MIDlet Code Overview and Guide for Image Demo MIDlet

Last Updated: July 27, 2010

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 526-0883

Text Part Number: <Part Number>

Preface

Overview

MIDlet Code Overview and Guide for Image Demo MIDlet, Version 1.0 provides you an introduction to the sample MIDlet code included with the Image Demo MIDlet. After reading this guide, you should have a good understanding of how the Image Demo MIDlet is designed and how to build your own MIDlet using Images both locally and via a URL.

Audience

This document is targeted at developers who wish to develop MIDlet applications for the Cisco Unified IP Phones, specifically the CP-7926G. While this document is targeted at developers with no previous experience with MIDlets, developers of MIDlets on other platforms will find the information contained useful in learning how to leverage the 7926 platform. See the Related Documentation section on page iv for a list of related documentation.

Organization

Part	***Description***
Chapter 1, "Overview of Image Demo MIDlet"	*Provides an overview of the MIDlets design and purpose. In addition, it provides a list of the features exhibited.*
Chapter 2, "Use Cases for Image Demo MIDlet"	*Provides a possible use case, with screenshots, for using this MIDlet.*
Chapter 3, "Line-by-line Reference Guide"	*Describes the code behavior on a line-by-line basis.*

General MIDlet Information

The topics listed below are not covered in this document. Please see the corresponding chapters in the Java MIDlet Developer Guide for more information.

- Java IDE and Emulators. (Chapter 4)
- Installing, configuring and provisioning MIDlet services. (Chapter 6)
- Debugging and Troubleshooting Techniques. (Chapter 10)

Related Documentation

For more information about the Cisco Unified Communications Manager or Cisco Unified IP Phones, refer to the following publications:

- *Cisco Unified Communications Manager Administration Guide, Release 8.0(1), which can be found at this URL:*
  http://www.cisco.com/en/US/products/sw/voicesw/ps556/prod_maintenance_guides_list.html
- *Cisco Unified Communications Manager System Guide, Release 8.0(1)), which can be found at this URL:*
  http://www.cisco.com/en/US/products/sw/voicesw/ps556/prod_maintenance_guides_list.html

Providing Feedback

If you have comments or questions regarding this document please provide your feedback to the following Cisco Developer Support email alias:

j2me-api-beta@cisco.com

If you have questions regarding the usage of this sample application or about developing for this platform, please email to the following Cisco Developer Support email alias:

7926-midlets@cisco.com

Document Conventions

Convention	Description
boldface font	Commands and keywords are in **boldface**.
italic font	Arguments for which you supply values are in *italics*.
[ ]	Elements in square brackets are optional.
{ x | y | z }	Alternative keywords are grouped in braces and separated by vertical bars.
[x | y | z]	Optional alternative keywords are grouped in braces and separated by vertical bars.
string	A nonquoted set of characters. Do not use quotation marks around the string or the string will include the quotation marks.
screen font	Terminal sessions and information the system displays are in screen font.
boldface screen font	Information you must enter is in **boldface screen** font.
italic screen font	Arguments for which you must supply values are in *italic screen* font.
→	This pointer highlights an important line of text in an example.
^	The symbol ^ represents the key labeled Control—for example, the key combination ^D in a screen display means you hold down the Control key while you press the D key.
< >	Nonprinting characters, such as passwords, are in angle brackets.

Note Means *reader take note*. Notes contain helpful suggestions or references to material not covered in the publication.

Caution Means *reader be careful*. In this situation, you might do something that could result in equipment damage or loss of data.

Warning Means *danger*. You are in a situation that could cause bodily injury. Before you work on any equipment, be aware of the hazards involved with electrical circuitry and be familiar with standard practices for preventing accidents.

CHAPTER 1

Overview of Image Demo MIDlet

This chapter provides an overview of the Image Demo MIDlet. This sample application is intended to provide you, the developer, with useful examples and directions for incorporating images into your applications.

Upon starting this application two images are displayed on to the screen. One image is the default background image for the Cisco 7926 phone and is loaded from a file. The second image is a PNG format image that is located on a website and is loaded via its URL.

The application also has a slideshow demonstration that can be viewed by pressing the “Slideshow” Soft Key. This screen loads multiple images from files and displays them onto the screen in a slideshow format. Each image is displayed onto the screen for a predetermined time and then the next image in the list is displayed on top of it. Pressing a button on the phone during the slideshow will stop the slideshow and allow for manual changing of images.

Features

This is a list of the functions and features included in the Image Demo MIDlet.

- Display an Image From a File
- Display an Image From a URL
- Image Slideshow Capability
- Image Error Handling

Note This MIDlet requires require the following minimum software versions:

- Cisco Unified Communications Manager version 7.1.3
- Cisco Unified IP Phone 7926 Firmware 1.4.1.JAVA

Note After a slideshow has been stopped it can be started again by hitting the “Back” Soft Key in the slideshow display and reopening the slideshow.

CHAPTER 2

Use Cases

This chapter lists will show you the interaction a user might have with this MIDlet.

Step 1	The user opens the MIDlet by selecting it from the Services Menu. 
Step 2	An image loaded from a URL is displayed on top of an image loaded from a file. 

