BlackBerry Java Application UI and Navigation

Version: 5.0

Development Guide

Published: 2010-08-20
SWD-810476-0820021303-001

Contents

Creating a UI that is consistent with standard BlackBerry UIs 1

You can use the standard MIDP APIs and the BlackBerry® UI APIs to create a BlackBerry® Java® Application UI.

The UI components in the BlackBerry UI APIs are designed to provide layouts and behaviors that are consistent with BlackBerry® Device Software applications.

- Screen components can provide a standard screen layout, a default menu, and standard behavior when a BlackBerry device user presses the Escape key, clicks the trackwheel, trackball, trackpad, or touches the screen.
- Field components can provide the standard UI elements for date selection, option button, check box, list, text field, label, and progress bar controls.
- Layout managers can provide an application with the ability to arrange the components on a BlackBerry device screen in standard ways, such as horizontally, vertically, or in a left-to-right flow.

You can use the BlackBerry UI APIs to create a UI that includes tables, grids, and other specialized features. You can use a standard Java event model to receive and respond to specific types of events. A BlackBerry device application can receive and respond to user events, such as when the user clicks the trackball or types on the keyboard, and to system events, such as global alerts, clock changes, and USB port connections.

BlackBerry device user input and navigation 2

BlackBerry® devices include a keyboard, a trackwheel, trackball, or trackpad, and an Escape key, for input and navigation. On BlackBerry devices with a SurePress™ touch screen, clicking the screen is equivalent to clicking the trackball or trackwheel.

A BlackBerry® Java® Application should use the following input and navigation model as closely as possible.

- Clicking the trackwheel, trackball, trackpad, or touch screen typically invokes a menu.
- Pressing the Escape key cancels actions or returns to the previous screen. Pressing the Escape key repeatedly returns users to the Home screen. Holding the Escape key closes the browser or media application.

By default, the BlackBerry screen objects provide this functionality without customization; however, you must add menu items and additional UI and navigation logic.

Touch screen

On BlackBerry® devices with a SurePress™ touch screen, users use a finger to interact with the applications on the device. Users type text and navigate screens by performing various actions on the touch screen.

Users can also perform actions by clicking icons on the shortcut bar or by pressing the Menu key.

On BlackBerry devices with a touch screen, users can perform the following actions:

Action	Result
touch the screen lightly	This action highlights an item. In a text field, if users touch the screen near the cursor, an outlined box displays around the cursor. This box helps users reposition the cursor more easily.
tap the screen	In applications that support a full-screen view, such as BlackBerry® Maps and the BlackBerry® Browser, this action hides and shows the shortcut bar.
tap the screen twice	On a web page, map, picture, or presentation attachment, this action zooms in to the web page, map, picture, or presentation attachment.
hold a finger on an item	On the shortcut bar, this action displays a tooltip that describes the action that the icon represents. In a message list, if users hold a finger on the sender or subject of a message, the BlackBerry device searches for the sender or subject.

Action	Result
touch and drag an item on the screen	This action moves the content on the screen in the corresponding direction. For example, when users touch and drag a menu item, the list of menu items moves in the same direction. In a text field, this action moves the outlined box and the cursor in the same direction.
touch the screen in two locations at the same time	This action highlights the text or the list of items, such as messages, between the two locations. To add or remove highlighted text or items, users can touch the screen at another location.
click (press) the screen	This action initiates an action. For example, when users click an item in a list, the screen that is associated with the item appears. This action is equivalent to clicking the trackwheel, trackball or trackpad. On a map, picture, or presentation attachment, this action zooms in to the map, picture, or presentation attachment. On a web page, this action zooms in to the web page or follows a link. In a text field, this action positions the cursor. If the field contains text, an outlined box appears around the cursor.
slide a finger up or down quickly on the screen	Quickly sliding a finger up displays the next screen. Quickly sliding a finger down displays the previous screen. When the keyboard appears, quickly sliding a finger down hides the keyboard and displays the shortcut bar.
slide a finger to the left or right quickly on the screen	This action displays the next or previous picture or message, or the next or previous day, week, or month in a calendar.
slide a finger up or down on the screen	In the camera, sliding a finger up zooms in to a subject. Sliding a finger down zooms out from a subject.
slide a finger in any direction	This action pans a map or web page. If users zoom in to a picture, this action also pans a picture.
press the Escape key	This action removes the highlight from text or a list of items. On a web page, map, or picture, this action zooms out one level. Users can press the Escape key twice to zoom back to the original view.

Trackball or trackpad

On BlackBerry® devices with a trackball or trackpad, the trackball or trackpad is the primary control for user navigation. Users can perform the following actions:

- Roll the trackball or slide a finger on the trackpad to move the cursor.
- Click the trackball or trackpad to perform default actions or open a context menu.
- Click the trackball or trackpad while pressing the Shift key to highlight text or highlight messages in a message list.

BlackBerry devices with a trackball or trackpad also include a Menu key that is located to the left of the trackball or trackpad. Users can press the Menu key to open a full menu of available actions.

Trackball sensitivity

Trackball sensitivity refers to the amount of trackball movement that the system requires to identify the movement as a navigation event, and to send a navigation event to the software layer. The BlackBerry® device hardware measures physical trackball movement by using units called ticks. When the number of ticks along an axis surpasses the threshold of the system or a BlackBerry device application, a navigation event is sent to the software layer, and the system resets the tick count to zero. Tick counts are also reset to zero after a certain amount of idle time passes.

You can use the Trackball API to set the trackball sensitivity. High trackball sensitivity equates to a smaller tick threshold (a small amount of trackball movement generates a navigation event). Low trackball sensitivity equates to a larger tick threshold (a larger amount of trackball movement is required to generate a navigation event).

Trackball movement

You can use the Trackball API to filter the trackball movement data that the BlackBerry® device hardware sends to the software layer. The Trackball API can filter out movement "noise" or unwanted movements.

You can also use the Trackball API to change settings such as trackball movement acceleration. Increasing the trackball movement acceleration setting can result in the software layer identifying trackball movements as moving at a faster rate than the rate detected by the BlackBerry device hardware, as long as the user continually rolls the trackball. The trackball sensitivity temporarily increases as the user rolls the trackball without pausing.

Trackwheel

BlackBerry® devices that precede the BlackBerry® Pearl™ 8100 Series use a trackwheel as the primary control for user navigation. The trackwheel is located on the right side of the BlackBerry® device.

Users can perform the following actions:

- roll the trackwheel to move the cursor vertically
- roll the trackwheel while pressing the Alt key to move the cursor horizontally
- click the trackwheel to initiate an action or open the menu

Keyboard

Users use the keyboard primarily to type text. Character keys send a character to the BlackBerry® device. A modifier key alters the functionality of character keys. Modifier keys include the Shift key and the Alt key. When users press a modifier key, a typing mode indicator appears in the upper-right corner of the screen.

On BlackBerry devices without a touch screen, users can also use the keyboard to move around a screen (for example, to move around a map). However, navigation using the keyboard should always be an alternative to navigation using the trackwheel, trackball, or trackpad.

BlackBerry devices have either a full keyboard or a reduced keyboard. On BlackBerry devices with a SurePress™ touch screen, in most cases, the full keyboard appears in landscape view and the reduced keyboard appears in portrait view.

Interaction methods on BlackBerry devices

BlackBerry device model	Interaction method
BlackBerry® 7100 Series	trackwheel
BlackBerry® 8700 Series	trackwheel
BlackBerry® 8800 Series	trackball
BlackBerry® Bold™ 9000 smartphone	trackball
BlackBerry® Bold™ 9650 smartphone	trackpad
BlackBerry® Bold™ 9700 smartphone	
BlackBerry® Curve™ 8300 Series	trackball
BlackBerry® Curve™ 8500 Series	trackpad
BlackBerry® Curve™ 8900 smartphone	trackball
BlackBerry® Pearl™ 8100 Series	trackball
BlackBerry® Pearl™ Flip 8200 Series	trackball
BlackBerry® Storm™ 9500 Series	touch screen
BlackBerry® Tour™ 9630 smartphone	trackball

Screens

The main component of a BlackBerry® device UI is the `Screen` object.

A BlackBerry device can have multiple screens open at the same time, but only one screen can be displayed at a time. The BlackBerry® Java® Virtual Machine maintains an ordered set of `Screen` objects in a display stack. The screen on the top of the stack is the active screen that the BlackBerry device user sees. When a BlackBerry device application displays a screen, it pushes the screen on the top of the stack. When a BlackBerry device application closes a screen, it pushes the screen off the top of the stack and displays the next screen on the stack, redrawing the screen as necessary.

Each screen can appear only once on the display stack. The BlackBerry JVM throws a runtime exception if a screen that the BlackBerry device application pushes on the stack already exists. A BlackBerry device application must push a screen off the display stack when the user finishes interacting with the screen so that the BlackBerry device application can use memory efficiently. You should use only a few modal screens at one time, because each screen uses a separate thread.

The UI APIs initialize simple `Screen` objects. After you create a screen, you can add fields and a menu, and display the screen to the user by pushing it on the display stack. You can change the BlackBerry device UI and implement new field types, as required. You can also add custom navigation.

The `Screen` class does not implement disambiguation, which is required for complex input methods, such as international keyboards. For the integration of the different input methods, you can extend the `Field` class or one of its subclasses. You should not use `Screen` objects for typing text.

Screen classes

Class	Description
`Screen`	You can use the `Screen` class to create a screen and use a layout manager to lay out the UI components on the screen. You can define a specific type of screen by using the styles that the constants in the `Field` superclass define.
`FullScreen`	You can use the `FullScreen` class to create an empty screen to which you can add UI components in a standard vertical layout. If you want to use another layout style, such as horizontal or diagonal, you can use the `Screen` class instead and add a layout manager to it.
`MainScreen`	You can use the `MainScreen` class to create a screen with the following standard UI components: • default screen title, with a `SeparatorField` after the title • main scrollable section contained in a `VerticalFieldManager`

Class	Description
	• default menu with a Close menu item • default close action that closes the screen when the BlackBerry® device user clicks the Close menu item or presses the Escape key You should consider using a `MainScreen` object for the first screen of your BlackBerry device application to maintain consistency with other BlackBerry device applications.

Manage a drawing area

The `Graphics` object represents the entire drawing surface that is available to the BlackBerry® device application. To limit this area, divide it into `XYRect` objects. Each `XYPoint` represents a point on the screen, which is composed of an X co-ordinate and a Y co-ordinate.

1. Import the following classes:
   - `net.rim.device.api.ui.Graphics`
   - `net.rim.device.api.ui.XYRect`
   - `net.rim.device.api.ui.XYPoint`
2. Create `XYPoint` objects, to represent the top left and bottom right points of a rectangle. Create an `XYRect` object using the `XYPoint` objects, to create a rectangular clipping region.

```
XYPoint bottomRight = new XYPoint(50, 50);
XYPoint topLeft = new XYPoint(10, 10);
XYRect rectangle = new XYRect(topLeft, bottomRight);
```

3. Invoke `Graphics.pushContext()` to make drawing calls that specify that the region origin should not adjust the drawing offset. Invoke `Graphics.pushContext()` to push the rectangular clipping region to the context stack. Invoke `Graphics.drawRect()` to draw a rectangle, and invoke `Graphics.fillRect()` to fill the rectangle. Invoke `Graphics.popContext()` to pop the current context off of the context stack.

```
graphics.pushContext(rectangle, 0, 0);
graphics.fillRect(10, 10, 30, 30);
graphics.drawRect(15, 15, 30, 30);
graphics.popContext();
graphics.drawRect(15, 15, 30, 30);
graphics.pushContext(rectangle, 0, 0);
graphics.fillRect(10, 10, 30, 30);
graphics.drawRect(15, 15, 30, 30);
graphics.popContext();
graphics.drawRect(15, 15, 30, 30);
```

4. Invoke `pushRegion()` and specify that the region origin should adjust the drawing offset. Invoke `Graphics.drawRect()` to draw a rectangle and invoke `Graphics.fillRect()` to fill a rectangle. Invoke `Graphics.popContext()` to pop the current context off of the context stack.

```
graphics.pushRegion(rectangle);
graphics.fillRect(10, 10, 30, 30);
graphics.drawRect(15, 15, 30, 30);
graphics.popContext();
```

5. Specify the portion of the `Graphics` object to push onto the stack.
6. After you invoke `pushContext()` (or `pushRegion()`), provide the portion of the `Graphics` object to invert.

```
graphics.pushContext(rectangle);
graphics.invert(rectangle);
graphics.popContext();
```

7. Invoke `translate()`. The `XYRect` is translated from its origin of (1, 1) to an origin of (20, 20). After translation, the bottom portion of the `XYRect` object extends past the bounds of the graphics context and clips it.

```
XYRect rectangle = new XYRect(1, 1, 100, 100);
XYPoint newLocation = new XYPoint(20, 20);
rectangle.translate(newLocation);
```

Creating a screen transition

You can create a screen transition to apply a visual effect that appears when your application opens or closes a screen on a BlackBerry® device. You can create the following types of screen transitions for your application by using the `net.rim.device.api.ui.TransitionContext` class.

Transition	Description
`TRANSITION_FADE`	This transition reveals or removes a screen by fading it in or fading it out. This screen transition creates a fading visual effect by changing the opacity of the screen.
`TRANSITION_SLIDE`	This transition reveals or removes a screen by sliding it on or sliding it off the display on the device. You can use attributes to specify that both the new screen and the current screen slide, to create an effect where both screens appear to move, or that the new screen slides over the current screen.
`TRANSITION_WIPE`	This transition reveals or removes a screen by simulating wiping on or wiping off the display on the device.
`TRANSITION_ZOOM`	This transition reveals or removes a screen by zooming it in or zooming it out of the display on the device.
`TRANSITION_NONE`	No transition occurs.

Each type of screen transition has attributes that you can use to customize the visual effects of the screen transition. For example, you can customize a sliding effect so that a screen slides from the bottom of the display on the device to the top of the display. If you do not customize the screen transition, the application uses the default attributes. For more information on the default attributes, see the API reference for the BlackBerry® Java® Development Environment.

After you create a screen transition, you must register it within your application by invoking `UiEngineInstance.setTransition()` and specify the outgoing screen to remove and the incoming screen to reveal, the events that cause the transition to occur, and the transition to display.

To download a sample application that demonstrates how to use screen transitions, visit www.blackberry.com/go/screentransitionssample. For more information about screen transitions, see the API reference for the BlackBerry Java Development Environment.

Note: The theme on the BlackBerry device controls the visual effects that display when a user opens an application. For more information, see the *BlackBerry Theme Studio User Guide*.

Code sample: Creating a screen transition

The following code sample illustrates a slide transition and a fade transition. When the user opens the application, the first screen appears on the BlackBerry® device and displays a button. When the user clicks the button, a second screen slides onto the display from the right. The second screen automatically fades off of the display after two seconds.

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.ui.decor.*;

public class ScreenTransitionSample extends UiApplication implements
     FieldChangeListener {
    private Screen _secondaryScreen;
    private Runnable _popRunnable;

    public static void main(String[] args) {
        ScreenTransitionSample  theApp = new ScreenTransitionSample ();
        theApp.enterEventDispatcher();
    }

    public ScreenTransitionSample () {
        _secondaryScreen = new FullScreen();
        _secondaryScreen.setBackground(
     BackgroundFactory.createSolidBackground(Color.LIGHTBLUE) );

        LabelField labelField = new LabelField("The screen closes automatically in
     two seconds by using a fade transition");
        _secondaryScreen.add(labelField);

        TransitionContext transition = new
     TransitionContext(TransitionContext.TRANSITION_SLIDE);
```

```
        transition.setIntAttribute(TransitionContext.ATTR_DURATION, 500);
        transition.setIntAttribute(TransitionContext.ATTR_DIRECTION, 
     TransitionContext.DIRECTION_RIGHT);
        transition.setIntAttribute(TransitionContext.ATTR_STYLE, 
     TransitionContext.STYLE_PUSH);

        UiEngineInstance engine = Ui.getUiEngineInstance();
        engine.setTransition(null, _secondaryScreen, UiEngineInstance.TRIGGER_PUSH, 
     transition);

        transition = new TransitionContext(TransitionContext.TRANSITION_FADE);
        transition.setIntAttribute(TransitionContext.ATTR_DURATION, 500);
        transition.setIntAttribute(TransitionContext.ATTR_KIND, 
     TransitionContext.KIND_OUT);
        engine.setTransition(_secondaryScreen, null, UiEngineInstance.TRIGGER_POP, 
     transition);

        MainScreen baseScreen = new MainScreen();
        baseScreen.setTitle("Screen Transition Sample");

        ButtonField buttonField = new ButtonField("View Transition", 
     ButtonField.CONSUME_CLICK) ;
        buttonField.setChangeListener(this);
        baseScreen.add(buttonField);

        pushScreen(baseScreen);

        _popRunnable = new Runnable() {
            public void run() {
                popScreen(_secondaryScreen);
            }
        };
    }

    public void fieldChanged(Field field, int context)
    {
        pushScreen(_secondaryScreen);
        invokeLater(_popRunnable, 2000, false);
    }
}
```

Specifying the orientation and direction of the screen

In touch screen applications, you can take into account both the orientation and the direction of the screen. Orientation relates to the screen's aspect ratio. Direction relates to the drawing area of the screen.

Orientation

The user of a BlackBerry® device with a touch screen can change the orientation of the screen by rotating the device. If a user holds the BlackBerry device with the BlackBerry logo at the top, the screen is in portrait orientation. If the user rotates the device 90 degrees to the left or right, the screen is in landscape orientation.

You can use the `net.rim.device.api.system.Display` class to retrieve the orientation of the screen. The `Display` class contains the constants that correspond to the orientations that the device with a touch screen can use to display information. For example, the portrait, landscape, and square orientations correspond to the `Display.ORIENTATION_PORTRAIT`, `Display.ORIENTATION_LANDSCAPE`, and `Display.ORIENTATION_SQUARE` constants.

To retrieve the orientation of the screen, you can invoke the `Display.getOrientation()` method. This method returns an integer that corresponds to one of the orientations that the BlackBerry device can use.

To detect orientation changes, you can override `Screen.sublayout()`. In that method, invoke `super.sublayout()` and watch for changes in the `width` and `height` values.

Direction

A BlackBerry device with a touch screen can display information on the screen in different directions. Direction refers to the top of the drawing area of the screen, which is relative to the location of the BlackBerry logo. The direction is north when the top of the drawable area is the screen side closest to the BlackBerry logo; the direction is west when the top of the drawable area is to the left of the BlackBerry logo; the direction is east when the top of the drawable area is to the right of the BlackBerry logo.

You can use the `net.rim.device.api.system.Display` class, the `net.rim.device.api.ui.Ui` class, and the `net.rim.device.api.ui.UiEngineInstance` class to control the direction that the BlackBerry device uses to display information on the screen. The `Display` class contains constants that correspond to the directions that the device can use to display information. For example, the north, west, and east directions correspond to the `Display.DIRECTION_NORTH`, `Display.DIRECTION_WEST`, and `Display.DIRECTION_EAST` constants.

You can create an object of the `net.rim.device.api.ui.Ui` class and invoke `Ui.getUiEngineInstance()` to retrieve an object of the `UiEngineInstance` class. Invoking `UiEngineInstance.setAcceptableDirections()` with one of the constants that correspond to directions from the `Display` class sets the direction that the BlackBerry device can use to display information.

Code sample: Retrieving screen orientation

```
switch(Display.getOrientation())
{
   case Display.ORIENTATION_LANDSCAPE:
      Dialog.alert("Screen orientation is landscape"); break;
   case Display.ORIENTATION_PORTRAIT:
      Dialog.alert("Screen orientation is portrait"); break;
   case Display.ORIENTATION_SQUARE:
      Dialog.alert("Screen orientation is square"); break;
   default:
      Dialog.alert("Screen orientation is not known"); break;
}
```

Code sample: Forcing portrait view in a BlackBerry API application

```
// Use code like this before invoking UiApplication.pushScreen()
int direction = Display.DIRECTION_NORTH;
Ui.getUiEngineInstance().setAcceptableDirections(direction);
```

Code sample: Forcing landscape view in a MIDlet application

```
// Use code like this before invoking Display.setCurrent() in the MIDlet constructor
DirectionControl dc =
  (DirectionControl) ((Controllable) Display.getDisplay(this)).
    getControl("net.rim.device.api.lcdui.control.DirectionControl");
int directions = DirectionControl.DIRECTION_EAST | DirectionControl.DIRECTION_WEST;
dc.setAcceptableScreenDirections(directions);
```

Retrieve the orientation of the touch screen

1. Import the `net.rim.device.api.system.Display` class.
2. Invoke `net.rim.device.api.system.Display.getOrientation()`.

   ```
   int orientation = Display.getOrientation();
   ```

Restrict the touch screen direction

1. Import the following classes:
   - `net.rim.device.api.ui.Ui`
   - `net.rim.device.api.ui.UiEngineInstance`
2. Invoke `net.rim.device.api.ui.Ui.getUiEngineInstance()`.

   ```
   UiEngineInstance _ue;
   _ue = Ui.getUiEngineInstance();
   ```

3. Invoke `net.rim.device.api.ui.UiEngineInstance.setAcceptableDirections(byte flags)` and pass the argument for the direction of the `Screen`.

   ```
   _ue.setAcceptableDirections(Display.DIRECTION_WEST);
   ```

Receive notification when the size of the drawable area of the touch screen changes

1. Import the following classes:
   - `javax.microedition.lcdui.Canvas`
   - `net.rim.device.api.ui.component.Dialog`
2. Extend `javax.microedition.lcdui.Canvas`.
3. Override `Canvas.sizeChanged(int, int)`.

