import java.io.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class TravelList
    extends MIDlet
    implements CommandListener {
    private List mList;
    private Command mExitCommand, mNextCommand;

    public TravelList() {
        String[] stringElements = { "Airplane", "Car", "Hotel" };
        Image[] imageElements = { loadImage("/res/airplane.png"),
                                  loadImage("/res/car.png"), loadImage("/res/hotel.png") };
        mList = new List("Reservation type", List.IMPLICIT,
                          stringElements, imageElements);
        mNextCommand = new Command("Next", Command.SCREEN, 0);
        mExitCommand = new Command("Exit", Command.EXIT, 0);
        mList.addCommand(mNextCommand);
        mList.addCommand(mExitCommand);
        mList.setCommandListener(this);
    }

    public void startApp() {
        Display.getDisplay(this).setCurrent(mList);
    }

    public void commandAction(Command c, Displayable s) {
        if (c == mNextCommand || c == List.SELECT_COMMAND) {
            int index = mList.getSelectedIndex();
            Alert alert = new Alert("Your selection",
                                    "You chose " + mList.getString(index) + ",",
                                    null, AlertType.INFO);
            Display.getDisplay(this).setCurrent(alert, mList);
        } else if (c == mExitCommand)
            notifyDestroyed();
    }

    public void pauseApp() {}

    public void destroyApp(boolean unconditional) {}

    private Image loadImage(String name) {
        Image image = null;
        try {
            image = Image.createImage(name);
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
        return image;
    }
}
Modified GageTracker

```java
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import com.smu.util.*;

public class GaugeTracker
    extends MIDlet
    implements ItemStateListener, CommandListener, Observer {
    private Gauge mGauge;
    private StringItem mStringItem;
    private ThreadCounter tc;
    private Thread myThread;

    public GaugeTracker() {
        int initialValue = 0;
        tc = new ThreadCounter();
        myThread = new Thread(tc);
        tc.addObserver(this);

        mGauge = new Gauge("GaugeTitle", true, 100, initialValue);
        mStringItem = new StringItem(null, "[value]");
        itemStateChanged(mGauge);
    }

    public void update(Observable obs, Object obj) {
        Integer intobj = (Integer)obj;
        int val = intobj.intValue();
        mGauge.setValue(val);
        itemStateChanged(mGauge);
    }

    public void itemStateChanged(Item item) {
        if (item == mGauge)
            mStringItem.setText("Value = " + mGauge.getValue());
    }

    public void commandAction(Command c, Displayable s) {
        if (c.getCommandType() == Command.EXIT)
            notifyDestroyed();
    }

    public void startApp() {
        Form form = new Form("GaugeTracker");
        form.addCommand(new Command("Exit", Command.EXIT, 0));
        form.setCommandListener(this);
        // Now add the selected items.
        form.append(mGauge);
        form.append(mStringItem);
        form.setItemStateListener(this);
        Display.getDisplay(this).setCurrent(form);
        //launch thread
        myThread.start();
    }

    public void pauseApp() {}
```
public class ThreadCounter extends Observable implements Runnable{

    private int counter = 0;

    /** Creates a new instance of ThreadCounter */
    public ThreadCounter() {
    }

    public int getCounter() {
        return counter;
    }

    public void run() {
        while (counter < 100) {
            try {
                Thread.sleep(300);
            } catch (InterruptedException e) { }
            counter++;
            notifyObservers(new Integer(counter) );
        }
    }
}
Ch 7 Custom Items

javax.microedition.lcdui
Class CustomItem
java.lang.Object
javax.microedition.lcdui.Item
javax.microedition.lcdui.CustomButton

public abstract class CustomItem
extends Item

A CustomItem is customizable by subclassing to introduce new visual and interactive elements into Forms. Subclasses are responsible for their visual appearance including sizing and rendering and choice of colors, fonts and graphics. Subclasses are responsible for the user interaction mode by responding to events generated by keys, pointer actions, and traversal actions. Finally, subclasses are responsible for calling Item.notifyStateChanged() to trigger notification of listeners that the CustomItem's value has changed.