Step 3	The user presses the “Slideshow” Soft Key to navigate to the slideshow screen. 
Step 4	The slideshow loads the second image on top of the first image. 
Step 5	After 5 seconds the slideshow loads the next image on top of the last image. 
Step 6	Slideshow continues to run until the user exits or goes back to the beginning screen.

CHAPTER 3

Line-by-line Reference Guide

This chapter walks through the code for Image Demo MIDlet line-by-line. The function and usage of each line is explained. In addition, best practices for coding are mentioned where appropriate.

ImageDemoMIDlet.java

This class is the main MIDlet. It handles launching the application and creating the display. This class manages the lifecycle of the application, responding to pause and exit requests from the platform.

```
1   /*
2    * Copyright (c) 2010 Cisco Systems, Inc.
3    * All Rights Reserved.
4    */
5
6   package com.cisco.sdk.image;
7
8   import javax.microedition.lcdui.Alert;
9   import javax.microedition.lcdui.Display;
10  import javax.microedition.lcdui.Displayable;
11  import javax.microedition.midlet.*;
12
13  /**
14   * @author Michael Drewett (mdrewett@cisco.com)
15   */
16  public class ImageDemoMIDlet extends MIDlet {
17
18      private boolean midletPaused = false;
19
20      private ImageCanvas imageCanvas;
21
22       /**
23        * The ImageDemoMIDlet constructor.
24        */
25      public ImageDemoMIDlet(){
26      }
```

Default constructor for the ImageDemoMIDlet class. Since this class is simple and doesn't need any initialization during construction this is left blank.

```
27
28          /**
29           * Initilizes the application.
30           * It is called only once when the MIDlet is started. The method is called before the startMIDlet method.
31           */
32          public void initialize(){
33          }
```

This function is used for initializing variables and other information before starting the MIDlet. This sample application does not need to initialize anything.

```
34
35          /**
36          * Performs an action assigned to the Mobile Device - MIDlet Started point.
37          */
38          public void startMIDlet() {
39              if (imageCanvas == null){
40                  imageCanvas = new ImageCanvas(this);
41              }
42
43              switchDisplayable(null, imageCanvas);
44          }
```

This function is called when the MIDlet is started; it creates an *ImageCanvas* class and displays it on the screen.

```
45
46          /**
47          * Performs an action assigned to the Mobile Device - MIDlet Resumed point.
48          */
49          public void resumeMIDlet() {
50          }
```

This function performs an action when a MIDlet resumes from being paused, this sample application does not require any actions to be performed on a resume.

```
51
52          /**
53          * Switches a current displayable in a display. The display instance is taken from getDisplay method. This method is used by all actions
        in the design for switching displayable.
54          * @param alert the Alert which is temporarily set to the display; if null, then nextDisplayable is set immediately
55          * @param nextDisplayable the Displayable to be set
56          */
57          public void switchDisplayable(Alert alert, Displayable nextDisplayable) {
58              Display display = getDisplay();
59              if (alert == null) {
60                  display.setCurrent(nextDisplayable);
61              } else {
62                  display.setCurrent(alert, nextDisplayable);
63              }
64          }
```

This function can be used to switch the current displayable or set an alert on to the display.

```
65
66          /**
67           * Returns a display instance.
68           * @return the display instance.
69           */
70          public Display getDisplay () {
71              return Display.getDisplay(this);
72          }
73
74          /**
75          * Exits MIDlet.
76          */
77          public void exitMIDlet() {
78              switchDisplayable(null, null);
79              destroyApp(true);
```

```
80              notifyDestroyed();
81          }
```

This function is called when a MIDlet is exiting. It removes the displayable from the display and signals to destroyApp() that it is okay to destroy the MIDlet. Finally notifyDestroyed() is called to ensure that the MIDlet is destroyed.

```
82
83          /**
84          * Called when MIDlet is started.
85          * Checks whether the MIDlet have been already started and initialize/starts or resumes the MIDlet.
86          */
87          public void startApp() {
88              if (midletPaused){
89                  resumeMIDlet();
90              }else{
91                  initialize();
92                  startMIDlet();
93              }
94              midletPaused = false;
95          }
```

This function is used to start an application and resume from being paused. Since startApp() is called both when a MIDlet is first created and when it resumes after being paused we make sure that the initialize() function is called only once.

```
96
97          /**
98          * Called when MIDlet is paused.
99          */
100         public void pauseApp() {
101             midletPaused = true;
102         }
```

This function is called before the MIDlet is paused. This sample MIDlet simply updates the midletPaused Boolean.

```
103
104         /**
105         * Called to signal the MIDlet to terminate.
106         * @param unconditional if true, then the MIDlet has to be unconditionally terminated and all resources has to be released.
107         */
108         public void destroyApp(boolean unconditional) {
109             ImageLoader.destroyImages();
110         }
```

This function is called to signal the MIDlet to terminate. This sample application calls the destroyImages() method in the ImageLoader class before terminating.

```
111     }
```