```
protected void sizeChanged(int w, int h) {
   Dialog.alert("The size of the Canvas has changed");
}
```

Accelerometer

A BlackBerry® device with a touch screen includes an accelerometer, which is designed to sense the orientation and acceleration of the BlackBerry device. When a BlackBerry device user moves the BlackBerry device, the accelerometer can sense the movement in 3-D space along the x, y, and z axis. A device user can change the orientation of the device which can change the display direction of a screen for a BlackBerry device application between portrait and landscape.

You can use the Accelerometer APIs in the `net.rim.device.api.system` package to respond to the orientation and acceleration of a BlackBerry device. For example, you can manipulate a game application to change the direction and speed of a moving object on the screen as a user moves and rotates the BlackBerry device at different speeds.

To download a sample application that demonstrates how to use the accelerometer, visit www.blackberry.com/go/accelerometersample. For more information about the sample application, visit www.blackberry.com/go/devguides to read the *Accelerometer Sample Application Overview*.

Types of accelerometer data

A BlackBerry® device application can retrieve data from the accelerometer.

Data type	Description
orientation	The orientation of the BlackBerry device with respect to the ground.
acceleration	The acceleration of the rotation of the BlackBerry device.

For more information about types of data from the accelerometer, see the API reference for the BlackBerry® Java® Development Environment.

Retrieving accelerometer data

The accelerometer is designed to track the direction and speed of the movement along an x, y, and z axis when a BlackBerry® device user moves the BlackBerry device. The x axis is parallel to the width of the BlackBerry device. The y axis is parallel to the length of the BlackBerry device. The z axis is parallel to the depth (front to back) of the BlackBerry device. For more information about the x, y, and z axes of the accelerometer, see the `net.rim.device.api.system.AccelerometerSensor` class in the API reference for the BlackBerry® Java® Development Environment.

You can enable a BlackBerry device application to react to the acceleration of the BlackBerry device that includes an accelerometer. For example, a BlackBerry device user could move the BlackBerry device to control the direction and speed of an object that moves through a maze in a game application.

You can use the Accelerometer APIs, in the `net.rim.device.api.system` package, to react to the acceleration of the BlackBerry device. You must first determine whether the BlackBerry device supports an accelerometer by invoking `net.rim.device.api.system.AccelerometerSensor.isSupported()`. If the method returns the value `true`, the BlackBerry device supports an accelerometer.

You can use the `AccelerometerData` class to identify the direction the user moves the BlackBerry device. Invoking `AccelerometerData.getOrientation()` returns one of the `AccelerometerSensor` class constants that correspond to the direction of the BlackBerry device. For example, if `AccelerometerData.getOrientation()` returns a value equal to `AccelerometerSensor.ORIENTATION_LEFT_UP`, the left side of the BlackBerry device is in an upward direction.

You can use the `AccelerometerSensor` class to retrieve acceleration data from the BlackBerry device. Invoking `AccelerometerSensor.openRawDataChannel()` returns an object of the `net.rim.device.api.system.AccelerometerSensor.Channel` class. The `AccelerometerSensor.Channel` class allows you to open a connection to the accelerometer. You can retrieve data from the accelerometer by invoking `AccelerometerSensor.Channel.getLastAccelerationData()`.

Maintaining a connection to the accelerometer uses power from the BlackBerry device battery. When the BlackBerry device application no longer needs to retrieve data from the accelerometer, you should invoke `AccelerometerSensor.Channel.close()` to close the connection.

Code sample: Retrieving data from the accelerometer

```
short[] xyz = new short[3];
while( running ) {
    channel.getLastAccelerationData(xyz);
}
channel.close();
```

Retrieve accelerometer data at specific intervals

If a BlackBerry® device application opens a channel to the accelerometer when the application is in the foreground, when the application is in the background, the channel pauses and the accelerometer is not queried. If a BlackBerry device application invokes `AccelerometerSensor.Channel. getLastAccelerationData(short[])` at close intervals or when the BlackBerry device is not in motion, the method might return duplicate values.

1. Import the following classes:
   - `net.rim.device.api.system.AccelerometerSensor.Channel;`
   - `net.rim.device.api.system.AccelerometerSensor;`
2. Open a channel to the accelerometer. While a channel is open, the BlackBerry device application will query the accelerometer for information.

   ```
   Channel rawDataChannel = AccelerometerSensor.openRawDataChannel
   ( Application.getApplication() );
   ```

3. Create an array to store the accelerometer data.
   ```
   short[] xyz = new short[ 3 ];
   ```
4. Create a thread.
   ```
    while( running ) {
   ```
5. Invoke `Channel.getLastAccelerationData(short[])` to retrieve data from the accelerometer.
   ```
   rawDataChannel.getLastAccelerationData( xyz );
   ```
6. Invoke `Thread.sleep()` to specify the interval between queries to the accelerometer, in milliseconds.
   ```
   Thread.sleep( 500 );
   ```
7. Invoke `Channel.close()` to close the channel to the accelerometer.
   ```
   rawDataChannel.close();
   ```

Query the accelerometer when the application is in the foreground

1. Import the following classes:
   - `net.rim.device.api.system.AccelerometerChannelConfig`
   - `net.rim.device.api.system.AccelerometerSensor.Channel`
2. Open a channel to retrieve acceleration data from the accelerometer.
   ```
   Channel channel = AccelerometerSensor.openRawAccelerationChannel
   ( Application.getApplication());
   ```
3. Invoke `Thread.sleep()` to specify the interval between queries to the accelerometer, in milliseconds.
   ```
   short[] xyz = new short[ 3 ];
     while( running ) {
          channel.getLastAccelerationData( xyz );
          Thread.sleep( 500 );
     }
   ```
4. Invoke `Channel.close()` to close the channel to the accelerometer.
   ```
   channel.close();
   ```

Query the accelerometer when the application is in the background

1. Import the following classes:
   - `net.rim.device.api.system.AccelerometerChannelConfig;`
   - `net.rim.device.api.system.AccelerometerSensor.Channel;`
2. Create a configuration for a channel to the accelerometer.

```
AccelerometerChannelConfig channelConfig = new AccelerometerChannelConfig
( AccelerometerChannelConfig.TYPE_RAW );
```

3. Invoke `AccelerometerChannelConfig.setBackgroundMode(Boolean)`, to specify support for an application that is in the background.

```
channelConfig.setBackgroundMode( true );
```

4. Invoke `AccelerometerSensor.openChannel()`, to open a background channel to the accelerometer.

```
Channel channel = AccelerometerSensor.openChannel( Application.getApplication(),
channelConfig );
```

5. Invoke `Thread.sleep()` to specify the interval between queries to the accelerometer, in milliseconds.

```
short[] xyz = new short[ 3 ];
  while( running ) {
       channel.getLastAccelerationData( xyz );
       Thread.sleep( 500 );
}
```

6. Invoke `Channel.close()` to close the channel to the accelerometer.

```
channel.close();
```

Store accelerometer readings in a buffer

1. Import the following classes:
   - `net.rim.device.api.system.AccelerometerChannelConfig;`
   - `net.rim.device.api.system.AccelerometerSensor.Channel;`
2. Create a configuration for a channel to the accelerometer.

```
AccelerometerChannelConfig channelConfig = new AccelerometerChannelConfig
( AccelerometerChannelConfig.TYPE_RAW );
```

3. Invoke `AccelerometerChannelConfig.setSamplesCount()`, to specify the number of acceleration readings to store in the buffer. Each element in the buffer contains acceleration readings for the x, y, and z axes and data on when the reading took place.

```
channelConfig.setSamplesCount( 500 );
```

4. Invoke `AccelerometerSensor.openChannel()` to open a channel to the accelerometer.

```
Channel bufferedChannel = AccelerometerSensor.openChannel
( Application.getApplication(), channelConfig );
```

Retrieve accelerometer readings from a buffer

1. Import the following classes:
   - `net.rim.device.api.system.AccelerometerData;`
   - `net.rim.device.api.system.AccelerometerSensor.Channel;`
2. Query the buffer for accelerometer data.

   ```
   AccelerometerData accData = bufferedChannel.getAccelerometerData();
   ```
3. Invoke `AccelerometerData.getNewBatchLength()`, to get the number of readings retrieved since the last query.

   ```
   int newBatchSize = accData.getNewBatchLength();
   ```
4. Invoke `AccelerometerData.getXAccHistory()`, `AccelerometerData.getYAccHistory()`, and `AccelerometerData.getZAccHistory()` to retrieve accelerometer data from the buffer for each axis.

   ```
   short[] xAccel = accData.getXAccHistory();
   short[] yAccel = accData.getYAccHistory();
   short[] zAccel = accData.getZAccHistory();
   ```

Get the time a reading was taken from the accelerometer

1. Import the `net.rim.device.api.system.AccelerometerData` class.
2. Query the buffer for accelerometer data.

   ```
   AccelerometerData accData;
   accData = bufferedChannel.getAccelerometerData();
   ```
3. Invoke `AccelerometerData.getSampleTsHistory()`.

   ```
   long[] queryTimestamps = accData.getSampleTsHistory();
   ```

Events 5

Respond to navigation events

You can use the `Screen` navigation methods to create a BlackBerry® device application. If your existing BlackBerry device application implements the `TrackwheelListener` interface, update your BlackBerry device application to use the `Screen` navigation methods.

1. Import the `net.rim.device.api.ui.Screen` class.
2. Manage navigation events by extending the `net.rim.device.api.ui.Screen` class (or one of its subclasses) and overriding the following navigation methods:

```
navigationClick(int status, int time)
navigationUnclick(int status, int time)
navigationMovement(int dx, int dy, int status, int time)
```

Determine the type of input method

1. Import one or more of the following classes:
   - `net.rim.device.api.ui.Screen`
   - `net.rim.device.api.ui.Field`
2. Import the `net.rim.device.api.system.KeypadListener` interface.
3. Implement the `net.rim.device.api.system.KeypadListener` interface.
4. Extend the `Screen` class, the `Field` class, or both.
5. In your implementation of one of the `navigationClick`, `navigationUnclick`, or `navigationMovement` methods, perform a bitwise AND operation on the `status` parameter to yield more information about the event. Implement the `navigationClick(int status, int time)` method to determine if the trackwheel or a four way navigation input device (for example, a trackball) triggers an event.

```
public boolean navigationClick(int status, int time) {
   if ((status & KeypadListener.STATUS_TRACKWHEEL) ==
KeypadListener.STATUS_TRACKWHEEL) {
      //Input came from the trackwheel
   } else if ((status & KeypadListener.STATUS_FOUR_WAY) ==
KeypadListener.STATUS_FOUR_WAY) {
      //Input came from a four way navigation input device
   }
   return super.navigationClick(status, time);
}
```

For more information about status modifiers for the `net.rim.device.api.system.KeypadListener` class, see the API reference for the BlackBerry® Java® Development Environment.

Responding to touch screen events

BlackBerry® device users can perform the same actions on a device with a touch screen as they can on a BlackBerry device with a trackball. For example, BlackBerry device users can use the touch screen to open a menu, scroll through a list of options, and select an option.

If you use a version of the BlackBerry® Java® Development Environment earlier than version 4.7 to create a BlackBerry device application, you can alter the .jad file of the application to enable the application to respond when a BlackBerry device user touches the touch screen. In the .jad file, you would set the `RIM-TouchCompatibilityMode` option to `false`.

If you use BlackBerry JDE version 4.7 or later to create a BlackBerry device application, you can enable the application to identify the action the BlackBerry device user performs on the touch screen by extending the `net.rim.device.api.ui.Screen` class, the `net.rim.device.api.ui.Field` class, or one of the subclasses of the `Field` class and overriding the `touchEvent()` method. Within the `touchEvent()` method, compare the value that `TouchEvent.getEvent()` returns to the constants from the `net.rim.device.api.ui.TouchEvent` class and the `net.rim.device.api.ui.TouchGesture` class.

The `TouchEvent` class contains the constants that represent the different actions that a user can perform on the touch screen. For example, the click, touch, and move actions correspond to the `TouchEvent.CLICK`, `TouchEvent.DOWN`, and `TouchEvent.MOVE` constants.

The `TouchGesture` class contains the constants that represent the different gestures that a user can perform on a touch screen. For example, the tap and swipe gestures correspond to the `TouchGesture.TAP` and `TouchGesture.SWIPE` constants.

Code sample: Identifying the action a user performs on the touch screen

The following code sample uses a `switch` statement to identify the action.

```
protected boolean touchEvent(TouchEvent message) {
switch(message.getEvent()) {
    case TouchEvent.CLICK:
    Dialog.alert("A click action occurred");
    return true;

case TouchEvent.DOWN:
    Dialog.alert("A down action occurred");
    return true;

case TouchEvent.MOVE:
    Dialog.alert("A move action occurred");
    return true;
}
return false;
}
```

Respond to system events while the user touches the screen

1. Import the following classes:
   - `net.rim.device.api.ui.TouchEvent`
   - `net.rim.device.api.ui.Field`
   - `net.rim.device.api.ui.Manager`
   - `net.rim.device.api.ui.Screen`
   - `net.rim.device.api.ui.component.Dialog`
2. Create a class that extends the `Manager` class, the `Screen` class, the `Field` class, or one of the `Field` subclasses.

   ```
   public class newButtonField extends ButtonField {
   ```
3. In your implementation of the `touchEvent(TouchEvent message)` method, invoke `TouchEvent.getEvent()`.
4. Check if the value that `TouchEvent.getEvent()` returns is equal to `TouchEvent.CANCEL`.

   ```
   protected boolean touchEvent(TouchEvent message) {
   switch(message.getEvent()) {
     case TouchEvent.CANCEL:
      Dialog.alert("System event occurred while processing touch events");
       return true;
      }
    return false;
   }
   ```

Respond to a user sliding a finger up quickly on the screen

1. Import the following classes:
   - `net.rim.device.api.ui.TouchEvent`
   - `net.rim.device.api.ui.TouchGesture`
   - `net.rim.device.api.ui.Field`
   - `net.rim.device.api.ui.Manager`
   - `net.rim.device.api.ui.Screen`
   - `net.rim.device.api.ui.component.Dialog`
2. Create a class that extends the `Manager` class, the `Screen` class, the `Field` class, or one of the `Field` subclasses.

   ```
   public class newButtonField extends ButtonField {
   ```
3. In your implementation of the `touchEvent(TouchEvent message)` method, invoke `TouchEvent.getEvent()`.

4. Check if the value that `TouchGesture.getSwipeDirection()` returns is equal to `TouchGesture.SWIPE_NORTH`.

```
protected boolean touchEvent(TouchEvent message) {
   switch(message.getEvent()) {
      case TouchEvent.GESTURE:
         TouchGesture gesture = message.getGesture();
         switch(gesture.getEvent()) {
            case TouchGesture.SWIPE:
               if(gesture.getSwipeDirection() == TouchGesture.SWIPE_NORTH) {
                  Dialog.alert("Upward swipe occurred");
                  return true;
               }
         }
         return false;
   }
}
```

Respond to a user sliding a finger down quickly on the screen

1. Import the following classes:
   - `net.rim.device.api.ui.TouchEvent`
   - `net.rim.device.api.ui.TouchGesture`
   - `net.rim.device.api.ui.Field`
   - `net.rim.device.api.ui.Manager`
   - `net.rim.device.api.ui.Screen`
   - `net.rim.device.api.ui.component.Dialog`
2. Create a class that extends the `Manager` class, the `Screen` class, the `Field` class, or one of the `Field` subclasses.

```
public class newButtonField extends ButtonField {
```

3. In your implementation of the `touchEvent(TouchEvent message)` method, invoke `TouchEvent.getEvent()`.
4. Check if the value that `TouchGesture.getSwipeDirection()` returns is equal to `TouchGesture.SWIPE_SOUTH`.

```
protected boolean touchEvent(TouchEvent message) {
   switch(message.getEvent()) {
   case TouchEvent.GESTURE:
      TouchGesture gesture = message.getGesture();
      switch(gesture.getEvent()) {
         case TouchGesture.SWIPE:
            if(gesture.getSwipeDirection() == TouchGesture.SWIPE_SOUTH) {
               Dialog.alert("Downward swipe occurred");
               return true;
            }
      }
```

```
        return false;
    }
}
```

Respond to a user sliding a finger to the left quickly on the screen

1. Import the following classes:
   - `net.rim.device.api.ui.TouchEvent`
   - `net.rim.device.api.ui.TouchGesture`
   - `net.rim.device.api.ui.Field`
   - `net.rim.device.api.ui.Manager`
   - `net.rim.device.api.ui.Screen`
   - `net.rim.device.api.ui.component.Dialog`
2. Create a class that extends the `Manager` class, the `Screen` class, the `Field` class, or one of the `Field` subclasses.

```
public class newButtonField extends ButtonField {
```

3. In your implementation of the `touchEvent(TouchEvent message)` method, invoke `TouchEvent.getEvent()`.
4. Check if the value that `TouchGesture.getSwipeDirection()` returns is equal to `TouchGesture.SWIPE_EAST`.

```
protected boolean touchEvent(TouchEvent message) {
    switch(message.getEvent()) {
        case TouchEvent.GESTURE:
            TouchGesture gesture = message.getGesture();
            switch(gesture.getEvent()) {
                case TouchGesture.SWIPE:
                    if(gesture.getSwipeDirection() == TouchGesture.SWIPE_EAST) {
                        Dialog.alert("Eastward swipe occurred");
                        return true;
                    }
            }
            return false;
    }
}
```

Respond to a user sliding a finger to the right quickly on the screen

1. Import the following classes:
   - `net.rim.device.api.ui.TouchEvent`
   - `net.rim.device.api.ui.TouchGesture`
   - `net.rim.device.api.ui.Field`
   - `net.rim.device.api.ui.Manager`
   - `net.rim.device.api.ui.Screen`

- `net.rim.device.api.ui.component.Dialog`

2. Create a class that extends the `Manager` class, the `Screen` class, the `Field` class, or one of the `Field` subclasses.

```
public class newButtonField extends ButtonField {
```

3. In your implementation of the `touchEvent(TouchEvent message)` method, invoke `TouchEvent.getEvent()`.
4. Check if the value that `TouchGesture.getSwipeDirection()` returns is equal to `TouchGesture.SWIPE_WEST`.

```
protected boolean touchEvent(TouchEvent message) {
   switch(message.getEvent()) {
      case TouchEvent.GESTURE:
         TouchGesture gesture = message.getGesture();
         switch(gesture.getEvent()) {
            case TouchGesture.SWIPE:
               if(gesture.getSwipeDirection() == TouchGesture.SWIPE_WEST) {
                  Dialog.alert("Westward swipe occurred");
                  return true;
               }
         }
         return false;
   }
}
```

Respond to a user clicking the screen

1. Import the following classes:
   - `net.rim.device.api.ui.TouchEvent`
   - `net.rim.device.api.ui.Field`
   - `net.rim.device.api.ui.Manager`
   - `net.rim.device.api.ui.Screen`
   - `net.rim.device.api.ui.component.Dialog`
2. Create a class that extends the `Manager` class, the `Screen` class, the `Field` class, or one of the `Field` subclasses.

```
public class newButtonField extends ButtonField {
```

3. In your implementation of the `touchEvent(TouchEvent message)` method, invoke `TouchEvent.getEvent()`.
4. Check if the value that `TouchEvent.getEvent()` returns is equal to `TouchEvent.CLICK`.

```
protected boolean touchEvent(TouchEvent message) {
   switch(message.getEvent()) {
      case TouchEvent.CLICK:
         Dialog.alert("CLICK occurred");
         return true;
   }
   return false;
}
```

Respond to a user touching the screen twice quickly

1. Import the following classes:
   - `net.rim.device.api.ui.TouchEvent`
   - `net.rim.device.api.ui.TouchGesture`
   - `net.rim.device.api.ui.Field`
   - `net.rim.device.api.ui.Manager`
   - `net.rim.device.api.ui.Screen`
   - `net.rim.device.api.ui.component.Dialog`
2. Create a class that extends the `Manager` class, the `Screen` class, the `Field` class, or one of the `Field` subclasses.