Like other Items, CustomItems have the concept of minimum and preferred sizes. These pertain to the total area of the Item, which includes space for the content, label, borders, etc. See Item Sizes for a full discussion of the areas and sizes of Items.

CustomItem subclasses also have the concept of the content size, which is the size of only the content area of the CustomItem. The content area is a rectangular area inside the total area occupied by the CustomItem. The content area is the area within which the CustomItem subclass paints and receives input events. It does not include space consumed by labels and borders. The implementation is responsible for laying out, painting, and handling input events within the area of the Item that is outside the content area.

All coordinates passed between the implementation and the CustomItem subclass are relative to the item's content area, with the upper-left corner of this area being located at (0, 0). Size information passed between the implementation and the CustomItem subclass with the getMinContentSizeHeight, getMinContentSizeWidth, getPrefContentSizeHeight, getPrefContentSizeWidth, and sizeChanged methods all refer to the size of the content area. The implementation is responsible for computing and maintaining the difference between the size of the content area and the size of the total area of the Item as reported by the Item size methods Item.getMinimumHeight, Item.getMinimumWidth, Item.getPreferredHeight, and Item.getPreferredWidth.

The implementation may disregard sizing information returned from a CustomItem if it exceeds limits imposed by the implementation's user interface policy. In this case, the implementation must always report the actual size granted to the CustomItem via the sizeChanged and the paint methods. For example, this situation may occur if the implementation prohibits an Item from becoming wider than the screen. If the CustomItem subclass code returns a value from getMinContentSizeWidth that would result
in the CustomItem being wider than the screen, the implementation may assign a width smaller than the minimum width returned by getMinContentWidth.

The implementation is allowed to call the CustomItem's content size methods getMinContentHeight, getMinContentWidth, getPrefContentHeight, and getPrefContentWidth, in any order with respect to other CustomItem methods. For all of these methods, the CustomItem subclass code must return values that are consistent with the current contents of the CustomItem. If the contents changes, it is not sufficient for the CustomItem subclass code simply to begin returning different values from the content size methods. Instead, the subclass code must call the invalidate method whenever its contents changes. This indicates to the implementation that it may need to perform its layout computation, which will call the content size methods to get new values based on the CustomItem's new contents.

**Interaction Modes**

The CustomItem class is intended to allow edit-in-place on many items, but it does not allow every conceivable interaction. Desire for flexibility has been balanced against a requirement that these APIs be simple enough to master easily, along with a need to allow for platform-specific variations in look-and-feel, all without sacrificing interoperability.

The general idea is that there are multiple interaction "modes" and that the Form implementation can convey which ones it supports. The CustomItem can then choose to support one or more interaction modes. There is no requirement for a CustomItem to implement all combinations of all interaction modes. Typically, a CustomItem will implement an approach (such as the separate editing screen technique discussed below) that works on all platforms, in addition to a highly interactive approach that relies on a particular interaction mode. At run time, the CustomItem code can query the system to determine whether this interaction mode is supported. If it is, the CustomItem can use it; otherwise, it will fall back to the approach that works on all platforms.

CustomItem can always use item commands to invoke a separate editing screen, although components with a small number of discrete states could simply respond by changing the state and then causing a notifyStateChanged notification. A technique for using a separate editing screen would be to load the value into another Displayable object (such as a List) and then to call Display.setCurrent(Displayable) on it. When the user issues a command (such as "OK") to indicate that editing of this value is complete, the listener can retrieve the value from that Displayable object and then call Display.setCurrentItem(Item) to return to this item.

**Keypad Input**

The implementation may optionally support delivery of keypad events to the CustomItem. The implementation indicates the level of support by setting the KEY_PRESS, KEY_RELEASE, and KEY_REPEAT bits in the value returned by getInteractionModes. Events corresponding to these bits are delivered through calls to the keyPressed(), released(), and repeated() methods, respectively. If an
implementation supports KEY_RELEASE events, it must also support KEY_PRESS events. If an implementation supports KEY_REPEAT events, it must also support KEY_PRESS and KEY_RELEASE events. If supported, KEY_RELEASE events will generally occur after a corresponding KEY_PRESS event is received, and KEY_REPEAT events will generally occur between KEY_PRESS and KEY_RELEASE events. However, it is possible for the CustomItem to receive KEY_RELEASE or KEY_REPEAT events without a corresponding KEY_PRESS if a key is down when the CustomItem becomes visible.

