Mobile Information Device Programming (18)

Lecturer: Alireza Mousavi
School of Engineering & Design
www.brunel.ac.uk/~emstaam

Record Management System

Additional Source:

Sing Li, Jonathan Knudsen (2005), Beginning J2ME From Novice to Professional, Third Edition, Apress, ISBN: 1-59059-479-7

Persistent Storage

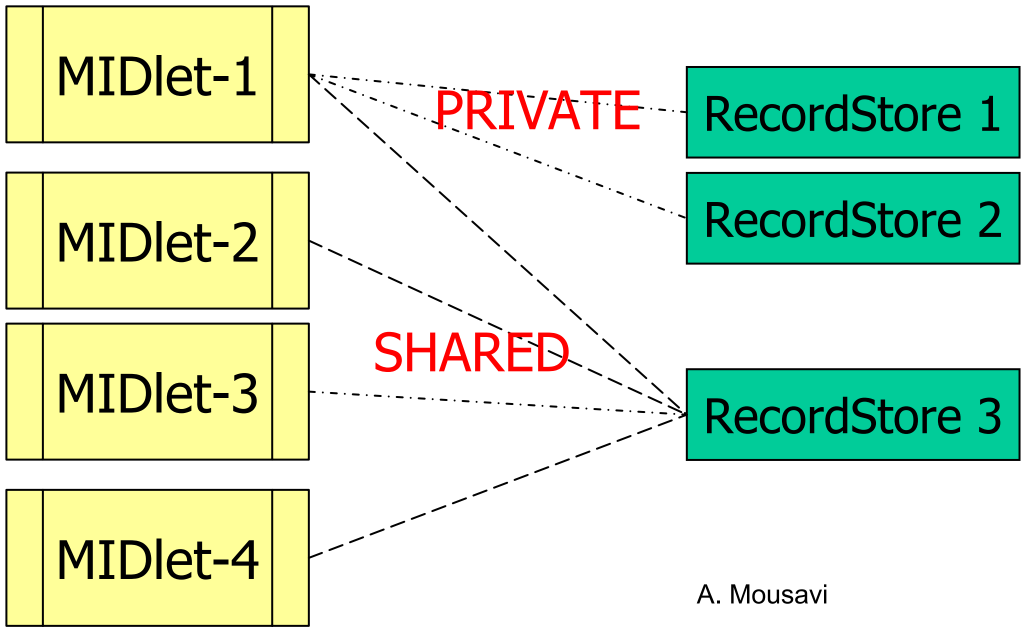
- The abstract concept of MIDP UI applications is extended to the persistent storage
- MIDP applications recognise small databases called record store (Li et al, 2005)
- It is the responsibility of the MIDP to map record stores in an acceptable manner to the available storage

Record Stores

- Record stores are small databases that contain small pieces of data called *records* [8Kb]

- Record store objects are instances of:

  javax.microedtion.rms.RecordStore

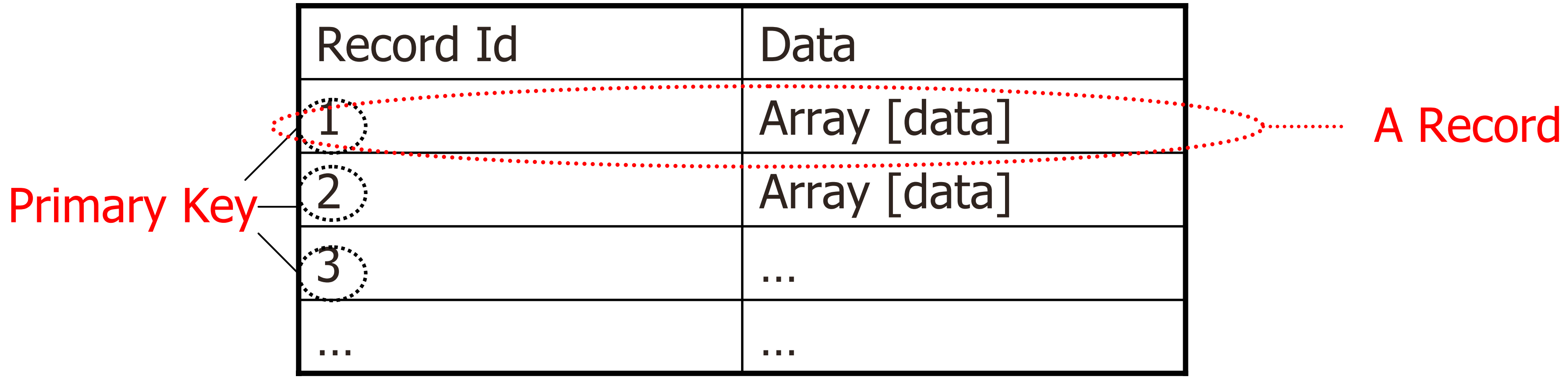


Record

- The Record Management System (RMS) uses non-volatile memory to store information.

