

Java 2 Micro Edition Persistent Storage Management



F. Ricci
2010/2011

Content

- ❑ Record store
- ❑ Managing record stores
- ❑ Private and shared record stores
- ❑ Records
- ❑ Listening to record changes
- ❑ Query processing: `RecordEnumeration`,
`RecordFilter`, `RecordComparator`
- ❑ File connection package (optional package)
- ❑ Opening and closing files
- ❑ Reading and writing from and to a file
- ❑ Personal Information Management (PIM) package
(optional package)

Persistent Storage: MIDP Record Store

- ❑ In MIDP persistent storage is centered around the **record store**: a small database
- ❑ The minimum amount of persistent storage defined in the MIDP specification **is only 8kb!**
- ❑ Record stores are represented by instances of `javax.microedition.rms.RecordStore`
- ❑ The scope of a record store can be **limited to a single** MIDlet or **shared** between MIDlets
- ❑ Record stores are identified by a **name**
- ❑ Within a MIDlet suite the names of the record stores must be unique.

Managing Record Stores

- ❑ To **open** a record store you need to name it

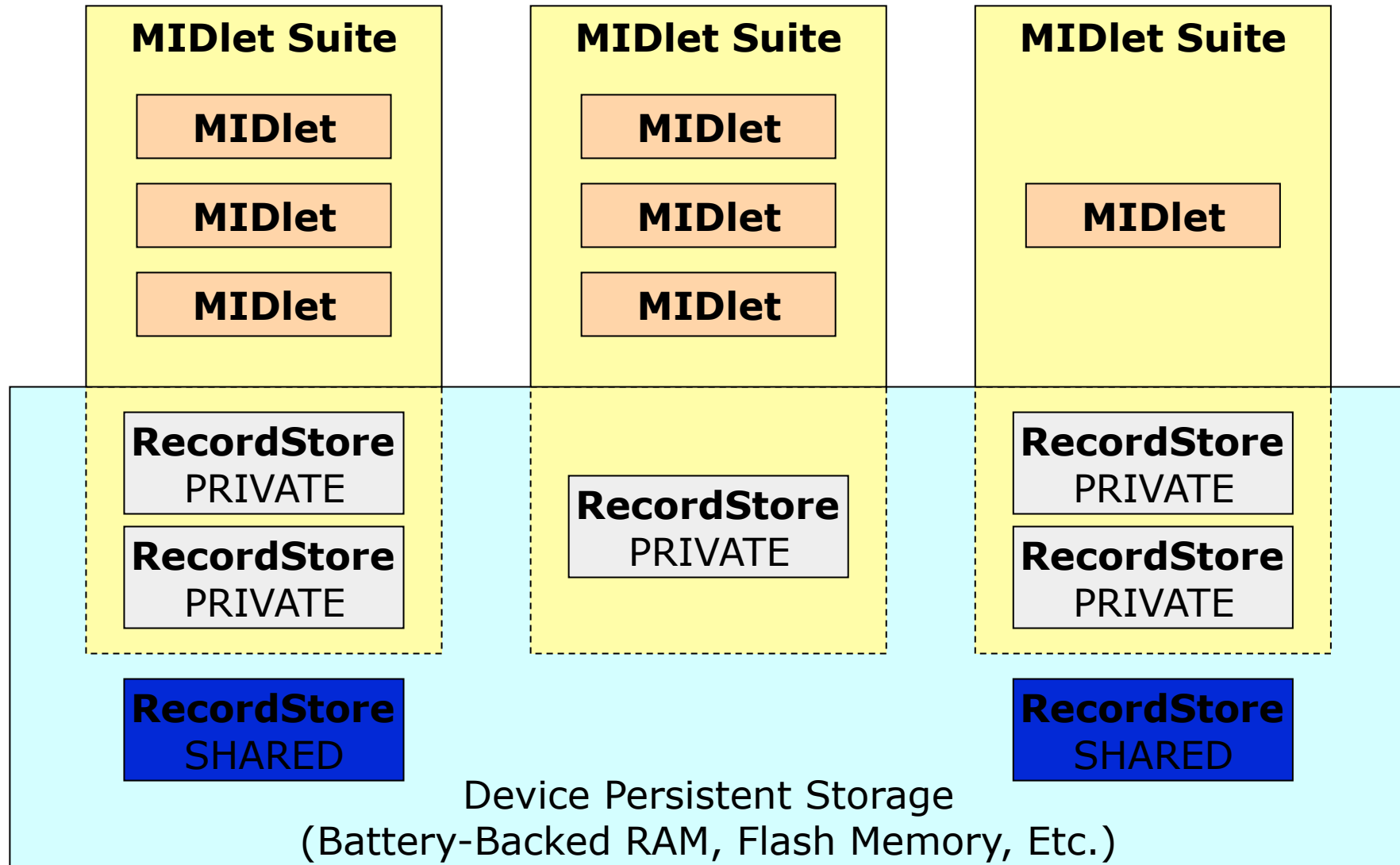
```
public static RecordStore openRecordStore (String  
    recordStoreName, boolean createIfNecessary) throws  
    RecordStoreException, RecordStoreFullException,  
    RecordStoreNotFoundException
```

- ❑ If the record store **does not exist**, the `createIfNecessary` parameter determines whether a new record store will be **created or not**
- ❑ The following (creates and) opens a record store named "Address"

```
RecordStore rs = RecordStore.openRecordStore  
    ("Address", true);
```

- ❑ Call `closeRecordStore()` to **close** an open record store
- ❑ To **find out all the record stores** available to the MIDlet, call the `listRecordStore()` method - it returns a `String[]` array containing a list of available record stores
- ❑ To **remove** a record store call the static method `deleteRecordStore()`.

Private and shared record stores



Sharing Record Stores

- Record stores have an **authorization mode**
 - The default mode is `AUTHMODE_PRIVATE`, that record store is accessible only inside a MIDlet suite that created the record store
 - Record store can be **shared** changing the authorization mode to `AUTHMODE_ANY`

- You can decide also if you want a record store to be **writable** or **read-only**

- Open (and possibly create) a record store that can be shared with other MIDlet suites:

```
public Static RecordStore openRecordStore (String  
recordStoreName, boolean createIfNecessary, int  
authmode, boolean writable)
```

- You can **change** the authorization mode and writable flag of an open record store using the following method

```
public void setMode(int authmode, boolean  
writable)
```

Sharing Record Store, Size

- ❑ To access an available **shared** record store use:

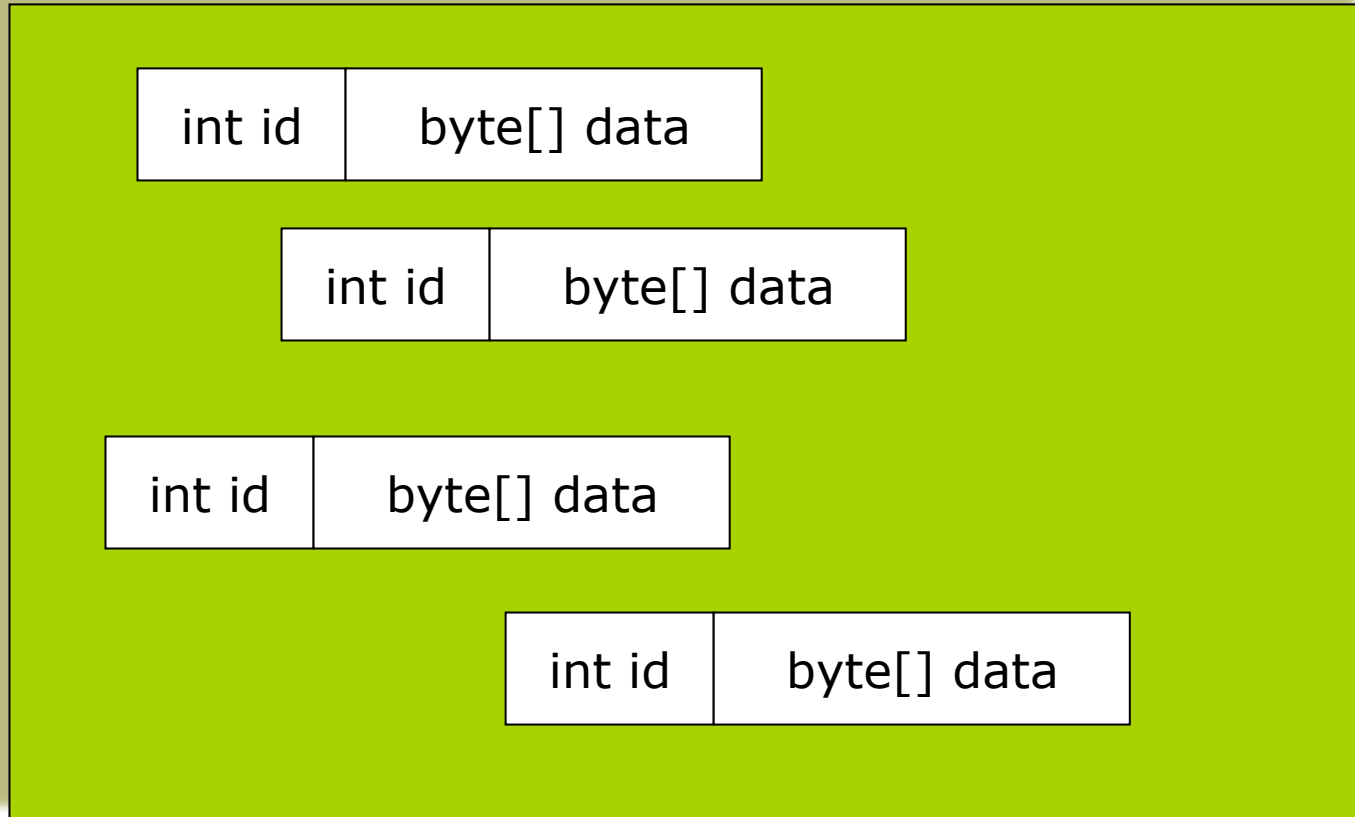
```
static RecordStore openRecordStore(String  
    recordStoreName, String vendorName, String  
    suiteName)
```

- ❑ You need to know the name of the MIDlet that created it and the vendor name
- ❑ A **Record Store consist of records**, each record is simply an **array of bytes**
- ❑ To find the **number of bytes used by a record store** use the `getSize()` method
- ❑ To know **how much space is available** call the method `getSizeAvailable()`.

Version and Timestamp

- ❑ Record stores maintain both a **version number** and **timestamp**
- ❑ Call the method `getVersion()` for the **version**
- ❑ Each time a record store is modified (by `addRecord`, `setRecord`, or `deleteRecord` methods) its *version* is incremented
- ❑ This can be used by MIDlets to quickly tell if anything has been modified
- ❑ Call `getLastModified()` for request the **last time the record store was modified**, expressed in milliseconds since midnight on January 1, 1970 (a `long` type value)
- ❑ To build a corresponding `Date` object:
 - `Date(mStore.getLastModified())`

Inside a RecordStore



Adding Records

- ❑ A **record** is simply an **array of bytes**
- ❑ Each record has an **integer identification number (id)**
- ❑ To **add** a new record, supply the byte array to the `addRecord()` method:

```
int addRecord(byte[] data, int offset, int numBytes)
```

- ❑ The record will be `numBytes` long taken from the `data` array, starting at `offset`
- ❑ The new record `ID` is returned - most of the other methods need this `ID` to identify a particular record
- ❑ The following illustrates adding a new record to Record Store named `rs`

```
String record = "This is a record"
```

```
Byte[] data = record.getBytes();
```

```
Int id = rs.addRecord(data, 0, data.length);
```

Retrieving Records

- ❑ You can **retrieve** a record by supplying the record ID to the following method (returns a freshly created byte array)

```
byte[] getRecord(int recordId)
```

- ❑ Another method **puts the record data into an array** that you supply and returns the number of bytes copied into your array

```
int getRecord(int recordId, byte[] buffer, int  
offset)
```

- ❑ `offset` - the index into the buffer in which to start copying
- ❑ For **efficiency** you would create one array and use it over and over again to retrieve all the records
- ❑ It is possible to use the method `getRecordSize(id)` before to call the `getRecord(...)` to check if the provided array is large enough - or needs to be expanded.

