

From Wireless Java Programming with J2ME, 0-672-32135-1

Chapter 8

Persistent Storage

Wireless devices, such as cell phones and pagers, normally have two types of memory: memory for running applications and memory for persistent storage. Data stored in persistent storage outlasts application programs that operate on that data. Users can store personal preference settings and/or information that can be dynamically changed over time in persistent storage.

A *database*, by definition, is a large, persistent, integrated collection of dynamic data that provides some operations to describe, establish, manipulate, and access this data. J2ME MIDP defines a simple record-oriented database system called a *record management system (RMS)*. Each table of the database system is called a record store, and each entry in a record store is called a record. The APIs for creating a record store and accessing and manipulating its records are described in the `javax.microedition.rms` package. This package defines two classes, three interfaces, and five exceptions:

- **Classes**--`RecordStore`, `RecordEnumeration`
- **Interfaces**--`RecordComparator`, `RecordFilter`, `RecordListener`
- **Exceptions**--`InvalidRecordIDException`, `RecordStoreException`, `RecordStoreFullException`, `RecordStoreNotFoundException`, `RecordStoreNotOpenException`

In the MIDP specification, `RecordEnumeration` is defined as an interface. However, an implementation of MIDP must provide an implementation of this interface. You don't need to implement the interface to use it. So, it is treated as a regular class in this chapter.

This chapter discusses how to create, delete, and access a record store and how to manipulate its records.

Record Stores

The RMS in MIDP is a flat-file database system. A record store, which is equivalent to a table in a database, is a file that consists of a collection of records. A record store is represented by a `RecordStore` class. Here are the rules regarding record store names:

1. Record store names are case sensitive and may consist of any combination of up to 32 Unicode

characters.

2. Record stores created by MIDlets within a MIDlet suite are located in the same directory. However, record store files created by MIDlets within different MIDlet suites are located in different directories. (This rule is not true for MotoSDK version 0.7.)

3. Record store names must be unique within a MIDlet suite.

4. Record stores created by MIDlets within one MIDlet suite are not accessible to MIDlets within other MIDlet suites.

Operations on a Record Store

The `RecordStore` class provides the following methods to create/open, close, and delete a record store, respectively:

```
public static RecordStore openRecordStore(String recordStoreName,
boolean createIfNecessary)
public void closeRecordStore()
public static void deleteRecordStore(String recordStoreName)
```

MIDlets can create a record store using

```
RecordStore.openRecordStore(recordStoreName, true)
```

If the record store specified by `recordStoreName` does not exist, a record store with `recordStoreName` will be created. If the record store already exists, no new record store will be created and the record store with `recordStoreName` will be opened.

MIDlets can open an existing record store with

```
RecordStore.openRecordStore(recordStoreName, false)
```

If the record store exists, it will be opened. However, if the record store does not exist, then a `RecordStoreNotFoundException` will be thrown. You can use this method to test whether a record store file exists, like this:

```
public boolean exist(String recordStoreName) {
boolean existF=true;
RecordStore rs=null;
if(recordStoreName.length(>32) return false;
try {
rs=RecordStore.openRecordStore(recordStoreName,false);
```

```

}catch(RecordStoreNotFoundException e) {existF=false;}

catch(Exception e){}

finally{

try{

rs.close();

}catch(Exception e){}

}

return existF;

}

```

Each record store contains at least a header field (the information maintained in the header is discussed in the following section, "The Record Store Header"). If not enough memory is allocated for a header, the record store will not be created and a `RecordStoreFullException` will be thrown. If other problems occur, such as a record store name that is too long or a corrupted record store file, then a `RecordStoreException` will be thrown.

For example, the MIDlet in Listing 8.1 will create a record store called `file1`.

Listing 8.1 `RecordStoreTest1.java`

```

import javax.microedition.midlet.*;

import javax.microedition.rms.*;

public class RecordStoreTest1 extends MIDlet {

public RecordStoreTest1() {

}

public void startApp() throws MIDletStateChangeException {

RecordStore rs=null;

try {

rs = RecordStore.openRecordStore("file1",true);

System.out.println("record store file1 is opened.");

}catch(Exception e){

System.out.println("Error: "+e.getMessage());

}finally{

//close the record store

```

```

try {

rs.closeRecordStore();

System.out.println("record store file1 is closed");

}catch (Exception e){

System.out.println("Error: "+e.getMessage());

}

}

destroyApp(true);

notifyDestroyed();

}

/**

* Pause the MIDlet

*/

public void pauseApp() {

}

/**

* Called by the framework before the application is unloaded

*/

public void destroyApp(boolean unconditional) {

}

}

```

If this MIDlet is within the MIDlet suite Chapter8 and you run it using Sun's emulator, you will find the record store file in \$J2MEWTK_DIR/nojam/Chapter8 with the name file1.db, where \$J2MEWTK_DIR is the directory in which J2MEWTK is installed. If you start the ktoolbar program from a command prompt or run the MIDlet from command line, you will find the file under the directory \$Current_DIR/nojam/Chapter8, where \$Current_DIR is the directory from which you start ktoolbar or the MIDlet. If you would like to start ktoolbar or a MIDlet from the command line, you should start from the same directory every time. Otherwise, you will not find the same record store you create before.

If you run the MIDlet using Motorola's emulator, you will find the record store file1.db in \$MOTOSDK_DIR\lib\resources\1, where \$MOTOSDK_DIR is the directory in which MotoSDK is installed. (Currently, MotoSDK does not differentiate MIDlet suites, so all record stores are located in the same directory.)

A record store should be closed with `closeRecordStore()` when you finish using it. (As long as the record store is open, it consumes resources of the wireless devices.) Each process or thread that opens a record store should call the close method explicitly. The `closeRecordStore()` method does not actually close the record store file; it just notifies the application manager that this process or thread has finished using the record store. The record store file will not *actually* close until all processes that have opened the record store call `closeRecordStore()`.

When a record store is no longer needed, you can delete it using the `deleteRecordStore(String recordStoreName)` method. Before you can delete a record store, the record store must be closed. Otherwise, a `RecordStoreException` will be thrown. The name rules also apply here; a MIDlet can only delete record stores associated with its MIDlet suite. If the record store is not found, a `RecordStoreNotFoundException` will be thrown. When a MIDlet suite is removed from a wireless device, all the record stores associated with it will also be removed.

The Record Store Header

Similar to other flat-file databases, a record store contains a header and many data blocks. Each data block is a record within the record store. Data blocks are linked together as a linked list, and each block maintains a pointer to the next block. To make such a flat-file database work, the header must maintain a link to the first data block and a link to the first block of free space. The header of a record store also maintains the following information:

1. The number of records in the record store. The initial value is zero. When a new record is added, this value increases by one. When a record is deleted, it decreases by one.
2. Version number. The initial version number is implementation dependent (normally, it is zero). The version number increments by a positive integer greater than zero (usually one). Each time a record store is modified (a record is added, modified, or deleted), the version number is incremented. This incrementing enables MIDlets to quickly tell if the record store has been modified by other processes or threads.
3. Last modified time. The time the record store was last modified, in the same format used by `System.currentTimeMillis()`. This value is complementary to the version number; they change together.
4. Next `recordId`. The `recordId` of the next record to be added to the record store. When the `addRecord()` method is called, the next `recordId` increases by one. This method can be useful for setting up pseudo-relational relationships. That is, if you have two or more record

stores whose records need to refer to one another, you can predetermine the recordIds of the records that will be created in one record store, before populating the fields and allocating the records in another record store.

