CHAPTER 9

MIDP User Interface Libraries

In light of the wide variations in cellular phones and other MIDP target devices, the requirements for user interface support are very challenging. MIDP target devices differ from desktop systems in many ways, especially in how the user interacts with them. The following requirements must be kept in mind when designing a user interface library for mobile information devices:

- The devices and applications should be useful to users who are not necessarily experts in using computers.
- The devices and applications should be useful in situations where the user cannot pay full attention to the application. For example, many cellular phones and other wireless devices are commonly operated with one hand while the user is driving a car, cooking, fishing, skiing, and so forth.
- The form factors and user interface concepts of MIDP target devices vary considerably between devices, especially compared to desktop systems. For example, display sizes are much smaller, and input devices do not always include pointing devices.
- The Java applications for mobile information devices should have user interfaces that are compatible with the native applications so that the user finds them intuitive to use.

Given these requirements and the capabilities of devices that implement the MIDP (see Section 3.2, "Target Devices"), the MIDP expert group decided that neither the Abstract Windowing Toolkit (AWT) provided by Java™ 2 Standard Edition nor a subset would meet the requirements. Reasons for this decision include:

- AWT was designed and optimized for desktop computers. Desktop requirements and assumptions are not appropriate for smaller screens.
- When a user interacts with AWT, event objects are created dynamically. These objects are short-lived and exist only until each associated event is processed by the system. The event object then becomes garbage and must be reclaimed by the system's garbage collector. The limited CPU and memory subsystems of a mobile information device (MID) typically cannot afford the overhead of unnecessary garbage objects.
- AWT has a rich but desktop-oriented feature set. This feature set includes support for features not found on MIDs. For example, AWT has extensive support for window management (such as overlapping windows, window resize, and so forth). MIDs have small displays that are not large enough for multiple overlapping windows. Limited display size also makes window resizing impractical. As such, the windowing and layout manager support within AWT is not required for MIDs.
- AWT assumes certain desktop user interaction models. The component set of AWT was designed to work with a *pointer device* (for instance, a mouse or pen input). As mentioned earlier, this assumption is valid for a small subset of MIDs, since many of these devices have only a keypad for user input.

9.1 Structure of the MIDP User Interface API

9.1.1 Screen model

The central abstraction of the MIDP user interface is the *screen*. Simple screens help organize the user interface into manageable pieces. This results in user interfaces that are easy to use and learn. Each MIDP application has a `Display` on which a single screen is shown. The application sets and resets the current screen on the `Display` for each step of the task, based on user interactions. The application is notified of commands selected by the user and changes the screen as necessary. The device software manages the sharing of the physical display between the native applications and the MIDP applications.

The rationale behind the screen-oriented approach is based on the wide variations in display and keypad configurations found in MIDP devices. Each device provides a consistent look and feel by handling the component layout, painting, scrolling, and focus traversal. If an application needed to be aware of these details, portability would be difficult to achieve, and smooth integration with the look and

feel of the device and its native applications would place a heavy burden on application developers.

Each `Screen` (technically, each `Displayable` object) is a functional user interface element that encapsulates device-specific graphics rendering and user input handling. Figure 9.1 shows the hierarchy of the classes.

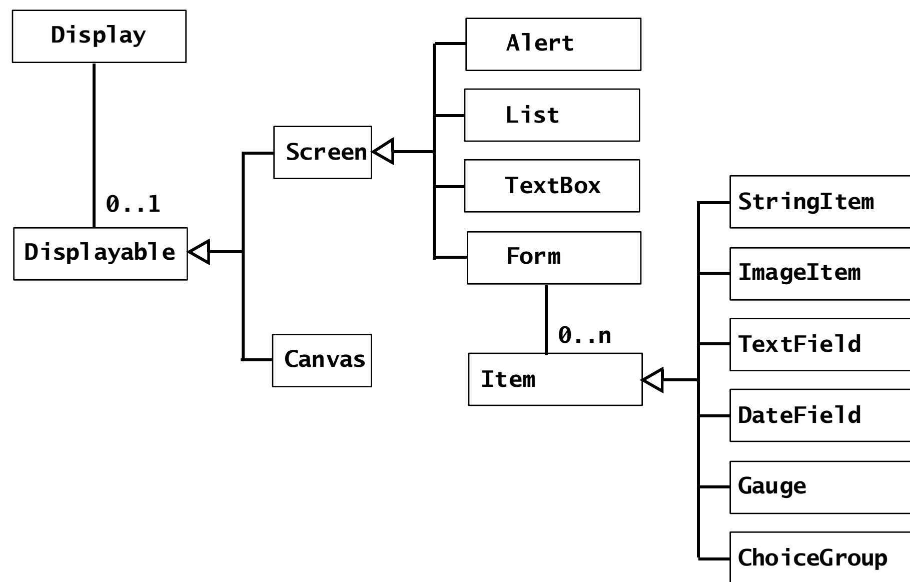


Figure 9.1 MIDP user interface class hierarchy

There are two types of `Displayable` object:

- `Canvas`: low-level objects that allow the application to provide the graphics and handle input.
- `Screen`: high-level objects that encapsulate a complete user interface component (for example, classes `Alert, List`, `TextBox`, `or Form`).

Any application may use combinations of `Screens` and `Canvases` to present an integrated user interface. For instance, in a game application, `Lists` and `Forms` can be used to select or configure the options of a game, while `Canvas` can be used for the interactive game components.

9.1.2 Low-level user interface

The *low-level* user interface API for `Canvas` is designed for applications that need precise placement and control of graphic elements, as well as need access to low-level input events. Typical examples are a game board, a chart object, or a graph. Using the low-level user interface API, an application can:

- Control what is drawn on the display
- Handle primitive events like key presses and releases
- Access concrete keys and other input devices

Applications that program to the low-level API can be portable if the application uses only the standard features; applications should stick to the platform-independent part of the low-level API whenever possible. This means that applications should not directly assume the presence of any keys other than those defined in class `Canvas`. Also, applications should inquire about the size of the display and adjust behavior accordingly.

9.1.3 High-level user interface

The *high-level* user interface API is designed for business applications whose client components run on mobile information devices. For these applications, portability across devices is important. To achieve such portability, the high-level user interface API employs a high level of abstraction and provides very little control over look and feel. In addition:

- Drawing to the display is performed by the device's system software. Applications do not define the visual appearance (such as shape, color, font, and so forth) of the components.
- Navigation, scrolling, and other primitive interactions with the user interface components are performed by the device. The application is not aware of these interactions.
- Applications cannot access concrete input mechanisms, such as individual keys.

The high-level API is provided through the `Screen` classes such as:

- `List`: select from a predefined set of choices
- `TextBox`: ask for textual input

- `Alert`: display temporary messages containing text and images
- `Form`: display multiple, closely-related user interface elements

The class `Form` is defined for cases where a screen with a single function is not sufficient. Class `Form` is designed to contain a small number of closely related user interface elements, or `Items`. For example, an application might have two `TextFields`, or a `TextField` and a simple `ChoiceGroup`. If all the `Items` of a `Form` do not fit on the screen, the implementation might either make the `Form` scrollable, expand each `Item` automatically when the user edits it, or it might use a "pop up" representation. Each `Form` can contain a combination of the following `Item` subclasses:

- `StringItem`, used for strings and text
- `ImageItem`, used for images
- `TextField`, used for textual input with constraints
- `DateField`, used to display time and dates
- `Gauge`, used to display a value graphically
- `ChoiceGroup`, used for single and multiple selections

Although the `Form` class allows creation of arbitrary combinations of components, developers should keep in mind the limited display size and make forms simple and functional.

