FORUM NOKIA

MIDP: FileConnection API Developer's Guide

Version 2.0; October 31st, 2006

Java™

Change history

October 31st, 2006	Version 2.0	This document is based on MIDP: Introduction to FileConnection API, version 1.0, earlier published on Forum Nokia at http://www.forum.nokia.com/. The structure of the document has been harmonized and other minor updates have been made. In addition, some code comments have been added.

Disclaimer

The information in this document is provided "as is," with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

1 About this document

This document describes the FileConnection API (FC API), specified in JSR-75: PDA Optional Packages for the J2ME™ Platform, which includes two Java™ Platform, Micro Edition (Java™ ME) 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.

The document provides a step-by-step guide to help you to build a MIDlet for creating and managing a simple file connection.

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; Security, Signed MIDlets section of the Java ME Developer's Library at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm provides insight into the security model and signing procedures.

Intended audience

This document is intended for MIDP developers, as well as Java™ EE and Java SE developers wishing to develop mobile Java applications or services. The reader should be familiar with MIDP 2.0 (especially the LCDUI API) before attempting to understand this example.

Scope

This document assumes a good knowledge of application and service development and the Java programming language. Furthermore, it assumes familiarity with enterprise application development. However, previous knowledge of developing for the mobile environment is not necessary.

This document focuses on file connections and therefore explaining the Java technology is out of the scope of this documentation. For information on Java technology, see Java Technology web site at http://java.sun.com/ and Java Mobility Development Center at http://developers.sun.com/techtopics/mobility/. For additional information on MIDP tools and documentation, see Forum Nokia, Mobile Java section at http://www.forum.nokia.com/java.

2 Data Handling in Java™ ME

Java™ applications (MIDlets) may need to access native application data such as contact lists, or files stored on the device. Therefore, the current APIs that address data handling try to resolve the issues applications may face in the area, mainly handling persistent data and accessing databases on the device.

Persistent data

The `javax.microedition.rms` package of the MIDP 2.0 specification allows you to store persistent data such as saved games and high-score tables and later retrieve it. It does not enable writing or reading data outside the RMS. For more information on this package, see Mobile Information Device 2.0 (JSR-118) specification at http://www.jcp.org/en/jsr/detail?id=118.

Accessing databases

As a complement the `javax.microedition.rms` package, PDA Optional Packages for the J2ME™ Platform (JSR-75) specification at http://jcp.org/en/jsr/detail?id=75 includes two optional packages. The specification allows you to create, read, and write files and directories located on mobile devices and external memory cards.

Of these optional packages, PIM API gives access to personal information management databases such as contact lists, calendars, and to-do lists, whereas File API allows you to access local file systems (for example, removable storage media such as an external memory card) residing on the device through the Generic Connection Framework (GCF) at http://developers.sun.com/techtopics/mobility/midp/articles/genericframework/index.html. These two packages are completely independent of each other, and thus devices may contain either one or both.

3 FileConnection API overview

The FileConnection API (FC API), (JSR-75 specification at http://www.jcp.org/en/jsr/detail?id=75) was first introduced in S60 MIDP SDK 2nd Edition Feature Pack 2 and in Series 40 MIDP SDK 2nd Edition. It is supported in S60 and Series 40 with clarifications detailed in the FileConnection API (JSR-75): Implementation Notes.

The FC API gives access to file systems and supports 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 FC API they have been made accessible to CLDC/CDC applications.

The API's minimum requirement is CLDC 1.0 so that basic Java ME devices, which may not even have a user interface, can implement it. However, this document and the example MIDlet will assume that the FC 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 (Signed MIDlets section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™ ME Developer's Library).

For S60 specific implementation details of the API, see FC API Implementation notes at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm.

For devices that support the FC API, see Forum Nokia, Devices at http://www.forum.nokia.com/devices/.

