JSR 75 - File Connection

Version 0.9, Draft

API GUIDE

COPYRIGHT

Trademarks and Service Marks

About This Document

This document describes about FileConnection optional package *javax.microedition.io.file* (JSR-75) and provides sample code snippets.

Scope

This document is intended for users who have knowledge of Java programming language. Focusing on Java and Java ME is out of scope of this document.

Document History:

Date	Version	Comment
17/06/09	0.9	Draft

References:

1. JSR 75 specification:
   http://jcp.org/en/jsr/detail?id=75

2. JSR 75 Article:
   http://developers.sun.com/mobility/apis/articles/fileconnection/

Abbreviations:

1.	Java ME	Java Micro Edition
2.	MIDP	Mobile Information Device Profile
3.	CLDC	Connection Limited Device Configuration
4.	API	Application Programming Interface
5.	JSR	Java Specification Request
6.	GCF	Generic Connection Framework
7.	FC	File Connection
8.	FCOP	File Connection Optional Package
9.	PIM	Personal Information Management

Table of Contents

Introduction

JSR 75 is an optional package that includes two packages. These are the light weighted APIs on devices that implement JSR 75, FileConnection (FC) *javax.microedition.io.file.*;* and Personal Information Management (PIM)*javax.microedition. pim.*;*
FileConnection package enables Java ME applications to create, read, and write files and directories located in mobile devices and external memory cards.
CLDC 1.0 is the minimum requirement for this API set.

Overview

JSR 75 allows developers to access two major areas. They are FileConnection and Personal Information Management.

- FileConnection

The FileConnection Optional Package (FCOP) API gives Java ME devices access to file systems residing on mobile devices, primarily access to removable storage media such as external memory cards and to do file operations.

- Personal Information Management

The PIM Optional Package (PIM) APIs give Java ME devices access to personal information management data native to mobile devices, such as address books, calendars, events and to-do lists.

API Description of FCOP:

FileConnection Optional Package API is defined in the package *javax.microedition.io.file* which includes the two interface and three classes.

Interface	Description
FileConnection	This interface is intended to access files or directories that are located on removable media and/or file systems on a device.
FileSystemListener	This class is used for receiving status notification when adding or removing a file system root.

Class	Description
FileSystemRegistry	The FileSystemRegistry is a central registry for file system listeners interested in adding and removing (or mounting and unmounting) of file systems on a device.
ConnectionClosedException	Exception thrown when a method of a file connection is invoked but cannot be completed because the connection is closed.
IllegalModeException	Exception thrown when a method is invoked that requires a particular security mode, such as READ or WRITE, but the connection opened is not in that mode.

Detecting FileConnection API's presence:

To check whether the handset supports FileConnection API,

```
System.getProperty("microedition.io.file.FileConnection.version ").
```

can be used. If it is supported, the FileConnection version is returned else null will be returned if it is not supported.

```
public class MainMidlet extends MIDlet {

public MainMidlet() {

 Form form = new Form("File Connection version");
 form.append(System.getProperty("microedition.io.file.FileConnection.version"));
 Display.getDisplay(this).setCurrent(form);
}
…
}
```

File Separator

In addition to the system property *microedition.io.file.FileConnection.version*, FileConnection implementations support system property *file.separator*.

The file.separator property returns a string representing the file separator character(s) for the underlying platform. For example, "\" would be returned on a Windows OS based platform, and "/" would be returned on a Unix system.

```
…
form.append(System.getProperty("microedition.io.file.FileConnection.version"));
form.append(System.getProperty("file.separator "));
…
```

Establishing Connection:

File Connection APIs use the Generic Connection Framework (GCF) for file-system connectivity. This application opens the file connection using *Connector.open (String protocol)* the string protocol must comprise a fully qualified absolute path like:

file://host/root/directory/.../name

host - The host element may be empty - and often will be, when the string refers to a file on the local host.
root – The root element may be empty and corresponds to a logical mount point for a particular storage unit.
directory – directory name
name – file name.

