FORUM NOKIA

Introduction To The FileConnection API

Version 1.1; November 26, 2004

Java™

NOKIA

Contents

Change History

November 2, 2004	Version 1.0	Initial document release
November 26, 2004	Version 1.1	Document and example application package updated

1 Introduction

This document describes the FileConnection API [JSR-075]. It also includes a brief description of the example MIDlet that is included in this package and some Nokia-specific implementation details. The document assumes familiarity with Java™ programming and the basics of Mobile Information Device Profile (MIDP) programming, as described in the Forum Nokia document *MIDP 1.0: Introduction to MIDlet Programming* [MIDPPROG]. The FileConnection API is a restricted API and as such it is subject to security restrictions. Therefore, you should also be familiar with the MIDP 2.0 security framework concepts; the Forum Nokia document *MIDP 2.0: Tutorial On Signed MIDlets* [SIGNMID] provides insight into the security model and signing procedures.

The FileConnection API was specified in JSR-75: PDA Optional Packages for the J2ME™ Platform, which includes two Java™ 2 Platform, Micro Edition (J2ME™) optional packages oriented to support features typical of PDA-like devices. The optional packages give access to personal information management (PIM API) databases and local file systems (FileConnection API). These two packages are completely independent of each other, and thus devices may contain either one or both.

2 FileConnection API

2.1 Introduction

I/O operations in J2ME devices are handled using the Generic Connection Framework (GCF), by means of `Connection` interface implementations specific to each connection type. The different `Connection` extensions are built using a URL adequate to the different connection types such as http://, sockets://, and so on. In principle, the GCF is general enough to support connections to files but this has never been a mandatory part of J2ME or MIDP and in general it has been left out of most implementations. Even if such kind of connection would be built, there is a lack of support for file-specific operations such as rename or delete. In addition, access to local files has important implications with respect to security, privacy, and system stability that have to be taken into account.

The FileConnection API [JSR-075] fills the above-mentioned gap by giving access to file systems and support for file-oriented operations. The API assumes the existence of a file system in the device that can be located, for example, in removable memory cards, flash memory, or other types of persistent storage. This API is not meant to be a replacement for the Record Management System (RMS) but rather a complement to it allowing MIDlets to interact with native applications. For example, a MIDlet could access and manipulate images previously captured by a native application using a built-in digital camera. Those images are commonly stored in the device's memory and with the FileConnection API they are made accessible to CLDC/CDC[1] applications.

The API's minimum requirement is CLDC 1.0 so that basic J2ME devices, which may not even have a user interface, can implement it. However, this document and the example MIDlet will assume that the FileConnection API is implemented on a MIDP 2.0 device. The security implications of this API are also studied in the context of the MIDP 2.0 security framework [SIGNMIDP].

Since the FileConnection API is an optional extension, a system property has been added to indicate the API's presence. The `microedition.io.file.FileConnection.version` system property contains the implemented version of the API. Currently this property should have the value `1.0` to indicate the current status of the API or null if the API is not present. Another useful system property is `file.separator`, which contains the character used to separate directories, typically with the value "/".

The API is very simple containing just one class, two interfaces, and two exceptions. The most important part is the `FileConnection` interface, which extends the `Connection` interface and gives access to directories and individual files. Implementations of `FileConnection` are created using the `Connector.open()` method. The argument of the `open()` method is a URL with the format `file://<host>/<path>`, as defined in RFC 1738 [RFC 1738] and RFC 2396 [RFC 2396], where `host` is normally left empty and `path` starts with the root of the file system down to a particular file or directory. An example of a typical file URL in a Symbian device looks like the following:

```
file:///C:/Nokia/Images/Image(001).jpg
```

The roots of the file system are device-specific and they don't necessarily correspond to physical memory units since they are logically defined by the device's operating system. Furthermore, some Nokia devices support virtual roots that are basically links pointing to certain denoted directories. For instance, the location of captured images in a memory card could be located in the `file:///e:/Nokia/Images` path under the `e:` root, but additionally there is an `Images/` virtual root which points to the actual physical location. This makes it easier to find such locations and also eases the security permissions, given that a MIDlet may have access rights to the `Images/` root but not necessarily to the `e:/` root.

[1] Connected Limited Device Configuration / Connected Device Configuration

The `FileSystemRegistry` class provides the `listRoots()` utility method that returns an enumeration of the roots on the file system. This includes both logical and virtual roots. The API also takes into account that certain devices have the ability of having file systems added or removed at run time. The `FileSystemRegistry` class provides methods for registering `FileSystemListener` listeners that are called when the roots on the device are modified. It is recommended that every application register a `FileSystemListener` listener to be informed about these changes and act accordingly.

While the `FileConnection` interface extends `Connection` and objects are created using the GFC, there are some important differences with respect to other commonly used `Connection` implementations. One of the most important differences is that a call to `Connector.open()` can be successful even if the file doesn't currently exist. This is necessary when creating new files or directories. Nevertheless, it is illegal to open an `InputStream` to a non-existing file.

