Mobile Systeme
Grundlagen und Anwendungen standortbezogener Dienste

Location Based Services in the Context of Web 2.0

Department of Informatics - MIN Faculty - University of Hamburg
Lecture Summer Term 2007

Dr. Thilo Horstmann

CLDC

NMEA

MIDP

Google Earth

OpenGIS

SQL

KML

Bluetooth

Mash-Ups

Web 2.0

J2ME

Loxodrome

Euler Spaces

RDMS

GPS

PostGIS

GPX

Maps

JSR 179

Polar Coordinates

API

Threads

Today: J2ME (II)

- Threads: What‘s different to J2SE
- Persistent Storage: RMS
- Homework :-)

Threads in J2ME

The minimal J2ME application

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

public class Hello extends MIDlet{

    public Hello() {}

    public void pauseApp() {}

    public void destroyApp(boolean unconditional) {}

    public void startApp(){

        //only in WTK emulator window

        System.out.println("Hello World!");

    }

}
```

Threads in J2ME

- perform multiple activities simultaneously
  - keep the User Interface responsive if some background tasks need to be executed (network, Record Store, File IO, Complex Computations, etc.)
- Built-in support for threads in Java incl. J2ME
  - in contrast to many other languages
- Thread handling in J2ME similar to J2SE
  - therefore, not all aspects of thread handling are covered here
  - but some differences exist though
- Due to lack of resources on mobile devices a proper thread handling is important

What is a thread?

- a thread is a **a single sequential flow of control within a program.** Threads are the fundamental units of program execution. Every running application has at least one thread. An application consisting of two or more threads is known as a multithreaded application.
- Each thread has its **own execution stack** (control stack, function stack, run-time stack, or just stack). Consequently method arguments and local variables are not shared by other threads.
- Unlike (UNIX) processes, threads share data. In particular, all **Objects in the heap are shared by all threads**. This includes all class instances and arrays.

The four states of a thread

- **Live** thread:
  - A **running** thread is executing code.
  - A **ready** thread is ready to execute code.
  - A **suspended** thread is waiting on an external event. For example, it may be waiting for data to arrive from another device. After the event occurs, the thread transitions back to the ready state.
- **Dead** thread:
  - A **terminated** thread has finished executing code.

As in J2SE there are two ways to use threads (1)

- Subclass class Thread

```
public class DoAnotherThing extends Thread {
    public void run(){
     // do something
    }
}


...
DoAnotherThing doIt = new DoAnotherThing();
doIt.start();
...
```

As in J2SE there are two ways to use threads (2)

- Extend Interface Runnable

```
public class DoSomething implements Runnable {
    public void run(){
     // here is where you do something
    }
}


...
DoSomething doIt = new DoSomething();
Thread myThread = new Thread( doIt );
myThread.start();
...
```

No stop() in CLDC

- The stop() method has been deprecated because it is inherently unreliable and cannot be implemented on all platforms safely and consistently.
- a J2ME thread lives until it intentionally or unintentionally exits the run() method it invoked on startup.
  - unintentionally: e.g. as a result of an uncaught exception or if the AMS terminates the application for some reason (low battery, low on resources)
- There is no way for one thread to force another thread to stop
- And consequently, there is no restart().

But how to stop a thread then?

Each thread periodically checks a boolean variable

```
public class MyThread implements Runnable {
    private boolean quit = false;

    public void run(){
         while( !quit ){
          // do something
         }
    }