Following are some examples of root values and how to open them:

To open CFcard (Compact flash card):	*(FileConnection) Connector.open("file:///CFCard/");*
To open the SDcard (Secure Digital card):	*(FileConnection) Connector.open("file:///SDCard/");*
To open Memory Stick:	*(FileConnection) Connector.open("file:///MemoryStick/");*
To open the any device drive:	*(FileConnection) Connector.open("file:///C:/");*

Example to create a FileConnection:

```
…
private final String URL=file://localhost/root1/mytext.txt;
FileConnection fileConn=
   (FileConnection)Connector.open(URL,Connector.READ_WRITE);
/*do file/directory operations */
…
fileConn.close();
…
```

Actual file or directory that the URL is pointing to, does not exist while creating a FileConnection. Therefore opening a file/directory and creating a new file/directory is very similar. Remember to close the FileConnection object using *fileConn.close();*

FC, which is pointing to a non-existent file, will not be able to perform any file or directory specific operations.

Example to create file in the directory:

```
…

        public void createFile() {
                final String URL="file:///root1/SMI";
                FileConnection fileConn=null;
                try {
        fileConn=(FileConnection)Connector.open(URL,Connector.READ_WRITE);
                if(!fileConn.isDirectory())
                   fileConn.mkdir();
                   fileConn.close();
        fileConn=(FileConnection)Connector.open(URL+ ”temp.txt”,Connector.READ_WRITE);

                        if(!fileConn.exists())
                        {
                                fileConn.create();
                        }
                        fileConn.close();
                } catch (IOException e) {
                        e.printStackTrace();
                }

        }
…
```

FileConnection Operations:

You can perform several operations with FileConnection object once the file connection is successful. Some of them are:

- To discover whether a file or directory exists using *exists().*
- To discover whether a file or directory is hidden using *isHidden().*
- To create or delete a file or directory using *create(), mkdir(), or delete().*

- For a list of all the valid root values in a device, call the *listRoots*() method of FileSystemRegistry.
- To list the content of a directory that your FileConnection is pointing to use *list()*

```
…
Enumeration e = fileConn.list();
while (e.hasMoreElements()) {
form.append(((String)e.nextElement()));
}
…
```

- Call method *list(String filter, boolean includeHidden)* to get the filtered list of files and directory.

filter - String against which all files and directories are matched for retrieval. An asterisk ("*") can be used as a wildcard to represent 0 or more occurrences of any character.

- *includeHidden* - boolean indicating whether files marked as hidden should be included or not in the list of files and directories returned.

- To write to a file you need to get an *OutputStream* from the FileConnection object that points to the existing file:
  OutputStream os = fileConn.openOutputStream()
- To read from a file you need to get an *InputStream* from the FileConnection object that points to the existing file:
  InputStream os = fileConn.openInputStream()
- *fileConn. canRead()* and *fileConn.canWrite()* are used to check whether the file/directory is readable or writable. Both methods returns boolean, which indicates whether a file can be read / write.
- *fileConn.directorySize(boolean includeSubDirs)* is used to determine the size of all the files that are contained in a directory in a file system.
- *fileConn.fileSize()* is used to determine the size of a file. You can set your file to be hidden/readable/writeable by using *fileConn.setHidden(boolean hidden), fileConn.setReadable(boolean readable)*, *fileConn.setWritable(boolean writable)*
- *fileConn.lastModified()* is used to determine when was the file last modified.