The header information of a record store can be retrieved by using the following methods of the RecordStore class:

```
public long getLastModified()
```

```
public int getNextRecordID()
```

```
public int getNumRecords()
```

```
public int getVersion()
```

Additional Record Store Data

Some information about a record store is not maintained in its header, but is provided through implementations. For example,

```
public int getSizeAvailable()
```

returns the amount of additional room (in bytes) available for this record store to grow. Note that this is not necessarily the amount of extra MIDlet-level data that can be stored, because implementations may store additional data structures with each record to support integration with native applications, synchronization, and so on.

The method

```
public int getSize()
```

returns the amount of space, in bytes, that the record store occupies. The size returned includes any overhead associated with the implementation, such as the data structures used to hold the state of the record store.

The method

```
public static String[] listRecordStores()
```

returns an array of the names of record stores owned by the MIDlet suite. Note that if the MIDlet suite does not have any record stores, this function will return null.

Record Store Limitations

The RMS in MIDP is a bare-bones database system. The API provides only a minimum set of operations that allow you to initialize, update, and retrieve data. The schema of each record store is very simple: an integer recordId as the primary key and a byte array as a record field. Many popular database features, such as transaction control, crash recovery, and data integrity control, are not

supported by RMS.

Multiple MIDlets within a MIDlet suite or multiple threads of a MIDlet can open and access the same record store simultaneously. All record store operations are atomic, synchronous, and serialized, thus guaranteeing that no corruption will occur due to simultaneous accesses. Any operations (both reads and writes) will lock the record store until they finish. If a MIDlet uses multiple threads to access a record store, it is the MIDlet's responsibility to coordinate these accesses--otherwise, unintended consequences may result. For example, suppose you want to set up pseudo-relational relationships as mentioned earlier. One thread t1 predetermines the next recordId of record store A using the getNextRecordID() method and puts the recordId into a record of record store B. Before thread t1 creates a record in record store A, another thread t2 adds a record into record store A. At this time, the relationship between record stores A and B is corrupted. Coordinating threads t1 and t2 is the application's responsibility.

Records

Entries in a record store are called records. These records are represented by byte arrays and are uniquely identified by recordIds. Records can be added, deleted, retrieved, and modified from an opened record store with the following methods in the API:

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

public void deleteRecord(int recordId)

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

public byte[] getRecord(int recordId)

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

Adding Records

The first record created in a record store will have a recordId value of 1. Each subsequent record added to the record store will be assigned a recordId one greater than the record added before it. For example, the MIDlet in Listing 8.2 adds two records to the record store file1 that you created when executing Listing 8.1.

Listing 8.2 RecordStoreTest2.java

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

import javax.microedition.rms.*;

public class RecordStoreTest2 extends MIDlet {

    public RecordStoreTest2() {

    }

    public void startApp() throws MIDletStateChangeException {
```

```
RecordStore rs=null;

try {

rs = RecordStore.openRecordStore("file1",true);

byte data[]= new byte[4];

for(int j=0; j<2; j++) {

int i=rs.getNextRecordID();

data[0] = (byte)((i >> 24) & 0xff);

data[1] = (byte)((i >> 16) & 0xff);

data[2] = (byte)((i >> 8) & 0xff);

data[3] = (byte)(i & 0xff);

System.out.println("record "+rs.addRecord(data,0,4)+

" is added.");

}

}catch(Exception e){}

finally{

//close the record store

try {

rs.closeRecordStore();

}catch (Exception e){}

}

destroyApp(true);

notifyDestroyed();

}

/**

* Pause the MIDlet

*/

public void pauseApp() {

}

/**

* Called by the framework before the application is unloaded

*/
```

```
public void destroyApp(boolean unconditional) {
}
}
```

If you run the MIDlet in Listing 8.2 multiple times, you will get the following results:

```
Adding Record to RMS.....
record 1 is added.

Adding Record to RMS.....
record 2 is added.

Adding Record to RMS.....
record 3 is added.

Adding Record to RMS.....
record 4 is added.

...
```

The text "Adding Record to RMS....." is a system message that appears when adding records.

Deleting Records

You can delete a record from a record store by calling deleteRecord (int recordId). If the method is called before the record store is open, a RecordStoreNotOpenException will be thrown. If the record specified by the recordId does not exist in the record store, an InvalidRecordIDException will be thrown. If another general record store exception occurs, a RecordStoreException will be thrown.

The recordIds of deleted records are not reused; the primary key of the next new record, nextRecordID, increases by one over that of the previous new record.

Deleting a record does not make the record store file smaller; the data block of the deleted record is just marked free for future use. The example in Listing 8.3 adds two records and then removes one from the record store file1.

Listing 8.3 RecordStoreTest3.java

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

import javax.microedition.rms.*;

public class RecordStoreTest3 extends MIDlet {

public RecordStoreTest3() {
```

```
}

public void startApp() throws MIDletStateChangeException {

    RecordStore rs=null;

    try {

        rs = RecordStore.openRecordStore("file1",true);

        byte data[]= new byte[4];

        for(int j=0; j<2; j++) {

            int i=rs.getNextRecordID();

            data[0] = (byte)((i >> 24) & 0xff);

            data[1] = (byte)((i >> 16) & 0xff);

            data[2] = (byte)((i >> 8) & 0xff);

            data[3] = (byte)(i & 0xff);

            System.out.println("record "+rs.addRecord(data,0,4)+

                " is added.");

        }

        try {

            rs.deleteRecord(2);

            System.out.println("record 2 is deleted.");

        }catch(InvalidRecordIDException e) {

            System.out.println("record 2 does not exist");

        }

        }catch(Exception e){}

    finally{

        //close the record store

        try {

            rs.closeRecordStore();

        }catch (Exception e){}

    }

    destroyApp(true);

    notifyDestroyed();

}
```

```

/**
 * Pause the MIDlet
 */
public void pauseApp() {
}

/**
 * Called by the framework before the application is unloaded
 */
public void destroyApp(boolean unconditional) {
}
}

```

When you run the MIDlet in Listing 8.3 for the first time, you will get output similar to the following:

```

Adding Record to rMS.....
record 7 is added.

Adding Record to rMS.....
record 8 is added.

record 2 is deleted.

```

When you run it again, you will see output like this:

```

Adding Record to rMS.....
record 9 is added.

Adding Record to rMS.....
record 10 is added.

record 2 does not exist.

