BlackBerry Java SDK

UI and Navigation

版本: 6.0

开发指南

BlackBerry

发布日期： 2011-02-09
SWD-1239696-0209022105-011

内容

创建与 BlackBerry 标准 UI 一致的 UI

1

您可以使用标准 MIDP API 和 BlackBerry® UI API 创建 BlackBerry® Java® Application UI。

BlackBerry UI API 中的 UI 组件旨在提供与 BlackBerry® Device Software 应用程序一致的布局和行为。使用这些 API 不仅让您可以使用 BlackBerry 用户熟悉的组件，而且节省您的时间，因为无需自己创建组件。

- 屏幕组件可提供标准屏幕布局、默认菜单，以及当 BlackBerry 设备用户按退出键、单击触控板或者触按触摸屏时的标准行为。
- 字段组件提供用于日期选择、选项按钮、复选框、列表、文本字段、标签和进度栏控件的标准 UI 元素。
- 布局管理器可向应用程序提供相关功能，以在 BlackBerry 设备屏幕上以标准方式排列组件，如水平排列、垂直排列或以从左到右的顺序排列。

您可以使用 BlackBerry UI API 创建包括表格、网格及其它专用功能的 UI。 您可以使用标准 Java 事件模型接收和响应特定类型的事件。 BlackBerry 设备应用程序可接收和响应用户事件（如当用户单击触控板或在键盘上键入时），也可以接收并响应系统事件（如全局提醒、时钟更改及 USB 端口连接）。

BlackBerry 设备用户输入和导航

2

BlackBerry® 设备包括键盘、拨轮、轨迹球或触控板以及退出键，用于输入和导航。 在具有 SurePress™ 触摸屏的 BlackBerry 设备上，单击屏幕等同于单击拨轮或轨迹球。

BlackBerry® Java® Application 应尽可能使用以下输入和导航模式。

- 单击拨轮、轨迹球、触控板或屏幕通常调用菜单。
- 按退出键可取消操作或返回上个屏幕。 重复按退出键可让用户返回主屏幕。 按住退出键可关闭浏览器或多媒体应用程序。

默认情况下，BlackBerry 屏幕对象无需自定义即提供这些功能；但是，您必须添加菜单项以及其他 UI 和导航逻辑。

触摸屏

在带有 SurePress™ 触摸屏的 BlackBerry® 设备上，用户使用手指与设备上的应用程序交互。 用户可在触摸屏上执行各种操作，以便键入文本并导航屏幕。

用户也可通过单击快捷方式栏上的图标或按菜单键来执行操作。

在带有触摸屏的 BlackBerry 设备上，用户可执行以下操作：

操作	结果
轻按屏幕	此操作可高亮显示项目。 在文本字段中，如果用户触摸光标附近的屏幕，则将围绕光标显示一个轮廓方框。 此方框有助于用户更轻易地重新定位光标。
点击屏幕	在支持全屏视图的应用程序中，比如 BlackBerry® Maps 和 BlackBerry® Browser，此操作可隐藏和显示快捷方式栏。
双击屏幕	此操作可放大网页、地图、图片或演示文稿附件。
用手指按住项目	在快捷方式栏上，此操作将显示一个工具提示，其中说明了图标表示的操作。 在消息列表中，当用户用手指按住发件人或消息主题时，BlackBerry 设备将搜索该发件人或主题。
在屏幕上触摸并拖动项目	此操作可沿相应的方向移动屏幕上的内容。 例如，当用户触摸并拖动某菜单项时，各菜单项的列表就向相同的方向移动。 在文本字段中，此操作可向同一方向移动轮廓方框和光标。
同时触摸屏幕中的两个位置	此操作可高亮显示两个位置之间的文本和项目列表，比如，消息。 要添加或删除高亮显示的文本或项目，用户可以触摸屏幕的另一个位置。
单击（按）屏幕	此操作可启动操作。 例如，当用户单击列表中某项目时，与该项目关联的屏幕就会显示。 此操作相当于单击拨轮、轨迹球或触控板。

操作	结果
	此操作可放大地图、图片或演示文稿附件。 在网页上，此操作可放大网页或跟随链接。 在文本字段中，此操作可定位光标。 如果字段包含文本，则其周围将显示一个轮廓方框。
在屏幕上快速向上或向下滑动手指	快速向上滑动手指可显示下一个屏幕。 快速向下滑动手指可显示上一个屏幕。 当键盘出现时，快速向下滑动手指可隐藏键盘并显示快捷方式栏。
在屏幕上向左或向右快速滑动手指	此操作显示下一个或上一个图片或消息，或显示日历中的上一或下一工作日、周或月。
在屏幕上向上或向下滑动手指	在相机中，滑动手指可放大对象。 向下滑动手指可缩小对象。
沿任何方向滑动手指	此操作可平移地图或网页。 当用户放大图片时，此操作也可平移图片。
按 Escape 键	此操作可取消高亮显示文本或项目列表。 在网页、地图或图片上，此操作可缩小一级。 用户按 Escape 键两次可返回原始视图。

轨迹球或触控板

在带轨迹球或触控板的 BlackBerry® 设备上，轨迹球或触控板是用户导航的主要控件。 用户可执行以下操作：

- 滚动轨迹球或在触控板上滑动手指以移动光标。
- 单击轨迹球或触控板以执行默认操作或打开上下文菜单。
- 单击轨迹球或触控板，同时按 Shift 键，以高亮显示文本或高亮显示消息列表中的消息。

带轨迹球或触控板的 BlackBerry 设备包括位于轨迹球或触控板左侧的菜单键。 用户可以按菜单键以打开可用操作的完整菜单。

轨迹球敏感度

轨迹球敏感度是指系统将移动标识为导航事件并将导航事件发送至软件层时需要的轨迹球移动量。 BlackBerry® 设备硬件使用名为刻度的单位衡量轨迹球移动。 沿某条轴的刻度数超过系统或 BlackBerry 设备用程序的阈值时，导航事件将发送至软件层，而系统会将刻度计数重置为零。 某个空闲时间过后，刻度计数也将重置为零。

您可以使用轨迹球 API 设置轨迹球敏感度。 高轨迹球敏感度相当于较小的刻度阈值（较小的轨迹球移动量就将生成导航事件）。 低轨迹球敏感度相当于较大的刻度阈值（需要较大的轨迹球移动量才能生成导航事件）。

轨迹球移动

您可以使用轨迹球 API 过滤 BlackBerry® 设备硬件发送至软件层的轨迹球移动数据。 轨迹球 API 可过滤出移动 "噪音" 或不需要的移动。

您也可以使用轨迹球 API 更改设置，如轨迹球移动加速。 只要用户持续滚动轨迹球，增加轨迹球移动加速设置就可导致软件层将轨迹球移动标识为移动速率大于 BlackBerry 设备硬件检测到的速率。 轨迹球敏感度将临时增加，因为用户持续滚动轨迹球，无暂停。

拨轮

BlackBerry® Pearl™ 8100 Series 以前的 BlackBerry®设备使用拨轮作为用户导航的主要控件。 拨轮位于 BlackBerry® 设备的右侧。

用户可执行以下操作：

- 滚动拨轮以垂直移动光标
- 按住 Alt 键同时滚动拨轮以水平移动光标
- 单击拨轮以启动操作或打开菜单

键盘

用户主要使用键盘键入文本。 在不带触摸屏的 BlackBerry® 设备上，用户也可使用键盘围绕屏幕移动（例如，围绕地图移动）。 但是，使用键盘进行导航始终只是一种备选方法，使用拨轮、轨迹球或触控板进行导航是首选方法。

带拨轮或轨迹球的 BlackBerry 设备有 QWERTY 键盘或 SureType® 键盘。 带拨轮的 BlackBerry 设备有 QWERTY 键盘。 两种类型的键盘均包括字符键和修饰语键。 字符键向 BlackBerry 设备发送字符，包括文本键、菜单键和 Escape 键。 修饰词键更改字符键的功能。 修饰词键包括 Shift 键和 Alt 键。

在带有 SurePress™ 触摸屏的 BlackBerry 设备上，大多数情况下，QWERTY 键盘显示在纵向视图中，而 SureType® 键盘显示在横向视图中。

BlackBerry 设备上的交互方法

BlackBerry 设备型号	交互方法
BlackBerry® 7100 Series	拨轮
BlackBerry® 8700 Series	拨轮
BlackBerry® 8800 Series	轨迹球
BlackBerry® Bold™ 9000 智能手机	轨迹球
BlackBerry® Bold™ 9650 智能手机	触控板
BlackBerry® Bold™ 9700 智能手机	
BlackBerry® Curve™ 8300 Series	轨迹球
BlackBerry® Curve™ 8500 Series	触控板
BlackBerry® Curve™ 8900 智能手机	轨迹球
BlackBerry® Pearl™ 8100 Series	轨迹球
BlackBerry® Pearl™ Flip 8200 Series	轨迹球

BlackBerry 设备型号	交互方法
BlackBerry® Storm™ 9500 Series	触摸屏
BlackBerry® Tour™ 9630 智能手机	轨迹球

屏幕

3

BlackBerry® 设备 UI 的主要组件是 Screen 对象。

BlackBerry 设备可同时打开多个屏幕，但一次只能显示一个屏幕。 BlackBerry® Java® Virtual Machine 在显示堆栈中按顺序维护一组 Screen 对象。 堆栈顶部的屏幕是 BlackBerry 设备用户看到的活动屏幕。 BlackBerry 设备应用程序显示屏幕时，它会将屏幕推至堆栈顶部。 BlackBerry 设备应用程序关闭屏幕时，它会将屏幕推出屏幕顶部并显示堆栈上的下个屏幕，以便根据需要重新绘制屏幕。

每个屏幕都只能在显示堆栈中显示一次。 如果 BlackBerry 设备应用程序推至堆栈上的屏幕已存在，BlackBerry JVM 将引发运行时例外。 用户完成与屏幕的交互后，BlackBerry 设备应用程序必须将屏幕推出显示堆栈，这样 BlackBerry 设备应用程序才能有效使用内存。 一次只能使用几个模式屏幕，因为每个屏幕都使用单独的线程。

UI API 将初始化简单的 Screen 对象。 创建屏幕后，可添加字段和菜单，且通过将其推入显示堆栈，可向用户显示屏幕。 您可根据需要更改 BlackBerry 设备 UI 和实施新字段类型。 您也可以添加自定义导航。

Screen 类不实施消除歧义，而这是复杂输入法（如国际键盘）需要的。 要集成各种输入法，可扩展 Field 类或其子类之一。 您无法将 Screen 对象用于键入文本。

Screen 类

类	说明
Screen	您可以使用 Screen 类创建屏幕，并使用布局管理器在屏幕上放置 UI 组件。 通过使用 Field 超类中的常量定义的样式，可定义特定类型的屏幕。
FullScreen	您可以使用 FullScreen 类创建空屏幕，以便可采用标准垂直布局形式将 UI 组件添加至其中。 如果要使用其它布局样式，如水平或对角，则可使用 Screen 类并在其中添加布局管理器。
MainScreen	您可以使用 MainScreen 类创建带以下标准 UI 组件的屏幕： • 默认屏幕标题，其中 SeparatorField 位于标题后 • VerticalFieldManager 中包含的主要可滚动部分 • 带“关闭”菜单项的默认菜单 • BlackBerry® 设备用户单击“关闭”菜单项或按退出键时关闭屏幕的默认关闭操作 您应该考虑将 MainScreen 对象用于 BlackBerry 设备应用程序的第一个屏幕，以保持与其它 BlackBerry 设备应用程序的一致性。

管理绘图区域

Graphics 对象表示 BlackBerry® 设备应用程序可用的整个绘制表面。 要限制此区域，请将其分为多个 XYRect 对象。 每个 XYPoint 都表示屏幕上的一个点，而该点由 X 坐标和 Y 坐标组成。

1. 导入以下类：
   - net.rim.device.api.ui.Graphics
   - net.rim.device.api.ui.XYRect
   - net.rim.device.api.ui.XYPoint
2. 创建 XYPoint 对象，以表示矩形的左上点和右下点。 使用 XYPoint 对象创建 XYRect 对象，以创建矩形剪辑区域。

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

3. 调用 Graphics.pushContext() 以提出指定区域原点不能调整绘图抵消的绘图要求。 调用 Graphics.pushContext() 以将矩形剪辑区域推入上下文堆栈。 调用 Graphics.drawRect() 以绘制矩形，并调用 Graphics.fillRect() 以填充矩形。 调用 Graphics.popContext() 以关闭上下文堆栈的当前上下文。

```
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. 调用 pushRegion() 并指定区域原点应调整绘图抵消。 调用 Graphics.drawRect() 以绘制矩形，并调用 Graphics.fillRect() 以填充矩形。 调用 Graphics.popContext() 以关闭上下文堆栈的当前上下文。

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

5. 指定要推入堆栈的 Graphics 对象部分。
6. 调用 pushContext()（或 pushRegion()）后，提供要反转的 Graphics 对象部分。

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

7. 调用 translate()。 XYRect 将从其 (1, 1) 原点转换为 (20, 20) 原点。 转换后，XYRect 对象将扩展经过图形上下文的边界，并剪辑它。

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

创建屏幕转换

您可以创建屏幕转换，以应用在应用程序打开或关闭 BlackBerry® 设备上的屏幕时显示的视觉效果。 通过使用 net.rim.device.api.ui.TransitionContext 类，可为应用程序创建以下类型的屏幕转换。

转换	说明
TRANSITION_FADE	通过淡入或淡出，这种转换将显示或删除屏幕。 通过更改屏幕的不透明度，这种屏幕转换可创建消失视觉效果。
TRANSITION_SLIDE	通过在设备上以滑动方式打开或关闭显示，这种转换将显示或删除屏幕。 您可以使用各种属性指定新屏幕和当前屏幕都滑动，以创建两种屏幕都在移动的效果，或指定新屏幕滑动到当前屏幕上。
TRANSITION_WIPE	通过模拟在设备上以擦除方式打开或关闭显示，这种转换将显示或删除屏幕。
TRANSITION_ZOOM	通过在设备上放大或缩小显示，这种转换将显示或删除屏幕。
TRANSITION_NONE	无转换发生。

每种类型的屏幕转换都具有可用于自定义屏幕转换的视觉效果的属性。 例如，您可以自定义滑动效果，以便屏幕从设备显示底部滑动至显示顶部。 如果未自定义屏幕转换，应用程序将使用默认属性。 有关这些默认属性的详细信息，请参阅 BlackBerry® Java® Development Environment 的 API 参考。

创建屏幕转换后，必须通过调用 UiEngineInstance.setTransition() 在应用程序内注册它，并指定要删除的传入屏幕和要显示的传入屏幕、导致发生转换的事件和要显示的转换。

要下载演示如何使用屏幕转换的示例应用程序，请访问 www.blackberry.com/go/screentransitionssample。 有关屏幕转换的详细信息，请参阅 BlackBerry Java Development Environment 的 API 参考。

注： BlackBerry 设备上的主题控制用户打开应用程序时显示的视觉效果。 有关详细信息，请参阅 *《BlackBerry Theme Studio User Guide》*。

代码示例：创建屏幕转换

以下代码示例说明了滑动转换和消失转换。 用户打开应用程序时，第一个屏幕将显示在 BlackBerry® 设备上并显示一个按钮。 用户单击该按钮时，第二个屏幕将从右滑动显示。 两秒后，第二个屏幕将自动从显示中消失。

```
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);
    }
}
```

指定屏幕的方位和方向

在触摸屏应用程序中，您可以同时考虑屏幕的方位和方向。 方位与屏幕的宽高比相关。 方向与屏幕的绘图区域相关。

方位

触摸屏的 BlackBerry® 设备用户可通过旋转设备更改屏幕的方位。 如果用户在 BlackBerry 徽标位于顶部时停下 BlackBerry 设备，则屏幕为纵向。 如果用户将设备向左或向右旋转 90 度，则屏幕为横向。

您可以使用 `net.rim.device.api.system.Display` 类检索屏幕的方位。 `Display` 类包含对应于触摸屏的设备可用于显示信息的方位的常量。 例如，纵向、横向和正方形方位对应于 `Display.ORIENTATION_PORTRAIT`、`Display.ORIENTATION_LANDSCAPE` 和 `Display.ORIENTATION_SQUARE` 常量。

要检索屏幕的方位，可调用 `Display.getOrientation()` 方法。 此方法将返回一个对应于 BlackBerry 设备可使用的方位之一的整型值。

要检测方位更改，可覆盖 `Screen.sublayout()`。 在该方法中，调用 `super.sublayout()` 并监视 `width` 和 `height` 值的更改。

方向

触摸屏的 BlackBerry 设备可采用各种方向在屏幕上显示信息。 方向是指屏幕绘图区域的顶部，与 BlackBerry 徽标的位置相关。 可绘图区域的顶部是最靠近 BlackBerry 徽标的屏幕侧时，方向为北；可绘图区域的顶部位于 BlackBerry 徽标左侧时，方向为西；可绘图区域的顶部位于 BlackBerry 徽标右侧时，方向为东。

您可以使用 `net.rim.device.api.system.Display` 类、`net.rim.device.api.ui.Ui` 类和 `net.rim.device.api.ui.UiEngineInstance` 类控制 BlackBerry 设备用于在屏幕上显示信息的方向。 `Display` 类包含对应于设备可用于显示信息的方向的常量。 例如，北、西和东方向对应于 `Display.DIRECTION_NORTH`、`Display.DIRECTION_WEST` 和 `Display.DIRECTION_EAST` 常量。

您可以创建 `net.rim.device.api.ui.Ui` 类的对象并调用 `Ui.getUiEngineInstance()`，以检索 `UiEngineInstance` 类的对象。 使用对应于 `Display` 类中的方向的常量之一调用 `UiEngineInstance.setAcceptableDirections()` 可设置 BlackBerry 设备可用于显示信息的方向。

代码示例：检索屏幕方位

```
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;
}
```

代码示例：强制在 BlackBerry API 应用程序中使用纵向视图

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

代码示例：强制在 MIDlet 应用程序中使用横向视图

```
// 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);
```

检索触摸屏的方位

1. 导入 net.rim.device.api.system.Display 类。
2. 调用 net.rim.device.api.system.Display.getOrientation()。

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

限制触摸屏方向

1. 导入以下类：
   - net.rim.device.api.ui.Ui
   - net.rim.device.api.ui.UiEngineInstance
2. 调用 net.rim.device.api.ui.Ui.getUiEngineInstance()。

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

3. 调用 net.rim.device.api.ui.UiEngineInstance.setAcceptableDirections(byte flags) 并传递 Screen 的方向自变量。

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

触摸屏的可绘图区域大小更改时接收通知

1. 导入以下类：
   - javax.microedition.lcdui.Canvas
   - net.rim.device.api.ui.component.Dialog

2. 扩展 javax.microedition.lcdui.Canvas。
3. 覆盖 Canvas.sizeChanged(int, int)。

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

加速度计

4

带触摸屏的 BlackBerry® 设备包括加速度计，这旨在探测 BlackBerry 设备的方位和加速。 BlackBerry 设备用户移动 BlackBerry 设备时，加速度计可探测 3-D 空间中沿 x、y 和 z 轴的移动。 设备用户可更改设备方位，从而在纵向和横向之间更改 BlackBerry 设备应用程序的屏幕显示方向。

您可以使用 net.rim.device.api.system 数据包中的加速度计 API，以对 BlackBerry 设备的方位和加速做出响应。 例如，用户以各种速度移动和旋转 BlackBerry 设备时，可操控游戏应用程序，以更改屏幕上移动对象的方向和速度。

要下载演示如何使用加速度计的示例应用程序，请访问 www.blackberry.com/go/accelerometersample。有关示例应用程序的详细信息，请访问 www.blackberry.com/go/devguides 以阅读 *加速度计示例应用程序概述*。

加速度计数据的类型

BlackBerry® 设备应用程序可从加速度计检索数据。

数据类型	说明
方位	BlackBerry 设备相对于地面的方位。
加速	BlackBerry 设备的旋转加速。

有关加速度计中的数据类型的详细信息，请参阅 BlackBerry® Java® Development Environment 的 API 参考。

检索加速度计数据

加速度计旨在跟踪 BlackBerry® 设备用户移动 BlackBerry 设备时沿 x、y 和 z 轴的移动方向和速度。 x 轴与 BlackBerry 设备的宽度平行。 y 轴与 BlackBerry 设备的长度平行。 z 轴与 BlackBerry 设备的深度（前后）平行。 有关加速度计的 x、y 和 z 轴的详细信息，请参阅 BlackBerry® Java® Development Environment 的 API 引用中的 net.rim.device.api.system.AccelerometerSensor 类。

您可以让 BlackBerry 设备应用程序对包括加速度计的 BlackBerry 设备的加速度做出反应。 例如，BlackBerry 设备用户可移动 BlackBerry 设备，以控制游戏应用程序中在迷宫中移动的对象的方向和速度。

您可以使用 net.rim.device.api.system 数据包中的加速度计 API，以对 BlackBerry 设备的加速度做出反应。 您必须先确定 BlackBerry 设备是否支持加速度计，方法是调用 net.rim.device.api.system.AccelerometerSensor.isSupported()。 如果方法返回值 true，则 BlackBerry 设备支持加速度计。

您可以使用 AccelerometerData 类标识用户移动 BlackBerry 设备的方向。 调用 AccelerometerData.getOrientation() 将返回其中一个 AccelerometerSensor 类常量，这对应于 BlackBerry 设备的方向。 例如，当 AccelerometerData.getOrientation() 返回的值等于 AccelerometerSensor.ORIENTATION_LEFT_UP 时，BlackBerry 设备的左侧将为向上的方向。

您可以使用 AccelerometerSensor 类从 BlackBerry 设备检索加速度数据。 调用 AccelerometerSensor.openRawDataChannel() 将返回 net.rim.device.api.system.AccelerometerSensor.Channel 类的对象。AccelerometerSensor.Channel 类允许打开至加速度计的连接。 通过调用 AccelerometerSensor.Channel.getLastAccelerationData()，可从加速度计检索数据。

维护至加速度计的连接使用 BlackBerry 设备电池的电量。 当 BlackBerry 设备应用程序不再需要从加速度计检索数据时，应该调用 AccelerometerSensor.Channel.close() 以关闭连接。

代码示例：从加速度计检索数据

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

按特定间隔检索加速度计数据

如果 BlackBerry® 设备应用程序在应用程序位于前台时打开至加速度计的通道，在应用程序位于后台时，该通道将暂停且不查询加速度计。 如果 BlackBerry 设备应用程序按较短的间隔调用 AccelerometerSensor.Channel. getLastAccelerationData(short[]) 或 BlackBerry 设备未在移动，该方法可能会返回重复值。

1. 导入以下类：
   - net.rim.device.api.system.AccelerometerSensor.Channel;
   - net.rim.device.api.system.AccelerometerSensor;
2. 打开至加速度计的通道。 通道打开时，BlackBerry 设备应用程序将在加速度计中查询信息。

   ```
   Channel rawDataChannel =
   AccelerometerSensor.openRawDataChannel( Application.getApplication() );
   ```
3. 创建数组以存储加速度计数据。

   ```
   short[] xyz = new short[ 3 ];
   ```
4. 创建线程。

   ```
    while( running ) {
   ```
5. 调用 Channel.getLastAccelerationData(short[]) 以从加速度计检索数据。

   ```
   rawDataChannel.getLastAccelerationData( xyz );
   ```
6. 调用 Thread.sleep() 以指定加速度计查询间隔(以毫秒为单位)

   ```
   Thread.sleep( 500 );
   ```
7. 调用 Channel.close() 以关闭至加速度计的通道。

   ```
   rawDataChannel.close();
   ```

当应用程序位于前台时查询加速度计

1. 导入以下类：
   - net.rim.device.api.system.AccelerometerChannelConfig
   - net.rim.device.api.system.AccelerometerSensor.Channel
2. 打开从加速度计检索加速度数据的通道。