9.2 Abstract Commands

9.2.1 Commands and command types

Since the MIDP user interface is highly abstract, it does not dictate any concrete user interaction technique such as soft buttons or menus. An abstract command mechanism is provided to adjust to the widely varying input mechanisms of MIDP target devices, and to allow the applications to be unaware of device specifics, such as number of keys, key locations, and key bindings. Low-level user interactions such as traversal or scrolling are not visible to the application. MIDP applications define `Commands`, and the implementation can provide user control over these with buttons, menus, or whatever mechanisms are appropriate for the device. Commands must be added to each screen using `Displayable.addCommand`.

Command types and priorities allow the device to place the commands to match the native style of the device, and the device might put certain types of commands in standard places. For example, the "GO BACK" operation might always be mapped to the right soft button. The Command class allows the application to communicate the semantic meaning to the implementation so that the standard mappings can be used.

The Command object has three parameters:

1. **Label**: Shown to the user as a hint.
2. **CommandType**: The meaning of the command. The most commonly used hint is *BACK*, which causes the application to go back to a previous state. Most device designs have a standard policy on which button is used for this operation.
3. **Priority**: Allows the implementation to make high priority commands more accessible.

There is one predefined command (List.SELECT_COMMAND) for selection within a list that could be, for example, implemented with *GO*, *SELECT* or a similar button. The physical button does not need to have a label, but the meaning of a button should always be obvious to the user. For example, if the user is presented with a set of mutually exclusive options, the *SELECT* operation should choose one of these options in an obvious manner.

9.2.2 Command listeners

The application-level handling of commands is based on a *listener* model. Each Displayable object has a single listener. When the user invokes a Command on a Screen, its listener is called. Listeners are registered using the method Displayable.setCommandListener. To define itself as a listener, an object must implement the interface CommandListener and its method, commandAction.

9.3 Interactions with MIDlet Application Lifecycle

User interface components play a key role in a MIDP application. When a MIDlet starts, the initialization of the user interface components is one of the first tasks for the application. The MIDlet application lifecycle is described in Section 8.2, "MIDlets."

The application management software of a MIDP system assumes that the application is well behaved with respect to the MIDlet events. The bullets below

summarize what a well behaved MIDlet is expected to do when each of its methods is called:

- **constructor:** The constructor initializes the application state. The method can access the `Display` for the `MIDlet` by calling `MIDlet.getDisplay`. It can create screens and objects needed when the application is started. It does not have access to the user interface, so information or alerts cannot be shown. The method can set the first screen to be shown to the user. The initialization performed should be brief; lengthy delays caused by long operations such as network access should be performed in the background or delayed until the user can be informed appropriately.
- **startApp:** The application manager calls this function to notify the MIDlet that it has been started (or restarted) and makes the screen set with `Display.setCurrent` visible when `startApp` returns. Note that `startApp` can be called several times if `pauseApp` is called in between. This means that objects or resources freed by `pauseApp` might need to be recreated.
- **pauseApp:** The application should release any unneeded resources that can be reinitialized if the application is restarted. The application should pause its threads unless they are needed for background activities. Also, if the application should restart with another screen when the application is reactivated, the new screen should be set with `Display.setCurrent`.
- **destroyApp:** The application should close all active resources, stop any active threads, and unregister or free any objects that would not be freed by a normal garbage collection cycle.

9.4 Graphics and Canvas in the Low-Level API

To directly use the display and the low-level user interface API, the developer uses the `Graphics` and `Canvas` classes. The `Graphics` class provides methods to paint lines, rectangles, arcs, text, and images to a `Canvas` or an `Image`. The `Canvas` class provides the display surface, its dimensions, and callbacks used to deliver events for key and pointer events and for painting the display when requested. Applications may draw by using this graphics object only for the duration of the `paint` method. The application implements a subclass of `Canvas`. The methods of this class must be overridden by the developer to respond to events and to paint the screen when requested.

The combination of the event handling capabilities of the Canvas and the drawing capabilities of the Graphics class allows applications to have complete control over the application region of the screen.[1] These low-level classes can be used, for example, to create new screens, implement highly interactive games, and manipulate images to provide rich and compelling displays for MIDP device owners.

9.4.1 Redrawing mechanism

The application is responsible for redrawing the screen whenever repainting is requested. The application implements the paint method in its subclass of Canvas. Painting of the screen is done on demand so that the device can optimize the use of graphics and screen resources, for example by coalescing several repaint requests into a single call to the paint method of the Canvas. To request a repaint of the entire display, the application calls the repaint method of the Canvas object. If only part of the screen needs to be updated, another repaint method can be given the origin, width, and height of the region that needs to be updated. The performance of graphic-intensive applications can be enhanced greatly by requesting repaints only for the region of the Canvas that changed, and implementing the paint method to only paint the area within the clip region.

Since painting is done asynchronously, the application might need to wait for repaint requests to be completed before it can continue. The Canvas.serviceRepaints method blocks until all of the repaint requests queued have resulted in a paint call.

If the device is double buffering the display, the Graphics.isDoubleBuffered method returns true. If so, the application need not separately buffer the drawing to get a clean update of the display. An Image can be used for off-screen buffering for immutable and mutable images. Use of images is described in Section 9.7, "Creating and Using Images." If double buffering is not in use, the application might be able to optimize its display updates by creating images before they are needed, and then drawing them to the display in its paint method.

9.4.2 Drawing model

The only operation on pixels is pixel replacement. The destination pixel value is replaced by the pixel value specified in the Graphics object that is used for rendering. A MIDP implementation provides no facility for combining pixel values, such as raster operations, transparency, or alpha blending.

[1] MIDP target devices often reserve certain areas of the screen for system use. Therefore, a MIDP application may not be able to draw on all the parts of the screen.

Graphics may be rendered either directly to the display or to an off-screen `Image`. The destination of rendered graphics depends on the source of the `Graphics` object. A `Graphics` object for rendering to the display is passed to the `Canvas` object's `paint` method. This is the only way to obtain a `Graphics` object whose destination is the display. Only for the duration of the `paint` method may applications draw using this `Graphics` object.

A `Graphics` object for rendering to an off-screen image is obtained by calling the `Image.getGraphics` method on the desired image. The `Graphics` object may be held indefinitely by the application, and drawing methods may be invoked on these `Graphics` objects at any time.

9.4.3 Coordinate system

The origin (0,0) of the available drawing area and images is in the upper-left corner of the display (see Figure 9.2.)

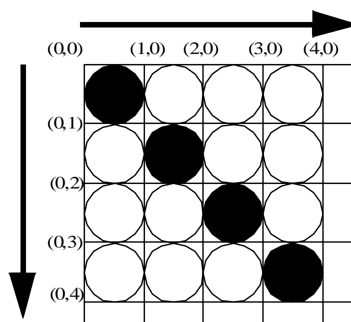


Figure 9.2 Pixel coordinate system

All coordinates are specified as integers. The numeric values of the *x*-coordinates monotonically increase from left to right, and the numeric values of the *y*-coordinates monotonically increase from top to bottom. The maximum values in *x* and *y* are available from the `getWidth` and `getHeight` methods of classes `Canvas` and `Image`. Applications may assume that horizontal and vertical distances in the coordinate system represent equal distances on the actual device display. If the shape of the pixels of the device is significantly different from square, the device does the required coordinate transformation. A MIDP implementation provides a facility for translating the origin of the coordinate system.