JSR 75 – File Connection Example:

This code snippet demonstrates how to create, delete a file and folder using *javax.microedition.io.file.*;* package.

```
import java.io.IOException;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;
import javax.microedition.lcdui.Alert;
import javax.microedition.lcdui.AlertType;
import javax.microedition.lcdui.Choice;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.Displayable;
import java.util.Enumeration;
import javax.microedition.io.Connector;
import javax.microedition.io.file.FileConnection;
import javax.microedition.io.file.FileSystemRegistry;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.TextField;
import javax.microedition.lcdui.ChoiceGroup;
public class FileMidletDemo extends MIDlet implements CommandListener{

        private Display display = null;
        private final Command CMD_EXIT = new Command("Exit",Command.EXIT,1);
        private final Command CMD_VIEW = new
                                          Command("View",Command.SCREEN,2);
        private final Command CMD_NEW  = new
                                           Command("New",Command.SCREEN,3);
        private final Command CMD_OK   = new Command("Ok",Command.OK,4);
        private final Command CMD_DELETE =  new
                                          Command("Delete",Command.SCREEN,5);
        private final String SEPERATOR  = "/";
        private Image image_dir = null;
        private Image image_file = null;
        private final char SEP ='/';
        private final String INIT_DIR="..";
        private String currentDir;
        private TextField textField=null;
        private ChoiceGroup choiceGroup=null;
        public FileMidletDemo() {

                currentDir = SEPERATOR;
```

```
            initRes();
            display = Display.getDisplay(this);
            initFileSys();

    }

    protected void destroyApp(boolean arg0) throws MIDletStateChangeException {


    }

    protected void pauseApp() {


    }

    protected void startApp() throws MIDletStateChangeException {


    }
    private void createFile()
    {
            Form form = new Form("New File");
            textField = new TextField("Enter Name",null,256,TextField.ANY);
            choiceGroup = new ChoiceGroup("Enter the file
          Name",Choice.EXCLUSIVE, new String[]{"Regular File","Directory"},new
                                                    Image[]{image_file,image_dir});
            form.addCommand(CMD_EXIT);
            form.addCommand(CMD_OK);
            form.append(textField);
            form.append(choiceGroup);
            form.setCommandListener(this);
            display.setCurrent(form);
    }
    private void executeMentionFile(String newFile,boolean isDir)
    {
            try {
                    FileConnection fileConn
               = (FileConnection)Connector.open("file:///"+currentDir+newFile);

                    if(isDir)
                            fileConn.mkdir();
                    else
                            fileConn.create();

                    initFileSys();
```

```
		} catch (IOException e) {

			e.printStackTrace();
		}
	}
	private void showFile(String fileName)
	{

	}
	private void traversDir(String fileName)
	{

		if (currentDir.equals(SEPERATOR)) {

			if (fileName.equals(INIT_DIR))
				return;
			currentDir = fileName;
		} else if (fileName.equals(INIT_DIR)) {

			int i = currentDir.lastIndexOf(SEP, currentDir.length() - 2);

			if (i != -1) {
				currentDir = currentDir.substring(0, (i + 1));
			} else {
				currentDir = SEPERATOR;
			}
		} else {
			currentDir = currentDir + fileName;
		}

		initFileSys();

	}
	private void deleteFolder(String folder)
	{
		try {
			FileConnection fileConn
		=  (FileConnection)Connector.open("file://localhost/"+currentDir+folder);

			Enumeration e = fileConn.list("*",true);
			if(!e.hasMoreElements())
			{
				fileConn.delete();
				initFileSys();
			}
			else
			{
```

```
                    Alert alert = new Alert("Error!!!","Cann't delete empty
                                                folder",null,AlertType.ERROR);
                        alert.setTimeout(Alert.FOREVER);
                        alert.addCommand(CMD_EXIT);
                        alert.setCommandListener(this);
                        display.setCurrent(alert);
                    }

            } catch (IOException e) {
                    e.printStackTrace();
            }
    }
    private void deleteFile(String file)
    {

            try {
                    FileConnection fileConn
              = (FileConnection)Connector.open("file://localhost/"+currentDir+file);
                    fileConn.delete();
            } catch (IOException e) {
                    Alert alert =  new Alert("Error!",
       "Can not access/delete file " + file + " in directory " + currentDir +
       "\nException: " + e.getMessage(), null, AlertType.ERROR);
                    alert.setTimeout(Alert.FOREVER);
                    alert.addCommand(CMD_EXIT);
                    alert.setCommandListener(this);
                    display.setCurrent(alert);
            }

    }
```

```
    private void delete(String file)
    {
            if(!file.endsWith(INIT_DIR))
            {