4 Using FC API

The API contains one class, two interfaces, and two exceptions. The most important one of these 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 at http://www.ietf.org/rfc/rfc1738.txt and RFC 2396 at http://www.ietf.org/rfc/rfc2396.txt (RFC is the Request for Comments repository at http://www.ietf.org/rfc.html maintained by the IETF Secretariat), 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 S60 device looks as follows:

```
file:///C:/data/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 and virtual directories, which act as links pointing to certain denoted directories. For example, `c:/` and `e:/` are physical roots. The corresponding virtual roots are `Internal/` and `Memory card/`. Virtual directories map to the same directories that are defined in FC API System properties . This makes it easier to find locations and also eases the security permissions, given that a MIDlet may have access rights to the `Internal/Images` root but not necessarily to the `c:/` root. For more information on virtual file systems, see FC API Implementation notes at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm.

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 to have 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 for every application to register a `FileSystemListener` listener to be informed about these changes and act accordingly.

Since the FC 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 "/".

4.1 `FileConnection` interface considerations

While the `FileConnection` interface extends `Connection` and objects are created using the Generic Connection Framework (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 `FileConnection.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 child files and directories (files or directories existing within the original

directory). The members of this enumeration can be passed to `setFileConnection()` as an argument, and then the original `FileConnection` points to this newly specified child file or directory. Basically, the argument of `setFileConnection()` is a relative path to another child 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 FC 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.

4.2 Security considerations

When developing an application using the FC 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 required 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 (see Security, Signed MIDlets section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™ ME Developer's Library). With untrusted MIDlets, 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.

With trusted MIDlets, the device can determine their origin and integrity by means of X.509 certificates at http://www.ietf.org/rfc/rfc2459.txt. 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.

File permissions

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

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

  This permission is necessary for opening files in READ mode and to obtain input streams for those files. It is also required when registering listeners with the `FileSystemRegistry` class.

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

  This 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 the table below:

Table 4.1: Allowed and default permission modes

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	Oneshot	Oneshot, No

In practical terms, the table tells that an untrusted MIDlet will show a user prompt every time a connection to a file or directory is created. Furthermore, if the connection is opened 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 `c:/` to `c:/data/Images/` may not be allowed, because `c:` could not be accessible by a MIDlet.

Several file-related operations check if the appropriate security permissions have been acquired, but you should be careful 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.

4.3 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 to access those directories, Nokia devices implementing the FC 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. You should take care of noting when the property is null and find an alternative.