Another difference is that a `FileConnection` can remain open after the input or output streams are closed. Hence, it is important to call the `FileConnction.close()` method after the file has been accessed and thus ensure that it is available for other applications. In a related matter, modifications to a file using the `OutputStream` are not necessarily visible immediately by the file system. This depends on the actual implementation and the device's operating system. The `flush()` method ensures that any buffer is cleared and its contents written to the actual file.

An extra disparity with other `Connection` objects is that a `FileConnection` object can be reused using the `setFileConnection()` method. This method is mainly meant for doing directory traversal. The idea is that having built a `FileConnection` on a particular directory, the `list()` method can be called to obtain an enumeration of the children files and directories. The members of this enumeration can be passed to `setFileConnection()` as an argument and then the original `FileConnection` points to this particular children file or directory. Basically, the argument of `setFileConnection()` is a relative path to another children file or directory already existing or the ".." argument for the upper directory.

One general consideration that has been highlighted for all kinds of I/O operations is that they should be performed in a different thread than the GUI thread. The same recommendation applies when using the FileConnection API. This is highlighted when considered that due to the security framework, file-related operations could generate user prompts to authorize them. If an I/O operation is executed in the GUI thread and a user prompt is needed, the MIDlet may deadlock.

2.2 Security

When developing an application using the FileConnection API, it is important to take into account the security implications of the API. File operations are restricted with the aim of protecting the user's private data and the overall system security. File operations can be executed only if the needed permission has been acquired before; otherwise a `SecurityException` will be thrown. It is important to be aware of this and include a catch `SecurityExceptions` statement when appropriate.

MIDP 2.0 MIDlets are either untrusted or trusted [SIGNMID]. In the first case the device cannot assure the MIDlet's origin and integrity, and therefore calling restricted APIs is not allowed without explicit user permission. This means that whenever you need to access a file or a directory, a user prompt will appear and the user must explicitly authorize the operation.

In the case of trusted MIDlets, the device can determine their origin and integrity by means of X.509 certificates. These MIDlets may acquire permissions automatically depending on the security domain settings they were installed with. In addition, the MIDlet needs to include the requested file permissions in its Java Application Descriptor (JAD) file under the `MIDlet-Permission` property.

Two permissions have been defined in relation to the FileConnection API:

```
javax.microedition.io.Connector.file.read
javax.microedition.io.Connector.file.write
```

The first permission is necessary for opening files in READ mode and to obtain input streams to those files. It is also required when registering listeners with the `FileSystemRegistry` class. The second permission is required to open files in WRITE mode and for opening output streams to those files. In addition, operations such as `delete`, `mkdir`, and others need the write permission. If you open a file in READ_WRITE mode, you need both permissions. These permissions are contained in the `Read User Data Access` and `Write User Data Access` function groups.

Permissions are granted or denied depending on which security domain the MIDlet was installed in. Some domains may fully grant those permissions and others may allow them only with explicit user approval. The definition of what permissions are allowed for each domain is implementation-specific. Nevertheless, it is expected that for the *third-party* and *untrusted* domain the permissions mode will be as shown in Table 1:

Function group	Trusted third-party domain		Untrusted domain	
	Default setting	Allowed settings	Default setting	Allowed settings
Read User Data Access	Oneshot	Session, Blanket, Oneshot, No	Oneshot	Oneshot, No
Write User Data Access	Oneshot	Session, Blanket, Oneshot, No	No	Oneshot, No

Table 1: Allowed and default permission modes

In practical terms, Table 1 tells that an untrusted MIDlet will pop a user prompt every time a connection to a file or directory is created. Furthermore, if the connection is open in READ_WRITE mode, there will be two prompts, one for both permissions. In the case of trusted third-party MIDlets the situation is the same but the user has the option of manually changing this setting to *session* and therefore be asked only once while running the MIDlet. It is also important to notice that the permissions are given in a file-to-file basis. This means that the user may be prompted for each file or directory that is being accessed. This situation is particularly noteworthy for MIDlets such as the one in this example, which traverses the file system and thus gets multiple user prompts. This situation makes a strong point for why MIDlets should be signed when using restricted APIs.

In addition, there is an extra layer of restrictions with respect to file access. Depending on the security domain the MIDlet has been assigned during installation, it will have access to a subset of the file system. This is designed to protect the user data and prevent damage to the operating system. In particular, MIDlets located in the *trusted third-party* and *untrusted* domains have access only to a set of designated public directories including those for images, videos, public files, and a private directory assigned to each MIDlet for its own usage. This is one reason why using virtual roots is recommended since access to the `Images/` root may be allowed but doing traversal from `e:/` to `e:/Nokia/Images/` may not allowed, because `e:` could not be accessible by a MIDlet.

Several file-related operations check if the appropriate security permissions have been acquired, but the developer needs to take care in particular when the `Connector.open()` method is called. After a `FileConnection` has been created and the appropriate permission has been granted, it could be assumed that the permissions will hold for other operations requiring the same permission. For instance, once a `FileConnection` has been created for writing, invoking `delete` should also have been authorized. If the `FileConnection` has been created with a read permission and the

`delete()` method is called, the write permission will be needed and the user will be prompted if necessary.