The coordinate system represents locations between pixels, not the pixels themselves. Therefore, the first pixel in the upper-left corner of the display lies in the square bounded by coordinates (0,0), (1,0), (0,1), (1,1).

An application can inquire about the available drawing area by calling the methods `Canvas.getWidth` and `Canvas.getHeight`. Applications should take advantage of the available screen pixels and adapt to the size as provided by the device.

9.4.4 Clipping and translation

The coordinate system described above can be translated through the use of horizontal and vertical offsets. The method `Graphics.translate` moves the location of the origin. When it is called, the offsets are added to every *x* and *y* value. The current origin can be retrieved from `Graphics.getTranslateX` and `Graphics.getTranslateY`.

There is a single clipping rectangle. Operations are provided for intersecting the current clip rectangle with a given rectangle (`Graphics.clipRect`) and for setting the current clip rectangle outright (`Graphics.setClip`). The only pixels touched by graphics operations are those that lie entirely within the clip rectangle. Pixels outside the clip rectangle are not affected by any graphics operations. Clipping is done in the same coordinates as the drawing so the clipping is relative to the untranslated coordinates. The methods `Graphics.getClipWidth`, `Graphics.getClipHeight`, `Graphics.getClipX`, and `Graphics.getClipY` are used for obtaining the current clipping region.

Clipping can be used to copy regions of `Images` from one `Image` to another or to the `Screen`. For an example, see Section 9.7.3, “Animation using images.” For another complete example, see Section 13.1, “The PhotoAlbum Application.”

9.4.5 Color model

Both color and grayscale models are supported concurrently. The 24-bit color model has eight bits for each of red, green, and blue. The current color is set using the method `Graphics.setColor`. The current color is used for all lines, text, and fills for rectangles and arcs. Separate background and foreground colors are not supported.

Few devices support full 24 bits of color. The device maps the color requested by the application into a color available on the device. The details of the mapping are device specific. The `Display.isColor` method returns `true` if the display is capable of supporting multiple colors, and `false` if the display supports grayscale or black and white only. The number of distinct color or gray levels is available by calling the method `Display.numColors`. The color values are converted to grayscale values by the device. The grayscale equivalent can be retrieved with method `getGrayScale` after a color has been set.

Grayscale is supported with values within the range of 0 to 255. The current grayscale value is set with method `Graphics.setGrayScale`. The device maps the values into the number of gray levels in the display. The corresponding color is available after setting a grayscale value by calling method `Graphics.getColor`.

9.4.6 Line styles

Lines, arcs, rectangles, and rounded rectangles can be drawn with either a *SOLID* or a *DOTTED* stroke style, as set by the `Graphics.setStrokeStyle` method. The stroke style does not affect the fill, text and image operations.

For the *SOLID* stroke style, drawing operations are performed with a one pixel-wide pen that fills the pixel immediately below and to the right of the specified coordinate. Drawn lines touch pixels at both endpoints.

Drawing operations under the *DOTTED* stroke style touches a subset of pixels that would have been touched under the *SOLID* stroke style. The frequency and length of dots is device-dependent. The endpoints of lines and arcs might not be drawn. Similarly, the corner points of rectangles might not be drawn. Dots are drawn by painting with the current color. Spaces between dots are left untouched.

9.4.7 Fonts

MID devices typically have a limited number of fonts with a fixed set of styles, sizes, and faces. The `Font.getFont` method returns the `Font` associated with a given style, size, and face. It is up to the device to select a font that most closely matches the requested attributes. The following attributes can be used to request a font (from the `Font` class):

- **Size**: *SMALL*, *MEDIUM*, *LARGE.*
- **Face**: *PROPORTIONAL*, *MONOSPACE*, *SYSTEM.*
- **Style**: *PLAIN*, *BOLD*, *ITALIC*, *UNDERLINED.*

The `Font` class provides methods to access the font metrics, including `getHeight and getBaselinePosition`. For the widths of strings and sequences of chars the `Font` class methods `charWidth`, `charsWidth`, `stringWidth`, and `substringWidth` can be used. The whitespace for interline and intercharacter spacing is included in the metrics and is below and to the right of the characters, respectively. The metrics allow an application to precisely place text relative to graphics.

The methods `Graphics.setFont` and `Graphics.getFont` are used to set and get the current `Font`. Each `Graphics` object always has a font set with a default chosen by the device.

9.4.8 Canvas visibility

The display on a device is shared among native applications and MIDP applications. A particular application might or might not be displayed on the screen. When the application's choice of current screen is a `Canvas`, the canvas is notified when its visibility changes. The `Canvas.showNotify` and `Canvas.hideNotify` methods are invoked automatically on the `Canvas` when it is shown and hidden, respectively. The application should override these methods to be informed of changes in visibility. When made visible, the `paint` method of the `Canvas` is called, so the application does not need to call `repaint` explicitly. Show and hide notifications are useful if, for example, the `Canvas` is providing an animation that it could start animating when shown and stop animating when hidden.

9.5 Low-level API for Events in Canvases

9.5.1 Key events

If the application needs to handle key events in its `Canvas`, it must override the `Canvas` methods `keyPressed`, `keyReleased`, and `keyRepeated`. When a key is pressed, the `keyPressed` method is called with the key code. If the key is held down long enough to repeat, the `keyRepeated` method is called for each repeated `keyCode`. When the key is released, the `keyReleased` method is called. Some devices might not support repeating keys, and if not, the `keyRepeated` method is never called. The application can check the availability of repeat actions by calling the method `Canvas.hasRepeatEvents`.

MIDP target devices are required to support the ITU-T telephone keys. Key codes are defined in the `Canvas` class for the digits `0` through `9`, `*`, and `#`. Although an implementation may provide additional keys, applications relying on these keys may not be portable. For portable applications, the action key mappings described below should be used whenever possible, since other key codes are device-specific.

9.5.2 Action keys

The `Canvas` class has methods for handling portable *action events* (also known as *game actions*) from the low-level user interface. The API defines a set of action events: *`UP`*, *`DOWN`*, *`LEFT`*, *`RIGHT`*, *`FIRE`*, *`GAME_A`*, *`GAME_B`*, *`GAME_C`*, and *`GAME_D`*. The device maps the action events to suitable key codes on the device. For example, a device with four navigation keys and a *`SELECT`* key in the middle could use those keys for mapping the action events, but a simpler device might use keys on the numeric keypad (such as 2, 4, 5, 6, 8) instead.

The mapping between keys and the abstract action events does not change during the execution of an application. An application can get the mapping of the key codes to action events by calling the method `Canvas.getGameAction`. If the logic of the application is based on the values returned by this method, the application is portable and runs regardless of the keypad design.

Action events are mapped to device-specific key codes that can be retrieved with method `Canvas.getKeyCode`. The application may choose to determine the key codes once during initialization for each action it uses, and later utilize those key codes in the `keyPressed` methods.