Key event methods are passed the keyCode indicating the key on which the event occurred. Implementations must provide means for the user to generate events with key codes Canvas.KEY_NUM0 through Canvas.KEY_NUM9, Canvas.KEY_STAR, and Canvas.KEY_POUND. Implementations may also deliver key events for other keys, include device-specific keys. The set of keys available to a CustomItem may differ depending upon whether commands have been added to it.

The application may map key codes to game actions through use of the getGameAction method. If the implementation supports key events on CustomItems, the implementation must provide a sufficient set of key codes and a mapping to game actions such that all game actions are available to CustomItems.

The set of keys and the key events available to a CustomItem may differ from what is available on a Canvas. In particular, on a system that supports traversal, the system might use directional keys for traversal and elect not to deliver these keys to CustomItems. The mapping between key codes and game actions in a CustomItem may differ from the mapping in a Canvas. See Key Events and Game Actions on class Canvas for further information about key codes and game actions.

**Pointer Input**

The implementation may optionally support delivery of pointer events (such as taps with a stylus) to the CustomItem. The implementation indicates the level of support by setting the POINTER_PRESS, POINTER_RELEASE, and POINTER_DRAG bits in the value returned by getInteractionModes. Events corresponding to these bits are delivered through calls to the pointerPressed(), pointerReleased(), and pointerDragged() methods, respectively. If an implementation supports POINTER_RELEASE events, it must also support POINTER_PRESS events. If an implementation supports POINTER_DRAG events, it must also support POINTER_PRESS and POINTER_RELEASE events. If supported, POINTER_RELEASE events will generally occur after a corresponding POINTER_PRESS event is received, and POINTER_DRAG events will generally occur between POINTER_PRESS and POINTER_RELEASE events. However, it is possible for the CustomItem to receive POINTER_RELEASE or POINTER_DRAG events without a corresponding POINTER_PRESS if the pointer is down when the CustomItem becomes visible.

The (x, y) location of the pointer event is reported with every pointer event. This location is expressed in the coordinate system of the CustomItem, where (0, 0) is the upper-left corner of the CustomItem. Under certain circumstances, pointer events may occur outside the bounds of the item.
Traversals

An implementation may support traversal **internal** to a `CustomItem`, that is, the implementation may temporarily delegate the responsibility for traversal to the item itself. Even if there is only one traversal location inside the `CustomItem`, the item may want support the internal traversal protocol so that it can perform specialized highlighting, animation, etc. when the user has traversed into it.

The implementation indicates its support for traversal internal to a `CustomItem` by setting one or both of the TRAVERSE_HORIZONTAL or TRAVERSE_VERTICAL bits in the value returned by `getInteractionModes()`. If neither of these bits is set, the implementation is unwilling to let `CustomItems` traverse internally, or the implementation does not support traversal at all. If the implementation does support traversal but has declined to permit traversal internal to `CustomItems`, the implementation will supply its own highlighting outside the `CustomItem's` content area.

The `CustomItem` need not support internal traversal at all. It can do this by returning `false` to the initial call to the `traverse` method. (This is the default behavior if this method hasn't been overridden by the `CustomItem`.) If this occurs, the system must arrange for the user to be able to traverse onto and past this item. The system must also arrange for proper scrolling to take place, particularly if the item exceeds the height of the screen, regardless of whether internal traversal is occurring.

An implementation may provide support for delivering keypad or pointer events to `CustomItems` even if it has declined to support delivering traverse events to `CustomItems`. If an implementation provides support for delivering keypad or pointer events to `CustomItems`, it must provide a means to do so for every `CustomItem`, even for those that have refused internal traversal by returning `false` to the initial `traverse()` call. This implies that such implementations must still support some notion of focus for an item, even if that item is not supporting internal traversal.