```
Channel channel =
AccelerometerSensor.openRawAccelerationChannel( Application.getApplication());
```

3. 调用 Thread.sleep() 以指定加速度计查询间隔（以毫秒为单位）

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

4. 调用 Channel.close() 以关闭至加速度计的通道。

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

当应用程序位于后台时查询加速度计

1. 导入以下类：
   - net.rim.device.api.system.AccelerometerChannelConfig;
   - net.rim.device.api.system.AccelerometerSensor.Channel;
2. 创建至加速度计的通道的配置。

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

3. 调用 AccelerometerChannelConfig.setBackgroundMode(Boolean) 指定对后台应用程序的支持。

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

4. 调用 AccelerometerSensor.openChannel() 以打开至加速度计的后台通道。

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

5. 调用 Thread.sleep() 以指定加速度计查询间隔（以毫秒为单位）

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

6. 调用 Channel.close() 以关闭至加速度计的通道。

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

将加速度计读数存储在缓冲区中

1. 导入以下类：
   - net.rim.device.api.system.AccelerometerChannelConfig;
   - net.rim.device.api.system.AccelerometerSensor.Channel;
2. 创建至加速度计的通道的配置。

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

3. 调用 AccelerometerChannelConfig.setSamplesCount()，以指定要存储在缓冲区中的加速读数个数。缓冲区中的每个元素都包含 x、y 和 z 轴的加速读数以及有关读取时间的数据。

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

4. 调用 AccelerometerSensor.openChannel() 以打开至加速度计的通道。

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

从缓冲区检索加速度计读数

1. 导入以下类：
   - net.rim.device.api.system.AccelerometerData;
   - net.rim.device.api.system.AccelerometerSensor.Channel;
2. 在缓冲区中查询加速度计数据。

```
AccelerometerData accData = bufferedChannel.getAccelerometerData();
```

3. 调用 AccelerometerData.getNewBatchLength() ，以获取自上次查询检索到的读数个数。

```
int newBatchSize = accData.getNewBatchLength();
```

4. 调用 AccelerometerData.getXAccHistory()、AccelerometerData.getYAccHistory() 和 AccelerometerData.getZAccHistory()，以从每个轴的缓冲区检索加速度计数据。

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

获取从加速度计读取的时间

1. 导入 net.rim.device.api.system.AccelerometerData 类。
2. 在缓冲区中查询加速度计数据。

```
AccelerometerData accData;
accData = bufferedChannel.getAccelerometerData();
```

3. 调用 AccelerometerData.getSampleTsHistory()。

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

事件

5

响应导航事件

您可以使用 Screen 导航方法创建 BlackBerry® 设备应用程序。 您可以使用 BlackBerry 设备应用程序实施 TrackwheelListener 接口，请更新 BlackBerry 设备应用程序，以使用 Screen 导航方法。

1. 导入 net.rim.device.api.ui.Screen 类。
2. 通过扩展 net.rim.device.api.ui.Screen 类(或其子类之一)并覆盖以下导航方法,管理导航事件：

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

确定输入法的类型

1. 导入以下一个或多个类：
   - net.rim.device.api.ui.Screen
   - net.rim.device.api.ui.Field
2. 导入 net.rim.device.api.system.KeypadListener 接口。
3. 实施 net.rim.device.api.system.KeypadListener 接口。
4. 扩展 Screen 类、Field 类或同时扩展两者。
5. 在实施 navigationClick、navigationUnclick 或 navigationMovement 方法中的其中一个时,在 status 参数上执行按位 AND 操作,以获得有关事件的详细信息。 实施 navigationClick(int status, int time) 方法,以确定拨轮或四向导航输入设备(如轨迹球)是否触发事件。

```
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);
}
```

有关 net.rim.device.api.system.KeypadListener 类的状态修饰符的详细信息,请参阅 BlackBerry® Java® Development Environment 的 API 参考。

触摸屏交互模型

带触摸屏和触控板的 BlackBerry® 设备使用弱单击交互模型，其中 BlackBerry 设备用户点按屏幕或使用触控板来执行操作。 此模型与带 SurePress™ 触摸屏的 BlackBerry 设备不同，其中用户单击（或按）屏幕来执行操作。

在这两种类型的设备上，用户触按屏幕时，将发生 TouchEvent.DOWN 事件。

在带 SurePress 触摸屏的设备上，用户应用姿势并单击屏幕时，将发生 TouchEvent.CLICK 事件。 压力降低时，将发生 TouchEvent.UNCLICK 事件。 通常，将在发生 TouchEvent.UNCLICK 事件时执行操作（如根据按钮调用操作）。 用户完全释放屏幕时，将发生 TouchEvent.UP 事件。

在带触摸屏且支持弱单击的设备上，无物理单击和取消单击。 相反，预计将在用户点按屏幕时执行操作。点按是 TouchEvent.DOWN 事件后紧跟 TouchEvent.UP 事件。 此触摸事件组合称为姿势，因此将发生 TouchGesture。 点按时，将发生 TouchGesture.TAP 事件（以及 TouchEvent.DOWN 和 TouchEvent.UP 事件）。

为了在这两种类型的触摸屏交互模型之间实现更大的兼容性，在支持弱单击的设备上，TouchEvent.CLICK 事件和 TouchEvent.UNCLICK 事件将在 TouchGesture.TAP 事件后发生，因此可在所有触摸屏设备上在发生 TouchEvent.UNCLICK 事件时调用操作，而不管设备是否具有 SurePress 触摸屏。

用户操作	支持弱单击的设备	带 SurePress 触摸屏的设备
触按屏幕。	TouchEvent.DOWN	TouchEvent.DOWN
在项目上触按并按住手指。	TouchGesture.HOVER	TouchGesture.HOVER
单击屏幕。	—	TouchEvent.CLICK
单击并按住屏幕。	—	TouchEvent.CLICK_REPEAT
释放屏幕。	—	TouchEvent.UNCLICK
从屏幕拿开手指。	TouchEvent.UP	TouchEvent.UP
点按屏幕。	TouchEvent.DOWN TouchGesture.TAP TouchEvent.CLICK TouchEvent.UNCLICK TouchEvent.UP	TouchEvent.DOWN TouchGesture.TAP TouchEvent.UP

通过调用 DeviceCapability.isTouchClickSupported()，可确定带触摸屏的设备是否支持弱单击。 如果设备支持弱单击，此方法将返回 false。 如果设备具有 SurePress 触摸屏，此方法将返回 true。

响应触摸屏事件

BlackBerry® 设备用户可在触摸屏的设备上执行与在轨迹球的 BlackBerry 设备上相同的操作。 例如，BlackBerry 设备用户可使用触摸屏打开菜单、滚动选项列表和选择选项。

如果使用早于 BlackBerry® Java® Development Environment 4.7 版的版本创建 BlackBerry 设备应用程序，则可以修改应用程序的 .jad 文件，以在 BlackBerry 设备用户触摸触摸屏时让应用程序响应。 在 .jad 文件中，可将 RIM-TouchCompatibilityMode 选项设置为 false。

如果使用 BlackBerry JDE 4.7 或更高版本创建 BlackBerry 设备应用程序，则可让应用程序标识 BlackBerry 设备用户在触摸屏上执行的操作，方法是扩展 net.rim.device.api.ui.Screen 类、net.rim.device.api.ui.Field 类或 Field 类的子类之一并覆盖 touchEvent() 方法。 在 touchEvent() 方法内，比较 TouchEvent.getEvent() net.rim.device.api.ui.TouchEvent 类和 net.rim.device.api.ui.TouchGesture 类的常量的值。

TouchEvent 类包含表示用户可在触摸屏上执行的各种操作的常量。 例如，单击、触摸和移动操作对应于 TouchEvent.CLICK、TouchEvent.DOWN 和 TouchEvent.MOVE 常量。

TouchGesture 类包含表示用户可在触摸屏上执行的各种姿势的常量。 例如，点按和滑动姿势对应于 TouchGesture.TAP 和 TouchGesture.SWIPE 常量。

代码示例：标识用户在触摸屏上执行的操作

以下代码示例使用 switch 语句标识操作。

```
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;
}
```

用户触摸屏幕时响应系统事件

1. 导入以下类：
   - 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. 创建扩展 Manager 类、Screen 类、Field 类或其中一个 Field 子类的类。

   ```
   public class newButtonField extends ButtonField {
   ```
3. 在 touchEvent(TouchEvent message) 方法的实施中，调用 TouchEvent.getEvent()。
4. 检查 TouchEvent.getEvent() 返回的值是否等于 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;
}
```

响应在屏幕上快速向上滑动手指的用户

1. 导入以下类：
   - 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. 创建扩展 Manager 类、Screen 类、Field 类或其中一个 Field 子类的类。

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

3. 在 touchEvent(TouchEvent message) 方法的实施中，调用 TouchEvent.getEvent()。
4. 检查 TouchGesture.getSwipeDirection() 返回的值是否等于 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;
   }
}
```

响应在屏幕上快速向下滑动手指的用户

1. 导入以下类：
   - 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. 创建扩展 Manager 类、Screen 类、Field 类或其中一个 Field 子类的类。

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

3. 在 touchEvent(TouchEvent message) 方法的实施中，调用 TouchEvent.getEvent()。
4. 检查 TouchGesture.getSwipeDirection() 返回的值是否等于 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;
   }
}
```

响应在屏幕上向左快速滑动手指的用户

1. 导入以下类：
   - 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. 创建扩展 Manager 类、Screen 类、Field 类或其中一个 Field 子类的类。

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

3. 在 touchEvent(TouchEvent message) 方法的实施中，调用 TouchEvent.getEvent()。
4. 检查 TouchGesture.getSwipeDirection() 返回的值是否等于 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;
        }
    }
```

响应在屏幕上向右快速滑动手指的用户

1. 导入以下类：
   - 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. 创建扩展 Manager 类、Screen 类、Field 类或其中一个 Field 子类的类。

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

3. 在 touchEvent(TouchEvent message) 方法的实施中，调用 TouchEvent.getEvent()。
4. 检查 TouchGesture.getSwipeDirection() 返回的值是否等于 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;
   }
}
```

响应单击屏幕的用户

1. 导入以下类：
   - 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. 创建扩展 Manager 类、Screen 类、Field 类或其中一个 Field 子类的类。

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

3. 在 touchEvent(TouchEvent message) 方法的实施中，调用 TouchEvent.getEvent()。
4. 检查 TouchEvent.getEvent() 返回的值是否等于 TouchEvent.CLICK。

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

响应快速触摸两次屏幕的用户

1. 导入以下类：
   - 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. 创建扩展 Manager 类、Screen 类、Field 类或其中一个 Field 子类的类。

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

3. 在 touchEvent(TouchEvent message) 方法的实施中，检查是否发生 TouchGesture.TAP 事件以及 TouchGesture.getTapCount 是否返回 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;
}
```

响应在屏幕上触摸并拖动项目的用户

1. 导入以下类：

- 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. 创建扩展 Manager 类、Screen 类、Field 类或其中一个 Field 子类的类。

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

3. 在 touchEvent(TouchEvent message) 方法的实施中，调用 TouchEvent.getEvent()。
4. 检查 TouchEvent.getEvent() 返回的值是否等于 TouchEvent.MOVE。

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

响应轻按屏幕的用户

1. 导入以下类：
   - 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. 创建扩展 Manager 类、Screen 类、Field 类或其中一个 Field 子类的类。

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

3. 在 touchEvent(TouchEvent message) 方法的实施中，调用 TouchEvent.getEvent()。
4. 检查 TouchEvent.getEvent() 返回的值是否等于 TouchEvent.DOWN。

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

响应滚动操作

1. 导入以下类：
   - 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. 创建扩展 Manager 类的类。

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

3. 在 touchEvent(TouchEvent message) 方法的实施中，检查 TouchEvent.getEvent() 返回的值是否等于 TouchEvent.MOVE。

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

响应同时触摸屏幕中的两个位置的用户

1. 导入以下类：
   - 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. 创建扩展 Manager 类、Screen 类、Field 类或其中一个 Field 子类的类。

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

3. 在 touchEvent(TouchEvent message) 方法的实施中，检查以下方法调用是否返回大于零的值：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;
    }
```

多点触控姿势

带触摸屏的 BlackBerry® 设备支持多点触控姿势。 多点触控姿势通常将放大和缩小屏幕上的对象。

在触摸屏上检测到两个触摸点时，多点触控就将开始。 多点触控的焦点将定义为两个初始触摸点的中点。

- 检测到两个触摸点时，将发生 `TouchGesture.PINCH_BEGIN` 事件。 多点触控姿势的焦点的坐标点将存储为 `TouchEvent.getX(1)` 和 `TouchEvent.getY(1)`。 通过调用 `TouchGesture.getPinchMagnitude()`，可访问初始缩放值。
- 在用户移动一个或两个触摸点时，将发生一系列 `TouchGesture.PINCH_UPDATE` 事件。 通过调用 `TouchGesture.getPinchMagnitude()`，可在多点触控期间访问缩放值。
- 用户完成此姿势时，将发生 `TouchGesture.PINCH_END` 事件。 通过调用 `TouchGesture.getPinchMagnitude()`，可访问最终缩放值。

代码示例：检索有关多点触控姿势的信息

此示例演示如何在屏幕的 `touchEvent()` 方法中处理多点触控姿势。

```
protected boolean touchEvent(TouchEvent message)
{
    switch(message.getEvent())
    {
        case TouchEvent.GESTURE:
            TouchGesture gesture = message.getGesture();
            switch(gesture.getEvent())
            {
              case TouchGesture.PINCH_END:
                Dialog.alert("Focal point: " + message.getX(1)
                +             ", " + message.getY(1)
                +             "\nFinal zoom value: " + gesture.getPinchMagnitude());
                return true;
            }
    }
    return false;
}
```

启用多点触控姿势

默认情况下，未在 BlackBerry® 设备应用程序的屏幕上启用多点触控姿势。 您必须设置屏幕属性 ，才能在屏幕上启用多点触控姿势生成。

1. 导入所需的类和接口。

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

2. 在屏幕对象中，创建要使用触摸屏属性填充的 InputSettings 对象。

```
InputSettings ts = TouchscreenSettings.createEmptySet();
```

3. 将 TouchscreenSettings.DETECT_PINCH 属性设置为 1。

```
ts.set(TouchscreenSettings.DETECT_PINCH, 1);
```

4. 调用 Screen.addInputSettings() 以将输入设置添加至屏幕。

```
addInputSettings(ts);
```

您可随时更改 TouchscreenSettings.DETECT_PINCH 属性的值，而不是仅在创建屏幕时。

代码示例

此示例演示如何在屏幕构造器中启用多点触控姿势。

```
public MyScreen()
{
    setTitle("Sample Screen");
    InputSettings ts = TouchscreenSettings.createEmptySet();
    ts.set(TouchscreenSettings.DETECT_PINCH, 1);
    addInputSettings(ts);
    ...
}
```

触控板的滑动姿势

与带触摸屏的 BlackBerry® 设备类似，带触控板的设备也支持 BlackBerry 设备用户在触控板上执行的滑动姿势。

用户在触控板上滑动时，将生成事件类型为 TouchGesture.NAVIGATION_SWIPE 的 TouchEvent 对象。通过使用以下方法，可检索有关触控板滑动的信息：

- TouchGesture.getSwipeAngle()
- TouchGesture.getSwipeDirection()
- TouchGesture.getSwipeContentAngle()
- TouchGesture.getSwipeContentDirection()
- TouchGesture.getSwipeMagnitude()

代码示例：检索有关触控板滑动的信息

此示例演示如何在屏幕的 touchEvent() 方法中处理触控板滑动。

```
protected boolean touchEvent(TouchEvent message)
{
   switch(message.getEvent())
   {
      case TouchEvent.GESTURE:
         TouchGesture gesture = message.getGesture();
         switch(gesture.getEvent())
         {
           case TouchGesture.NAVIGATION_SWIPE:
             Dialog.alert("Swipe direction: " + gesture.getSwipeDirection()
             +            ", "
             +            "\nMagnitude: " + gesture.getSwipeMagnitude());
             return true;
         }
   }
   return false;
}
```

启用用户在触控板上执行的滑动姿势

默认情况下，未在屏幕上启用 BlackBerry® 设备用户在触控板上执行的滑动姿势。 您必须设置导航属性，才能在屏幕上启用触控板滑动姿势检测。

1. 导入所需的类和接口。

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

2. 在屏幕对象中，创建要使用导航属性填充的 InputSettings 对象。

```
InputSettings nd = NavigationDeviceSettings.createEmptySet();
```

3. 将 NavigationDeviceSettings.DETECT_SWIPE 属性设置为 1。

```
nd.set(NavigationDeviceSettings.DETECT_SWIPE, 1);
```

4. 调用 Screen.addInputSettings() 以将输入设置添加至屏幕。

```
addInputSettings(nd);
```

您可随时更改 NavigationDeviceSettings.DETECT_SWIPE 属性的值，而不是仅在创建屏幕时。

代码示例

此示例演示如何在屏幕构造器中启用触控板滑动姿势。

```
public MyScreen()
{
   setTitle("Sample Screen");
   InputSettings nd = NavigationDeviceSettings.createEmptySet();
   nd.set(NavigationDeviceSettings.DETECT_SWIPE, 1);
```

```
    addInputSettings(nd);
    ...
}
```

事件插入

通过使用 EventInjector 类及其内部类，可以编程方式生成 UI 事件。 在运行 BlackBerry® Device Software 5.0 版或更高版本且具有触摸屏的 BlackBerry® 设备上，可插入触摸事件，如滑动和点按。 您可以使用其中一个 EventInjector 内部类进行事件建模，并可使用 invokeEvent() 方法插入事件。 系统会将该事件发送至当前选定的应用程序，并准备接受输入。

您可以使用事件插入实现测试自动化。 您也可以使用事件插入提供外围设备与 BlackBerry 设备交互的方式。您也可以将其用于可访问的应用程序，如语音到文本的转换器。 有关演示事件插入的示例应用程序，请访问 www.blackberry.com/go/toucheventinjectorsampleapp 以下载“触摸事件插入器”示例应用程序。 有关该示例应用程序的详细信息，请访问 www.blackberry.com/go/docs/developers 以阅读 *“触摸事件插入器示例应用程序概述”*。

命令框架 API

6

您可使用 net.rim.device.api.command 数据包中的命令框架 API 定义可在您的应用程序或 BlackBerry® 设备上的其他应用程序的不同位置使用的功能。

命令框架 API 包含命令处理程序、命令元数据和命令。

组件	说明
命令处理程序	您可使用 CommandHandler 类定义可用于您的应用程序或设备上的其他应用程序的功能。 您可创建扩展抽象 CommandHandler 类的类，并在类的 execute() 方法中定义功能。 该功能称为命令。
命令元数据	您可使用 CommandMetadata 类定义描述命令的元数据。 每个命令的元数据都封装在 CommandMetadata 对象中。 唯一需要的命令元数据部分是命令 ID，这将在构造 CommandMetadata 对象并将其存储在对象的 COMMAND_ID 字段中时提供。 CommandMetadata 提供其他可用于描述命令的字段（如 RESOURCE_BUNDLE_NAME），您可为这些字段分配命令使用的资源捆绑包的名称。 您还可以定义特定命令的特定元数据项。
命令	您可使用 Command 类执行命令。 您可将 Command 实例视为扩展 CommandHandler 类的实例的代理。 调用 Command.execute() 时，调用将委派给关联 CommandHandler 实例，从而传递当前上下文并透明传递要执行的关联命令元数据的只读版本。

命令框架 API 提供两个可用于存储命令的注册表：

- 通过使用 LocalCommandRegistrarConnection 类，可使用本地注册表存储用于您的应用程序的命令。
- 通过使用 RemoteCommandRegistrarConnection 类，可使用全局注册表存储用于其他应用程序的命令。

BlackBerry® Java® SDK 包括“命令框架演示”和“命令框架演示远程应用程序”示例应用程序。 这些示例应用程序演示如何在单一应用程序中定义并使用命令，以及如何在一个应用程序中定义命令并在其他应用程序中使用。

将命令用于一个 UI 组件

您可定义要用于 BlackBerry® 设备应用程序中一个 UI 组件的命令。 通过此方法，就无需定义命令的元数据。 BlackBerry 设备应用程序中支持命令的核心 UI 组件包括菜单项、按钮、弹出菜单项和工具栏。

1. 导入所需的类和接口。

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

2. 通过创建扩展抽象 CommandHandler 类的类，创建命令处理程序。 在 execute() 中，定义要与 UI 组件关联的功能。 要指定命令对于指定上下文对象是否可执行，可在命令处理程序中实施 canExecute() 方法。 如果未实施 canExecute()，则命令始终可执行。

```
// this command is always executable
class DialogCommandHandler extends CommandHandler
{
   public void execute(ReadOnlyCommandMetadata metadata, Object context)
   {
      Dialog.alert("Executing command for " + context.toString());
   }
}
```

3. 创建 UI 组件。

```
MenuItem myItem = new MenuItem(new StringProvider("My Menu Item"),
   0x230000, 0);
```

4. 对于菜单项或按钮，可根据需要使用 UI 组件的 setCommandContext() 方法设置命令上下文。 对于某些命令，可能需要上下文才能确定命令应该执行的操作。 如果未设置上下文，菜单项对象或按钮对象就是上下文。

```
myItem.setCommandContext(new Object()
{
   public String toString()
   {
      return "My MenuItem Object";
   }
});
```

5. 调用 setCommand() 以指定 UI 组件的命令并提供命令处理程序作为方法的自变量。

```
myItem.setCommand(new Command(new DialogCommandHandler()));
```

6. 将组件添加至屏幕。

```
addMenuItem(myItem);
```

BlackBerry 设备用户在 UI 组件上执行操作时（例如，用户单击菜单项时），将执行此命令。

代码示例

有关使用此方法定义和使用命令的示例，请参阅 MenuItem 和 ButtonField 的 API 参考。

将命令用于一个或多个应用程序

您可将命令存储在本地注册表中以用于您的应用程序，或将命令存储在全局注册表中以用于 BlackBerry® 设备上的任何应用程序。

1. 导入所需的类和接口。

```
import net.rim.device.api.command.*;
import net.rim.device.api.command.registrar.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.container.*;
```

2. 通过创建扩展抽象 CommandHandler 类的类,创建命令处理程序。 在 execute() 中,定义要提供的功能。 要指定命令对于指定上下文对象是否可执行,可根据需要在命令处理程序中实施 canExecute()。如果未实施 canExecute(),则命令始终可执行。

```
private static class ViewContactCommand extends CommandHandler
{
   public void execute(ReadOnlyCommandMetadata metadata, Object context)
   {
      // Invoke the Contacts application to view a contact here
   }
   public boolean canExecute(ReadOnlyCommandMetadata metadata, Object context)
   {
      // Return true if the command can be executed
      // with the passed-in context object.
   }
}
```

