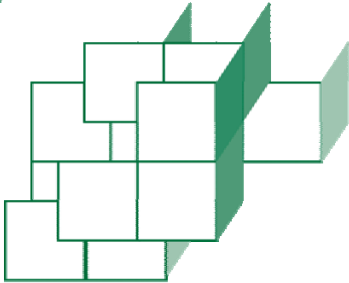
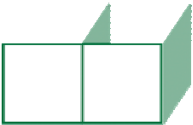




CHAPTER 7
THE RECORD MANAGEMENT
SYSTEM



OBJECTIVES

After completing “The Record Management System,” you will be able to:

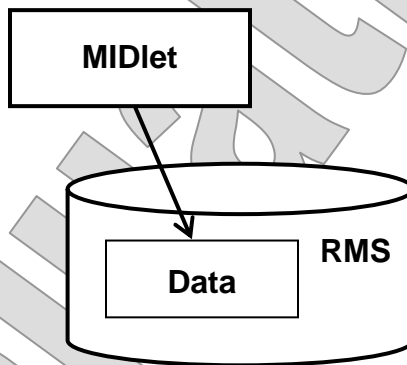
- Explain the unique challenges involved in persistent storage for wireless applications, and how the MIDP Record Management System is designed to support minimal persistent storage for applications.
- Create, open and delete RMS record stores.
- Add, change and remove individual records.
- Enumerate records in a record store:
 - In total, ordered by time of creation
 - Through an application-defined filter
 - In an application-defined sort order
- Check version and timestamp information.
- Manage object persistence to an RMS record.
- Implement persistence for a MIDP application using multiple record stores and records.
- Link an RMS store into the Model in MIDP Model/View-oriented application design.

The Challenge of Persistence

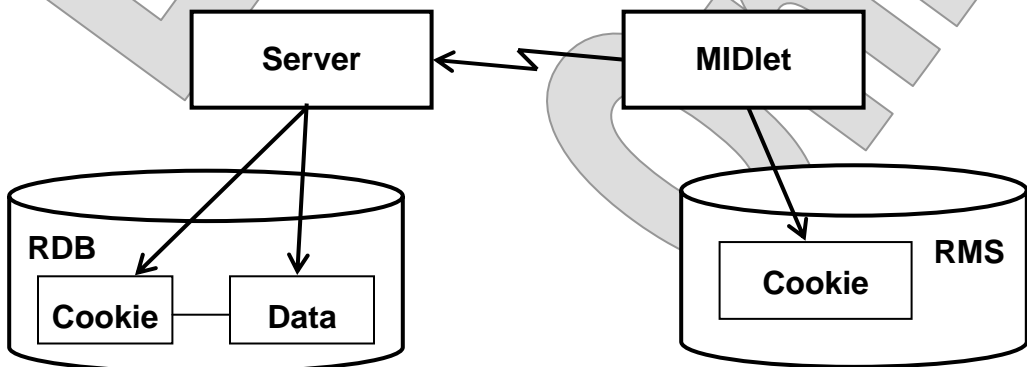
- Many applications have a need for persistent data.
- This is quite a challenge to support on tiny wireless devices; at the physical level, storage options are few:
 - There are typically no fixed or removable storage media such as hard drives, floppy disks, and CD-ROM drives.
 - “Persistent memory,” as defined in the MIDP specification, must be available on a target device, but only in a minimum quantity of 8 kilobytes!
- At the logical level, this means that storage options so common in PC/server software are unavailable:
 - No file system
 - No relational or OO databases

Common Scenarios

- With persistent storage space in such short supply, software for wireless devices must economize.
- This means different things for different application architectures:
 - Standalone applications may save some session state, or keep a document of minimal size on the device for later uplink.



- Networked applications are much more likely to store everything they possibly can on a server somewhere.
- Thus the only persistent data they will need will be a key to the unique location of their data on the network: a **cookie**, most likely, containing host name or ID and data store ID.



The Record Management System

- MIDP defines a system for persistent data that can be managed by applications, called the **record management system**, or **RMS**.
- As with many aspects of MIDP, the RMS system is largely **opaque to the application**:
 - The actual location and format of persistent data is managed by the device's MIDP implementation.
 - These data stores are not directly accessible by MIDP software; they can only be found through the RMS API.
 - The data stores are not portable and can't be pre-deployed with MIDlet suites.
- The RMS system is, no surprise, **optimized for space**.
 - It is based in records which are exposed as **arrays of bytes**.
 - These records have sequential IDs and can be **randomly accessed** based on these IDs.

The RecordStore Class

- The only class in the **javax.microedition.rms** package is **RecordStore**. Following is a listing of its public interface, complete except for exception signatures:

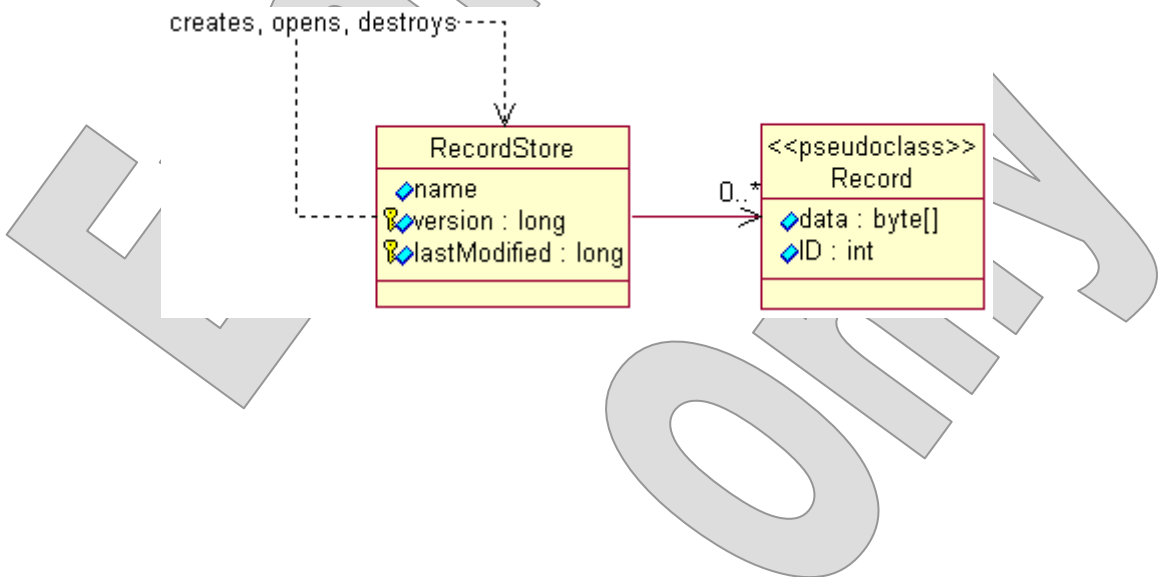
```
public class RecordStore
{
    public static RecordStore openRecordStore
        (String name, boolean create);
    public static String[] listRecordStores ();
    public static void deleteRecordStore
        (String name);
    public int addRecord
        (byte[] data, int offset, int length);
    public void deleteRecord (int ID);
    public void closeRecordStore ();
    public int getRecordSize (int ID);
    public int getRecord
        (int ID, byte[] buffer, int offset);
    public byte[] getRecord (int ID);
    public void setRecord
        (int ID, byte[] data, int offset, int length);
    public String getName ();
    public int getVersion ();
    public int getNumRecords ();
    public int getSize ();
    public int getSizeAvailable ();
    public long getLastModified ();
    public void addRecordListener
        (RecordListener listener);
    public void removeRecordListener
        (RecordListener listener);
    public int getNextRecordID ();
    public RecordEnumeration enumerateRecords
        (RecordFilter filter, RecordComparator compare,
         boolean update);
}
```

A Single Class

- Note that **RecordStore** offers three static methods, shaded in the above listing.
 - These are used to create, open and remove persistent record stores from the system.

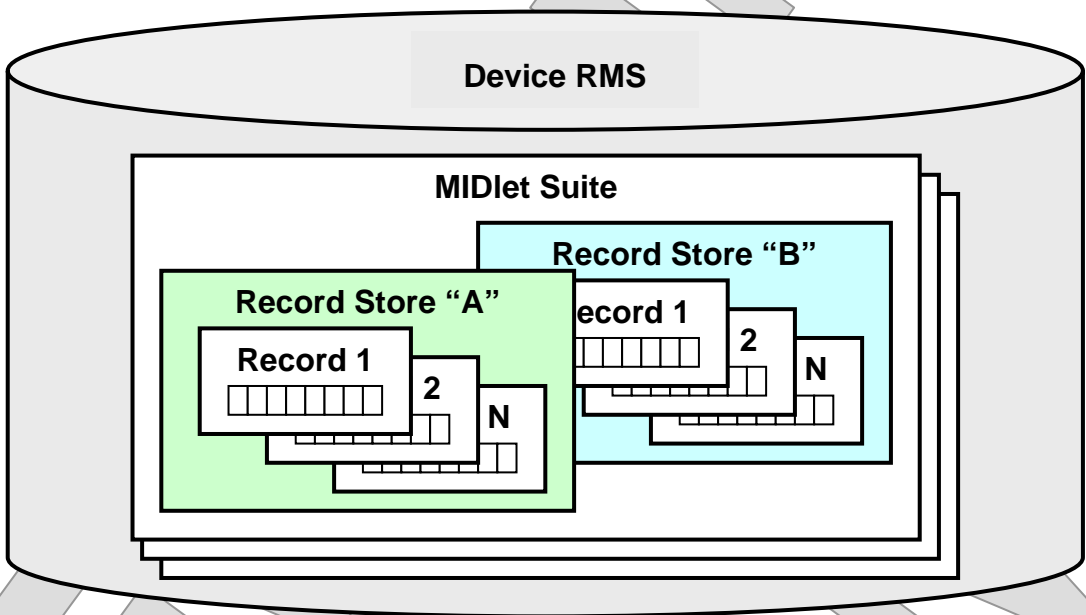
```
RecordStore myRecords =  
    RecordStore.openRecordStore ("MyRecords", true);
```

- In essence **RecordStore** is acting as its own **factory**.
- We'll see in a moment that records are represented as primitives: a byte array, an ID, a version number, etc.
- So where there might have been three classes in the RMS package – call it an **RMS** class utility to manage the system and to dole out **RecordStores**, and a **Record** class – there is instead just one:



Scope of Record Storage

- Record stores are associated with an installed MIDlet suite.
 - The MIDlet suite’s identity on the device serves as a partition of sorts in the device’s persistent memory.
 - A record store is given a name of up to 32 Unicode characters, which must be unique within the MIDlet suite (comparisons are case-sensitive).



- Record stores can be shared within the MIDlet suite.
 - That is, any MIDlet in the suite can open or remove a store, not just the MIDlet that created it.
 - There is **no locking API** built into RMS, no support for safe concurrent access.
 - So while concurrent shared access is possible, it must be managed carefully at the application level.

Managing Record Stores

- Any class in a MIDlet can open or create a record store.
 - The method call is the same in both cases – **open** – and the second parameter defines whether the call should create the record store if it doesn't already exist.

```
RecordStore myRecords =  
    RecordStore.openRecordStore ("MyRecords", true);
```

- If this parameter is **false** and the record doesn't yet exist, the method will throw a **RecordStoreNotFoundException** exception.
 - If the parameter is **true** and the record already exists, it will simply be opened. It will not be overwritten with a new, empty record store – as one might expect from a lifetime of working with files!
 - The result of this call is an instance of **RecordStore**, which can then be manipulated record-by-record.
 - When done working with a record store, one must call **closeRecordStore**. A reference count of usage is kept based on this method and **openRecordStore**.
- To remove a record store, call **deleteRecordStore**:

```
deleteRecordStore ("MyRecords");
```

- This will fail with a **RecordStoreException** if the record is currently open anywhere in the MIDlet suite.
- It also fails with a **RecordStoreNotFoundException** if the named record store does not exist.