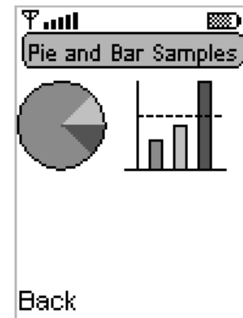
9.5.3 Pointer events

The `Canvas` class has methods that the application can override to handle pointer events. If the application needs to handle pointer events, it must override and implement the methods `pointerPressed`, `pointerReleased`, and `pointerDragged`.

Not all MIDP target devices support pointer events, and therefore the pointer methods might never be called. The application can check whether the pointer and pointer motion events are available by calling `Canvas.hasPointerEvents` and `Canvas.hasPointerMotionEvents`.

9.6 Graphics Drawing Primitives

The `Graphics` class provides various low-level drawing primitives. In general, drawing is started by calling the various methods of the `Graphics` object to set the color, translation, and clipping. Methods are available for drawing lines, filled and outlined rectangles, filled and outlined rounded rectangles, text, and images. Each of the drawing primitives is explained below, along with figures that illustrate the result of the sample drawing calls. The figure on the right shows the cumulative effect of these sample drawing calls.



9.6.1 Scaling to the Canvas

Typically, before any low-level drawing operations can be used, the application should perform certain initialization operations and calculations based on the size of the `Canvas` or `Image` and the `Fonts` to be used. The code snippet below illustrates how to compute the relevant size and position information for the drawing examples that we provide in this section. Below, the constructor for the sample `Canvas` computes and saves the width and height of the `Canvas`, the `Font`, the height of the `Font`,

a padding amount, and the height of the title bar based on the `Font` size. It uses the remaining screen space for the pie chart example (Section 9.6.3, “Drawing and filling arcs”) by subtracting the title height from the screen height and padding. The size of the charts may be reduced so that the pie and bar charts can fit side by side with padding in between. The size of the bar chart is set to be the same as the pie chart. These layout computations are different for every use of a `Canvas`, and might need some adjustments to look good on various devices.

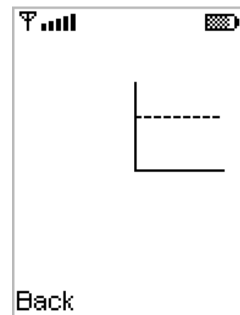
```
int w = getWidth();
int h = getHeight();
int font = Font.getFont(Font.FACE_SYSTEM,
                        Font.STYLE_PLAIN, Font.SIZE_SMALL);
int fh = font.getHeight();
int pad = 2;
int titleHeight = fd + pad * 2;
int barSize = h - (titleHeight + pad);
if (barSize > (w - pad) / 2)
    barSize = (w - pad) / 2;
int pieSize = barSize;
```

9.6.2 Drawing lines

The `Graphics.drawLine` method draws lines from a starting (x, y) coordinate to an ending (x, y) location, touching each pixel below and to the right of the coordinates that the line touches. The pixels are set to the value of the current color value. The lines are drawn with the current stroke style (see Section 9.4.6, “Line styles.”)

For example, here is the code to draw the axes for the bar chart in the figure on the right:

```
Graphics g = ...
int h1 = barSize / 3, h2 = barSize / 2,
    h3 = barSize; // Scaled data
int avg = (h1 + h2 + h3) / 3;
int yorig = barSize;
g.translate((w + pad) / 2, titleHeight + pad);
g.setGrayScale(0);
g.drawLine(0, 0, 0, yorig);
g.drawLine(0, yorig, barSize, yorig);
g.setStrokeStyle(Graphics.DOTTED);
g.drawLine(0, yorig-avg, barSize, yorig-avg);
```



9.6.3 Drawing and filling arcs

The `Graphics.drawArc` method draws the outline of a circular or elliptical arc covering the specified rectangle, using the current color and stroke style. The resulting arc begins at *`startAngle`* degrees and extends for *`arcAngle`* degrees. Angles are interpreted such that 0 degrees is at the 3 o'clock position. A positive value indicates a counter-clockwise rotation, while a negative value indicates a clockwise rotation.

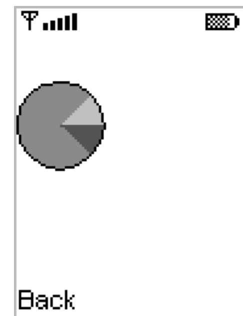
The center of the arc is the center of the rectangle whose origin is (*x*,*y*) and whose size is specified by the *`width`* and *`height`* arguments. The resulting arc covers an area *`width+1`* pixels wide by *`height+1`* pixels tall. If either *`width`* or *`height`* is less than zero, nothing is drawn.

The angles are specified relative to the non-square extents of the bounding rectangle such that 45 degrees always falls on the line from the center of the ellipse to the upper-right corner of the bounding rectangle. As a result, if the bounding rectangle is noticeably longer in one axis than the other, the angles to the start and end of the arc segment are skewed farther along the longer axis of the bounds.

The `Graphics.fillArc` method draws and fills a circular or elliptical arc covering the specified rectangle. The filled region consists of the "pie wedge" region bounded by the arc segment as if drawn by `drawArc`, the radius extending from the center to this arc at *`startAngle`* degrees, and radius extending from the center to this arc at *`startAngle`* + *`arcAngle`* degrees.

For example, here is the code to fill the arcs and draw the outlines of the chart shown at right:

```
Graphics g = ...
g.translate(0, titleHeight + pad);
g.setColor(255, 0, 0);
g.fillArc(0, 0, pieSize, pieSize, 45, 270);
g.setColor(0, 255, 0);
g.fillArc(0, 0, pieSize, pieSize, 0, 45);
g.setColor(0, 0, 255);
g.fillArc(0, 0, pieSize, pieSize, 0, -45);
g.setColor(0);
g.drawArc(0, 0, pieSize, pieSize, 0, 360);
```



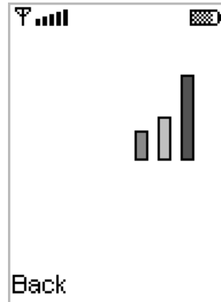
9.6.4 Drawing and filling rectangles

The `Graphics.drawRect` method draws a rectangle with the current color and stroke style. The resulting rectangle covers an area *`width+1`* pixels wide by *`height+1`* pixels tall. If either *`width`* or *`height`* is less than zero, nothing is drawn.

The `Graphics.fillRect` method fills rectangles with the current color. If either *width* or *height* is less than zero, nothing is drawn.

For example, see below for the code to draw the bar chart (shown in the figure) using filled and outlined rectangles. Note that the widths and spacing of the bars is computed from the size of the chart so they scale to the screen area available.

```
Graphics g = ...
int h1 = barSize / 3, h2 = barSize / 2,
    h3 = barSize; // Scaled data
// width of spaces and bars
int bw = barSize / 7;
// translate to right half of screen
g.translate((w + pad) / 2, titleHeight + pad);
g.setColor(255, 0, 0);
g.fillRect(bw, yorig-h1, bw+1, h1);
g.setColor(0, 255, 0);
g.fillRect(bw*3, yorig-h2, bw+1, h2);
g.setColor(0, 0, 255);
g.fillRect(25, yorig-h3, bw+1, h3);
g.setColor(0);
g.drawRect(bw, yorig-h1, bw, h1);
g.drawRect(bw*3, yorig-h2, bw, h2);
g.drawRect(bw*5, yorig-h3, bw, h3);
```



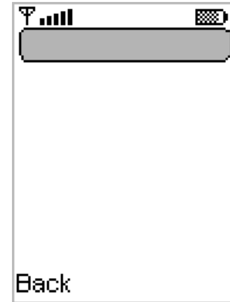
9.6.5 Drawing and filling rounded rectangles

The `Graphics.drawRoundRect` method draws a rectangle with rounded corners in the current color and stroke style. The resulting rectangle covers an area *width+1* pixels wide by *height+1* pixels tall. If either *width* or *height* is less than zero, nothing is drawn. The corners are rounded using *width* and *height* diameter measurements of the curve.