ImageCanvas.java

This class extends the Canvas class to create a display for the MIDlet. It loads an image from a file and an image from a URL and displays them onto the canvas. Additionally, it handles soft key input and allows navigation to the Slideshow display.

```
1               /*
2           * Copyright (c) 2010 Cisco Systems, Inc.
3           * All Rights Reserved.
4           */
5
6           package com.cisco.sdk.image;
7
8           import java.io.IOException;
```

```
9   import javax.microedition.lcdui.*;
10
11  /**
12   * @author Michael Drewett (mdrewett@cisco.com)
13   */
14  public class ImageCanvas extends Canvas implements CommandListener, ImageLoaderListener {
```

ImageCanvas extends the Canvas class in order to display the images to specific locations on the screen. It implements the ImageLoaderListener class in order to utilize the ImageLoader class for loading an image from a URL.

```
15      private ImageDemoMIDlet parentMIDlet;
16      private SlideshowCanvas slideCanvas;
17
18      private Command exitCommand;
19      private Command slideCommand;
20
21      private int backgroundColor;
22      private int textColor;
23
24      private Image testImage;
25      private Image urlImage;
26
27      private final String TEST_IMAGE_NAME = "/_imgs/background.png";    // Filename for local image
28      private final String IMAGE_URL = "http://www.cisco.com/web/fw/i/netpro.png";    // URL for remote image
29
30      /**
31       * ImageCanvas constructor
32       * @param parentMIDlet the parent MIDlet from where the displayable is shown
33       */
34      public ImageCanvas(ImageDemoMIDlet parentMIDlet) {
35          this.parentMIDlet = parentMIDlet;
36
37          //Get default colors from the device
38          Display defaultDisplay = Display.getDisplay(parentMIDlet);
39          backgroundColor = defaultDisplay.getColor(Display.COLOR_BACKGROUND);
40          textColor = defaultDisplay.getColor(Display.COLOR_FOREGROUND);
```

The getColor() function is used to get the appropriate color. 7925 colors for background and foreground are white and black

```
41
42          //Create softkeys
43          exitCommand = new Command("Exit", Command.EXIT, 1);
44          slideCommand = new Command("Slideshow", Command.OK, 1);
```

These commands create Soft Keys that can be added to the canvas for navigation.

```
45
46
47          try{
48              // Load background image from file
49              testImage = ImageLoader.getInstance().getImageFromFile(TEST_IMAGE_NAME);
```

This uses the ImageLoader class to load an image from a file; the parameter is the name of the file without the .png extension. For this example TEST_IMAGE_NAME is “background” or a copy of the background image for the phone.

```
50              // Load icon image from URL, supports png files
51              ImageLoader.getInstance().getImageFromUrl(this, IMAGE_URL);
```

This uses the ImageLoader class to load an image from a URL; the parameter is the full URL of the file including the .png extension. This example uses the icon found at http://www.cisco.com/web/fw/i/netpro.png.

```
52          }
53          catch(IOException ioe){
```

```
54              System.err.println("Error loading images");
55          }
56
57          setTitle("Image Demo");
```

This command sets the canvas title bar to display “Image Demo.”

```
58
59          // Set up this canvas to listen to command events
60          setCommandListener(this);
61          addCommand(exitCommand);
62          addCommand(slideCommand);
63      }
```

These lines add the Soft Keys to the display.

```
64
65      /**
66       * Image Load Listener for URL Image.
67       * @param image The image loaded from a URL
68       */
69      public void imageLoaded(Image image){
70          urlImage = image;
71          repaint();
72      }
```

This function is called after an image has been downloaded from the URL. Since there is some delay in downloading an image, you do not want to lock the main application thread. The ImageLoader class works in the background and calls ImageLoaderListener which in turn calls this when it completes.

```
73
74      /**
75       * Draw items onto the canvas
76       * @param g the Graphics object to be painted.
77       */
78      public void paint(Graphics g) {
79          //Get Device height and width
80          int height = getHeight();
81          int width = getWidth();
```

The commands getHeight() and getWidth() return the height and width of the paintable area in pixels.

```
82
83          // Set and draw background
84          g.setColor(backgroundColor);
85          g.fillRect(0, 0, width, height);
```

These commands set the Graphics object g’s color to white and draws a background that is the same size as the screen.

```
86
87          // Draw the test image onto the screen
88          if (testImage != null)
89              g.drawImage(testImage, 0, 0, Graphics.TOP|Graphics.LEFT);
90
91          // Draw the url image onto the screen
92          if (urlImage != null)
93              g.drawImage(urlImage, 0, 0, Graphics.TOP|Graphics.LEFT);
```

This draws the file-based image and the URL-based image onto the canvas if they have been loaded successfully. The URL image will be placed on top of the file-based image.

```
94
95
96          // Set and draw text
97          g.setColor(textColor);
98      }
```

```
99
100         /**
101          * Handle actions for hard keys being pressed
102          * @param keyCode the key code of the pressed key
103          */
104         protected  void keyPressed(int keyCode) {
105         }
106
107         /**
108          * Handle actions for hard keys being released
109          * @param keyCode the key code of the released key
110          */
111         protected  void keyReleased(int keyCode) {
112         }
113
114         /**
115          * Handle actions for hard keys being repeated (held down)
116          * @param keyCode the key code of the repeated key
117          */
118         protected  void keyRepeated(int keyCode) {
119         }
```