3. 定义命令的元数据。

```
String id = "com.example.contacts.view.emailaddr";
CommandMetadata metadata = new CommandMeta(id);
```

4. 如果使用命令的应用程序必须创建命令的实例,请将命令的类名称存储在元数据中。

```
metadata.setClassname(ViewContactCommand.class.getName());
```

注: 只有将命令存储在本地注册表中,应用程序才能动态创建第三方命令的实例。

5. (可选)定义命令的命令类别和上下文类别。 使用命令的应用程序将使用类别在注册表中查找命令并确定命令是否可执行。

类别类型	说明
命令	指定要用于命令的类别类型。 使用命令的应用程序可在注册表中查询属于指定类别的命令。
上下文	指定适于在其中执行命令的上下文。 您可使用此类别告诉使用命令的应用程序可将哪些类型的上下文对象传递至 CommandHandler.Execute()。

```
String[] myCommandCategories = {"app.contacts", "action.view"};
metadata.setCommandCategories(new CategoryCollection(myCommandCategories));
String[] myContextCategories = {emailaddr};
metadata.setContextCategories(new CategoryCollection(myContextCategories));
```

6. 注册命令。
   a. 如果要让命令仅可用于您的应用程序,请使用 LocalCommandRegistrarConnection 对象。 本地注册表仅在当前进程中提供。

```
LocalCommandRegistrarConnection connection =
   new LocalCommandRegistrarConnection();
connection.registerCommand(new ViewContactCommand(), metadata);
```

注：要让应用程序动态创建命令的实例，必须调用 `registerCommand(CommandMetadata)` 以仅在元数据中注册命令。 否则，将使用注册的 `CommandHandler`。

b. 如果要将命令用于任何应用程序，请使用 `RemoteCommandRegistrarConnection` 对象。 全局注册表可用于所有在设备上运行的进程。

```
RemoteCommandRegistrarConnection connection =
   new RemoteCommandRegistrarConnection();
connection.registerCommand(new ViewContactCommand(), metadata);
```

7. 从应用程序中的其他位置（对于本地命令）或从设备上的任何应用程序（对于全局命令）检索并使用命令。

a. 创建 `CommandRequest` 对象并使用要从注册表检索的命令的相关信息填充它。

```
CommandRequest request = new CommandRequest();
String[] myCommandCategories = {"app.contacts", "action.view"};
request.setCommandCategories(new CategoryCollection(myCommandCategories));
...
```

b. 从注册表检索命令。

```
// using the local registry
LocalCommandRegistrarConnection connection =
   new LocalCommandRegistrarConnection();
Command command = connection.getCommand(request);
// using the global registry
RemoteCommandRegistrarConnection connection =
   new RemoteCommandRegistrarConnection();
Command command = connection.getCommand(request);
```

c. 使用命令。 以下示例将根据上下文执行命令。

```
command.execute(context); // context is the context object
```

代码示例

BlackBerry® Java® SDK 包括两个使用命令框架 API 的示例应用程序：

- “命令框架演示远程应用程序”演示如何创建命令并将其存储在全局注册表中以用于其他应用程序。
- “命令框架演示”演示如何创建命令、将其存储在本地注册表中并将其作为菜单项添加至字段的弹出菜单中，具体取决于字段的内容。 此示例应用程序还将执行“命令框架演示远程应用程序”存储在全局注册表中的命令。

排列 UI 组件

7

通过使用 BlackBerry® API 布局管理器，可在应用程序屏幕上排列 UI 组件。 以下类扩展了 net.rim.device.apu.ui 数据包中提供的 Manager 类，并提供 UI 组件在应用程序屏幕上的预定义布局。

布局管理器	说明
FlowFieldManager	此布局管理器先垂直再水平排列 UI 组件，具体取决于屏幕大小。 第一个 UI 组件放在屏幕左上角，而后续组件水平放在第一个组件的右侧，直至达到屏幕宽度。 一旦不能再将 UI 组件放在第一行，下个 UI 组件将放在第一行组件下高度等于上行的最高组件的行中。 您可应用垂直样式位（例如，Field.FIELD_TOP、Field.FIELD_BOTTOM 或 Field.FIELD_VCENTER），以垂直对齐行内的 UI 组件。
HorizontalFieldManager	此布局管理器在单个水平行中排列 UI 组件，其中此行从屏幕左侧开始并在屏幕右侧结束。 因为此布局管理器排列 UI 组件，所以无法将水平样式位应用于 UI 组件（例如，Field.FIELD_LEFT、Field.FIELD_HCENTER 或 Field.FIELD_RIGHT）。 您可以应用垂直样式位（例如，Field.FIELD_TOP、Field.FIELD_BOTTOM 或 Field.FIELD_VCENTER）。 如果 UI 组件不适合屏幕的可用宽度，应使用 Manager.HORIZONTAL_SCROLL 样式位启用水平滚动。 否则，屏幕将在可用屏幕宽度内显示尽可能多的 UI 组件，而不显示其余组件。 这些 UI 组件存在，但不可见。 这种情况可为用户创建意外滚动行为。
VerticalFieldManager	此布局管理器在单个垂直行中排列 UI 组件，其中此列从屏幕顶部开始并在屏幕底部结束。 因为此布局管理器旨在垂直排列项，所以无法将垂直样式位应用于 UI 组件（例如，Field.FIELD_TOP、Field.FIELD_BOTTOM 或 Field.FIELD_VCENTER）。 您可以应用水平样式位（例如，Field.FIELD_LEFT、Field.FIELD_HCENTER 或 Field.FIELD_RIGHT）。

您可以使用其它布局管理器在应用程序中排列 UI 组件。 例如，可使用 GridFieldManager 布局管理器将 UI 组件放在屏幕上的行和列中，以创建网格。 您可以使用 EyelidFieldManager 布局管理器在一对管理器上显示 UI 组件，而这对管理器分别临时显示在屏幕顶部和底部。

排列 UI 组件

1. 导入所需的类和接口。

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

2. 创建 HorizontalFieldManager 的实例。

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

3. 调用 `add()` 以将 `HorizontalFieldManager` 添加至屏幕。

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

4. 创建 `ButtonField` 的实例。

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

5. 将 `ButtonField` 添加至 `HorizontalFieldManager`。

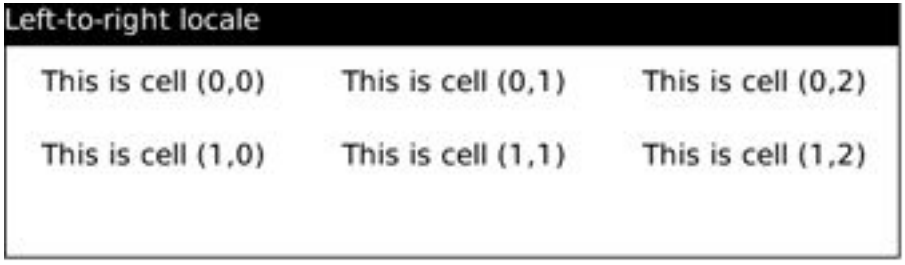
```
_fieldManagerBottom.add(mySubmitButton);
```

创建网格布局

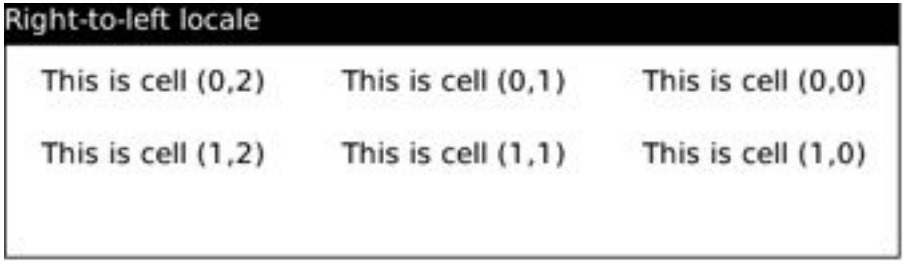
注： 有关在低于 5.0 版的 BlackBerry® Java® Development Environment 创建网格布局的信息，请访问 http://www.blackberry.com/knowledgecenterpublic 以阅读 DB-00783。

通过使用 `GridFieldManager` 类，可将字段放在屏幕上的行和列中，以创建网格。 创建网格时，可指定行数和列数。 创建网格后，无法更改其包含的行数和列数。

网格为零索引，因此第一个单元格位于行 0、列 0。 在具有从左到右的文本方向的区域设置中，第一个单元格位于网格的左上角。



在具有从右到左的文本方向的区域设置中，第一个单元格位于网格的右上角。



您可以按顺序（具有从左到右的文本方向的区域设置中的从左到右和从上到下；具有从右到左的文本方向的区域设置中的从右到左和从上到下）或通过在网格中指定行和列将字段添加至网格。 您可以删除字段、插入字段、指定列和行之间的间距以及检索网格的属性。

网格没有定义的标题行或标题列。 通过更改网格第一行或第一列中的字段的外观，可模拟标题的外观。

如果网格的宽度和高度超过屏幕的可见区域，网格可水平或垂直滚动。

通过调用 GridFieldManager.setColumnProperty()，可指定列宽度；通过调用 GridFieldManager.setRowProperty()，可指定行高度。 调用这些方法时，必须指定 GridFieldManager 属性。

属性	说明
FIXED_SIZE	宽度或高度是以像素表示的固定大小
PREFERRED_SIZE	宽度或高度是基于列或行中字段的最大首选大小的首选大小（PREFERRED_SIZE 是默认属性）
PREFERRED_SIZE_WITH_MAXIMUM	宽度或高度是高达最大大小的首选大小
AUTO_SIZE	宽度或高度基于可用屏幕空间

创建网格布局

1. 导入所需的类和接口。

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

2. 通过扩展 UiApplication 类，创建应用程序框架。 在 main() 中，创建新类的实例并调用 enterEventDispatcher()，以启用应用程序来接收事件。 在应用程序构造器中，调用 pushScreen() 以显示应用程序的自定义屏幕。 在此示例中，步骤 3 中介绍的 GridScreen 类表示自定义屏幕。

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

3. 通过扩展 MainScreen 类创建自定义屏幕的框架。

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

4. 在屏幕构造器中，调用 setTitle() 以设置要在屏幕的标题部分中显示的文本。

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

5. 在屏幕构造器中，创建 GridFieldManager 类的实例。 指定行数、列数和网格样式（使用继承自 net.rim.device.api.ui.Manager 的样式）。 将样式指定为 0，则使用默认样式。

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

6. 在屏幕构造器中，调用 GridFieldManager.add() 以将字段添加至网格。

```
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. 在屏幕构造器中，调用 GridFieldManager set() 方法以指定网格的属性。

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

8. 在屏幕构造器中，调用 Screen.add() 以将网格添加至屏幕。

```
add(grid);
```

完成之后：

创建网格后，可对其进行更改。 例如，可添加字段、删除字段或更改网格的属性。

代码示例：创建网格布局

```
/*
 * 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
    }
}
```

在一对临时管理器上显示字段

您可以使用 `EyelidFieldManager` 类在一对管理器上显示字段，而这对管理器分别临时显示在屏幕顶部和底部。

默认情况下，在 BlackBerry® 设备用户移动轨迹球或在带触摸屏的设备上发生触摸事件时，将显示这些字段。这些字段将经过一段空闲时间段（默认情况下为 1.2 秒）后将消失。 您可覆盖这些默认属性。

字段的数量和大小无限制。 如果管理器包含的字段多于屏幕可容纳的字段，顶部管理器和底部管理器将与前台的顶部管理器重叠。

在屏幕顶部和底部临时显示 ButtonField 和 LabelField

1. 导入所需的类和接口。

```
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. 通过扩展 UiApplication 类,创建应用程序框架。 在 main() 中,创建新类的实例并调用 enterEventDispatcher(),以启用应用程序来接收事件。 在应用程序构造器中,调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 EyelidFieldManagerDemoScreen 类表示自定义屏幕。

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

3. 通过扩展 MainScreen 类创建自定义屏幕的框架。

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

4. 在屏幕构造器中,调用 setTitle() 以指定要在屏幕的标题部分中显示的文本。

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

5. 在屏幕构造器中,创建 EyelidFieldManager 类的实例。

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

6. 在屏幕构造器中,调用 EyelidFieldManager.addTop() 以将 LabelField 对象添加至 EyelidFieldManager. 的顶部管理器。

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

7. 在屏幕构造器中,创建 HorizontalFieldManager 对象。 调用 HorizontalFieldManager.add() 以将按钮添加至 HorizontalFieldManager。 调用 EyelidFieldManager.addBottom() 将 HorizontalFieldManager 添加到 EyelidFieldManager 的底部管理器。

```
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. 在屏幕构造器中,调用 EyelidFieldManager.setEyelidDisplayTime() 以指定两个管理器消失之前必须经过的空闲时间(以毫秒为单位)

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

9. 在屏幕构造器中，调用 add() 以将 EyelidFieldManager 添加至屏幕。

```
add(manager);
```

代码示例：在屏幕顶部和底部临时显示 ButtonField 和 LabelField

```
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);
     }
}
```

将字段显示在屏幕上的绝对位置

您可以使用 AbsoluteFieldManager 类根据 x-y 坐标指定屏幕上某个字段的绝对位置，而不是相对于屏幕上其他字段的位置。

您可以使用 AbsoluteFieldManager 类控制字段放置和重叠屏幕上的字段。

将标签显示在屏幕上的绝对位置

1. 导入所需的类和接口。

```
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.*;
```

2. 通过扩展 UiApplication 类,创建应用程序框架。 在 main() 中,创建新类的实例并调用 enterEventDispatcher(),以启用应用程序来接收事件。 在应用程序构造器中,调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 AbsoluteFieldManagerDemoScreen 类表示自定义屏幕。

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

3. 通过扩展 MainScreen 类创建自定义屏幕的框架。 调用 setTitle() 以指定要在屏幕的标题部分中显示的文本。

```
class AbsoluteFieldManagerDemoScreen extends MainScreen
{
     public AbsoluteFieldManagerDemoScreen()
     {
          setTitle("AbsoluteFieldManager Demo");
     }
}
```

4. 在屏幕构造器中,创建 AbsoluteFieldManager 类的实例。

```
AbsoluteFieldManager manager = new AbsoluteFieldManager();
```

5. 在屏幕构造器中,创建并初始换变量,以存储与屏幕的垂直中心对应的 y 坐标。

```
int halfwayDown = Display.getHeight() / 2;
```

6. 在屏幕构造器中,调用 AbsoluteFieldManager.add() 以在屏幕的垂直中心使用 2 的 x 坐标添加 LabelField 对象。

```
manager.add(new LabelField("Hello world"), 2, halfwayDown);
```

7. 在屏幕构造器中,调用 add() 以将 AbsoluteFieldManager 添加至屏幕。

```
add(manager);
```

代码示例：将标签显示在屏幕上的绝对位置

```
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.*;
public class AbsoluteFieldManagerDemo extends UiApplication
{
        public static void main(String[] args)
        {
             AbsoluteFieldManagerDemo app = new AbsoluteFieldManagerDemo();
             app.enterEventDispatcher();
        }
        public AbsoluteFieldManagerDemo()
        {
             pushScreen(new AbsoluteFieldManagerDemoScreen());
        }
}
class AbsoluteFieldManagerDemoScreen extends MainScreen
{
        public AbsoluteFieldManagerDemoScreen()
        {
             setTitle("AbsoluteFieldManager Demo");
             AbsoluteFieldManager manager = new AbsoluteFieldManager();
             int halfwayDown = Display.getHeight() / 2;
             manager.add(new LabelField("Hello world"), 2, halfwayDown);
             add(manager);
       }
}
```

UI 组件

8

将 UI 组件添加至屏幕

1. 导入以下类：
   - net.rim.device.api.ui.component.CheckboxField
   - net.rim.device.api.ui.container.MainScreen
2. 创建 UI 组件的实例。

```
CheckboxField myCheckbox = new CheckboxField("First checkbox", true);
```

3. 将 UI 组件添加至 Screen 类的扩展。

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

将字段与文本行对齐

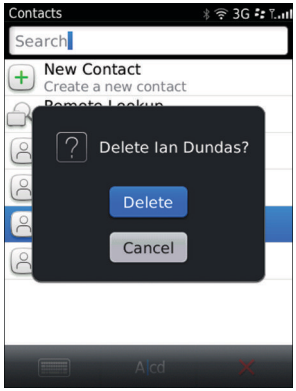
通过使用标记 Field.FIELD_LEADING，可创建可将 Field 对象与文本行自然开头对齐的应用程序。 例如，如果使用对齐标记 Field.FIELD_LEADING 创建 Field，并将 Field 添加至 VerticalFieldManager，则在应用程序开始使用英语或中文区域设置时，Field 将对齐至屏幕左侧。 如果应用程序开始使用阿拉伯语或希伯莱语区域设置，Field 将对齐至屏幕右侧。

按钮

使用按钮以允许用户从对话框中执行操作。 菜单通常包括与屏幕相关联的操作。

用户可以使用按钮执行以下操作：

操作	仅带触控板的 BlackBerry 设备	带触摸屏和触控板的 BlackBerry 设备
高亮显示一个按钮。	在触控板上移动手指。	• 轻轻触按按钮。 • 在触控板上移动手指。
执行操作。	单击触控板或者按输入键。	• 点按项目。 • 单击触控板。 • 按输入键。



最佳实践：使用按钮

- 避免在应用程序屏幕上使用按钮。 要提供与屏幕关联的操作，请使用应用程序菜单。 在带触控板的 BlackBerry® 设备上，用户可以立即使用该菜单，不管光标在屏幕上的位置如何。 按钮是静态的，需要用户高亮显示按钮才能执行关联操作。 如果您使用按钮，也请在应用程序菜单中包括这些操作的菜单项。在带触摸屏的 BlackBerry 设备上，可将按钮用于执行重要操作。
- 在选项中使用复选框（如打开或关闭一项功能）
- 使用 `ButtonField` 类创建按钮。
- 对于默认按钮，请使用用户最可能单击的按钮。 避免将与破坏性操作相关联的按钮用作默认按钮。

标签准则

- 使用简洁明了的标签。
- 尽量使用名称为单个词的标签。 按钮的大小会根据标签的长度而变化。 如果标签太长，则省略号（...）表示文本被截断。
- 使用描述相关操作的动词作为标签名称（例如，“取消”、“删除”、“放弃”或“保存”） 如有必要，请在屏幕上的其他位置（例如，在一则应用程序消息中）包括更多的描述性文本。
- 避免使用“是”和“否”标签。
- 避免在标签中使用标点符号。 在按钮标签中使用省略号表示用户单击该按钮后还须执行另一个操作。

创建按钮

1. 导入 `net.rim.device.api.ui.component.ButtonField` 类。
2. 使用样式参数创建 `ButtonField` 的实例。

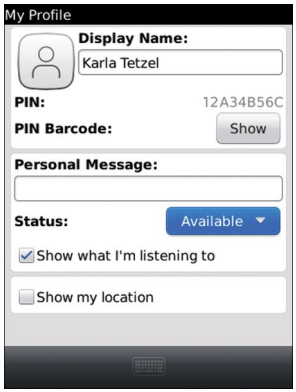
```
ButtonField mySubmitButton = new ButtonField("Submit");
```

复选框

对于二元选项，请使用复选框，便于用户理解。 例如，将复选框用于可打开和关闭的选项。

用户可以使用复选框执行以下操作：

操作	仅带触控板的 BlackBerry 设备	带触摸屏和触控板的 BlackBerry 设备
选择一个复选框。	按空格键或单击触控板。	• 点按项目。 • 按空格键。 • 单击触控板。



最佳实践：使用复选框

- 当用户可以选择多个选项时，请使用复选框。
- 使用 `CheckboxField` 类创建复选框。
- 当用户选择复选框时，不要启动操作。 例如，不要打开新屏幕。
- 将复选框垂直对齐。
- 按照逻辑将复选框分组和排序（例如，将相关选项分成一组或者首先包括最常用的选项） 避免将复选框按照字母顺序进行排序；字母顺序仅针对特定语言而言。

标签准则

- 使用简洁明了的标签。 请确定标签能清晰地描述当用户选择复选框时所发生的内容。
- 尽量使用正值标签。 例如，如果用户可以选择打开或关闭某项功能，则请在标签中使用“打开”而不要使用“关闭”。
- 将标签放置在复选框右侧。 在“选项”屏幕上，将标签放在复选框的左侧。
- 请遵循句子类的首字母大写规则。
- 请使用冒号（:）来分隔复选框标签。

创建复选框

1. 导入所需的类和接口。

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

2. 通过扩展 UiApplication 类,创建应用程序框架。 在 main() 中,创建新类的实例并调用 enterEventDispatcher(),以启用应用程序来接收事件。 在应用程序构造器中,调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 MyUiScreen 类表示自定义屏幕。

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

3. 通过扩展 MainScreen 类来创建应用程序的自定义屏幕。 在屏幕构造器中,调用 setTitle() 以指定屏幕的标题。

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

4. 在屏幕构造器中,使用 CheckboxField 类创建复选框。 在 CheckboxField 构造器中,指定复选框的标签,并使用布尔值指明该复选框是否默认选中(例如, true 指明默认情况下选中该复选框)。 调用 add() 以将复选框添加至屏幕。

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

5. 要更改复选框的默认行为,请使用从 Field 类继承的样式。 例如,要创建用户无法更改的复选框,请使用 READONLY 样式。

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

6. 要覆盖提示用户在应用程序关闭前保存更改的默认功能,请在 MainScreen 类的扩展中覆盖 MainScreen.onSavePrompt() 方法。 在以下代码示例中,返回值 true 指明应用程序在关闭前不提示用户。

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

代码示例：创建复选框

```
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;
    }
}
```

创建位图

1. 导入 net.rim.device.api.ui.component.BitmapField 类。
2. 创建 BitmapField 的实例。

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

创建自定义字段

您只能将自定义上下文菜单项和自定义布局添加至自定义字段。

1. 导入以下类：
   - 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. 导入 net.rim.device.api.ui.DrawStyle 接口。
3. 扩展 Field 类或其子类之一，其中实施 DrawStyle 接口以指定自定义字段的特征并打开绘图样式。

```
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. 实施各个构造器以定义自定义按钮的标签、形状和样式。

```
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. 实施 layout() 以指定字段数据的排列。 在 layout() 中执行最复杂的计算，而不在 paint() 中执行。 字段的管理器将调用 layout()，以确定字段如何在可用空间中排列其内容。 调用 Math.min() 以返回指定宽度和高度的较小者，以及字段的首选宽度和高度。 调用 Field.setExtent(int,int) 以设置字段的必需尺寸。

```
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. 实施 getPreferredWidth()，其中使用字段标签的相对尺寸确保标签不会超过组件的尺寸。 使用 switch 块根据自定义字段的形状确定首选宽度。 对于每种类型的形状，使用 if 语句比较尺寸并确定自定义字段的首选宽度。

```
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. 实施 getPreferredHeight(),其中使用字段标签的相对尺寸确定首选高度。 使用 switch 块根据自定义字段的形状确定首选高度。 对于每种类型的形状,使用 if 语句比较尺寸并确定自定义字段的首选高度。

