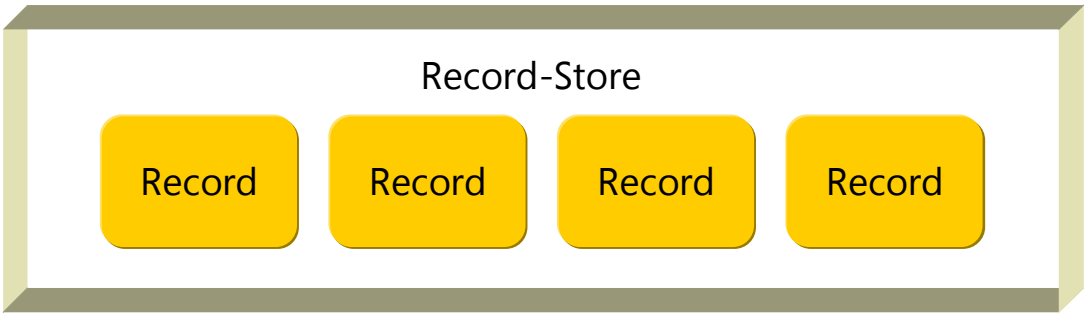
Record Management System (RMS)

Notwendigkeit der Persistenz

- Konfigurations- und nutzerspezifischen Daten müssen auf einem mobilen Endgerät persistent gemacht werden.
  - Benutzername/Passwort
  - Kontakte
  - Termine
  - PIN-Nummern
  - Letzter Zugriff
- Die Techniken, die für die Java SE eingesetzt werden – etwa JDBC oder Schreiben in Dateien – kommt für die Java ME nicht in Frage.

Record Management System (RMS)

- MIDP definiert mit dem **Record Management System** (**RMS**) eine Möglichkeit, Daten im Speicher des Geräts abzulegen.
- Das RMS ist nicht mit einer Datenbank zu vergleichen.
- Ein **Record** speichert binäre Informationen und ist vergleichbar mit einer Datei.
- Eine MIDP-Datenbasis besteht aus einer Sammlung von Records, die **Record-Store** genannt wird.



Leistung vom RMS

- Mit dem RMS kann ein Midlet
  - neue Records zu einem Record-Store hinzufügen
  - existierende Records aus einem Record-Store löschen
  - mit einem anderen Midlet aus der gleichen Anwendung Records teilen
- Die Datenbasis ist für jedes Midlet einer Anwendung gegen fremde Anwendungen gesichert.
  - Erst in MIDP 2.0 gibt es die Möglichkeit, dass ein Midlet einem fremden Midlet Daten freigeben kann.
- Für das RMS ist das Paket javax.microedition.rms reserviert.

Der Record-Store und seine Records

- Ein Record-Store wird durch einen Namen identifiziert, unter dem dann die Daten wiedergefunden werden können.
  - Es setzt sich durch eine Sammlung von Record-Objekten zusammen.
- Jedes Record ist durch eine Record-ID (int) gekennzeichnet.
  - Sie ist eindeutig und vergleichbar mit einem Primärschlüssel.
  - Die Nummerierung beginnt bei 1.

Namen des Record-Stores

- Der Name eines Stores ist
  - auf 32 Unicode-Zeichen beschränkt
  - abhängig von der Groß-/Kleinschreibung.
- Innerhalb einer Applikation kann ein Midlet keine zwei Record-Stores mit dem gleichen Namen anlegen.
  - Der Name identifiziert das Record-Store eindeutig, aber zwei unterschiedliche Applikationen können natürlich den gleichen Store-Namen wählen.
- Die statische Funktion RecordStore.listRecordStores() liefert ein String-Feld mit allen Record-Stores, die ein Midlet besitzt.

Ein Record-Store öffnen

- Mit der statischen Funktion RecordStore.openRecordStore() wird ein Record-Store geöffnet.
- Der Parameter ist
  - der Name der Datenbasis und
  - ein Wahrheitswert, der bei true im Fall einer fehlenden Datenbasis unter diesem Namen eine neue Record-Store anlegt.

```
RecordStore db = RecordStore.openRecordStore( "db", true );
```

- Bei einem geöffneten Record-Store liefert openRecordStore() die gleiche Referenz auf das RecordStore-Objekt.