These functions allow for handling of key press, key release, and key repeated events. This canvas does not use key events; the only interaction is by the use of the Soft Keys.

```
120
121         /**
122          * Handle softkey press events.
123          * @param command the command button being pressed
124          * @param displayable current displayable
125          */
126         public void commandAction(Command command, Displayable displayable) {
```

The function commandAction() is called whenever a Soft Key button is pressed. This function is used specify the functionality for each Soft Key.

```
127             if (command == slideCommand){
128                 // Construct a new canvas for a slideshow
129                 if (slideCanvas == null){
130                     slideCanvas = new SlideshowCanvas(parentMIDlet);
131                 }
132
133                 // Start the slideshow timer
134                 slideCanvas.startTimer();
135                 Display.getDisplay(parentMIDlet).setCurrent(slideCanvas);
```

This block creates a new SlideshowCanvas object if one has not already been created and then displays it when the “Slideshow” Soft Key is pressed. The timer in the SlideshowCanvas is started to signal the start of the slideshow.

```
136             }
137             else if (command == exitCommand){
138                 parentMIDlet.exitMIDlet();
139             }
```

This block calls the exitMIDlet() function in the PlatformMIDlet in order to exit the MIDlet.

```
140         }
141     }
```

SlideshowCanvas.java

This class extends the Canvas class to create a secondary display for the MIDlet. It loads a series of images from files and displays them in a slideshow format. A timer is used to trigger the changing of each image.

```
1                   /*
2            * Copyright (c) 2010 Cisco Systems, Inc.
3            * All Rights Reserved.
4            */
5
6           package com.cisco.sdk.image;
7
8           import java.io.IOException;
9           import java.util.Timer;
10          import java.util.TimerTask;
11          import java.util.Vector;
12          import javax.microedition.lcdui.*;
13
14          /**
15           * @author Michael Drewett (mdrewett@cisco.com)
16           */
17          public class SlideshowCanvas extends Canvas implements CommandListener {
```

SlideshowCanvas also extends Canvas in order to allow for displaying images to the screen.

```
18              private ImageDemoMIDlet parentMIDlet;
19              protected Displayable previousDisplay;
20
21              private Command exitCommand;
22              private Command backCommand;
23
24              private Vector pictures = null;
25              private int nextPicture = 0;
26
27              private Image slideImage;
28              private Timer updateTimer = null;
29              private final static int TIMER_DELAY = 5 * 1000; // 5 seconds
30              private boolean timerOn = false;
```

Variables for the timer used to update the image on the display; the time between images is set to 5 seconds.

```
31
32              /**
33               * SlideshowCanvas constructor
34               * @param parentMIDlet the parent MIDlet from where the displayable is shown
35               */
36              public SlideshowCanvas(ImageDemoMIDlet parentMIDlet) {
37                  this.parentMIDlet = parentMIDlet;
38                  previousDisplay = Display.getDisplay(parentMIDlet).getCurrent();
39
40                  // Create vector for slideshow pictures and populate
41                  pictures = new Vector();
42                  loadPictures();
43
44                  //Create softkeys
45                  exitCommand = new Command("Exit", Command.EXIT, 1);
46                  backCommand = new Command("Back", Command.BACK, 1);
47
48                  setTitle("Slideshow Demo");
49
50                  // Set up this canvas to listen to command events
51                  setCommandListener(this);
52                  // Add the Exit command
53                  addCommand(exitCommand);
54                  addCommand(backCommand);
55              }
56
57              /**
58               * Load all pictures that will be in the slideshow
59               */
60              public void loadPictures(){
61                  try{
62                      // Load two local images; old logo and new cisco logo
```

```
63              Image loadImage = ImageLoader.getInstance().getImageFromFile("/_imgs/cisco.png");
64              pictures.addElement(loadImage);
65              loadImage = ImageLoader.getInstance().getImageFromFile("/_imgs/oldCisco.png");
66              pictures.addElement(loadImage);
67          }
```

Function used to load all the images to be displayed by the slideshow. For this example we use two versions of the Cisco company logo. Images are stored in a vector to allow for a dynamic number of images.

```
68          catch (IOException ioe){
69              System.err.println("Could not load images!");
70          }
71
72      }
73
74      /**
75       * Create a timer to be used for updating the slideshow
76       */
77      public void startTimer(){
78          // Start a timer and begin slideshow
79          updateTimer = new Timer();
80          PictureUpdater updateTask = new PictureUpdater();
81          updateTimer.schedule(updateTask, 0, TIMER_DELAY);
82          timerOn = true;
83      }
```

A timer is used to allow for the updating of images every 5 seconds. The PictureUpdater class contains the actions to be performed when the timer executes (see line 165).

```
84
85      /**
86       * End the timer
87       */
88      public void endTimer(){
89          updateTimer.cancel();
90          timerOn = false;
91      }
```