```
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. 实施 paint()。 字段的管理器将调用 paint(),以在某个字段区域被标记为无效时重新绘制字段。 使用 switch 块根据自定义字段的形状重画自定义字段。 对于具有三角形或八角形形状的字段,使用字段的宽度计算线条起点和终点的水平位置和垂直位置。 调用 graphics.drawLine() 并使用起点和终点绘制定义自定义字段的线条。 对于具有矩形形状的字段,调用 graphics.drawRect() 并使用字段的宽度和高度绘制自定义字段。 调用 graphics.drawText() 并使用字段的宽度将文本字符串绘制到某个字段区域

```
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. 实施 `Field set()` 和 `get()` 方法。 实施 `Field.getLabel()`、`Field.getShape()`、`Field.setLabel(String label)` 和 `Field.setShape(int shape)` 方法,以返回自定义字段的实例变量。

```
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;
}
```

创建一个字段显示 web 内容

将 HTML 内容显示在浏览器字段中

1. 导入所需的类和接口。

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

2. 通过扩展 UiApplication 类,创建应用程序框架。 在 main() 中,创建新类的实例并调用 enterEventDispatcher(),以启用应用程序来接收事件。 在应用程序构造器中,调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 BrowserFieldDemoScreen 类表示自定义屏幕。

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

3. 通过扩展 MainScreen 类来创建自定义屏幕。

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

4. 在屏幕构造器中,创建 BrowserField 类的实例。

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

5. 在屏幕构造器中,调用 add() 以将 BrowserField 对象添加至屏幕。

```
add(myBrowserField);
```

6. 在屏幕构造器中,调用 BrowserField.displayContent(),以指定并显示 HTML 内容。

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

代码示例：将 HTML 内容显示在浏览器字段中

```
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");
    }
}
```

将来自网页的 HTML 内容显示在浏览器字段中

1. 导入所需的类和接口。

```
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. 通过扩展 UiApplication 类,创建应用程序框架。 在 main() 中,创建新类的实例并调用 enterEventDispatcher(),以启用应用程序来接收事件。 在应用程序构造器中,调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 BrowserFieldDemoScreen 类表示自定义屏幕。

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

3. 通过扩展 MainScreen 类创建自定义屏幕的框架。

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

4. 在屏幕构造器中,创建 BrowserField 类的实例。

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

5. 在屏幕构造器中，调用 add() 以将 BrowserField 对象添加至屏幕。

```
add(myBrowserField);
```

6. 在屏幕构造器中，调用 BrowserField.requestContent()，以指定 HTML 内容的位置并显示它。

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

代码示例：将来自网页的 HTML 内容显示在浏览器字段中

```
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");
    }
}
```

将来自资源的 HTML 内容显示在应用程序中

1. 导入所需的类和接口。

```
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. 通过扩展 UiApplication 类，创建应用程序框架。 在 main() 中，创建新类的实例并调用 enterEventDispatcher()，以启用应用程序来接收事件。 在应用程序构造器中，调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 BrowserFieldDemoScreen 类表示自定义屏幕。

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

```
            BrowserFieldDemo app = new BrowserFieldDemo();
            app.enterEventDispatcher();
        }
        public BrowserFieldDemo()
        {
            pushScreen(new BrowserFieldDemoScreen());
        }
}
```

3. 通过扩展 MainScreen 类来创建自定义屏幕。

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

4. 在屏幕构造器中,创建 BrowserField 类的实例。

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

5. 在屏幕构造器中,调用 add() 以将 BrowserField 对象添加至屏幕。

```
add(myBrowserField);
```

6. 在屏幕构造器中,调用 BrowserField.requestContent(),以在应用程序中指定资源的位置并显示 HTML 内容。

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

注: BrowserField 类不使用文件夹结构访问资源。 BrowserField 类显示找到的与指定文件名匹配的第一项资源。

代码示例:将来自资源的 HTML 内容显示在应用程序中

```
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");
    }
}
```

配置浏览器字段

1. 导入所需的类和接口。

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

2. 通过扩展 UiApplication 类，创建应用程序框架。 在 main() 中，创建新类的实例并调用 enterEventDispatcher()，以启用应用程序来接收事件。 在应用程序构造器中，调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 BrowserFieldDemoScreen 类表示自定义屏幕。

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

3. 通过扩展 MainScreen 类创建自定义屏幕的框架。

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

4. 在屏幕构造器中，创建 BrowserFieldConfig 类的实例。

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

5. 在屏幕构造器中，调用 BrowserFieldConfig.setProperty() 以指定 BrowserField 的属性。 setProperty() 中的第一个参数指定该属性，而第二个参数指定该属性的值。 例如，以下代码示例指定 BrowserField 对象的 NAVIGATION_MODE 属性：

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

6. 在屏幕构造器中，创建使用定义的配置的 BrowserField 类。

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

代码示例：配置浏览器字段

```
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);
    }
}
```

将表单数据发送至浏览器字段中的 Web 地址

1. 导入所需的类和接口。

```
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. 通过扩展 `UiApplication` 类，创建应用程序框架。 在 `main()` 中，创建新类的实例并调用 `enterEventDispatcher()`，以启用应用程序来接收事件。 在应用程序构造器中，调用 `pushScreen()` 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 `BrowserFieldDemoScreen` 类表示自定义屏幕。

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

```
            pushScreen(new BrowserFieldDemoScreen());
        }
}
```

3. 通过扩展 MainScreen 类来创建自定义屏幕。

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

4. 在屏幕构造器中,创建 BrowserField 类的实例。

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

5. 在屏幕构造器中,调用 add() 以将 BrowserField 对象添加至屏幕。

```
add(browserField);
```

6. 在屏幕构造器中,创建 String 对象,以包含要将表单数据发送至的网页的基础 Web 地址。

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

7. 在屏幕构造器中,创建指定应用程序发送至网页的表单数据的 String。

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

8. 在屏幕构造器中,创建 Hashtable 对象,以存储表单数据的题头信息。

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

9. 在屏幕构造器中,调用 Hashtable.put() 以指定表单数据的题头信息。

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

10. 在屏幕构造器中,调用 BrowserField.requestContent(),以将表单数据发送至网页并显示网页。

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

代码示例:将表单数据发送至浏览器字段中的 Web 地址

```
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();
```

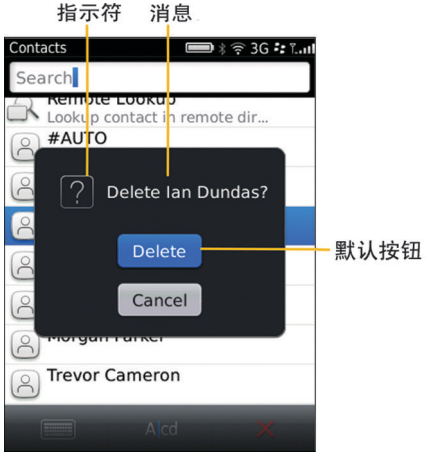
```
        app.enterEventDispatcher();
    }
    public BrowserFieldDemo()
    {
        pushScreen(new BrowserFieldDemoScreen());
    }
}
class BrowserFieldDemoScreen extends MainScreen
{
    public BrowserFieldDemoScreen()
    {
        BrowserField browserField = new BrowserField();}
        add(browserField);
        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));
    }
}
```

对话框

使用对话框执行以下操作：

- 提示用户完成用户启动的任务所需要的信息。
- 通知用户紧急信息或重要操作的状态。
- 警告用户意外操作或潜在的破坏性操作或情况。

对话框是模态的；它们会中断 BlackBerry® 设备的操作并被推送到堆栈顶部。 对话框包括一则消息、允许用户执行操作的按钮以及用于表示对话框类型的指示符。 对话框的大小取决于 BlackBerry 设备屏幕的大小。用户在 BlackBerry 设备上选择的主题确定了对话框的视觉风格。

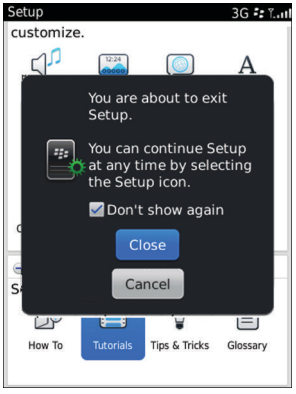


最佳实践：使用对话框

- 在对话框中使用按钮确认或取消操作。 避免使用链接或其他组件。
- 使用适合对话框类型的标准指示符。 避免在一个对话框中使用多个指示符。
- 尝试避免让用户在对话框中滚动。 如果对话框消息或按钮无法在对话框上完全显示，请包括滚动箭头。 如果您使用标准组件，则滚动箭头会自动出现（如有必要）。
- 始终允许用户按退出键来关闭对话框。 避免在用户按退出键关闭对话框时执行另一个操作。 例如，如果对话框允许用户更改设置，则当用户按退出键时，不要保存任何更改。 如有必要，稍后显示该对话框。
- 当应用程序屏幕上显示对话框时，如果用户按结束/电源键，则显示主屏幕或应用程序列表。 如果用户返回到该应用程序，则再次显示该对话框。

布局准则

- 将屏幕上的对话框居中。 如果您使用标准组件，BlackBerry® 设备会自动将对话框居中。
- 创建对话框时，对话框的宽度和高度不大于屏幕的宽度和高度的 90%。 如果您使用标准组件，BlackBerry 设备会自动为对话框计算适当的大小。
- 将对话框指示符与对话框消息垂直居中对齐。
- 对于大部分语言，都在指示符右侧及任何按钮上方的区域内显示消息。
- 请首先放置确认操作的按钮。 例如，先放置“保存”，然后再放置“放弃”或“取消”。
- 将对话框中的按钮水平居中。
- 将对话框中的按钮垂直居中。 垂直布局允许按钮进行扩展，以容纳本地化的按钮标签。
- 如果包含复选框，请将复选框对齐与对话框消息对齐匹配。 将复选框放到按钮上。 默认情况下，应选中该复选框，除非该对话框显示含有对用户非常重要之信息的消息。



消息准则

- 内容要明确。 如果可能，请使用短句来清晰地描述显示该对话框的原因以及如何消除它的操作。
- 尽量在消息中使用完整的句子。
- 使用用户理解的词汇。 例如，使用“文件未能保存，因为媒体卡已满”，而不用“将文件写入磁盘时发生错误”。
- 尽量使用肯定的语言，并避免责备用户。 永不书写责备用户错误或意外情况的消息。 相反，请着重于那些用户可以执行以解决问题的操作。

- 使用第二人称(您、您的)来指代用户。
- 请遵循句子类的首字母大写规则。
- 避免在消息中使用感叹号(！)。
- 除非您在表示进度(例如，“请稍候...”)，否则避免在消息中使用省略号(...)。

按钮准则

- 对于默认按钮，请使用用户最可能单击的按钮。 避免将与破坏性操作相关联的按钮用作默认按钮。 此规则的例外情况：当用户发起一次破坏性较小的操作(如删除单个项目)时，以及当用户的最常用操作是为了继续该操作时。
- 避免在对话框中使用超过三个按钮。 如果按钮数量超过三个，请考虑使用应用程序屏幕，而不使用单选按钮。
- 在带物理键盘的 BlackBerry 设备上，提供按钮的快捷键。 通常，快捷键是按钮标签的首个字母。
- 使用简洁明了的标签。
- 尽量使用名称为单个词的标签。 按钮的大小会根据标签的长度而变化。 如果标签太长，则省略号 (...) 表示文本被截断。
- 避免使用“是”和“否”标签，而使用描述相关操作的动词(例如，“取消”、“删除”、“放弃”或“保存”)。这种方法会帮助用户快速、容易地理解当他们单击该按钮时所执行的操作。 如有必要，请在屏幕上的其他位置(例如，在一则应用程序消息中)包括更多的描述性文本。
- 避免在标签中使用符号或图形。
- 避免在标签中使用标点符号。 在按钮标签中使用省略号表明需要附加信息才能执行相关联的操作。

创建对话框

警报对话框可用于向用户通知重要操作（比如，关闭 BlackBerry® 设备）或错误（比如，键入无效的信息）。警报对话框中将显示一个感叹号 (!) 指示符。 要关闭警报对话框，用户可以单击“确定”或按退出键。有关对话框的其它类型的详细信息，请参阅 BlackBerry® Java® Development Environment 的 API 参考。

1. 导入 `net.rim.device.api.ui.component.Dialog` 类。
2. 创建指定要显示的警报文本的警报对话框。

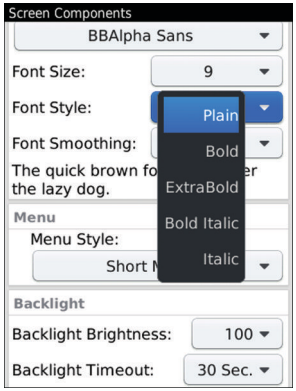
```
Dialog.alert("Specify the alert text that you want to display.")
```

下拉列表

使用下拉列表来提供一组互相排斥的值。

用户可通过下拉列表执行以下操作：

操作	仅带触控板的 BlackBerry 设备	带触摸屏和触控板的 BlackBerry 设备
从下拉列表中单击某值。	按输入键或单击触控板。	• 点按项目。 • 按输入键。 • 单击触控板。



最佳实践：使用下拉列表

- 当空间不足时，请为多个选项使用下拉列表。 如果有足够的空间，请考虑使用单选按钮，以便用户可以在屏幕上查看可用选项。
- 使用 `ObjectChoiceField` 或 `NumericChoiceField` 类创建下拉列表。
- 对于默认值，请使用用户最可能单击的值。
- 当用户滚动浏览该列表时，将高亮显示的选项用作默认焦点。
- 如果不需要用户单击某个值，请在下拉列表中包括“无”值。 始终将“无”值放置在列表顶部。
- 按照逻辑将值分组和排序（例如，将相关值分成一组或者首先包括最常用的值）。 避免将值按照字母顺序进行排序；字母顺序仅针对特定语言而言。

标签准则

- 对于下拉列表及下拉列表中的值，请使用简洁明了的标签。 请确定标签能清晰地描述当用户单击值时所发生的内容。 下拉列表的宽度根据值标签的长度而改变。 如果标签太长，则会显示一个省略号（...），表示文本被截断。
- 避免将标签“是”和“否”用作下拉列表值。 重新考虑选项措辞，并使用复选框。
- 将标签放置在下拉列表左侧。
- 对于下拉列表标签和值，请使用标题首字母大写规则（除非值读起来更像句子）。
- 请使用冒号（：）来分隔标签和下拉列表。

创建下拉列表

1. 导入所需的类和接口。

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

2. 通过扩展 `UiApplication` 类，创建应用程序框架。 在 `main()` 中，创建新类的实例并调用 `enterEventDispatcher()`，以启用应用程序来接收事件。 在应用程序构造器中，调用 `pushScreen()` 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 `MyUiScreen` 类表示自定义屏幕。

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

3. 通过扩展 MainScreen 类来创建应用程序的自定义屏幕。 在屏幕构造器中，调用 setTitle() 以指定屏幕的标题。

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

4. 在屏幕构造器中，使用 ObjectChoiceField 类创建显示单词或短语列表的下拉列表。 创建 String 数组以存储要在下拉列表中显示的项。 创建 int 以存储要在下拉列表中显示的默认项。 在 ObjectChoiceField 构造器中，指定下拉列表的标签、要显示的项的数组以及默认项。 在以下代码示例中，默认情况下显示星期三。 调用 add() 以将下拉列表添加至屏幕。

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

5. 在屏幕构造器中，使用 NumericChoiceField 类创建显示数字列表的第二个下拉列表。 在 NumericChoiceField 构造器中，指定下拉列表的标签、要在下拉列表中显示的第一个数字和最后一个数字、要用于数字列表的增量以及默认数字。 在以下代码示例中，数值参数存储在 int 对象中。 该下拉列表包括数字 1 至 31，默认情况下显示数字 10。 调用 add() 以将第二个下拉列表添加至屏幕。

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

6. 要覆盖提示用户在应用程序关闭前保存更改的默认功能，请在 MainScreen 类的扩展中覆盖 MainScreen.onSavePrompt() 方法。 在以下代码示例中，返回值 true 指明应用程序在关闭前不提示用户。

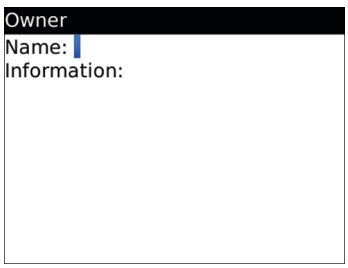
```
public boolean onSavePrompt()
{
    return true;
}
```

代码示例：创建下拉列表

```
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;
    }
}
```

标签

标签可用于显示标识控件的文本。



最佳实践：使用标签

- 使用 `LabelField` 类创建标签。
- 使用简洁明了的标签。
- 按照逻辑将标签分组和排序(例如,将相关标签分成一组或者首先包括最常用的标签) 避免将值按照字母顺序进行排序；字母顺序仅针对特定语言而言。
- 请使用冒号(：)来分隔标签。

创建文本标签

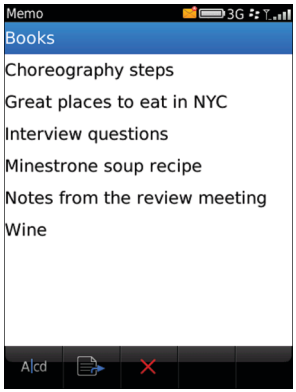
1. 导入 `net.rim.device.api.ui.component.LabelField` 类。
2. 创建 `LabelField` 的实例,以将文本标签添加至屏幕。

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

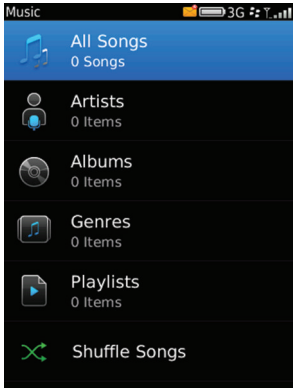
列表和表

列表和表可用于显示用户可高亮显示并打开的项目。 如果列表较长，将成批获取和显示项目。 用户到达列表中的最有一个项目时，下一批项目将显示在列表末尾。

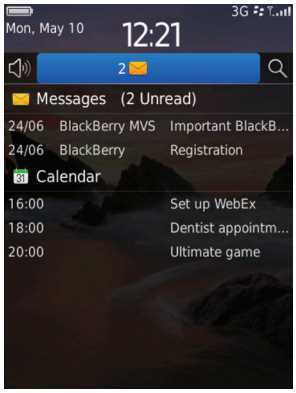
使用简单列表即可在行中显示文本。



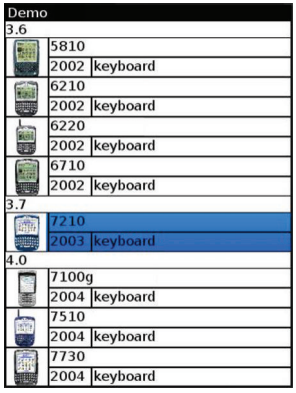
使用丰富列表可轻松显示多行文本和图标。 当前，丰富列表仅显示信息而无法交互。



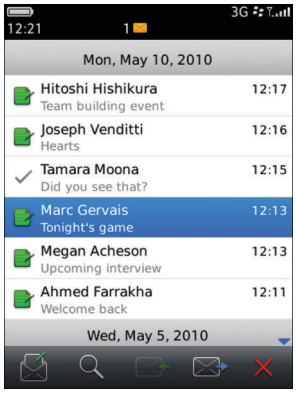
如果要在列和行中显示项目，请使用表。



您可以按标题对项目进行分组，帮助用户导航长列表。 例如，您可以创建折叠标题，便于用户在列表中查找项目。



您还可以将项目分组在始终显示在列表顶部的标题下。 例如，您可以添加日期作为标题，并将在该日期收到的消息都分组在该标题下。 用户可高亮显示标题以在项目组上执行操作，也可使用快捷键在列表中移动。



用户可在列表和表中执行以下操作：

操作	仅带触控板的 BlackBerry 设备	带触摸屏和触控板的 BlackBerry 设备
滚动浏览列表中的项目。	在触控板上垂直移动手指。	• 在屏幕上垂直拖动手指。 • 在屏幕上向上或向下滑动。 • 在触控板上垂直移动手指。
高亮显示列表中的项目。	在触控板上垂直移动手指。	• 轻触该项目。 • 在触控板上垂直移动手指。
打开列表中的项目。	• 单击触控板。 • 按输入键。	• 点按项目。 • 单击触控板。 • 按输入键。

最佳实践：实施列表和表

- 使用 SimpleList 类可创建带文本的列表。 使用 RichList 类可创建带文本和图标的唯读列表。 使用 TableView 类可创建带文本和图标的交互式丰富列表。
- 使用 TableView 类可创建带行和列的表。 如果表中的行数和列数固定，则可使用 GridFieldManager。
- 如果列表太长，因此希望在单独屏幕中显示项目，则在屏幕底部包括“下一页”和“上一页”按钮。 或者，如果列表太长（如成千上万个项目），则提供屏幕编号。
- 如果希望用户以列表形式浏览项目（例如，在消息列表或订阅源中），请分配用于移至列表中的上一项或下一项的快捷键。 尽量采用英语让用户可以通过按 N 移至列表中的下一项，按 P 移至列表中的上一项。

创建列表框

使用列表框可显示用户可从中选择一个或多个值的列表。

1. 导入以下类：

```
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. 通过扩展 UiApplication 类，创建应用程序框架。 在 main() 中，创建新类的实例并调用 enterEventDispatcher()，以让应用程序来接收事件。 在构造器中，调用 pushScreen 以显示应用程序的自定义屏幕。 CreateMenuScreen 类表示自定义屏幕（在步骤 3 中介绍）。

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

3. 通过扩展 MainScreen 类来创建应用程序的自定义屏幕。 在构造器中，调用 setTitle() 以显示屏幕的标题。 调用 add() 以在屏幕上显示文本字段。 调用 addMenuItem() 以将菜单项添加至 MainScreen 创建的菜单。

```
class ListFieldScreen extends MainScreen
{
    private ListField _listField;
    private Vector _listElements;
    public ListFieldScreen()
    {
        setTitle("List Field Sample");
    }
}
```

4. 在屏幕构造器中，创建列表框。 使用 Vector 类创建要添加至列表框的项目数组。 使用 ListField() 类创建列表框。 调用 add() 以将列表框添加至屏幕。 调用步骤 4 中介绍的 initializeList() 以构建列表框。

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

5. 使用 String 对象创建方法，以指定要显示在列表框中的项目。 调用 addElement() 以将项目添加至列表。 调用 setSize() 以指定列表框中的项目数。

```
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. 创建实施 ListFieldCallback 接口的类。 实施 drawListRow() 以将列表框项目添加至屏幕。 实施 get() 以返回位于指定索引的列表框项目。 实施 indexOfList() 以返回指定字符串的第一个匹配项。 实施 getPreferredWidth() 以检索列表框的宽度。

```
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();
    }
}
```

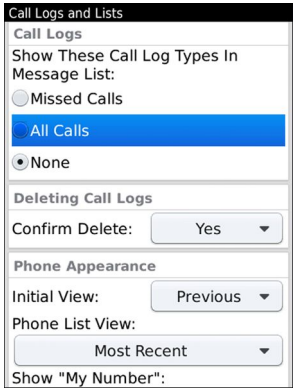
单选按钮

使用单选按钮可表明一组互相排斥但相关的选项。

通过单选按钮，用户可执行以下操作：

操作	仅带触控板的 BlackBerry 设备	带触摸屏和触控板的 BlackBerry 设备
单击某个单选按钮。	按空格键或单击触控板。	• 点按项目。 • 按空格键。

