

Sun Certified Mobile Application Developer (311-110)
Sun Microsystems, Inc.

Exam Notes
Sathya Srinivasan

02/09/2004

Table of Contents

Handling a RecordStore	3
Opening a RecordStore.....	3
Closing a RecordStore	3
Sharing a RecordStore.....	4
Removing a RecordStore.....	4
Adding a record	5
Retrieving a record	5
Modifying a record.....	5
Deleting a record	5
Converting RecordStore data to byte arrays.....	6
Converting RecordStore data from byte arrays.....	6
Performance Implications	6
RecordFilter.....	7
RecordComparator.....	7
RecordListener	7
RecordEnumeration.....	7

Chapter 6 MIDP 2.0 Persistent Storage

Develop code that correctly implements handling, sharing and removing RecordStores within MIDlet suites.

Handling a RecordStore

- A MIDlet suite can have multiple RecordStores.
- RecordStores remain persistent across multiple invocations.
- Each RecordStore should have a unique name in a MIDlet suite.
- RecordStore names are case-sensitive and cannot exceed 32 characters.
- The MIDlet suite is identified by the MIDlet-Vendor and MIDlet-Name attributes.
- No locking operations are provided.
- RecordStore operations are atomic, synchronized, and serialized.

- Time stamp format is same as that of `System.currentTimeMillis()`
- Records are byte arrays.
- Records are identified by an integer record id.
- Record ID starts with 1 and increases by 1 monotonically.
- `RecordStore.getVersion()` returns a number indicating the number of times a change (add, delete, or change) has been made on it.

Opening a RecordStore

- `static RecordStore.openRecordStore(String name, boolean create)`
 - Created if one is not already available.
 - Name cannot exceed 32 characters. If more or null or empty, does this throw an `IllegalArgumentException`?
 - `RecordStoreFullException` if there is no space.
 - `RecordStoreNotFoundException` if there is no RecordStore by the given name and if the `create` parameter is `false`.
 - `RecordStoreException` if there is a general error.

Closing a RecordStore

- `RecordStore.closeRecordStore()` removes all listeners and makes the `RecordEnumeration` invalid.
- Number of open calls and close calls should be the same for proper functioning.
- `RecordStoreNotOpenException` is thrown if you attempt to close an already closed store.

Sharing a RecordStore

- Sharing of records is possible only from MIDP 2.0
- `RecordStore.openRecordStore(String name, boolean create, int authMode, boolean writable)`
- If `authMode` is `AUTHMODE_PRIVATE`, it is same as the normal open.
- If `authMode` is `AUTHMODE_ANY`, the `RecordStore` opened will be shared with other MIDlet suites.
- If the `authMode` is `AUTHMODE_ANY` and you try to access a `RecordStore` opened using `AUTHMODE_PRIVATE`, the `RecordStore` is NOT shared.
- Untrusted MIDlets CAN share `RecordStores`.
- `RecordStore.openRecordStore(String name, String vendor, String suite)`
 - Opens a shared `RecordStore`.
 - If requested `RecordStore` was opened using `AUTHMODE_PRIVATE`, this succeeds only if this requesting MIDlet has the same vendor and suite as the MIDlet that created the requested `RecordStore`.
 - If requested `RecordStore` was opened using `AUTHMODE_ANY`, it always succeeds.

Removing a RecordStore

- If a MIDlet suite is removed, all associated `RecordStores` will be removed.
- `static RecordStore.deleteRecordStore(String storeName)`
 - Only a MIDlet in the MIDlet suite that created the `RecordStore` can delete the `RecordStore`.
 - If the `RecordStore` is open when an attempt is made to delete it, then a `RecordStoreException` is thrown.
 - `RecordStoreNotFoundException` is thrown if there is no `RecordStore` with the given name.
 - No delete events are sent to listeners.

Develop code that correctly implements adding, retrieving, modifying, and deleting individual records in a RecordStore, and converting RecordStore record data to and from byte arrays, and that reflects performance implications.

Adding a record

- `int addRecord(byte[] data, int offset, int numBytes)`
- `data` may be an empty array or `null`.
- `numBytes` may be zero.
- `RecordStoreNotOpenException` thrown if the `RecordStore` is not open.
- `RecordStoreFullException` thrown if there is no space in the device.
- `RecordStoreException` is thrown for general errors.
- `SecurityException` is thrown if the operation is not allowed.

Retrieving a record

- `RecordStore.getRecord(int recordId):byte[]` gets the data for the given ID.
 - `InvalidRecordIDException` is thrown if the passed id is invalid.
- `RecordStore.getRecord(int recordId, byte[] data, int offset):int` gets the data for the given ID and copies it to the passed array. The return value represents the number of bytes read from the offset.
 - `InvalidRecordIDException` is thrown if the passed id is invalid.
 - `ArrayIndexOutOfBoundsException` is thrown if the given array is smaller than the record.
- `RecordStore.getRecordSize(int recordId)` returns the size of the data.