    public void quit(){
         quit = true;
    }
}
---
MyThread myThread = new MyThread();
myThread1 = new Thread( myThread );
myThread2 = new Thread( myThread );
myThread.quit(); ?
```

Discussion

- works well when only one thread at a time executes a given run() method
- but multiple threads can share the same runnable object, i.e. they share the same run() method
  - the example before would terminate all threads
- Often you want to terminate all but the most recent thread.
  - Thread.currentThread() returns a reference to the current executing Thread. If many threads share the same Runnable object, it may be important to determine which thread is actually running at any given time.

What else to consider

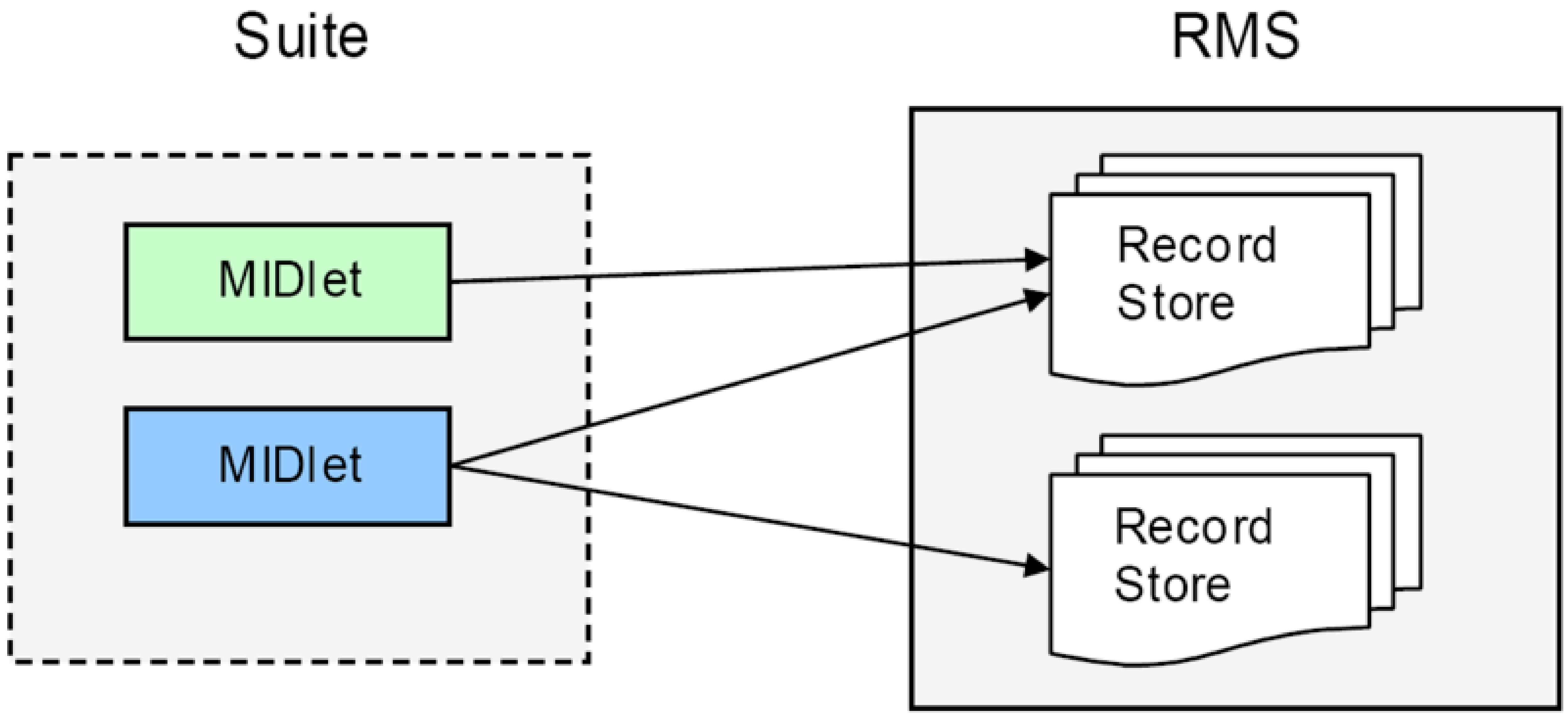
- Generally Thread handling (synchronization) in J2ME is very similar to J2SE
  - synchronization to shared data as in J2SE
  - No ThreadGroups: Use your own Collections instead
  - Synchronization in J2SE classes generally carry over to its J2ME counterparts
- Prefer implementing the Runnable Interface
  - slightly lower overhead (not that important in J2SE but on some limited devices)
- Release resources (used by threads) as soon as possible
  - avoiding memory leaks is especially important in J2ME applications.

J2ME Record Management System (RMS)

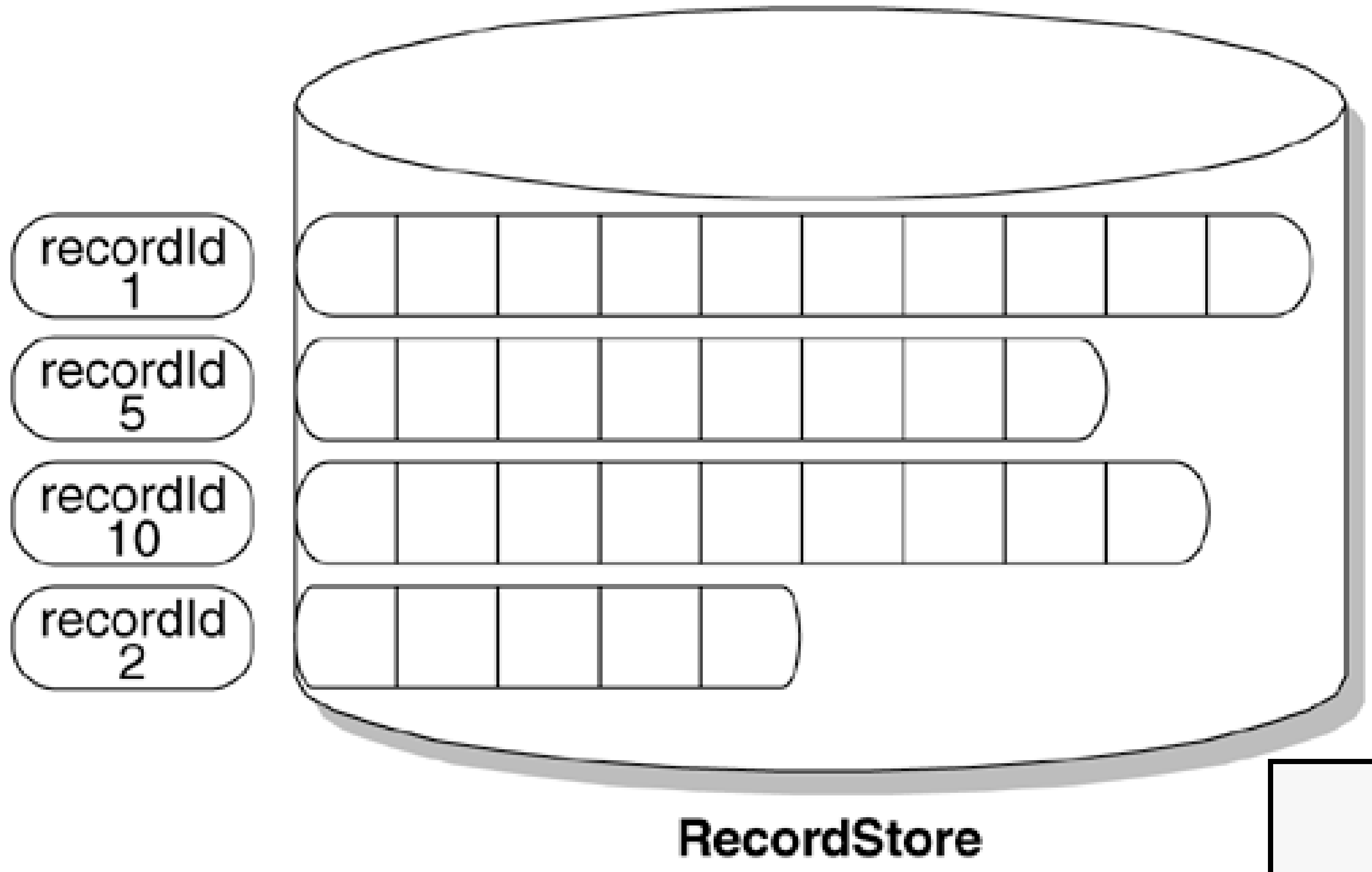
Overview

- MIDP stores all information in non-volatile memory, using a storage system called the Record Management System (RMS)
- On many MIDP devices there is no FileSystem
    - or there may be no access to the file system (Java API not available)
- MIDP Record Management System (RMS) provides a means to store application data that persists across invocations of a MIDlet
    - sort of a very simple database, where each row consists of two columns: one containing a unique row identifier, the other a series of bytes representing the data in the record