操作	仅带触控板的 BlackBerry 设备	带触摸屏和触控板的 BlackBerry 设备
		• 单击触控板。



最佳实践：实施单选按钮

- 当空间足够时，请对两个或多个选项使用单选按钮。 如果空间不足，则考虑使用下拉列表。
- 使用 RadioButtonField 类可创建单选按钮。
- 请确定单选按钮的内容是静态的。 单选按钮的内容不应随上下文而改变。
- 当用户选择一个单选按钮时，不要启动操作。 例如，不要打开新屏幕。
- 将单选按钮垂直对齐。
- 按照逻辑将值分组和排序（例如，将相关单选按钮分成一组或者首先包括最常用的值） 避免将单选按钮按照字母顺序进行排序；字母顺序仅针对特定语言而言。

标签准则

- 使用简洁明了的标签。 请确定标签能清晰地描述当用户选择单选按钮时发生的情况。 如果标签太长，标签会换行。
- 将标签放在单选按钮右侧。
- 请遵循句子类的首字母大写规则。
- 不要使用终止标点符号。

单击单选按钮。

通过使用 RadioButtonGroup 类，可创建单选按钮组。 用户只能从单选按钮组中选择一个选项。

1. 导入所需的类和接口。

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

2. 通过扩展 UiApplication 类,创建应用程序框架。 在 main() 中,创建新类的实例并调用 enterEventDispatcher(),以启用应用程序来接收事件。 在应用程序构造器中,调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 MyUiScreen 类表示自定义屏幕。

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

3. 通过扩展 MainScreen 类来创建应用程序的自定义屏幕。 在屏幕构造器中,调用 setTitle() 以指定屏幕的标题。

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

4. 在屏幕构造器中,使用 RadioButtonGroup 类创建单选按钮组。 使用 RadioButtonField 类创建要添加至组的单选按钮。 在 RadioButtonField 构造器中,指定单选按钮的标签、组和布尔值,以指明默认选择(例如,true 表示默认情况下将选择该单选按钮)。 调用 add() 以将单选按钮添加至屏幕。

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

5. 要覆盖提示用户在应用程序关闭前保存更改的默认功能,请在 MainScreen 类的扩展中覆盖 MainScreen.onSavePrompt() 方法。 在以下代码示例中,返回值 true 指明应用程序在关闭前不提示用户。

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

代码示例:创建单选按钮

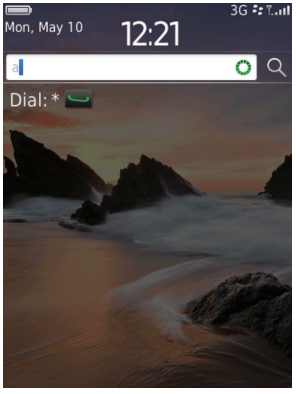
```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
public class MyUi extends UiApplication
{
    public static void main(String[] args)
    {
```

```
        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;
    }
}
```

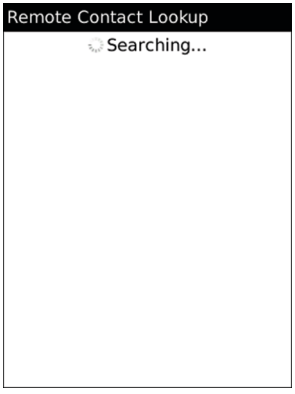
活动指示符和进度指示符

活动指示符和进度指示符向用户显示其 BlackBerry® 设备正在执行操作，如搜索项目或删除语言。

如果要显示 BlackBerry® 设备正在工作且无法确定操作的持续时间，请使用活动指示符。 您可将活动指示符添加至任何组件，如屏幕、文本字段或列表项目。 您也可以将文本添加至活动指示符以描述操作。

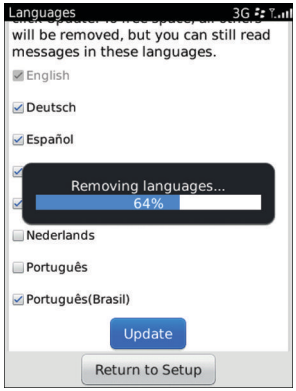


字段中的活动指示符



带文本的活动指示符

如果可以确定操作的持续时间时，请使用进度指示符。 进度指示符包括一个标签和一个水平条，标签表明操作的类型，水平条随着操作的进展从左到右填充。 水平条中的百分比表明已完成的操作比例。



最佳实践：实施活动指示符和进度指示符

- 当完成操作所花费的时间长于 2 秒时，请始终指明进度。
- 当您可以确定操作的持续时间时，请使用进度指示符。
- 当您不能确定操作的持续时间时，请使用活动指示符。
- 使用 `ActivityIndicatorView` 类可创建活动指示符。 使用 `ProgressIndicatorView` 类可创建进度指示符。
- 提供有用的进度信息。 例如，如果用户在将应用程序下载至设备，请指明 BlackBerry® 设备已下载的数据百分比。 进度信息要尽量准确。
- 始终允许用户使用结束键来隐藏进度指示符。

文本准则

- 使用简洁明了的描述性文本（例如，“正在加载数据...”或“正在构建应用程序列表...”）
- 如果操作太长，因此想要传达每个阶段发生的情况，则提供描述每个阶段的文本（例如，“正在下载...”或“正在安装...”）
- 请遵循句子类的首字母大写规则。
- 请将省略号 (...) 用作标点符号。

指示活动或任务进度

您可使用 `net.rim.device.api.ui.component.progressindicator` 数据包中提供的活动和进度指示符 API 在屏幕上显示可视线索，以指示工作正在进行或任务正在进行。 您可表示持续时间未知的活动，也可以表示可采用数值（例如，已完成任务的百分比）表示的进度。

活动和进度指示符 API 使用模型-视图-控制器设计模式，并包括两个负责呈现活动或进度的字段：

- 通过使用包含动画帧的位图，`ActivityImageField` 类可表示活动。 显示的帧将随时间而变化。 通常，此字段是旋转器、沙漏或其他类似动画可视线索。 使用 `ActivityIndicatorView` 类构建和设置此字段。
- `ProgressBarField` 类使用将在任务完成时填充的条形表示任务进度。 使用 `ProgressIndicatorView` 类构建和设置此字段。

“进度指示符演示”示例应用程序包括在 BlackBerry® Java® SDK 中。 此示例应用程序演示如何创建和操控各种活动指示符和进度指示符条形。

指示活动

您可在 BlackBerry® 设备应用程序中显示指示工作正在进行的字段。 通常，此字段是旋转器、沙漏或其他类似动画可视线索。 使用 ActivityImageField 类实施此字段。

1. 导入所需的类和接口。

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

2. 创建包含动画帧的图像并将其包括在项目中。 图像必须包含一系列水平排列的动画帧。 图像的宽度必须为帧的宽度乘以帧数。

   例如，以下图像具有 5 个大小相等的帧。

3. 根据图像创建位图。

```
Bitmap bitmap = Bitmap.getBitmapResource("spinner.png");
```

4. 创建 ActivityIndicatorView 对象。 您可在构造器中指定视图样式。

```
ActivityIndicatorView view = new ActivityIndicatorView(Field.USE_ALL_WIDTH);
```

   要指定用于布局和焦点的管理器，请在创建 ActivityIndicatorView 时提供第二个自变量。 如果未指定管理器，将使用 VerticalFieldManager 对象。

5. 创建模型和控制器。

```
ActivityIndicatorModel model = new ActivityIndicatorModel();
ActivityIndicatorController controller = new ActivityIndicatorController();
```

6. 连接视图、模型和控制器。

```
view.setController(controller);
view.setModel(model);
controller.setModel(model);
controller.setView(view);
model.setController(controller);
```

7. 根据位图创建呈现活动的字段。

```
view.createActivityImageField(bitmap, 5, Field.FIELD_HCENTER);
```

   在此示例中，位图包含 5 个帧，并显示在视图管理器的中心。

8. 将视图添加至屏幕。

```
add(view);
```

9. 通过根据需要控制视图模型，控制活动指示。

```
MenuItem _stopIndicator = new MenuItem("Stop spinner", 66000, 0)
{
    public void run()
    {
        view.getModel().cancel();
    }
};
MenuItem _resumeIndicator = new MenuItem("Resume spinner", 66010, 0)
{
    public void run()
    {
        view.getModel().resume();
    }
};
```

此示例将调用模型的 `cancel()` 方法停止动画。 此示例将调用模型的 `resume()` 方法恢复动画。

代码示例

```
public class ActivityIndicatorScreen extends MainScreen
{
    ActivityIndicatorView view = new ActivityIndicatorView(Field.USE_ALL_WIDTH);
    ActivityIndicatorModel model = new ActivityIndicatorModel();
    ActivityIndicatorController controller = new ActivityIndicatorController();
    public ActivityIndicatorScreen ()
    {
        setTitle("Activity Indicator Demo");
        view.setController(controller);
        view.setModel(model);
        controller.setModel(model);
        controller.setView(view);
        model.setController(controller);
        // Define the indicator image and create a field from it
        Bitmap bitmap = Bitmap.getBitmapResource("spinner.png");
        view.createActivityImageField(bitmap, 5, Field.FIELD_HCENTER);
        // add the view to the screen
        add(view);
        MenuItem _stopIndicator = new MenuItem("Stop spinner", 66000, 0)
        {
            public void run()
            {
                view.getModel().cancel();
            }
        };
        MenuItem _resumeIndicator = new MenuItem("Resume spinner", 66010, 0)
        {
            public void run()
            {
                view.getModel().resume();
            }
        };
        addMenuItem(_stopIndicator);
        addMenuItem(_resumeIndicator);
    }
}
```

BlackBerry® Java® SDK 中包括的“进度指示符演示”示例应用程序创建并操控各种活动指示符，包括如上所示的旋转器。

指示进度

您可在 BlackBerry® 设备应用程序中显示指示任务正在进行的字段。 进度由将在任务完成时填充的条形表示。

1. 导入所需的类和接口。

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

2. 创建 ProgressIndicatorView 对象。 您可在构造器中指定视图样式。 在以下示例中，未指定样式。

```
ProgressIndicatorView view = new ProgressIndicatorView(0);
```

要指定用于布局和焦点的管理器，请在创建 ProgressIndicatorView 时提供第二个自变量。 如果未指定管理器，将使用 VerticalFieldManager 对象。

3. 创建 ProgressIndicatorController 对象。

```
ProgressIndicatorController controller = new ProgressIndicatorController();
```

4. 创建 ProgressIndicatorModel 对象。 此对象表示任务的进度。 创建此对象时，可指定模型的初始值、最大值和最小值。 ProgressIndicatorModel 使用 Adjustment 对象允许以线程方式访问数据模型，且允许以整数值表示任务。

```
ProgressIndicatorModel model = new ProgressIndicatorModel(0, 100, 0);
```

在此示例中，任务从值 0 开始，并可达到 100。 这些值将任务完成量建模为百分比。

5. 连接控制器、模型和视图。

```
model.setController(controller);
view.setModel(model);
view.setController(controller);
controller.setModel(model);
controller.setView(view);
```

6. 创建线程以处理任务。 通常，使用线程执行需要进度指示符的任务。 随着任务继续，将更新模型值以反映任务进度。

```
class ProgressThread extends Thread
{
    private boolean _paused;
    private boolean _stop;
    public void run()
    {
        // perform the task here and update the model's value as appropriate
    }
    public synchronized void setPaused(boolean paused)
    {
        // pause the indicator here
    }
    public synchronized void stopThread()
```

```
    {
        // stop the indicator here
    }
}
```

7. 创建实施 ProgressIndicatorListener 接口的类,以便获得数据模型更改通知。 您可获得有关模型重置、恢复或取消的通知,或有关以非编程方式更改模型值的通知。

```
private final class DemoProgressIndicatorListener implements
ProgressIndicatorListener
{
    public void cancelled()
    {
        _progressThread.setPaused(true);
    }
    public void resumed()
    {
        _progressThread.setPaused(false);
    }
...
}
```

8. 将监听器与模型关联。

```
model.addListener(new DemoProgressIndicatorListener());
```

9. 设置视图的标签。 标签显示上面呈现的 ProgressIndicatorField(假定管理器是 VerticalFieldManager,这是默认管理器)。

```
view.setLabel("Percent completion");
```

10. 创建呈现进度的字段。 您可在 ProgressBarField 中提供定义的样式,以指定是否在条形上显示文本及其显示方式。 默认情况下,将模型值显示在条形占用空间的中心。

```
view.createProgressBar(Field.FIELD_HCENTER);
```

在此示例中,进度条使用默认文本样式,并显示在视图管理器的中心。

11. 将视图添加至屏幕。

```
add(view);
```

12. 通过根据需要控制视图模型,控制进度指示。

```
MenuItem _pauseIndicator = new MenuItem("Pause indicator", 66010, 0)
{
    public void run()
    {
        view.getModel().cancel();
    }
};
MenuItem _resumeIndicator = new MenuItem("Resume indicator", 66020, 0)
{
    public void run()
    {
```

```
            view.getModel().resume();
        }
    };
```

此示例将调用模型的 `cancel()` 方法停止进度指示。 此示例将调用模型的 `resume()` 方法恢复进度指示。

代码示例

```
public class ProgressIndicatorScreen extends MainScreen
{
    ProgressIndicatorView view = new ProgressIndicatorView(0);
    ProgressIndicatorModel model = new ProgressIndicatorModel(0, 100, 0);
    ProgressIndicatorController controller = new ProgressIndicatorController();
    ProgressThread _progressThread;
    public ProgressIndicatorScreen()
    {
        setTitle("Progress Indicator Screen");
        model.setController(controller);
        model.addListener(new DemoProgressIndicatorListener());
        view.setModel(model);
        view.setController(controller);
        controller.setModel(model);
        controller.setView(view);
        view.setLabel("Percent completion");
        view.createProgressBar(Field.FIELD_HCENTER);
        add(view);
        MenuItem _startIndicator = new MenuItem("Start indicator", 66000, 0)
        {
            public void run()
            {
              if(_progressThread != null)
              {
                  _progressThread.stopThread();
              }
              _progressThread = new ProgressThread();
              _progressThread.start();
            }
        };
        MenuItem _pauseIndicator = new MenuItem("Pause indicator", 66010, 0)
        {
            public void run()
            {
                view.getModel().cancel();
            }
        };
        MenuItem _resumeIndicator = new MenuItem("Resume indicator", 66020, 0)
        {
            public void run()
            {
                view.getModel().resume();
            }
        };
        addMenuItem(_startIndicator);
        addMenuItem(_pauseIndicator);
        addMenuItem(_resumeIndicator);
```

```
    }
    class ProgressThread extends Thread
    {
        private boolean _paused;
        private boolean _stop;
        public void run()
        {
            // Run dummy operations to simulate the processing of
            // a collection of data.
            for(int i = 0; i <= 100; ++i)
            {
                synchronized(this)
                {
                    if(_stop)
                    {
                        break;
                    }
                    if(_paused)
                    {
                        try
                        {
                            wait();
                        }
                        catch(InterruptedException ie)
                        {
                        }
                    }
                }
                ProgressIndicatorScreen.this.model.setValue(i);
                try
                {
                    // Simulate work
                    sleep(250);
                }
                catch(InterruptedException ie)
                {
                }
            }
        }
        public synchronized void setPaused(boolean paused)
        {
            _paused = paused;
            this.notify();
        }
        public synchronized void stopThread()
        {
            _stop = true;
            if(_paused)
            {
                // If the thread is in a paused state, wake it up
                this.notify();
            }
        }
    }
    private final class DemoProgressIndicatorListener implements
    ProgressIndicatorListener
```

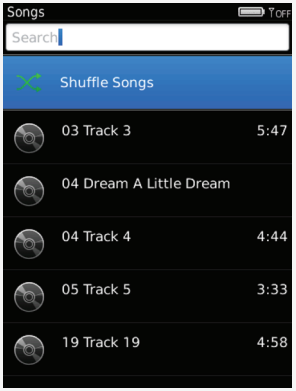
```
    {
        public void cancelled()
        {
            _progressThread.setPaused(true);
        }
        public void resumed()
        {
            _progressThread.setPaused(false);
        }
        public void reset()
        {
            // Not implemented
        }
        public void setNonProgrammaticValue(int value)
        {
            // Not implemented
        }
        public void configurationChanged(Adjustment source)
        {
            // Not implemented
        }
        public void valueChanged(Adjustment source)
        {
            // Not implemented
        }
    }
}
```

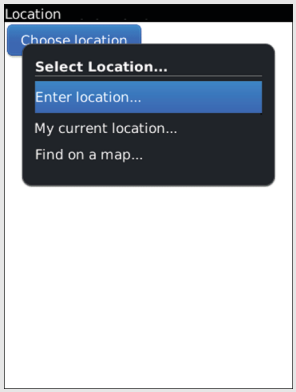
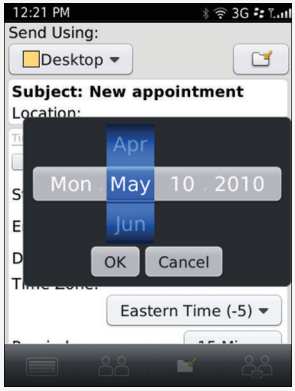
BlackBerry® Java® SDK 中包括的“进度指示符演示”示例应用程序创建并操控进度条。

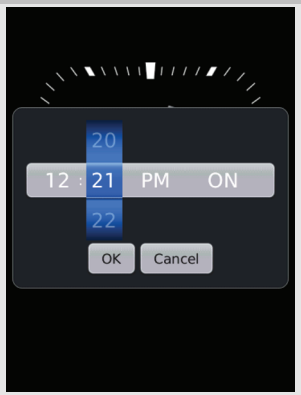
选择器

选择器可帮助用户轻松地选择列表中的项目。

选择器类型	说明
文件	此选择器可供用户浏览其 BlackBerry® 设备上的文件。



选择器类型	说明
位置	此选择器可供用户从定义的列表中选择位置。 例如，此选择器可供用户选择其 GPS 位置或先前选择的位置。 
日期	此选择器可供用户选择特定的年月日。 例如，此选择器可供用户选择年份和月份以指定其信用卡到期时间。 
时间	此选择器可供用户选择特定的小时、分钟和秒数。

选择器类型	说明
	

最佳实践：实施选择器

使用 FilePicker、LocationPicker 和 DateTimePicker 类可创建选择器。

文件选择器的准则

- 指定特定视图以显示与用户目标匹配的文件类型。 例如，如果用户在浏览图片，则在文件选择器中显示图片。
- 如果可能，允许用户从适当的默认文件夹开始浏览。

创建日期选择器

1. 导入所需的类和接口。

```
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. 通过扩展 UiApplication 类，创建应用程序框架。 在 main() 中，创建新类的实例并调用 enterEventDispatcher()，以启用应用程序来接收事件。 在构造器中，调用 pushScreen() 以显示应用程序的自定义屏幕。 DatePickScreen 类表示自定义屏幕（在步骤 3 中介绍）。

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

```
        pushScreen(new DatePickScreen());
    }
}
```

3. 通过扩展 MainScreen 类来创建应用程序的自定义屏幕。 在构造器中，调用 setTitle() 以在屏幕上显示标题。 调用 add() 以在屏幕上显示丰富文本字段。

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

4. 通过调用 invokeLater()，将代码段添加至应用程序的事件队列。 创建 Runnable 对象并将其作为参数传递至 invokeLater()。 覆盖 Runnable 定义中的 run()。

```
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. 在 run() 中，调用 DateTimePicker.getInstance() 以返回 DateTimePicker 对象。 调用 doModal() 以显示日期选择器。 调用 getDateTime() 以返回表示用户选择的日期和时间的 Calendar 对象。 使用 getTime() 将日期和时间作为 Date 对象返回。 使用 Dialog.alert() 显示选定日期和时间。

```
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());
            }
```

```
            });
        }
    }
```

代码示例：创建日期选择器

```
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());
            }
        });
    }
}
```

创建文件选择器

1. 导入所需的类和接口。

```
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. 通过扩展 UiApplication 类,创建应用程序框架。 在 main() 中,创建新类的实例并调用 enterEventDispatcher(),以启用应用程序来接收事件。 在应用程序构造器中,调用 pushScreen() 以显示应用程序的自定义屏幕。 FilePickScreen 类表示自定义屏幕(在步骤 3 中介绍)。

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

3. 通过扩展 MainScreen 类来创建应用程序的自定义屏幕。 在屏幕构造器中,调用 setTitle() 以指定屏幕的标题。 调用 add() 以将标签字段添加至屏幕。

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

4. 在屏幕构造器中,调用 invokeLater() 以将代码段添加至应用程序的事件队列。 创建 Runnable 对象并将其作为参数传递至 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. 覆盖 Runnable 定义中的 run()。 在 run() 中,调用 FilePicker.getInstance() 以返回 FilePicker 对象。 创建在步骤 6 中介绍的 FilePickListener 对象。 调用 setListener() 以注册文件选择器的监听器。 调用 show() 以在屏幕上显示文件选择器。

```
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. 调用 Dialog.alert 以创建对话框,其中包含显示选定哪个文件的消息。 通过覆盖 selectionDone(),在实施 FilePicker.Listener 接口的类中调用 Dialog.alert。 用户使用文件选择器选择文件时,应用程序将调用 selectionDone()。

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

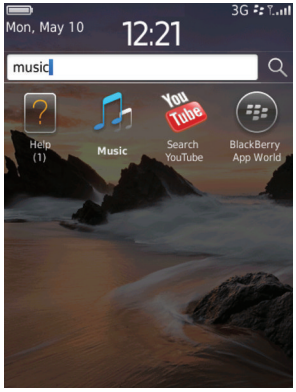
代码示例：创建文件选择器

```
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);
    }
}
```

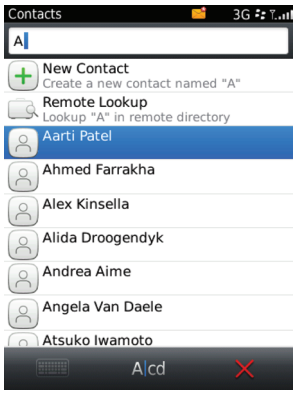
搜索

用户可使用主屏幕中的搜索字段搜索设备上任何应用程序中的项目，包括第三方应用程序。 搜索还可包括未存储在设备上的内容，如组织的数据库或网站。



用户还可使用应用程序中的搜索字段搜索该应用程序中的项目。 例如，用户可在消息列表中搜索电子邮件，在媒体应用程序中搜索歌曲，或在联系人列表中搜索联系人。 在某些情况下，可能需要显示其他应用程序中的搜索结果。

在某些应用程序中，搜索字段显示在屏幕上。 在某些情况下，在完整菜单、弹出菜单或工具栏中提供搜索。 用户在搜索字段中键入文本时，将显示搜索结果。 如果返回大量搜索结果，则可让用户将搜索范围缩小至某个字段或字段组。 例如，如果用户搜索消息列表，则可使用搜索字段右侧的下拉列表将搜索缩小为“收件人”字段或“主题”字段。 有关将搜索字段添加至应用程序的详细信息，请参阅 *BlackBerry《 Java Application Integration Development Guide》*。



用户可在搜索字段中执行以下操作：

操作	仅带触控板的 BlackBerry 设备	带触摸屏和触控板的 BlackBerry 设备
在搜索结果中打开高亮显示的项目。	• 按输入键。 • 单击触控板。	• 点按屏幕。 • 按输入键。 • 单击触控板。
显示带搜索结果的相关操作的弹出菜单（例如，呼叫联系人）	单击并按住触控板。	• 触按并按住屏幕上的搜索结果。 • 单击并按住触控板。