Modifying a record

- `RecordStore.setRecord(int recordId, byte[] newData, int offset, int numBytes)`
 - Overwrites the existing data.
 - `SecurityException` is thrown if the operation is not allowed.

Deleting a record

- `RecordStore.deleteRecord(int recordId)` deletes a given record from the store.
- The `recordId` of the deleted record WILL NOT be reused.

Converting RecordStore data to byte arrays

```
byte[] data = recordStore.getRecord(recordId);
ByteArrayInputStream byteIn = new ByteArrayInputStream(data);
DataInputStream in = new DataInputStream(byteIn);
int someInt = in.readInt();
int someString = in.readUTF();
```

Converting RecordStore data from byte arrays

```
int someInt = 1;
String someString = "Hello";
ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
DataOutputStream out = new DataOutputStream(byteOut);
out.writeInt(someInt);
out.writeUTF(someString);
byte[] data = byteOut.toByteArray();
recordStore.addRecord(data, 0, data.length);
```

Note

If the data contains only one String, then the `String.getBytes()` method can be used instead of `ByteArrayStreams` (Thanks, Mark Spritzler).

Performance Implications

- Adding a `RecordFilter` can have performance implications on a `RecordEnumeration`.
- Setting the `keepUpdated` flag on a `RecordEnumeration` will have SEVERE performance implications as the index of the enumeration will be rebuilt each time the underlying `RecordStore` is changed.
- Adding a `RecordListener` to a `RecordStore` might have an impact on the performance of the `RecordStore`.

Identify correct and incorrect statements or examples about filtering, comparing, event listening, and enumerating records in a RecordStore.

RecordFilter

- `boolean matches(byte[] candidate)`
- The `candidate` should be considered as read-only.

RecordComparator

- `int compare(byte[] candidate1, byte[] candidate2)`
- `EQUIVALENT` (0) should be returned if the candidates are equal.
- `PRECEDES` (-1) should be returned if `candidate1` precedes `candidate2`.
- `FOLLOWS` (1) should be returned if `candidate1` follows `candidate2`.

RecordListener

- `void recordAdded(RecordStore store, int recordId)` invoked if a record is added.
- `void recordChanged(RecordStore store, int recordId)` invoked if a record is changed.
- `void recordDeleted(RecordStore store, int recordId)` invoked if a record is deleted.
- `RecordStore.addRecordListener(RecordListener listener)` can be used to register a listener to a `RecordStore`.

RecordEnumeration

- `RecordStore.enumerateRecords(RecordFilter filter, RecordComparator comparator, boolean keepUpdated)` is used to get an enumeration for a `RecordStore`.
- `RecordEnumeration` keeps an internal list of record ids which is a snapshot of the ids in the `RecordStore` when the enumeration is created first.
- If the associated `RecordStore` is closed, the `RecordEnumeration` becomes invalid.
- If an operation is performed on a `RecordEnumeration` after the `RecordStore` is closed, the record MAY return an invalid ID or throw a `RecordStoreNotOpenException`, even if the `RecordStore` is opened later.
- During enumeration, if some records are deleted, the enumeration might still return the invalid ids (if it is not set to listen for updates).
- During enumeration, if some records are added, the enumeration might not return the new ids (if it is not set to listen for updates).
- First call to `nextRecord()` will return the record data from the first record in the enumeration's sequence.
- First call to `previousRecord()` will return the record data from the last record in the enumeration's sequence.

- An `IllegalStateException` is thrown if the `RecordEnumeration` is accessed after `RecordEnumeration.destroy()` has been called.
- If the `keepUpdated` is `true`, then the index of the `RecordEnumeration` is updated after EVERY change to the underlying `RecordStore`. This `RecordEnumeration` is registered as a listener to the `RecordStore`.
- Having the `keepUpdated` parameter as `true` might SEVERLY affect the performance of the application.
- `keepUpdated(true) == rebuild()`
- The `byte` array returned by `nextRecord()` method is ONLY A COPY. Changes made to the array will not be reflected in the data stored in the `RecordStore`.
- Calling `numRecords()` will force the enumeration to recreate the indexes.
- After calling `nextRecord()` or `nextRecordId()`, the enumeration is advanced to the next available record.
- After calling `previousRecord()` or `previousRecordId()`, the enumeration is advanced to the previous available record.
- The `reset()` method will cause the internal indexes to be restored to the same set that was created when the enumeration was instantiated.