The `setFileConnection()` method will also check for permissions to those files depending on which mode the original `FileConnection` was created. This is quite logical since `setFileConnection` changes the current connection to point to a different file or directory.

2.3 Nokia-Specific Directories

In many Nokia devices a number of directories are designated for specific tasks. For instance, a camera device stores captured photos in a specific "images" directory. To make it easier for developers to access those directories, Nokia devices implementing the FileConnection API contain additional system properties to locate them.

The existence of these properties cannot be taken for granted since not all devices have the need for them. The developer should take care of noting when the property is null and find an alternative.

Table 2 shows the system properties. The first column contains the property name that points to a particular directory in URL format. This URL can be passed directly to `Connection.open()`. The second column is an extra property containing the localized name for that directory. It is strongly recommended that instead of using a generic non-localized name for a directory, the property in the second column be used to make the MIDlet look consistent with the rest of the device UI.

Property	Localized property	Description
fileconn.dir.photos	fileconn.dir.photos.name	This points to the directory where photos captured with an integrated camera or other images are stored.
fileconn.dir.videos	fileconn.dir.videos.name	The same as the previous entry but with reference to videos. Downloaded videos are also stored here by default.
fileconn.dir.tones	fileconn.dir.tones.name	Ring tones and other similar audio files are stored in this directory.
fileconn.dir.memorycard	fileconn.dir.memorycard.name	Root directory of a memory card in case it is available.
fileconn.dir.private	fileconn.dir.private.name	Private work directory of MIDlet suite.

Table 2: Nokia's denoted directories

3 Image Viewer

The example developed for the purpose of this document using the FileConnection API is a simple Image Viewer. The MIDlet has a file browser to move around the file system and images can be selected and displayed on the screen. The purpose is to demonstrate how to access files and navigate the file system. The file browser includes some basic file management operations. Note that this example will not be signed, and consequently user prompts will often be displayed.

Figure 1 is a class diagram of the Image Viewer MIDlet.

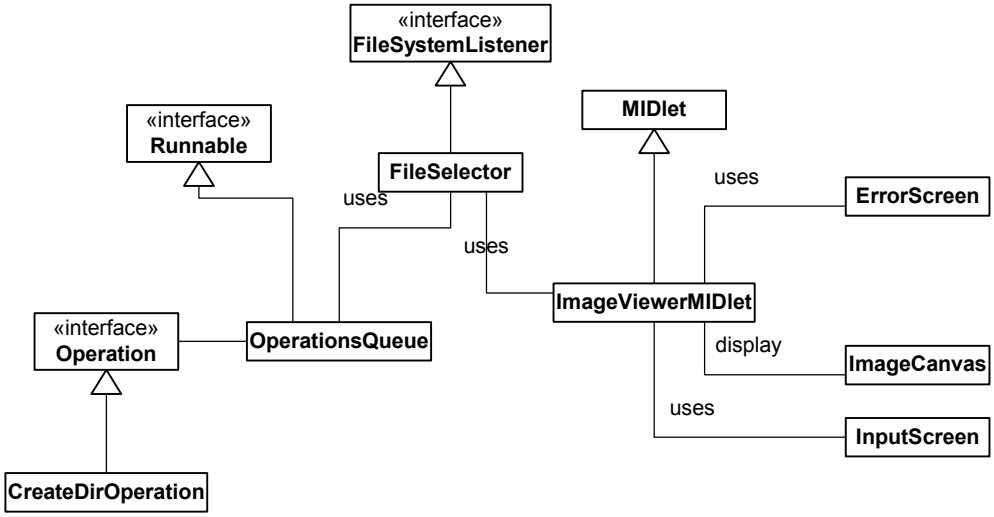

Figure 1: ImageViewerMIDlet class diagram

`ImageViewerMIDlet` is the starting point of the application. It controls the display and handles the transitions between the different screens.

The `FileSelector` class contains the bulk of the application. It contains the user interface and navigates the device's file system. It also contains support for file-oriented operations such as delete, rename, and directory creation. `FileSelector` will check whether the `fileconn.dir.photos` system property is available and will start navigating the file system on that directory if available. Otherwise, it will display a list of all the available roots.

The `ImageCanvas` class displays a selected image on the screen and upon detecting a key being pressed returns to the `FileSelector`. `InputScreen` is a simple form that is used to prompt the user to enter some text. This is used when creating a new directory and when renaming a file.

One important consideration when designing the Image Viewer MIDlet is that I/O operations are to be executed in a separate thread. The `OperationsQueue` class accomplishes this by executing commands serially in a separate thread.

The GUI consists of a simple file browser displaying the current directories and allowing navigation up and down the directory tree. Figure 2 shows two screenshots of the file browser.