Function to end the timer when it is no longer in use.

```
92
93      /**
94       * Choose the next picture and draw it to the canvas
95       */
96      public void drawNextPicture(){
97          System.out.println("Displaying image number: " + nextPicture);
98
99          // Choose next picture and repaint
100         slideImage = (Image) pictures.elementAt(nextPicture);
101         repaint();
```

Set the next image to be displayed and call the repaint() function so that it is drawn to the screen.

```
102         nextPicture = (nextPicture + 1) % pictures.size();
```

Select the next image to be displayed; this example starts over at the beginning of the vector of images when each image has been displayed.

```
103     }
104
105     /**
106      * Draw items onto the canvas
107      * @param g the Graphics object to be painted.
108      */
109     public void paint(Graphics g) {
110         //Get Device height and width
```

```
111             int height = getHeight();
112             int width = getWidth();
113
114             g.setColor(255, 255, 255);
115             g.fillRect(0, 0, width, height);
116
117             // Draw current image on screen
118             if (slideImage != null)
119                 g.drawImage(slideImage, 0, 0, Graphics.TOP|Graphics.LEFT);
120         }
```

paint() method for the canvas, simply draw the current image in the slideshow.

```
121
122         /**
123          * Handle actions for hard keys being pressed
124          * @param keyCode the key code of the pressed key
125          */
126         protected  void keyPressed(int keyCode) {
127             if (timerOn)
128                 endTimer();
129             drawNextPicture();
130         }
```

If a key is pressed we stop the slideshow and advance to the next picture.

```
131
132         /**
133          * Handle actions for hard keys being released
134          * @param keyCode the key code of the released key
135          */
136         protected  void keyReleased(int keyCode) {
137         }
138
139         /**
140          * Handle actions for hard keys being repeated (held down)
141          * @param keyCode the key code of the repeated key
142          */
143         protected  void keyRepeated(int keyCode) {
144             if (timerOn)
145                 endTimer();
146             drawNextPicture();
147         }
```

If a key is repeated we stop the timer and advance through the images.

```
148
149         /**
150          * Handle softkey press events.
151          * @param command the command button being pressed
152          * @param displayable current displayable
153          */
154         public void commandAction(Command command, Displayable displayable) {
155             if (command == exitCommand){
156                 endTimer();
157                 updateTimer = null;
158                 parentMIDlet.exitMIDlet();
159             }
```

Before exiting we end the timer and set it to null.

```
160             else if (command == backCommand){
161                 // Return display to previous canvas
162                 Display.getDisplay(parentMIDlet).setCurrent(previousDisplay);
163                 endTimer();
164                 if (previousDisplay == null)
165                     parentMIDlet.notifyDestroyed();
166             }
```

Return display to the previous canvas.

```
167         }
168
169         /**
170          * PictureUpdater class used to redraw canvas for slideshow
171          */
172         public class PictureUpdater extends TimerTask {
```

PictureUpdater extends TimerTask to allow for a task to be performed when a timer is fired.

```
173
174             /**
175              * Task to be performed whenever timer is ran
176              */
177             public void run() {
178                 drawNextPicture();
179             }
```

When the timer is fired, draw the next picture.

```
180         }
181     }
```

ImageLoader.java

This class abstracts the creation of images from the rest of the MIDlet. This can be used to manage the download and reuse of images. Due to the limited memory on the 7926, the same image should not be created more than once. By using this class to create and get images, you assure the application only maintains one copy of each image.

```
1               /**
2        * Copyright (c) 2009 Cisco Systems, Inc.
3        * All Rights Reserved.
4        */
5       package com.cisco.sdk.image;
6
7       import javax.microedition.lcdui.Image;
8       import java.util.Hashtable;
9       import java.io.IOException;
10      import java.util.Vector;
11
12      /**
13       * Provides Image loading and guarantees that an image is only loaded once
14       * each time the app is run. Also provides for image unloading.
15       * @author Riley Marsh (rimarsh@cisco.com)
16       */
17      public class ImageLoader implements HttpRequestListener {
```

This class implements the *HttpRequestListener* so that it can receive the raw data from the URL and convert it into an image. That image will be stored for future use and returned to the calling application.

```
18
19          private static ImageLoader singleton = null;
```

The *ImageLoader* class uses a singleton pattern so that any object in the MIDlet can access images and be sure that it has only been created once. This will save processing time on downloading or creating the image as well as memory in storage of the image.

```
20          private Hashtable images = null;
21          private Hashtable listeners = null;
22
23          private ImageLoader() {
24                  images = new Hashtable();
25                  listeners = new Hashtable();
```

```
26          }
```

Create two Hashtable classes, one to store the images created. This table uses the URL or path of the image requested as the key and stores the Image object as the value. The second Hashtable stores the objects requesting an image and the image they are requesting. This is so that the *ImageLoader* can call the correct listener when an image is loaded.

```
27
28      /**
29       * Retrieve an ImageLoader instance. This method creates an ImageLoader
30       * if one does not already exist, otherwise it returns the previous
31       * instance.
32       * @return The ImageLoader instance to be used for fetching images.
33       */
34      public static ImageLoader getInstance() {
35          // Create an instance if necessary
36          if (singleton == null) {
37                  synchronized (ImageLoader.class) {
38                          if (singleton == null) {
39                                  System.err.println("ImageLoader> Singleton Created");
40                                  singleton = new ImageLoader();
41                          }
42                  }
43          }
44          return singleton;
45      }
```