Ein Record-Store schließen/löschen

- Die Objektfunktion closeRecordStore() eines RecordStore-Objekts schließt das Record-Store.
- Um das ganze Record-Store zu löschen dient die statische Funktion RecordStore.deleteRecordStore().
  - Der Parameter ist der Name des Stores.

Ein Record anlegen

- Das Record war Teil des Record-Stores und im Wesentlichen ein Array von Bytes.
- Mit der Funktion addRecord() kann ein neues Record in das Store gelegt werden.
  - Es wird automatisch hinter das letzte Record gelegt.
  - addRecord() gibt die Record-ID zurück.

```
byte[] b = new byte[len];
b[i] = v;
db.addRecord( b, 0, b.length );     // Feld, Start, Länge
```

- Das erste Record bekommt die ID 1.

Ausnahmen bei addRecord()

- Folgende Ausnahmen sind möglich:
  - RecordStoreNotOpenException
  - RecordStoreFullException. Das dümmste was passieren kann.
  - SecurityException
  - RecordStoreException. Ganz allgemein. (RecordStoreNotOpenException, RecordStoreFullException sind von diesem Typ)
- Um einem RecordStoreFullException vorzubeugen, kann man mit getSizeAvailable() nachfragen, wie viel Bytes noch verfügbar sind.

Daten über Ströme schreiben

- Das nackte Bytefeld ist im allgemeinen schlecht zu bearbeiten.
- Mit einem ByteArrayOutputStream lässt sich ein Bytefeld mit write()-Methoden sequentiell füllen.
- Ein OutputStream kann zu einem DataOutputStream erweitert werden, dass sich auch primitive Datentypen und Strings schreiben lassen.

```
ByteArrayOutputStream baos =
  new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream( baos );
dos.writeUTF( "Eine Zeichenkette" );
byte[] b = baos.toByteArray();
db.addRecord( b, 0, b.length );
```


Die Datenströme wiederverwenden

- Bei der Programmierung von mobilen Anwendungen muss man immer die limitierte Hardware vor Augen haben.
- Es ist teuer, Bytefelder immer neu anzulegen und auch die Datenströme immer neu anzulegen.
- Der Trick ist, die Datenströme immer weiterzuverwenden.
- Ein ByteArrayOutputStream setzt man mit reset() in der Startzustand.
  - Man muss (darf) ihn nicht mit close() schließen, bevor man mit toByteArray() Daten entnimmt.

Daten lesen

- Mit zwei Funktionen lasen sich Daten aus dem Record auslesen:
  - byte[] getRecord(int recordId)
  - int getRecord(int recordId, byte[] buffer, int offset)
- Die zweite Variante ist im Allgemeinen klüger, da kein neues Bytefeld aufgebaut werden muss.
- Einen direkten Zugriff ohne Kopie in ein Feld gibt es nicht!

```
RecordStore rs;
rs = RecordStore.openRecordStore( "Hello" ,true );
if ( rs.getNumRecords() > 0 )
  String s = new String( rs.getRecord(1) );
```

- getRecordSize() liefert die Größe des Records für den nötigen Speicherbedarf

Daten lesen, verändern, speichern

```
RecordStore rs = null;
int         id = ....;          // die Record-ID

try {
  rs = RecordStore.openRecordStore( "mydata", false );
  byte[]  data = rs.getRecord( id );

  ... Daten modifizieren ...

  rs.setRecord( id, data, 0, data.length );
  rs.closeRecordStore();
}
catch( Exception e ) { ... }
```

Nebenläufiger Zugriff

- Alle Operationen auf dem Record-Store sind gegen parallele Zugriffe gesichert.
  - Sie sind atomar und synchronisiert.
- Das heißt aber nicht, dass mehrere Operationen hintereinander automatisch atomar sind.
  - Hier muss man mit einem passenden Lock-Objekt selbst über mehreren zu schützenden Operationen synchronisieren.