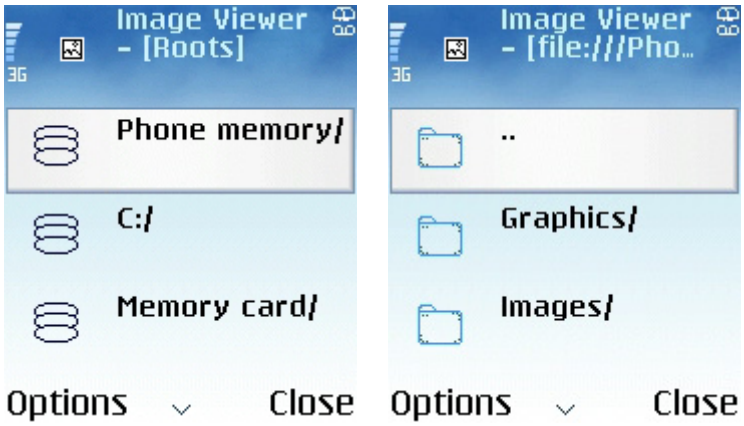

Figure 2: File Browser user interface in Nokia 6630

Once an image is selected, it is displayed in a simple canvas with a black background as shown Figure 3.

Figure 3: ImageViewer user interface

3.1 ImageViewerMIDlet

This is the MIDlet class implementation of the example. It has control over the Display and makes transitions between screens. On the first `startApp` call it will check that the FileConnection API is effectively present, otherwise it will inform the user. If it is available, it will create a `FileSelector` and assign it to the display.

The `requestInput()` and `input()` methods are used to show the `InputScreen` and to indicate the result of it respectively. The `displayImage()` method is invoked to show `ImageCanvas` displaying a particular image.

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

// Main class which inits the MIDlet and creates the screens
public class ImageViewerMIDlet
  extends MIDlet
{
  private final Image logo;
  private final ImageCanvas imageCanvas;
  private FileSelector fileSelector;
  private final InputScreen inputScreen;
  private int operationCode = -1;
```

```
public ImageViewerMIDlet()
{
  // init basic parameters
  logo = makeImage("/logo.png");
  ErrorScreen.init(logo, Display.getDisplay(this));
  imageCanvas = new ImageCanvas(this);
  fileSelector = new FileSelector(this);
  inputScreen = new InputScreen(this);
}


public void startApp()
{
  Displayable current = Display.getDisplay(this).getCurrent();

  if (current == null)
  {
    // Checks whether the API is available
    boolean isAPIAvailable = System.getProperty(
      "microedition.io.file.FileConnection.version") != null;
    // shows splash screen
    String text = getAppProperty("MIDlet-Name")
      + "\n"
      + getAppProperty("MIDlet-Vendor");
    if (!isAPIAvailable)
    {
      text += "\nFile Connection API is not available";
    }
    Alert splashScreen = new Alert(null,
      text,
      logo,
      AlertType.INFO);
    if (isAPIAvailable)
    {
      splashScreen.setTimeout(30);
      Display.getDisplay(this).setCurrent(splashScreen,
        fileSelector);
    }
    else
    {
      Display.getDisplay(this).setCurrent(splashScreen);
    }
  }
  else
  {
    Display.getDisplay(this).setCurrent(current);
  }
}


public void pauseApp()
{
}


public void destroyApp(boolean unconditional)
{
  // stop the commands queue thread
  fileSelector.stop();
  notifyDestroyed();
}


void fileSelectorExit()
{
  destroyApp(false);
}
```

```
  void requestInput(String text, String label, int operationCode)
  {
    inputScreen.setQuestion(text, label);
    this.operationCode = operationCode;
    Display.getDisplay(this).setCurrent(inputScreen);
  }

  void cancelInput()
  {
    Display.getDisplay(this).setCurrent(fileSelector);
  }


  void input(String input)
  {
    fileSelector.inputReceived(input, operationCode);
    Display.getDisplay(this).setCurrent(fileSelector);
  }


  void displayImage(String imageName)
  {
    imageCanvas.displayImage(imageName);
    Display.getDisplay(this).setCurrent(imageCanvas);
  }


  void displayFileBrowser()
  {
    Display.getDisplay(this).setCurrent(fileSelector);
  }


  void showError(Exception e)
  {
    ErrorScreen.showError(e.getMessage(), fileSelector);
  }


  void showMsg(String text)
  {
    Alert infoScreen = new Alert("Image Viewer",
      text,
      logo,
      AlertType.INFO);
    infoScreen.setTimeout(3000);
    Display.getDisplay(this).setCurrent(infoScreen, fileSelector);
  }


  // loads a given image by name
  static Image makeImage(String filename)
  {
    Image image = null;

    try
    {
      image = Image.createImage(filename);
    }
    catch (Exception e)
    {
      // use a null image instead
    }
    return image;
  }

}
```

3.2 FileSelector

This is the main class of the application. It consists of the file browser user interface, and it additionally invokes all the file operations using the FileConnection API. `FileSelector` includes a list that is filled by the contents of the current directory being explored. The content is presented using icons to denote directories and files. When starting, the class checks whether the Nokia's images directory is available and starts navigating from there. Otherwise it will present a list of all available roots. The current directory is stored in the `currentRoot` field, and a null value indicates pointing to the roots set. On opening a directory, its contents are searched for other directories, and png and jpg files that are displayed as the directory's content.

The class also registers itself in the `FileSystemRegistry` to listen for new file systems being added or removed, and in that case it will restart itself.

The file selector contains an inner class implementing the `Operation` interface. In most cases this class will just invoke private methods of `FileSelector`. Methods of interest are `createDir()`, which creates a new directory under the current directory, `deleteCurrent()`, which deletes the currently selected file or directory, `renameCurrent()`, which changes the name of the currently selected file, and `openSelected()`, which updates the list of files displayed. `displayCurrentRoot()` takes care of reading the contents of the directory and displaying them on the screen, whereas `displayAllRoots()` will show the list of all roots available.

File-related operations are enclosed in try/catch blocks to detect both `IOExceptions` and `SecurityExceptions`.