Defining a Record

- To create a record within a store, call **addRecord**.

- Provide the data for the record as a byte array.

```
byte[] data = someString.getBytes ();  
myRecords.addRecord (data, 0, data.length);
```

- You can store a segment of a larger array using **offset** and **length** parameters. These must be provided, so the usage above is common.

- The result is the unique record ID, an integer.

- If the data for a record is not yet formed, the record can be created empty and then populated using **setRecord**:

```
int ID = myRecords.addRecord (null, 0, 0);  
byte[] data = getDataFromSomewhereElse ();  
myRecords.setRecord (ID, data, 0, data.length);
```

- Typically it is no easier to do this, and it is marginally less efficient.

- The **setRecord** method is more useful when modifying existing records, probably from an earlier MIDlet session.

Reading Record Data

- Read the data from a record using one of the overloads of **getRecord**:

- The simpler version returns the whole record as a byte array:

```
byte[] data = myRecords.getRecord (2);
```

- The second version will write the record's bytes into an existing buffer – it's the application's job to make sure the buffer is big enough.

```
int size = 0;
for (int r = 1; r <= 3; ++r)
    size += myRecords.getRecordSize (r);
```

```
byte[] data = new byte[size];
int offset = 0;
for (int r = 1; r <= 3; ++r)
{
    myRecords.getRecord (r, data, offset);
    offset += myRecords.getRecordSize (r);
}
```

Record IDs

- As we've seen, a record store is accessed by name.
- Individual records in the store are given numeric IDs.
 - These IDs are assigned by the RMS on creation.
 - They are generated sequentially, the first record ID being 1.
 - For efficiency's sake, when records are deleted, IDs following the deleted record are not decremented.
 - Therefore the record ID does not behave quite like an array or **Vector** index, since the maximum record ID in a record store may be greater than the number of records:

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

7 Records Total

- Ordering of records as created, however, is strictly preserved.

Deleting Records

- Delete a record by calling **deleteRecord**:

```
myRecords.deleteRecord (2);
```

- It is safe to work from the front of a sequence of records and delete as you go, so long as you know all record IDs to be present.

```
for (int x = 1; x <= 3; ++x)  
  myRecords.deleteRecord (x);
```

- This algorithm is notoriously unsafe when working with dynamic collections such as **Vector**, because one advances “double-time”, with the loop index increasing and the records shuffling in the opposite direction.

Record Enumeration

- When reading a record store, typically the sequence of record IDs is not known – unless the application never deletes individual records.
- One could work from 1 to **getNextRecordID**, catching exceptions along the way, but this would be clumsy.
- Better to use the handy enumeration feature!
 - Call **enumerateRecords** as follows:

```
RecordEnumeration enum =  
    myRecords.enumerateRecords (null, null, false);
```

- This returns a **RecordEnumeration**, which can be used to work through the existing records in order of creation.
- There are other, fancier ways to derive enumerations, and we'll come back to those later.

The RecordEnumeration Interface

```
public interface RecordEnumeration
{
    public boolean hasNextElement ();
    public byte[] nextRecord ();
    public int nextRecordId ();

    public boolean hasPreviousElement ();
    public byte[] previousRecord ();
    public int previousRecordId ();

    public int numRecords ();
    public void reset ();
    public void rebuild ();
    public void keepUpdated (boolean);
    public boolean isKeptUpdated ();
    public void destroy ();
}
```

- **RecordEnumeration** provides a bi-directional iterator over a record store.
- Common usage is as follows:

```
RecordEnumeration enum =
    myRecords.enumerateRecords (null, null, false);
while (enum.hasNextElement ())
{
    byte[] data = enum.nextRecord ();
    // Processing here
}
```

- The **destroy** call is important. Until it is called, the enumerator must hold a number of resources since it might at any time get another call to iterate. As with all MIDP code, it pays to be aggressive about memory/resource cleanup.

Version and Time Stamp

- Every record store has a built-in version number.
 - This is managed by the RMS very simply: it starts at zero and is incremented on every change to the record store.
 - Thus each call to **addRecord**, **setRecord**, or **deleteRecord** bumps the version number by one.
 - Although crude, this method does provide a useful indicator of actual version, at least by comparison to a previously-stored value.
- More concretely, there is also a time stamp, which is updated on every mutation as well.
 - This is kept as a number of milliseconds, and is in sync with **System.getTimeMillis** and the raw values used by **Date** and **Calendar** classes.

- In **Examples/Versions**, a MIDlet creates a single record in a new record store, makes a few changes and displays the version count before closing:

```
RecordStore store = RecordStore.openRecordStore  
("Versions", true);
```

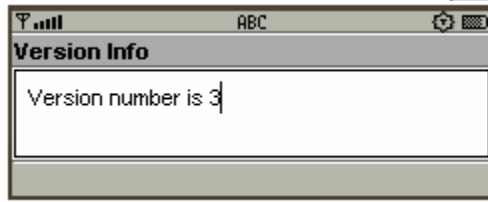
```
byte[] data = "Some data".getBytes ();  
store.addRecord (data, 0, data.length);
```

```
byte[] data2 = { 0, 1, 2, 3 };  
int doomed = store.addRecord  
 (data2, 0, data2.length);  
store.deleteRecord (doomed);
```

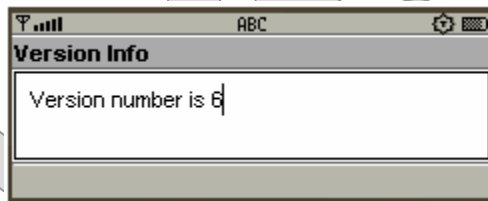
```
results.append ("Version number is ")  
 .append (store.getVersion ());
```

Checking Versions

- Build and test this to see that the version number reflects three operations: add, add, delete.