- The record style database is like a series of rows in a table

Record Store



Record Id	Data
1	Array [data]
2	Array [data]
3	…
…	…

Note: Record stores have unique names in a MIDlet

Record Store API

This class is the core of RMS – Using this class we are able to **create, update, query, and delete** records and record stores

Record Store Class Methods

javax.microedition.rms.RocordStore

Method	Description
static openRecordStore(String recordStoreName, boolean createifNecessary) *Example: RecordStore rsID = RecordStore.open("ID", true);*	Open record store (e.g.myRecordStore), create if it does not already exist
void closeRecordStore( );	Close record store
static void deleteRecordStore(String recordStoreName) *Example: deleteRecordStore(" ID ");*	Delete record store
static String [ ] listRecordStore( )	List record store in MIDlet suite
int addRecord(byte [ ] data, int offset, int numBytes) *Example: byte [ ] myRecord = record.getBytes( );* *int id = rsID.addRecord(myRecord, 0, data.length*	Add a record
void setRecord(int recordID, byte [ ] newData, int offset, int numBytes)	Set or replace data in a record
void deleteRecord(int recordID)	Delete a record

Record Store Class Methods contd.

javax.microedition.rms.RocordStore

Method	Description
byte [ ] getRecord(int recordID)	Get byte array containing the data in the record (read the record)
int getRecord(int recordID, byte[ ] buffer, int offset)	Get contents of record into byte array parameter copying the content into a specified offset
int getRecordSize(int recordID)	Returns the size of the record
int getNextRecordID( )	Gets the number of the next record when adding a new record
int getNumRecords( )	Get number of records in the record store
RecordEnumeration enumerateRecords(RecordFilter filter, RecordComparator comparator, boolean keepUpdated)	Build an enumerator for running forward and backward in a record store
And other Methods (further reading Machow et al or Li et al)	

Managing Record Stores

1. Accessing (Opening, Closing and Removing) Record Stores

2. Sharing Record Stores

Record Enumerator

- Conduct simple DB queries
- RecordEnumeration provides methods to go forward and backward in a record store
- More efficient than simple looping since it provides filtering features (to search for specific) or comparator (sorting)
- Two main methods *nextRecord( )* to move forward and *previousRecord( )* to move backward.

Example:

```
…
RecordStore MyRec = RecordStore.open("ID", true); //creates a record store
…
RecordEnumeration myREn = myRec.enumerateRecords(null, null, false);
while( myREn.hasNextElement( )){
String MyStr = new String(myRec.nextRecord( ));
….
}
```

Back to 8

Record Store Enumerator - RecordFilter

- RecordFilter method is *match( )*
- It returns a boolean value of *true* if the record matches the search criteria from *enumerateRecords( )*

```
public class MyFilter implements RecordFilter {
public boolean matches( byte[ ] recordData )
{ ... // matching code here } }
```

// Here you could match a subset of data in a record store

```
 if (recordData.length != 0);
     return (recordData [0] == 1);  // in this case return the record where the first byte of
     data is equal to 1.
```

Record Store Enumerator - Comparator

- Similar to *java.util.Cpmpator* in J2SE
- To determine the order of two sets of record data
- Method associated with Comparator:

```
public int compare(byte[ ] myRecord 1, byte[ ] myREcord 2){
        int value 1 = getRecord(myRecord 1);
        int value 2 = getRecord(myRecord 2);
        if (value 1>= value 2){
        return PRECEDES; }
        else if (value 1 ==value 2){
        return EQUIVALENT; }
        else {
        FOLLOWS;
        }
```

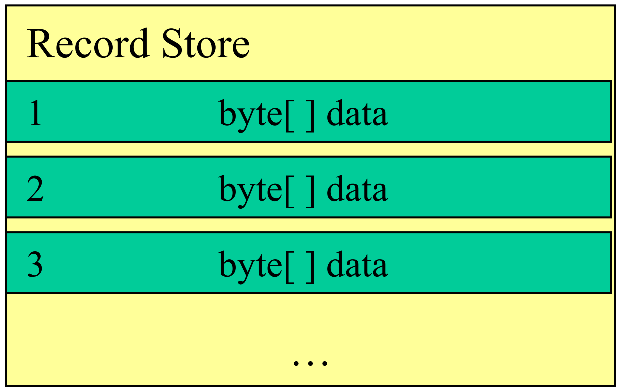
Write (add), read, and delete record store