```

Two more records are added, and their recordIds are consecutive from the previous run. The record with a recordId value of 2 is not deleted, because that record no longer exists in the record store.

Monitoring Record Changes

A record store can monitor changes that happen to it using the RecordListener interface. When a change is made to the record store, an event is delivered to the registered RecordListener. (The event-handling model is the same as the delegation-based event-handling model discussed in Chapter 5, "Central Components of the

UI for Wireless Devices.") Three types of events--recordChanged, recordAdded, and recordDeleted--can be delivered to a RecordListener. These events are handled by the following methods of the RecordListener interface:

```
void recordAdded(RecordStore recordStore, int recordId)
void recordChanged(RecordStore recordStore, int recordId)
void recordDeleted(RecordStore recordStore, int recordId)
```

The RecordStore class provides the following methods to add or remove a RecordListener:

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

Unlike a Displayable, which can have at most one registered CommandListener, or a Form, which can have at most one registered ItemStateListener, a RecordStore object can have multiple registered RecordListeners.

For instance, suppose you want to keep two record stores synchronized. You can do so using a RecordListener as shown in Listing 8.4. When records in record store rs1 are modified, the changes will be automatically reflected in record store rs2.

Listing 8.4 RecordListenerTest.java

```
import javax.microedition.midlet.*;
import javax.microedition.rms.*;
public class RecordListenerTest extends MIDlet implements RecordListener {
RecordStore rs1=null;
RecordStore rs2=null;
public RecordListenerTest() {
}
public void startApp() throws MIDletStateChangeException {
//open two record stores
try {
rs1 = RecordStore.openRecordStore("test1",true);
rs1.addRecordListener(this);
}catch(Exception e) {}
try {
rs2 = RecordStore.openRecordStore("test2",true);
```

```
}catch(Exception e) {}

//add two records to rs1

byte data[]= new byte[4];

for(int j=0; j<2; j++) {

try {

int i=rs1.getNextRecordID();

data[0] = (byte)((i >> 24) & 0xff);

data[1] = (byte)((i >> 16) & 0xff);

data[2] = (byte)((i >> 8) & 0xff);

data[3] = (byte)(i & 0xff);

System.out.println("record #"+rs1.addRecord(data,0,4)+

" is added to record store 1");

}catch (Exception e){}

}

//modified the second last added record

try {

int id=rs1.getNextRecordID()-2;

data=rs1.getRecord(id);

data[0]+=1;

rs1.setRecord(id, data, 0, 4);

System.out.println("record #"+id+" of record store 1 is modified.");

}catch(Exception e) {}

//delete the last added

try {

int id=rs1.getNextRecordID()-1;

rs1.deleteRecord(id);

System.out.println("record #"+id+" of record store 1 is deleted.");

}catch (Exception e){}

// end

destroyApp(true);

notifyDestroyed();
```

```
}

/**
 * Pause the MIDlet
 */

public void pauseApp() {

}

/**
 * Called by the framework before the application is unloaded
 */

public void destroyApp(boolean unconditional) {

    //close the two record stores

    try {

        if(rs1!=null) rs1.closeRecordStore();

    }catch(Exception e){}

    try {

        if(rs2!=null) rs2.closeRecordStore();

    }catch (Exception e){}

}

//implement RecordListener

public void recordAdded(RecordStore rs, int rid) {

    if(rs==rs1) {

        try {

            byte data[]=rs.getRecord(rid);

            int id=rs2.addRecord(data, 0, data.length);

            System.out.println("record #"+id+

                " is added to record store 2.");

        }catch(Exception e) {}

    }

}

public void recordChanged(RecordStore rs, int rid) {

    if(rs==rs1) {
```

```

try {

byte data[]=rs.getRecord(rid);

rs2.setRecord(rid, data, 0, data.length);

System.out.println("record #"+rid+

" of record store 2 is modified.");

}catch(Exception e) {}

}

}

public void recordDeleted(RecordStore rs, int rid) {

if(rs==rs1) {

try {

rs2.deleteRecord(rid);

System.out.println("record #"+rid+

" of record store 2 is deleted.");

}catch(Exception e) {}

}

}

}

```

If you run the MIDlet, you will see output like following:

```

Adding Record to RMS.....

Adding Record to RMS.....

record #1 is added to record store 2.

record #1 is added to record store 1

Adding Record to RMS.....

Adding Record to RMS.....

record #2 is added to record store 2.

record #2 is added to record store 1

record #1 of record store 2 is modified.

record #1 of record store 1 is modified.

record #2 of record store 2 is deleted.

record #2 of record store 1 is deleted.

```

Record store rs2 automatically copies record changes made to record store rs1.

RecordEnumeration

After records are deleted, recordIds of records in the record store are no longer consecutive. You can retrieve all the records using the MIDlet shown in Listing 8.5, but it is *not* a good and efficient way to do so.

Listing 8.5 RecordStoreList1.java

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

import javax.microedition.rms.*;

public class RecordStoreList1 extends MIDlet {

    public RecordStoreList1() {

    }

    public void startApp() throws MIDletStateChangeException {

        RecordStore rs=null;

        try {

            rs = RecordStore.openRecordStore("file1", true);

            byte data[];

            System.out.println(rs.getNumRecords()+

                " records are in the record store.");

            for(int i=1; i<rs.getNextRecordID(); i++) {

                try{

                    data=rs.getRecord(i);

                    System.out.println("record "+i+" is retrieved.");

                }catch (Exception e) {}

            }

            }catch(Exception e){}

            finally{

                //close the record store

                try {

                    rs.closeRecordStore();

                }catch (Exception e){}
```

```
}  
  
destroyApp(true);  
  
notifyDestroyed();  
  
}  
  
/**  
  
* Pause the MIDlet  
  
*/  
  
public void pauseApp() {  
  
}  
  
/**  
  
* Called by the framework before the application is unloaded  
  
*/  
  
public void destroyApp(boolean unconditional) {  
  
}  
  
}
```

If you run the MIDlet in Listing 8.5, you will get output as follows:

```
9 records are in the record store.  
  
record 1 is retrieved.  
  
record 3 is retrieved.  
  
record 4 is retrieved.  
  