It is strongly recommended that instead of using a generic non-localized name for a directory, you should use properties returning the localized names of common directories to make the MIDlet look consistent with the rest of the device UI.

For the system properties that return the localized names of common directories, see FC API Implementation notes at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm.

5 Example: Creating a file access system

This tutorial shows you how to create a simple file browser using the FC API. The example application used in this tutorial is a simple Image Viewer. The purpose of the example is to demonstrate how to access files and navigate the file system. The MIDlet has a file browser to move around the file system and images can be selected and displayed on the screen. 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. For more information on signing, see the Signed MIDlets section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™ ME Developer's Library .

With the help of this tutorial, you can build the example from scratch and familiarize yourself with some common features related to the FC API. Alternatively, you can download the example files and run them immediately with your S60 MIDP SDK (3rd Edition or later). For more information on importing, building, and running Java projects, see Getting Started with Mobile Java section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™ ME Developer's Library .

Note: All files in this example application contain the Nokia copyright statement.

Note: The example is also provided on Forum Nokia at http://www.forum.nokia.com/info/sw.nokia.com/id/de0f933c-0bd3-4143-b62a-ab867a43409a/MIDP_FileConnection_API_Developers_Guide_v2_0_en.zip.html. Instead of creating the example from scratch as shown in this document, you can download the example files and run them immediately with your SDK.

5.1 Prerequisites

You will need a S60 or a Series 40 MIDP SDK that supports the packages used in this example.

5.2 Implementation of ImageViewer

The following UML diagram identifies the classes used in the Image Viewer example:

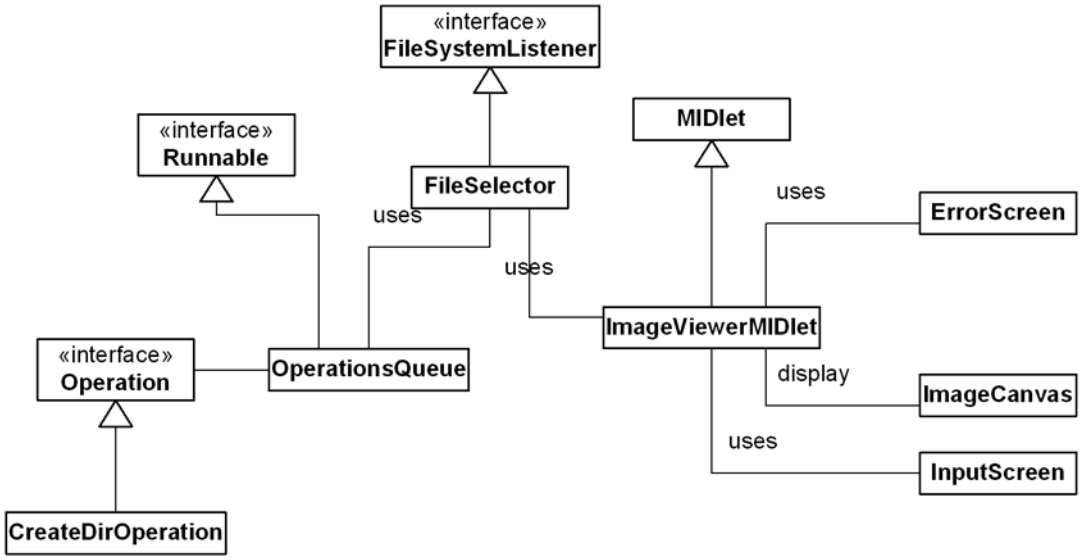

Figure 1: ImageViewerMIDlet class diagram

The ImageViewer example consists of the following classes:

- `ImageViewerMIDlet`

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

  The 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.
- `ImageCanvas`

  The class displays a selected image on the screen and upon detecting a key being pressed returns to the `FileSelector`.
- `OperationsQueue`

  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.
- `Operation`

  The class defines the `Operation` interface used by `OperationsQueue`.
- `InputScreen`

  The class `InputScreen` is a simple form used to prompt the user to enter some text. This is used when creating a new directory or when renaming a file.
- `ErrorScreen`

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

The GUI consists of a simple file browser displaying the current directories and allowing navigation up and down the directory tree. The figures below show two screenshots of the file browser.

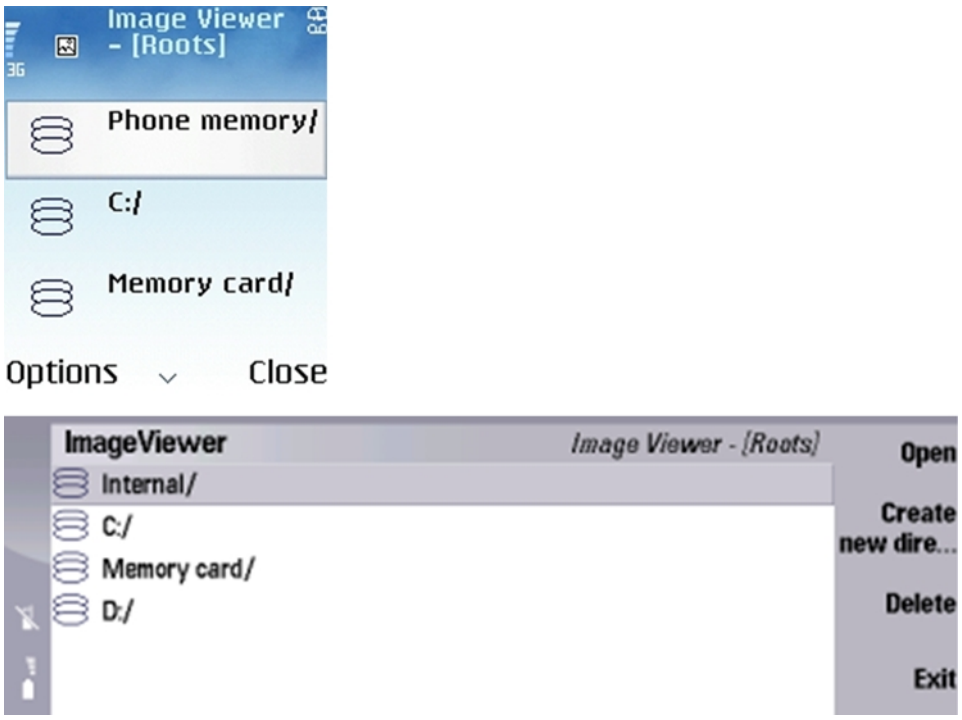

Figure 2: File Browser user interface in a Nokia device and a Nokia Communicator

Once an image is selected, it is displayed on a simple canvas with a black background as shown in the figure below.

Figure 3: ImageViewer user interface

5.3 Creating the project environment

The project environment is constructed using the instructions for building MIDlet in IDEs or using command line, that come with the Getting Started with Mobile Java section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™ ME Developer's Library.

5.4 Developing ImageViewer

This section provides a step-by step tutorial showing how to develop the example application.

5.4.1 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.

To create this class:

1 Create the `ImageViewerMIDlet` class file.

2 Import the required classes.

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

3 Implement the class.