您可以在应用程序中注册内容，以便可将其包括在搜索结果中。 您也可以注册应用程序，作为用户扩展搜索的方式。 例如，如果用户在媒体应用程序中搜索歌曲，但未找到歌曲，则可让用户搜索应用程序，作为搜索结果的备选来源。 有关注册内容或应用程序的详细信息，请参阅 *《BlackBerry Java Application Integration Development Guide》*。

最佳实践：实施搜索

- 使用 `net.rim.device.api.unifiedsearch` 数据包可实施搜索功能。
- 仔细选择注册要包括在搜索结果中的内容。 仅注册可向用户提供有意义搜索结果的内容。
- 尝试将最相关项目显示在搜索结果列表开头。 例如，如果用户在查找具有多个不同位置的餐馆，则将最靠近用户位置的餐馆显示在搜索结果列表开头。
- 在搜索结果中，将与用户键入的文本匹配的文本设置为粗体。 此方法可帮助用户了解每个项目显示在搜索结果列表中的原因。

- 如果用户需要在屏幕上搜索某个单词(例如,在消息或网页中),则使用菜单中的术语“查找”。

实施搜索字段

- 将默认焦点设置为搜索字段。 显示搜索结果时,将默认焦点设置为搜索结果列表中的第一个项目。
- 在搜索字段中将术语“搜索”用作提示文本。
- 不要让搜索字段区分大小写。
- 将标题栏下的搜索字段放在应用程序屏幕上。
- 不要给包括搜索字段的屏幕分配快捷键。 如果要使用快捷键,请提供备选搜索功能。 例如,允许用户使用菜单在消息列表中搜索消息。

创建搜索字段

您可以创建使用 net.rim.device.api.ui.component 数据包中包括的 KeywordFilterField 类的应用程序，以提供包含单个文本输入字段和可选择元素列表的 UI 字段。 用户在搜索字段中键入文本时，应用程序将过滤列表中以搜索文本开头的元素。 有关使用 KeywordFilterField 类的详细信息，请参阅 BlackBerry® Java® Development Environment 4.3.1 或更高版本中附带的“关键字过滤器字段”示例应用程序。

1. 导入所需的类和接口。

```
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. 创建变量。 在以下代码示例中，CountryList 扩展 SortedReadableList 类并实施 KeywordProvider 接口。

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

3. 要创建可选择文本项列表,请使用文本文件中的数据填充矢量。

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

4. 创建扩展 SortedReadableList 类的类的实例。

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

5. 要指定列表的元素,请创建 KeywordFilterField 对象的新实例。

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

6. 调用 KeywordFilterField.setList()。

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

7. 设置 KeywordFilterFIeld 的输入字段的标签。

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

8. 创建应用程序的主屏幕并将 KeywordFilterField 添加至主屏幕。

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

9. 要创建填充并返回 Country 对象(包含文本文件中的数据)的矢量的方法,请在方法签名中将 Vector 指定为返回类型。

```
public Vector getDataFromFile()
```

10. 创建并存储新 Vector 对象的引用。

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

11. 创建至文本文件的输入流。

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

12. 从输入流读取以 CRLF 分隔的行。

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

13. 从输入流读取数据,一次读取一行,直至到达文件结尾标记。 每行都将解析,以提取用于构造 Country 对象的数据。

```
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. 要将关键字添加至可选择文本项列表,请调用 SortedReadableList.doAdd(element)。

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

15. 要更新可选择文本项列表，请调用 KeywordFilterField.updateList()。

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

16. 要获取 BlackBerry 设备用户键入 KeywordFilterField 的关键字，请调用 KeywordFilterField.getKeyword()。

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

自动填充文本字段

您可以使用自动填充文本字段预测 BlackBerry® 设备用户要键入的内容，以及在用户完全键入前显示单词或短语。

创建 AutoCompleteField 对象时，必须将 BasicFilteredList 对象与其关联。 BasicFilteredList 维护用于比较的数据对象的引用，以生成单词和短语列表。 您可以配置数据对象中用于比较的字段以及找到匹配项时显示的字段。 例如，可将用户键入的文本与 DATA_SOURCE_CONTACTS 数据源中的 DATA_FIELD_CONTACTS_BIRTHDAY 字段的值进行比较，并返回对应 DATA_FIELD_CONTACTS_NAME_FULL 字段的值。

可绑定至 BasicFilteredList 以便与 AutoCompleteField 结合使用的数据类型有四种。

您可以采用以下方式之一指定用于比较的字符串集：

- 文字字符串的数组
- 支持 toString() 的对象数组
- BlackBerry 设备上的数据源，如联系人、记事、任务和各种类型的媒体文件
- 具有对应索引的对象数组和字符串数组

默认情况下，自动填充文本字段在下拉列表中显示比较进程返回的字符串集。 通过在创建自动填充文本字段时指定样式标记，可配置此列表的外观。 您可以更改列表的显示方式以及用户与列表交互的方式。

从数据集创建自动填充文本字段

1. 导入所需的类和接口。

```
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. 通过扩展 UiApplication 类，创建应用程序框架。 在 main() 中，创建新类的实例并调用 enterEventDispatcher()，以启用应用程序来接收事件。 在构造器中，调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 HomeScreen 类表示自定义屏幕。

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

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

3. 通过扩展 MainScreen 类来创建自定义屏幕。

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

4. 在构造器中,创建 BasicFilteredList 对象。 创建 String 数组并将要用于匹配的字符串存储在此字符串中。 在此示例中,字符串是星期几。 调用 addDataSet() 以将数组中的数据与 BasicFilteredList 绑定。

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

5. 在构造器中,创建 AutoCompleteField 对象。 将 BasicFilteredList 的实例传递至 AutoCompleteField 构造器,以将 BasicFilteredList 绑定至自动填充文本字段。 调用 add() 以将字段添加至屏幕。

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

代码示例:从数据源创建自动填充字段

```
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);
    }
}
```

从数据源创建自动填充文本字段

1. 导入所需的类和接口。

```
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. 通过扩展 UiApplication 类,创建应用程序框架。 在 main() 中,创建新类的实例并调用 enterEventDispatcher(),以启用应用程序来接收事件。 在应用程序构造器中,调用 pushScreen() 以显示应用程序的自定义屏幕。 HomeScreen 类表示自定义屏幕(在步骤 3 中介绍)。

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

3. 通过扩展 MainScreen 类来创建应用程序的自定义屏幕。

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

4. 在屏幕构造器中,创建 BasicFilteredList 对象。 调用 addDataSource() 以将数据源与 BasicFilteredList 绑定。 在此示例中,数据是联系人信息,而数据源是联系人列表。 传递至 addDataSource() 的第一个自变量是唯一 ID。 第二个自变量将 BasicFilteredList 对象绑定至数据源。 第三个自变量指定要用于比较的数据源字段集。 第四个自变量指定要在找到匹配项时提供的数据源字段集。 在此示例中,要用于比较的字段与要在找到匹配项时提供的字段相同。 第五个自变量指定主要显示字段。 第六个自变量指定次要显示字段,并设置为 -1(如果不使用它)。 最后的自变量指定 BasicFilteredList 的名称;如果指定 null,将自动生成其值。

```
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. 在屏幕构造器中,创建 AutoCompleteField 对象。 将在步骤 4 中创建的 BasicFilteredList 对象传递至 AutoCompleteField 构造器,以将 BasicFilteredList 绑定至自动填充文本字段。 调用 add() 以将字段添加至屏幕。

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

代码示例：从数据源创建自动填充文本字段

```
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);
    }
}
```

将数据源和字段用于自动填充文本字段

您可以使用 `AutoCompleteField` 类和 `BasicFilteredList` 类，以将用户在自动填充文本字段中键入的文本与指定数据源中的字段的值进行比较。 通过使用作为参数传递至 `BasicFilteredList` 类的构造器的 `AutoCompleteField` 对象，可指定要使用的字段及其数据源。

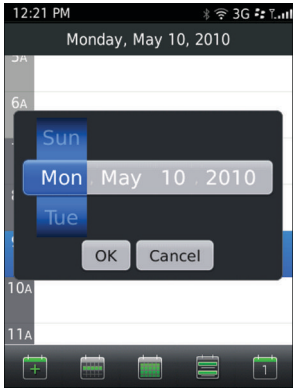
数据源	字段
DATA_SOURCE_APPOINTMENTS	• DATA_FIELD_APPOINTMENTS_ALL • DATA_FIELD_APPOINTMENTS_ATTENDEES • DATA_FIELD_APPOINTMENTS_ORGANIZER • DATA_FIELD_APPOINTMENTS_SUBJECT
DATA_SOURCE_CONTACTS	• DATA_FIELD_CONTACTS_ADDRESS_ALL • DATA_FIELD_CONTACTS_ADDRESS_HOME • DATA_FIELD_CONTACTS_ADDRESS_WORK • DATA_FIELD_CONTACTS_ANNIVERSARY • 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_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

数据源	字段
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

旋转框

旋转框可供用户轻易地从有序列表中选择项目。 例如，用户可使用旋转框查找数字或更改星期几。

操作	仅带触控板的 BlackBerry 设备	带触摸屏和触控板的 BlackBerry 设备
在列表中查找项目。	在触控板上垂直移动手指。	• 在屏幕上垂直拖动手指。 • 在屏幕上向上或向下滑动。 • 在触控板上垂直移动手指。 在屏幕或触控板中上移或下移手指。
从列表中选择项目。	单击触控板。	• 从屏幕提起手指。 • 单击触控板。
移至另一个旋转框。	在触控板上垂直移动手指。	• 在屏幕上垂直拖动手指。 • 在触控板上垂直移动手指。



创建旋转框

1. 导入所需的类和接口。

```
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. 通过扩展 `UiApplication` 类，创建应用程序框架。 在 `main()` 中，创建新类的实例并调用 `enterEventDispatcher()`，以启用应用程序来接收事件。 在应用程序构造器中，调用 `pushScreen()` 以显示应用程序的自定义屏幕。 `HomeScreen` 类表示自定义屏幕（在步骤 3 中介绍）。

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

3. 通过扩展 `MainScreen` 类来创建应用程序的自定义屏幕。 声明旋转框中每个字段的变量并声明旋转框字段管理器的变量。

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

```
    {
    }
}
```

4. 在屏幕构造器中，创建每个旋转框字段的 String 对象的数组。

```
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. 在屏幕构造器中，创建旋转框字段管理器和两个旋转框字段的新实例。 将字符串的相应数组作为自变量传递至每个旋转框字段构造器。

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

6. 在屏幕构造器中，将旋转框字段添加至旋转框字段管理器。 调用 `add()` 以将管理器及其包含的字段添加至屏幕。

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

代码示例：创建旋转框

```
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();
    }
}
```

最佳实践：使用旋转框

- 对于一列具有先后顺序的项目，请使用旋转框。
- 对于没有先后顺序的项目或间隔不规律的项目，请使用下拉列表。 对于一小列没有先后顺序的项目，您可以使用旋转框让用户获得更多交互式体验。
- 如果屏幕上同时出现其他几个组件，请避免使用旋转框。
- 使用 SpinBoxField 和 SpinBoxFieldManager 类创建旋转框。
- 尽量将旋转框添加到对话框中，而不添加到屏幕中。
- 当用户高亮显示旋转框时，纵向显示三到五个项目。
- 对于一系列的项目，请使用可识别模式（例如，5、10、15），以便用户可以估测他们需要滚动多少才能找到目标项目。
- 避免让用户水平滚动来查看多个旋转框。 如有必要，请将旋转框分隔到多个字段中。
- 如果旋转框中的文本太长，请使用省略号（...）。

文本字段

用户可以使用文本字段键入文本。

文本字段的类型	说明
电子邮件	用户可在电子邮件地址字段中插入符号 (@) 或句号 (.)，只需按住 Space 键。
日期和时间	通过键盘或通过在触控板上垂直移动手指，用户可更改带触控板的 BlackBerry® 设备上的日期或时间。 通过在屏幕上向上或向下滑动，用户可更改带触摸屏的 BlackBerry 设备的日期或时间。
号码	用户需要使用物理键盘在数字字段中键入内容时，BlackBerry 设备将切换至数字锁模式，因此用户无需按 Alt 键来键入数字。

文本字段的类型	说明
	用户需要使用虚拟键盘在数字字段中键入内容时，将显示数字键盘。
密码	当用户在密码字段中键入时，将出现星号 (*) 而不是文本。 在密码字段中，用户不能剪切、复制或粘贴文本或使用自动图文集功能。 在采用了 SureType® 技术的 BlackBerry 设备上，密码字段默认采用多次击键输入法。
电话号码	用户需要使用物理键盘在电话号码字段中键入内容时，BlackBerry 设备将切换至数字锁模式，因此用户无需按 Alt 键来键入数字。 您也可以允许用户在电话号码字段中执行以下操作： • 在国际电话号码前键入加号 (+)。 • 键入格式化字符，比如，减号 (-)、句号 (.)、括号 (()) 和空格。 • 键入数字符号 (#) 或星号 (*)。 • 键入逗号 (,) 表示暂停，或键入感叹号(！)表示等待。 • 按 Alt 键及 E、X 或 T 以表明分机号。 用户需要使用虚拟键盘在电话号码字段中键入内容时，将显示数字键盘。 您也可以允许用户在电话号码字段中执行以下操作： • 键入数字符号 (#) 或星号 (*)。 • 按住星号键 (*) 以表明暂停或分机。 • 按住数字符号键 (#) 以表明等待或分机号。
文本	在文本字段中，用户键入文本。 在文本字段中，用户可剪切、复制和粘贴文本。 当光标到达文本行的末尾时，文本就自动换行至下一行。 在文本字段中，BlackBerry 设备也可将电话号码、网页和电子邮件地址自动转为链接。
Web 地址	用户可通过按 Space 键在地址字段中插入句号 (.)。

最佳实践：实施文本字段

- 使用 `TextField` 类创建文本字段。
- 根据您期望用户键入的内容来选择字段类型。 使用最适当的文本字段很重要，以便当用户键入文本时，屏幕上才会显示适当的键入指示符。 例如，当用户在电话号码字段中键入文本时，数字锁指示符会显示在屏幕右上角。 所选择的字段还会影响 BlackBerry 设备(配备 SureType® 技术)上用于字段的默认输入法。
- 尽量使用或扩展现有字段，而不创建自定义字段，以便字段可以继承适当的默认行为。
- 如有必要，请包括提示文本，以向用户提供指导。 如果屏幕空间有限且您不能包括标签或说明性文本，请使用提示文本。 提示文本显示在字段内，当用户键入时将消失。

标签准则

- 使用简洁明了的描述性标签。 避免使用换行的标签。
- 请遵循标题类的首字母大写规则。
- 请使用冒号 (:) 来分隔字段标签。

- 如果您使用提示文本，请使用简洁明了的语句。 请遵循句子类的首字母大写规则。

创建文本字段

创建允许格式的只读文本字段

1. 导入 net.rim.device.api.ui.component.RichTextField 类。
2. 创建 RichTextField 的实例。

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

创建无格式且接受过滤器的可编辑文本字段

1. 导入以下类：
   - net.rim.device.api.ui.component.BasicEditField
   - net.rim.device.api.ui.component.EditField
2. 创建 BasicEditField 的实例。

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

创建允许使用特殊字符的可编辑文本字段

1. 导入 net.rim.device.api.ui.component.EditField 类。
2. 创建 EditField 的实例。

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

创建自动图文集的文本字段

如果文本字段支持自动图文集功能，当用户按 Space 键两次，BlackBerry® 设备就会插入句号，将句号后的下一个字母变成大写，并将单词替换为自动图文集应用程序中定义的单词。

1. 导入以下类：
   - net.rim.device.api.ui.component.AutoTextEditField
   - net.rim.device.api.ui.autotext.AutoText
   - net.rim.device.api.ui.component.BasicEditField
2. 创建 AutoTextEditField 的实例。

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

创建日期字段

1. 导入所需的类和接口。

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

2. 通过扩展 UiApplication 类，创建应用程序框架。 在 main() 中，创建新类的实例并调用 enterEventDispatcher()，以启用应用程序来接收事件。 在应用程序构造器中，调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 MyUiScreen 类表示自定义屏幕。

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

3. 通过扩展 MainScreen 类来创建应用程序的自定义屏幕。 在屏幕构造器中，调用 setTitle() 以指定屏幕的标题。

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

4. 在屏幕构造器中，使用 DateField 类创建日期字段。 提供 System.currentTimeMillis() 作为参数以返回当前时间。 使用 DateField.DATE_TIME 样式同时显示日期和时间。 您可以使用其它样式仅显示日期或仅显示时间。

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

创建密码字段

1. 导入 net.rim.device.api.ui.component.PasswordEditField 类。
2. 创建 PasswordEditField 的实例。
   例如，以下代码示例使用构造器，让您可以提供 PasswordEditField 的默认初始值：

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

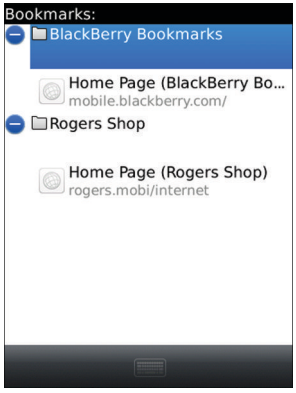
树视图

使用树视图以分层方式显示对象，如文件夹。

在树视图中的对象是节点。 最高节点是根节点。 树视图中的节点下方有子节点。 有子节点的节点称为父节点。

在树视图中，用户可执行以下操作：

操作	仅带触控板的 BlackBerry 设备	带触摸屏和触控板的 BlackBerry 设备
展开或折叠层次结构中带有加号 (+) 或负号 (-) 的对象。	按空格键或单击触控板。	• 点按对象。 • 按空格键。 • 单击触控板。



最佳实践：使用树视图

- 使用 `TreeField` 类可创建树视图。
- 如果用户在单击父节点时可以执行多个操作，请提供弹出菜单。
- 只有当用户需要对整个树执行操作时，才包括根节点。 否则，请排除根节点。

创建字段以显示树视图

使用树视图可按层次结构显示对象，如文件夹结构。 `TreeField` 包含节点。 最高节点是根节点。 树视图中的节点下方有子节点。 有子节点的节点称为父节点。

1. 导入所需的类和接口。

```
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. 实施 TreeFieldCallback 接口。
3. 在 TreeField 对象上调用 TreeField.setExpanded(),以指定文件夹是否可折叠。 创建 TreeField 对象及 TreeField 对象的多个子节点。 调用 TreeField.setExpanded(),其中将 node4 用作参数,以折叠文件夹。

```
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. 要在节点更改时重画 TreeField,请创建实施 TreeFieldCallback 接口的类并实施 TreeFieldCallback.drawTreeItem 方法。 TreeFieldCallback.drawTreeItem 方法将 Cookie 用于树节点,以在节点位置绘制 String。 TreeFieldCallback.drawTreeItem 方法调用 Graphics.drawText(),以绘制 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);
    }
}
```

图像

9

使用已编码图像

通过输入流访问已编码图像

1. 导入所需的类。

```
import java.io.InputStream;
```

2. 将图像保存至项目文件夹或子文件夹。
3. 将图像添加至 BlackBerry® Java® Plug-in Eclipse® 或 BlackBerry® Java® Development Environment 中的项目。
4. 调用 `getClass().getResourceAsStream()` 以作为字节输入流检索图像。

```
private InputStream input;
...
Class _class = this.getClass();
input = _class.getResourceAsStream("/images/example.png");
```

编码图像

1. 导入所需的类。

```
import net.rim.device.api.system.EncodedImage;
```

2. 调用 `EncodedImage.createEncodedImage()`。 此方法使用字节数组中未处理的图像数据创建 `EncodedImage` 实例。
3. 检查是否存在 `IllegalArgumentException`。如果作为参数提供的字节数组不包含已识别的图像格式，`EncodedImage.createEncodedImage()` 会引发此例外。

```
// Store the contents of the image file.
private byte[] data = new byte[2430];
try {
    // Read the image data into the byte array
    input.read(data);
} catch (IOException e) {
    // Handle exception.
}
try {
    EncodedImage image = EncodedImage.createEncodedImage(data, 0, data.length);
} catch (IllegalArgumentException iae) {
    System.out.println("Image format not recognized.");
}
```

显示已编码图像

1. 导入所需的类。

```
import net.rim.device.api.ui.component.BitmapField;
```

2. 调用 BitmapField.setImage() 以将已编码图像分配给 BitmapField。
3. 调用 add() 以将 BitmapField 添加至屏幕。

```
BitmapField field = new BitmapField();
field.setImage(image);
add(field);
```

指定已编码图像的显示大小

1. 导入所需的类。

```
import net.rim.device.api.system.EncodedImage;
```

2. 调用 EncodedImage.scaleImage32(int scaleX, int scaleY)。 传递的缩放值参数必须为 net.rim.device.api.math.Fixed32 数字(0 < 缩放 < 1 表示放大,缩放 > 1 表示缩小)。 scaleImage32() 创建新 EncodedImage,而不是修改当前图像。

指定图像的解码模式

1. 导入所需的类。

```
import net.rim.device.api.system.EncodedImage;
```

2. 调用将以下模式之一用作参数的 EncodedImage.setDecodeMode():
   - 使用 DECODE_ALPHA 解码 alpha 通道(如果存在)(这是默认模式)
   - 使用 DECODE_NATIVE 强制将位图图像解码为 BlackBerry® Device Software 的本地位图图像。
   - 使用 DECODE_READONLY 将已解码位图图像标记为只读。

显示图像以缩放和平移

ZoomScreen 类允许您向图像提供缩放和平移功能。 BlackBerry® 设备用户单击轨迹球或触摸屏时，屏幕显示放大的图像的中心区域。

图像放大时，屏幕还将显示高亮显示缩放区域的叠加。 滚动轨迹球或绕屏幕滑动手指可绕图像平移。 用户停止平移和缩放时，叠加将从屏幕消失。

在附带触摸屏的 BlackBerry 设备上，用户可以定义要缩放的图像区域。 用户可在图像上触摸两点，以便定义区域的斜对角。 区域选定后，用户可以绕图像移动手指，以移动缩放区域。 要放大定义的区域，用户可单击屏幕。 要缩小，用户可按退出键。

代码示例：显示图像以缩放和平移

```
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.extension.container.*;
public class ZoomableImageApp extends UiApplication
{
     public static void main(String[] args)
     {
          ZoomableImageApp theApp = new ZoomableImageApp();
          theApp.enterEventDispatcher();
     }
     public ZoomableImageApp()
     {
          EncodedImage myImg = new EncodedImage("myImg.jpg");
          ZoomScreen zoomableImg = new ZoomScreen(myImg);
          pushScreen(zoomableImg);
     }
}
```

显示图像以缩放和平移

1. 导入所需的类和接口：

```
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.extension.container.*;
```

2. 通过扩展 `UiApplication` 类，创建应用程序框架。 在 `main()` 中，创建新类的实例并调用 `enterEventDispatcher()`，以启用应用程序来接收事件。 在构造器中，使用项目中的资源创建 `EncodedImage`，使用 `EncodedImage` 创建 `ZoomScreen`，然后调用 `pushScreen()`，以便显示要缩放和平移的图像。

```
public class ZoomableImageApp extends UiApplication
{
     public static void main(String[] args)
     {
          ZoomableImageApp theApp = new ZoomableImageApp();
          theApp.enterEventDispatcher();
     }
     public ZoomableImageApp()
     {
          EncodedImage myImg = new EncodedImage("myImg.jpg");
          ZoomScreen zoomableImg = new ZoomScreen(myImg);
          pushScreen(zoomableImg);
     }
}
```