```
public class newButtonField extends ButtonField {
```

3. In your implementation of the `touchEvent(TouchEvent message)` method, check for the occurrence of a `TouchGesture.TAP` event and that `TouchGesture.getTapCount` returns 2.

```
protected boolean touchEvent(TouchEvent message) {
   switch(message.getEvent()) {
      case TouchEvent.GESTURE:
         TouchGesture gesture = message.getGesture();
         switch(gesture.getEvent()) {
            case TouchGesture.TAP:
               if(gesture.getTapCount() == 2) {
                  Dialog.alert("Double tap occurred");
                  return true;
               }
          }
   }
 return false;
}
```

Respond to a user touching and dragging an item on the screen

1. Import the following classes:
   - `net.rim.device.api.ui.TouchEvent`
   - `net.rim.device.api.ui.Field`
   - `net.rim.device.api.ui.Manager`
   - `net.rim.device.api.ui.Screen`
   - `net.rim.device.api.ui.component.Dialog`
2. Create a class that extends the `Manager` class, the `Screen` class, the `Field` class, or one of the `Field` subclasses.

```
public class newButtonField extends ButtonField {
```

3. In your implementation of the `touchEvent(TouchEvent message)` method, invoke `TouchEvent.getEvent()`.
4. Check if the value that `TouchEvent.getEvent()` returns is equal to `TouchEvent.MOVE`.

```
protected boolean touchEvent(TouchEvent message) {
   switch(message.getEvent()) {
      case TouchEvent.MOVE:
         Dialog.alert("Move event occurred");
         return true;
   }
   return false;
}
```

Respond to a user touching the screen lightly

1. Import the following classes:
   - `net.rim.device.api.ui.TouchEvent`
   - `net.rim.device.api.ui.TouchGesture`
   - `net.rim.device.api.ui.Field`
   - `net.rim.device.api.ui.Manager`
   - `net.rim.device.api.ui.Screen`
   - `net.rim.device.api.ui.component.Dialog`
2. Create a class that extends the `Manager` class, the `Screen` class, the `Field` class, or one of the `Field` subclasses.

```
public class newButtonField extends ButtonField {
```

3. In your implementation of the `touchEvent(TouchEvent message)` method, invoke `TouchEvent.getEvent()`.
4. Check if the value that `TouchEvent.getEvent()` returns is equal to `TouchEvent.DOWN`.

```
protected boolean touchEvent(TouchEvent message) {
   switch(message.getEvent()) {
      case TouchEvent.DOWN:
         Dialog.alert("Touch occurred");
         return true;
   }
   return false;
}
```

Respond to a scroll action

1. Import the following classes:
   - `net.rim.device.api.ui.TouchEvent`
   - `net.rim.device.api.ui.TouchGesture`
   - `net.rim.device.api.ui.Field`

- `net.rim.device.api.ui.Manager`
- `net.rim.device.api.ui.Screen`

2. Create a class that extends the `Manager` class.

```
 public class newManager extends Manager {
```

3. In your implementation of the `touchEvent(TouchEvent message)` method, check if the value `TouchEvent.getEvent()` returns is equal to `TouchEvent.MOVE`.

```
protected boolean touchEvent(TouchEvent message) {
   switch(message.getEvent()) {
      case TouchEvent.MOVE:
      return true;
   }
   return false;
}
```

Respond to a user touching the screen in two locations at the same time

1. Import the following classes:
   - `net.rim.device.api.ui.TouchEvent`
   - `net.rim.device.api.ui.Field`
   - `net.rim.device.api.ui.Manager`
   - `net.rim.device.api.ui.Screen`
   - `net.rim.device.api.ui.component.Dialog`
2. Create a class that extends the `Manager` class, the `Screen` class, the `Field` class, or one of the `Field` subclasses.

```
public class newButtonField extends ButtonField {
```

3. In your implementation of the `touchEvent(TouchEvent message)` method, check if the following method invocations return values greater than zero: `TouchEvent.getX(1)`, `TouchEvent.getY(1)`, `TouchEvent.getX(2)`, `TouchEvent.getY(2)`.

```
protected boolean touchEvent(TouchEvent message) {
   switch(message.getEvent()) {
   case TouchEvent.MOVE:
      if(message.getX(1) > 0 && message.getY(1) > 0) {
         Dialog.alert("First finger touched/moved");
      }
      if(message.getX(2) > 0 && message.getY(2) > 0) {
         Dialog.alert("Second finger touched/moved");
      }
      return true;
   }
   return false;
}
```

Event injection

You can generate UI events programmatically by using the `EventInjector` class and its inner classes. On BlackBerry® devices that run BlackBerry® Device Software version 5.0 or later and have touch screens, you can inject touch events such as swipes and taps. You can use one of the `EventInjector` inner classes to model an event and you can use the `invokeEvent()` method to inject the event. The event is sent to the application that is currently selected and ready to accept input.

You can use event injection to automate testing. You can also use event injection to provide a way for peripherals to interact with a BlackBerry device. You can also use it in accessible applications such as speech-to-text converters. For a sample application that demonstrates event injection, visit www.blackberry.com/go/toucheventinjectorsampleapp to download the Touch Event Injector sample application. For more information about the sample application, visit www.blackberry.com/go/docs/developers to read the *Touch Event Injector Sample Application Overview*.

Arranging UI components 6

You can arrange the UI components on an application screen by using BlackBerry® API layout managers. The following classes extend the `Manager` class that is provided in the `net.rim.device.apu.ui` package and provide predefined layouts for the UI components on your application's screen.

Layout manager	Description
`FlowFieldManager`	This layout manager arranges UI components vertically and then horizontally depending on the size of the screen. The first UI component is positioned in the upper-left corner of the screen and subsequent components are placed horizontally to the right of the first component until the width of the screen is reached. Once UI components can no longer fit on the first row, the next UI component is placed below the first row of components on a row that has a height that is equal to the tallest component of the row above it. You can apply vertical style bits (for example, `Field.FIELD_TOP`, `Field.FIELD_BOTTOM`, or `Field.FIELD_VCENTER`) to align UI components vertically within their row.
`HorizontalFieldManager`	This layout manager arranges UI components in a single horizontal row starting at the left side of the screen and ending at the right side of the screen. Because this layout manager arranges UI components horizontally, you cannot apply horizontal style bits to UI components (for example, `Field.FIELD_LEFT`, `Field.FIELD_HCENTER`, or `Field.FIELD_RIGHT`). You can apply vertical style bits (for example, `Field.FIELD_TOP`, `Field.FIELD_BOTTOM`, or `Field.FIELD_VCENTER`). If the UI components do not fit the available width of the screen, you should use the `Manager.HORIZONTAL_SCROLL` style bit to enable horizontal scrolling. Otherwise, the screen displays as many UI components as possible within the available screen width, and the rest are not shown. The UI components exist but are not visible. This situation can create unexpected scrolling behavior for your users.
`VerticalFieldManager`	This layout manager arranges UI components in a single vertical column starting at the top of the screen and ending at the bottom of the screen. Because this layout manager is designed to arrange items vertically, you cannot apply vertical style bits to UI components (for example, `Field.FIELD_TOP`,

Layout manager	Description
	Field.FIELD_BOTTOM, or Field.FIELD_VCENTER). You can apply horizontal style bits (for example, Field.FIELD_LEFT, Field.FIELD_HCENTER, or Field.FIELD_RIGHT).

You can use additional layout managers to arrange UI components in your application. For example, you can use the `GridFieldManager` layout manager to position UI components in rows and columns on a screen to create a grid. You can use the `EyelidFieldManager` layout manager to display UI components on a pair of managers that appear at the top and bottom of the screen temporarily.

Arrange UI components

1. Import the required classes and interfaces.

   ```
   net.rim.device.api.ui.container.HorizontalFieldManager;
   net.rim.device.api.ui.component.ButtonField;
   ```

2. Create an instance of a `HorizontalFieldManager`.

   ```
   HorizontalFieldManager _fieldManagerBottom = new HorizontalFieldManager();
   ```

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

   ```
   myScreen.add(_fieldManagerBottom);
   ```

4. Create an instance of a `ButtonField`.

   ```
   ButtonField mySubmitButton = new ButtonField("Submit");
   ```

5. Add the `ButtonField` to the `HorizontalFieldManager`.

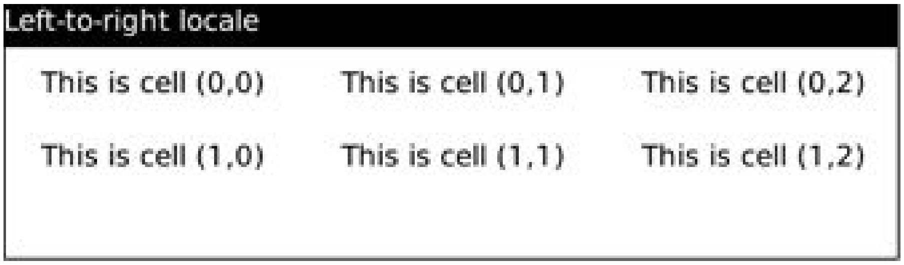
   ```
   _fieldManagerBottom.add(mySubmitButton);
   ```

Creating a grid layout

Note: For information on creating a grid layout in the BlackBerry® Java® Development Environment before version 5.0, visit http://www.blackberry.com/knowledgecenterpublic to read DB-00783.

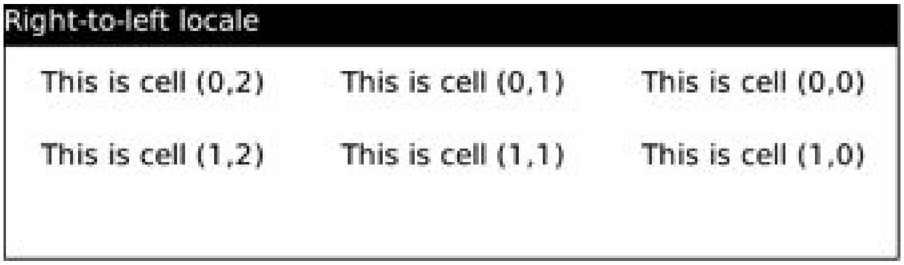
You can position fields in rows and columns on a screen to create a grid by using the `GridFieldManager` class. When you create a grid, you can specify the number of rows and columns. After you create a grid, you cannot change the number of rows and columns that it contains.

Grids are zero-indexed, so the first cell is located at row 0, column 0. In a locale with a left-to-right text direction, the first cell is in the upper-left corner of the grid.


Left-to-right locale

This is cell (0,0)	This is cell (0,1)	This is cell (0,2)
This is cell (1,0)	This is cell (1,1)	This is cell (1,2)

In a locale with a right-to-left text direction, the first cell is in the upper-right corner of the grid.


Right-to-left locale

This is cell (0,2)	This is cell (0,1)	This is cell (0,0)
This is cell (1,2)	This is cell (1,1)	This is cell (1,0)

You can add fields to a grid sequentially (left-to right, top-to-bottom in locales with a left-to-right text direction; right-to-left, top-to-bottom in locales with a right-to-left text direction) or by specifying a row and column in the grid. You can delete fields, insert fields, specify the spacing between columns and rows, and retrieve a grid's properties.

Grids do not have defined heading rows or heading columns. You can emulate the appearance of headings by changing the appearance of the fields in the grid's first row or first column.

Grids can scroll horizontally or vertically if the grid's width or height exceeds the screen's visible area.

You can specify column width by invoking `GridFieldManager.setColumnProperty()`, and you can specify row height by invoking `GridFieldManager.setRowProperty()`. When you invoke these methods, you must specify a `GridFieldManager` property.

Property	**Description**
`FIXED_SIZE`	width or height is a fixed size in pixels
`PREFERRED_SIZE`	width or height is a preferred size based on the maximum preferred size of the fields in the column or row (`PREFERRED_SIZE` is the default property)
`PREFERRED_SIZE_WITH_MAXIMUM`	width or height is a preferred size up to a maximum size
`AUTO_SIZE`	width or height is based on available screen space

Create a grid layout

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. In the sample, the `GridScreen` class, described in step 3, represents the custom screen.

```
class GridFieldManagerDemo extends UiApplication
{
    public static void main(String[] args)
    {
        GridFieldManagerDemo theApp = new GridFieldManagerDemo();
        theApp.enterEventDispatcher();
    }

    GridFieldManagerDemo()
    {
        pushScreen(new GridScreen());
    }
}
```

3. Create the framework for the custom screen by extending the `MainScreen` class.

```
class GridScreen extends MainScreen
{

    public GridScreen()
    {
    }
}
```

4. In the screen constructor, invoke `setTitle()` to set the text that you want to appear in the title section of the screen.

```
setTitle("GridFieldManager Demo");
```

5. In the screen constructor, create an instance of the `GridFieldManager` class. Specify the number of rows, the number of columns, and the style of the grid (using a style inherited from `net.rim.device.api.ui.Manager`). Specify `0` for the style to use the default style.

```
GridFieldManager grid;
grid = new GridFieldManager(2,3,0);
```

6. In the screen constructor, invoke `GridFieldManager.add()` to add fields to the grid.

```
grid.add(new LabelField("one"));
grid.add(new LabelField("two"));
grid.add(new LabelField("three"));
grid.add(new LabelField("four"));
grid.add(new LabelField("five"));
```

7. In the screen constructor, invoke the `GridFieldManager` `set()` methods to specify the properties of the grid.

```
grid.setColumnPadding(20);
grid.setRowPadding(20);
```

8. In the screen constructor, invoke `Screen.add()` to add the grid to the screen.

```
add(grid);
```

After you finish:

You can change the grid after you create it. For example, you can add fields, delete fields, or change the grid's properties.

Code sample: Creating a grid layout

```
/*
 * GridFieldManagerDemo.java
 *
 * Research In Motion Limited proprietary and confidential
 * Copyright Research In Motion Limited, 2009
 */
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;

public class GridFieldManagerDemo extends UiApplication
{
    public static void main(String[] args)
    {
        GridFieldManagerDemo theApp = new GridFieldManagerDemo();
        theApp.enterEventDispatcher();
    }

    GridFieldManagerDemo()
    {
        pushScreen(new GridScreen());
    }
}

class GridScreen extends MainScreen
{
    public GridScreen()
    {
        setTitle("GridFieldManager Demo");
        GridFieldManager grid = new GridFieldManager(2,3,0);

        grid.add(new LabelField("one"));
        grid.add(new LabelField("two"));
        grid.add(new LabelField("three"));
        grid.add(new LabelField("four"));
        grid.add(new LabelField("five"));

        grid.setColumnPadding(20);
        grid.setRowPadding(20);

        add(grid);

        // The grid looks like this:

        //    |  one    |  two      | three
        //    |  four   |  five     |

        // insert a cell first row, second column
        grid.insert(new LabelField("insert"), 0, 1);

        // The grid now looks like this:

        //    |  one    |  insert   | two
        //    |  three  |  four     | five

        // delete a cell second row, second column
        grid.delete(1,1);

        // The grid now looks like this:

        //    |  one    |  insert   | two
        //    |  three  |           | five

        // Add field to first unoccupied cell
        grid.add(new LabelField("new"));

        // The grid now looks like this:

        //    |  one    |  insert   | two
        //    |  three  |   new     | five

    }
}
```

Displaying fields on a temporary pair of managers

You can use the `EyelidFieldManager` class to display fields on a pair of managers that appear on the top and bottom of the screen temporarily.

By default, the fields appear when the BlackBerry® device user moves the trackball or, for a device with a touch screen, when a touch event occurs. The fields disappear after a period of inactivity (1.2 seconds by default). You can override these default properties.

There is no limit to the number and size of the fields. If the managers contain more fields than can fit on the screen, the top and bottom managers overlap with the top manager in the foreground.

Display a ButtonField and a LabelField temporarily on the top and bottom of the screen

1. Import the required classes and interfaces.

```
import net.rim.device.api.system.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.ui.extension.container.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `EyelidFieldManagerDemoScreen` class, described in step 3, represents the custom screen.

```
public class EyelidFieldManagerDemo extends UiApplication
{
     public static void main(String[] args)
     {
          EyelidFieldManagerDemo app = new EyelidFieldManagerDemo();
          app.enterEventDispatcher();
     }

     public EyelidFieldManagerDemo()
     {
          pushScreen(new EyelidFieldManagerDemoScreen());
     }
}
```

3. Create the framework for the custom screen by extending the `MainScreen` class.

```
class EyelidFieldManagerDemoScreen extends MainScreen {
{
     public EyelidFieldManagerDemoScreen()
     {
     }
}
```

4. In the screen constructor, invoke `setTitle()` to specify the text that appears in the title section of the screen.

```
setTitle("EyelidFieldManager Demo");
```

5. In the screen constructor, create an instance of the `EyelidFieldManager` class.

```
EyelidFieldManager manager = new EyelidFieldManager();
```

6. In the screen constructor, invoke `EyelidFieldManager.addTop()` to add a `LabelField` object to the top manager of the `EyelidFieldManager`.

```
manager.addTop(new LabelField("Hello World"));
```

7. In the screen constructor, create a `HorizontalFieldManager` object. Invoke `HorizontalFieldManager.add()` to add buttons to the `HorizontalFieldManager`. Invoke `EyelidFieldManager.addBottom()` to add the `HorizontalFieldManager` to the bottom manager of the `EyelidFieldManager`.

```
HorizontalFieldManager buttonPanel = new HorizontalFieldManager
(Field.FIELD_HCENTER | Field.USE_ALL_WIDTH);

buttonPanel.add(new SimpleButton("Button 1"));
buttonPanel.add(new SimpleButton("Button 2"));
manager.addBottom(buttonPanel);
```

8. In the screen constructor, invoke `EyelidFieldManager.setEyelidDisplayTime()` to specify, in milliseconds, the period of inactivity that must elapse before the pair of managers disappear.

```
manager.setEyelidDisplayTime(3000);
```

9. In the screen constructor, invoke `add()` to add the `EyelidFieldManager` to the screen.

```
add(manager);
```

Code sample: Displaying a ButtonField and a LabelField temporarily on the top and bottom of the screen

```
import net.rim.device.api.system.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.ui.extension.container.*;

public class EyelidFieldManagerDemo extends UiApplication
{
     public static void main(String[] args)
     {
          EyelidFieldManagerDemo app = new EyelidFieldManagerDemo();
          app.enterEventDispatcher();
     }

     public EyelidFieldManagerDemo()
     {
          pushScreen(new EyelidFieldManagerDemoScreen());
     }
}

class EyelidFieldManagerDemoScreen extends MainScreen
{
     public EyelidFieldManagerDemoScreen()
     {
          setTitle("EyelidFieldManager Demo");

          EyelidFieldManager manager = new EyelidFieldManager();

          manager.addTop(new LabelField("Hello World"));

          HorizontalFieldManager buttonPanel = new HorizontalFieldManager
(Field.FIELD_HCENTER | Field.USE_ALL_WIDTH);

          buttonPanel.add(new ButtonField("Button 1"));
          buttonPanel.add(new ButtonField("Button 2"));

          manager.addBottom(buttonPanel);

          manager.setEyelidDisplayTime(3000);

          add(manager);
     }
}
```

UI components

Add a UI component to a screen

1. Import the following classes:
   - `net.rim.device.api.ui.component.CheckboxField`
   - `net.rim.device.api.ui.container.MainScreen`
2. Create an instance of a UI component.

   ```
   CheckboxField myCheckbox = new CheckboxField("First checkbox", true);
   ```
3. Add the UI component to your extension of a `Screen` class.

   ```
   mainScreen.add(myCheckbox);
   ```

Aligning a field to a line of text

You can create an application that can align a `Field` object to the natural beginning of a line of text by using the flag `Field.FIELD_LEADING`. For example, if you create a `Field` with the alignment flag `Field.FIELD_LEADING`, and add the `Field` to a `VerticalFieldManager`, if the application starts using either English or Chinese locales for example, the `Field` aligns to the left side of the screen. If the application starts using either Arabic or Hebrew locales, the `Field` aligns to the right side of the screen.

Autocomplete text field

You can use an autocomplete text field to predict what a BlackBerry® device user wants to type, and display a word or phrase before the user types it completely.

When you create an `AutoCompleteField` object, you must associate a `BasicFilteredList` object with it. The `BasicFilteredList` maintains references to data objects that are compared with to produce the list of words and phrases. You can configure which fields in the data objects are compared with and which fields are displayed when a match is found. For example, you can compare the text that the user types with the value of the `DATA_FIELD_CONTACTS_BIRTHDAY` field in the `DATA_SOURCE_CONTACTS` data source, and return the value of the corresponding `DATA_FIELD_CONTACTS_NAME_FULL` field.

There are four types of data that you can bind to a `BasicFilteredList` to use with an `AutoCompleteField`.