The `Graphics.fillRoundRect` method fills rectangles with rounded corners in the current color. If either *width* or *height* is less than zero, nothing is drawn.

For example, to draw the title bar shown in the figure at the top of the next page, the font metrics of the title string are used to compute the size of the filled and drawn rounded rectangles, as follows:

```
Graphics g = ...
Font font = g.getFont();
int swidth = pad * 2 +
    font.stringWidth("Pie and Bar Samples");
int title_x = (w - swidth) / 2;
g.setGrayScale(128);
g.fillRoundRect(title_x, 0, swidth, fh, 5, 5);
g.setGrayScale(0);
g.drawRoundRect(title_x, 0, swidth, fh, 5, 5);
```



9.6.6 Drawing text and images

The drawing of text is based on anchor points instead of the standard notion of baseline, as illustrated in Figure 9.3.

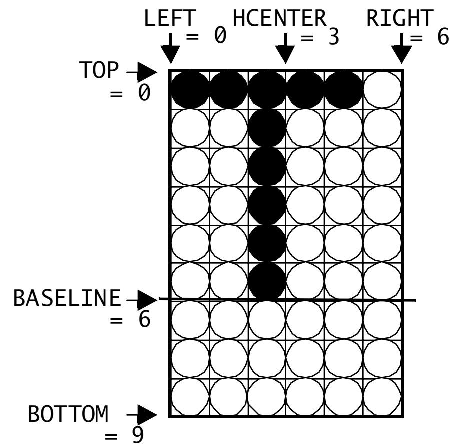


Figure 9.3 Text anchor points

Anchor points are used to minimize the amount of computation required when placing text. The `Graphics.drawString` method draws text in the current foreground color using the current font with its anchor point at (*x*, *y*). The anchor point should be one of the horizontal constants (*LEFT*, *HCENTER*, *RIGHT*), logically combined (OR-ed) with one of the vertical constants (*TOP*, *BASELINE*, *BOTTOM*). The default anchor point is 0, which signifies that the upper-left corner of the text's bounding box is used.

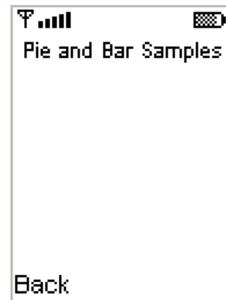
The position of the bounding box of the text relative to the (*x*, *y*) location is determined by the anchor point. These reference points occur at named locations along the outer edge of the bounding box.

For text drawing, character and line spacing are included in the values returned in the `Font.stringWidth` and `Font.getHeight` method calls. The interline and intercharacter space is below and to the right of the pixels belonging to the characters drawn. Reasonable vertical spacing is achieved simply by adding the font height to the *y*-position of subsequent lines.

The positioning of `Images` using anchors is the same as for text, but the *`BASELINE`* anchor is not used and the *`VCENTER`* anchor is allowed.

For example, to draw the title text shown in the figure on the right, the text is positioned with padding relative to the top and left anchor point:

```
Graphics g = ...
g.setColor(0, 0, 0);
g.drawString("Pie and Bar Samples", pad, pad,
             Graphics.TOP | Graphics.LEFT);
```



9.7 Creating and Using Images

9.7.1 Mutable and immutable images

Images in a MIDP implementation may be either *immutable* or *mutable*. Immutable images can be used in `Alert`, `List`, and `Form` screens. By allowing only immutable images in these screens, the screen update mechanism is kept simple, and the implementation is easier and smaller. Immutable images can be created directly from resource files, from binary data provided by the application, or from other images.

- The `Image.createImage(String` *`name`*`)` method is used to load and create an immutable image from a resource file bundled with the application in the application's JAR file. The name must begin with "/" and include the full name of the PNG image within the JAR file.
- The `Image.createImage(byte[], int` *`offset`*`, int` *`length`*`)` method is used to create an immutable image from binary PNG format data.
- The `Image.createImage(Image` *`image`*`)` method is used to create an immutable image from another `Image` (which could be either mutable or immutable).

In the first two cases the image data must be in Portable Network Graphics (PNG) as specified by the W3C-PNG (Portable Network Graphics) Specification, Version 1.0. W3C Recommendation, October 1, 1996. This specification is available at `http://www.w3.org/TR/REC-png.html` and as RFC 2083, available at `http://www.ietf.org/rfc/rfc2083.txt`.

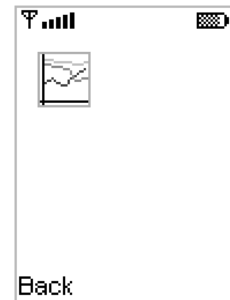
9.7.2 Drawing to a mutable image

Mutable images are created with the method `Image.createImage(`*`int width, int height`*`)`. The `Image.getGraphics` method returns a `Graphics` object that can be used for drawing into the `Image`. All normal graphics methods operate on the `Image`. The `Image` has the same characteristics as the display of the device: for example, whether the is color or grayscale, and the number of available colors or gray levels.

For a complete example that illustrates the use of mutable and immutable images, refer to Section 13.1, "The PhotoAlbum Application." The code excerpt below demonstrates the creation of a new mutable image. An immutable image read from a resource from the JAR file is drawn on the mutable image; in addition, a rectangle is drawn for a border.

The figure shows the result when the image is drawn to the screen.

```
Image image = Image.createImage(50, 50);
Image icon = Image.createImage(
             "/icons/icon.png");
Graphics g = image.getGraphics();
g.drawImage(icon, 10, 10, TOP | LEFT);
g.setColor(128 128, 128);
g.drawRect(9, 9, icon.getWidth()+1,
                 icon.getHeight())+1);
```



9.7.3 Animation using images

A very useful technique for animating graphics takes advantage of the clipping and translation capabilities of the `Graphics` object to move a region of one `Image` to another `Image`. This technique can improve the performance of graphics intensive applications considerably. In other graphics systems, this technique is known as *bitblt*.

For example, the figure to the right shows a sequence of images that, if seen in rapid succession at the same location on the screen, provide an animation of a running dog. The images and techniques are courtesy of Mark Patel at Motorola.