...
```

This is a trial-and-error way to retrieve all records: If the record specified by the recordId exists, it will be retrieved with the getRecord() method; but if the record identified by the recordId does not exist, an InvalidRecordIDException will be thrown. If many records have been deleted since the record store was created, getting records with this MIDlet will be very inefficient.

The API of RMS provides a better way to traverse all records in a record store. RecordEnumeration is a class representing a bidirectional record store record enumerator. MIDP defines RecordEnumeration as an interface, because the detailed implementations are left for device manufacturers. Any device manufacturers that implement J2ME MIDP must implement RecordEnumeration. For developers like us, the RecordEnumeration is

a solid class.

A RecordEnumeration is similar to a double-linked list with each node representing a recordId. The RecordEnumeration logically maintains a sequence of the recordIds of the records in a record store. The RecordStore class provides the method for creating a RecordEnumeration:

```
public RecordEnumeration enumerateRecords(RecordFilter filter,  
RecordComparator comparator, boolean keepUpdated)
```

If a RecordEnumeration is created with keepUpdated set to true, the enumerator will keep its enumeration current with any changes in the record store's records. If the RecordEnumeration is created with keepUpdated set to false, the MIDlet is responsible for updating the enumerator with the rebuild() method. If the enumeration is not kept current, it may return recordIds for records that have been deleted, or it may miss records that are added later. When keepUpdated is set to true, performance may be penalized because some unnecessary enumeration updates can be triggered by changes in the record store. For example, when both the filter and comparator are set to null, a data change in a record should not trigger an update of the enumerator. A MIDlet also can wait to update the enumerator until all changes are done, if the enumerator is not used between changes.

The keepUpdated setting of a RecordEnumeration object can be changed with the method

```
void keepUpdated(boolean keepUpdated)
```

The setting can be retrieved by calling the method

```
boolean isKeptUpdated()
```

Accessing and Traversing Records

The API of the RecordEnumeration class provides the following methods to access or traverse all records in a record store:

```
void destroy()
```

```
boolean hasNextElement()
```

```
boolean hasPreviousElement()
```

```
byte[] nextRecord()
```

```
int nextRecordId()
```

```
int numRecords()
```

```
byte[] previousRecord()
```

```
int previousRecordId()
```

```
void rebuild()
```

```
void reset()
```

If you need to use the enumerator multiple times, remember to use the `reset()` method to reset the current node (pointer) to the first `recordId` in the linked list. The example in Listing 8.6 lists all records with even `recordIds` and then lists all records with odd `recordIds`.

Listing 8.6 RecordStoreList2.java

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

import javax.microedition.rms.*;

public class RecordStoreList2 extends MIDlet {

    public RecordStoreList2() {

    }

    public void startApp() throws MIDletStateChangeException {

        RecordStore rs=null;

        RecordEnumeration re=null;

        try {

            rs = RecordStore.openRecordStore("file1", true);

            byte data[];

            re= rs.enumerateRecords(null, null, false);

            System.out.println(re.numRecords()+

                " records are in the record store.");

            System.out.println("records with even recordIds:");

            for(int i=1; i<=re.numRecords(); i++) {

                try{

                    int j=re.nextRecordId();

                    if(j%2==0) {

                        data=rs.getRecord(j);

                        System.out.println("record "+j+" is retrieved.");

                    }

                }catch (Exception e) {}

            }

        }

    }

}
```

```
System.out.println("records with odd recordIds:");

/* now the current pointer points to the last node of the
 * enumerator. To use it again, you have to reset it.
 */

re.reset();

while(re.hasNextElement()) {

try{

int j=re.nextRecordId();

if(j%2==1) {

data=rs.getRecord(j);

System.out.println("record "+j+" is retrieved.");

}

}catch (Exception e) {}

}

}catch(Exception e){}

finally{

//destroy the record enumerator

try {

re.destroy();

}catch(Exception e){}

//close the record store

try {

rs.closeRecordStore();

}catch (Exception e){}

}

destroyApp(true);

notifyDestroyed();

}

/**
 * Pause the MIDlet
 */
```

```

public void pauseApp() {
}

/**
 * Called by the framework before the application is unloaded
 */

public void destroyApp(boolean unconditional) {
}
}

```

When you run the MIDlet in Listing 8.6, you will see output like the following:

```

9 records are in the record store.

records with even recordIds:

record 4 is retrieved.

record 6 is retrieved.

record 8 is retrieved.

record 10 is retrieved.

records with odd recordIds:

record 1 is retrieved.

record 3 is retrieved.

record 5 is retrieved.

...

```

Record 2 is not retrieved because it was deleted from the record store when you ran the MIDlet in Listing 8.3.

Creating Tables with Multiple Columns

Generally, a database table has multiple columns. A record in a record store has only one data field, which is represented by a byte array. However, you can pack multiple fields into a single record using UTF-8 encoding, so that the record store is equivalent to a multiple-column table. J2ME's `java.io` package inherits the classes `DataInputStream`, `DataOutputStream`, `ByteArrayInputStream`, and `ByteArrayOutputStream` from J2SE's `java.io` package. You can use these classes to pack and unpack different data types into and out of byte arrays.

For example, suppose you want to store an appointment record with the following four fields: `time` (long integer), `length` (integer),

location (string), and subject (string). You can create this class for appointments:

```
public class Appointment {
    private int id;
    private long time;
    private int length;
    private String location;
    private String subject;
    ...
}
```

The id in the class corresponds to the recordId in the record store. If you want to save an appointment record to a record store, you can first pack all the fields into a byte array like this:

```
public class Appointment {
    ...
    /* convert to the byte array that will be saved in the record store*/
    public byte[] toBytes() {
        byte data[]=null;
        try {
            ByteArrayOutputStream baos= new ByteArrayOutputStream();
            DataOutputStream dos= new DataOutputStream(baos);
            dos.writeLong(time);
            dos.writeInt(length);
            dos.writeUTF(location);
            dos.writeUTF(subject);
            data=baos.toByteArray();
            baos.close();
            dos.close();
        }catch(Exception e) {}
        return data;
    }
    ...
}
```

```
}

```

Then, you can save the byte array to the record store as follows:

```
public class CalendarDB {
    RecordStore rs=null;
    ...
    //save an appointment (new and old)
    public boolean save(Appointment app) {
        if(rs==null) return false;
        boolean success=false;
        try {
            byte[] data= app.toBytes();
            int id= app.getId();
            if(id==0) { //create a new record
                id=rs.addRecord(data,0,data.length);
                app.setId(id);
            }
            else { //update the old record
                rs.setRecord(id, data, 0, data.length);
            }
            success=true;
        }catch(Exception e){
            System.out.println("Error: "+e.getMessage());
        }
        return success;
    }
    ...
}
```

In the `CalendarDB.save()` method, the appointment's `id` is a flag that indicates whether this appointment record is an old appointment or a new one. If the appointment is a new one, a new record is created with

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

and a new `recordId` is assigned to the appointment's `id` as

```
app.setId(id);
```

If the appointment already exists in the record store, it is updated with

```
rs.setRecord(id,data,0,data.length);
```

*****Begin Warning*****

There is a bug in the `RecordStore.setRecord()` method of Sun's J2ME toolkit. If the length of the new record's data is larger than the length of the old data, the record store will be corrupted.