    - (often) the RMS is the only way to persist application data
- provided in package javax.microedition.rms

Overview: The RMS contains RecordStores



Structure of a RecordStore



ID	DATA
1	byte[]
2	byte[]
3	byte[]
...	...

Key Concepts

- ***Record***: an individual data item, i.e. anything that a sequence of bytes can represent. A record consists of a single binary field of variable size
  - You (the application developer) are responsible for interpreting the contents of a record!
  - At the API level, records are simply byte arrays
- ***Record Store***: an ordered collection of records.
  - A Record Store is uniquely identified (within the MIDlet suite) by its name (32 Unicode characters)
  - MIDP 2.0 optionally allows a MIDlet suite to share a record store with other suites, in which case the record store is identified by the names of the MIDlet suite and its vendor, along with the record store name itself.

RMS Aspects

- Storage Limits
  - MIDP specification requires devices to reserve at least 8K of non-volatile memory for persistent data storage.
  - limits on the size of an individual record is not specified by MIDP
  - as always, storage is a limited resource!
- Thread Safety
  - RMS operations are thread-safe. As with any other shared resource, threads must still coordinate the reading and writing of data to the same record store
- Speed: Write operations may take a long time (depending on device)
  - cache frequently accessed data in volatile memory

Using RMS: Locate a Record Store

- a MIDlet suite can discover its own RecordStores:
- `String[] names = RecordStore.listRecordStores();`
- A name contained in names can be used to open the RecordStore
- The MIDP specifications don't include any way to list the record stores of other MIDlet suites.
    - However, they can be used but you need to know their name

Using RMS (2): Creating, Opening, and Closing a Record Store