Deleting and Replacing Records

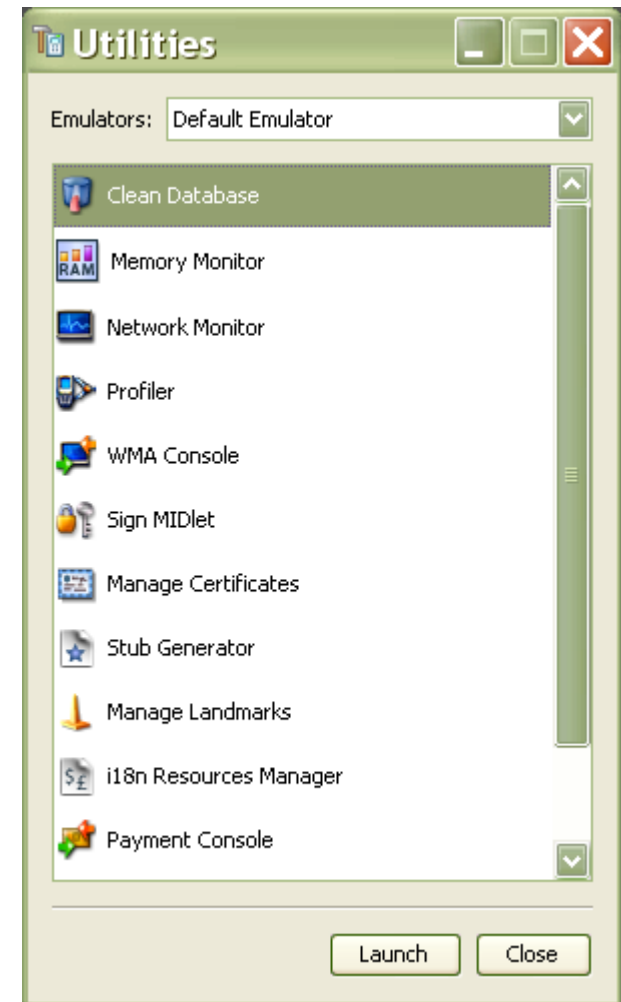
- ❑ There are two more record operations supported by `RecordStore`
- ❑ You can **remove** a record by calling the method
`deleteRecord(ID)`
- ❑ You can **replace** the data of an existing record by calling the following method
`void setRecord(int recordId, byte[] newData, int offset, int numBytes)`
- ❑ The `RecordStore` keeps an internal counter that it uses to assign record IDs
- ❑ You can find out what the **next record** ID will be by calling `getNextRecordID()`
- ❑ You can find out **how many record exist** in the `RecordStore` by calling `getNumRecords()`

Working with RecordEnumeration

- ❑ A `RecordEnumeration` - returned by a call to `enumerateRecords()` - allows you to **scroll** both **forward** and **backward**
- ❑ You can peek at the next or previous record ID
- ❑ `RecordEnumeration` offers the possibility of keeping its data synchronized with the actual `RecordStore` (*we shall see that later*)
- ❑ The available methods for moving through the selected records:
 - `nextRecord()`, `nextRecordId()`
 - `previousRecord()`, `previousRecordId()`
 - `reset()` moves the record pointer to the first record
 - `hasNext()` find out if there's a next record.

Where data are stored in WTK 2.5.2

- ❑ The emulator stores the RecordStores in `c:`
`\Documents and Settings\ricci\j2mewtk\2.5.2\appdb`
`\DefaultColorPhone`
- ❑ For instance if you created a RecordStore called "Bolzano-Store" you should find a file called like `"run_by_class_storage_#Bolzano%002d#Store.db"` in that directory
- ❑ If you want to delete all record stores in the WTK, select: **file>utilities** and then **Clean Databases**

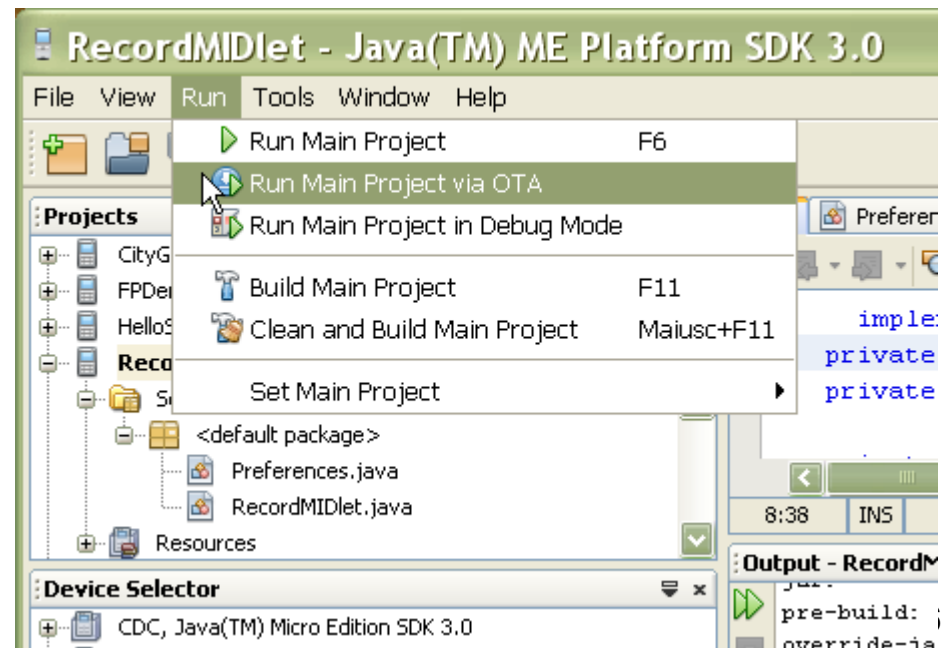


Record Store Files in WTK

- In WTK 3.0 the databases are stored in
 - C:\Documents and Settings\ricci\javame-sdk\3.0\work
 - Library/Application Support/javame-sdk/3.0/work (MAC)
 - There are directories called "1", "2", ... corresponding to the different emulators
 - But when you exit the Midlet the record store is cancelled
 - Example: in my case I have a file called "00000002-Bolzano#14#-Store.db" in directory "C:\Documents and Settings\ricci\javame-sdk\3.0\work\4\appdb"
- In WTK 2.5.2 you find these files in directories like "C:\Documents and Settings\ricci\j2mewtk\2.5.2\appdb\DefaultColorPhone"
 - When you exit the midlet the record store is not cancelled.