*****End Warning*****

In Chapter 6, "Using High-Level APIs in UI Development," you saw a UI program `AppointmentForm.java` that takes user input and outputs the result to a terminal. Now you can link the UI part with a record store and save users' input as follows (the complete program is listed in the section "Sample Application: Mobile Scheduler" later in the chapter):

```
public class AppointmentForm extends Form implements CommandListener{

    private Display display;

    private Displayable parent;

    private Command saveCommand = new Command("Save", Command.OK, 2);

    private Command deleteCommand = new Command("Delete",Command.OK,2);

    private Command cancelCommand = new Command("Cancel", Command.CANCEL, 1);

    private Appointment app;

    private CalendarDB calendarDB;

    ...

    public void commandAction(Command c, Displayable d) {

        if(c==saveCommand) {

            //time

            app.setTime(((DateField)get(0)).getDate().getTime());

            TextField tf;

            //length

            tf= (TextField) get(1);

            app.setLength(Integer.parseInt(tf.getString()));
```

```
//location
tf= (TextField) get(2);
app.setLocation(tf.getString());

//subject
tf= (TextField) get(3);
app.setSubject(tf.getString());

//create an alert
Alert saveInfo= new Alert("Save Appointment", "",
null, AlertType.INFO);
saveInfo.setTimeout(Alert.FOREVER);
if(calendarDB.save(app)) {
saveInfo.setString("Success!");
}
else {
saveInfo.setString("Fail!");
}
display.setCurrent(saveInfo,parent);
}
else if(c==deleteCommand) {
//create an alert
Alert deleteInfo= new Alert("Delete Appointment", "",
null, AlertType.INFO);
deleteInfo.setTimeout(Alert.FOREVER);
if(calendarDB.delete(app)) {
deleteInfo.setString("Success!");
}
else {
deleteInfo.setString("Fail!");
}
display.setCurrent(deleteInfo,parent);
}
}
```

```

else if(c==cancelCommand) {
display.setCurrent(parent);
}
}
}

```

For instance, suppose you want to create an appointment with these values

```

time: 1/1/01, 19:00
length: 120 minutes
location: Universal city
subject: new year celebration

```

and then save it. Because Motorola's emulator gives better performance when used with record stores, we will use it in the rest of this chapter. You can start the Scheduler MIDlet and select the Add Appointment function. After you input all the information (see Figure 8.1), click Save. If the appointment is saved properly, you will see a success message, as shown as Figure 8.2.

Figure 8.1

Adding an appointment.

*****Insert Figure 8.1 06fig01 PC Crop**

Figure 8.2

After successfully adding an appointment.

*****Insert Figure 8.2 06fig02 PC Crop**

After you save appointments, you want to retrieve them for review. The next example shows how to retrieve and display all the appointments in the record store:

```

public class CalendarDB {
RecordStore rs=null;
...
public Vector retrieveAll() {
RecordEnumeration re=null;
Vector apps= new Vector();

```

```

try {
    re = rs.enumerateRecords(null, null, false);
    while(re.hasNextElement()) {
        int rec_id=re.nextRecordId();
        apps.addElement(new Appointment(rec_id,
            rs.getRecord(rec_id)));
    }
} catch(Exception e) {}
finally{
    //destroy the enumerator
    if(re!=null) re.destroy();
}
return apps;
}
...
}

```

The `retrieveAll()` method creates a `RecordEnumeration` for use in stepping through all the records. After a record is retrieved with `rs.getRecord(rec_id)`, the byte array is parsed to get the value of each field in the appointment record as follows:

```

public class Appointment {
    ...

    /* rec is the byte array saved in record store */
    public Appointment (int id, byte[] rec) {
        id= id;
        init_app(rec);
    }

    /* rec is the byte array saved in record store */
    public void init_app(byte[] rec) {
        // parse the record
        ByteArrayInputStream bais= new ByteArrayInputStream(rec);
        DataInputStream dis= new DataInputStream(bais);
    }
}

```

```

try {
    time=dis.readLong();
    length=dis.readInt();
    location=dis.readUTF();
    subject=dis.readUTF();
} catch (Exception e) {}
}
...
}

```

You can add the Retrieve Appointments function to the Scheduler MIDlet (see the section "Sample Application: Mobile Scheduler"). If you run the Scheduler MIDlet and invoke Retrieve Appointments, you will see all the appointments listed as in Figure 8.3. If you select an appointment in the list, the appointment's detail information will be shown, as in Figure 8.4. From here, you can edit the appointment or you can save it or delete it by selecting the appropriate menu item (see Figure 8.5).

Figure 8.3

List all appointments.

*****Insert Figure 8.3 06fig03 PC Crop**

Figure 8.4

Edit an appointment selected from the list.

*****Insert Figure 8.4 06fig04 PC Crop**

Figure 8.5

Available operations on a selected appointment.

*****Insert Figure 8.5 06fig05 PC Crop**

The RecordFilter and RecordComparator Interfaces

In general, you will insert appointments into the record store whenever you need to. But when you view or retrieve appointments, you usually want to sort them by appointment time and filter out appointments that are too old. The RMS package provides two interfaces, RecordFilter and RecordComparator, to meet this need.

The RecordFilter interface has only one function:

```
public boolean matches(byte[] candidate)
```

You can define criteria for selecting a record in this method. Listing 8.7 defines an AppointmentFilter class that implements the RecordFilter interface. Any appointments that are later than a cutoff time will pass through the filter.

Listing 8.7 AppointmentFilter.java

```
/*
 * AppointmentFilter.java
 *
 */
import javax.microedition.rms.*;

public class AppointmentFilter implements RecordFilter{

    private long cutoff;

    public AppointmentFilter (long _cutoff) {

        cutoff= _cutoff;

    }

    public boolean matches(byte[] candidate) {

        Appointment app= new Appointment();

        app.init_app(candidate);

        if (app.getTime()>cutoff) {

            return true;

        }

        else {

            return false;

        }

    }

}
```

The RecordComparator interface provides an easy way to sort records. It also has only one function:

```
int compare(byte[] rec1, byte[] rec2)
```

The return value must be one of three constants--PRECEDES, FOLLOWS, and EQUIVALENT--that indicates the ordering of the two records. If rec1 precedes rec2 in the sort order, the function returns

RecordComparator.PRECEDES. If rec1 follows rec2 in the sort order, the function returns RecordComparator.FOLLOWS. If rec1 and rec2 are equivalent in terms of sort order, the function returns RecordComparator.EQUIVALENT.

To sort appointments by appointment time, you can create an AppointmentComparator class as shown in Listing 8.8. This class implements the RecordComparator interface.