```
import java.io.*;
import java.util.*;
import javax.microedition.io.*;
import javax.microedition.io.file.*;
import javax.microedition.lcdui.*;

// Simple file selector class.
// It navigates the file system and shows images currently available
class FileSelector
  extends List
  implements CommandListener, FileSystemListener
{
  private final static Image ROOT_IMAGE =
    ImageViewerMIDlet.makeImage("/root.png");
  private final static Image FOLDER_IMAGE =
    ImageViewerMIDlet.makeImage("/folder.png");
  private final static Image FILE_IMAGE =
    ImageViewerMIDlet.makeImage("/file.png");
  private final OperationsQueue queue = new OperationsQueue();

  private final static String FILE_SEPARATOR =
    (System.getProperty("file.separator")!=null)?
      System.getProperty("file.separator"):
      "/";
  private final static String UPPER_DIR = "..";

  private final ImageViewerMIDlet midlet;
  private final Command openCommand =
    new Command("Open", Command.ITEM, 1);
  private final Command createDirCommand =
    new Command("Create new directory", Command.ITEM, 2);
  private final Command deleteCommand =
    new Command("Delete", Command.ITEM, 3);
  private final Command renameCommand =
    new Command("Rename", Command.ITEM, 4);
  private final Command exitCommand =
    new Command("Exit", Command.EXIT, 1);
```

```
private final static int RENAME_OP = 0;
private final static int MKDIR_OP = 1;
private final static int INIT_OP = 2;
private final static int OPEN_OP = 3;
private final static int DELETE_OP = 4;

private Vector rootsList = new Vector();
// Stores the current root, if null we are showing all the roots
private FileConnection currentRoot = null;
// Stores a suggested title in case it is available
private String suggestedTitle = null;


FileSelector(ImageViewerMIDlet midlet)
{
  super("Image Viewer", List.IMPLICIT);
  this.midlet = midlet;
  addCommand(openCommand);
  addCommand(createDirCommand);
  addCommand(deleteCommand);
  addCommand(renameCommand);
  addCommand(exitCommand);
  setSelectCommand(openCommand);
  setCommandListener(this);
  queue.enqueueOperation(new ImageViewerOperations(INIT_OP));
  FileSystemRegistry.addFileSystemListener(FileSelector.this);
}


void stop()
{
  if (currentRoot != null)
  {
    try
    {
      currentRoot.close();
    }
    catch (IOException e)
    {
    }
  }
  queue.abort();
  FileSystemRegistry.removeFileSystemListener(this);
}


void inputReceived(String input, int code)
{
  switch (code)
  {
    case RENAME_OP:
      queue.enqueueOperation(new ImageViewerOperations(
        input,
        RENAME_OP));
      break;
    case MKDIR_OP:
      queue.enqueueOperation(new ImageViewerOperations(
        input,
        MKDIR_OP));
      break;
  }
}


public void commandAction(Command c, Displayable d)
{
  if (c == openCommand)
  {
    queue.enqueueOperation(new ImageViewerOperations(OPEN_OP));
```

```
    }
    else if (c == renameCommand)
    {
      queue.enqueueOperation(new ImageViewerOperations(RENAME_OP));
    }
    else if (c == deleteCommand)
    {
      queue.enqueueOperation(new ImageViewerOperations(DELETE_OP));
    }
    else if (c == createDirCommand)
    {
      queue.enqueueOperation(new ImageViewerOperations(MKDIR_OP));
    }
    else if (c == exitCommand)
    {
      midlet.fileSelectorExit();
    }
  }


  // Listen for changes in the roots
  public void rootChanged(int state, String rootName)
  {
    queue.enqueueOperation(new ImageViewerOperations(INIT_OP));
  }


  private void displayAllRoots()
  {
    setTitle("Image Viewer - [Roots]");
    deleteAll();
    Enumeration roots = rootsList.elements();
    while (roots.hasMoreElements())
    {
      String root = (String) roots.nextElement();
      append(root.substring(1), ROOT_IMAGE);
    }
    currentRoot = null;
  }


  private void createNewDir()
  {
    if (currentRoot == null)
    {
      midlet.showMsg("Is not possible to create a new root");
    }
    else
    {
      midlet.requestInput("New dir name", "", MKDIR_OP);
    }
  }


  private void createNewDir(String newDirURL)
  {
    if (currentRoot != null)
    {
      try
      {
        FileConnection newDir =
          (FileConnection) Connector.open(
            currentRoot.getURL() + newDirURL,
            Connector.WRITE);
        newDir.mkdir();
        newDir.close();
      }
      catch (IOException e)
      {
```