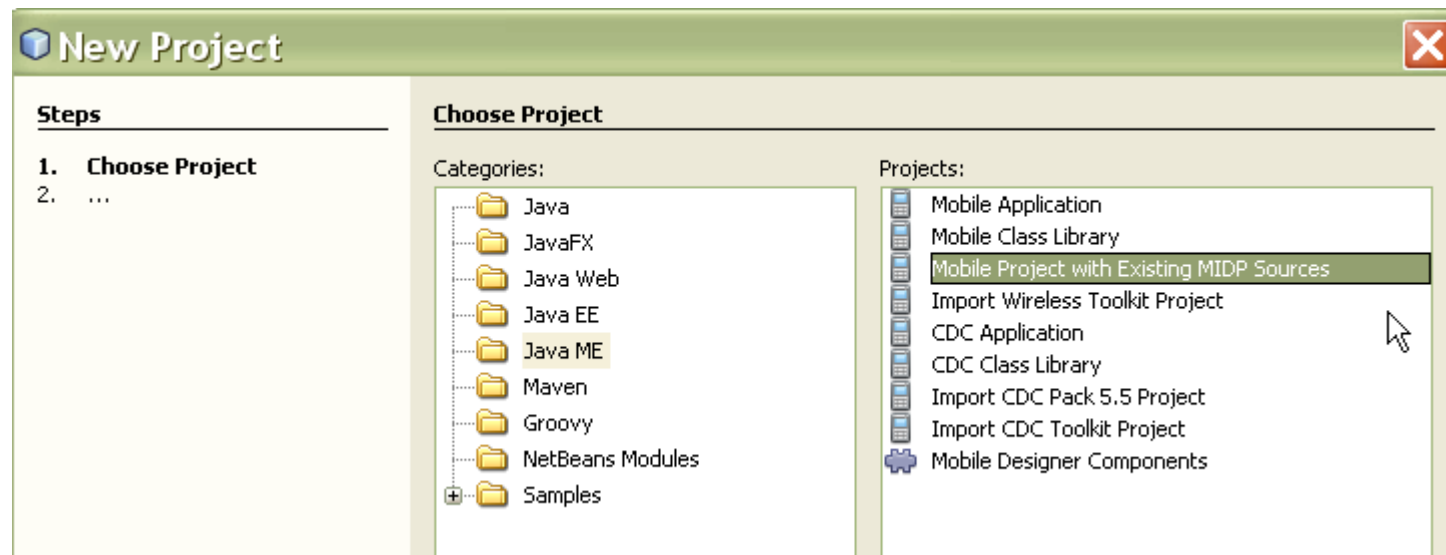
Run via OTA

- ❑ In WTK 3.0 after you have run a midlet in the emulator, the midlet – and the data created - is removed
- ❑ In order to keep the midlet (and the data) on the emulator you should install it via OTA
- ❑ This is also working in NetBeans with WTK 3.0
- ❑ Steps:
 - Set your project as "main"
 - Then choose the "run via OTA"
 - The midlet will be installed in your emulated device.



Working with NetBeans and WTK3.0

- ❑ You can manipulate a project using both NetBeans and WTK3.0
- ❑ Build the project in WTK3.0
- ❑ Import the sources of the project in NetBeans



Example: Saving User Preferences

- ❑ The following example saves a user name and password in `RecordStore`
- ❑ This record store contains only two records, e.g.: `<user|ciccio>`, `<password|occic>`
- ❑ The MIDlet screen is a Form that contains fields for entering the user name and password
- ❑ It uses a helper class, `Preferences`, to do all the `RecordStore` work
- ❑ `Preferences` is a wrapper for a map of string keys and values stored internally as a `Hashtable`
- ❑ A key and value pair is stored in a single record using a pipe character separator (`|`)
- ❑ `RecordMIDlet` saves the updated values back to the `RecordStore` in its `destroyApp()` method.

It works either in WTK2.5 or in WTK3.0 via OTA.

RecordMIDlet (I)

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

public class RecordMIDlet extends MIDlet implements CommandListener {
    private static final String kUser = "user";
    private static final String kPassword = "password";

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

    public RecordMIDlet() {
        try {
            mPreferences = new Preferences("preferences");
        }
        catch (RecordStoreException rse) {
            mForm = new Form("Exception");
            mForm.append(new StringItem(null, rse.toString()));
            mForm.addCommand(new Command("Exit", Command.EXIT, 0));
            mForm.setCommandListener(this);
            return;
        }

        mForm = new Form("Login");
        mUserField = new TextField("Name", mPreferences.get(kUser), 32, 0);
        mPasswordField = new TextField("Password", mPreferences.get(kPassword), 32, 0);
        mForm.append(mUserField);
        mForm.append(mPasswordField);

        mForm.addCommand(new Command("Exit", Command.EXIT, 0));
        mForm.setCommandListener(this);
    }
}
```



[code](#)

RecordMIDlet (II)

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

public void pauseApp() {}

public void destroyApp(boolean unconditional) {
    // Save the user name and password.
    mPreferences.put(kUser, mUserField.getString());
    mPreferences.put(kPassword, mPasswordField.getString());
    try { mPreferences.save(); }
    catch (RecordStoreException rse) {}
}

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

Preferences.java (I)

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

public class Preferences {
    private String mRecordStoreName;

    private Hashtable mHashtable;

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

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

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

[code](#)

Preferences.java (II)

```
private void load() throws RecordStoreException {
    RecordStore rs = null;
    RecordEnumeration re = null;

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

Preferences.java (III)

```
public void save() throws RecordStoreException {
    RecordStore rs = null;
    RecordEnumeration re = null;
    try {
        rs = RecordStore.openRecordStore(mRecordStoreName, true);
        re = rs.enumerateRecords(null, null, false);

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

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

Listening for Record Changes

- ❑ `RecordStores` support a JavaBeans-style listener mechanism
- ❑ The **listener** interface is
`javax.microedition.rms.RecordListener`
- ❑ It is possible to manage a listener with the following two methods

```
public void addRecordListener(RecordListener  
    listener) //add listener to a RecordStore
```

```
public void removeRecordListener(RecordListener  
    listener)
```

- ❑ The `RecordListener` interface has three methods, which must be implemented, for implementing a behavior if a record is added, changed or deleted:

```
recordAdded(), recordChanged(), recordDeleted()
```

Performing RecordStore Queries

- ❑ To perform a query to a `RecordStore` call:
`RecordEnumeration enumerateRecords(RecordFilter filter, RecordComparator comparator, boolean keepUpdated)`
- ❑ This method returns a sorted subset of the records in a `RecordStore`
- ❑ The `RecordFilter` (interface) determines **which records will be included** in the subset
- ❑ The `RecordComparator` (interface) is used to **sort the records**
- ❑ The returned `RecordEnumeration` (interface) allows to navigate through the returned records:
 - `nextRecord()`, `previousRecord()`, `hasNext()`, ...

Record Filter

- ❑ The simplest interface is `RecordFilter`
- ❑ When you call `enumerateRecords()` on a `RecordStore`, each record's data is retrieved
- ❑ `RecordFilter` **has a single method**, `matches()` which is called for each record
- ❑ Each record filter should examine the record data and return `true` if the record should be included
- ❑ The following filter ...