Listing 8.8 AppointmentComparator.java

```

/*
 * AppointmentComparator.java
 *
 */
import javax.microedition.rms.*;

public class AppointmentComparator implements RecordComparator{

public int compare(byte[] rec1, byte[] rec2) {

Appointment app1= new Appointment();

app1.init_app(rec1);

Appointment app2= new Appointment();

app2.init_app(rec2);

if (app1.getTime()==app2.getTime()) {

return RecordComparator.EQUIVALENT;

}

else if(app1.getTime()<app2.getTime()) {

return RecordComparator.PRECEDES;

}

else {

return RecordComparator.FOLLOWS;

}

}

}

```

The compare() method of the AppointmentComparator class determines the order of two records by appointment time. For instance, suppose you don't want to see any appointments that are 90 days old and you want to sort appointments by time. You can use AppointmentFilter and AppointmentComparator to modify the retrieveAll() method in the

CalendarDB class as follows:

```
public class CalendarDB {

public Vector retrieveAll() {

RecordEnumeration re=null;

Vector apps= new Vector();

try {

//cutoff is 90 days old

long cutoff=System.currentTimeMillis()-

new Integer(90).longValue()*24*60*60000;

RecordFilter rf = new AppointmentFilter(cutoff);

RecordComparator rc = new AppointmentComparator();

re = rs.enumerateRecords(rf,rc,false);

while(re.hasNextElement()) {

int rec_id=re.nextRecordId();

apps.addElement(new Appointment(rec_id,rs.getRecord(rec_id)));

}

}catch(Exception e) {}

finally{

//destroy the enumerator

if(re!=null) re.destroy();

}

return apps;

}
```

Sample Application: Mobile Scheduler

Chapter 6 presented the Scheduler.java MIDlet. To save appointment data on wireless devices, you need to use a record store. The ability to retrieve appointments has also been added. After all appointments are retrieved, they can be displayed in a list or can be graphically displayed using the MonthlyScheduleViewer developed in Chapter 7, "Using Low-Level APIs in UI Development." The updated Scheduler.java shown in Listing 8.9 links together pieces of programs developed in Chapters 6, 7 and this chapter.

Listing 8.9 Scheduler.java

```
import java.util.*;
```

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

import javax.microedition.lcdui.*;

public class Scheduler extends MIDlet implements CommandListener{

private Calendar calendar;

private List menu;

private AppointmentForm appForm=null;

private List appList=null;

private SynchOptionForm soForm=null;

private MonthlyScheduleViewer monthlyviewer=null;

private String[] options={"Add Appointment",

"Retrieve Appointments",

"Synch Option Setup",

"Calendar view"};

private Display display;

private Command backCommand= new Command("Back",Command.BACK,1);

private Command exitCommand = new Command("Exit",Command.EXIT,1);

private Command detailCommand = new Command("Detail",Command.SCREEN, 1);

private CalendarDB calendarDB;

private Vector apps;

private Hashtable app_table;

private SynchOption so;

public Scheduler() {

//create an implicit choice list, and use it as start menu

menu= new List("Scheduler", List.IMPLICIT,options,null);

menu.addCommand(exitCommand);

menu.setCommandListener(this);

//get a calendar

calendar=Calendar.getInstance();

//open the record store that stores appointments

calendarDB = new CalendarDB();

//retrieve synchoption
```

```
so = new SynchOption();

//retrieve display
display=Display.getDisplay(this);
}

public void startApp() throws MIDletStateChangeException {

display.setCurrent(menu);

}

/**
 * Pause the MIDlet
 */

public void pauseApp() {

}

/**
 * Called by the framework before the application is unloaded
 */

public void destroyApp(boolean unconditional) {

//close record store
calendarDB.close();

//clear everything
menu= null;

calendar=null;

display=null;

appForm = null;

appList =null;

apps=null;

soForm = null;

monthlyviewer=null;

}

public void commandAction(Command c, Displayable d) {

if(d==menu && c==List.SELECT_COMMAND) {

switch(menu.getSelectedIndex()) {
```

```
case 0: //Add appointment

//create a new appointment from

appForm = new AppointmentForm(display, menu, calendarDB);

appForm.setAppointment(new Appointment(calendar.getTime()));

display.setCurrent(appForm);

break;

case 1: //retrieve appointments

//create an appointment list

appList = new List("Appointments",List.IMPLICIT);

appList.addCommand(backCommand);

appList.setCommandListener(this);

//retrieve all the appointments

apps= calendarDB.retrieveAll();

for(int i=0; i<apps.size(); i++) {

Appointment app= (Appointment) apps.elementAt(i);

StringBuffer sb = new StringBuffer();

sb.append(app.getTimeString()).append(" ").append(app.getSubject());

appList.append(sb.toString(),null);

}

display.setCurrent(appList);

break;

case 2: //synchronization set up

if(soForm==null) {

//synchsetting

soForm = new SynchOptionForm(display,menu,so);

}

display.setCurrent(soForm);

break;

case 3: // monthly schedule view

//retrieve all the appointments

apps= calendarDB.retrieveAll();
```

```
app_table= new Hashtable();

for(int i=0; i<apps.size(); i++) {

Appointment app= (Appointment) apps.elementAt(i);

String key=app.getTimeString();

key=key.substring(0, key.indexOf(' '));

app_table.put(key, new Object());

}

monthlyviewer= new MonthlyScheduleViewer(calendar, app_table);

monthlyviewer.addCommand(detailCommand);

monthlyviewer.addCommand(backCommand);

monthlyviewer.setCommandListener(this);

display.setCurrent(monthlyviewer);

break;

default:

}

}

else if(d==menu && c==exitCommand ) {

destroyApp(true);

notifyDestroyed();

}

else if(d==appList) {

if(c==List.SELECT_COMMAND) {

//create a new appointment from

appForm = new AppointmentForm(display,menu,calendarDB);

appForm.setAppointment(

(Appointment)apps.elementAt(appList.getSelectedIndex()));

display.setCurrent(appForm);

}

else if(c==backCommand) {

display.setCurrent(menu);

}

}
```

```

}

else if(d==monthlyviewer) {

if(c==backCommand) {

display.setCurrent(menu);

}

else {//detail command

//create an appointment list

appList = new List("Appointments",List.IMPLICIT);

appList.addCommand(backCommand);

appList.setCommandListener(this);

//retrieve all the appointments

apps= calendarDB.retrieveAllByDate(calendar);

for(int i=0; i<apps.size(); i++) {

Appointment app= (Appointment) apps.elementAt(i);

StringBuffer sb = new StringBuffer();

sb.append(app.getTimeString()).append(" ").

append(app.getSubject());

appList.append(sb.toString(),null);

}

display.setCurrent(appList);

}

}

}

}

```

We've already discussed the functions that add new appointments and retrieve all appointments. You also need to change (or edit) and cancel (or delete) appointments. These functions are added to AppointmentForm.java (Listing 8.10).

Listing 8.10 AppointmentForm.java

```

import javax.microedition.midlet.*;

import javax.microedition.lcdui.*;

import java.util.Date;

```

```
import java.util.Calendar;

public class AppointmentForm extends Form implements CommandListener{

private Display display;

private Displayable parent;

private Command saveCommand = new Command("Save", Command.OK, 2);

private Command deleteCommand = new Command("Delete", Command.OK, 2);

private Command cancelCommand = new Command("Cancel", Command.CANCEL, 1);

private Appointment app;

private CalendarDB calendarDB;

public AppointmentForm(Display d, Displayable p, CalendarDB calDB) {

super("Appointment");

display=d;

parent=p;

calendarDB=calDB;

addCommand(cancelCommand);

addCommand(saveCommand);

addCommand(deleteCommand);

setCommandListener(this);

//Appointment Time

DateField df= new DateField("Date and Time",DateField.DATE_TIME);

df.setDate(new Date(System.currentTimeMillis()));

append(df);