- There are two methods to write into a record store:

(1) *public int addRecord (byte[ ] data, int offset, int numBytes)*

(2) public void setRecord (int recordId, byte[ ] newData, int offset, int numBytes)

  - Both methods accepts an array of byte as input
  - The difference is in the way the data is managed for reading and writing



Write (add), read, delete & save record store example 1

```
import java.io.*;
import java.util.* ;                          // call the needed libraries
import javax.microedtion.midlet.*;
import javax.microedition.rms.*;

public class RSExample extends MIDlet {
    private RecordStore rsex = null;      // declare an empty record store

    public RSExample ( ) {          // constructor
           openMyRS ( );      // instantiate a record store

           writeRecord("Jonnie");   write a few records
           writeRecord("Ronnie");
           writeRecord("Donnie");

           retireveRecords ( );   // read method will be written in this object
                                  //  continued on next page
```

Write (add), read, delete & save record store example 2

```
// continued from previous page
            closeMyRS ( );     // close the record store methods are put in here
            delMyRS ( );     // delete the record store
            saveMyRS ( );   // save record store
    }

//  startApp ( ), pauseApp ( ), destroyApp ( )

    public void openMyRS ( ) {
    try {
            rsex = RecordStore.openRecordStore( “Names”, true); //create RS if not exist
            }
    catch (Exception e){
            db (e.toString( ));  // if  RS cannot be created catch exception
            }
    }
 // Continued on next page
```

Write (add), read, delete & save record store example 3

```
// From previous page
    public void closeMyRS ( ){      // Closing the RS
    try{
            rsex.closeRecordStore( );
    }
    catch( Exception e){
            db (e.toString( ));
    }
  }
    public void delMyRS ( ){                // Deleting the RS
    if(RecordStore.listRecordStores ( ) != null     /* check to see if there is a RS in the
    list of RSs */
    try{
            RecordStore.deleteRecordStore("Names" ); }
    catch (Exception e){ db (e.toString( )); }
}
}
// continued on next page
```

Write (add), read, delete & save record store example 4

```
public void  writeRecord (String s){    // add a record onto RS
byte[ ]  rec = s.getByte( );
try{
          rse.addRecord(rec, 0, rec.length);  // set the offset at 0
          }
catch (Exception e){
db.toString( ));  }
}
// read record
public void retrieveRecord( ) {
int Datalen;  // declare an integer for the length of the data to be read
byte[ ] recDat = new byte[30];
for (int i = 1; I <= rse.getNumRecords( ), i++) {  // for the number of existing records
Datalen = rse.getRecord( “Record No.” + i +” is: “ + new String(recDat, 0, Datalen));
// Get byte array containing the data in the record (read the record)
System.outprintln(“ ***********************”);
} } catch (Exception e) {db(e.toString ( )); }  }
```

save record store example 1

```
public void saveIDNumber(String number) {
     try {
                RecordEnumeration enum = enumerate( );
                while (enum.hasNextElement( )) {
    // retrieve the next record
                rs.deleteRecord(enum.nextRecordId());
                                     }
                          } catch (Exception e) {

                          }
// create the required output streams
                ByteArrayOutputStream baos = new
                ByteArrayOutputStream();
                DataOutputStream dos = new DataOutputStream(baos);
// create the output stream
                          try {
```

save record store example 2

```
// write the number as a UTF encoded String
                dos.writeUTF(number);
        } catch (IOException ioe) {
                System.out.println(ioe);
                ioe.printStackTrace();
        }
        // get an array of bytes from the output stream
        byte[] b = baos.toByteArray();

        try {
                // add a new record containing the byte array
                // we get the record ID which we will return
                rs.addRecord(b, 0, b.length);
        } catch (RecordStoreException rse) {
                System.out.println(rse);
                rse.printStackTrace();
        }
   }
```

Listening for Record changes

- One can design listening objects to listen for changes to the Record Store
- The listener interface can be found in:

  javax.microedition.rms.RecordListener

- The two methods associated with RMS are:

  public void addRecordListener(RecordListener listener);

  public void removeRecordListener(RecordListener listener)

- The RecordListener interface has 3 methods:

  recordAdded( ), recordDeleted( ) and *RecordChanged*

More on this subject see: Sing Li et al 2005