You can specify the set of strings to compare with in one of the following ways:

- an array of literal strings
- an array of objects that support `toString()`
- data sources on a BlackBerry device, such as contacts, memos, tasks, and various types of media files

- an array of objects and an array of strings with corresponding indexes

By default, the autocomplete text field displays the set of strings that is returned by the comparison process in a drop-down list. You can configure the appearance of this list by specifying style flags when you create the autocomplete text field. You can change how the list appears and how users can interact with the list.

Create an autocomplete text field from a data set

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.UiApplication;
import net.rim.device.api.ui.container.MainScreen;
import net.rim.device.api.ui.component.AutoCompleteField;
import net.rim.device.api.collection.util.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the constructor, invoke `pushScreen()` to display the custom screen for the application. The `HomeScreen` class, described in step 3, represents the custom screen.

```
public class AutoCompleteFieldApp extends UiApplication
{
    public static void main(String[] args)
    {
        AutoCompleteFieldApp app = new AutoCompleteFieldApp();
        app.enterEventDispatcher();
    }

    AutoCompleteFieldApp()
    {
            pushScreen(new HomeScreen());
    }
}
```

3. Create the custom screen by extending the `MainScreen` class.

```
class HomeScreen extends MainScreen
{
    public HomeScreen()
    {
    }
}
```

4. In the constructor, create a `BasicFilteredList` object. Create a `String` array and store the strings that you want to match against in the array. In this example, the strings are days of the week. Invoke `addDataSet()` to bind the data in the array to the `BasicFilteredList`.

```
BasicFilteredList filterList = new BasicFilteredList();
String[] days =
{"Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sund
    ay"};
filterList.addDataSet(1,days,"days",BasicFilteredList.COMPARISON_IGNORE_CASE);
```

5. In the constructor, create an `AutoCompleteField` object. Pass an instance of the `BasicFilteredList` to the `AutoCompleteField` constructor to bind the `BasicFilteredList` to the autocomplete text field. Invoke `add()` to add the field to the screen.

```
AutoCompleteField autoCompleteField = new
     AutoCompleteField(filterList);
add(autoCompleteField);
```

Code sample: Creating an autocomplete field from a data set

```
import net.rim.device.api.ui.UiApplication;
import net.rim.device.api.ui.container.MainScreen;
import net.rim.device.api.ui.component.AutoCompleteField;
import net.rim.device.api.collection.util.*;


public class AutoCompleteFieldApp extends UiApplication
{
    public static void main(String[] args)
    {
        AutoCompleteFieldApp app = new AutoCompleteFieldApp();
        app.enterEventDispatcher();
    }

    AutoCompleteFieldApp()
    {
           HomeScreen scr = new HomeScreen();
           this.pushScreen(scr);
    }
}

class HomeScreen extends MainScreen
{
    public HomeScreen()
    {
        BasicFilteredList filterList = new BasicFilteredList();
        String[] days = {"Monday","Tuesday","Wednesday",
                         "Thursday","Friday","Saturday","Sunday"};
        filterList.addDataSet(1,days,"days",BasicFilteredList
    .COMPARISON_IGNORE_CASE);
        AutoCompleteField autoCompleteField = new AutoCompleteField(filterList);
        add(autoCompleteField);
    }
}
```

Create an autocomplete text field from a data source

1. Import the required classes and interfaces.

   ```
   import net.rim.device.api.ui.UiApplication;
   import net.rim.device.api.ui.container.MainScreen;
   import net.rim.device.api.ui.component.AutoCompleteField;
   import net.rim.device.api.collection.util.*;
   ```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `HomeScreen` class represents the custom screen that is described in step 3.

   ```
   public class AutoCompleteFieldApp extends UiApplication
   {
       public static void main(String[] args)
       {
           AutoCompleteFieldApp app = new AutoCompleteFieldApp();
           app.enterEventDispatcher();
       }
       public AutoCompleteFieldApp()
       {
               pushScreen(new HomeScreen());
       }
   }
   ```

3. Create the custom screen for the application by extending the `MainScreen` class.

   ```
   class HomeScreen extends MainScreen
   {
       public HomeScreen()
       {
       }
   }
   ```

4. In the screen constructor, create a `BasicFilteredList` object. Invoke `addDataSource()` to bind a data source to the `BasicFilteredList`. In this example, the data is contact information and the data source is the contact list. The first argument that you pass to `addDataSource()` is a unique ID. The second argument binds the `BasicFilteredList` object to a data source. The third argument specifies the set of data source fields to compare with. The fourth argument specifies the set of data source fields to make available when a match is found. In this example, the fields to compare with are the same as the fields to make available when a match is found. The fifth argument specifies the primary display field. The sixth argument specifies the secondary display field and is set to -1 if you do not want to use it. The final argument specifies a name for the `BasicFilteredList`; its value is generated automatically if you specify `null`.

   ```
   BasicFilteredList filterList = new BasicFilteredList();
           filterList.addDataSource(
               1,
   ```

```
            BasicFilteredList.DATA_SOURCE_CONTACTS,

            BasicFilteredList.DATA_FIELD_CONTACTS_NAME_FULL |
            BasicFilteredList.DATA_FIELD_CONTACTS_COMPANY |
            BasicFilteredList.DATA_FIELD_CONTACTS_EMAIL,

            BasicFilteredList.DATA_FIELD_CONTACTS_NAME_FULL |
            BasicFilteredList.DATA_FIELD_CONTACTS_COMPANY |
            BasicFilteredList.DATA_FIELD_CONTACTS_EMAIL,

            BasicFilteredList.DATA_FIELD_CONTACTS_NAME_FULL,
            -1,
            null);
```

5. In the screen constructor, create an `AutoCompleteField` object. Pass the `BasicFilteredList` object that you created in step 4 to the `AutoCompleteField` constructor to bind the `BasicFilteredList` to the autocomplete text field. Invoke `add()` to add the field to the screen.

```
AutoCompleteField autoCompleteField = new
     AutoCompleteField(filterList);
add(autoCompleteField);
```

Code sample: Creating an autocomplete field from a data source

```
import net.rim.device.api.ui.UiApplication;
import net.rim.device.api.ui.container.MainScreen;
import net.rim.device.api.ui.component.AutoCompleteField;
import net.rim.device.api.collection.util.*;


public class AutoCompleteFieldApp extends UiApplication
{
    public static void main(String[] args)
    {
         AutoCompleteFieldApp app = new AutoCompleteFieldApp();
         app.enterEventDispatcher();
    }

       AutoCompleteFieldApp()
       {
           HomeScreen scr = new HomeScreen();
           this.pushScreen(scr);
       }
}

class HomeScreen extends MainScreen
{
    public HomeScreen()
    {
        BasicFilteredList filterList = new BasicFilteredList();
        filterList.addDataSource(1,
            BasicFilteredList.DATA_SOURCE_CONTACTS,
            BasicFilteredList.DATA_FIELD_CONTACTS_NAME_FULL |
BasicFilteredList.DATA_FIELD_CONTACTS_COMPANY |
BasicFilteredList.DATA_FIELD_CONTACTS_EMAIL,
            BasicFilteredList.DATA_FIELD_CONTACTS_NAME_FULL |
BasicFilteredList.DATA_FIELD_CONTACTS_COMPANY |
BasicFilteredList.DATA_FIELD_CONTACTS_EMAIL,
            BasicFilteredList.DATA_FIELD_CONTACTS_NAME_FULL,
            BasicFilteredList.DATA_FIELD_CONTACTS_NAME_FULL,
            null);

        AutoCompleteField autoCompleteField = new AutoCompleteField(filterList);
        add(autoCompleteField);
    }
}
```

Using data sources and fields with an autocomplete text field

You can use the `AutoCompleteField` class and the `BasicFilteredList` class to compare the text that a user types in an autocomplete text field with the values of fields in a specified data source. You specify the fields to use and their data sources by using a `BasicFilteredList` object that you pass as an argument to the constructor of the `AutoCompleteField` class.

Data source	Fields
DATA_SOURCE_APPOINTMENTS	• DATA_FIELD_APPOINTMENTS_ALL • DATA_FIELD_APPOINTMENTS_ATTENDEES • DATA_FIELD_APPOINTMENTS_ORGANIZER • DATA_FIELD_APPOINTMENTS_SUBJECT
DATA_SOURCE_CONTACTS	• DATA_FIELD_CONTACTS_ADDRESS_ALL • DATA_FIELD_CONTACTS_ADDRESS_HOME • DATA_FIELD_CONTACTS_ADDRESS_WORK • DATA_FIELD_CONTACTS_ANNIVERSARY • DATA_FIELD_CONTACTS_BIRTHDAY • DATA_FIELD_CONTACTS_CATEGORIES • DATA_FIELD_CONTACTS_COMPANY • DATA_FIELD_CONTACTS_EMAIL • DATA_FIELD_CONTACTS_FAX • DATA_FIELD_CONTACTS_JOB_TITLE • DATA_FIELD_CONTACTS_NAME_FULL

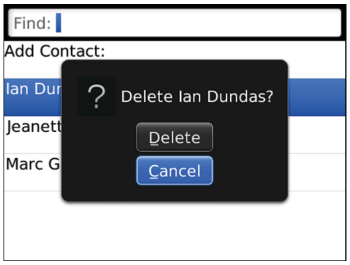
Data source	Fields
	• DATA_FIELD_CONTACTS_NAME_FIRST • DATA_FIELD_CONTACTS_NAME_LAST • DATA_FIELD_CONTACTS_NOTES • DATA_FIELD_CONTACTS_PAGER • DATA_FIELD_CONTACTS_PHONE_ALL • DATA_FIELD_CONTACTS_PHONE_HOME • DATA_FIELD_CONTACTS_PHONE_HOME2 • DATA_FIELD_CONTACTS_PHONE_MOBILE • DATA_FIELD_CONTACTS_PHONE_OTHER • DATA_FIELD_CONTACTS_PHONE_WORK • DATA_FIELD_CONTACTS_PHONE_WORK2 • DATA_FIELD_CONTACTS_PIN
DATA_SOURCE_MEMOS	• DATA_FIELD_MEMOS_TITLE
DATA_SOURCE_MESSAGES	• DATA_FIELD_MESSAGES_ALL • DATA_FIELD_MESSAGES_RECIPIENT • DATA_FIELD_MESSAGES_SENDER • DATA_FIELD_MESSAGES_SUBJECT
DATA_SOURCE_MUSIC	• DATA_FIELD_MUSIC_ALL • DATA_FIELD_MUSIC_ALBUM • DATA_FIELD_MUSIC_ARTIST • DATA_FIELD_MUSIC_GENRE • DATA_FIELD_MUSIC_PLAYLIST • DATA_FIELD_MUSIC_SONG
DATA_SOURCE_PICTURES	• DATA_FIELD_PICTURES_TITLE
DATA_SOURCE_RINGTONES	• DATA_FIELD_RINGTONES_TITLE
DATA_SOURCE_TASKS	• DATA_FIELD_TASKS_TITLE
DATA_SOURCE_VIDEOS	• DATA_FIELD_VIDEOS_TITLE
DATA_SOURCE_VOICENOTES	• DATA_FIELD_VOICENOTES_TITLE

Buttons

Use buttons to allow users to perform an action from a dialog box. Menus typically include actions that are associated with a screen.

Users can perform the following actions with a button:

Action	BlackBerry devices without a touch screen	BlackBerry devices with a touch screen
Highlight a button.	• Roll the trackwheel or trackball. • Slide a finger on the trackpad.	Touch the button lightly.
Perform an action.	Click the button or press the Enter key.	Click the screen.



Best practice: Implementing buttons

- Avoid using buttons on an application screen. To provide actions associated with a screen, use the application menu instead where possible. The menu is available to users immediately, regardless of the position of the cursor on the screen. Buttons are static and require users to highlight the button in order to perform the associated action. If you use buttons, include menu items for the actions in the application menu as well.
- Use check boxes or option buttons for options such as turning on or turning off a feature where possible.
- For the default button, use the button that users are most likely to select. Avoid using a button associated with a destructive action as the default button.

Guidelines for labels

- Use clear, concise labels.
- Use one-word labels where possible. The size of a button changes depending on the length of the label. If a label is too long, an ellipsis (...) indicates that the text is truncated.
- Use verbs for labels that describe the associated action (for example, "Cancel," "Delete," "Discard," or "Save"). If necessary, include more descriptive text elsewhere on the screen (for example, in an application message).
- Avoid using the labels "Yes" and "No."
- Avoid using symbols or graphics in a label.

- Avoid using punctuation in a label. Use an ellipsis in a button label to indicate that users must perform another action after they click the button.

Create a button

1. Import the `net.rim.device.api.ui.component.ButtonField` class.
2. Create an instance of a `ButtonField` using a style parameter.

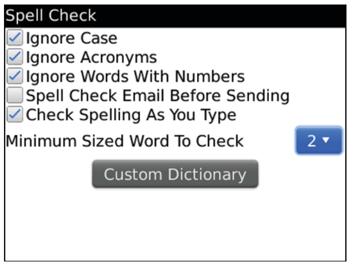
```
ButtonField mySubmitButton = new ButtonField("Submit");
```

Check boxes

Use check boxes for options that users can turn on or turn off.

Users can perform the following action with a check box:

Action	BlackBerry devices without a touch screen	BlackBerry devices with a touch screen
Select a check box.	Press the Space key or click the trackwheel, trackball, or trackpad.	Click the screen.



Spell Check
✓ Ignore Case
✓ Ignore Acronyms
✓ Ignore Words With Numbers
☐ Spell Check Email Before Sending
✓ Check Spelling As You Type
Minimum Sized Word To Check 2

Best practice: Implementing check boxes

- Use check boxes when users can select multiple options.
- Use the `CheckboxField` class to create check boxes.
- Do not start an action when users select a check box. For example, do not open a new screen.
- Align check boxes vertically.
- Group and order check boxes logically (for example, group related options together or include the most common options first). Avoid ordering check boxes alphabetically; alphabetical order is language specific.

Guidelines for labels

- Use clear, concise labels. Verify that the label clearly describes what occurs when users select the check box.

- Use positive labels where possible. For example, if users have the option of turning on or turning off a feature, use "turn on" instead of "turn off" in the label.
- Place labels on the right side of check boxes.
- Use title case capitalization.
- Do not use end punctuation.

Create a check box

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `MyUiScreen` class, which is described in step 3, represents the custom screen.

```
public class MyUi extends UiApplication
{
    public static void main(String[] args)
    {
        MyUi theApp = new MyUi();
        theApp.enterEventDispatcher();
    }
    public MyUi()
    {
        pushScreen(new MyUiScreen());
    }
}
```

3. Create the custom screen for the application by extending the `MainScreen` class. In the screen constructor, invoke `setTitle()` to specify the title for the screen.

```
class MyUiScreen extends MainScreen
{
    public MyUiScreen()
    {
        setTitle("UI Component Sample");
    }
}
```

4. In the screen constructor, create a check box by using the `CheckboxField` class. In the `CheckboxField` constructor, specify the label for the check box and use a Boolean value to indicate whether the check box is the default selection (for example,`true` indicates that by default the check box is selected). Invoke `add()` to add the check box to the screen.

```
add(new CheckboxField("First Check Box", true));
add(new CheckboxField("Second Check Box", false));
```

5. To change the default behavior of a check box, use the styles that are inherited from the `Field` class. For example, to create a check box that users cannot change, use the `READONLY` style.

   ```
   add(new CheckboxField("First Check Box", true, this.READONLY));
   ```

6. To override the default functionality that prompts the user to save changes before the application closes, in the extension of the `MainScreen` class, override the `MainScreen.onSavePrompt()` method. In the following code sample, the return value is `true` which indicates that the application does not prompt the user before closing.

   ```
   public boolean onSavePrompt()
   {
       return true;
   }
   ```

Code sample: Creating a check box

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;

public class MyUi extends UiApplication
{
    public static void main(String[] args)
    {
        MyUi theApp = new MyUi();
        theApp.enterEventDispatcher();
    }
    public MyUi()
    {
        pushScreen(new MyUiScreen());
    }
}
class MyUiScreen extends MainScreen
{
    public MyUiScreen()
    {
        setTitle("UI Component Sample");
        add(new CheckboxField("First Check Box", true));
        add(new CheckboxField("Second Check Box", false));
    }
    public boolean onSavePrompt()
    {
        return true;
    }
}
```

Create a bitmap

1. Import the `net.rim.device.api.ui.component.BitmapField` class.
2. Create an instance of a `BitmapField`.

   ```
   BitmapField myBitmapField = new BitmapField();
   ```

Create a custom field

You can only add custom context menu items and custom layouts to 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. 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 );
}
```

6. Implement `getPreferredWidth()`, using the relative dimensions of the field label to make sure that the label does not exceed the dimensions of the component. 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.

```
public int getPreferredWidth() {
   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;
   }
}
```

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

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

8. Implement `paint()`. The manager of a field invokes `paint()` to redraw the field when an area of the field is marked as invalid. 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 a lines start point and end point. Invoke `graphics.drawLine()` and 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. Invoke `graphics.drawText()` and 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, textWidth;
   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 );
}
```

9. Implement the `Field` `set()` and `get()` methods. 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;
}
```

Creating a field to display web content

Display HTML content in a browser field

1. Import the required classes and interfaces.

```
import net.rim.device.api.browser.field2.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.container.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `BrowserFieldDemoScreen` class, described in step 3, represents the custom screen.

```
public class BrowserFieldDemo extends UiApplication
{
     public static void main(String[] args)
     {
          BrowserFieldDemo app = new BrowserFieldDemo();
          app.enterEventDispatcher();
     }

     public BrowserFieldDemo()
     {
          pushScreen(new BrowserFieldDemoScreen());
     }
}
```

3. Create the custom screen by extending the `MainScreen` class.

   ```
   class BrowserFieldDemoScreen extends MainScreen
   {
        public BrowserFieldDemoScreen()
        {
        }
   }
   ```

4. In the screen constructor, create an instance of the `BrowserField` class.

   ```
   BrowserField myBrowserField = new BrowserField();
   ```

5. In the screen constructor, invoke `add()` to add the `BrowserField` object to the screen.

   ```
   add(myBrowserField);
   ```

6. In the screen constructor, invoke `BrowserField.displayContent()` to specify and display the HTML content.

   ```
   myBrowserField.displayContent("<html><body><h1>Hell
       o World!</h1></body></html>", "http://localhost");
   ```

Code sample: Displaying HTML content in a browser field

```
import net.rim.device.api.browser.field2.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.container.*;

public class BrowserFieldDemo extends UiApplication
{
    public static void main(String[] args)
    {
        BrowserFieldDemo app = new BrowserFieldDemo();
        app.enterEventDispatcher();
    }
    public BrowserFieldDemo()
    {
        pushScreen(new BrowserFieldDemoScreen());
```

```
    }
}

class BrowserFieldDemoScreen extends MainScreen
{
    public BrowserFieldDemoScreen()
    {
        BrowserField myBrowserField = new BrowserField();
        add(myBrowserField);
        myBrowserField.displayContent("<html><body><h1>Hello
     World!</h1></body></html>", "http://localhost");
    }
}
```

Display HTML content from a web page in a browser field

1. Import the required classes and interfaces.

```
import net.rim.device.api.browser.field2.*;
import net.rim.device.api.system.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.container.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `BrowserFieldDemoScreen` class, described in step 3, represents the custom screen.

```
public class BrowserFieldDemo extends UiApplication
{
     public static void main(String[] args)
     {
          BrowserFieldDemo app = new BrowserFieldDemo();
          app.enterEventDispatcher();
     }

     public BrowserFieldDemo()
     {
          pushScreen(new BrowserFieldDemoScreen());
     }
}
```

3. Create the framework for the custom screen by extending the `MainScreen` class.

```
class BrowserFieldDemoScreen extends MainScreen
{
     public BrowserFieldDemoScreen()
     {
     }
}
```

4. In the screen constructor, create an instance of the `BrowserField` class.

   ```
   BrowserField myBrowserField = new BrowserField();
   ```

5. In the screen constructor, invoke `add()` to add the `BrowserField` object to the screen.

   ```
   add(myBrowserField);
   ```

6. In the screen constructor, invoke `BrowserField.requestContent()` to specify the location of the HTML content and display it.

   ```
   myBrowserField.requestContent("http://www.blackberry.com");
   ```

Code sample: Displaying HTML content from a web page in a browser field

```
import net.rim.device.api.browser.field2.*;
import net.rim.device.api.system.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.container.*;

public class BrowserFieldDemo extends UiApplication
{
    public static void main(String[] args)
    {
        BrowserFieldDemo app = new BrowserFieldDemo();
        app.enterEventDispatcher();
    }
    public BrowserFieldDemo()
    {
        pushScreen(new BrowserFieldDemoScreen());
    }
}

class BrowserFieldDemoScreen extends MainScreen
{
    public BrowserFieldDemoScreen()
    {
        BrowserField myBrowserField = new BrowserField();
        add(myBrowserField);
        myBrowserField.requestContent("http://www.blackberry.com");
    }
}
```

Display HTML content from a resource in your application

1. Import the required classes and interfaces.

```
import net.rim.device.api.browser.field2.*;
import net.rim.device.api.system.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.container.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `BrowserFieldDemoScreen` class, described in step 3, represents the custom screen.