//Appointment Length

append(new TextField("Length (Min)", "30", 10,TextField.NUMERIC));

//Appointment location

append(new TextField("Location", "", 50,TextField.ANY));

//Subject

append(new TextField("Subject", "", 50,TextField.ANY));

}

public void setAppointment(Appointment _app) {

app= _app;
```

```
//Appointment Time
DateField df= (DateField) get(0);
df.setDate(new Date(app.getTime()));

//Appointment Length
TextField tf_length= (TextField) get(1);
tf_length.setString(String.valueOf(app.getLength()));

//Appointment location
TextField tf_location= (TextField) get(2);
tf_location.setString(app.getLocation());

//Appointment subject
TextField tf_subject= (TextField) get(3);
tf_subject.setString(app.getSubject());
}

public void commandAction(Command c, Displayable d) {

if(c==saveCommand) {

//time
app.setTime(((DateField)get(0)).getDate().getTime());

TextField tf;

//length
tf= (TextField) get(1);
app.setLength(Integer.parseInt(tf.getString()));

//location
tf= (TextField) get(2);
app.setLocation(tf.getString());

//subject
tf= (TextField) get(3);
app.setSubject(tf.getString());

//create an alert
Alert saveInfo= new Alert("Save Appointment","",null,AlertType.INFO);
saveInfo.setTimeout(Alert.FOREVER);

if(calendarDB.save(app)) {
```

```

saveInfo.setString("Success!");
}
else {
saveInfo.setString("Fail!");
}
display.setCurrent(saveInfo,parent);
}
else if(c==deleteCommand) {
//create an alert
Alert deleteInfo= new Alert("Delete Appointment","",null,AlertType.INFO);
deleteInfo.setTimeout(Alert.FOREVER);
if(calendarDB.delete(app)) {
deleteInfo.setString("Success!");
}
else {
deleteInfo.setString("Fail!");
}
display.setCurrent(deleteInfo,parent);
}
else if(c==cancelCommand) {
display.setCurrent(parent);
}
}
}
}

```

All methods related to the record store are provided in the CalendarDB class in Listing 8.11. It uses a calendarDB record store to save all appointment records. (You can give the file any name you like.)

Listing 8.11 CalendarDB.java

```

/*
 * CalendarDB.java
 *

```

```
*/  
  
import java.io.*;  
  
import java.util.*;  
  
import javax.microedition.rms.*;  
  
public class CalendarDB {  
  
    RecordStore rs=null;  
  
    public CalendarDB () {  
  
        //the file to store the db is "calendarDB"  
  
        String file="calendarDB";  
  
        try {  
  
            // open a record store named file  
  
            rs = RecordStore.openRecordStore(file,true);  
  
        }catch(Exception e) {  
  
            System.out.println("Error: "+e.getMessage());  
  
        }  
  
        }  
  
        //close the record store  
  
        public void close() {  
  
            if(rs!=null) {  
  
                try {  
  
                    rs.closeRecordStore();  
  
                }catch (Exception e){}  
  
            }  
  
        }  
  
        //delete a record  
  
        public boolean delete(Appointment app) {  
  
            boolean success=false;  
  
            int id=app.getId();  
  
            if(id==0) return false;  
  
            try {  
  
                rs.deleteRecord(id);  
  

```

```
    success=true;

    }catch(Exception e) {}

    return success;

}

public Appointment getAppointmentById(int id) {

    Appointment app=null;

    try {

        byte data[]=rs.getRecord(id);

        app= new Appointment (id, data);

    }catch(Exception e) {}

    return app;

}

//retrieve all appointments

public Vector retrieveAll() {

    RecordEnumeration re=null;

    Vector apps= new Vector();

    try {

        //cutoff is 90 days old

        long cutoff=System.currentTimeMillis()-new Integer(90).longValue()*24*60*60000;

        RecordFilter rf = new AppointmentFilter(cutoff);

        RecordComparator rc = new AppointmentComparator();

        re = rs.enumerateRecords(rf,rc,false);

        while(re.hasNextElement()) {

            int rec_id=re.nextRecordId();

            apps.addElement(new Appointment(rec_id,rs.getRecord(rec_id)));

        }

    }catch(Exception e) {}

    finally{

        //destroy the enumerator

        if(re!=null) re.destroy();

    }

}
```

```
return apps;

}

//retrieve all appointments on a single date

public Vector retrieveAllByDate(Calendar calendar) {

RecordEnumeration re=null;

Vector apps= new Vector();

try {

RecordFilter rf = new AppointmentDateFilter(calendar);

RecordComparator rc = new AppointmentComparator();

re = rs.enumerateRecords(rf,rc,false);

while(re.hasNextElement()) {

int rec_id=re.nextRecordId();

apps.addElement(new Appointment(rec_id,rs.getRecord(rec_id)));

}

}catch(Exception e) {}

finally{

//destroy the enumerator

if(re!=null) re.destroy();

}

return apps;

}

//save an appointment (new and old)

public boolean save(Appointment app) {

if(rs==null) return false;

boolean success=false;

try {

byte[] data= app.toBytes();

int id= app.getId();

if(id==0) { //create a new record

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

app.setId(id);
```

```

}

else {//update the old record

rs.setRecord(id, data, 0, data.length);

}

success=true;

}catch(Exception e){

System.out.println("Error: "+e.getMessage());

}

return success;

}

}

```

A method to pack appointment fields into a byte array using UTF-8 encoding is added to the Appointment class. A method to construct an appointment from a UTF-8 encoded byte array is added as well. The updated Appointment.java is shown in Listing 8.12.

Listing 8.12 Appointment.java

```

/*

* Appointment.java

*

*/

import java.util.Calendar;

import java.util.Date;

import java.util.Vector;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import javax.microedition.rms.*;

-

public class Appointment {

private int id;

private long time;

```

```
private int length;

private String location;

private String subject;

/* default constructor */

public Appointment () {

    id=0;

    time=0;

    length=0;

    location="";

    subject="";

}

public Appointment(int id,long time, int length, String location, String subject) {

    this();

    id= id;

    time= time;

    length= length;

    location= location;

    subject= subject;

}

public Appointment(Date date) {

    this();

    time=date.getTime();

    length=30;

}

/* rec is the byte array saved in record store */

public Appointment (int id, byte[] rec) {

    id= id;

    init_app(rec);

}

/* rec is the byte array saved in record store */

public void init_app(byte[] rec) {
```

```
// parse the record

ByteArrayInputStream bais= new ByteArrayInputStream(rec);

DataInputStream dis= new DataInputStream(bais);

try {

time=dis.readLong();

length=dis.readInt();

location=dis.readUTF();

subject=dis.readUTF();

}catch(Exception e){}

}

/* convert to the byte array that will be saved in the record store*/

public byte[] toBytes() {

byte data[]=null;

try {

ByteArrayOutputStream baos= new ByteArrayOutputStream();

