
 * Keeps the back buffer in sync with the screen size.
 */

private void updateBackBuffer()
{
    if (_offscreenBitmap != null)
    {
        if (_offscreenBitmap.getWidth() == getWidth() &&
            _offscreenBitmap.getHeight() == getHeight())
            return; // no change needed
    }

    _offscreenBitmap = new Bitmap(getWidth(), getHeight());
    _offscreenGraphics = Graphics.create(_offscreenBitmap);
}

private void pause()
{
    _paused = true;
}

private void resume()
{
    if (_running)
    {
        // Pause the render loop
        _paused = false;
        synchronized (this) { notifyAll(); }
    }
    else
    {
        // Start the render thread.
        _running = true;
        new Thread(this).start();
    }
}