Die Hardware: Der Handy-Speicher

- Die Daten des Benutzers werden im Handy-Speicher abgelegt.
  - Der Chip muss, der nach dem Ausschalten des Systems seine Daten bewahren.
- Ein EPROM kommt nicht in Frage, da die Speicherbausteine nur mit UV-Licht gelöscht werden können.
- Zwei Lösungen werden verfolgt:
  - Ein extra EEPROM (etwa von der Firma atmel).
  - Ein EEPROM-Bereich im Flash-Speicher. Aktuell.

Flash-Speicher

- Flash-Speicher ist eigentlich nichts weiter als eine verbesserte EEPROM-Technologie.
  - Sie lässt schnelleres Schreiben als ein normaler EEPROM zu.
  - Flash-Speicher haben oft viel mehr Speicherkapazität.
- Flash-Speicher sind etwa die Baureihe M45PExx des Herstellers STMicroelectronics.
  - Sie bieten eine Seitengröße von 256 Byte.
  - Eine 256-Byte-Seite lässt sich in 12 ms beschreiben, in 2 ms programmieren und in 10 ms löschen.
  - Der Deep-Power-Down Mode sorgt für eine minimale Stromaufnahme von 1µA im Standby-Modus.

Problem: Performance

- Einem DIM-Modul, das am 100-MHz Bus laufen soll, bleiben nur 6 ns für die Bereitstellung der Daten.
  - Beim Flash-Speicher liegt der Zugriff bei ~10 ms.
  - 1 ms = 10^{-3} sec, 1 ns = 10^{-9} sec.
  - Der Unterschied zwischen Milli und Nanno ist also Faktor 10^6, also im Millionenbereich.
- Das Lesen und Schreiben bei den Geräten in das Record-Store ist im Allgemeinen langsam.
  - Das liegt am Speichertyp.
  - Das Schreiben ist oft noch langsamer als das Lesen.
- Das heißt also, auf das Schreiben wenn möglich zu verzichten und die Datenblöcke klein zu halten.
- An Stelle eines Records mit 100 K, und häufigen Änderungen, von 10 Bytes besser 10 * 10 K.

Zugriffszeiten das RMS

- Zugriffszeiten auf das RMS zweier Handys:

Funktion	C55	Nokia 6610
getRecord()	210 ms	600 ms
addRecord()	280 ms	10 ms
setRecord()	370 ms	470 ms
deleteRecord()	720 ms	300 ms

Lesen über Datenströme

- Genauso wie man komfortabel über DataOutputStream geschrieben hat, kann man auch über DataInputStream die Daten lesen.

```
byte[] record = getRecord( recordId );
DataInputStream dis = new DataInputStream(
  new ByteArrayInputStream(record) );
String in = dis.readUTF();
```

Record aus Record-Store löschen

- Um ein Record zu löschen, muss man die ID haben.
- Die IDs muss man sich beim Hinzufügen merken!
  - Dazu kann man bei jedem addRecord() die ID in eine Datenstruktur wie den Vector schreiben.
  - Mit getNextRecordID() - 1 kommt man an die aktuelle ID.
- Dann löscht deleteRecord() den Record.
- Die ID des gelöschten Records wird nicht freigegeben!
  - getNextRecordID() liefert maxId + 1
  - Damit kann man nicht herausfinden, wie viele Records gerade im Store sind, da einige zwischendurch gelöscht worden konnten.
  - getNextRecordID() sagt aus, wie viele Records im Leben eines RecordStores schon angelegt wurden.

Records eines Typs aufzählen

- Das RMS bietet die Möglichkeiten mit enumerateRecords() Records eines gewünschten Schemas aufzuzählen.
  - Records können mit einem RecordFilter ausgefiltert werden.
  - Records können mit einem RecordComparator sortiert werden.

Alle Records aufzählen

- Um alle Records eines Record-Stores aufzuzählen kann man entweder alle IDs durchlaufen und mit getRecord() die Daten beziehen oder enumerateRecords() nutzen.
  - Dann sind in enumerateRecords() die Parameter für RecordFilter und RecordComparator null.