The series of images are stacked in a single Portable Network Graphics (PNG) image and stored in the application JAR. The general technique is to set the clipping region and translation coordinates in the destination image. Drawing the region is performed by calling the method `Graphics.drawImage` by picking the region in the source image that aligns with the target. For each iteration, the next image is selected and its offset on the screen is selected. A separate thread is used to advance the *`frameIndex`* and to issue the `repaint` requests at the proper time, with the region to erase the previous image as well as to draw the new image.

```
package examples.animation;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Image;

/**
 * This Canvas subclass demonstrates how to create a simple
 * animation sequence using the MIDP graphics APIs.
 *
 * @author Mark A. Patel - Motorola, Inc.
 **/
public class Doggy extends Canvas implements Runnable {

    /**
     * Number of frames in the animation
     **/
    static final int FRAME_COUNT = 17;

    /**
     * Normal frame delay (milliseconds)
     **/
    static final int FRAME_DELAY = 180;

    /**
     * Frame delay for the last frame where the dog is sleeping
     **/
    static final int LAST_FRAME_DELAY = 3000;
```

```
/**
 * Relative horizontal position where each of the frames
 * should be rendered. 0 represents the left edge of the
 * screen and 1024 represents the right edge of the run
 * distance (1024 is used so that scaling can be performed
 * using simple bit shifts instead of division operations).
 **/
static final int[] framePositions = {
    0, 50, 186, 372, 558, 744, 930, 1024, 1024,
    834, 651, 465, 279, 93, 0, 0, 0
};

/**
 * An Image containing the 17 frames of the dog running,
 * stacked vertically.
 * Using a single image is much more efficient than using
 * several images with each containing a single frame.
 * Each frame can be rendered separately by setting the clip
 * region to the size of a single frame, and then
 * rendering the image at the correct position so that the
 * desired frame is aligned with the clip region.
 **/
Image doggyImages = null;

/**
 * Width of a single animation frame
 **/
int frameWidth = 0;

/**
 * Height of a single animation frame
 **/
int frameHeight = 0;

/**
 * Index of the current frame
 **/
int frameIndex = 0;
```

```
/**
 * The distance, in pixels, that the dog can run
 * (screen width less the width of a single frame)
 **/
int runLength = 0;

/**
 * Indicates if the animation is currently running
 **/
boolean running = false;

/**
 * Called when this Canvas is shown. This method starts the
 * timer that runs the animation sequence.
 **/
protected void showNotify() {
    if (doggyImages == null) {
        try {
           doggyImages =
            Image.createImage("/examples/animation/Doggy.png");
           frameWidth = doggyImages.getWidth();
           frameHeight = doggyImages.getHeight()/FRAME_COUNT;
        } catch (Exception ioe) {
            return; // no image to animate
        }
    }
    runLength = getWidth() - frameWidth;
    running = true;
    frameIndex = 0;

    new Thread(this).start();
}

/**
 * Called when this Canvas is hidden. This method stops
 * the animation timer to free up processing
 * power while this Canvas is not showing.
 **/
protected void hideNotify() {
    running = false;
}
```

```
public void run() {

    // Need to catch InterruptedExceptions
    // and bail if one occurs
    try {
        while (running) {
            Thread.sleep((frameIndex == FRAME_COUNT - 1) ?
                        LAST_FRAME_DELAY : FRAME_DELAY);

            // Remember the last frame index so we can
            // compute the repaint region
            int lastFrameIndex = frameIndex;

            // Update the frame index
            frameIndex = (frameIndex + 1) % FRAME_COUNT;

            // Determine the left edge of the repaint region
            int repaintLeft = framePositions[lastFrameIndex];
            int repaintRight = framePositions[frameIndex];
            if (framePositions[lastFrameIndex] >
                    framePositions[frameIndex]) {
               repaintLeft = framePositions[frameIndex];
               repaintRight = framePositions[lastFrameIndex];
            }


            // Scale repaint coordinates to width of screen
            repaintLeft = (repaintLeft * runLength) >> 10;
            repaintRight = (repaintRight * runLength) >> 10;

            // Trigger repaint of the affected portion
            // of screen. Repaint the region where the
            // last frame was rendered
            // (ensures that it is cleared)
            repaint(repaintLeft, 0,
                    frameWidth + repaintRight - repaintLeft,
                    frameHeight);
        }
    } catch (InterruptedException e) {}
}
```

```
    public void paint(Graphics g) {

        // Clear the background (fill with white)
        // The clip region limits the area that actually
        // gets cleared to save time
        g.setColor(0xFFFFFF);
        g.fillRect(0, 0, getWidth(), frameHeight);

        // Translate the graphics to the appropriate position
        // for the current frame
        g.translate((framePositions[frameIndex]*runLength)
                        >> 10, 0);

        // Constrain the clip region to the size
        // of a single frame
        g.clipRect(0, 0, frameWidth, frameHeight);

        // Draw the current frame by drawing the entire image
        // with the appropriate vertical offset so that the
        // desired frame lines up with the clip region.
        g.drawImage(doggyImages, 0, -(frameIndex*frameHeight),
                    Graphics.LEFT + Graphics.TOP);
    }
}
```

9.8 Using Screens

Each subclass of `Screen` is a complete functional user interface element. The software in a given MIDP device implements each kind of `Screen` using the look and feel of the device. The device software takes care of all events that occur as the user navigates in a `Screen`. When user action causes a command invocation event, the event is passed to the application, which brings about a transition to a different `Screen`. Each `Screen` can have a title, multiple commands, and a `Ticker`. It is up to the device software to include these elements in the visual presentation to the user. The four kinds of `Screens` (`List`, `TextBox`, `Alert` and `Form`) are described below.

9.8.1 List

The `List` class is a `Screen` that contains a list of choices. Each choice has an associated string and may have an icon (an `Image`). When a `List` is being displayed, the user can interact with it, for instance, by traversing and possibly scrolling from ele-

ment to element. These traversing and scrolling operations are handled by the system and do not generate any events that are visible to the MIDP application. The system notifies the application only when a `Command` is fired. The notification to the application is carried out via the `CommandListener` of the `Screen`.

There are three types of `Lists`: *implicit*, *exclusive* and *multiple choice*. The type of the `List` is selected when the `List` is constructed, and cannot be changed during the lifetime of the `List` object. The look and feel of `Lists` varies from one type of `List` to another, as illustrated by the examples below. Also note that different MIDP implementations may render the `Lists` differently.

Operations on `List` objects include `insert`, `append` and `delete`, and operations to get the `String` or the `Image` from any element. Elements can be added and removed from the `List` at any time. The selected elements in the `List` can be retrieved or changed. However, changing the `List` while the user is viewing it or trying to select an item is not recommended since it might cause confusion.

Most of the behavior for managing a `List` of elements is common with class `ChoiceGroup`; the common API is defined in the interface class `Choice`. Elements in the `List` are manipulated with the methods `append`, `delete`, `getImage`, `getString`, `insert` and `set`. Item selection is handled with `setSelectedIndex`, `setSelectedFlags`, `getSelectedFlags`, `getSelectedIndex`, `isSelected` and `getSelectedIndex`.