                    if(file.charAt(file.length()-1)== SEP)
                    {
                            deleteFolder(file);
                    }
                    else
                    {
                            deleteFile(file);
                    }
            }
            initFileSys();

    }
```

```
private void initRes()
{
  try {
    image_dir = Image.createImage(getClass().getResourceAsStream("/dir.png"));
   image_file = Image.createImage(getClass().getResourceAsStream(/file.png"));
 } catch (IOException e) {
                e.printStackTrace();
        }

}

private void initFileSys() {
        List dirList = null;
        Enumeration e = null;
        FileConnection fileConn = null;

        if (SEPERATOR.equals(currentDir)) {
                e = FileSystemRegistry.listRoots();
                dirList = new List(currentDir, Choice.IMPLICIT);
        } else {
                try {
                        fileConn = (FileConnection) Connector.open("file://localhost/"
                                        + currentDir);
                        e = fileConn.list();
                        dirList = new List(currentDir, Choice.IMPLICIT);
                        dirList.append(INIT_DIR, image_dir);
                } catch (IOException e1) {

                        e1.printStackTrace();
                }
        }
        while (e.hasMoreElements()) {
                String element = (String) e.nextElement();
                if (element.charAt(element.length() - 1) == SEP)
                        dirList.append(element, image_dir);
                else
                        dirList.append(element, image_file);
        }
        if (fileConn != null)
                try {
                        fileConn.close();
                } catch (IOException e1) {
                        e1.printStackTrace();
                }
                dirList.addCommand(CMD_EXIT);
                dirList.addCommand(CMD_VIEW);
                dirList.addCommand(CMD_NEW);
```

```
                dirList.addCommand(CMD_DELETE);
                dirList.setCommandListener(this);
                display.setCurrent(dirList);
        }

        public void commandAction(Command cmd,Displayable dis)
        {
                if(cmd == CMD_EXIT)
                {
                        try {
                                destroyApp(true);
                                notifyDestroyed();
                        } catch (MIDletStateChangeException e) {
                                e.printStackTrace();
                        }
                }
                else if(cmd == CMD_VIEW)
                {

                        List dirList =(List)dis;
                        final String currentfile =
                        dirList.getString(dirList.getSelectedIndex());
                                new Thread(new Runnable(){
                                        public void run()
                                        {
                                                if(currentfile.charAt(currentfile.length()-
                                                                                1)==SEP
                                                ||currentfile.endsWith(INIT_DIR))
                                                {

                                                        traversDir(currentfile);

                                                }
                                                else
                                                {
                                                        showFile(currentfile);
                                                }
                                        }
                                }).start();
                }
                else if(cmd == CMD_NEW)
                {
                        createFile();
                }
                else if (cmd == CMD_OK)
                {
                        final String newFile = textField.getString();
```

```
                    if(newFile == null || newFile.equals(""))
                    {
                        Alert alert  = new Alert("Error!!!","File name is
                                empty....",null,AlertType.ERROR);
                        alert.setTimeout(Alert.FOREVER);
                        alert.addCommand(CMD_EXIT);
                        alert.setCommandListener(this);
                        display.setCurrent(alert);

                    }
                    else
                    {
                        new Thread(new Runnable(){
                            public void run()
                            {
            executeMentionFile(newFile,(choiceGroup.getSelectedIndex())!=0);
                            }
                        }).start();
                    }
                }
                else if(cmd == CMD_DELETE)
                {
                    List list = (List)dis;
                    final String file = list.getString(list.getSelectedIndex());
                    new Thread(new Runnable(){
                        public void run()
                        {
                            delete(file);
                        }
                    }).start();
                }
            }
}
```