```
public class BrowserFieldDemo extends UiApplication
{
     public static void main(String[] args)
     {
          BrowserFieldDemo app = new BrowserFieldDemo();
          app.enterEventDispatcher();
     }

     public BrowserFieldDemo()
     {
          pushScreen(new BrowserFieldDemoScreen());
     }
}
```

3. Create the custom screen by extending the `MainScreen` class.

```
class BrowserFieldDemoScreen extends MainScreen
{
     public BrowserFieldDemoScreen()
     {
     }
}
```

4. In the screen constructor, create an instance of the `BrowserField` class.

```
BrowserField myBrowserField = new BrowserField();
```

5. In the screen constructor, invoke `add()` to add the `BrowserField` object to the screen.

```
add(myBrowserField);
```

6. In the screen constructor, invoke `BrowserField.requestContent()` to specify the location of the resource in your application and display the HTML content.

```
myBrowserField.requestContent("local:///test.html");
```

Note: The BrowserField class does not access resources using a folder structure. The `BrowserField` class displays the first resource found that matches the specifed file name.

Code sample: Displaying HTML content from a resource in your application

```
import net.rim.device.api.browser.field2.*;
import net.rim.device.api.system.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.container.*;

public class BrowserFieldDemo extends UiApplication
{
    public static void main(String[] args)
    {
        BrowserFieldDemo app = new BrowserFieldDemo();
        app.enterEventDispatcher();
    }
    public BrowserFieldDemo()
    {
        pushScreen(new BrowserFieldDemoScreen());
    }
}

class BrowserFieldDemoScreen extends MainScreen
{
    public BrowserFieldDemoScreen()
    {
        BrowserField myBrowserField = new BrowserField();
        add(myBrowserField);
        myBrowserField.requestContent("local:///test.html");
    }
}
```

Configure a browser field

1. Import the required classes and interfaces.

```
import net.rim.device.api.browser.field2.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.container.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `BrowserFieldDemoScreen` class, described in step 3, represents the custom screen.

```
public class BrowserFieldDemo extends UiApplication
{
    public static void main(String[] args)
    {
        BrowserFieldDemo app = new BrowserFieldDemo();
```

```
            app.enterEventDispatcher();
        }

        public BrowserFieldDemo()
        {
            pushScreen(new BrowserFieldDemoScreen());
        }
}
```

3. Create the framework for the custom screen by extending the `MainScreen` class.

```
class BrowserFieldDemoScreen extends MainScreen
{
     public BrowserFieldDemoScreen()
     {
     }
}
```

4. In the screen constructor, create an instance of the `BrowserFieldConfig` class.

```
BrowserFieldConfig myBrowserFieldConfig = new BrowserFieldConfig();
```

5. In the screen constructor, invoke `BrowserFieldConfig.setProperty()` to specify a property of the `BrowserField`. The first parameter in `setProperty()` specifies the property, and the second parameter specifies the value of the property. For example, the following code sample specifies the `NAVIGATION_MODE` property of a `BrowserField` object:

```
myBrowserFieldConfig.setProperty(BrowserFieldConfig.NAVIGATION_MODE,
BrowserFieldConfig.NAVIGATION_MODE_POINTER);
```

6. In the screen constructor, create an instance of the `BrowserField` class that uses the configuration that you defined.

```
BrowserField browserField = new BrowserField(myBrowserFieldConfig);
```

Code sample: Configuring a browser field

```
import net.rim.device.api.browser.field2.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.container.*;

public class BrowserFieldDemo extends UiApplication
{
    public static void main(String[] args)
    {
        BrowserFieldDemo app = new BrowserFieldDemo();
        app.enterEventDispatcher();
    }
    public BrowserFieldDemo()
    {
        pushScreen(new BrowserFieldDemoScreen());
    }
}
```

```
class BrowserFieldDemoScreen extends MainScreen
{
    public BrowserFieldDemoScreen()
    {
        BrowserFieldConfig myBrowserFieldConfig = new BrowserFieldConfig();
        myBrowserFieldConfig.setProperty(BrowserFieldConfig
    .NAVIGATION_MODE,BrowserFieldConfig.NAVIGATION_MODE_POINTER);
        BrowserField browserField = new BrowserField(myBrowserFieldConfig);
    }
}
```

Send form data to a web address in a browser field

1. Import the required classes and interfaces.

   ```
   import net.rim.device.api.browser.field2.*;
   import net.rim.device.api.io.http.*;
   import net.rim.device.api.system.*;
   import net.rim.device.api.ui.*;
   import net.rim.device.api.ui.container.*;
   import java.lang.*;
   import java.util.*;
   ```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `BrowserFieldDemoScreen` class, described in step 3, represents the custom screen.

   ```
   public class BrowserFieldDemo extends UiApplication
   {
        public static void main(String[] args)
        {
             BrowserFieldDemo app = new BrowserFieldDemo();
             app.enterEventDispatcher();
        }

        public BrowserFieldDemo()
        {
             pushScreen(new BrowserFieldDemoScreen());
        }
   }
   ```

3. Create the custom screen by extending the `MainScreen` class.

   ```
   class BrowserFieldDemoScreen extends MainScreen
   {
        public BrowserFieldDemoScreen()
   ```

```
        {
        }
}
```

4. In the screen constructor, create an instance of the `BrowserField` class.

   ```
   BrowserField browserField = new BrowserField();
   ```

5. In the screen constructor, invoke `add()` to add the `BrowserField` object to the screen.

   ```
   add(browserField);
   ```

6. In the screen constructor, create a `String` object that contains the base web address of the web page that you are sending the form data to.

   ```
   String baseURL = "http://www.blackberry.com";
   ```

7. In the screen constructor, create a `String` that specifies the form data that your application sends to the web page.

   ```
   String postData = "fieldname1=value1&fieldname2=value2";
   ```

8. In the screen constructor, create a `Hashtable` object to store the header information for the form data.

   ```
   Hashtable header = new Hashtable();
   ```

9. In the screen constructor, invoke `Hashtable.put()` to specify the header information for the form data.

   ```
   header.put("Content-Length", "" + postData.length());
   header.put("Content-Type", "application/x-www-form-urlencoded");
   ```

10. In the screen constructor, invoke `BrowserField.requestContent()` to send the form data to the web page and display the web page.

    ```
    browserField.requestContent(baseURL, postData.getBytes(), new
         HttpHeaders(header));
    ```

Code sample: Sending form data to a web address in a browser field

```
import net.rim.device.api.browser.field2.*;
import net.rim.device.api.io.http.*;
import net.rim.device.api.system.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.container.*;
import java.lang.*;
import java.util.*;

public class BrowserFieldDemo extends UiApplication
{
    public static void main(String[] args)
    {
        BrowserFieldDemo app = new BrowserFieldDemo();
        app.enterEventDispatcher();
    }
```

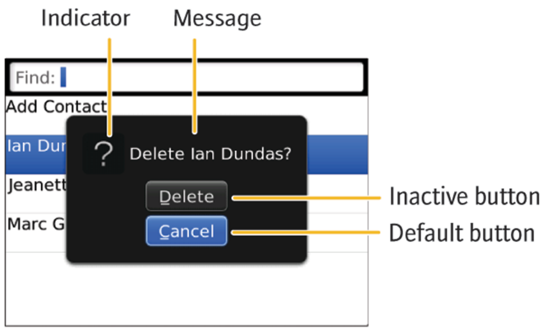
```
    public BrowserFieldDemo()
    {
        pushScreen(new BrowserFieldDemoScreen());
    }
}

class BrowserFieldDemoScreen extends MainScreen
{
    public BrowserFieldDemoScreen()
    {
        BrowserField browserField = new BrowserField();}
        add(browserField);
        String baseURL = "http://www.blackberry.com";
        String postData = "fieldname1=value1&fieldname2=value2";
        Hashtable header = new Hashtable();
        header.put("Content-Length", "" + postData.length());
        header.put("Content-Type", "application/x-www-form-urlencoded");
        browserField.requestContent(baseURL, postData.getBytes(), new
     HttpHeaders(header));
    }
}
```

Dialog boxes

Use dialog boxes to perform the following actions:

- Prompt users for information that is required to complete a user-initiated task.
- Inform users of urgent information or the status of important actions.
- Warn users of unexpected or potentially destructive conditions or situations.

Dialog boxes are modal; they interrupt the normal operation of the BlackBerry® device and are pushed to the top of the stack. A dialog box includes an indicator that indicates the type of dialog box, a message, and buttons that allow users to perform an action. The size of the dialog box depends on the size of the BlackBerry device screen. Scroll arrows appear if necessary. The theme that users select on their BlackBerry device determines the visual style of the dialog box.



Best practice: Implementing dialog boxes

- Always use buttons as controls in dialog boxes. Avoid using links or other controls.
- Use a standard indicator that is appropriate for the type of dialog box. Avoid using multiple indicators in a dialog box.
- Include scroll arrows if your dialog box message or buttons cannot be displayed fully on the dialog box. If you use standard components, scroll arrows appear automatically if necessary.
- Always allow users to use the Escape key to close a dialog box. Avoid implementing another action when users press the Escape key to close a dialog box. For example, if a dialog box allows users to change a setting, do not implement any changes when users press the Escape key. If necessary, display the dialog box at a later time.
- If users press the End/Power key when a dialog box appears on an application screen, display the Home screen or application list. If users return to the application, display the dialog box again.
- Choose a highlight color that has high contrast and clearly implies focus (for example, white text on a blue background). Use the same highlight color for all controls.
- Avoid hardcoding color values. The color values that you choose might not work with the theme that users have selected.

Guidelines for layout

- Center the dialog box on the screen. If you use standard components, the BlackBerry® device automatically centers the dialog box.
- Create dialog boxes that are 90% or less of the width and height of the screen. If you use standard components, the BlackBerry device automatically calculates the appropriate size for dialog boxes.
- Center the dialog box indicator vertically with the dialog box message.
- Display messages to the right of the indicator and above any buttons.
- Place buttons for confirmation actions first. For example, place "Save" before "Discard" or "Cancel."
- Center buttons horizontally in dialog boxes.
- Place buttons vertically in the dialog box. The vertical layout allows buttons to expand to accommodate localized button labels.

Guidelines for messages

- Be specific. If possible, use one short sentence to clearly state the reason for displaying the dialog box and the actions that can dismiss it.
- Use complete sentences for messages where possible.
- Use vocabulary that users understand. For example, use "The file could not be saved because the media card is full" instead of "Error writing file to disk."
- Use positive language where possible and avoid blaming the user. Never write messages that blame users for errors or unexpected conditions. Instead, focus on the actions that users can take to resolve the issue.
- Use the second person (you, your) to refer to users.
- Use sentence case capitalization.
- Avoid using exclamation points (!) in messages.
- Avoid using an ellipsis (...) in messages unless you are indicating progress (for example, "Please wait...").

Guidelines for buttons

- For the default button, use the button that users are most likely to click. Avoid using a button that is associated with a destructive action as the default button. An exception to this rule are those cases where users initiate a minor destructive action (such as deleting a single item) and the most common user action is to continue with the action.
- Avoid using more than three buttons in a dialog box. If there are more than three, consider using an application screen instead with option buttons.
- On BlackBerry devices with a trackwheel, trackball, or trackpad, provide shortcut keys for buttons. Typically, the shortcut key is the first letter of the button label.
- Use clear, concise labels.
- Use one-word labels where possible. The size of a button changes depending on the length of the label. If a label is too long, an ellipsis (...) indicates that the text is truncated.
- Avoid using the labels "Yes" and "No." Use verbs that describe the associated action (for example, "Cancel," "Delete," "Discard," or "Save"). This approach helps users quickly and easily understand what happens when they click the button. If necessary, include more descriptive text elsewhere on the screen (for example, in an application message).
- Avoid using symbols or graphics in labels.
- Avoid using punctuation in labels. Use an ellipsis in a button label to indicate that additional information is required before the associated action can be performed.

Create a dialog box

Use alert dialog boxes to notify users of a critical action such as turning off the BlackBerry® device or an error such as typing information that is not valid. An exclamation point (!) indicator appears in an alert dialog box. To close an alert dialog box, users can click OK or press the Escape key. For more information on other types of dialog boxes, see the API reference for BlackBerry® Java® Development Environment.

1. Import the `net.rim.device.api.ui.component.Dialog` class.
2. Create an alert dialog box specifying the alert text that you want to display.

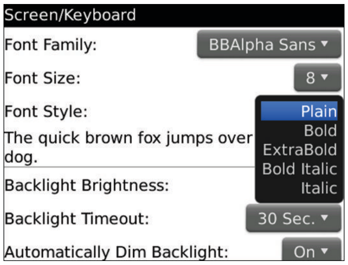
```
Dialog.alert("Specify the alert text that you want to display.")
```

Drop-down lists

Use drop-down lists to provide a set of mutually exclusive values.

Users can perform the following action with a drop-down list:

Action	BlackBerry devices without a touch screen	BlackBerry devices with a touch screen
Click a value from a drop-down list.	Press the Space key or click the trackwheel, trackball, or trackpad.	Click the screen.



Best practice: Implementing drop-down lists

- Use a drop-down list for two or more choices when space is an issue. If space is not an issue, consider using option buttons instead so that users can view the available options on the screen.
- Use the `ObjectChoiceField` class to create drop-down lists.
- For the default value, use the value that users are most likely to click.
- Use the highlighted option as the default focus when users scroll through the list.
- If users are not required to click a value, include a "None" value in the drop-down list. Always place the "None" value at the top of the list.
- Group and order values logically (for example, group related values together or include the most common values first). Avoid ordering values alphabetically; alphabetical order is language specific.

Guidelines for labels

- Use clear, concise labels for drop-down lists and for the values in drop-down lists. Verify that the label clearly describes what occurs when users click the value. The width of the drop-down list changes based on the length of the value labels. If a label is too long, an ellipsis (...) appears to indicate that the text is truncated.
- Place the label on the left side of a drop-down list.
- Use title case capitalization for drop-down list labels and for the value labels in a drop-down list.
- Punctuate labels for drop-down lists with a colon (:).

Create a drop-down list

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `MyUiScreen` class, which is described in step 3, represents the custom screen.

```
public class MyUi extends UiApplication
{
    public static void main(String[] args)
    {
        MyUi theApp = new MyUi();
        theApp.enterEventDispatcher();
    }
    public MyUi()
    {
        pushScreen(new MyUiScreen());
    }
}
```

3. Create the custom screen for the application by extending the `MainScreen` class. In the screen constructor, invoke `setTitle()` to specify the title for the screen.

```
class MyUiScreen extends MainScreen
{
    public MyUiScreen()
    {
        setTitle("UI Component Sample");
    }
}
```

4. In the screen constructor, create a drop-down list that displays a list of words or phrases by using the `ObjectChoiceField` class. Create a `String` array to store the items that you want to display in the drop-down list. Create an `int` to store the default item to display in the drop-down list. In the `ObjectChoiceField` constructor, specify the label for the drop-down list, the array of items to display, and the default item. In the following code sample, by default Wednesday is displayed. Invoke `add()` to add the drop-down list to the screen.

```
String choices[] =
{"Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"};
int iSetTo = 2;
add(new ObjectChoiceField("First Drop-down List",choices,iSetTo));
```

5. In the screen constructor, create a second drop-down list that displays a list of numbers by using the `NumericChoiceField` class. In the `NumericChoiceField` constructor, specify the label for the drop-down list, the first and last number to display in the drop-down list, the increment to use for the list of numbers, and the default number. In the following code sample, the numeric parameters are stored in `int` objects. The numbers 1 to 31 are included in the drop-down list and by default the number 10 is displayed. Invoke `add()` to add the second drop-down list to the screen.

```
int iStartAt   = 1;
int iEndAt     = 31;
int iIncrement = 1;
iSetTo         = 10;
add(new NumericChoiceField("Numeric Drop-Down
     List",iStartAt,iEndAt,iIncrement,iSetTo));
```

6. To override the default functionality that prompts the user to save changes before the application closes, in the extension of the `MainScreen` class, override the `MainScreen.onSavePrompt()` method. In the following code sample, the return value is `true` which indicates that the application does not prompt the user before closing.

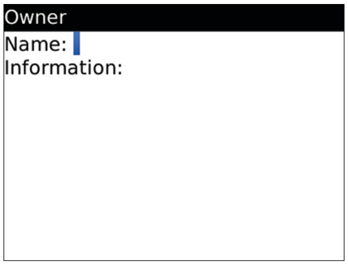
   ```
   public boolean onSavePrompt()
   {
       return true;
   }
   ```

Code sample: Creating a drop-down list

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;

public class MyUi extends UiApplication
{
    public static void main(String[] args)
    {
        MyUi theApp = new MyUi();
        theApp.enterEventDispatcher();
    }
    public MyUi()
    {
        pushScreen(new MyUiScreen());
    }
}
class MyUiScreen extends MainScreen
{
    public MyUiScreen()
    {
        setTitle("UI Component Sample");
        String choices[] =
{"Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"};
        int iSetTo = 2;
        add(new ObjectChoiceField("First Drop-down List",choices,iSetTo));
        int iStartAt   = 1;
        int iEndAt     = 31;
        int iIncrement = 1;
        iSetTo         = 10;
        add(new NumericChoiceField("Numeric Drop-Down
List",iStartAt,iEndAt,iIncrement,iSetTo));
    }
    public boolean onSavePrompt()
    {
        return true;
    }
}
```

Labels

Use a label to display text that identifies a control.


Owner
Name:
Information:

Best practice: Implementing labels

- Use the `LabelField` class to create labels.
- Use clear, concise labels.
- Group and order labels logically (for example, group related items together or include the most common items first). Avoid ordering values alphabetically; alphabetical order is language specific.
- Punctuate the label with a colon (:).

Create a text label

1. Import the `net.rim.device.api.ui.component.LabelField` class.
2. Create an instance of a `LabelField` to add a text label to a screen.

```
LabelField title = new LabelField("UI Component Sample", LabelField.ELLIPSIS);
```

Lists

Use a list to display items that users can highlight and open.

Users can perform the following actions in a list:

Action	BlackBerry devices without a touch screen	BlackBerry devices with a touch screen
Highlight an item in the list.	• Roll the trackwheel or trackball. • Slide a finger on the trackpad.	Touch the item lightly.
Open an item in the list.	Click the item or press the Enter key.	Click the screen.

Best practice: Implementing lists

- Use the `ListField` class to create lists.

Create a list box

Use a list box to display a list from which users can select one or more values.

1. Import the following classes:

```
import net.rim.device.api.system.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import java.util.Vector;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()`, to enable the application to receive events. In the constructor, invoke `pushScreen`, to display the custom screen for the application. The `CreateMenuScreen` class represents the custom screen which is described in step 3.

```
public class ListFields extends UiApplication
{
     public static void main(String[] args)
     {
         ListFields theApp = new ListFields();
         theApp.enterEventDispatcher();
     }
     public ListFields()
     {
         pushScreen(new ListFieldScreen());
     }
}
```

3. Create the custom screen for the application by extending the `MainScreen` class. In the constructor, invoke `setTitle()`, to display the title for the screen. Invoke `add()`, to display a text field on the screen. Invoke `addMenuItem()`, to add a menu item to the menu that `MainScreen` creates.

```
class ListFieldScreen extends MainScreen
{
    private ListField _listField;
    private Vector _listElements;

    public ListFieldScreen()
    {
        setTitle("List Field Sample");
    }
}
```

4. In the screen constructor, create the list box. Create an array for the items that you want to add to the list box by using the `Vector` class. Create the list box by using the `ListField()` class. Invoke `add()`, to add the list box to the screen. Invoke `initializeList()`, which is described in step 4, to add build the list box.

```
_listElements = new Vector();
_listField = new ListField();
ListCallback _callback = new ListCallback();
_listField.setCallback(_callback);
add(_listField);
initializeList();
```

5. Create a method to specify the items that you want to appear in the list box by using the `String` object. Invoke `addElement()` to add the items to the list. Invoke `setSize()`, to specify the number of items in the list box.

```
private void initializeList()
{
    String itemOne = "List item one";
    String itemTwo = "List item two";
    _listElements.addElement(itemOne);
    _listElements.addElement(itemTwo);
    reloadList();
}
private void reloadList()
{
    _listField.setSize(_listElements.size());
}
```

6. Create a class that implements the `ListFieldCallback` interface. Implement `drawListRow()`, to add the list box items to the screen. Implement `get()`, to return the list box item at the specified index. Implement `indexOfList()`, to return the first occurrence of a given string. Implement `getPreferredWidth()`, to retrieve the width of the list box.