DataOutputStream dos= new DataOutputStream(baos);

dos.writeLong(time);

dos.writeInt(length);

dos.writeUTF(location);

dos.writeUTF(subject);

data=baos.toByteArray();

baos.close();

dos.close();

}catch(Exception e) {}

return data;

}

/* get the appointment time in display format */

public String getTimeString() {

StringBuffer sb= new StringBuffer();

Calendar cal= Calendar.getInstance();

cal.setTime(new Date(time));
```

```

sb.append(cal.get(Calendar.MONTH)+1).append("/");
sb.append(cal.get(Calendar.DAY_OF_MONTH)).append("/");
sb.append(cal.get(Calendar.YEAR)).append(" ");
sb.append(cal.get(Calendar.HOUR_OF_DAY)).append(":");
if(cal.get(Calendar.MINUTE)<10) sb.append(0);
sb.append(cal.get(Calendar.MINUTE));

return sb.toString();
}

public int getId() {return id;}

public void setId(int id) { id= id;}

public long getTime() {return time;}

public void setTime(long time){time= time;}

public int getLength(){return length;}

public void setLength(int length) {length= length;}

public String getLocation() {return location;}

public void setLocation(String location){location= location;}

public String getSubject() {return subject;}

public void setSubject(String subject) {subject= subject;}

}

```

The `SynchOption` setting is saved to a record store with the name `synchOption`. Methods to save a setting to the record store and to retrieve a setting from the record store are added to the updated `SynchOption.java` (see Listing 8.13).

Listing 8.13 `SynchOption.java`

```

import java.io.*;

import javax.microedition.rms.*;

public class SynchOption {

private String url;

private String user;

private String passwd;

private boolean autoSynch;

private int frequency;

```

```
private String file="synchOption";

public SynchOption(){

//default value

url="";

user="";

passwd="";

autoSynch=false;

frequency=0;

RecordStore rs=null;

try {

// open a record store named file

rs = RecordStore.openRecordStore(file,true);

if(rs.getNumRecords(>0) {

byte data[]=rs.getRecord(1);

// parse the record

ByteArrayInputStream bais= new ByteArrayInputStream(data);

DataInputStream dis= new DataInputStream(bais);

try {

url=dis.readUTF();

user=dis.readUTF();

passwd=dis.readUTF();

autoSynch=dis.readBoolean();

frequency=dis.readInt();

}catch(Exception e) {}

dis.close();

}

}catch(Exception e) {

System.out.println("Error: "+e.getMessage());

}finally {

if(rs!=null) {

try {
```

```
rs.closeRecordStore();

}catch(Exception e) {}

}

}

}

public boolean save() {

//convert to byte array

byte data[]=null;

try {

ByteArrayOutputStream baos= new ByteArrayOutputStream();

DataOutputStream dos= new DataOutputStream(baos);

dos.writeUTF(url);

dos.writeUTF(user);

dos.writeUTF(passwd);

dos.writeBoolean(autoSynch);

dos.writeInt(frequency);

data=baos.toByteArray();

//close

baos.close();

dos.close();

}catch(Exception e) {}

boolean success=false;

//save data to file

RecordStore rs=null;

try {

// open a record store named file

rs = RecordStore.openRecordStore(file,true);

if(rs.getNumRecords(>0) {

rs.setRecord(1, data, 0, data.length);

}

else {
```

```
rs.addRecord(data,0,data.length);

}

success=true;

}catch(Exception e) {

System.out.println("Error: "+e.getMessage());

}finally {

if(rs!=null) {

try{

rs.closeRecordStore();

}catch(Exception e) {}

}

}

return success;

}

public String getUrl() {

return url;

}

public String getUser() {

return user;

}

public String getPasswd() {

return passwd;

}

public boolean getAutoSynch() {

return autoSynch;

}

public int getFrequency() {

return frequency;

}

public void setUrl(String url) {

this.url=url;
```

```

}

public void setUser(String user) {

    this.user=user;

}

public void setPasswd(String passwd) {

    this.passwd=passwd;

}

public void setAutoSynch(boolean autoSynch) {

    this.autoSynch=autoSynch;

}

public void setFrequency(int frequency) {

    this.frequency=frequency;

}

}

```

In MonthlyScheduleView.java, developed in Chapter 7, a day that has appointments is highlighted in the viewer. From the viewer, you know a day has appointments; now, you want to know how many appointments are on that day and what they are. In Scheduler.java, a Detail command is added to the MonthlyScheduleViewer object. If you click the Detail command, all appointments on the selected day will be retrieved and listed. The appointment filter that selects appointments on a single day is shown in Listing 8.14.

Listing 8.14 AppointmentDateFilter.java

```

/*
 * AppointmentDateFilter.java
 * Only the appointment that has the same date will match.
 */

import java.util.*;

import javax.microedition.rms.*;

public class AppointmentDateFilter implements RecordFilter{

    private long cutoff0;

    private long cutoff1;

    public AppointmentDateFilter (Calendar calendar) {

        long dayInMillis=24*60*60*1000;

```

```
    cutoff0=calendar.getTime().getTime()/dayInMillis*dayInMillis;

    cutoff1=cutoff0+dayInMillis;

}

public boolean matches(byte[] candidate) {

    Appointment app= new Appointment();

    app.init_app(candidate);

    if (app.getTime()>=cutoff0 && app.getTime()<cutoff1) {

        return true;

    }

    else {

        return false;

    }

}

}
```

SynchOptionForm.java from Chapter 6 and MonthlyScheduleView.java from Chapter 7 are used in this application. We haven't modified them; thus, they are not listed here, but are included on this book's Web site.

Now you have a complete local application that can store, retrieve, and edit your appointments. All the appointments can be summarized in a list or be graphically presented in monthly calendar view. Suppose you have appointments as shown in Figure 8.3. If you go to the monthly schedule viewer, you will see a display like Figure 8.6. You can change the selected date to March 10, 2001 using the arrow keys; then, choose the Detail command. You will see the appointments on that day, as shown in Figure 8.7. From the list, you can edit or delete selected appointments.

To make the application more useful, it should have the ability to synchronize with your appointments stored on a networked data server. In later chapters, we will discuss network connections and XML. In Chapter 12, "Data Synchronization for Wireless Applications," this application will be converted to a network application with the addition of synchronization functions.

Summary

MIDP defines a record-oriented database, RMS, for storing persistent data. A set of APIs for managing the database is provided in the package javax.microedition.rms. This package defines a RecordStore class for representing a record store. Each record store is a file that functions similarly to a table in a

database. Each record in a record store contains an integer recordId as its primary key and a byte array that holds data.

The RecordStore class provides methods for adding, modifying, and deleting records. The RecordEnumeration class provides methods to enumerate all records. Even though a record in a record store has only one data field, it can be used to store a record with multiple columns by using UTF-8 coding. In the javax.microedition.rms package, two interfaces (RecordComparator and RecordFilter) are provided for selecting and sorting records. Another interface, RecordListener, is provided for monitoring record changes in a record store.