```
        midlet.showError(e);
      }
      catch (SecurityException e)
      {
        midlet.showMsg(e.getMessage());
      }
      displayCurrentRoot();
    }
  }


  private void loadRoots()
  {
    if (!rootsList.isEmpty())
    {
      rootsList.removeAllElements();
    }
    try
    {
      Enumeration roots = FileSystemRegistry.listRoots();
      while (roots.hasMoreElements())
      {
        rootsList.addElement(FILE_SEPARATOR +
          (String) roots.nextElement());
      }
    }
    catch (SecurityException e)
    {
      midlet.showMsg(e.getMessage());
    }
  }


  private void deleteCurrent()
  {
    if (currentRoot == null)
    {
      midlet.showMsg("Is not possible to delete a root");
    }
    else
    {
      int selectedIndex = getSelectedIndex();
      if (selectedIndex >= 0)
      {
        String selectedFile = getString(selectedIndex);
        if (selectedFile.equals(UPPER_DIR))
        {
          midlet.showMsg("Is not possible to delete an upper dir");
        }
        else
        {
          try
          {
            FileConnection fileToDelete =
              (FileConnection) Connector.open(
                currentRoot.getURL() + selectedFile,
                Connector.WRITE);
            if (fileToDelete.exists())
            {
              fileToDelete.delete();
            }
            else
            {
              midlet.showMsg("File "
                + fileToDelete.getName() + " does not exists");
            }
            fileToDelete.close();
          }
          catch (IOException e)
```

```
            {
              midlet.showError(e);
            }
            catch (SecurityException e)
            {
              midlet.showError(e);
            }
            displayCurrentRoot();
          }
        }
      }
    }


    private void renameCurrent()
    {
      if (currentRoot == null)
      {
        midlet.showMsg("Is not possible to rename a root");
      }
      else
      {
        int selectedIndex = getSelectedIndex();
        if (selectedIndex >= 0)
        {
          String selectedFile = getString(selectedIndex);
          if (selectedFile.equals(UPPER_DIR))
          {
            midlet.showMsg("Is not possible to rename the upper dir");
          }
          else
          {
            midlet.requestInput("New name", selectedFile, RENAME_OP);
          }
        }
      }
    }


    private void renameCurrent(String newName)
    {
      if (currentRoot == null)
      {
        midlet.showMsg("Is not possible to rename a root");
      }
      else
      {
        int selectedIndex = getSelectedIndex();
        if (selectedIndex >= 0)
        {
          String selectedFile = getString(selectedIndex);
          if (selectedFile.equals(UPPER_DIR))
          {
            midlet.showMsg("Is not possible to rename the upper dir");
          }
          else
          {
            try
            {
              FileConnection fileToRename =
                (FileConnection) Connector.open(
                  currentRoot.getURL() + selectedFile,
                  Connector.WRITE);
              if (fileToRename.exists())
              {
                fileToRename.rename(newName);
              }
              else
              {
```

```
                midlet.showMsg("File "
                  + fileToRename.getName() + " does not exists");
              }
              fileToRename.close();
            }
            catch (IOException e)
            {
              midlet.showError(e);
            }
            catch (SecurityException e)
            {
              midlet.showError(e);
            }
            displayCurrentRoot();
          }
        }
      }
    }


    private void openSelected()
    {

      int selectedIndex = getSelectedIndex();
      if (selectedIndex >= 0)
      {
        String selectedFile = getString(selectedIndex);
        if (selectedFile.endsWith(FILE_SEPARATOR))
        {
          try
          {
            if (currentRoot == null)
            {
              currentRoot = (FileConnection) Connector.open(
                "file:///" + selectedFile, Connector.READ);
            }
            else
            {
              currentRoot.setFileConnection(selectedFile);
            }
            displayCurrentRoot();
          }
          catch (IOException e)
          {
            midlet.showError(e);
          }
          catch (SecurityException e)
          {
            midlet.showError(e);
          }
        }
        else if (selectedFile.equals(UPPER_DIR))
        {
          if(rootsList.contains(currentRoot.getPath()
              +currentRoot.getName()))
          {
            displayAllRoots();
          }
          else
          {
            try
            {
              currentRoot.setFileConnection(UPPER_DIR);
              displayCurrentRoot();
            }
            catch (IOException e)
            {
              midlet.showError(e);
            }
```

```
            catch (SecurityException e)
            {
              midlet.showMsg(e.getMessage());
            }
          }
        }
        else
        {
          String url = currentRoot.getURL() + selectedFile;
          midlet.displayImage(url);
        }
      }
    }