Since the singleton should only be created once, the *getInstance()* method checks if the singleton has been created, then it locks the class and verifies that the singleton still has not been created. Once sure, it creates a new instance of the *ImageLoader*.

```
46
47          /**
48           * Attempts to load an image from the specified imageName. If successful,
49           * the loaded image is stored for future use. This reduces the overhead of
50           * multiple creations of the same image.
51           * @param imageName The full path of the image to be loaded, relative to
52           * the base directory of the JAR file.
53           * @return An Image, ready for display.
54           * @throws IOException If the image file is not found or fails to load.
55           */
56          public Image getImageFromFile(String imageName) throws IOException {
57                  Image img = (Image)images.get(imageName);
```

When the *getImageFromFile()* method is called, the class checks if the image has already been loaded by checking the Hashtable of images. If it has, the class simply returns the existing image.

```
58                  // Check if the image has been loaded
59                  if (img == null) {
60                          // If not load the image
61                          try {
62                                  System.err.println("ImageLoader> Loading Image: " + imageName);
63                                  images.put(imageName, Image.createImage(imageName));
64                          } catch (IOException ioe) {
65                                  System.err.println("ImageLoader> Image not found: " + imageName);
66                                  throw new IOException("Image not found: " + imageName);
67                          }
68                          // Once loaded fetch the image.
69                          img = (Image) images.get(imageName);
70                  }
```

If it has not, the class will attempt to load the image from a file path. If it fails, it will throw an exception. Otherwise the loaded image is stored in the image Hashtable and returned to the calling object.

```
71                  return img;
72          }
73
74          /**
```

```
75          * Attempts to load an image form a URL. This method will first check
76          * if there exists a locally available copy of the image to skip
77          * downloading. If not it will start a HttpRequest thread.
78          * @param listener The object wishing to be notified when loading
79          * completes. This object must implement ImageloaderListener.
80          * @param imageUrl The URL of the image requested. This should be the full
81          * URL, ie. http://www.cisco.com/web/fw/i/netpro.png.
82          * @throws IllegalArgumentException If listener or imageUrl is null.
83          */
84         public void getImageFromUrl(ImageLoaderListener listener, String imageUrl) {
85             if (listener == null) {
86                 System.err.println("ImageLoader> Listner is null, aborting.");
87                 throw new IllegalArgumentException("ImageLoaderListener cannot be null.");
88             }
89             if (imageUrl == null || imageUrl.equals("")) {
90                 System.err.println("ImageLoader> ImageURL is null, aborting.");
91                 throw new IllegalArgumentException("ImageURL cannot be null.");
92             }
93             // Check if the image has been loaded
```

This method loads an image from an Http URL. The calling object must register an *ImageLoaderListener* to receive notification when the Image is loaded. The method will handle whether a download is necessary or if the image is locally stored.

```
94             if ( images.containsKey(imageUrl) ) {
95                 // Image has been loaded
96                 listener.imageLoaded( (Image)images.get(imageUrl) );
97                 return;
98             }
```

First, the method checks if the image has already been downloaded and returns it to the listener if that is the case.

```
99             boolean sendImage = false;
100            synchronized (ImageLoader.class) {
```

Second, if the image has not been downloaded before, it must be downloaded. Synchronizing this code ensures that only one thread will do the downloading of an image.

```
101                if (images.containsKey(imageUrl)) {
102                    // Image has been loaded since entering sync block
103                    sendImage = true;
104                }
```

Verify that no other thread was in the process of downloading the image when we last checked.

```
105                if (listeners.containsKey(imageUrl)) {
106                    // Image is being loaded in another thread
107                    // add to the interested list
108                    ((Vector) listeners.get(imageUrl)).addElement(listener);
```

Then the code verifies that no other thread is currently downloading the image. If another thread is working on it, this thread can simply register its interest so that it is alerted when the download completes.

```
109                } else {
110                    // If not load the image
111                    listeners.put(imageUrl, new Vector());
112                    ((Vector) listeners.get(imageUrl)).addElement(listener);
113                    HttpRequest request = new HttpRequest(this, imageUrl);
114                    request.start();
115                }
```

Lastly, if the image has not been downloaded or requested, the method starts a new thread to handle the download.

```
116            }
```

```
117             if (sendImage) {
118                 listener.imageLoaded((Image) images.get(imageUrl));
119             }
```

If an image has been downloaded, notify the listener. Otherwise, listeners will be notified when the download thread completes.

```
120         }
121
122         /**
123          * The listener for a HTTP response. Handles the conversion from a
124          * byte array to an Image.
125          * @param data The data retured from the webserver.
126          * @param responseURL The url where of the webserver.
127          */
128         public synchronized void httpRequestAction(byte[] data, String responseURL) {
```