- Run it again and note that the version number is now six – what happened here?



- The **openRecordStore** call doesn't create a fresh record store this time, and so the old store is re-used, and its version is bumped from 3 to 6.

Byte Arrays and Data Streams

- The sole interface to record data is the byte array.
- This is not the most facile structure to work with, and most often record data will be manipulated by converting the bytes to and from some other code-level representation.
 - Note the use of **String.getBytes** in the previous demo.
 - **Image** objects can be converted to and from byte arrays.
- The most generally useful conversion, however, is between byte array and the **DataXXXStream** classes for binary input and output of primitives and strings.
 - The **ByteArrayXXXStream** classes are the bridge between the two.
 - A data stream can be used to write a sequence of various data in a compact format, and then the byte-array stream can provide its underlying buffer as a byte array.
 - Or, the byte array can be populated from an RMS record, opened as a data stream, and read sequentially as integer, string, long, character, etc.

- In **Examples/Datebook/Step7**, we'll take a first look at the persistence architecture of the Datebook MIDlet.
- The **Item** class implements its own persistence logic:
 - A constructor takes a byte array and reads it as a data stream:

```
public Item (byte[] record)
{
    DataInputStream in = new DataInputStream
        (new ByteArrayInputStream (record));
    try
    {
        when.setTime (in.readLong ());
        what = in.readUTF ();
        tag = in.readUTF ();
        in.close ();
    }
    catch (IOException ex)
    {
        what = "Error reading from RMS record.";
        System.out.println (what);
    }
}
```

- This is used by the **Datebook** constructor to build the **Vector** of **Items** from an enumeration of RMS records:

```
while (each.hasNextElement ())
    insertItem (new Item (each.nextRecord ()),
        eachID.nextRecordId ());
```

- A **save** method writes via a data stream into a byte-array buffer, and then provides that to the caller:

```
public byte[] save ()
    throws IOException
{
    ByteArrayOutputStream buffer = new
ByteArrayOutputStream ();
    DataOutputStream out = new DataOutputStream
(buffer);
    out.writeLong (when.getTime ());
    out.writeUTF (what);
    out.writeUTF (tag);
    out.close ();

    return buffer.toByteArray ();
}
```

- Various mutator methods in **Datebook** rely on this method to provide the raw data for a new or existing record:

```
// In addItem ():
byte[] data = item.save ();
return insertItem (item, store.addRecord (data, 0,
data.length));

// In updateItem ():
byte[] data = incarnation.item.save ();
store.setRecord (incarnation.recordID, data, 0,
data.length);
```

- In **Demos/RMS**, we'll add some persistence code to the **CarRental** application.
 - As it exists now, the application requires the user to enter all information each time it is run.
 - Some of this is personal information that will not change very frequently, if at all: name, e-mail address, credit card number, etc.
 - It would be nice to store this on the user's behalf and save a lot of tedious data entry.
 - The completed demo is in **Examples/CarRental/Step3**.
- 1. Open the source file **src/cc/travel/CarRentalMIDlet.java** and find the **save** method. Note that this is already being called on application close, and that so far it simply cleans up a couple of record stores.
- 2. First, change the type of exception being caught in the outer system from **Exception** to **RecordStoreException**.

```
private void save ()
{
    try { ... }
    catch (RecordStoreException ex)
    {
        System.out.println
            ("Failed to save all persistent fields.");
        ex.printStackTrace ();
    }
}
```

- The code you're about to write can throw this type of exception, but previously catching this type would have generated a compile error, since it currently couldn't be thrown.

3. After the code to delete old records, add code to save the personal information, which is encapsulated in a persistent class **PersonalInfo**. (You may want to review the code in this class; it is very much like what we just saw in the Datebook **Item**.)

```
try
{
    ...
    try
    {
        RecordStore.deleteRecordStore
            ("CarPreference");
    }
    catch (RecordStoreNotFoundException ex) {}

    RecordStore storage = RecordStore.openRecordStore
        ("PersonalInfo", true);
    model.personalInfo.save (storage);
    storage.closeRecordStore ();
}
catch (RecordStoreException ex)
```

4. We'll also save the user's latest choice of rental-car class. It's not exactly personal info, and other MIDlets that might use the former probably can't use the latter, so we'll put this in its own record store. Open a new store and create a **DataOutputStream** backed by a byte array:

```
storage.closeRecordStore ();

storage = RecordStore.openRecordStore
    ("CarPreference", true);
ByteArrayOutputStream buffer =
    new ByteArrayOutputStream ();
DataOutputStream out =
    new DataOutputStream (buffer);
}
catch (RecordStoreException ex)
```

5. **try** writing the string value into the stream, and closing the stream, against the **IOException**:

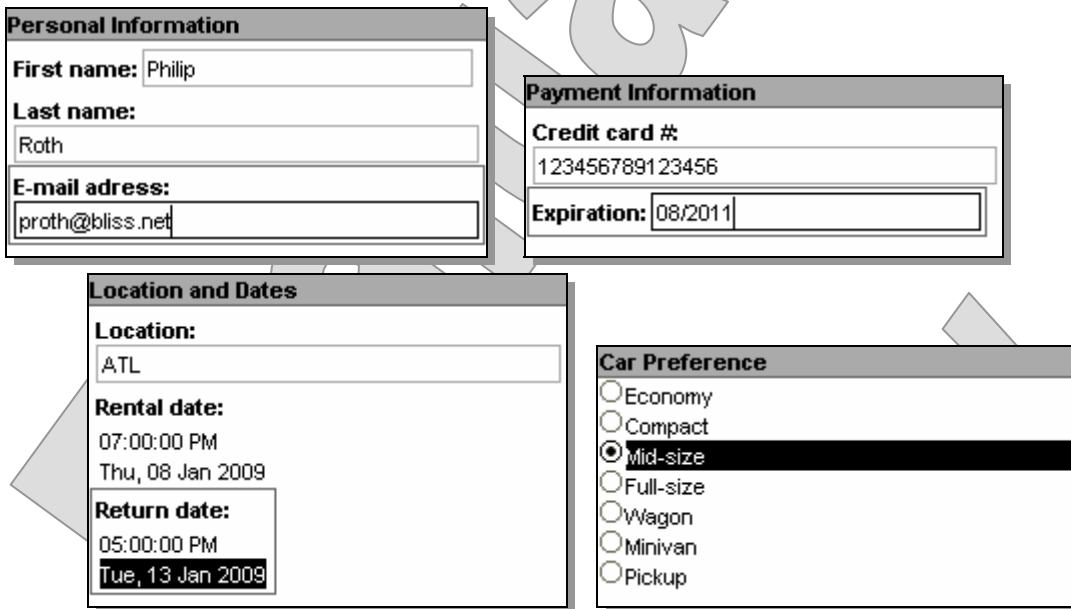
```
DataOutputStream out =
    new DataOutputStream (buffer);
try
{
    out.writeUTF (model.carClass);
    out.close ();
}
catch (IOException ex) {}
}
catch (RecordStoreException ex)
```

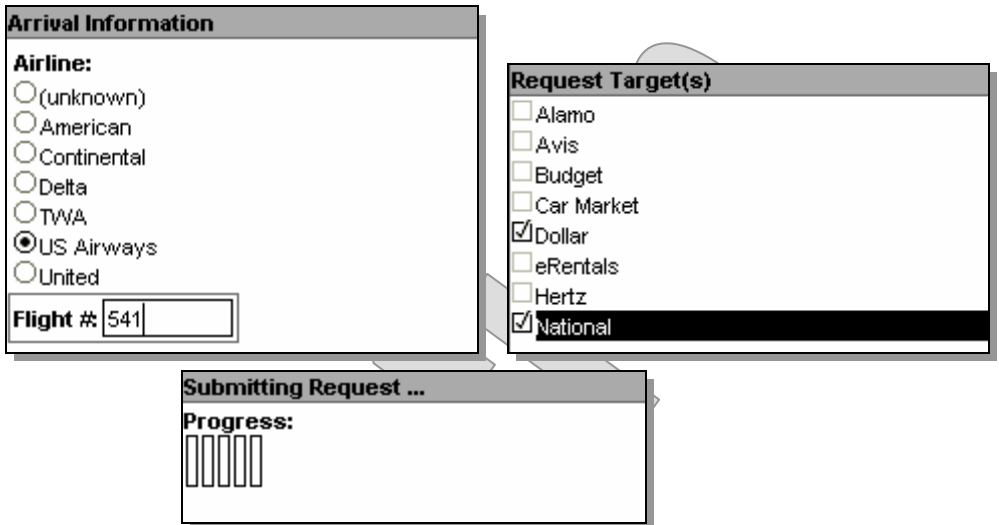
6. Now add a record and close the record store:

```
catch (IOException ex) {}

storage.addRecord
    (buffer.toByteArray (), 0, buffer.size ());
storage.closeRecordStore ();
}
catch (RecordStoreException ex)
```

7. Build the project and test the MIDlet. Go through the whole process, to the last screen, and then close.





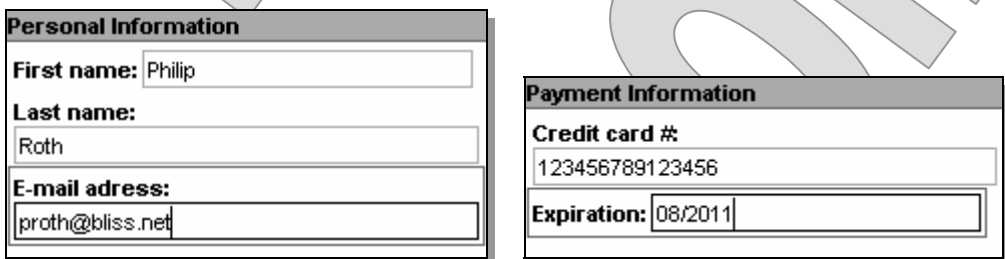
8. If you've been wondering where the emulator stores data written to its RMS implementation, take a look at the console output from your latest run:

run

```
Running with storage root C:\Documents and Settings\Your Name\j2mewtk\2.5.2\appdb\DB
Running with locale: ...
```

9. Look in the **appdb/DB** directory as indicated above. You should see a file called **run_by_class_storage_#Personal#Info.db**, and a similar one for the "Car Preference" store.

10. Test again, and you should find that the information is restored:



etc.