```
public class SevenFilter
implements javax.microedition.rms.RecordFilter {
    public boolean matches(byte[] candidate) {
        if (candidate.length == 0) return false;
        return (candidate[0] == 7);
    }
}
```

... selects records whose first byte is 7

Record Comparator

- ❑ The job of a `RecordComparator` is to determine the **order** of two sets of record data
- ❑ Without a `RecordComparator` the order of the records in the `RecordEnumeration` returned by `enumerateRecords()` **is not predictable**
- ❑ To implement the `RecordComparator` interface, you need to define one method:

```
int compare (byte[] rec1, byte[] rec2)
```
- ❑ This method examines the data contained in `rec1` and `rec2` and determines which of them should come first in a sorted list
- ❑ It must return one of the following constants defined in `RecordComparator`:
 - `PRECEDES` `rec1` come before `rec2`
 - `FOLLOWS` `rec1` come after `rec2`
 - `EQUIVALENT` `rec1` and `rec2` are the same

Example

```
public class SimpleComparator
    implements javax.microedition.rms.RecordComparator {
    public int compare(byte[] r1, byte[] r2) {
        int limit = Math.min(r1.length, r2.length);

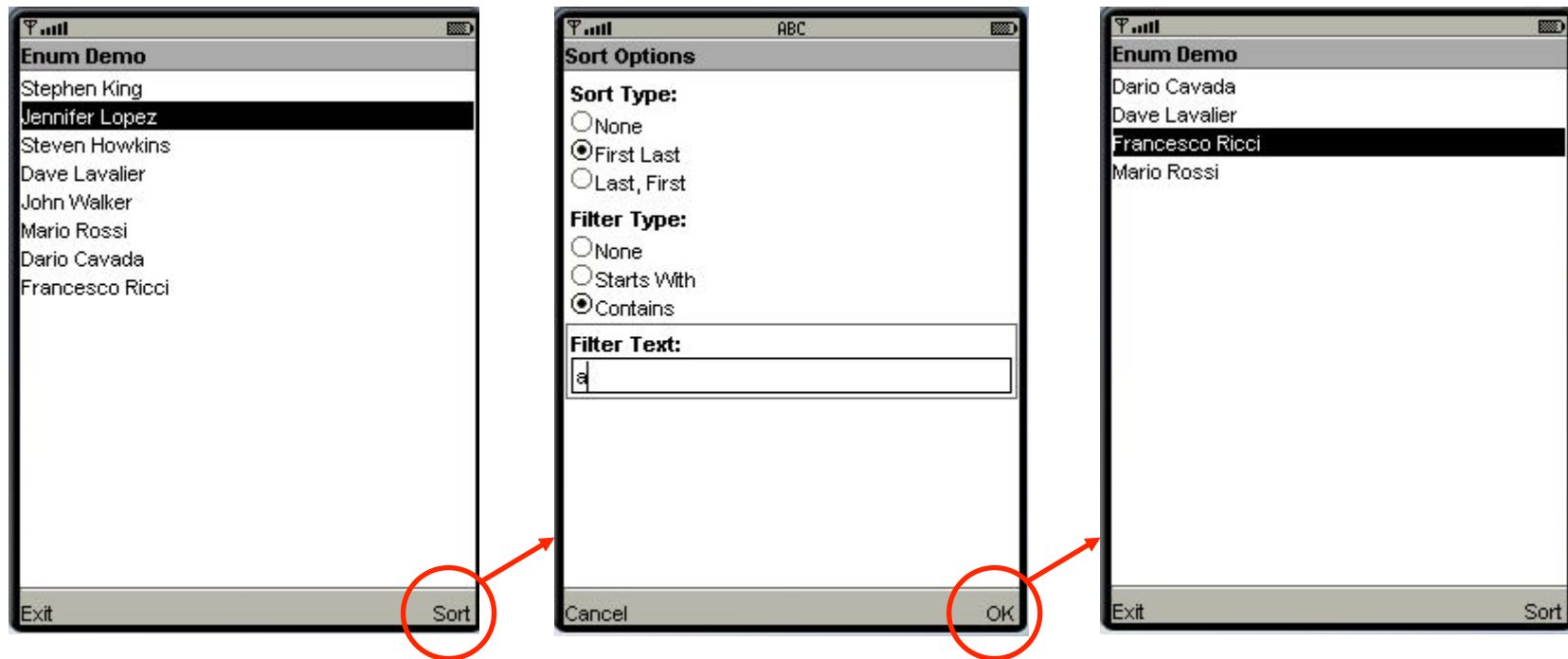
        for (int i=0; i<limit; i++) {
            if (r1[i]< r2[i])
                return PRECEDES;
            else if (r1[i] > r2[i]) ret
                return FOLLOWS;
        }
        return EQUIVALENT;
    }
}
```

Keeping a RecordEnumeration Up-to-Date

- ❑ It's possible that a `RecordStore` **will change** at the same time you're iterating through a `RecordEnumeration` (because of multithreads)
- ❑ To deal with this there are two ways
 - Call `rebuild()` which explicitly rebuilds the `RecordEnumeration`
 - Set the parameter `keepUpdated = true` in the `RecordEnumeration` method
- ❑ Using `keepUpdated` each time the `RecordStore` is changed, the `RecordEnumeration` is rebuild
- ❑ This is an expensive operation (in term of time), so if there are many `RecordStore` change, you'll be paying a price for it.

EnumDemo (RecordEnumeration example)

- ❑ In the `RecordStore` `String` are stored as objects
- ❑ You can sort and filter records using the `EnumList()` class (a `List` that implements the `RecordComparator` and `RecordFilter` interfaces)



Classes

- `EnumDemoMIDlet`: is the midlet
- `Record`: is a simple class with two member fields (string) for first and last names
- `EnumList`: is a `List` (displayable) defined as an inner class implementing the `RecordComparator` and `RecordFilter` interfaces – it shows the names after having sorted them
- `SortOptions`: is a `Form` where the `ChoiceGroup` (s) for specifying the sort and filter conditions are shown.

[code](#)

The Initial List

- The list of names displayed at the beginning is displayed by the following `List`:

```
class EnumList extends List implements RecordComparator,
    RecordFilter {
    private int sortBy; //sort conditions
    private int filterBy; //filter condition
    private String filterText; //filter condition
    private Record r1 = new Record();
    private Record r2 = new Record();

    // Constructor
    EnumList() {
        super("Enum Demo", IMPLICIT); //call the List
        addCommand(exitCommand); //constructor
        addCommand(sortCommand);
        setCommandListener(EnumDemoMIDlet.this);
    }
}
```

...

EnumDemo – The comparator