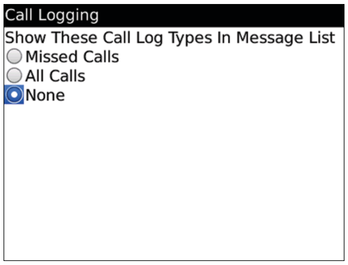
```
private class ListCallback implements ListFieldCallback
{
    public void drawListRow(ListField list, Graphics g, int index, int y, int w)
    {
        String text = (String)_listElements.elementAt(index);
        g.drawText(text, 0, y, 0, w);
    }
    public Object get(ListField list, int index)
    {
        return _listElements.elementAt(index);
    }
    public int indexOfList(ListField list, String prefix, int string)
    {
        return _listElements.indexOf(prefix, string);
    }
    public int getPreferredWidth(ListField list)
    {
        return Display.getWidth();
    }
}
```

Option buttons

Use option buttons to indicate a set of mutually exclusive but related choices.

Users can perform the following action with option buttons:

Action	BlackBerry devices without a touch screen	BlackBerry devices with a touch screen
Select an option button.	Press the Space key or click the trackwheel, trackball, or trackpad.	Click the screen.


Call Logging
Show These Call Log Types In Message List
Missed Calls
All Calls
[✓ None]

Best practice: Implementing option buttons

- Use option buttons for two or more choices when space is not an issue. If space is an issue, consider using a drop-down list instead.
- Use the `RadioButtonField` class to create option buttons.
- Verify that the content for option buttons remains static. Content for option buttons should not change depending on the context.
- Do not start an action when users select an option button. For example, do not open a new screen.
- Align option buttons vertically.
- Group and order values logically (for example, group related option buttons together or include the most common values first). Avoid ordering option buttons alphabetically; alphabetical order is language specific.

Guidelines for labels

- Use clear, concise labels. Verify that the label clearly describes what occurs when users select the option button. If the labels are too long, they wrap.
- Place labels on the right side of option buttons.
- Use title case capitalization.
- Do not use end punctuation.

Create a radio button

You can create a group of radio buttons by using the `RadioButtonGroup` class. A user can select only one option from a group of radio buttons.

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `MyUiScreen` class, which is described in step 3, represents the custom screen.

```
public class MyUi extends UiApplication
{
    public static void main(String[] args)
    {
        MyUi theApp = new MyUi();
        theApp.enterEventDispatcher();
    }
    public MyUi()
    {
        pushScreen(new MyUiScreen());
    }
}
```

3. Create the custom screen for the application by extending the `MainScreen` class. In the screen constructor, invoke `setTitle()` to specify the title for the screen.

```
class MyUiScreen extends MainScreen
{
    public MyUiScreen()
    {
        setTitle("UI Component Sample");
    }
}
```

4. In the screen constructor, create a group of radio buttons by using the `RadioButtonGroup` class. Create the radio buttons that you want to add to the group by using the `RadioButtonField` class. In the `RadioButtonField` constructor, specify the label for the radio button, the group, and a Boolean value to indicate the default selection (for example,`true` indicates that by default the radio is selected). Invoke `add()` to add the radio buttons to the screen.

```
RadioButtonGroup rbg = new RadioButtonGroup();
add(new RadioButtonField("Option 1",rbg,true));
add(new RadioButtonField("Option 2",rbg,false));
```

5. To override the default functionality that prompts the user to save changes before the application closes, in the extension of the `MainScreen` class, override the `MainScreen.onSavePrompt()` method. In the following code sample, the return value is `true` which indicates that the application does not prompt the user before closing.

   ```
   public boolean onSavePrompt()
   {
       return true;
   }
   ```

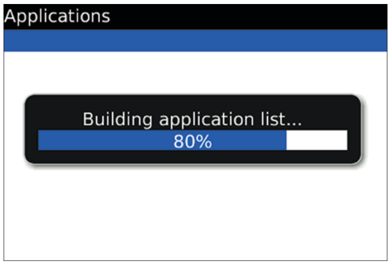
Code sample: Creating a radio button

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;

public class MyUi extends UiApplication
{
    public static void main(String[] args)
    {
        MyUi theApp = new MyUi();
        theApp.enterEventDispatcher();
    }
    public MyUi()
    {
        pushScreen(new MyUiScreen());
    }
}
class MyUiScreen extends MainScreen
{
    public MyUiScreen()
    {
        setTitle("UI Component Sample");
        RadioButtonGroup rbg = new RadioButtonGroup();
        add(new RadioButtonField("Option 1",rbg,true));
        add(new RadioButtonField("Option 2",rbg,false));
    }
    public boolean onSavePrompt()
    {
        return true;
    }
}
```

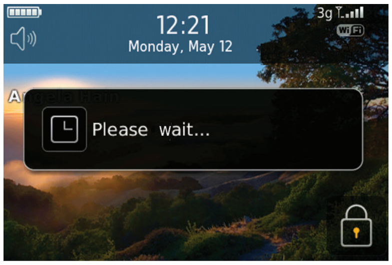
Progress indicators

Use progress indicators to indicate the status of an operation. There are two types of progress indicators:

- Use definite progress indicators when you can determine the duration of an operation. Definite progress indicators include a label to indicate what the operation is and a horizontal bar that fills from left to right as an operation progresses. A percentage appears in the bar to indicate how much of the operation is complete. In the browser, progress indicators also indicate the number of kilobytes out of a total number of kilobytes that a BlackBerry® device has downloaded. To hide a definite progress indicator, users press the End key. If users press the End key, the operation continues, but users can perform other tasks at the same time.
- Use an indefinite progress indicator when you cannot determine the duration of an operation. You can indicate progress using a status dialog box or by using an animated clock cursor for the Precision theme or an hourglass cursor for the Dimension theme. Users cannot perform any other actions while an application displays an indefinite progress indicator.



Definite progress indicator



Indefinite progress indicator

Best practice: Implementing progress indicators

- Always indicate progress when an operation takes more than 2 seconds to complete.
- Use a definite progress indicator when you can determine the duration of an operation.
- Use an indefinite progress indicator when you cannot determine the duration of an operation.
- Use the `GaugeField` class to create progress indicators.
- Always allow users to use the End key to hide a definite progress indicator.
- Provide useful progress information. For example, if users are loading a web page, indicate the amount of data that their BlackBerry® device has loaded (for example, 8 KB of 10 KB). If they are downloading applications to their device, indicate the progress as a percentage. Be as accurate as possible with the progress information.
- To indicate indefinite progress, use a clock icon for the Precision theme or an hourglass icon for the Dimension theme.
- Use a status dialog box to indicate indefinite progress when you can provide a meaningful message about the operation that is in progress. To indicate garbage collection, use the animated clock cursor for the Precision theme or an animated hourglass cursor for the Dimension theme.

Guidelines for labels

- Use a concise, descriptive label (for example, "Loading data" or "Building an application list").

- Use sentence case capitalization.
- Punctuate the dialog box message with an ellipsis (...).

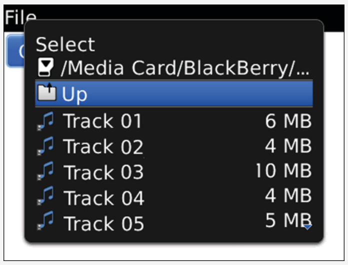
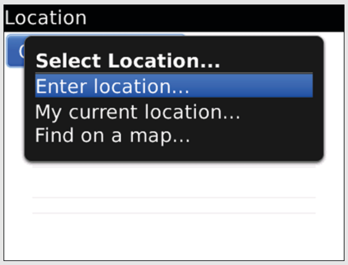
Create a progress indicator

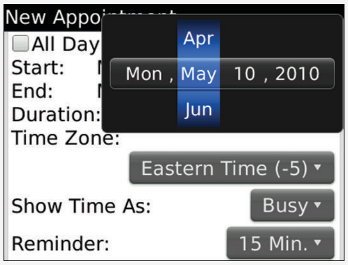
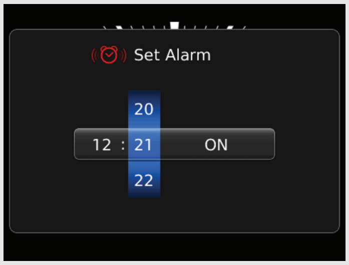
1. Import the `net.rim.device.api.ui.component.GaugeField` class.
2. Create an instance of a `GaugeField`.

```
GaugeField percentGauge = new GaugeField("Percent: ", 1, 100, 29,
GaugeField.PERCENT);
```

Picker dialog boxes

You can use picker dialog boxes to help users choose an item from a list easily.

Type of picker	Description
File	This picker allows users to browse for files on their BlackBerry® devices.  File Select /Media Card/BlackBerry/... Up Track 01 6 MB Track 02 4 MB Track 03 10 MB Track 04 4 MB Track 05 5 MB
Location	This picker allows users to choose a location from a list that you define. For example, you can allow users to choose their GPS location or a previously selected location.  Location Select Location... Enter location... My current location... Find on a map...

Type of picker	Description
Date	This picker allows users to choose a specific day, month, or year. For example, you can allow users to choose a month and year to specify when their credit card expires.  New Appo All Day Start: End: Duration: Time Zone: Eastern Time (-5) Show Time As: Busy Reminder: 15 Min. Apr Mon , May 10 , 2010 Jun
Time	This picker allows users to choose a specific hour, minute, or second.  Set Alarm 20 12 : 21 ON 22

Best practice: Implementing picker dialog boxes

Use the `FilePicker`, `LocationPicker`, and `DateTimePicker` classes to create picker dialog boxes.

Guidelines for file picker dialog boxes

- Allow users to start browsing from an appropriate default folder. If the application does not have a default folder and a media card is inserted in the BlackBerry device, allow users to start browsing from the root folder.
- If the application supports different file types and a media card is inserted in the BlackBerry device, allow users to start browsing from /Media Card/BlackBerry. If the application supports different file types and a media card is not inserted in the BlackBerry device, allow users to start browsing from /Device Memory/home/user.

Create a date picker

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.ui.picker.*;
import java.util.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the constructor, invoke `pushScreen()` to display the custom screen for the application. The `DatePickScreen` class represents the custom screen that is described in step 3.

```
public class DatePick extends UiApplication
{
    public static void main(String[] args)
    {
        DatePick theApp = new DatePick();
        theApp.enterEventDispatcher();
    }

    public DatePick()
    {
        pushScreen(new DatePickScreen());
    }
}
```

3. Create the custom screen for the application by extending the `MainScreen` class. In the constructor, invoke `setTitle()` to display a title on the screen. Invoke `add()` to display a rich text field on the screen.

```
class DatePickScreen extends MainScreen
{
    public DatePickScreen()
    {
        setTitle("Date Picker Sample");
        add(new RichTextField("Trying Date Picker"));
    }
}
```

4. Add a section of code to the event queue of the application by invoking `invokeLater()`. Create a `Runnable` object and pass it as a parameter to `invokeLater()`. Override `run()` in the definition of `Runnable`.

```
class DatePickScreen extends MainScreen
{
    public DatePickScreen()
    {
        setTitle("Date Picker Sample");
        add(new RichTextField("Trying Date Picker"));

        UiApplication.getUiApplication().invokeLater(new Runnable()
        {
            public void run()
            {
                }
            });
        }
}
```

5. In `run()`, invoke `DateTimePicker.getInstance()` to return a `DateTimePicker` object. Invoke `doModal()` to display the date picker. Invoke `getDateTime()` to return a `Calendar` object that represents the date and time the user selects. Use `getTime()` to return the date and time as a `Date` object. Use `Dialog.alert()` to display the selected date and time.

```
class DatePickScreen extends MainScreen
{
    public DatePickScreen()
    {
        setTitle("Date Picker Sample");
        add(new RichTextField("Trying Date Picker"));

        UiApplication.getUiApplication().invokeLater(new Runnable()
        {
            public void run()
            {
                DateTimePicker datePicker = DateTimePicker.getInstance();
                datePicker.doModal();
                Calendar cal = datePicker.getDateTime();
                Date date = cal.getTime();
                Dialog.alert("You selected " + date.toString());
            }
        });
    }
}
```

Code sample: Creating a date picker

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.picker.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.database.*;
import net.rim.device.api.io.*;
import java.util.*;

public class DatePick extends UiApplication
{
    public static void main(String[] args)
    {
        DatePick theApp = new DatePick();
        theApp.enterEventDispatcher();
    }

    public DatePick()
    {
        pushScreen(new DatePickScreen());
    }
}

class DatePickScreen extends MainScreen
{
    public DatePickScreen()
    {
        setTitle("Date Picker Sample");
        add(new RichTextField("Trying Date Picker"));

        UiApplication.getUiApplication().invokeLater(new Runnable()
        {
            public void run()
            {
                DateTimePicker datePicker = DateTimePicker.getInstance();
                datePicker.doModal();
                Calendar cal = datePicker.getDateTime();
                Date date = cal.getTime();
                Dialog.alert("You selected " + date.toString());
            }
        });
    }
}
```

Create a file picker

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.picker.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import java.util.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `FilePickScreen` class represents the custom screen that is described in step 3.

```
public class FilePick extends UiApplication
{
    public static void main(String[] args)
    {
        FilePick theApp = new FilePick();
        theApp.enterEventDispatcher();
    }
    public FilePick()
    {
        pushScreen(new FilePickScreen());
    }
}
```

3. Create the custom screen for the application by extending the `MainScreen` class. In the screen constructor, invoke `setTitle()` to specify a title for the screen. Invoke `add()` to add a label field to the screen.

```
class FilePickScreen extends MainScreen
{
    public FilePickScreen()
    {
        setTitle("File Picker Sample");
        add(new LabelField("Trying File Picker"));
    }
}
```

4. In the screen constructor, invoke `invokeLater()` to add a section of code to the event queue of the application. Create a `Runnable` object and pass it as a parameter to `invokeLater()`.

```
class FilePickScreen extends MainScreen
{
    public FilePickScreen()
    {
        setTitle("File Picker Sample");
        add(new LabelField("Trying File Picker"));

        UiApplication.getUiApplication().invokeLater(new Runnable()
        {
            public void run()
            {
            }
        });
    }
}
```

5. Override `run()` in the definition of `Runnable`. In `run()`, invoke `FilePicker.getInstance()` to return a `FilePicker` object. Create a `FilePickListener` object, which is described in step 6. Invoke `setListener()` to register the listener for the file picker. Invoke `show()` to display the file picker on the screen.

```

class FilePickScreen extends MainScreen
{
    public FilePickScreen()
    {
        setTitle("File Picker Sample");
        add(new LabelField("Trying File Picker"));

        UiApplication.getUiApplication().invokeLater(new Runnable()
        {
            public void run()
            {
                FilePicker fp = FilePicker.getInstance();
                FilePickListener fileListener = new FilePickListener();
                fp.setListener(fileListener);
                fp.show();
            }
        });
    }
}
```

6. Invoke `Dialog.alert` to create a dialog box with a message that displays which file was selected. Invoke `Dialog.alert` in a class that implements the `FilePicker.Listener` interface by overriding `selectionDone()`. The application calls `selectionDone()` when the user selects a file using the file picker.

```
class FilePickListener implements FilePicker.Listener
{
    public void selectionDone(String str)
    {
        Dialog.alert("You selected " + str);
    }
}
```

Code sample: Creating a file picker

```

import net.rim.device.api.ui.*;
import net.rim.device.api.ui.picker.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.io.*;

public class FilePick extends UiApplication
{
    public static void main(String[] args)
    {
        FilePick theApp = new FilePick();
        theApp.enterEventDispatcher();
    }
    public FilePick()
    {
        pushScreen(new FilePickScreen());
    }
}

class FilePickScreen extends MainScreen
{
    public FilePickScreen()
    {
        setTitle("File Picker Sample");
        add(new LabelField("Trying File Picker"));

        UiApplication.getUiApplication().invokeLater(new Runnable()
        {
            public void run()
            {
                FilePicker fp = FilePicker.getInstance();
                FilePickListener fileListener = new FilePickListener();
                fp.setListener(fileListener);
                fp.show();
            }
        });
    }
}
class FilePickListener implements FilePicker.Listener
{
    public void selectionDone(String str)
    {
        Dialog.alert("You selected " + str);
    }
}
```

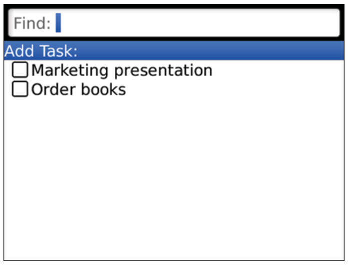
Search fields

Use search fields to allow users to search for items in an application. Several BlackBerry® device applications include search fields. For example, in the task list, users can use a search field to search for specific tasks.

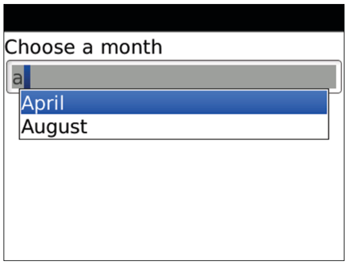
As users type text in a search field, the application searches for and displays items that begin with the search text. If users type multiple words in the search field, the application displays results that match both words. For example, if users type "ca ba", the application returns all items with a word that begins with "ca" and a word that begins with "ba" (such as "call back").

Users can perform the following actions in a search field:

Action	BlackBerry devices without a touch screen	BlackBerry devices with a touch screen
Open a highlighted item in the search results.	Press the Enter key or click the trackwheel, trackball, or trackpad.	Click the screen.
Delete the text in a search field.	Press the Escape key.	Press the Escape key.



To help users type a specific value quickly, you can create a field that supports autocompletion. As users type, a list of suggested values appears in a drop-down list. Users can click a value or continue typing to restrict the values in the list.



Best practice: Implementing search fields

- Provide a search field when you have a constrained list of items.
- Use the `KeywordFilterField` class to create search fields.
- Use the `AutoCompleteField` class and the `BasicFilteredList` class to create autocomplete fields.
- Allow users to use the Escape key to delete search text that they type.
- Set the default focus to the search field and the first item in the list.
- Display items in the search results that contain a word that begins with the typed search text.
- Do not assign shortcut keys for a screen that includes a search field. If you want to use shortcut keys and you have a constrained list of items, provide alternative search functionality. For example, users can search for messages in a message list by pressing S or using the menu.
- Place a search field in the title bar of an application screen.
- Do not make search fields case sensitive.

Guidelines for labels

- Use the label "Find:" for the search field.
- Do not include prompt text in the search field.

Create a search field

You can create an application that uses the `KeywordFilterField` class, included in the `net.rim.device.api.ui.component` package, to provide a UI field that consists of a single text input field and a list of selectable elements. As users type text in a search field, the application filters the elements in the list that begin with the search text. For more information about using the `KeywordFilterField` class, see the Keyword Filter Field sample application, included with the BlackBerry® Java® Development Environment version 4.3.1 or later.

1. Import the required classes and interfaces.

   ```
   import net.rim.device.api.collection.util.SortedReadableList;
   import net.rim.device.api.io.LineReader;
   import net.rim.device.api.ui.component.KeywordFilterField;
   import net.rim.device.api.ui.component.KeywordProvider;
   import java.io.InputStream;
   import java.lang.String;
   import java.util.Vector;
   ```

2. Create variables. In the following code sample, `CountryList` extends the `SortedReadableList` class and implements the `KeywordProvider` interface.

   ```
   private KeywordFilterField _keywordField;
   private CountryList _CountryList;
   private Vector _countries;
   ```

3. To create a list of selectable text items, populate a vector with data from a text file.

   ```
   _countries = getDataFromFile();
   ```

4. Create an instance of a class that extends the `SortedReadableList` class.

   ```
   _CountryList = new
        CountryList(StringComparator.getInstance(true),_countries);
   ```

5. To specify the elements of the list, create a new instance of a `KeywordFilterField` object.

   ```
   _keywordField = new KeywordFilterField();
   ```

6. Invoke `KeywordFilterField.setList()`.

   ```
   _keywordField.setList(_CountryList, _CountryList);
   ```

7. Set a label for the input field of the `KeywordFilterFIeld`.

   ```
   _keywordField.setLabel("Search: ");
   ```

8. Create the main screen of the application and add a `KeywordFilterField` to the main screen.

   ```
   KeywordFilterDemoScreen screen = new
        KeywordFilterDemoScreen(this,_keywordField);
   screen.add(_keywordField.getKeywordField());
   screen.add(_keywordField);
   pushScreen(screen);
   ```

9. To create a method that populates and returns a vector of `Country` objects containing data from text file, in the method signature, specify `Vector` as the return type.

   ```
   public Vector getDataFromFile()
   ```

10. Create and store a reference to a new `Vector` object.

    ```
    Vector countries = new Vector();
    ```

11. Create an input stream to the text file.

    ```
    InputStream stream =
         getClass().getResourceAsStream("/Data/CountryData.txt");
    ```

12. Read CRLF delimited lines from the input stream.

    ```
    LineReader lineReader = new LineReader(stream);
    ```

13. Read data from the input stream one line at a time until you reach the end of file flag. Each line is parsed to extract data that is used to construct `Country` objects.