11. In the MIDlet source file, find the **load** method review the implementation already in place:

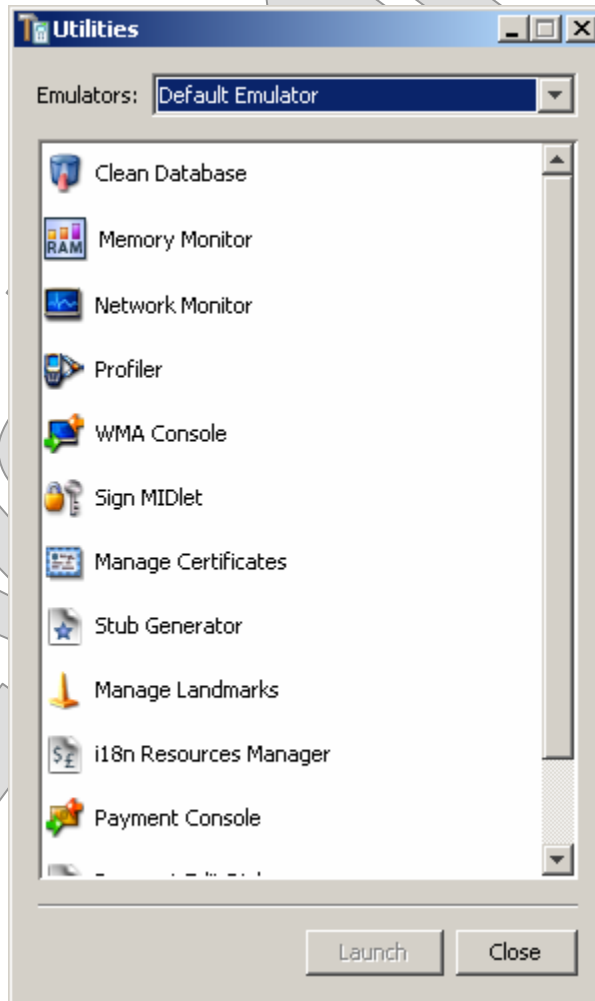
```
private void load ()
{
    try
    {
        RecordStore storage =
            RecordStore.openRecordStore
                ("PersonalInfo", false);
        model.personalInfo.load (storage);
        storage.closeRecordStore ();

        storage = RecordStore.openRecordStore
            ("CarPreference", false);
        DataInputStream in = new DataInputStream
            (new ByteArrayInputStream
                (storage.getRecord (1)));
        try
        {
            model.carClass = in.readUTF ();
            in.close ();
        }
        catch (IOException ex) {}

        storage.closeRecordStore ();
    }
    catch (RecordStoreException ex) {}
}
```

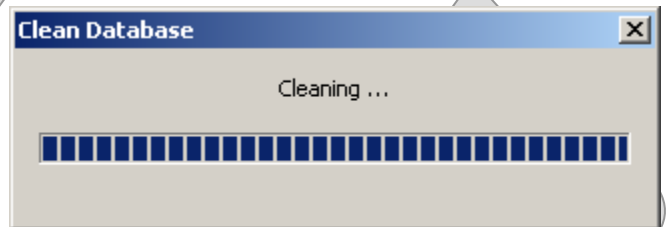
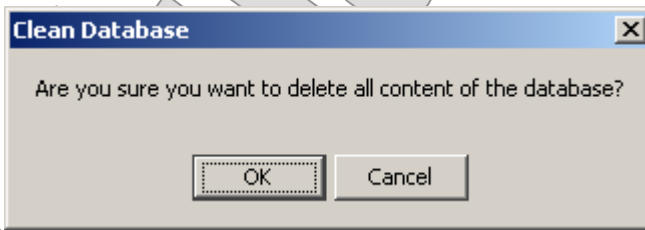
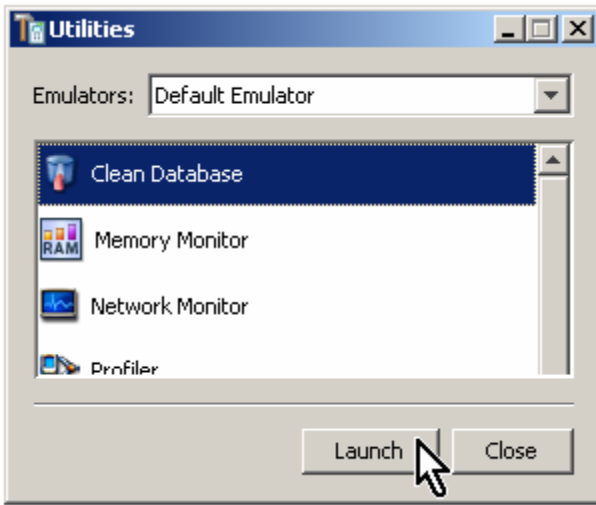
The Clean-Database Utility

- The Wireless Toolkit **utils.exe** application provides a number of useful utilities for developing MIDP code.
- One of particular help in building RMS code is a utility that wipes the RMS storage system clean.
- Run **utils.exe**:



The Clean-Database Utility

- Choose **Clean Database** and click **Launch**:



- Close the utilities application.
- This is especially helpful when developing RMS code incrementally, because a bug in storing code can write bad info that then won't work with correct loading code, even after the original bug is corrected.

Suggested time: 45 minutes

In this lab you will implement simple dump-and-load persistence for the Scribble MIDlet.

Detailed instructions are found at the end of the chapter.

Evaluation
Only

Data Compression

- The RMS is maximally space-efficient in and of itself.
- Still, it's only as efficient as the data stored in it, so to speak.
- It's a good idea to think about how compact your application data is.
 - **Binary** storage formats as used by the data-stream classes are a great start.
 - How “**normalized**” or redundant is your persistent data?
 - Are there values that could be considered **transient**, perhaps left out and restored based on other persistent values?
 - Are you “**using the whole byte**” – that is, does the information you're storing require all 8 bits, or all 4 bytes for an integer, etc.?
- This sort of analysis is especially important if your code can potentially be used multiple times to store many objects as created by the user.
- Consider the possibility of application-level compression algorithms – weigh costs in code size against benefits in storage size.

- A few possible savings for the Scribble MIDlet:
 1. Integers consume 4 bytes each, but the screen size is unlikely to be anywhere near $2^{32} \times 2^{32}$.
 - A single **short** would do for each of x and y coordinates, for even the largest mobile-device screen.
 - This would drop our segment size from 20 bytes to 12.
 2. The same color will likely be used for many segments in a row.
 - The high byte of the color encoding is unused, so if a single byte with a non-zero value were written instead, this could be decoded to mean that the segment uses the same color as the previous one.
 - This would drop the size of each such segment by 3 bytes.
 3. Many segments will be contiguous. That is, the end coordinates of segment N will be the same as the start coordinates of segment N + 1.
 - Break the drawing into M contiguous sequences, a sequence being written as N segments. Each segment then has just two coordinates, not four, and the final coordinate N + 1 is written last. Color info could be captured just once per sequence, too.
 - This would add 8 bytes per sequence (size and color) but drop segment size to 4 bytes.
- Can you think of others? Discuss.