See the documentation for the `traverse` method for a full specification of the behavior and responsibilities required for the item to perform internal traversal.

**Item Appearance**

The visual appearance of each item consists of a label (handled by the implementation) and its contents (handled by the subclass).

Labels are the responsibility of the implementation, not the item. The screen area that is allocated to the `CustomItem` for its contents is separate from the area that the implementation uses to display the `CustomItem's` label. The implementation controls the rendering of the label and its layout with respect to the content area.

The `CustomItem` is responsible for painting its contents whenever the `paint` method is called.

The colors for foreground, background, highlighted foreground, highlighted background, border, and highlighted border should be retrieved from `Display.getColor(int)`. This
will allow CustomItems to match the color scheme of other items provided with the device. The CustomItem is responsible for keeping track of its own highlighted and unhighlighted state.

The fonts used should be retrieved from Font.getFont(int). This will allow them to match the fonts used by other items on the device for a consistent visual appearance.
Example from chapter 7

```java
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class SimpleItemMIDlet
    extends MIDlet
    implements CommandListener {

    public void startApp() {
        Form form = new Form("SimpleItemMIDlet");
        form.append(new SimpleItem("SimpleItem"));

        Command c = new Command("Exit", Command.EXIT, 0);
        form.addCommand(c);
        form.setCommandListener(this);

        Display.getDisplay(this).setCurrent(form);
    }

    public void pauseApp() {}

    public void destroyApp(boolean unconditional) {}

    public void commandAction(Command c, Displayable s) {
        if (c.getCommandType() == Command.EXIT)
            notifyDestroyed();
    }
}
```

Class SimpleItem

```java
import javax.microedition.lcdui.*;

public class SimpleItem
    extends CustomItem {

    public SimpleItem(String title) { super(title); }

    // CustomItem abstract methods.
    public int getMinContentWidth() { return 100; }
    public int getMinContentHeight() { return 60; }
    public int getPrefContentWidth(int width) {
        return getMinContentWidth();
    }
    public int getPrefContentHeight(int height) {
        return getMinContentHeight();
    }
```
```java
public void paint(Graphics g, int w, int h) {
    g.drawRect(0, 0, w - 1, h - 1);
    g.setColor(0x000000ff);
    int offset = 0;
    for (int y = 4; y < h; y += 12) {
        offset = (offset + 12) % 24;
        for (int x = 4; x < w; x += 24) {
            g.fillTriangle(x + offset, y,
                            x + offset - 3, y + 6,
                            x + offset + 3, y + 6);
        }
    }
}
```
public class StationSignMIDlet
    extends MIDlet
    implements CommandListener {
    public void startApp() {
        Display display = Display.getDisplay(this);
        Form form = new Form("StationSignMIDlet");
        form.append(new StringItem("StringItem: ",
                "this is the first item");
        StationSign ss = new StationSign("Destination", display);
        ss.add("Albuquerque");
        ss.add("Savannah");
        ss.add("Pocatello");
        ss.add("Des Moines");
        form.append(ss);
        form.append(new StringItem("StringItem: ", "this is item two");
        Command c = new Command("Exit", Command.EXIT, 0);
        form.addCommand(c);
        form.setCommandListener(this);
        display.setCurrent(form);
    }
    public void pauseApp() {}
    public void destroyApp(boolean unconditional) {}
    public void commandAction(Command c, Displayable s) {
        if (c.getCommandType() == Command.EXIT)
            notifyDestroyed();
    }
}
import java.util.Vector;
import javax.microedition.lcdui.*;

public class StationSign
    extends CustomItem
    implements Runnable {
    private Vector mValues;
    private int mSelection;
    private boolean mTrucking;
    private Display mDisplay;
    private Font mFont;
    private int mVisibleIndexTimesTen;
    private boolean mFocus;

    public StationSign(String title, Display display) {
        super(title);
        mDisplay = display;
        mValues = new Vector();
        mSelection = 0;
        mTrucking = true;
        mFont = Font.getFont(Font.FONT_STATIC_TEXT);
        mVisibleIndexTimesTen = mSelection * 10;