    private void displayCurrentRoot()
    {
      try
      {
        setTitle("Image Viewer - ["
          + ((suggestedTitle!=null)?suggestedTitle:currentRoot.getURL())
          + "]");
        // open the root
        deleteAll();
        append(UPPER_DIR, FOLDER_IMAGE);
        // list all dirs
        Enumeration listOfDirs = currentRoot.list("*", false);
        while (listOfDirs.hasMoreElements())
        {
          String currentDir = (String) listOfDirs.nextElement();
          if (currentDir.endsWith(FILE_SEPARATOR))
          {
            append(currentDir, FOLDER_IMAGE);
          }
        }
        // list all png files and don't show hidden files
        Enumeration listOfFiles = currentRoot.list("*.png", false);
        while (listOfFiles.hasMoreElements())
        {
          String currentFile = (String) listOfFiles.nextElement();
          if (currentFile.endsWith(FILE_SEPARATOR))
          {
            append(currentFile, FOLDER_IMAGE);
          }
          else
          {
            append(currentFile, FILE_IMAGE);
          }
        }
        // also list the jpg files
        listOfFiles = currentRoot.list("*.jpg", false);
        while (listOfFiles.hasMoreElements())
        {
          String currentFile = (String) listOfFiles.nextElement();
          if (currentFile.endsWith(FILE_SEPARATOR))
          {
            append(currentFile, FOLDER_IMAGE);
          }
          else
          {
            append(currentFile, FILE_IMAGE);
          }
        }
      }
      catch (IOException e)
      {
        midlet.showError(e);
      }
      catch (SecurityException e)
```

```
    {
      midlet.showError(e);
    }
  }


  private class ImageViewerOperations implements Operation
  {
    private final String parameter;
    private final int operationCode;


    ImageViewerOperations(int operationCode)
    {
      this.parameter = null;
      this.operationCode = operationCode;
    }


    ImageViewerOperations(String parameter, int operationCode)
    {
      this.parameter = parameter;
      this.operationCode = operationCode;
    }


    public void execute()
    {
      switch (operationCode)
      {
        case INIT_OP:
          String initDir = System.getProperty("fileconn.dir.photos");
          suggestedTitle =
            System.getProperty("fileconn.dir.photos.name");
          loadRoots();
          if (initDir != null)
          {
            try
            {
              currentRoot =
                (FileConnection) Connector.open(
                  initDir,
                  Connector.READ);
              displayCurrentRoot();
            }
            catch (IOException e)
            {
              midlet.showError(e);
              displayAllRoots();
            }
            catch (SecurityException e)
            {
              midlet.showError(e);
            }
          }
          else
          {
            displayAllRoots();
          }
          break;
        case OPEN_OP:
          openSelected();
          break;
        case DELETE_OP:
          deleteCurrent();
          break;
        case RENAME_OP:
          if (parameter != null)
          {
```

```
            renameCurrent(parameter);
          }
          else
          {
            renameCurrent();
          }
          break;
        case MKDIR_OP:
          if (parameter != null)
          {
            createNewDir(parameter);
          }
          else
          {
            createNewDir();
          }
      }
    }
  }
}
```

3.3 OperationsQueue and Operation

The operations queue is a simple utility class that runs in a separate thread executing operations serially. The `OperationsQueue` class takes objects implementing the `Operation` interface, making it independent of the kind of operations executed. `Operation` implementers are expected to handle their exceptions locally since `OperationsQueue` will discard any Exceptions thrown.

```
// Defines the interface for a single operation executed by
// the commands queue
interface Operation
{
  // Implement here the operation to be executed
  void execute();
}


import java.util.*;

// Simple Operations Queue
// It runs in an independent thread and executes Operations serially
class OperationsQueue implements Runnable
{

  private volatile boolean running = true;
  private final Vector operations = new Vector();


  OperationsQueue()
  {
    // Notice that all operations will be done in another
    // thread to avoid deadlocks with GUI thread
    new Thread(this).start();
  }


  void enqueueOperation(Operation nextOperation)
  {
    operations.addElement(nextOperation);
    synchronized (this)
    {
      notify();
    }
  }


  // stop the thread
```

```
  void abort()
  {
    running = false;
    synchronized (this)
    {
      notify();
    }
  }


  public void run()
  {
    while (running)
    {
      while (operations.size() > 0)
      {
        try
        {
          // execute the first operation on the queue
          ((Operation) operations.firstElement()).execute();
        }
        catch (Exception e)
        {
          // Nothing to do. It is expected that each operation handles
          // its own exception locally but this block is to ensure
          // that the queue continues to operate
        }
        operations.removeElementAt(0);
      }
      synchronized (this)
      {
        try
        {
          wait();
        }
        catch (InterruptedException e)
        {
          // it doesn't matter
        }
      }
    }
  }