Advanced Record Enumeration

- Although packing all persistent data into a single record by way of a data stream is efficient, there are data designs that map naturally to multiple records, each with a similar format.
- Using one record for each instance of a common class also enables the use of some primitive querying features built into RMS.
- The **enumerateRecords** method can save a good deal of application code by providing enumerations that aren't just sequential by create time over the whole record store:
 - It can **filter** records by some criterion.
 - It can **sort** records based on given comparison logic.
- It can also provide an enumerator that stays in synch with changes made after the enumerator is created.
- We'll look at each of these techniques briefly.

Filtering Enumerations

- **enumerateRecords** will apply filtering logic as provided via the **filter** parameter.

- This is of type **RecordFilter**:

```
public interface RecordFilter
{
    public boolean matches (byte[] data);
}
```

- The implementation must indicate whether or not the record content should be included in the enumeration.

```
RecordFilter myFilter = new RecordFilter ()
{
    public boolean matches (byte[] data)
    {
        return data.length >= 2 &&
            data[0] * 256 + data[1] == MAGIC_NUMBER;
    }
};
```

```
RecordEnumeration enum = myRecords.enumerateRecords
(myFilter, null, false);
```

- Note that to check any useful criteria, the **matches** implementation may have to go so far as to unpack the data in the record using a stream.
- Remember that whatever is done here will be done N times, where N is the number of records.

Sorting Enumerations

- The next parameter to `enumerateRecords` is a **RecordComparator**:

```
public interface RecordComparator
{
    public static final int EQUIVALENT;
    public static final int FOLLOWS;
    public static final int PRECEDES;
    public int compare (byte[] one, byte[] two);
}
```

- The implementation must return one of the three constants defined in the interface to indicate, “one follows two,” “one precedes two,” or “one and two are equivalent.”

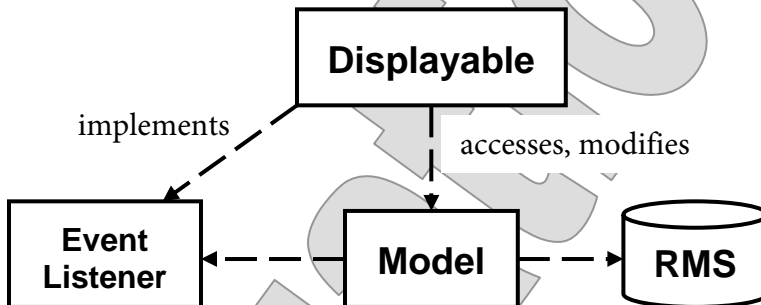
```
RecordComparator myComp = new RecordComparator ()
{
    public boolean compare (byte[] one, byte[] two)
    {
        return one.length < two.length
            ? PRECEDES : (one.length == two.length
                ? EQUIVALENT : FOLLOWS);
    }
};
```

```
RecordEnumeration enum = myRecords.enumerateRecords
    (null, myComp, false);
```

- Again, meaningful comparisons can require unpacking data.
- Here, the number of code executions will be more than proportional to the number of records: this will depend on the implementation, but at best this will scale “ $N \log N$ ”, meaning the code will be executed an average of $\log(N)$ times for each record.

The RecordListener Interface

- In the previous chapter we discussed the MVC and Model/View patterns, and the value of model events as means to notify of changes without establishing code dependencies in the model design.



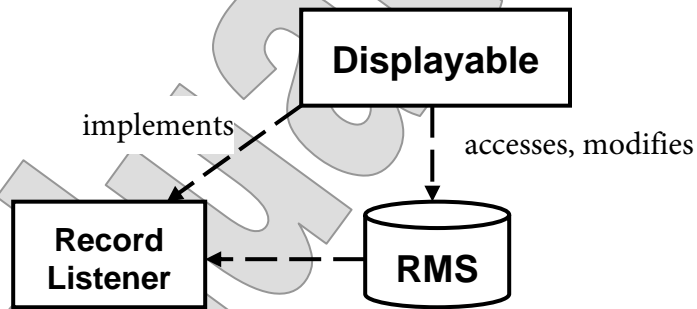
- The RMS system includes a model event by which classes can be notified of changes to a record store.
 - The listener interface is **RecordListener**:

```
public interface RecordListener
{
    public void recordAdded (RecordStore, int ID);
    public void recordChanged (RecordStore, int ID);
    public void recordDeleted (RecordStore, int ID);
}
```

- The **RecordStore** is the event source, and supports multicasting.
- There is no event type, as the listener methods all take a store reference and a record ID.

MVC Using the Record Store

- An application class can register itself as a listener and thus stay in synch with changes to the record store.
 - In this way, a record store can act as the Model for a MIDlet system: Views can register directly with the persistent store.
 - An “interpretive layer” or adapter class of some sort may be included in the system to gather the logic for reading the record store and records in one place.