显示一行图像以滚动显示

您可以使用 PictureScrollField 类呈现一水平行的图像，其中用户可通过轨迹球或触摸姿势滚动这些图像。

PictureScrollField 允许您定义选定图像的高亮显示样式、PictureScrollField 的背景样式、选定图像下显示的文本、初始选择图像时显示的工具提示文本以及图像的高度和宽度。

代码示例：显示一行图像以滚动显示

```
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.decor.*;
import net.rim.device.api.ui.extension.component.*;
public class PictureScrollFieldDemo extends UiApplication
{
     public static void main(String[] args)
     {
          PictureScrollFieldDemo app = new PictureScrollFieldDemo();
          app.enterEventDispatcher();
     }
     public PictureScrollFieldDemo()
     {
          pushScreen(new PictureScrollFieldDemoScreen());
     }
}
class PictureScrollFieldDemoScreen extends MainScreen
{
     public PictureScrollFieldDemoScreen()
     {
          setTitle("PictureScrollField Demo");
          Bitmap[] images = new Bitmap[3];
          String[] labels = new String[3];
          String[] tooltips = new String[3];
          images[0] = Bitmap.getBitmapResource("img1.jpg");
          labels[0] = "Label for image 1";
          tooltips[0] = "Tooltip for image 1";
          images[1] = Bitmap.getBitmapResource("img2.jpg");
          labels[1] = "Label for image 2";
          tooltips[1] = "Tooltip for image 2";
          images[2] = Bitmap.getBitmapResource("img3.jpg");
          labels[2] = "Label for image 3";
          tooltips[2] = "Tooltip for image 3";
          ScrollEntry[] entries =  ScrollEntry[3];
          for (int i = 0; i < entries.length; i++)
          {
               entries[i] = new ScrollEntry(images[i], labels[i],tooltips[i]);
          }
          PictureScrollField pictureScrollField = new PictureScrollField(150, 100);
          pictureScrollField.setData(entries, 0);
          pictureScrollField.setHighlightStyle(HighlightStyle.ILLUMINATE);
```

```
            pictureScrollField.setHighlightBorderColor(Color.BLUE);

pictureScrollField.setBackground(BackgroundFactory.createSolidTransparentBackground(C
olor.RED, 150));
            pictureScrollField.setLabelsVisible(true);
            add(pictureScrollField);
      }
}
```

显示一行图像以滚动显示

1. 导入所需的类和接口。

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

2. 通过扩展 `UiApplication` 类，创建应用程序框架。 在 `main()` 中，创建新类的实例并调用 `enterEventDispatcher()`，以启用应用程序来接收事件。 在构造器中，调用 `pushScreen()` 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 `PictureScrollFieldDemoScreen` 类表示自定义屏幕。

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

3. 通过扩展 `MainScreen` 类创建自定义屏幕的框架。

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

4. 在构造器中，调用 `setTitle()` 以设置要在屏幕的标题部分中显示的文本。

```
setTitle("PictureScrollField Demo");
```

5. 在构造器中，创建并初始化三个数组，以存储要在 `PictureScrollField` 中显示的图像以及选择每个图像时显示的标签和工具提示文本。 在此示例中，`PictureScrollField` 包含三个图像。

```
Bitmap[] images = new Bitmap[3];
String[] labels = new String[3];
String[] tooltips = new String[3];
images[0] = Bitmap.getBitmapResource("img1.jpg");
labels[0] = "Label for image 1";
tooltips[0] = "Tooltip for image 1";
images[1] = Bitmap.getBitmapResource("img2.jpg");
labels[1] = "Label for image 2";
tooltips[1] = "Tooltip for image 2";
images[2] = Bitmap.getBitmapResource("img3.jpg");
labels[2] = "Label for image 3";
tooltips[2] = "Tooltip for image 3";
```

6. 在构造器中,创建并初始化 ScrollEntry 对象的数组。 ScrollEntry 表示 PictureScrollField. 中的每个项目

```
ScrollEntry[] entries =  ScrollEntry[3];
for (int i = 0; i < entries.length; i++)
{
     entries[i] = new ScrollEntry(images[i], labels[i], tooltips[i]);
}
```

7. 在构造器中,创建并初始化 PictureScrollField,其中每个图像的宽均为 150 像素,而高均为 100 像素。 调用 setData() 以在 PictureScrollField 中设置条目。 setData() 中的第二个参数指定默认情况下选择的图像位置。

```
PictureScrollField pictureScrollField = new PictureScrollField(150,
     100);
pictureScrollField.setData(entries, 0);
```

8. 在构造器中,设置 PictureScrollField 的样式。 调用 setHighlightStyle() 以指定如何高亮显示选定图像。 调用 setHighlightBorderColor() 以指定选定图像的边框颜色。 调用 setBackground() 以指定 PictureScrollField 的背景。 调用 setLabelVisible() 以指定 PictureScrollField 是否显示选定图像的标签。

```
pictureScrollField.setHighlightStyle(HighlightStyle.ILLUMINATE);
pictureScrollField.setHighlightBorderColor(Color.BLUE);
pictureScrollField.setBackground(BackgroundFactory
    .createSolidTransparentBackground(Color.RED, 150));
pictureScrollField.setLabelsVisible(true);
```

9. 在构造器中,调用 add() 以显示 PictureScrollField。

```
add(pictureScrollField);
```

菜单项

10

创建菜单

MainScreen 类提供 BlackBerry® 设备应用程序的标准组件。 它包括默认菜单。

1. 导入所需的类和接口。

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

2. 通过扩展 UiApplication 类,创建应用程序框架。 在 main() 中,创建新类的实例并调用 enterEventDispatcher(),以启用应用程序来接收事件。 在应用程序构造器中,调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 CreateMenuScreen 类表示自定义屏幕。

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

3. 通过扩展 MainScreen 类来创建应用程序的自定义屏幕。 在屏幕构造器中,调用 setTitle() 以指定屏幕的标题。 调用 add() 以将文本字段添加至屏幕。 调用 addMenuItem() 以将菜单项添加至 MainScreen 创建的菜单。

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

4. 使用 MenuItem 类创建菜单项。 覆盖 run() 以指定用户单击菜单项时发生的操作。 用户单击菜单项时,应用程序将调用 Menu.run()。

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

5. 覆盖 close() 以在用户单击“关闭”菜单项时显示对话框。 默认情况下，“关闭”菜单项将包括在 MainScreen 创建的菜单中。 调用 super.close() 以关闭应用程序。 用户关闭对话框时，应用程序将调用 MainScreen.close() 以关闭应用程序。

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

代码示例：创建菜单

```
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();
    }
}
```

最佳实践：使用菜单

- 请始终提供完整菜单。

- 确保用户按菜单键可打开完整菜单，且可在菜单项高亮显示时调用操作。 确保用户按住菜单键还可打开用于切换应用程序的对话框。
- 对于默认菜单项，请使用用户最可能选择的菜单项。
- 请将默认菜单项和其他常用菜单项放置在菜单中部。
- 请确定菜单项的顺序与其他 BlackBerry® 设备应用程序中菜单项的顺序一致。
- 根据共有用途或共有功能将菜单项分组，并尽量通过用户测试分组情况。
- 在菜单项组之间插入分隔符。
- 不要将用于功能相反的操作的菜单项放在一起。 例如，不要将“删除”菜单项放在“打开”菜单项旁边。
- 始终包括“切换应用程序”和“关闭”菜单项。 将这些菜单项放在菜单的末尾。 如果您使用标准组件，则会自动包括这些菜单项。

标签准则

- 使用简洁明了的描述性标签，其长度不能长于 12 个字符。 如果标签太长，则会显示一个省略号(...)表示文本被截断。
- 使用动词作为标签名称
- 标签要遵循标题类的首字母大写规则。
- 在菜单项标签中使用省略号表明用户单击该菜单项后还须执行另一个操作。 例如，如果用户在日历中单击“转至日期...”菜单项，则他们必须在出现的屏幕上指定一个日期。
- 避免在标签中使用符号，如星号(*)

子菜单

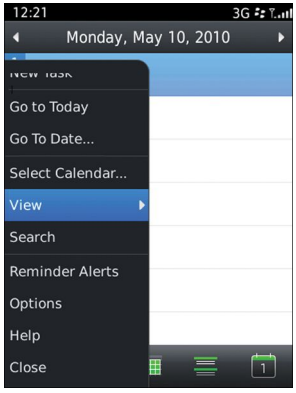
子菜单是一组显示为完整菜单中菜单项的子集的相关菜单项。 通过将项目包括在子菜单中，用户可在完整菜单更为轻易地查找常用项目或重要项目。 子菜单通常包括以下类型的操作：

- 以多种方式发送项目(例如，在电子邮件、彩信或有声明信片中发送图片)
- 以各种方式排序、搜索、查找和过滤项目(例如，按发件人或主题过滤消息)
- 更改视图(例如，日历中的日、周或日程视图)

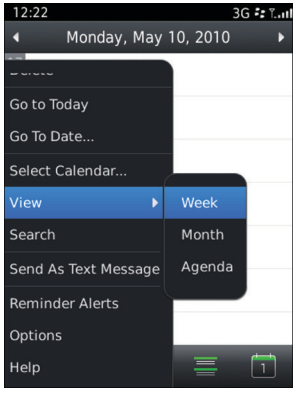
操作	仅带触控板的 BlackBerry 设备	带触摸屏和触控板的 BlackBerry 设备
当菜单项在完整菜单中高亮显示时，将显示子菜单。	• 单击触控板。 • 在触控板上向右移动手指。 • 按菜单键。 • 按输入键。	• 点按菜单项。 • 单击触控板。 • 在触控板上向右移动手指。 • 按菜单键。 • 按输入键。
从子菜单中选择菜单项。	• 单击触控板。 • 按菜单键。 • 按输入键。	• 点按菜单项。 • 单击触控板。 • 按菜单键。 • 按输入键。
关闭子菜单。	• 在触控板上向左移动手指。 • 按退出键。	• 在子菜单外面点按。 • 在触控板上向左移动手指。

操作	仅带触控板的 BlackBerry 设备	带触摸屏和触控板的 BlackBerry 设备
		• 按退出键。
关闭子菜单和完整菜单。	按两次退出键。	• 在完整菜单和子菜单外面点按两次。 • 按两次退出键。 • 打开或关闭滑盖。

当完整菜单中项目有子菜单项可供使用时，将显示一个箭头。



带箭头的完整菜单项



完整菜单和子菜单

最佳实践：使用子菜单

- 使用 Submenu 类可创建子菜单。
- 请使用子菜单来减少完整菜单中的菜单项数量。 例如，如果用户必须滚动才能查看完整菜单中的所有菜单项，请将某些菜单项分成一组放入一个子菜单中。
- 将相关项目分成一组放入一个子菜单中。 例如，如果“排序方式”出现在完整菜单中，请将“日期”、“名称”和“主题”分成一组放入一个子菜单中。
- 考虑将高级功能分成一组放入一个子菜单中。 例如，在完整菜单中使用“其他选项”并将选项分成一组放入子菜单中。
- 避免在只有一个或两个菜单项的菜单中使用子菜单。
- 如果子菜单包括的项多于 6 个，请考虑在子菜单中将这些项分组为两个部分。 将常用菜单项放在子菜单顶部，插入分隔符，然后按字母顺序对其余项进行排序。
- 如果菜单项需要用户执行除单击项之外的更多操作（例如，如果用户必须指定一个日期），请打开对话框。
- 避免在子菜单中包括经常使用的菜单项或重要的菜单项。
- 避免在子菜单中使用子菜单。

标签准则

- 在完整菜单中，使用简洁明了的描述性标签来定义操作。 在完整菜单中，用户应该不需要打开该子菜单即可理解该菜单项的含义。
- 在完整菜单中，请将动词用作标签。 只有当含义明确时才使用名词。 例如，如果“电子邮件帐户”出现在完整菜单中，请将所有电子邮件帐户分成一组放入该子菜单中。
- 避免在子菜单中重复使用动词。 例如，如果完整菜单中使用了“排序方式”，则避免在子菜单中使用“按日期排序”和“按名称排序”。
- 标签要遵循标题类的首字母大写规则。 即使是介词，也将子菜单中第一个词的首字母大写。 例如，使用“Send”>“As Email”，而不是“as Email”。
- 除非菜单项的位置明确指出了“更多”的含义，否则应避免在完整菜单中使用“更多”一项。 例如，如果“收件箱”、“发件箱”和“更多”出现在完整菜单中的分隔符之间，请将与邮箱关联的附加选项分成一组放入该子菜单中。

创建子菜单

您可创建 BlackBerry® 设备应用程序中的子菜单。 用户选择具有关联子菜单的菜单项时，将显示子菜单。

1. 导入所需的类和接口。

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

2. 通过扩展 `UiApplication` 类，创建应用程序框架。 在 `main()` 中，创建新类的实例并调用 `enterEventDispatcher()`，以启用应用程序来接收事件。 在应用程序构造器中，调用 `pushScreen()` 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 `CreateSubmenuScreen` 类表示自定义屏幕。

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

3. 通过扩展 `MainScreen` 类来创建应用程序的自定义屏幕。 在屏幕构造器中，调用 `setTitle()` 以指定屏幕的标题。 调用 `add()` 以将文本字段添加至屏幕。

```
class CreateSubmenuScreen extends MainScreen
{
     public CreateSubmenuScreen()
     {
        setTitle("Create Submenu Sample");
        add(new RichTextField("Create a submenu"));
     }
}
```

4. 使用 MenuItem 类创建子菜单项。 对于每个子菜单项,覆盖 run() 以指定用户单击菜单项时发生的操作。 用户单击菜单项时,应用程序将调用 Menu.run()。

```
private MenuItem _status1 = new MenuItem("Available", 100, 1)
{
    public void run()
    {
         Dialog.inform("I'm available");
    }
};
private MenuItem _status2 = new MenuItem("Unavailable", 200, 2)
{
    public void run()
    {
         Dialog.inform("I'm unavailable");
    }
};
```

5. 覆盖 makeMenu() 以创建应用程序的菜单。 使用 SubMenu 类创建子菜单。 调用 add() 以将子菜单项添加至子菜单。 调用 super.makeMenu() 以创建菜单。

```
protected void makeMenu( Menu menu, int instance )
{
    SubMenu statusSubMenu = new SubMenu(null,"My Status",300,3);
    statusSubMenu.add(_status1);
    statusSubMenu.add(_status2);
    menu.add(statusSubMenu);
    super.makeMenu(menu, instance);
};
```

代码示例：创建子菜单

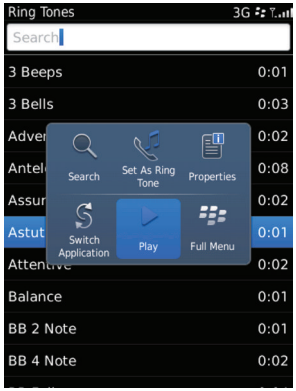
```
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
public class CreateSubmenu extends UiApplication
{
     public static void main(String[] args)
     {
         CreateSubmenu theApp = new CreateSubmenu();
         theApp.enterEventDispatcher();
     }
     public CreateSubmenu()
     {
         pushScreen(new CreateSubmenuScreen());
     }
}
class CreateSubmenuScreen extends MainScreen
{
    public CreateSubmenuScreen()
    {
      setTitle("Create Submenu Sample");
      add(new RichTextField("Create a submenu"));
    }
```

```
    private MenuItem _status1 = new MenuItem("Available", 100, 1)
    {
        public void run()
        {
            Dialog.inform("I'm available");
        }
    };
    private MenuItem _status2 = new MenuItem("Unavailable", 200, 2)
    {
        public void run()
        {
            Dialog.inform("I'm unavailable");
        }
    };
    protected void makeMenu( Menu menu, int instance )
    {
        SubMenu statusSubMenu = new SubMenu(null,"My Status",300,3);
        statusSubMenu.add(_status1);
        statusSubMenu.add(_status2);
        menu.add(statusSubMenu);
        super.makeMenu(menu, instance);
    };
}
```

弹出菜单

弹出菜单向用户提供访问高亮显示项目的最常用操作的快速方式。 如果高亮显示项目具有多个关联操作，或高亮显示项目没有任何关联的主要操作，也可以使用弹出菜单。 您可以创建包含 9 个操作、6 个操作或 3 个操作的弹出菜单。

操作	仅带触控板的 BlackBerry 设备	带触摸屏和触控板的 BlackBerry 设备
打开弹出菜单。	• 单击触控板。 • 如果已为某个项目（如打开消息）分配了主要操作，用户可单击并按住触控板，以打开弹出菜单。	• 点按项目。 • 单击触控板。 • 如果已为某个项目（如打开消息）分配了主要操作，用户可单击并按住触控板或触按触摸屏并按住手指，以打开弹出菜单。
从弹出菜单中选择项目。	• 单击触控板。 • 按输入键。	• 点按项目。 • 单击触控板。 • 按输入键。
关闭弹出菜单。	按退出键。	• 在弹出菜单外点按。 • 按退出键。 • 打开或关闭滑盖。



弹出菜单将替换上下文菜单或快捷菜单。 任何现有上下文菜单都将自动转换为弹出菜单。

最佳实践：实施弹出菜单

- 使用弹出菜单，而不使用上下文菜单（快捷菜单）。 确保弹出菜单可向用户提供值。
- 将用户最有可能选择的菜单项设置为默认菜单项。 弹出菜单中的默认项应该与完整菜单中的默认项相同。
- 仅在弹出菜单中包括高亮显示项目的最常用操作。
- 在弹出菜单中包括每项的图标和标签。 使用平均大小 33 x 33 像素创建图标。 这些图标显示在 60 x 40 像素的画布上，应该有一些实体周围的空间。

在弹出菜单中放置项的准则

- 将默认菜单项放在弹出菜单的中部。
- 根据以下编号位置按最常用到最不常用的顺序对其余项进行排序。 通过将操作顺序与其他 BlackBerry® 设备应用程序中的操作顺序保持一致，尝试利用用户的肌肉记忆。

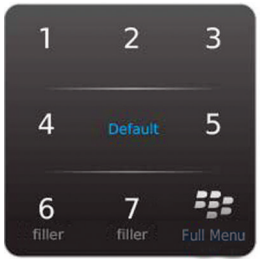
- 如果动态填充位置，则不要显示不可用的菜单项。 允许可用菜单项改变位置。
- 如果无足够操作可填充菜单，请使用较小菜单。 如果需要填充一个或两个位置，则包括有用操作，如“复制”或“搜索”。 如果未填充某个位置，则显示“切换应用程序”或“主页”。

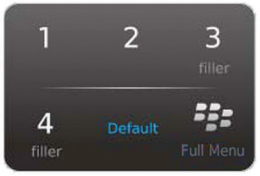
关于在弹出菜单中放置项

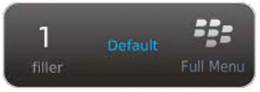
弹出菜单将采用 3 x 3、3 x 2 或 3 x 1 网格显示菜单项，具体取决于项数。 默认情况下，弹出菜单始终包括用于打开完整菜单的项。 您最多可提供 8 个额外项。 如果提供的项多于 8 个，将不会显示额外项。您可将额外项添加至完整菜单。

添加至 `CommandItem` 元素矢量的第一项是弹出菜单中的默认项。 在 BlackBerry® 设备用户打开弹出菜单时，默认项将高亮显示在网格中心。 其他项将根据添加顺序依次显示在弹出菜单中。

如果添加至弹出菜单的项数将在网格中留下空单元格，则在菜单中添加填充项。 第一个填充项是用于切换应用程序的选项。 第二个填充项时用于返回主屏幕的选项。







支持原有上下文菜单

自 BlackBerry® Java® SDK 6.0 开始，应用程序的上下文菜单或快捷菜单将转换为弹出菜单。 但是，不管您是将菜单项添加至上下文菜单还是默认使用添加至上下文菜单的项，都无需更改代码来将这些项显示在弹出菜单上。 但是，BlackBerry Java SDK 6.0 包括可用于构建弹出菜单的类和接口，并允许您使用命令框架 API 让代码更加模块化。 例如，如果您的应用程序具有 UI 组件和执行同一功能的菜单项，则通过使用命令框架 API，只需为该功能编写编码一次，即可在整个应用程序中使用它并将其用于其他应用程序。

上下文菜单项包括确定项在菜单中的位置的序数。 序数越小，项在菜单中的位置越高。 将上下文菜单转换为弹出菜单时，序数最小的菜单项将成为默认弹出菜单项。 其余项将按照序数最小到最大的顺序添加至弹出菜单。 如果图标与原有上下文菜单中的菜单项关联，则将其用于弹出菜单项。 否则，使用默认图标。与其他弹出菜单类似，仅将前 8 个上下文菜单显示在弹出菜单中。 如果上下文菜单的菜单项多于 8 个，请确保将相应序数用于菜单项，以便将最重要和最常用的项显示在弹出菜单中。

有关创建上下文菜单的详细信息，请访问 *BlackBerry 开发人员资源中心*（www.blackberry.com/developer）以阅读区分完整菜单和主要操作菜单。

创建弹出菜单

通过使用 `net.rim.device.api.ui.menu` 数据包中提供的类，可创建弹出菜单。通过使用命令框架 API，可定义弹出菜单项的功能。

弹出菜单包含上下文菜单提供程序、命令项提供程序和命令项。

组件	说明
上下文菜单提供程序	您可使用 DefaultContextMenuProvider 类创建并显示屏幕的弹出菜单。 在 BlackBerry® 设备用户打开弹出菜单时，上下文菜单提供程序将查找屏幕上作为命令项提供程序的字段。DefaultContextMenuProvider 是 ContextMenuProvider 的默认实施。 如果未向屏幕提供上下文菜单提供程序，将转换原有上下文菜单，并将其显示为弹出菜单。
命令项提供程序	您可使用 CommandItemProvider 类在 UI 屏幕上配置字段，以便该字段根据当前上下文向弹出菜单提供上下文和命令项。 例如，可将电子邮件地址配置为命令项提供程序，以便根据该电子邮件地址是否位于用户的联系人列表来向用户提供不同的操作。
命令项	您可使用 CommandItem 类指定弹出菜单项的文本、图标和行为。 通过使用命令，可定义弹出菜单的行为。 此命令将用作命令处理程序（定义弹出菜单项的功能）的实例的代理。 例如，对于电子邮件地址，可提供命令项，以便根据选定电子邮件地址是否显示在用户的联系人列表中添加或查看联系人。 命令处理程序将包含用于添加或查看联系人的代码。 有关命令和命令处理程序的详细信息，请参阅命令框架 API。BlackBerry® Java® SDK 中提供的“命令框架”示例应用程序演示命令和命令处理程序。

如果使用原有上下文菜单，BlackBerry® Device Software 6.0 会将上下文菜单项转换为命令项，并将其提供给命令项提供程序。 有关详细信息，请参阅支持原有上下文菜单。

创建弹出菜单

1. 导入所需的类和接口。

```
import net.rim.device.api.command.*;
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.menu.*;
import net.rim.device.api.ui.image.*;
import net.rim.device.api.util.*;
import java.util.*;
```