        Thread t = new Thread(this);
        t.start();
    }

    public void add(String value) {
        if (value == null) return;
        mValues.addElement(value);
    }

    public void remove(String value) {
        if (value == null) return;
        mValues.removeElement(value);
    }

    public String getSelection() {
        if (mValues.size() == 0) return "";
        return (String)mValues.elementAt(mSelection);
    }

    public void flip() {
        mSelection++;
        if (mSelection >= mValues.size()) mSelection = 0;
    }

    public void dispose() {
        mTrucking = false;
    }
}
// Runnable interface.

public void run() {
    int target = mSelection * 10;
    while (mTrucking) {
        if (mVisibleIndexTimesTen != target) {
            mVisibleIndexTimesTen++;
            if (mVisibleIndexTimesTen >= mValues.size() * 10)
                mVisibleIndexTimesTen = 0;
            repaint();
        }
        try { Thread.sleep(50); } 
        catch (InterruptedException ie) {
        }
    }
}

// CustomItem abstract methods.

public int getMinContentWidth() {
    int maxWidth = 0;
    for (int i = 0; i < mValues.size(); i++) {
        String value = (String)mValues.elementAt(i);
        int width = mFont.stringWidth(value);
        maxWidth = Math.max(maxWidth, width);
    }
    // Don't forget about the title, although we don't
    // really know what font is used for that.
    int width = mFont.stringWidth(getLabel()) + 20;
    maxWidth = Math.max(maxWidth, width);
    return maxWidth;
}

public int getMinContentHeight() {
    return mFont.getHeight();
}

public int getPrefContentWidth(int width) {
    return getMinContentWidth();
}

public int getPrefContentHeight(int height) {
    return getMinContentHeight();
}

public void paint(Graphics g, int w, int h) {
    int fraction = mVisibleIndexTimesTen % 10;
    int visibleIndex = (mVisibleIndexTimesTen - fraction) / 10;
    String value = (String)mValues.elementAt(visibleIndex);
    g.setFont(mFont);
    int bc = mDisplay.getColor(Display.COLOR_BACKGROUND);
    int fc = mDisplay.getColor(Display.COLOR_FOREGROUND);
    if (mFocus == true) {
        bc = mDisplay.getColor(Display.COLOR_HIGHLIGHTED_BACKGROUND);
        fc = mDisplay.getColor(Display.COLOR_HIGHLIGHTED_FOREGROUND);
    }
    g.setColor(bc);
    g.fillRect(0, 0, w, h);
    g.setColor(fc);
    g.setFont(mFont);
    int bc = mDisplay.getColor(Display.COLOR_BACKGROUND);
    int fc = mDisplay.getColor(Display.COLOR_FOREGROUND);
    if (mFocus == true) {
        bc = mDisplay.getColor(Display.COLOR_HIGHLIGHTED_BACKGROUND);
        fc = mDisplay.getColor(Display.COLOR_HIGHLIGHTED_FOREGROUND);
    }
    g.setColor(bc);
    g.fillRect(0, 0, w, h);
    g.setColor(fc);
// Simple case: visibleIndex is aligned on a single item.
if (fraction == 0) {
    g.drawString(value, 0, 0, Graphics.TOP | Graphics.LEFT);
    return;
}

// Complicated case: show two items and a line.
int lineHeight = mFont.getHeight();
int divider = lineHeight - lineHeight * fraction / 10;

// Draw the piece of the visible value.
g.drawString(value, 0, divider - lineHeight,
             Graphics.TOP | Graphics.LEFT);
// Now get the next value.
visibleIndex = (visibleIndex + 1) % mValues.size();
value = (String)mValues.elementAt(visibleIndex);

// Draw the line.
g.setStrokeStyle(Graphics.DOTTED);
g.drawLine(0, divider, w, divider);

// Draw the line.
g.drawString(value, 0, divider,
             Graphics.TOP | Graphics.LEFT);

// CustomItem methods.