Once the Http data is downloaded, this method is called. The raw byte array data is returned along with the URL of the image requested. This is used for associating the response with the requesting object.

```
129             if (responseURL == null) {
130                 // Since the responseURL is null, there is no way to know
131                 // where to send the data or error.
132                 System.err.println("ImageLoader> ResponseURL is null, aborting.");
133                 return;
134             }
135
136             Image image = null;
137             // Check for bad data.
138             if (data != null && data.length > 0) {
139                 try {
140                     System.out.println("ImageLoader> Loading Image: " + responseURL);
141                     image = Image.createImage(data, 0, data.length);
```

Creating an image from a byte array is straight forward.

```
142                 } catch (IllegalArgumentException e) {
143                     System.err.println("ImageLoader> Error loading image: " + responseURL);
144                     image = null;
145                 }
146             }
147
148             // If the image is loaded successfully, store it for future use.
149             if (image != null) {
150                 images.put(responseURL, image);
151             }
```

This method will store the image for future use. This helps in reducing the bandwidth requirements of an application and removing wasted effort.

```
152
153             // Notify all listeners with image binary
154             Vector v = ((Vector) listeners.remove(responseURL));
155             if (v == null || v.size() == 0) {
156                 // No listeners
157                 System.err.println("ImageLoader> Image loaded, but no listeners for " + responseURL);
158                 return;
159             }
160
161             // Alert the listeners.
162             for (int i = 0; i < v.size(); i++) {
163                 ((ImageLoaderListener) v.elementAt(i)).imageLoaded(image);
164             }
```

Lastly, the method notifies all listeners of the successful download and provides them with the Image.

```
165         }
166
```

```
167         /**
168          * Removes an image from internal storage. This can be used to
169          * clean up the application memory space.
170          * @param imageName The complete path of the Image to be removed.
171          * @return The removed Image if it exists, otherwise null.
172          */
173         public Image removeImage(String imageName) {
174                 if (imageName == null) {
175                         return null;
176                 }
177                 System.out.println("ImageLoader> Removing Image: " + imageName);
178                 return (Image) images.remove(imageName);
179         }
180
181         /**
182          * Removes all images and cleans up the ImageLoader instance.
183          */
184         public static void destroyImages() {
185                 System.out.println("ImageLoader> Destroying Images");
186                 if (singleton != null) {
187                         synchronized (ImageLoader.class) {
188                                 if (singleton != null) {
189                                         singleton.images = null;
190                                         singleton = null;
191                                 }
192                         }
193                 }
194         }
195 }
```

HttpRequest.java

This class issues a request for a given URL and parses the response. It verifies that the response is valid, and then calls the HttpRequestListener with the response data.

```
1                   /*
2          * Copyright (c) 2010 Cisco Systems, Inc.
3          * All Rights Reserved.
4          */
5         package com.cisco.sdk.image;
6
7         import java.io.ByteArrayOutputStream;
8         import java.io.IOException;
9         import java.io.InputStream;
10        import javax.microedition.io.ConnectionNotFoundException;
11        import javax.microedition.io.Connector;
12        import javax.microedition.io.HttpConnection;
13
14        /**
15        * Provides Http download mechanizm. It packages the requesting and
16         * status checking of getting an http resource.
17         * @author Riley Marsh (rimarsh@cisco.com)
18         */
19        public class HttpRequest extends Thread {
20
21            // Listener to recieve data once downloaded.
22            private HttpRequestListener listener = null;
23            // URL to download data from.
24            private String requestUrl = null;
25
26            /**
27             * Construct a new HttpRequest object.
28             * Call start() method to begin download.
29             * @param listener The object to notify with the downloaded data. This
30             * object must implement HttpRequestListener.
31             * @param requestUrl The URL of the resource to download.
32             * @throws IllegalArgumentException If listener or requestUrl is null.
33             */
```

```
34      public HttpRequest(HttpRequestListener listener, String requestUrl) {
```

The HttpRequest class takes a single listener to URL mapping. One listener object will provide a URL to be downloaded. While this class is only used for downloading images in this MIDlet, it could be used for downloading any http page. Since HttpConnections can be slow, this class creates a new thread for the processing and returns the resulting data to a listener when it is complete.

```
35              super();
36
37              // Validate the input
38              if (listener == null) {
39                      System.err.println("HttpRequest> Listner is null, aborting.");
40                      throw new IllegalArgumentException("HttpRequestListener cannot be null.");
41              }
42              if (requestUrl == null || requestUrl.equals("")) {
43                      System.err.println("HttpRequest> RequestUrl is null, aborting.");
44                      throw new IllegalArgumentException("RequestUrl cannot be null.");
45              }
46
47              // Store the listener and requestUrl
48              this.listener = listener;
49              this.requestUrl = requestUrl;
50      }
51
52      /**
53       * Overriden to issue the HTTP Request when the thread starts. Listeners
54       * are called when a result is downloaded.
55       */
56      public void run() {
57              System.out.println("HttpRequest> Fetching data from " + this.requestUrl);
58              byte[] data = null;
59              HttpConnection connection = null;
60              InputStream inputStream = null;
61              int responseCode;
62
63              try {
64                      // Open a connection as an HttpConnection
65                      // Using HttpConnection gives us convinence features
66                      // such as getResonseCode() used below.
67                      connection = (HttpConnection) Connector.open(this.requestUrl);
```