An example:

```
List list = new List(null,
                     Choice.IMPLICIT);
list.append("Canvas", null);
list.append("Form", null);
list.append("Alert", null);
list.append("TextBox", null);
list.append("Exclusive List", null);
list.append("Multiple Choice", null);
list.setCommandListener(this);
list.addCommand(exitCommand);
```

Implicit `List`

An *`Implicit`* `List` is used for a quick selection when the only action that is needed from the user is to select an element. The `List` is presented like a menu from which the user can pick one of the elements. The device's default *`SELECT`* key is used as the trigger to deliver the predeclared `Command.SELECT_COMMAND` to the `List's CommandListener`. To be notified of the selection, the `List` must have a `CommandListener` set. The application can add other `Commands` to the `List`, and the user may choose one of them instead of selecting one of the elements in the `List`.

```
public void commandAction(Command cmd, Displayable d) {
    if (cmd == Command.SELECT_COMMAND) {
        int i = (List)d.getSelectedIndex();
        // Act on the item selected
    } else if (cmd == ...) {
        // Check for and handle other commands
    }
}
```

Exclusive Choice `List`

An *Exclusive* `List` allows the user to select a single element. This type of a `List` is commonly rendered to the screen as a group of radio buttons. When the user selects an element, any previously selected element is automatically deselected. Selecting an element does not notify the application. Notification to the `List's` `CommandListener` occurs when a `Command` on the `List` is chosen. When the listener is invoked, it can determine which element is selected with the `List.getSelected` method. An exclusive `List` *must* have `Commands` added to it, or the user will not be able to trigger the action and will be stuck on the screen. In the example below, the user is able to pick one choice and then can select either the *OK* or *BACK* command.

```
List list = new List("Border Style",
                     Choice.EXCLUSIVE);
list.append("None", null);
list.append("Plain", null);
list.append("Fancy", null);
list.addCommand(backCommand);
list.addCommand(okCommand);
...
```

```
public void commandAction(Command cmd,
                          Displayable d) {
    if (cmd == okCommand) {
        int i = (List)d.getSelectedIndex();
        // Use the index of selected list element...
    } else if (cmd == backCommand) {
        // handle the back command
    }
}
```

Multiple Choice `List`

A *Multiple Choice* `List` allows the user to select zero or more elements. Each element can be selected or deselected individually. Typically, this type of a `List` is presented on the screen as a set of check boxes. Each element has an indicator displaying whether the element it is currently selected or not, and the user can toggle individual elements on or off. Toggling the selection of an element does not notify the application. Notification to the `List's` `CommandListener` occurs when a `Command` on the `List` is chosen. When the listener is invoked, it can determine which element(s) are selected with the `List.getSelectedFlags` or `List.isSelected` method. A multiple choice `List` *must* have `Commands` added to it, or the user will not be able to trigger the action and will be stuck on the screen.

In this example, the user can select and deselect individually choices for "*Red*", "*Green*", and "*Blue*". When satisfied with their choices, users can select either the *OK* or *BACK* command to exit the screen.

```
List list = new List("Colors to mix",
                     Choice.MULTIPLE);
list.append("Red", null);
list.append("Green", null);
list.append("Blue", null);
text.addCommand(backCommand);
text.addCommand(okCommand);
...
```

```
public void commandAction(Command cmd,
                          Displayable d) {
    if (cmd == okCommand) {
        List list = (List)d;
        for (int i = 0; i < list.size(); i++) {
            boolean selected = list.isSelected(i);
            // If selected, take action...
        }
    }
}
```

9.8.2 TextBox

The `TextBox` class defines a `Screen` that allows the user to enter and edit text. The application sets the maximum number of characters that the text box can contain and

the defines the input constraints. The characters in the TextBox can be modified and retrieved either as a String or a sequence of characters by the application.

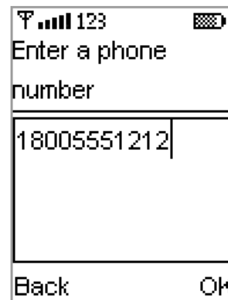
The application can set the input constraint individually for each TextBox. The available input constraint modes are *ANY*, *NUMERIC*, *PASSWORD*, *PHONENUMBER*, *URL*, and *EMAILADDR*. Each mode defines a specific set of characters that are valid to be entered. However, the mode does not automatically validate the format or syntax of the input.

The device can use the input constraints to make it easier for the user to input allowed characters, or can use them to format the field value for display. For example, in a *PHONENUMBER* TextBox, the numeric keypad could perhaps be automatically enabled for direct input of numbers, and the output can be formatted as a phone number. The device may also enable the "*TALK*" button on the phone to dial the number. This provides applications a simple way to present a phone number to the user and to allow them to dial that number.

A TextBox *must* have Commands added to it, or the user will not be able to trigger any action and will be stuck editing text in the TextBox.

For example, the TextBox created with the code below is created to with an initial phone number that can be edited and it has two commands. (Refer to the figure at the left.)

```
TextBox text = new TextBox("Phone",
                           "18005551212", 10,
                           TextField.PHONENUMBER);
   text.addCommand(backCommand);
   text.addCommand(okCommand);
```



9.8.3 Alert

An Alert is a Screen that shows a message and an optional Image to the user for a certain period of time before proceeding to the next screen. Alerts are used to inform the user about errors and other exceptional conditions. When an Alert is displayed, the application can choose the Screen to be displayed after the Alert is complete. By default, after an Alert exits, the display reverts to the current screen.

The AlertType of an Alert can be set to indicate the nature of the information provided in the Alert. There are five AlertTypes; *ALARM*, *CONFIRMATION*, *ERROR*, *INFO*, and *WARNING*. AlertTypes can have sounds associated with them so that the device can audibly alert the user. The specific sound or whether a sound is used at

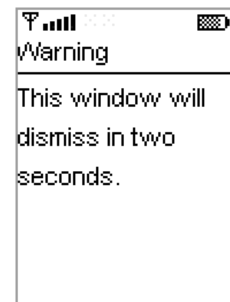
all is device dependent. The sound associated with the `AlertType` can be played anytime by calling the method `AlertType.playSound`.

The application can set the alert time to be infinite with the method `setTimeout(Alert.FOREVER)`. In this case, the `Alert` is considered to be *modal*, and the device provides a feature that allows the user to dismiss the alert, whereupon the next screen is displayed.

Timed `Alerts` can be used when the user does not need to be aware of the information presented and can safely ignore it. `Alerts` without a time-out should be used when the user must be made aware of the information or condition. `Alerts` cannot be used if the user must be able to choose from multiple responses, because `Commands` are not allowed on `Alerts`. In that case, another type of `Screen` should be used to present the information to the user instead of an `Alert`.

An example:

```
Alert alert = new Alert("Warning");
alert.setString("This window will
                 dismiss in two seconds.");
alert.setTimeout(2000);
display.setCurrent(alert);
```



9.8.4 Ticker

A `Ticker` contains a `String` that scrolls continuously across the display. The direction and the speed of scrolling are determined by the device. When the `String` finishes scrolling off the display, the message starts over at the beginning of the `String`. `Tickers` can be set on any `Screen`. A `Ticker` object can be shared between `Screens`, so that when switching between `Screens` the message appears continuously.

For example, a `Ticker` can be added to a `List` and is displayed across the top.

```
Ticker ticker = new Ticker("Select an item to display");
List list = new List("", List.IMPLICIT);
... // Add choices to the List and set command listener
list.addCommand(okCommand);
list.setTicker(ticker);
display.setCurrent(list);
```

9.8.5 Form

A `Form` is a `Screen` that may contain a combination of `Items` including `Strings`, `Images`, editable `TextFields`, editable `DateFields`, `Gauges` and `ChoiceGroups`. Any of the subclasses of `Item` defined by the *MIDP Specification* may be contained within a `Form`. The device handles layout, traversal and possible scrolling automatically. None of the `Items` contained within a `Form` has any internal scrolling; rather, the entire contents of the `Form` scroll up and down together. Horizontal scrolling is not usually appropriate on small screens with consumer type users. Restricting to vertical scrolling makes the layout and traversal model easier and matches the limited set of controls available on a typical mobile information device.