}
```

3.4 ImageCanvas

This class is used to display an image on the screen. The background has previously been set to black and the image is displayed in the center. The image is drawn with its original dimensions and no attempt is made to scale it if it is bigger than the screen. `ImageCanvas` listens for any key presses and returns the control to the MIDlet.

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

// This class displays a selected image centered on the screen
class ImageCanvas
  extends Canvas
{
  private final ImageViewerMIDlet midlet;
  private static final int CHUNK_SIZE = 1024;
  private Image currentImage = null;


  ImageCanvas(ImageViewerMIDlet midlet)
```

```
{
  this.midlet = midlet;
}


public void displayImage(String imgName)
{
  try
  {
    FileConnection fileConn =
      (FileConnection)Connector.open(imgName, Connector.READ);
    // load the image data in memory
    // Read data in CHUNK_SIZE chunks
    InputStream fis = fileConn.openInputStream();
    long overallSize = fileConn.fileSize();

    int length = 0;
    byte[] imageData = new byte[0];
    while (length < overallSize)
    {
      byte[] data = new byte[CHUNK_SIZE];
      int readAmount = fis.read(data, 0, CHUNK_SIZE);
      byte[] newImageData = new byte[imageData.length + CHUNK_SIZE];
      System.arraycopy(imageData, 0, newImageData, 0, length);
      System.arraycopy(data, 0, newImageData, length, readAmount);
      imageData = newImageData;
      length += readAmount;
    }

    fis.close();
    fileConn.close();
    if (length > 0)
    {
      currentImage = Image.createImage(imageData, 0, length);
    }
    repaint();
  }
  catch (IOException e)
  {
    midlet.showError(e);
  }
  catch (SecurityException e)
  {
    midlet.showError(e);
  }
  catch (IllegalArgumentException e)
  {
    // thrown in case the file format is not understood
    midlet.showError(e);
  }

}


protected void paint(Graphics g)
{
  int w = getWidth();
  int h = getHeight();

  // Set background color to black
  g.setColor(0x00000000);
  g.fillRect(0, 0, w, h);

  if (currentImage != null)
  {
    g.drawImage(currentImage,
      w / 2,
      h / 2,
      Graphics.HCENTER | Graphics.VCENTER);
```

```
      }
      else
      {
        // If no image is available, display a message
        g.setColor(0x00FFFFFF);
        g.drawString("No image",
          w / 2,
          h / 2,
          Graphics.HCENTER | Graphics.BASELINE);
      }
    }


    protected void keyReleased(int keyCode)
    {
      // Exit with any key
      midlet.displayFileBrowser();
    }
}
```

3.5 InputScreen

`InputScreen` is a simple screen used to enter text. It is used in this application to enter text for creating new directories and renaming files. The screen sets an input text field and returns the results to the MIDlet.

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

// This class displays an input field on the screen
// and returns the value entered to the MIDlet
class InputScreen
  extends Form
  implements CommandListener
{
  private final ImageViewerMIDlet midlet;
  private final TextField inputField =
        new TextField("Input", "", 32, TextField.ANY);
  private final Command okCommand =
        new Command("OK", Command.OK, 1);
  private final Command cancelCommand =
        new Command("Cancel", Command.OK, 1);


  InputScreen(ImageViewerMIDlet midlet)
  {
    super("Input");
    this.midlet = midlet;
    append(inputField);
    addCommand(okCommand);
    addCommand(cancelCommand);
    setCommandListener(this);
  }


  public void setQuestion(String question, String text)
  {
    inputField.setLabel(question);
    inputField.setString(text);
  }


  public String getInputText()
  {
    return inputField.getString();
  }


  public void commandAction(Command command, Displayable d)
```

```
    {
      if (command == okCommand)
      {
        midlet.input(inputField.getString());
      }
      else if (command == cancelCommand)
      {
        midlet.cancelInput();
      }
    }

}
```

3.6 ErrorScreen

`ErrorScreen` is a simple class used for reporting errors to the end user.

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

class ErrorScreen
  extends Alert
{
  private static Image image;
  private static Display display;
  private static ErrorScreen instance = null;


  private ErrorScreen()
  {
    super("Error");
    setType(AlertType.ERROR);
    setTimeout(5000);
    setImage(image);
  }


  static void init(Image img, Display disp)
  {
    image = img;
    display = disp;
  }


  static void showError(String message, Displayable next)
  {
    if (instance == null)
    {
      instance = new ErrorScreen();
    }
    instance.setTitle("Error");
    instance.setString(message);
    display.setCurrent(instance, next);
  }

}
```

4 Terms and Abbreviations

Term or abbreviation	Meaning
CDC	Connected Device Configuration
CLDC	Connected Limited Device Configuration
GCF	Generic Connection Framework
J2ME™	Java™ 2 Platform, Micro Edition
MIDP	Mobile Information Device Profile
PIM API	Personal Information Management API
RMS	Record Management System

5 References

[JSR-075] *JSR 75: PDA Optional Packages for the J2ME™ Platform*, Java Community Process, 2004,
http://jcp.org/aboutJava/communityprocess/final/jsr075/index.html

[MIDPPROG] *MIDP 1.0: Introduction to MIDlet Programming*, Forum Nokia, 2004,
http://www.forum.nokia.com | Technologies | Java | Code and Examples

[SIGNMID] *MIDP 2.0: Tutorial On Signed MIDlets*, Forum Nokia, 2004,
http://www.forum.nokia.com/documents

[RFC 1738] *Uniform Resource Locators (URL)*, IETF, December 1994,
http://www.ietf.org/rfc/rfc1738.txt

[RFC 2396] Uniform Resource Identifiers (URI): Generic Syntax, IETF, August 1998,
http://www.ietf.org/rfc/rfc2396.txt

[MIDP 2.0] Mobile Information Device Profile 2.0, Java Community Process, 2002,
http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html

6 Evaluate This Document

In order to improve the quality of documentation, we kindly ask you to fill in the document survey.