This call opens the connection to the URL and creates a Connection class. Here the Connection is cast to an HttpConnection, this tells the connection that the resource should be treated as an Http URL. It provides extra features such as response code verification and header parsing. Using this class simplifies the connection.

```
68
69                      // Getting the response code will open the connection,
70                      // send the request, and read the HTTP response headers.
71                      // The headers are stored until requested.
72                      responseCode = connection.getResponseCode();
73                      if (responseCode != HttpConnection.HTTP_OK) {
74                              System.err.println("HttpRequest>  HTTP response code: " + responseCode);
75                              data = null;
76                      } else {
77                              // Open the inputStream
78                              inputStream = connection.openInputStream();
```

After verifying the response is a valid status, the class prepares to read data from the URL.

```
79
80                              // Determine if the program should return when
81                              // content-length inputStream reached or continue to EOD.
82                              boolean contentLengthSpecified = true;
83                              // Get the content-length if specified
84                              int contentLength = (int) connection.getLength();
85                              // If content-length not found, set to go until end of stream.
86                              if (contentLength == -1) {
87                                      contentLengthSpecified = false;
```

```
88                          contentLength = Integer.MAX_VALUE;
89                  }
```

If the http page is properly formatted, it will have a content-length specified. If this is available, this class will use that as the point to stop reading the URL. However, if that data is not specified, this class will read until the end of the document.

```
90
91                  if (contentLength > 0) {
92                          // If length inputStream valid
93                          int actual = 0; // count of bytes read
94                          ByteArrayOutputStream downloadStream = new ByteArrayOutputStream();
95                          while (true) {
96                                  byte[] buf = new byte[1024];
97                                  // Read some amount of data from the stream.
98                                  actual = inputStream.read(buf, 0, Math.min(buf.length, contentLength -
                                            downloadStream.size()));
```

This class downloads the data in chunks. Each time it checks to see how much data is left to download so that it does not download more than the content-length, this prevents downloading garbage.

```
99                                  if (actual == -1) {
100                                         // If end of stream was reached.
101                                         if (contentLengthSpecified && downloadStream.size() !=
                                                                                    contentLength) {
102                                                 // Content-length was specified but not reached.
103                                                 System.err.println("HttpRequest>  Content-Length
                                                                    given, but not matched to stream length.");
104                                                 data = null;
105                                         } else {
106                                                 // Content-length not given, end here.
107                                                 data = downloadStream.toByteArray();
108                                         }
109                                         break;
```

In each iteration, the class checks if there was no more data to read. If this is the case, it verifies the integrity of the data using the content-length, if given, and saves the download stream to the byte array for sending back to the listener.

```
110                                 }
111                                 downloadStream.write(buf, 0, actual);
112                                 if (contentLengthSpecified && downloadStream.size() ==
                                                                    contentLength) {
113                                         // Content-length reached, end here.
114                                         data = downloadStream.toByteArray();
115                                         break;
116                                 }
117                                 buf = null;
```

If it has not reached the end of the content, the class stores the data in its internal buffer and loops back.

```
118                         }
119                         downloadStream.close();
120                         downloadStream = null;
121                 } else {
122                         // Length inputStream either negative or 0
123                         System.err.println("HttpRequest>  Content-Length was negative or zero.");
124                         data = null;
125                 }
126         }
127 } catch (ConnectionNotFoundException cnfe) {
128         System.err.println("HttpRequest>  Could not establish connection to " + this.requestUrl);
129         System.err.println("ERROR: " + cnfe.getMessage());
130         data = null;
131 } catch (IOException ioe) {
132         System.err.println("HttpRequest>  Could not read data from " + this.requestUrl);
133         System.err.println("ERROR: " + ioe.getMessage());
134         data = null;
```

```
135                 } catch (ClassCastException e) {
136                         System.err.println("HttpRequest>  URL is not HTTP: " + this.requestUrl);
137                         System.err.println("ERROR: " + e.getMessage());
138                         data = null;
139                 } catch (IllegalArgumentException e) {
140                         System.err.println("HttpRequest> URL is not valid: " + this.requestUrl);
141                         System.err.println("ERROR: " + e.getMessage());
142                         data = null;
143                 } finally {
144                         if (inputStream != null) {
145                                 try {
146                                         inputStream.close();
147                                 } catch (Exception e) {
148                                         // ignore it
149                                         }
150                                 inputStream = null;
151                         }
152                         if (connection != null) {
153                                 try {
154                                         connection.close();
155                                 } catch (Exception e) {
156                                         // ignore it
157                                         }
158                                 connection = null;
159                         }
160                 }
161
162                 // Send the response to the listener.
163                 // Provide the data array and the request URL for verification.
164                 this.listener.httpRequestAction(data, this.requestUrl);
```

At this point there is either valid data in the byte array or it is equal to null. In either case it is sent to the listener, null indicating an error.

```
165         }
166     }
```