2. 通过扩展 UiApplication 类，创建应用程序框架。 在 main() 中，创建新类的实例并调用 enterEventDispatcher()，以启用应用程序来接收事件。 在应用程序构造器中，调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 MyPopUpMenuScreen 类表示自定义屏幕。

```
public class MyPopUpMenuApp extends UiApplication
{
    public static void main(String[] args)
    {
```

```
        Mypop-upMenuApp theApp = new Mypop-upMenuApp();
        theApp.enterEventDispatcher();
    }
    public Mypop-upMenuApp()
    {
        pushScreen(new Mypop-upMenuScreen());
    }
}
```

3. 通过扩展 MainScreen 类来创建应用程序的自定义屏幕。 在屏幕构造器中，调用 setTitle() 以指定屏幕的标题。

```
class MyPopUpMenuScreen extends MainScreen
{
    EmailAddressEditField emailAddress;
    public Mypop-upMenuScreen()
    {
        setTitle("Pop-Up Menu Demo");
    }
}
```

4. 在屏幕构造器中，指定上下文菜单提供程序。 将 DefaultContextMenuProvider 对象传递至 Screen.setContextMenuProvider()，以让屏幕显示弹出菜单。

```
setContextMenuProvider(new DefaultContextMenuProvider());
```

5. 在屏幕构造器中，创建可调用弹出菜单的 UI 组件。 在以下代码示例中，标签使用 Field.FOCUSABLE 属性允许用户高亮显示该字段。

```
LabelField labelField = new LabelField("Click to invoke pop-up menu",
     Field.FOCUSABLE);
emailAddress = new EmailAddressEditField("Email address: ", "name@blackberry.com",
     40);
```

6. 在屏幕构造器中，将 UI 组件配置为命令项提供程序。 DefaultContextMenuProvider 对象将查找配置为命令项提供程序的字段，并将其用于创建和显示弹出菜单。 对于每个组件，调用 Field.setCommandItemProvider() 以将字段配置为命令项提供程序。 ItemProvider 类在步骤 7 中介绍。

```
ItemProvider itemProvider = new ItemProvider();
labelField.setCommandItemProvider(itemProvider);
emailAddress.setCommandItemProvider(itemProvider);
```

7. 在自定义屏幕中，实施 CommandItemProvider 接口。 在 getContext() 中，返回配置为命令项提供程序的字段。 在 getItems() 中，创建要将弹出菜单项添加至的 Vector 对象。

```
class ItemProvider implements CommandItemProvider
{
    public Object getContext(Field field)
    {
        return field;
    }
    public Vector getItems(Field field)
```

```
    {
    }
}
```

8. 在 getItems() 中，通过创建 CommandItem 类的实例，提供弹出菜单项。 对于每个菜单项，指定弹出菜单文本、图标和命令。 在以下代码示例中，在创建 CommandItem 之前，使用 if-then-else 语句检查哪个组件调用了弹出菜单。 此命令是扩展抽象 CommandHandler 类（在步骤 9 中介绍）的类的实例的代理。 调用 Vector.addElement() 以将弹出菜单项添加至 Vector。 返回 Vector。

```
CommandItem defaultCmd;
Image myIcon = ImageFactory.createImage(Bitmap.getBitmapResource("my_logo.png"));
if(field.equals(emailAddress)){
    defaultCmd = new CommandItem(new StringProvider("Email Address"), myIcon, new
     Command(new DialogCommandHandler()));
}
else {
    defaultCmd = new CommandItem(new StringProvider("Label Field"), myIcon, new
     Command(new DialogCommandHandler()));
}
items.addElement(defaultCmd);
return items;
```

9. 在自定义屏幕中，通过创建扩展抽象 CommandHandler 类的类，创建命令处理程序。 在 execute() 中，定义要与弹出菜单项关联的功能。 屏幕上的任何 UI 组件都可使用此命令处理程序（例如，完整菜单项、按钮等）。 由于未实施 canExecute()，因此此命令始终可执行。 在以下代码示例中，在用户单击弹出菜单项时，将显示对话框。

```
class DialogCommandHandler extends CommandHandler
{
    public void execute(ReadOnlyCommandMetadata metadata, Object context)
    {
        Dialog.alert("Executing command for " + context.toString());
    }
}
```

代码示例：创建弹出菜单

```
import net.rim.device.api.command.*;
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.menu.*;
import net.rim.device.api.ui.image.*;
import net.rim.device.api.util.*;
import java.util.*;
public class MyPopUpMenuApp extends UiApplication
{
    public static void main(String[] args)
    {
        MyPopUpMenuApp theApp = new MyPopUpMenuApp();
        theApp.enterEventDispatcher();
    }
    public MyPopUpMenuApp()
```

```
    {
        pushScreen(new MyPopUpMenuScreen());
    }
}
class MyPopUpMenuScreen extends MainScreen
{
    EmailAddressEditField emailAddress;
    public MyPopUpMenuScreen()
    {
        setTitle("Pop-Up Menu Demo");
        setContextMenuProvider(new DefaultContextMenuProvider());
        LabelField labelField = new LabelField("Click to invoke pop-up menu",
     Field.FOCUSABLE);
        emailAddress = new EmailAddressEditField("Email address: ",
     "name@blackberry.com", 40);
        ItemProvider itemProvider = new ItemProvider();
        labelField.setCommandItemProvider(itemProvider);
        emailAddress.setCommandItemProvider(itemProvider);
        add(labelField);
        add(emailAddress);
    }
    /* To override the default functionality that prompts the user to save changes
     before the application closes,
     * 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.
     */
    protected boolean onSavePrompt()
    {
        return true;
    }
    class ItemProvider implements CommandItemProvider
    {
        public Object getContext(Field field)
        {
            return field;
        }
        public Vector getItems(Field field)
        {
            Vector items = new Vector();
            CommandItem defaultCmd;
            Image myIcon = ImageFactory.createImage(Bitmap
    .getBitmapResource("my_logo.png"));
            if(field.equals(emailAddress)){
                 defaultCmd = new CommandItem(new StringProvider("Email Address"),
     myIcon, new Command(new DialogCommandHandler()));
            }
            else{
                defaultCmd = new CommandItem(new StringProvider("Label Field"),
     myIcon, new Command(new DialogCommandHandler()));
            }
            items.addElement(defaultCmd);
            return items;
        }
    }
    class DialogCommandHandler extends CommandHandler
```

```
    {
        public void execute(ReadOnlyCommandMetadata metadata, Object context)
        {
            Dialog.alert("Executing command for " + context.toString());
        }
    }
}
```

将菜单项添加至 BlackBerry Device Software 应用程序

通过使用 net.rim.blackberry.api.menuitem 数据包中的菜单项 API，可将菜单项添加至 BlackBerry® Device Software 应用程序。 例如，可将名为“查看销售订单”的菜单项添加至 BlackBerry 设备上的联系人应用程序，以便在用户单击菜单项时，CRM 应用程序将打开并显示该联系人的销售订单列表。

ApplicationMenuItemRepository 类提供的常量指定菜单项在其中显示的 BlackBerry Device Software 应用程序。 例如，MENUITEM_MESSAGE_LIST 常量指定菜单项应显示在消息应用程序中。

将菜单项添加至 BlackBerry Device Software 应用程序

1. 导入所需的类和接口。

```
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. 扩展抽象 ApplicationMenuItem 类以创建菜单项。 使用整数覆盖 ApplicationMenuItem() 构造器，以指定菜单项在菜单中的位置。 数字越大，表示菜单项在菜单中的位置越低。

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

3. 实施 toString() 以指定菜单项文本。

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

4. 调用 getInstance() 以检索应用程序库。

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

5. 创建类的实例以扩展 MenuItem 类。

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

6. 调用 ApplicationMenuItemRepository.addMenuItem() 以将菜单项添加至相关 BlackBerry® 设备应用程序库。

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

7. 实施 run() 以指定菜单项的行为。 在以下代码示例中，在用户单击新菜单项且 Contact 对象存在时，ContactsDemo 应用程序将接收事件并调用 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;
}
```

更改菜单的外观

通过使用 net.rim.device.api.ui.component 数据包中的 Menu 类，可更改菜单的背景、边框和字体。例如，可更改菜单的外观，以便使其具有与 BlackBerry® 设备应用程序的其余部分类似的观感。 更改菜单的外观时，BlackBerry 设备应用程序将覆盖在 BlackBerry 设备上设置的主题。

更改菜单的外观

1. 导入所需的类和接口。

```
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. 通过扩展 UiApplication 类，创建应用程序框架。 在 main() 中，创建新类的实例并调用 enterEventDispatcher()，以启用应用程序来接收事件。 在应用程序构造器中，调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 CreateCustomMenuScreen 类表示自定义屏幕。

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

```
    public CreateCustomMenu()
    {
        pushScreen(new CreateCustomMenuScreen());
    }
}
```

3. 通过扩展 MainScreen 类来创建应用程序的自定义屏幕。 在屏幕构造器中，调用 setTitle() 以指定屏幕的标题。 调用 add() 以在屏幕上添加文本字段。

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

4. 在屏幕构造器中，指定菜单的外观。 通过创建 XYEdges 对象，创建菜单周围的边框间距。 调用 createRoundedBorder() 以创建圆角边框。 调用 createSolidTransparentBackground() 以为菜单创建透明背景色。

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

5. 在屏幕构造器中，使用 FontFamily 对象指定菜单的字体。 调用 forName() 以从 BlackBerry® 设备上检索字体。 调用 getFont() 以指定字体的样式和大小。

```
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. 在 Screen 类中，覆盖 makeMenu() 以应用菜单的外观。 调用 setBackground()、setBorder() 和 setFont()，以应用在步骤 4 和 5 中指定的菜单外观。

```
protected void makeMenu(Menu menu, int context)
{
    menu.setBorder(_menuBorder);
```

```
        menu.setBackground(_menuBackground);
        menu.setFont(_menuFont);
        super.makeMenu(menu, context);
    }
```

代码示例：更改菜单的外观

```
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);
    }
}
```

自定义字体

11

net.rim.device.api.ui 数据包中的 FontManager 类提供的常量和方法可用于安装和卸载 BlackBerry® 设备上的 TrueType 字体。 TrueType 字体文件允许的最大大小是 60 KB。 您可以指定字体是可用于安装了该字体的应用程序，还是可用于 BlackBerry 设备上的所有应用程序。

FontManager 类还提供用于设置 BlackBerry 设备或应用程序的默认字体的方法。

在 BlackBerry Java 应用程序中安装和使用自定义字体

1. 导入所需的类和接口。

```
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. 通过扩展 UiApplication 类，创建应用程序框架。 在 main() 中，创建新类的实例并调用 enterEventDispatcher()，以启用应用程序来接收事件。 在应用程序构造器中，调用 pushScreen() 以显示应用程序的自定义屏幕。 步骤 3 中介绍的 FontLoadingDemoScreen 类表示自定义屏幕。

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

3. 通过扩展 MainScreen 类来创建自定义屏幕。 调用 setTitle() 以设置要在屏幕的标题部分中显示的文本。 创建新 LabelField 对象。 您可将自定义字体应用于此对象。

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

4. 在屏幕构造器中，调用 FontManager.getInstance() 方法以获取 FontManager 对象的引用，然后调用 load() 方法以安装字体。 将 load() 调用封装在 IF 语句中，以检查安装是否成功。 load() 方法将返回指定字体是否安装成功的标记。 以下代码指定该字体只能由应用程序使用。

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

5. 在屏幕构造器中，在步骤 5 中创建的 IF 语句的 try/catch 块中，为刚安装的字体创建 Font 对象。 调用 setFont() 方法以将该字体应用于在步骤 5 中创建的 LabelField。

```
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. 在屏幕构造器中，调用 add() 以将 LabelField 添加至屏幕。

```
add(helloWorld);
```

代码示例：在 BlackBerry Java Application 中安装和使用自定义字体

```
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);
    }
}
```

拼写检查器

12

您可以使用 net.rim.blackberry.api.spellcheck 数据包中的项目将拼写检查器功能添加至应用程序。SpellCheckEngine 接口可让应用程序检查 UI 字段值的拼写，并为 BlackBerry® 设备用户提供拼写更正选项。 SpellCheckUI 接口让应用程序可通过与 SpellCheckEngine 实施交互来提供 UI，以便让 BlackBerry 设备用户可解决拼写问题。

有关使用拼写检查 API 的详细信息，请参阅 BlackBerry® Java® Development Environment 4.3.1 或更高版本及 BlackBerry® Java® Plug-in for Eclipse® 附带的“拼写检查”示例应用程序。

添加拼写检查功能

1. 导入以下类：
   - net.rim.blackberry.api.spellcheck.SpellCheckEngineFactory
   - java.lang.StringBuffer
2. 导入以下接口：
   - net.rim.blackberry.api.spellcheck.SpellCheckEngine
   - net.rim.blackberry.api.spellcheck.SpellCheckUI
   - net.rim.blackberry.api.spellcheck.SpellCheckUIListener
3. 创建拼写检查对象的变量。

```
SpellCheckEngine _spellCheckEngine;
SpellCheckUI _spellCheckUI;
```

4. 调用 createSpellCheckUI()。

```
_spellCheckUI = SpellCheckEngineFactory.createSpellCheckUI();
```

5. 要在发生拼写检查事件时通知应用程序，请调用 addSpellCheckUIListener()，其中 SpellCheckUIListener 对象用作参数。

```
_spellCheckUI.addSpellCheckUIListener(new SpellCheckUIListener());
```

6. 要让应用程序对 UI 字段进行拼写检查并向 BlackBerry 设备用户提供拼写更正建议，请获取 SpellCheckEngine 对象并调用 getSpellCheckEngine()。

```
_spellCheckEngine = _spellCheckUI.getSpellCheckEngine();
```

7. 要使用拼写错误的词的更正，请调用 SpellCheckEngine.learnCorrection()。 使用参数 new StringBuffer(text) 和 new StringBuffer(correction)，其中 text 表示拼写错误的词，而 correction 表示正确的词。

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

8. 要在字段上执行拼写检查操作，请调用 SpellCheckUI.spellCheck()，其中将 field 用作参数。

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

9. 要将拼写错误的词接受作为拼写正确的词，请调用 SpellCheckEngine.learnWord()，其中将要学习的词用作参数。

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

监听拼写检查事件

1. 导入以下类：
   - java.lang.StringBuffer
   - net.rim.device.api.ui.UiApplication
   - net.rim.device.api.ui.Field
2. 导入以下接口：
   - net.rim.blackberry.api.spellcheck.SpellCheckUIListener
   - net.rim.blackberry.api.spellcheck.SpellCheckEngine
3. 创建在 SpellCheckEngine 学习新词时返回 SpellCheckUIListener.LEARNING_ACCEPT 常量的方法。

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

4. 创建在 SpellCheckEngine 学习词更正时返回 SpellCheckUIListener.LEARNING_ACCEPT 常量的方法。

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

5. 创建在 SpellCheckEngine 找到拼写错误的词时返回 SpellCheckUIListener.ACTION_OPEN_UI 常量的方法。

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

相关资源

13

- www.blackberry.com/go/apiref：查看 BlackBerry® Java® SDK API 参考的最新版本。
- www.blackberry.com/go/devguides：查找 BlackBerry Java SDK 的开发指南、发布说明和示例应用程序概述。
- www.blackberry.com/developers：访问 BlackBerry® Developer Zone，了解 BlackBerry 设备应用程序开发资源。
- www.blackberry.com/go/developerkb：查看有关 BlackBerry 开发知识库的知识库文章。
- www.blackberry.com/developers/downloads：查找用于开发 BlackBerry 设备应用程序的最新开发工具和下载。

词汇表

14

3-D
三维

API
Application Programming Interface（应用程序编程接口）

JVM
Java® Virtual Machine（Java® 虚拟机）

MIDP
Mobile Information Device Profile（移动信息设备配置文件）

提供反馈

15

要提供关于此交付项目的反馈，请访问 www.blackberry.com/docsfeedback。

文档修订历史记录

16

日期	说明
2010 年 8 月 16 日	已删除以下主题： • 创建进度指示符
2010 年 8 月 3 日	已添加以下主题： • 通过输入流访问已编码图像 • 关于在弹出菜单中放置项 • 最佳实践：实施弹出菜单 • 最佳实践：使用子菜单 • 命令框架 API • 创建弹出菜单 • 创建弹出菜单 • 创建子菜单 • 将标签显示在屏幕上的绝对位置 • 显示已编码图像 • 显示图像以缩放和平移 • 显示一行图像以滚动显示 • 将字段显示在屏幕上的绝对位置 • 显示图像以缩放和平移 • 显示一行图像以滚动显示 • 启用多点触控姿势 • 启用用户在触控板上执行的滑动姿势 • 编码图像 • 图像 • 指示活动 • 指示进度 • 指示活动或任务进度 • 多点触控姿势 • 弹出菜单 • 指定图像的解码模式 • 指定已编码图像的显示大小 • 子菜单 • 支持原有上下文菜单 • 触控板的滑动姿势 • 触摸屏交互模型 • 将命令用于一个或多个应用程序 • 将命令用于一个 UI 组件

日期	说明
	• 使用已编码图像 已添加以下代码示例： • 代码示例：创建弹出菜单 • 代码示例：创建子菜单 • 代码示例：显示一行图像以滚动显示 • 代码示例：将标签显示在屏幕上的绝对位置 • 代码示例：显示图像以缩放和平移 已更改以下主题： • 活动指示符和进度指示符 • 按钮 • 创建与 BlackBerry 标准 UI 一致的 UI • 对话框 • 下拉列表 • 列表和表 • 选择器 • 单选按钮 • 搜索 • 旋转框 • 文本字段 • 树视图 已删除以下主题： • 将图标添加至菜单项 • 将图标添加至菜单项 • 代码示例：将图标添加至菜单项

法律声明

17

©2011 Research In Motion Limited。 保留所有权利。 BlackBerry®、RIM®、Research In Motion® 以及相关商标、名称和徽标均为 Research In Motion Limited 的专有财产，并且已在美国和全球其他 国家（地区）注册和/或使用。

Eclipse 是 Eclipse Foundation, Inc. 的商标。。Java 是 Oracle America, Inc. 的商标。。TrueType 是 Apple Inc. 的商标。。所有其他商标均为其各自所有者的财产。

本文档包括所有加入包含参考内容的文档，如提供的说明文档或 www.blackberry.com/go/docs 提供的文档，以“原样”和“可提供性”提供并可访问，不具备 Research In Motion Limited 及其附属公司 ("RIM") 的条件、背书、保证、陈述或任何种类的担保，同时 RIM 对本文档中的任何印刷、技术或其它错误、遗漏不承担任何责任。 为了保护 RIM 的所有权以及机密信息和/或商业秘密，本说明文档可能会以普通术语介绍 RIM 技术的某些方面。 RIM 保留定期更改此说明文档中信息的权利；但 RIM 不承诺及时向您提供对此说明文档的更改、更新、改进或其它添加内容，并可能完全不提供。

本文档可能包含对第三方信息来源、硬件或软件、产品或服务，包括组件和内容，如受版权和/或第三方网站（统称为“第三方产品和服务”）所保护内容的引用。 对于任何第三方产品和服务，包括但不限于内容、准确性、版权符合性、兼容性、性能、可靠性、合法性、适当性、链接或任何其他方面的第三方产品和服务，RIM 不控制且不承担任何责任。 在本文档中包括对第三方产品和服务的引用并不表示 RIM 认可第三方产品和服务或以任何方式认可第三方。

除当地司法机关禁止的特定范围外，本文档中提及的任何明示或暗示的条件、认可、保证、陈述或任何种类的担保，包括无限制、任何条件、认可、保证、陈述或耐用性担保、适用于某特定目的、适销性、可销售品质、非侵权性、满意质量，或所有权、法令引起、第三方、交易过程、交易用途，或与文档及其用途相关的、任何软件、硬件、服务或任何第三方产品和服务的履行或不履行均排除在外。 您可能还具有按州或省份区分的其他权利。 某些司法机关可能不允许排除和限制暗示的担保和条件。 除法律允许外，如果无法按上述条件排除但可限制的任何与本文档相关的暗示担保或条件，可将其限制为在您初次获得作为索赔主因的文档或项目之日起九十（90）天内生效。

除当地司法机关适用法律允许的最大范围外，对文档及其用途的任何类型损坏，或本文提及的任何软件、硬件、服务、任何第三方产品和服务的履行或不履行，包括不限于以下任何损坏：直接的、后果性的、惩戒性的、伴随的、间接的、特殊的、惩罚性的或严重的损坏，利润后收入的损失，未实现预计的盈利，业务中断，商业信息损失，商业机会损失，数据损坏或丢失，无法传输或接收任何数据，与组合 RIM 产品或服务一起使用的任何应用程序相关的问题，停工时间成本，无法使用 RIM 产品或服务或任何及其任何部分或任何开播服务，替换商品成本，包装、设备或服务成本，资本成本或其他类似财务损失，无论此类损坏可预见或不可预见，或者被告知存在损失的可能，RIM 概不承担任何责任。

除当地司法机关适用法律允许的最大范围外，RIM 对合同、侵权行为或包括任何过失责任或严格赔偿责任在内的其他行为概不承担任何义务和责任。

本文档包含的限制、排除事项和免责声明应适用于：(A) 不考虑操作、需求或用户操作的原因性质，包括但不限于违约、疏忽、侵权行为、严格赔偿责任或任何其他法律理论且应克服根本性违约、违约、此协议基本目的失败、或内含的任何补救措施；和 (B) RIM 及其附属公司，其继任人、分配、代理、供应商（包括开播服务提供商）、授权 RIM 分销商（也包括开播服务提供商）及其董事、雇员和独立承包商。

除上述限制和排除事项外，RIM 及其附属公司的任何董事、雇员、代理、分销商、供应商、独立承包商对由本文档引起或相关的事故概不承担任何责任。

在订购、安装或使用任何第三方产品和服务前，用户有责任确保其开播服务提供商已同意支持所有功能。 某些无线服务提供商可能不会在订购 BlackBerry® Internet Service 时提供 Internet 浏览功能。 请与服务提供商联系，以了解可用性、漫游安排计划、服务计划和功能。 安装或使用具有 RIM 产品和服务的第三方产品和服务可能会要求一个或多个专利、商标、版权或其他许可证以避免侵害或违反第三方权利。 您应独自负责确定是否使用第三方产品和服务，如果任何第三方许可证要求如此。 如果有此要求，则您有责任获取这些许可证。 除非已获取所有必需的许可证，否则您不应安装或使用第三方产品和服务。 对于为了方便而随 RIM 产品和服务一起提供的和按“原样”形式（不具有 RIM 所做的任何种类的明示或暗示条件、认可、保证、陈述或担保）提供的任何第三方产品和服务，RIM 概不承担任何责任。 除了许可证已清楚表明或与 RIM 签订的其他协议，您使用第三方产品和服务应该受您同意这些产品或服务的单独许可证和其他第三方适用协议条款所约束。

本说明文档中介绍的某些功能可能需要安装最低版本的 BlackBerry® Enterprise Server、BlackBerry® Desktop Software 和/或 BlackBerry® Device Software。

此外已在单独的许可证或 RIM 适用的其他协议中陈述了使用任何 RIM 产品或服务的条款。 对于除本文档之外任何部分的 RIM 产品或服务，本文档中的任何内容不得用于代替由 RIM 提供的任何明确书面协议或担保。

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

加拿大出版