The methods for modifying the sequence of `Items` stored in a `Form` include `insert`, `append`, `delete`, `get`, `set` and `size`.

An example:

```
Form form = new Form("Options");
form.addCommand(backCommand);
form.addCommand(okCommand);
form.addCommandListener(this);
... // Add Items (see below)
display.setCurrent(form);
```

9.9 Using Items

A `Form` contains a sequence of `Items`. The different `Item` types are discussed below, along with a description of class `ItemStateListener` that can be used for listening for changes in `Items`.

9.9.1 Item

Class `Item` is a superclass for components that can be attached to a `Form`. All `Item` objects have a label field, which is a `String` representing the title of the `Item`. The label is typically displayed near the `Item` when the `Item` is visible on the screen. If the screen is scrolling, the implementation tries to keep the label visible at the same time with the `Item`. An `Item` may attached to one `Form` only; this simplifies the implementation since it does not have to share the internal state of `Item` objects.

9.9.2 String and StringItem

Read-only strings can be added to a `Form` either directly as `Strings` (Java string objects) or as `StringItems`. `StringItem` is a simple class that is used to wrap `Strings` so they can be treated consistently with other `Items`. `Strings` are converted to `StringItems` automatically when they are appended to a `Form`. When an `Item` that was appended as a `String` is retrieved from a `Form`, it is returned as a `StringItem`.

Continuing example:

```
form.append("The year is ");
form.append(new StringItem("2001"));
```

9.9.3 Image and ImageItem

`Image` objects can be added to a `Form` either directly as an `Image` or as an `ImageItem`. `ImageItem` is a simple class that wraps an `Image` and allows the `Item` to be positioned relative to the other `Items` of the `Form`. The `Image` can be placed *centered*, *left-justified* or *right-justified,* and with a line break either before or after the `Image`. Combinations of position and newline options are allowed. The `ImageItem` should include alternate text to be used if the `Image` cannot be displayed by the device.

Continuing example:

```
Image image = Image.createImage("/images/PhotoAlbum.png");
imageItem = new ImageItem("Preview:", image,
                ImageItem.LAYOUT_NEWLINE_BEFORE, "Mountain");
```

9.9.4 TextField

A `TextField` is an editable text component that may be placed in a `Form`. As with `TextBox`, it has a maximum size, input constraints on the valid input mode (see Section 9.8.2, “TextBox”), a label and a value. The application can set the input constraint individually for each `TextField`. The available input constraint modes are *`ANY`*, *`NUMERIC`*, *`PASSWORD`*, *`PHONENUMBER`*, *`URL`*, and *`EMAILADDR`*. Each mode defines a specific set of characters that are valid to be entered. The value can be retrieved and set as a sequence of characters or as a `String`. The number of characters displayed and their arrangement in rows and columns are determined by the device.

An example:

```
TextItem textItem =
    new TextField("Title:", "Mountain", 32, TextField.ANY);
form.append(textItem);
```

9.9.5 DateField

A `DateField` is a component that may be placed in a `Form` in order to present date and time (calendar) information. The value for a `DateField` can be initially set or left unset. If the value is not set, the `DateField.getDate` method returns `null`. The user interface must visually indicate that the date and time is unknown.

Each instance of a `DateField` can be configured to accept date or time information or both. This input mode configuration is done by choosing the `DATE`, `TIME` or `DATE_TIME` mode of this class. The `DATE` input mode configures the `DateField` to allow only date information and `TIME` only time information (hours, minutes). The `DATE_TIME` mode allows both clock time and date values to be set.

An example:

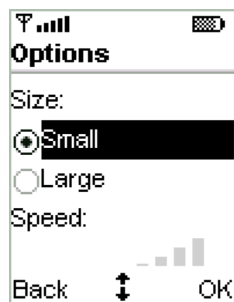
```
DateField date =
    new DateField("Date", DateField.DATE);
date.setDate(new Date());
// Set date to "now"
form.append(date);
```

9.9.6 ChoiceGroup

A `ChoiceGroup` defines a group of selectable elements that can be placed within a `Form`. A `ChoiceGroup` is similar to `List` (Section 9.8.1, "List"), but it supports only the exclusive and multiple choice modes. The device is responsible for providing the graphical representation of these modes and must provide visually different graphics for different modes. For example, it might use radio buttons for the exclusive choice mode and check boxes for the multiple choice mode.

An example:

```
ChoiceGroup choice =
    new ChoiceGroup("Size:",
                    Choice.EXCLUSIVE);
choice.append("Small", null);
choice.append("Large", null);
form.append(choice);
```

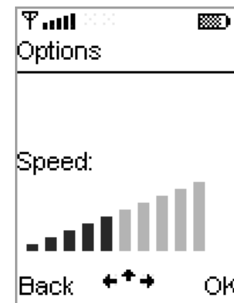


9.9.7 Gauge

The `Gauge` class implements a graphical value display that may be placed in a `Form`. A `Gauge` displays a visual representation of a numeric value in the range between zero and the maximum value defined by the programmer. If the `Gauge` is set to be interactive, the user actions can increase and decrease the value. The application can get and set the value of `Gauge`. Changes to the `Gauge` value are reported using an `ItemStateListener`.

An example:

```
Gauge gauge =
    new Gauge("Speed:", true, 10, 5);
form.append(gauge);
```



9.9.8 ItemStateListener

When the user changes an editable `Item` in a `Form`, the application can be notified of the change by implementing the `ItemStateChanged` interface and the `itemStateChanged` method. The `itemStateChanged` method is called automatically when the value of an interactive `Gauge`, `ChoiceGroup`, or `TextField` changes. The listener is set for the `Form` using the `setItemListener` method. It is not expected that the listener is called after every event that causes a change. However, if the value has changed, the listener is called sometime before it is called for another `Item`, or before a `Command` is delivered to the `Form's CommandListener`.

Continuing example:

```
form.setItemListener(this);
```

9.10 A Note on Concurrency

The MIDP user interface API has been designed to be thread-safe. The methods may be called from callbacks, `TimerTasks`, or multiple threads created by the application. The MIDP system implementation handles its own locking and synchronization to make this possible.

The application is responsible for the synchronization of its own objects and data structures. Care must be taken when calling the method `Canvas.serviceRepaints` that forces the painting of the display. The `paint` method may be called by a different thread than the caller of `serviceRepaints`. If the `paint` method tries to synchronize on any object that was locked by the application when `serviceRepaints` was called, the application deadlocks. Therefore, the application should not hold any locks when it calls the method `serviceRepaints`.

The MIDP user interface API purposefully serializes callbacks to listeners and calls to the event notifications methods of class `Canvas`, so in most cases the application does not need to perform locking. The application can schedule its own activities with `Display.callSerially`. The application implements the `Runnable.run` method with the application logic and, when appropriate, the system queues this method to be called between the other events. The `run` method call requested by a call to `callSerially` is made after any pending `repaint` requests have been satisfied, so that the application can rely on the screen being up-to-date before the `run` method is invoked.