```
// 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;
    }
  }
```

5.4.2 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 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.

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

To create this class:

1 Create the `FileSelector` class file.

2 Import the required classes.

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

3 Set the `FileSelector` class to extend `List` and to implement `CommandListener` and `FileSystemListener`. Create the methods used later to manage the file system.

When starting, the class checks whether the images directory is available and starts navigating from there. Otherwise it will present a list of all available roots.

```
// 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;
```

4 The current directory is stored in the `currentRoot` field, and a null value indicates pointing to the roots set. Here, the class also registers itself in the `FileSystemRegistry` in order to listen to new file systems being added or removed, and in that case it will restart itself. For more information on file system listeners and `FileSystemRegistry`, see the `FileSystemListener` interface and the `FileSystemRegistry` class.

```
  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));
  }
```

5 Use the method `displayAllRoots()` to display the list of all roots available.

```
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;
}
```

6 Use the method `createNewDir()`, created earlier, to create a new directory under the current directory. Note that `FileConnection` is used here to access the file system.

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.

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

7 Use the method `deleteCurrent()`, created earlier, to delete the currently selected file. Note that `FileConnection` is used here to access the file system.

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

8 Use the method `renameCurrent()`, created earlier, to change the name of the currently selected file. Note that `FileConnection` is used here to access the file system.

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

9 Use the method `openSelected()`, created earlier, to update the list of files displayed on screen. Note that `FileConnection` is used here to access the file system. The `setFileConnection()` method resets the `FileConnection` object to another file or directory, therefore allowing the reuse of the `FileConnection` object for directory traversal. This allows that the `FileConnection` instance object can remain open, referring now to the new file or directory.

```
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);
          }
        }
      }
```

10 Use the method `displayCurrentRoot()` to handle reading the contents of the directory and display the contents on the screen. When a directory is opened, its contents are searched for subdirectories and PNG or JPG files. The found files and subdirectories are then displayed as the directory's content.

```
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);
    }
  }
```

11 Set up a new class to implement the `Operation` interface. In most cases this new class is used to invoke the private methods of `FileSelector`.

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

5.4.3 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.

To create the `OperationsQueue` class:

1 Create the `OperationsQueue` class file.

2 Import the required classes and implement the interface.

```
// 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.*;
```

3 Create the operations queue and use the `run()` method to execute the operations on the queue.

```
// 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
        }
      }
    }
  }
}
```

To create the `Operation` interface:

1 Create the `Operation` class file.

2 Define the interface.

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

5.4.4 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.

To create this class:

1 Create the `ImageCanvas` class file.

2 Import the required classes.

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

3 Implement the class `ImageCanvas`.

```
// 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;
  }
```

4 Use the method `displayImage()` to display the image on screen. In order to access the file system and to load the image into memory and then to display the image on screen, also create an instance of the `FileConnection` interface, provided by the FileConnection API.

Call the `fileConn.close()` method to close the connection. Calling `fileConn.close()` after the file has been accessed is important to ensure that it is available for other applications. For more information on file connections and file connection security, see the `FileConnection` interface.

```
  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();
        }
      }
```

5.4.5 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.

To create this class:

1 Create the `InputScreen` class file.

2 Import the required classes.

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

3 Create the implementation of the input screen.

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

5.4.6 ErrorScreen

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

To create this class:

1. Create the `ErrorScreen` class file.
2. Import the required classes.

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

3. Create the implementation of the error screen.

```
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);
        }
    }
```

5.5 Building and running in an emulator

After you have created all the necessary files for the project, you can build the project and run the application in an emulator.

Note: The MIDlet used in this example was built using the Eclipse IDE. Please see the instructions for building MIDlet projects for Eclipse that come with the Getting Started with Mobile Java section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™ ME Developer's Library.

You can also construct them using command line or other IDEs of your choice.

Additionally, a `build.xml` file is included in the ZIP file if you want to build the example using Ant at http://ant.apache.org .

Note: The example is also provided on Forum Nokia at http://www.forum.nokia.com/info/sw.nokia.com/id/de0f933c-0bd3-4143-b62a-ab867a43409a/MIDP_FileConnection_API_Developers_Guide_v2_0_en.zip.html. Instead of creating the example from scratch as shown in this document, you can download the example files and run them immediately with your SDK.

5.6 Deploying to a device

Deploy test packages onto your mobile device, using a USB cable, Bluetooth connection, infrared, or other hardware compatible connection. Alternatively, deploy the application using OTA technology.

For instructions on how to do this, see the Getting Started with Mobile Java section at http://www.forum.nokia.com/main/resources/technologies/java/documentation/Java_ME_developers_library.htm of the Java™ ME Developer's Library.