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

- Der Vorteil liegt aber auf Seiten von getRecord(), denn hier konnte man das Bytefeld wiederverwenden.

RecordComparator

- Für den RecordComparator definiert man dann, ob ein Record größer/gleich/kleiner ist.
  - Dazu definiert die Schnittstelle einer compare()-Methode, die zwei Records vergleicht und
  - passend FOLLOWS, EQUIVALENT oder PRECEDES liefert.

```
public class MyComparator implements RecordComparator
{
  public int compare( byte[] rec1, byte[] rec2 )
  {
    // entweder FOLLOWS, EQUIVALENT oder PRECEDES
  }
}
```

Beispiel-RecordComparator

```
public class MyComparator implements RecordComparator {
  public int compare( byte[] rec1, byte[] rec2 )
  {
    String str1 = new String( rec1 ),
           str2 = new String( rec2 );
    int result = str1.compareTo( str2 );
    if ( result == 0 )
      return RecordComparator.EQUIVALENT;
    else if ( result < 0 )
      return RecordComparator.PRECEDES;
    else
      return RecordComparator.FOLLOWS;
  }
}
```

Diese Lösung ist nicht besonders performant!

Listener

- Änderungen an der Datenbasis können mit einem RecordListener verfolgt werden.
- Mit addRecordListener() an ein RecordStore hinzufügen:
  - void addRecordListener(RecordListener listener)
- Die Schnittstelle RecordListener definiert drei Methoden:
  - void recordAdded(RecordStore recordStore, int recordId)
  - void recordChanged(RecordStore recordStore, int recordId)
  - void recordDeleted(RecordStore recordStore, int recordId)

Synchronisation

Die Versionsnummer

- Immer wenn der RecordStore geändert wird, zum Beispiel wenn Records eingefügt/gelöscht oder verändert werden, ist die Änderung an zwei Eigenschaften sichtbar:
  - Dem Datum der letzen Änderung
  - Einer Versionsnummer.
- Das Datum wird mit System.currentTimeMillis() gespeichert und ist mit getLastModified() auszulesen.
- Die Version ist eine Ganzzahl, die bei jeder Änderung inkrementiert wird.
  - Sie lässt sich mit getVersion() auslesen.
- Damit kann eine Synchronisationssoftware implementiert werden.

Synchronisation

- Mit dem Datum der letzen Änderung kann eine Synchronisation der Inhalte vorgenommen werden.
  - Von MIDP-Seite gibt es dafür allerdings keine Unterstützung.
  - Jedes Record muss von einem Programm neu überschrieben oder korrigiert werden.

Synchronisation mit Palm Pilot

- Falls das Midlet innerhalb des Palm Pilots läuft, hilft eine besondere Beziehung zwischen den MIDP-Records und den Palm-Records bei der Synchronisation.
  - Die Implementierung des RMS für die Java ME bildet automatisch jedes Record auf ein Record des Palms ab.
  - Mit anderen Worten: Wenn man einen Palm-Record synchronisiert, aktualisiert man damit das Java-Record mit.
- Info über diese Art liefert „Sync up Palm OS with J2ME. Develop Palm HotSync conduits that interact with MIDP apps“ unter http://www.javaworld.com/javaworld/jw-05-2002/jw-0531-palm_p.html.

SyncML

- Aus dem Konsortium von IBM, Lotus, Motorola, Nokia, Palm, Psion und Starfish ging der offene Standard „Synchronization Markup Language“ (SyncML) vor.
- SyncML basiert auf XML und ermöglicht den Datenabgleich zwischen mobilen Endgerät und Server.
  - EMail, Kalender, Todo-Listen, Kontaktinformationen
- Serverseitig arbeitet ein Sync-Server, mit dem Protokoll-Kommandos ausgetauscht werden.
  - Add, Alert, Atomic, Copy, Delete, Exec, Get, Map, Replace, Search, Sequence, Sync.
- Für MIDP gibt es im Moment noch keine generische Unterstützung durch eine Bibliothek.