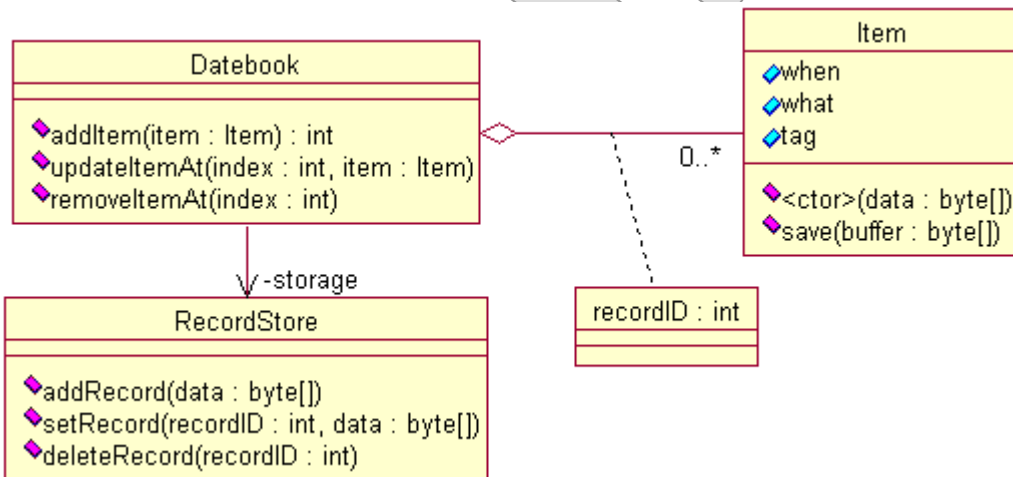
“Live” Enumerations

- The **enumerateRecords** method can also take advantage of model events.
 - It can create a “live” enumeration – that is, one that registers as a listener for model events and updates itself accordingly.
 - Pass **true** for the final argument:

```
RecordEnumeration enum = myRecords.enumerateRecords  
(null, null, true);
```

- By creating a live enumeration, an application can be assured that each time it queries the enumerator – say for the next or previous record – it is getting information synchronized to the latest changes.
- **Live enumerations are expensive, not only in event-handling costs but in that the enumeration needs to keep updating itself whenever changes are made.**
 - The cost here is less in memory use and more in performance.
 - Code elsewhere in a MIDlet might run very slowly even though all it seems to do is delete a record.

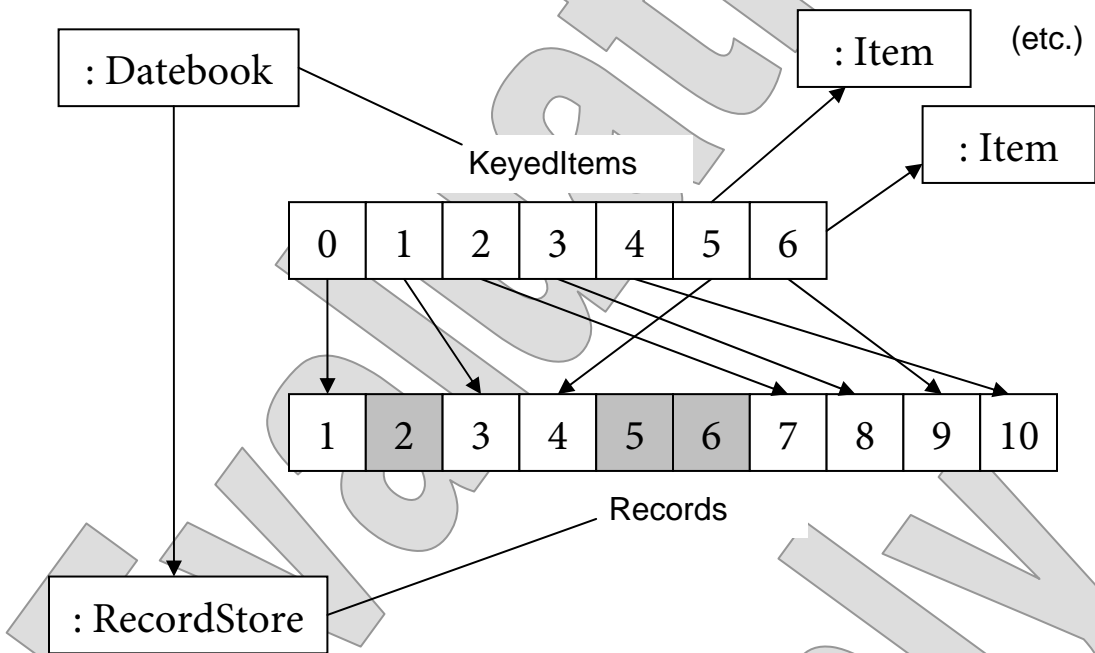
- In **Examples/Datebook/Step7**, we can see the complete persistence design for the Datebook MIDlet.
 - We've seen the persistence code for **Items**.
 - Each instance of this class is stored in a separate record.
 - The **Datebook** model manages an open record store over the run of the MIDlet.



Datebook Persistence

EXAMPLE

- If record IDs were adjusted down each time an old record were deleted, then this management would be trivial – perhaps to the point at which the **Datebook** class wouldn't even be needed.
- This is not the case, though, and so **Datebook** has among its responsibilities the mapping from an *Item* instance to its record ID:



- Note that **Datebook** also represents items in order of their **when** dates, where RMS records are ordered by create time.

- The **Datebook** class therefore keeps a **Vector** whose instances are not **Items** *per se*, but instances of an inner class **KeyedItem**:

```
private class KeyedItem
{
    public KeyedItem () {}
    public KeyedItem (Item item, int recordID)
    {
        this.recordID = recordID;
        this.item = item;
    }
    public int recordID;
    public Item item;
}
```

- Thus each element in the vector includes the incarnated **Item** and also a link back to its persistent source.
- Mutators on **Datebook** can therefore find the correct record to change or remove at a later time:

```
// From removeItemAt ():
store.deleteRecord
(((KeyedItem) elementAt (index)).recordID);
```

- The class **adapts** the RMS record-ID arrangement so that it is of no concern to the rest of the application. Calls to the **Datebook** all speak in terms of translating a zero-based index to an **Item**:

```
public int addItem (Item item);
public void updateItemAt (int index);
public void removeItemAt (int index);
```

SUMMARY

- **The Record Management System is, like much of MIDP, “minimal and complete.”**
 - The interface is primitive, forcing the application to work directly with byte arrays.
 - This is necessary to facilitate the most efficient possible storage of application data.
 - Working through the streams model can save greatly on application code, without much sacrifice, if any, of data compactness.
- **The partitioning of data into named record stores and then numbered records offers a few logical options for various data schemes.**
- **Application-specific compression schemes are recommended wherever they can be reasonably implemented.**
 - Just keep that number in mind, like something from Harper’s Index: “Number of kilobytes of persistent data storage guaranteed to be available on a MIDP device: 8.”