    ```
    for(;;){
        //Obtain a line of text from the text file
        String line = new String(lineReader.readLine());

        //If we are not at the end of the file, parse the line of text
        if(!line.equals("EOF")) {
            int space1 = line.indexOf(" ");
            String country = line.substring(0,space1);
            int space2 = line.indexOf(" ",space1+1);
            String population = line.substring(space1+1,space2);
            String capital = line.substring(space2+1,line.length());

            // Create a new Country object
           countries.addElement(new Country(country,population,capital));

        }
        else {
            break;
        }
    } // end the for loop
    return countries;
    ```

14. To add a keyword to the list of selectable text items, invoke `SortedReadableList.doAdd(element)`.

    ```
    SortedReadableList.doAdd(((Country)countries.elementAt(i))
        .getCountryName()) ;
    ```

15. To update the list of selectable text items, invoke `KeywordFilterField.updateList()`.

    ```
    _keywordField.updateList();
    ```

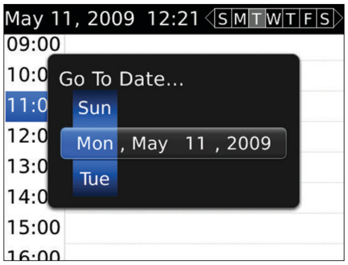
16. To obtain the key word that a BlackBerry device user typed into the `KeywordFilterField`, invoke `KeywordFilterField.getKeyword()`.

```
String userTypedWord = _keywordField.getKeyword();
```

Spin boxes

Use a spin box to allow users to choose an item from an ordered list easily. For example, use spin boxes to allow users to find a number or to change the day of the week.

Action	BlackBerry devices without a touch screen	BlackBerry devices with a touch screen
Find an item in the list.	• Roll the trackball up or down. • Slide a finger up or down on the trackpad.	Slide a finger up or down quickly.
Choose an item from the list.	Click the trackball or trackpad.	Click OK.
Move to another spin box.	• Roll the trackball to the left or right. • Slide a finger to the left or right on the trackpad.	Touch the spin box.



Best practice: Implementing spin boxes

- Use spin boxes for a list of sequential items.
- Use drop-down lists for non-sequential items or items with irregular intervals. For a short list of non-sequential items, you can use a spin box to provide a more interactive experience for users.
- Avoid using a spin box if several other components appear on the screen.
- Use the `SpinBoxField` class and the `SpinBoxFieldManager` class to create spin boxes.
- Add spin boxes to dialog boxes instead of screens where possible.
- When users highlight a spin box, display three to five items vertically.
- Use an identifiable pattern for the sequence of items (for example, 5, 10, 15) so that users can estimate how much they need to scroll to find the target item.
- Avoid making users scroll horizontally to view multiple spin boxes. If necessary, separate spin boxes into multiple fields.

- If the text in a spin box is too long, use an ellipsis (...).

Create a spin box

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.UiApplication;
import net.rim.device.api.ui.component.Dialog;
import net.rim.device.api.ui.component.TextSpinBoxField;
import net.rim.device.api.ui.container.MainScreen;
import net.rim.device.api.ui.container.SpinBoxFieldManager;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `HomeScreen` class represents the custom screen that is described in step 3.

```
public class SpinBoxApp extends UiApplication
{
    public static void main(String[] args)
    {
        SpinBoxApp app = new SpinBoxApp();
        app.enterEventDispatcher();
    }

    public SpinBoxApp()
    {
        pushScreen(new HomeScreen());
    }
}
```

3. Create the custom screen for the application by extending the `MainScreen` class. Declare a variable for each field in the spin box and declare a variable for the spin box field manager.

```
class HomeScreen extends MainScreen
{
    TextSpinBoxField spinBoxDays;
       TextSpinBoxField spinBoxMonths;
       SpinBoxFieldManager spinBoxMgr;

    public HomeScreen()
    {
    }
}
```

4. In the screen constructor, create an array of String objects for each of the spin box fields.

```
final String[] DAYS   =
     {"Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"};
```

```
final String[] MONTHS =
{"January","February","March","April","May","June","July","A
    ugust","September","October","November","December"};
```

5. In the screen constructor, create new instances of the spin box field manager and the two spin box fields. Pass the appropriate array of strings as arguments to each of the spin box field constructors.

```
spinBoxMgr = new SpinBoxFieldManager();

spinBoxDays   = new TextSpinBoxField(DAYS);
spinBoxMonths = new TextSpinBoxField(MONTHS);
```

6. In the screen constructor, add the spin box fields to the spin box field manager. Invoke `add()` to add the manager, and the fields that it contains, to the screen.

```
spinBoxMgr.add(spinBoxDays);
spinBoxMgr.add(spinBoxMonths);
add(spinBoxMgr);
```

Code sample: Creating a spin box

```
import net.rim.device.api.ui.UiApplication;
import net.rim.device.api.ui.container.MainScreen;
import net.rim.device.api.ui.container.SpinBoxFieldManager;
import net.rim.device.api.ui.component.Dialog;
import net.rim.device.api.ui.component.TextSpinBoxField;


public class SpinBoxApp extends UiApplication
{
    public static void main(String[] args)
       {
              SpinBoxApp app = new SpinBoxApp();
              app.enterEventDispatcher();
    }

       public SpinBoxApp()
       {
              HomeScreen homeScreen = new HomeScreen();
              pushScreen(homeScreen);
       }
}

class HomeScreen extends MainScreen
{
    TextSpinBoxField spinBoxDays;
       TextSpinBoxField spinBoxMonths;
       SpinBoxFieldManager spinBoxMgr;

       public HomeScreen()
       {
           final String[] DAYS   =
         {"Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday"};
              final String[] MONTHS =
             {"January","February","March","April","May","June","July","August","Sep
            tember","October","November","December"};

           spinBoxMgr = new SpinBoxFieldManager();

           spinBoxDays   = new TextSpinBoxField(DAYS);
              spinBoxMonths = new TextSpinBoxField(MONTHS);

              spinBoxMgr.add(spinBoxDays);
              spinBoxMgr.add(spinBoxMonths);
              add(spinBoxMgr);
       }

       public void close()
       {
              Dialog.alert("You selected " +
             (String)spinBoxDays.get(spinBoxDays.getSelectedIndex()) + " and " +
             (String)spinBoxMonths.get(spinBoxMonths.getSelectedIndex()));
              super.close();
       }
}
```

Text fields

Users can use a text field to type text.

Type of text field	Description
email	Users can insert an at sign (@) or a period (.) in an email address field by pressing the Space key.
date and time	On BlackBerry® devices with a trackwheel or trackball, users can change the date or time using the keyboard or by rolling the trackwheel or trackball vertically. On BlackBerry devices with a trackpad, users can change the date or time using the keyboard or by sliding a finger vertically on the trackpad. On BlackBerry devices with a touch screen, users can change the date or time by clicking the Date or Time field and sliding a finger up or down.
number	On BlackBerry devices without a touch screen, when users type in a number field, the BlackBerry device switches to number lock mode so that users do not need to press the Alt key to type numbers.

Type of text field	Description
	On BlackBerry devices with a touch screen, when users need to type in a number field, the number keyboard appears.
password	When users type in a password field, asterisks (*) appear instead of text. Users cannot cut, copy, or paste text or use AutoText in password fields. On BlackBerry devices with SureType® technology, multi-tap is the default typing input method in password fields.
phone number	On BlackBerry devices without a touch screen, when users type in a phone number field, the BlackBerry device switches to number lock mode so that users do not need to press the Alt key to type numbers. You can also allow users to perform the following actions in phone number fields: • Type the plus sign (+) for international phone numbers. • Type formatting characters such as the minus sign (-), period (.), parentheses (()), and spaces. • Type a number sign (#) or an asterisk (*). • Indicate a pause by typing a comma (,) or indicate a wait by typing an exclamation point (!). • Indicate an extension by pressing the Alt key and pressing E, X, or T. On BlackBerry devices with a touch screen, when users need to type in a phone number field, the number keyboard appears. You can also allow users to perform the following actions in phone number fields: • Type a number sign (#) or an asterisk (*). • Indicate a pause or an extension by holding the asterisk (*) key. • Indicate a wait or an extension by holding the number sign (#) key.
text	In text fields, users type text. Users can cut, copy, and paste text in text fields. When the cursor reaches the end of a line of text, the text wraps to the next line. In text fields, BlackBerry devices can also turn telephone numbers, web pages, and email addresses into links automatically.
web address	Users can insert a period (.) in an address field by pressing the Space key.

Best practice: Implementing text fields

- Use the `RichTextField` class to create text fields.

- Choose a type of text field based on what you expect users to type. Using the most appropriate text field is important so that the appropriate typing indicator appears on the screen when users type text. For example, when users type text in a phone number field, the number lock indicator appears in the upper-right corner of the screen. The field that you choose also affects the default typing input method for the field on BlackBerry devices with SureType® technology.
- Use or extend existing fields instead of creating custom fields where possible so that fields inherit the appropriate default behavior.
- Use prompt text only when space on a screen is limited and you cannot include a label or instructional text. Prompt text appears in the field and disappears when users type.

Guidelines for labels

- Use concise, descriptive labels. Avoid labels that wrap.
- Use title case capitalization.
- Punctuate labels for fields with a colon (:).
- If you use prompt text, keep it concise. Use sentence case capitalization.

Creating a text field

Create a read-only text field that allows formatting

1. Import the `net.rim.device.api.ui.component.RichTextField` class.
2. Create an instance of a `RichTextField`.

   ```
   RichTextField rich = new RichTextField("RichTextField");
   ```

Create an editable text field that has no formatting and accepts filters

1. Import the following classes:
   - `net.rim.device.api.ui.component.BasicEditField`
   - `net.rim.device.api.ui.component.EditField`
2. Create an instance of a `BasicEditField`.

   ```
   BasicEditField bf = new BasicEditField("BasicEditField: ", "", 10,
   EditField.FILTER_UPPERCASE);
   ```

Create an editable text field that allows special characters

1. Import the `net.rim.device.api.ui.component.EditField` class.
2. Create an instance of an `EditField`.

   ```
   EditField edit = new EditField("EditField: ", "", 10, EditField.FILTER_DEFAULT);
   ```

Create a text field for AutoText

If a text field supports AutoText, when users press the Space key twice, the BlackBerry® device inserts a period, capitalizes the next letter after a period, and replaces words as defined in the AutoText application.

1. Import the following classes:
   - `net.rim.device.api.ui.component.AutoTextEditField`
   - `net.rim.device.api.ui.autotext.AutoText`
   - `net.rim.device.api.ui.component.BasicEditField`
2. Create an instance of an `AutoTextEditField`.

```
AutoTextEditField autoT = new AutoTextEditField("AutoTextEditField: ", "");
```

Create a date field

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import java.lang.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `MyUiScreen` class, which is described in step 3, represents the custom screen.

```
public class MyUi extends UiApplication
{
    public static void main(String[] args)
    {
        MyUi theApp = new MyUi();
        theApp.enterEventDispatcher();
    }
    public MyUi()
    {
        pushScreen(new MyUiScreen());
    }
}
```

3. Create the custom screen for the application by extending the `MainScreen` class. In the screen constructor, invoke `setTitle()` to specify the title for the screen.

```
class MyUiScreen extends MainScreen
{
    public MyUiScreen()
    {
        setTitle("UI Component Sample");
    }
}
```

4. In the screen constructor, create a date field by using the `DateField` class. Provide `System.currentTimeMillis()` as a parameter to return the current time. Use the `DateField.DATE_TIME` style to display both the date and the time. You can use other styles to display only the date or only the time.

```
add(new DateField("Date: ", System.currentTimeMillis(),
     DateField.DATE_TIME));
```

Create a password field

1. Import the `net.rim.device.api.ui.component.PasswordEditField` class.
2. Create an instance of a `PasswordEditField`.

   For example, the following instance uses a constructor that lets you provide a default initial value for the `PasswordEditField`:

```
PasswordEditField pwd = new PasswordEditField("PasswordEditField: ", "");
```

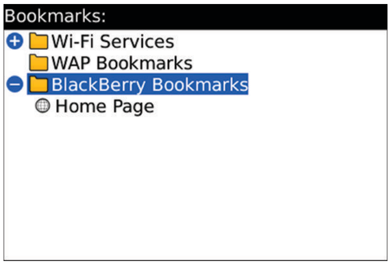
Tree views

Use a tree view to display objects, such as folders, in a hierarchical manner.

Objects in the tree view are nodes. The highest node is the root node. A node in the tree can have child nodes under it. A node that has a child is a parent node.

Users can perform the following action in a tree view:

Action	BlackBerry devices without a touch screen	BlackBerry devices with a touch screen
Expand or collapse an object with a plus sign (+) or minus sign (-) in a hierarchy .	Press the Space key or click the trackwheel, trackball, or trackpad.	Click the screen.



Best practice: Implementing tree views

- Use the `TreeField` class to create tree views.
- On BlackBerry devices with a trackball or trackpad, provide a context menu if users can perform multiple actions when they click a parent node. For example, when users click a folder that contains subfolders and files, a context menu should appear that allows users to either expand the folder or view the files.
- Include a root node only if users need the ability to perform actions on the entire tree. Otherwise, exclude the root node.

Create a field to display a tree view

Use a tree view to display objects, such as a folder structure, in a hierarchical manner. A `TreeField` contains nodes. The highest node is the root node. A node in the tree can have child nodes under it. A node that has a child is a parent node.

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.component.TreeField;
import net.rim.device.api.ui.component.TreeFieldCallback;
import net.rim.device.api.ui.container.MainScreen;
import java.lang.String;
```

2. Implement the `TreeFieldCallback` interface.

3. Invoke `TreeField.setExpanded()` on the `TreeField` object to specify whether a folder is collapsible. Create a `TreeField` object and multiple child nodes to the `TreeField` object. Invoke `TreeField.setExpanded()` using `node4` as a parameter to collapse the folder.

```
String fieldOne =  new String("Main folder");
...
TreeCallback myCallback = new TreeCallback();
TreeField myTree = new TreeField(myCallback, Field.FOCUSABLE);
int node1 = myTree.addChildNode(0, fieldOne);
int node2 = myTree.addChildNode(0, fieldTwo);
int node3 = myTree.addChildNode(node2, fieldThree);
int node4 = myTree.addChildNode(node3, fieldFour);
...
int node10 = myTree.addChildNode(node1, fieldTen);
```

```
myTree.setExpanded(node4, false);
...
mainScreen.add(myTree);
```

4. To repaint a `TreeField` when a node changes, create a class that implements the `TreeFieldCallback` interface and implement the `TreeFieldCallback.drawTreeItem` method. The `TreeFieldCallback.drawTreeItem` method uses the cookie for a tree node to draw a `String` in the location of a node. The `TreeFieldCallback.drawTreeItem` method invokes `Graphics.drawText()` to draw the `String`.

```
private class TreeCallback implements TreeFieldCallback
{
    public void drawTreeItem(TreeField _tree, Graphics g, int node, int y, int
width, int indent)
    {
        String text = (String)_tree.getCookie(node);
        g.drawText(text, indent, y);
    }
}
```

Menu items 8

Create a menu

The MainScreen class provides standard components of a BlackBerry® device application. It includes a default menu.

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
```

2. Create the application framework by extending the UiApplication class. In main(), create an instance of the new class and invoke enterEventDispatcher() to enable the application to receive events. In the application constructor, invoke pushScreen() to display the custom screen for the application. The CreateMenuScreen class, which is described in step 3, represents the custom screen.

```
public class CreateMenu extends UiApplication
{
   public static void main(String[] args)
   {
      CreateMenu theApp = new CreateMenu();
      theApp.enterEventDispatcher();
   }
   public CreateMenu()
   {
      pushScreen(new CreateMenuScreen());
   }
}
```

3. Create the custom screen for the application by extending the MainScreen class. In the screen constructor, invoke setTitle() to specify the title for the screen. Invoke add() to add a text field to the screen. Invoke addMenuItem() to add a menu item to the menu that MainScreen creates.

```
class CreateMenuScreen extends MainScreen
{
   public CreateMenuScreen()
   {
      setTitle("Create Menu Sample");
      add(new RichTextField("Create a menu"));
      addMenuItem(_viewItem);
   }
}
```

4. Create the menu item by using the MenuItem class. Override run() to specify the action that occurs when the user clicks the menu item. When the user clicks the menu item, the application invokes Menu.run().

```
private MenuItem _viewItem = new MenuItem("More Info", 110, 10)
{
   public void run()
   {
      Dialog.inform("Display more information");
   }
};
```

5. Override `close()` to display a dialog box when the user clicks the Close menu item. By default, the Close menu item is included in the menu that `MainScreen` creates. Invoke `super.close()` to close the application. When the user closes the dialog box, the application invokes `MainScreen.close()` to close the application.

```
public void close()
{
    Dialog.alert("Goodbye!");
    super.close();
}
```

Code sample: Creating a menu

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;

public class CreateMenu extends UiApplication
{
     public static void main(String[] args)
     {
         CreateMenu theApp = new CreateMenu();
         theApp.enterEventDispatcher();
     }
     public CreateMenu()
     {
         pushScreen(new CreateMenuScreen());
     }
}
class CreateMenuScreen extends MainScreen
{
    public CreateMenuScreen()
    {
      setTitle("Create Menu Sample");
      add(new RichTextField("Create a menu"));
      addMenuItem(_viewItem);
    }
    private MenuItem _viewItem = new MenuItem("More Info", 110, 10)
    {
        public void run()
        {
            Dialog.inform("Display more information");
        }
    };
    public void close()
    {
        Dialog.alert("Goodbye!");
        super.close();
    }
}
```

Best practice: Implementing menus

- Always provide a full menu.
- Make sure that users can press the Menu key to open the full menu and to initiate an action when a menu item is highlighted. Make sure that users can also hold the Menu key to open the dialog box for switching applications.
- For the default menu item, use the menu item that users are most likely to select.
- Place the default menu item and other common menu items in the middle of the menu.
- Verify that the order of menu items is consistent with the order of menu items in other BlackBerry® device applications.
- Group menu items according to common usage or common functionality, and where possible, test your groupings with users.
- Insert separators between menu item groupings.
- Do not place menu items for conflicting actions close together. For example, do not place a "Delete" menu item beside an "Open" menu item.
- Always include the "Switch application" and "Close" menu items. Place these menu items at the end of the menu. If you use standard components, these menu items are included automatically.

Guidelines for labels

- Use concise, descriptive labels that are no longer than 12 characters. If a label is too long, an ellipsis (...) appears to indicate that the text is truncated.
- Use verbs for labels.
- Use title case capitalization for labels.
- Use an ellipsis in a menu item label to indicate that users must perform another action after they click the menu item. For example, if users click the Go To Date... menu item in the calendar, they must specify a date on the screen that appears.
- Avoid using symbols such as an asterisk (*) in labels.

Adding a menu item to a BlackBerry Device Software application

You can add a menu item to a BlackBerry® Device Software application by using the Menu Item API in the `net.rim.blackberry.api.menuitem` package. For example, you can add a menu item called View Sales Order to the contacts application on a BlackBerry device so that when a user clicks the menu item, a CRM application opens and displays a list of sales orders for that contact.

The `ApplicationMenuItemRepository` class provides the constants that specify the BlackBerry Device Software application that your menu item should appear in. For example, the `MENUITEM_MESSAGE_LIST` constant specifies that the menu item should appear in the messages application.

Add a menu item to a BlackBerry Device Software application

1. Import the required classes and interfaces.

```
import net.rim.blackberry.api.menuitem.*;
import net.rim.blackberry.api.pdap.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
```

2. Extend the abstract `ApplicationMenuItem` class to create a menu item. Override the `ApplicationMenuItem()` constructor with an integer to specify the position of the menu item in the menu. A higher number positions the menu item lower in the menu.

```
public class SampleMenuItem extends ApplicationMenuItem
{
   SampleMenuItem()
   {
      super(20);
   }
}
```

3. Implement `toString()` to specify the menu item text.

```
public String toString()
{
   return "Open the Contacts Demo application";
}
```

4. Invoke `getInstance()` to retrieve the application repository.

```
ApplicationMenuItemRepository repository =
     ApplicationMenuItemRepository.getInstance();
```

5. Create an instance of a class to extend the `MenuItem` class.

```
ContactsDemoMenuItem contactsDemoMenuItem = new
     ContactsDemoMenuItem();
```

6. Invoke `ApplicationMenuItemRepository.addMenuItem()` to add the menu item to the relevant BlackBerry® device application repository.