protected void keyPressed(int keyCode) { flip(); }

protected void pointerPressed(int x, int y) { flip(); }

protected boolean traverse(int dir,
             int viewportWidth, int viewportHeight,
             int[] visRect_inout) {
    mFocus = true;
    repaint();
    return false;
}

protected void traverseOut() {
    mFocus = false;
    repaint();
}

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.rms.RecordStoreException;

public class RecordMIDlet
    extends MIDlet
    implements CommandListener {

    private static final String kUser = "user";
    private static final String kPassword = "password";

    private Preferences mPreferences;
    private Form mForm;
    private TextField mUserField, mPasswordField;

    public RecordMIDlet() {
        try {
            // Preferences is wrapper for map of key value pairs
            // moved in and out of persistent storage
            mPreferences = new Preferences("preferences");
        } catch (RecordStoreException rse) {
            mForm = new Form("Exception");
            mForm.append(new StringItem(null, rse.toString()));
            mForm.addCommand(new Command("Exit", Command.EXIT, 0));
            mForm.setCommandListener(this);
            return;
        }

        mForm = new Form("Login");
        mUserField = new TextField("Name",
            mPreferences.get(kUser), 32, 0);
        mPasswordField = new TextField("Password",
            mPreferences.get(kPassword), 32, 0);
        mForm.append(mUserField);
        mForm.append(mPasswordField);
        mForm.addCommand(new Command("Exit", Command.EXIT, 0));
        mForm.setCommandListener(this);
    }

    public void startApp() {
        Display.getDisplay(this).setCurrent(mForm);
    }

    public void pauseApp() {} 

    public void destroyApp(boolean unconditional) {
        // Save the user name and password.
        mPreferences.put(kUser, mUserField.getString());
        mPreferences.put(kPassword, mPasswordField.getString());
        try { mPreferences.save(); } catch (RecordStoreException rse) {} 
    }
```java
public void commandAction(Command c, Displayable s) {
    if (c.getCommandType() == Command.EXIT) {
        destroyApp(true);
        notifyDestroyed();
    }
}

Preferences Class

Wrapper for key value pairs from RecordStore

import java.util.*;
import javax.microedition.lcdui.*;
import javax.microedition.rms.*;

public class Preferences {
    private String mRecordStoreName;
    private Hashtable mHashtable;

    public Preferences(String recordStoreName)
        throws RecordStoreException {
        mRecordStoreName = recordStoreName;
        mHashtable = new Hashtable();
        load();
    }

    public String get(String key) {
        return (String)mHashtable.get(key);
    }

    public void put(String key, String value) {
        if (value == null) value = "";
        mHashtable.put(key, value);
    }

    private void load() throws RecordStoreException {
        RecordStore rs = null;
        RecordEnumeration re = null;
        try {
            rs = RecordStore.openRecordStore(mRecordStoreName, true);
            re = rs.enumerateRecords(null, null, false);
            while (re.hasNextElement()) {
                byte[] raw = re.nextRecord();
                String pref = new String(raw);
                // Parse out the name.
                int index = pref.indexOf('|');
                String name = pref.substring(0, index);
                String value = pref.substring(index + 1);
                put(name, value);
            }
        } finally {
            if (re != null) re.destroy();
            if (rs != null) rs.closeRecordStore();
        }
    }
}
```
public void save() throws RecordStoreException {
    RecordStore rs = null;
    RecordEnumeration re = null;
    try {
        rs = RecordStore.openRecordStore(mRecordStoreName, true);
        re = rs.enumerateRecords(null, null, false);

        // First remove all records, a little clumsy.
        while (re.hasNextElement()) {
            int id = re.nextRecordId();
            rs.deleteRecord(id);
        }

        // Now save the preferences records.
        Enumeration keys = mHashtable.keys();
        while (keys.hasMoreElements()) {
            String key = (String)keys.nextElement();
            String value = get(key);
            String pref = key + "|" + value;
            byte[] raw = pref.getBytes();
            rs.addRecord(raw, 0, raw.length);
        }
    } finally {
        if (re != null) re.destroy();
        if (rs != null) rs.closeRecordStore();
    }
}