```
public int compare(byte[] rec1, byte[] rec2){
    try {
        ByteArrayInputStream bin = new ByteArrayInputStream(rec1);
        DataInputStream din = new DataInputStream(bin);

        r1.firstName = din.readUTF(); // r1 is defined in the comparator
        r1.lastName = din.readUTF(); // and is an instance of a class
                                    // containing two member fields

        bin = new ByteArrayInputStream(rec2); // firstName and lastName
        din = new DataInputStream(bin);     // that are strings

        r2.firstName = din.readUTF();
        r2.lastName = din.readUTF();

        if( sortBy == SORT_FIRST_LAST ){
            int cmp = r1.firstName.compareTo(r2.firstName );
            if (cmp != 0) return (cmp < 0 ? PRECEDES : FOLLOWS);
            cmp = r2.lastName.compareTo(r2.lastName);
            if (cmp != 0) return (cmp < 0 ? PRECEDES : FOLLOWS);
        } else if(sortBy == SORT_LAST_FIRST){
            int cmp = r1.lastName.compareTo(r2.lastName);
            if (cmp != 0) return (cmp < 0 ? PRECEDES : FOLLOWS);
            cmp = r2.firstName.compareTo(r2.firstName);
            if(cmp != 0) return (cmp < 0 ? PRECEDES : FOLLOWS);
        }
    } catch(Exception e){ }
    return EQUIVALENT;
}
```

< 0 if r1 is
lexicographically
less
than r2

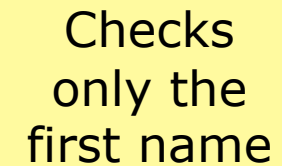
If they
have the
same first
name then
compare
last name

EnumDemo – The filter

```
public boolean matches(byte[] rec) {
    try {
        ByteArrayInputStream bin = new ByteArrayInputStream(rec);
        DataInputStream din = new DataInputStream(bin);

        r1.firstName = din.readUTF();
        r1.lastName = din.readUTF();

        if (filterBy == FILTER_STARTSWITH){ //if a filter condition
                                           //was set in the SortOption Form
            return (r1.firstName.startsWith(filterText) ||
                    r1.lastName.startsWith(filterText));
        } else if (filterBy == FILTER_CONTAINS){
            return (r1.firstName.indexOf(filterText) >= 0);
        }
    } catch( Exception e ){
    }
    return false;
}
```



Checks
only the
first name

Using Resource Files

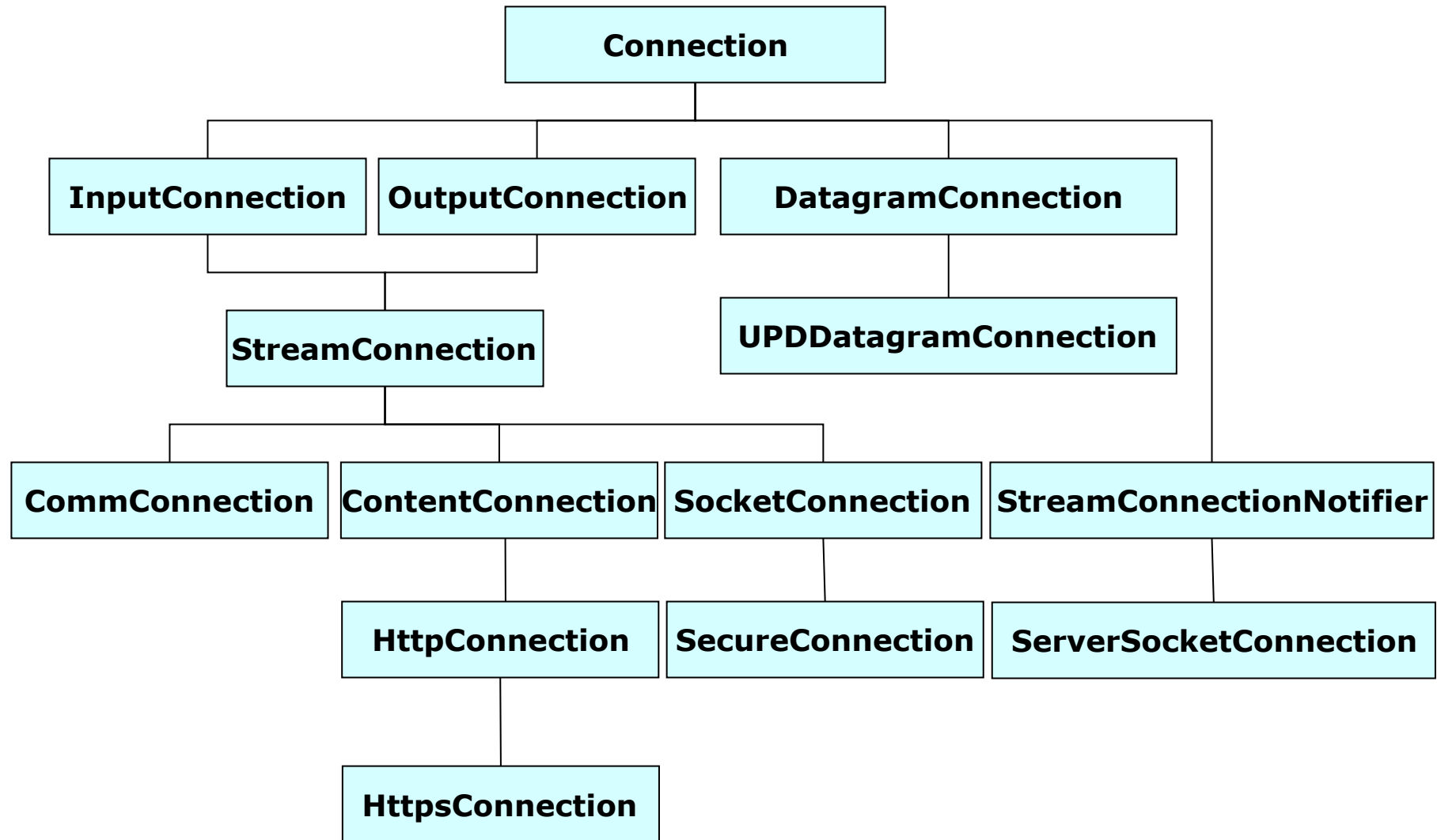
- ❑ **Resource files** are another form of persistent storage
- ❑ Accessing resource files is very simple, but they are important
- ❑ Resource files can be **images, text**, or other types of files that **are stored in** a MIDlet suite **JAR**
- ❑ These files are read only
- ❑ You can access a resource file as an `InputStream` by using the `getResourceAsStream()` method in `Class`
- ❑ A typical usage look like this:

```
InputStream is = this.getClass().getResourceAsStream("/myImage.png");
```

File Connection

- ❑ In the **optional package** `javax.microedition.io.file` (JSR 75) are included two additional persistent data storage mechanisms
 - **File systems**
 - **Personal Information Management (PIM)**
- ❑ Modern devices may have a memory card with megabytes or even gigabytes of data
- ❑ The **record store** mechanism of MIDP is **inefficient** for handling such **large-capacity storage**
- ❑ The persistent storage on these cards is accessed **as a file system** with **directories** and **files**
- ❑ Once you obtain an instance of a `FileConnection` (interface) using the `Connector` class, you can start working with the file system using the CLDC IO stream classes to read and write data.

Connection Interface Hierarchy



No `FileConnection` because is in an optional package

Generic Connection Framework

- General form

- `Connector.open("<protocol>:<address>;<parameters>")`

- HTTP

- `Connector.open("http://www.sun.com")`

- Sockets

- `Connector.open("socket://129.144.111.222:2800")`

- Communication port

- `Connector.open("comm:comm0,baudrate=9600")`

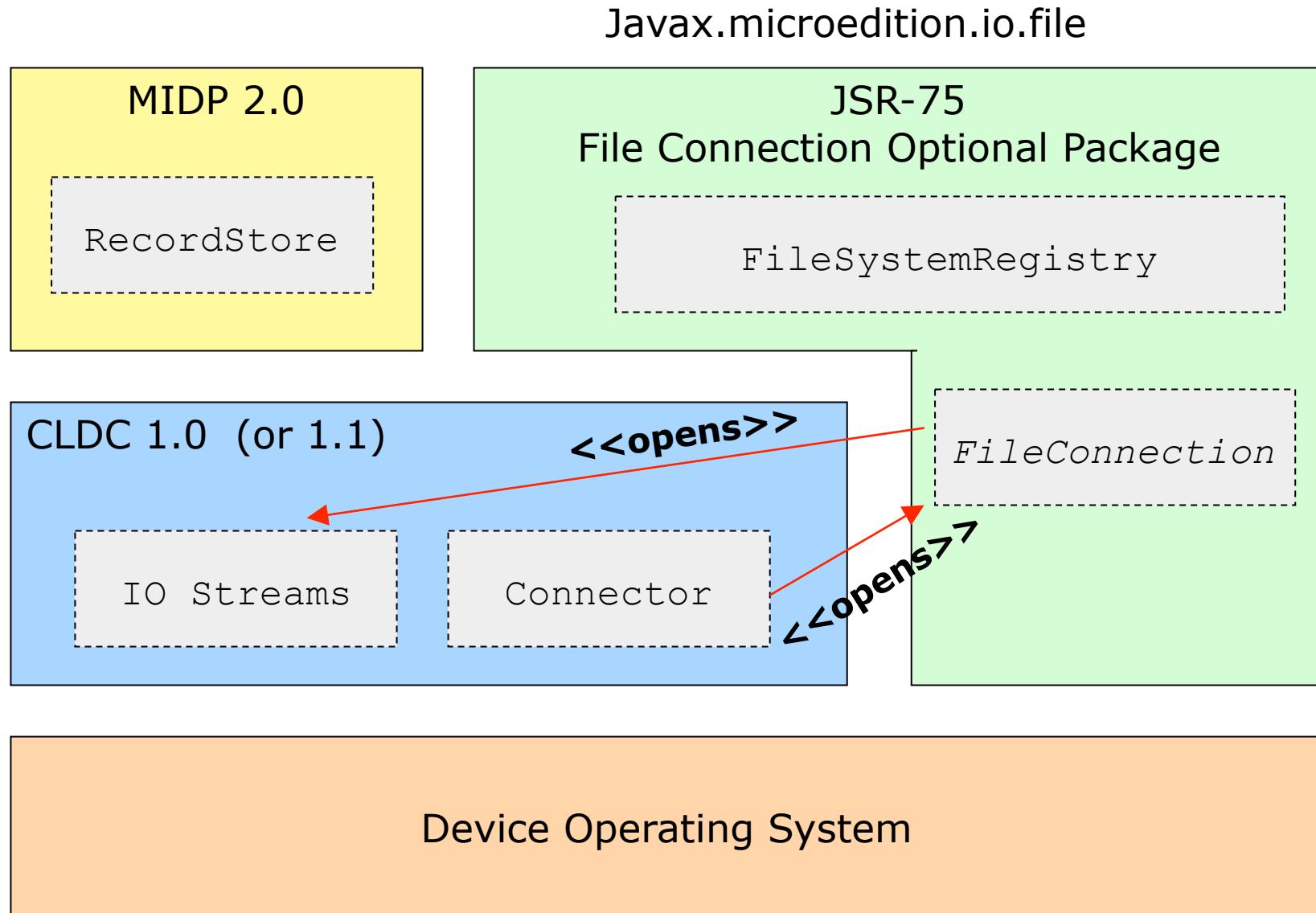
- Datagrams

- `Connector.open("datagram://129.144.111.222:2800")`

- These calls will return an object that implements one of `javax.microedition.io.Connection` interface

- Hence a binding of a protocol in J2ME can be done at run time!

Relationship between File Connection and CLDC



Determine if FileConnection API is Available

- ❑ To determine if the optional API is available and which version is installed you have to call:

```
String currentVersion = System.getProperty  
    ("microedition.io.file.FileConnection.vers  
ion")
```

- ❑ If the API is available a string with the version will be returned
- ❑ If the API is not available a **null** value is returned
- ❑ Currently only version "1.0" has been defined
- ❑ API documentation is not included in Netbeans or SDK – download it!

Obtaining a `FileConnection` from GCF

- ❑ To obtain a file connection use the following method of the `Connector` class

```
public static Connection open(String URL, int mode)
```

- ❑ The URL to obtain a file connection starts with `"file:///"` indicating that a file is on the local host
- ❑ The mode indicates the type of access, you can use `Connector.READ`, `Connector.WRITE` or `Connector.READ_WRITE`
- ❑ Example, opening a file on a SD card:

```
FileConnection fc = (FileConnection)
    Connector.open("file:///SDCard/abc.txt",
        Connector.READ);

InputStream is = fc.openInputStream();
```

Streams

- ❑ The `FileConnection` interface has five methods for obtaining a stream:
 - `DataInputStream openDataInputStream()`
 - `DataOutputStream openDataOutputStream()`
 - `InputStream openInputStream()`
 - `OutputStream openOutputStream()`
 - `OutputStream openOutputStream(long offset)`
- ❑ A `DataInputStream` is a subclass of `InputStream`, with many more methods for reading different data types – this is what you’ll use manage input (or output)
- ❑ Similarly for `DataOutputStream` and `OutputStream`

File or Directory

- ❑ An open `FileConnection` can be referring to either a **directory** or a **file**
- ❑ You can determine if the connection is associated with a directory calling the following method:

```
public boolean isDirectory()
```

- ❑ Some file system support **hidden file** - you can determine whether a file or directory is hidden by calling the method:

```
public boolean isHidden()
```

- ❑ You can change the attribute of a file using the method:

```
public void setHidden(boolean hiddenFlag)
```

Modifying File Attributes

- ❑ Some file **attributes** may prevent you from reading or writing to a file
- ❑ You can determine whether a file **can be read** by using this method:

```
public boolean canRead()
```

- ❑ Or find out if a file **can be written** using the following:

```
public boolean canWrite()
```

- ❑ To change the read/write attribute of a file use:

```
public void setReadable(boolean readable)
```

```
public void setWritable(boolean writable)
```

Directory and File Size

- ❑ Your application may need to determine the **available space** on a file system associated with a `FileConnection` instance
- ❑ You can call `availableSize()` method to obtain the available size in bytes
- ❑ Another method that retrieves the **size of storage already used** is `usedSize()`
- ❑ To find out **the size of the specific file** associated with the current `FileConnection` instance use the method `fileSize()` (*do not call it on a directory, you'll get an exception*)
- ❑ If `FileConnection` refers to a directory you can find the **total size of all the files in the directory** by calling `directorySize()` method.

Creating New Files or Directories

- ❑ To create a new file, you first have to call `Connector.open()` with the new file name and `Connector.WRITE` mode
 - `fc = (FileConnection) Connector.open("file:///root1/prefs.pfs", Connector.WRITE);`
- ❑ A `FileConnection` will be returned, **but the file does not yet exist**
- ❑ To verify its nonexistence use the method `boolean exists()`
- ❑ To **create** the file you simply call the `create()` method
- ❑ **Creating a new directory** is similar, after the `Connector.open()` operation (with the name of the new dir), call the `mkdir()` method.

Renaming and Deleting Files and Directories

- ❑ To **delete** a file or directory, you need to first open it (get a file connection) with `Connector.WRITE` mode enabled then call the method:

```
public void delete() throws IOException
```

- ❑ You should immediately call `close()` on the `FileConnection` **after** a `delete()`
- ❑ The `FileConnection` is no longer valid once the underlying file has been deleted
- ❑ Similarly to **rename** a file or directory open it with `Connector.WRITE` mode enabled and call the `rename(String newName)` method of the `FileConnection` instance with the new name as parameter.

Listing Directory Content

- When you have a `FileConnection` **to a directory**, you can obtain an `Enumeration` of its **content** (files and subdirectory) using these methods:

```
Enumeration list() throw IOException
```

```
Enumeration list(String filter, boolean includeHidden)  
    throw IOException
```

- The `Enumeration` **contains** objects of **string** type
- Each object in the enumeration is the **name** of a **file** or **directory**
- If the object is a directory, the name will end with `/`
- The second form of `list()` uses a filter that can contain wildcard characters (e.g., `*.txt`)
- To make **directory traversal** more efficient, a convenient method allows you to dive (**from a directory**) down a specific subdirectory or file (or move up `..`) with the current `FileConnection`:

```
setFileConnection(String itemName)
```

- This will reset the `FileConnection` to specified subdirectory, parent directory or file.

Path and URL Information

- ❑ The strings in the `Enumeration`, returned from a call to `list()`, **do not contain full path information**
- ❑ You can **get the complete URL** associated to an opened `FileConnection` calling the method `getURL()` (e.g., `file:///SDCARD1/MyIm/IM_123.jpg`)
- ❑ To get the complete path and preamble you can call the method `getPath()` (`"file:///SDCARD1/MyIm/"`)
- ❑ To get **just the name of the file or directory**, without the path and the preamble, you can call the `getName()` method
- ❑ If you are constructing file paths manually, you should always obtain the file separator to use by get the system property called `file.separator`
- ❑ Example:

```
String fileSep = System.getProperty  
("file.separator")
```

An Example - FCMIDlet

- ❑ The example stores preferences to the file system using the File Connection Optional Package
- ❑ The `FileBasedPreferences` example is similar to the `RecordStore` based Preferences class
 - In the file are stored `key|value` pairs (e.g., `user|francesco`, `password|ghgdsd`)
- ❑ It **maintains a preferences hash table** that is made persistent into the file system using the File Connection API
- ❑ To obtain the file system roots the method `listRoots()` is called on the `FileSystemRegistry` class and the first returned file root is used (usually `root1/` for the Wireless Toolkit)
- ❑ The `run()` method contains the code to write the content of the `HashTable` to the file system
- ❑ The user interface is identical to the one in `RecordMIDlet` (see previous slides).

FCMIDlet

```
public class FCMIDlet
extends MIDlet
implements CommandListener {
    private static final String kUser = "user";
    private static final String kPassword = "password";
    private FileBasedPreferences mPreferences;
    private Form mForm;
    private TextField mUserField, mPasswordField;
    private Command mExitCommand, mSaveCommand;

    public FCMIDlet() {
        try {
            verifyFileConnectionSupport();
            mPreferences = new FileBasedPreferences("preferences");
        }
        catch (IOException ex) {
            ... // open a form an say what is wrong
        }

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

[code](#)

FCMIDlet (II)

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

public void pauseApp() {}

public void savePrefs() {
    // Save the user name and password.
    mPreferences.put(kUser, mUserField.getString());
    mPreferences.put(kPassword, mPasswordField.getString());
    mPreferences.save();
}

public void destroyApp(boolean flg) {
}

public void commandAction(Command c, Displayable s) {
    if (c == mExitCommand) {
        if (mPreferences == null) {
            destroyApp(true);
            notifyDestroyed();
        }
        else if (!mPreferences.isSaving()) {
            destroyApp(true);
            notifyDestroyed();
        }
    }
    else if (c == mSaveCommand)
        savePrefs();
}
```

FCMIDlet (III)

```
public void verifyFileConnectionSupport() throws
    IOException {
    String version = "";
    version = System.getProperty
("microedition.io.file.FileConnection.version");
    if (version != null) {
        if (!version.equals("1.0"))
            throw new IOException("Package is not
version 1.0.");
    }
    else
        throw new IOException("File connection
optional package is not available.");
    }
}
```

FileBasedPreferences.java

- ❑ In the constructor the hash table is built, file root is found and the username and password loaded from the file
- ❑ The `load()` method actually load username and password
- ❑ The saving of the username and password is done in a separate thread
(`FileBasedPreferences` is a `Runnable`)
- ❑ In the `SavePref()` method the file is first opened, if there exist is deleted and a new file is created for writing the preferences.

[code](#)

PIM Optional Package - Overview

- ❑ Many devices have the ability to **maintain lists of phone numbers and names**
- ❑ Some devices also store addresses, e-mails, events, to-do lists and other personal information
- ❑ This PIM data is stored in **PIM database**
- ❑ A device vendor may now expose access to its PIM database through the PIM Optional Package specified in JSR 175 `javax.microedition.pim`
- ❑ The API centers around the PIM abstract class
- ❑ You cannot instantiate this class with the `new` operator, but using the class factory method to obtain the one and only instance

```
public static PIM getInstance()
```

Hierarchy of major classes and interfaces in PIM API