```
repository.addMenuItem(ApplicationMenuItemRepository
    .MENUITEM_ADDRESSCARD_VIEW, contactsDemoMenuItem);
```

7. Implement `run()` to specify the behavior of the menu item. In the following code sample, when a user clicks the new menu item and a `Contact` object exists, the `ContactsDemo` application receives the event and invokes `ContactsDemo.enterEventDispatcher()`.

```
public Object run(Object context)
{
   BlackBerryContact c = (BlackBerryContact)context;
   if ( c != null )
   {
      new ContactsDemo().enterEventDispatcher();
   }
   else
   {
      throw new IllegalStateException( "Context is null, expected a Contact
     instance");
   }
   Dialog.alert("Viewing an email message in the email view");
   return null;
}
```

Changing the appearance of a menu

You can change the background, border, and font of a menu by using the `Menu` class in the `net.rim.device.api.ui.component` package. For example, you can change the appearance of the menu so that it has a look and feel that is similar to the rest of your BlackBerry® device application. When you change the appearance of a menu, your BlackBerry device application overrides the theme that is set on the BlackBerry device.

Change the appearance of a menu

1. Import the required classes and interfaces.

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.ui.decor.*;
import net.rim.device.api.system.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `CreateCustomMenuScreen` class , which is described in step 3, represents the custom screen.

```
public class CreateCustomMenu extends UiApplication
{
   public static void main(String[] args)
   {
      CreateCustomMenu theApp = new CreateCustomMenu();
      theApp.enterEventDispatcher();
   }
   public CreateCustomMenu()
   {
        pushScreen(new CreateCustomMenuScreen());
    }
}
```

3. Create the custom screen for the application by extending the `MainScreen` class. In the screen constructor, invoke `setTitle()` to specify the title for the screen. Invoke `add()` to add a text field on the screen.

```
class CreateCustomMenuScreen extends MainScreen
{
    Background _menuBackground;
    Border _menuBorder;
    Font _menuFont;
    CreateCustomMenuScreen()
    {
        setTitle("Custom Menu Sample");
        add(new RichTextField("Creating a custom menu"));
    }
}
```

4. In the screen constructor, specify the appearance of the menu. Create the spacing for the border around the menu by creating an `XYEdges` object. Invoke `createRoundedBorder()` to create a border with rounded corners. Invoke `createSolidTransparentBackground()` to create a transparent background color for the menu.

```
XYEdges thickPadding = new XYEdges(10, 10, 10, 10);
_menuBorder = BorderFactory.createRoundedBorder(thickPadding,
Border.STYLE_DOTTED);
_menuBackground = BackgroundFactory.createSolidTransparentBackground(Color
    .LIGHTSTEELBLUE, 50);
```

5. In the screen constructor, specify the font for the menu by using a `FontFamily` object. Invoke `forName()` to retrieve a font from on the BlackBerry® device. Invoke `getFont()` to specify the style and size of the font.

```
try
{
    FontFamily family = FontFamily.forName("BBCasual");
    _menuFont = family.getFont(Font.PLAIN, 30, Ui.UNITS_px);
}
catch(final ClassNotFoundException cnfe)
{
    UiApplication.getUiApplication().invokeLater(new Runnable()
    {
        public void run()
        {
            Dialog.alert("FontFamily.forName() threw " + cnfe.toString());
        }
    });
}
```

6. In the screen class, override `makeMenu()` to apply the appearance of the menu. Invoke `setBackground()`, `setBorder()`, and `setFont()` to apply the appearance the menu that you specified in steps 4 and 5.

```
protected void makeMenu(Menu menu, int context)
{
    menu.setBorder(_menuBorder);
    menu.setBackground(_menuBackground);
    menu.setFont(_menuFont);
    super.makeMenu(menu, context);
}
```

Code sample: Changing the appearance of a menu

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.ui.decor.*;
import net.rim.device.api.system.*;

public class CreateCustomMenu extends UiApplication
{
   public static void main(String[] args)
   {
      CreateCustomMenu theApp = new CreateCustomMenu();
      theApp.enterEventDispatcher();
   }
   public CreateCustomMenu()
   {
      pushScreen(new CreateCustomMenuScreen());
   }
}
class CreateCustomMenuScreen extends MainScreen
{
    Border _menuBorder;
    Background _menuBackground;
    Font _menuFont;

    CreateCustomMenuScreen()
    {
        setTitle("Custom Menu Sample");
        add(new RichTextField("Creating a custom menu"));

        XYEdges thickPadding = new XYEdges(10, 10, 10, 10);
        _menuBorder = BorderFactory.createRoundedBorder(thickPadding,
     Border.STYLE_DOTTED);
        _menuBackground = BackgroundFactory.createSolidTransparentBackground(Color
    .LIGHTSTEELBLUE, 50);
        try
        {
            FontFamily family = FontFamily.forName("BBCasual");
            _menuFont = family.getFont(Font.PLAIN, 30, Ui.UNITS_px);
        }
        catch(final ClassNotFoundException cnfe)
        {
            UiApplication.getUiApplication().invokeLater(new Runnable()
            {
                public void run()
                {
                    Dialog.alert("FontFamily.forName() threw " + cnfe.toString());
                }
            });
        }
    }
    protected void makeMenu(Menu menu, int context)
    {
        menu.setBorder(_menuBorder);
        menu.setBackground(_menuBackground);
        menu.setFont(_menuFont);
        super.makeMenu(menu, context);
    }
}
```

Adding an icon to a menu item

You can add an icon to a menu item by using the `MenuItem` in the `net.rim.device.api.ui` package. The BlackBerry® device calculates the size of the icon and centers the icon to the left or right of the menu item text when it lays out the menu. The icon aligns to the left or right of the menu item text depending on the direction of the text (for example, if the text direction is left-to-right, the icon aligns to the left of the text). If necessary, the BlackBerry device resizes the height and width of the the icon in relation to the height of the default font that the BlackBerry device application uses. If the BlackBerry device resizes the image, it preserves the aspect ratio of the image.

Note: You cannot add an icon to a predefined menu item (for example, `MenuItem.SAVE_CLOSE`).

Add an icon to a menu item

Before you begin: Add the icon to your project.

1. Import the required classes and interfaces.

   ```
   import net.rim.device.api.ui.*;
   import net.rim.device.api.ui.component.*;
   import net.rim.device.api.ui.container.*;
   import net.rim.device.api.ui.image.*;
   import net.rim.device.api.system.*;
   ```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `AddIconScreen` class, which is described in step 3, represents the custom screen.

```
public class AddIcon extends UiApplication
{
   public static void main(String[] args)
   {
      AddIcon theApp = new AddIcon();
      theApp.enterEventDispatcher();
   }
   public AddIcon()
   {
      pushScreen(new AddIconScreen());
   }
}
```

3. Create the custom screen for the application by extending the `MainScreen` class. In the screen constructor, invoke `setTitle()` to specify the title for the screen. Invoke `add()` to add a text field to the screen. Invoke `addMenuItem()` to add a menu item to the menu that `MainScreen` creates.

```
class AddIconScreen extends MainScreen
{
    public AddIconScreen()
    {
      setTitle("Menu Icon Sample");
      add(new RichTextField("Add an icon to a menu"));
      addMenuItem(_viewItem);
    }
}
```

4. In the screen constructor, specify the icon that you want to add to the menu item by using the `Image` class. Invoke `createImage()` to create the icon for the menu item. Invoke `setIcon()` to add the icon to the menu item.

```
Image menuIcon =
     ImageFactory.createImage(Bitmap.getBitmapResource("View.jpg"));
_viewItem.setIcon(menuIcon);
```

5. In the screen class, create the menu item by using the `MenuItem` class. Override `run()` to specify the action that occurs when the user clicks the menu item.

```
private MenuItem _viewItem = new MenuItem("More Info", 110, 10)
{
   public void run()
   {
      Dialog.inform("Display more information");
   }
};
```

Code sample: Adding an icon to a menu item

```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.ui.image.*;
import net.rim.device.api.system.*;

public class AddIcon extends UiApplication
{
    public static void main(String[] args)
    {
        AddIcon theApp = new AddIcon();
        theApp.enterEventDispatcher();
    }
    public AddIcon()
    {
        pushScreen(new AddIconScreen());
    }
}
class AddIconScreen extends MainScreen
{
    public AddIconScreen()
    {
     setTitle("Menu Icon Sample");
     add(new RichTextField("Add an icon to a menu"));

     Image menuIcon = ImageFactory.createImage(Bitmap.getBitmapResource("View
    .jpg"));
     _viewItem.setIcon(menuIcon);
     addMenuItem(_viewItem);
  }
  private MenuItem _viewItem = new MenuItem("More Info", 110, 10)
  {
         public void run()
         {
              Dialog.inform("Display more information");
         }
  };
}
```

Custom fonts

9

The `FontManager` class in the `net.rim.device.api.ui` package provides constants and methods that you can use to install and uninstall a TrueType font on a BlackBerry® device. The maximum size allowed for the TrueType font file is 60 KB. You can specify whether the font is available to the application that installs the font or to all applications on the BlackBerry device.

The `FontManager` class also provides methods to set the default font for the BlackBerry device or application.

Install and use a custom font in a BlackBerry Java application

1. Import the required classes and interfaces.

```
import net.rim.device.api.system.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.ui.component.*;
import java.util.*;
```

2. Create the application framework by extending the `UiApplication` class. In `main()`, create an instance of the new class and invoke `enterEventDispatcher()` to enable the application to receive events. In the application constructor, invoke `pushScreen()` to display the custom screen for the application. The `FontLoadingDemoScreen` class, described in step 3, represents the custom screen.

```
public class FontLoadingDemo extends UiApplication
{
    public static void main(String[] args)
    {
        FontLoadingDemo app = new FontLoadingDemo();
        app.enterEventDispatcher();
    }

    public FontLoadingDemo()
    {
        pushScreen(new FontLoadingDemoScreen());
    }
}
```

3. Create the custom screen by extending the `MainScreen` class. Invoke `setTitle()` to set the text that appears in the title section of the screen. Create a new `LabelField` object. You will apply the custom font to this object.

```
class FontLoadingDemoScreen extends MainScreen
{
    public FontLoadingDemoScreen()
    {
        setTitle("Font Loading Demo");
        LabelField helloWorld = new LabelField("Hello World");
    }
}
```

4. In the screen constructor, invoke the `FontManager.getInstance()` method to get a reference to the `FontManager` object, then invoke the `load()` method to install the font. Wrap the `load()` invocation in an IF statement to check if the installation was successful. The `load()` method returns a flag specifying if font is successfully installed. The following code specifies that the font that can be used only by the application.

```
if (FontManager.getInstance().load("Myfont.ttf", "MyFont",
     FontManager.APPLICATION_FONT) == FontManager.SUCCESS)
{
}
```

5. In the screen constructor, in a try/catch block in the IF statement you created in step 5, create a `Font` object for the font you just installed. Invoke the `setFont()` method to apply the font to the `LabelField` you created in step 5.

```
try
{
    FontFamily family = FontFamily.forName("MyFont");
    Font myFont = family.getFont(Font.PLAIN, 50);
    helloWorld.setFont(myFont);
}
catch (ClassNotFoundException e)
{
    System.out.println(e.getMessage());
}
```

6. In the screen constructor, invoke `add()` to add the `LabelField` to the screen.

```
add(helloWorld);
```

Code sample: Installing and using a custom font in a BlackBerry Java application

```
import net.rim.device.api.system.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.container.*;
import net.rim.device.api.ui.component.*;
import java.util.*;

public class FontLoadingDemo extends UiApplication
{
    public static void main(String[] args)
    {
        FontLoadingDemo app = new FontLoadingDemo();
        app.enterEventDispatcher();
    }
    public FontLoadingDemo()
    {
        pushScreen(new FontLoadingDemoScreen());
    }
}

class FontLoadingDemoScreen extends MainScreen
{
    public FontLoadingDemoScreen()
    {
        setTitle("Font Loading Demo");

        LabelField helloWorld = new LabelField("Hello World");

        if (FontManager.getInstance().load("Myfont.ttf", "MyFont",
     FontManager.APPLICATION_FONT) == FontManager.SUCCESS)
        {
            try
            {
                FontFamily typeface = FontFamily.forName("MyFont");
                Font myFont = typeface.getFont(Font.PLAIN, 50);
                helloWorld.setFont(myFont);
            }
            catch (ClassNotFoundException e)
            {
                System.out.println(e.getMessage());
            }
        }
        add(helloWorld);
    }
}
```

Spelling checker 10

You can use the items in the `net.rim.blackberry.api.spellcheck` package to add spelling checker functionality to an application. The `SpellCheckEngine` interface enables an application to check the spelling of a UI field value and provide a BlackBerry® device user with options for spelling corrections. The `SpellCheckUI` interface enables an application to provide a UI that allows a BlackBerry device user to resolve a spelling issue- by interacting with the `SpellCheckEngine` implementation.

For more information about using the Spell Check API, see the Spell Check sample application, which is provided with BlackBerry® Java® Development Environment 4.3.1 or later, and with the BlackBerry® Java® Plug-in for Eclipse®.

Add spell check functionality

1. Import the following classes:
   - `net.rim.blackberry.api.spellcheck.SpellCheckEngineFactory`
   - `java.lang.StringBuffer`
2. Import the following interfaces:
   - `net.rim.blackberry.api.spellcheck.SpellCheckEngine`
   - `net.rim.blackberry.api.spellcheck.SpellCheckUI`
   - `net.rim.blackberry.api.spellcheck.SpellCheckUIListener`
3. Create variables for spell check objects.
   ```
   SpellCheckEngine _spellCheckEngine;
   SpellCheckUI _spellCheckUI;
   ```
4. Invoke `createSpellCheckUI()`.
   ```
   _spellCheckUI = SpellCheckEngineFactory.createSpellCheckUI();
   ```
5. To notifiy an application when a spell check event occurs, invoke `addSpellCheckUIListener()` with a `SpellCheckUIListener` object as a parameter.
   ```
   _spellCheckUI.addSpellCheckUIListener(new SpellCheckUIListener());
   ```
6. To let an application spell check UI fields and suggest spelling corrections to a BlackBerry device user, obtain a `SpellCheckEngine` object, invoke `getSpellCheckEngine()`.
   ```
   _spellCheckEngine = _spellCheckUI.getSpellCheckEngine();
   ```
7. To use a correction for a misspelled word, invoke `SpellCheckEngine.learnCorrection()`. Use the parameters `newStringBuffer(text),newStringBuffer(correction)`, where `text` represents the misspelled word, and `correction` represents the correct word.

```
_spellCheckEngine.learnCorrection(new StringBuffer(text), new StringBuffer
(correction));
```

8. To perform spell check operations on a field, invoke `SpellCheckUI.spellCheck()`, with a `field` as a parameter.

```
_spellCheckUI.spellCheck(field);
```

9. To accept a misspelled word as correctly spelled, invoke `SpellCheckEngine.learnWord()`, with the word to learn as a parameter.

```
_spellCheckEngine.learnWord(new StringBuffer(word));
```

Listen for a spell check event

1. Import the following classes:
   - `java.lang.StringBuffer`
   - `net.rim.device.api.ui.UiApplication`
   - `net.rim.device.api.ui.Field`
2. Import the following interfaces:
   - `net.rim.blackberry.api.spellcheck.SpellCheckUIListener`
   - `net.rim.blackberry.api.spellcheck.SpellCheckEngine`
3. Create a method that returns the `SpellCheckUIListener.LEARNING_ACCEPT` constant when the `SpellCheckEngine` learns a new word.

```
public int wordLearned(SpellCheckUI ui, StringBuffer word) {
   UiApplication.getUiApplication().invokeLater(new popUpRunner("Word learned"));
   return SpellCheckUIListener.LEARNING_ACCEPT;
}
```

4. Create a method that returns the `SpellCheckUIListener.LEARNING_ACCEPT` constant when the `SpellCheckEngine` learns a correction for a word.

```
public int wordCorrectionLearned(SpellCheckUI ui, StringBuffer word, StringBuffer
correction){
   UiApplication.getUiApplication().invokeLater(new popUpRunner("Correction
learned"));
   return SpellCheckUIListener.LEARNING_ACCEPT;
}
```

5. Create a method that returns the `SpellCheckUIListener.ACTION_OPEN_UI` constant when the `SpellCheckEngine` finds a misspelled word.

```
public int misspelledWordFound(SpellCheckUI ui, Field field, int offset, int len){
   UiApplication.getUiApplication().invokeLater(new popUpRunner("Misspelled word
found"));
   return SpellCheckUIListener.ACTION_OPEN_UI;
}
```

Related resources

11

- www.blackberry.com/go/apiref: View the latest version of the API reference for the BlackBerry® Java® SDK.
- www.blackberry.com/go/devguides: Find development guides, release notes, and sample application overviews for the BlackBerry Java SDK.
- www.blackberry.com/developers: Visit the BlackBerry® Developer Zone for resources on developing BlackBerry device applications.
- www.blackberry.com/go/developerkb: View knowledge base articles on the BlackBerry Development Knowledge Base.
- www.blackberry.com/developers/downloads: Find the latest development tools and downloads for developing BlackBerry device applications.

Glossary

3-D	three-dimensional
API	application programming interface
JVM	Java® Virtual Machine
MIDP	Mobile Information Device Profile

Provide feedback

13

To provide feedback on this deliverable, visit www.blackberry.com/docsfeedback.

Document revision history

Date	Description
7 July 2010	Changed the following topics: • Code sample: Installing and using a custom font in a BlackBerry Java application
6 April 2010	Added the following topics: • Create a check box • Create a check box • Arranging UI components • Autocomplete text field • Buttons • Check boxes • Dialog boxes • Drop-down lists • Interaction methods on BlackBerry devices • Keyboard • Labels • Lists • Option buttons • Picker dialog boxes • Progress indicators • Search fields • Spin boxes • Text fields • Trackball or trackpad • Tree views Added the following code samples: • Code sample: Configuring a browser field • Code sample: Creating a check box

Date	Description
	• Code sample: Creating a drop-down list • Code sample: Creating a radio button • Code sample: Displaying HTML content from a resource in your application • Code sample: Displaying HTML content in a browser field • Code sample: Displaying HTML content from a web page in a browser field • Code sample: Sending form data to a web address in a browser field Changed the following topics: • Create a check box • Create a date field • Create a drop-down list • Create a radio button • Respond to a user touching the screen twice quickly • Specifying the orientation and direction of the screen • Spelling checker Deleted the following topics: • Best practice: Implementing submenus • Code sample: Creating a submenu • Code sample: Displaying a label at an absolute position on the screen • Create a layout manager • Create a screen • Create a submenu • Displaying a field at an absolute position on a screen • Display a label at an absolute position on the screen • Providing screen navigation when using a MainScreen
9 November 2009	Added the following topics: • Creating a grid layout • Create a grid layout • Create a file picker • Using data sources and fields with an autocomplete text field

Date	Description
	Added the following code samples: • Code sample: Creating a grid layout • Code sample: Creating a file picker
6 October 2009	Added the following topics: • Best practice: Implementing submenus • Create an autocomplete text field from a data set • Create an autocomplete text field from a data source • Create a date picker • Create a spin box • Creating a screen transition • Custom fonts • Install and use a custom font in a BlackBerry Java application Added the following code samples: • Code sample: Creating an autocomplete field from a data set • Code sample: Creating an autocomplete field from a data source • Code sample: Creating a screen transition • Code sample: Creating a spin box • Code sample: Installing and using a custom font in a BlackBerry Java application
14 August 2009	Added the following topics: • Add an icon to a menu item • Adding an icon to a menu item • Change the appearance of a menu • Changing the appearance of a menu • Configure a browser field • Display HTML content in a browser field • Create a menu • Create a submenu • Creating a field to display web content • Displaying a field at an absolute position on a screen

Date	Description
	• Display a ButtonField and a LabelField temporarily on the top and bottom of the screen • Display HTML content from a resource in your application • Display HTML content from a web page in a browser field • Display a label at an absolute position on the screen • Displaying fields on a temporary pair of managers • Event injection • Send form data to a web address in a browser field Added the following code samples: • Code sample: Adding an icon to a menu item • Code sample: Changing the appearance of a menu • Code sample: Creating a date picker • Code sample: Creating a menu • Code sample: Creating a submenu • Code sample: Displaying a ButtonField and a LabelField temporarily on the top and bottom of the screen • Code sample: Displaying a label at an absolute position on the screen Changed the following topics: • Adding a menu item to a BlackBerry Device Software application • Add a menu item to a BlackBerry Device Software application • Receive notification when the size of the drawable area of the touch screen changes • Retrieving accelerometer data • Retrieve accelerometer data at specific intervals • Specifying the orientation and direction of the screen • Store accelerometer readings in a buffer

Legal notice 15

Research In Motion Limited
295 Phillip Street

Waterloo, ON N2L 3W8
Canada

Research In Motion UK Limited
Centrum House
36 Station Road
Egham, Surrey TW20 9LF
United Kingdom

Published in Canada