```
public void doSomething( String rsName ) {
   RecordStore rs = null;
   try {
       // 2nd parameter set to false does not create a RS
      rs = RecordStore.openRecordStore( rsName, false );
      // do something
   } catch( RecordStoreException e ){
      // log exception
   //carefully release resources
   } finally {
      try {
         rs.closeRecordStore();
      } catch( RecordStoreException e ){
              // Ignore this exception
      }
   }
}
```

The CRUD operations of a record are simple!

- CRUD: Create, Read, Update, Delete
- Remember, only byte arrays are allowed!

```
...
byte[] data = new byte[]{ 0, 1, 2, 3 };
int    recordID;
recordID = rs.addRecord( data, 0, data.length );
...
```

- Similarly, the `setRecord` method is used to update a record
- Use `getRecord` to read the contents of a record
  - make sure the supplied byte array is large enough to hold the data being read (use `rs.getRecordSize( recordID )`)
- and delete a record with `rs.deleteRecord( recordID )`

A (bad) brute force search through all records

```
...
int    nextID = rs.getNextRecordID();
byte[] data = null;
for( int id = 0; id < nextID; ++id ){
    try {
        int size = rs.getRecordSize( id );
        if( data == null || data.length < size ){
            data = new byte[ size ];
        }
        rs.getRecord( id, data, 0 );
        processRecord( rs, id, data, size ); // process it
    } catch( InvalidRecordIDException e ){
        // ignore, move to next record
    } catch( RecordStoreException e ){
        handleError( rs, id, e ); // call an error routine
    }
}
...
```

Use enumerations instead

- `enumerateRecords` is used to retrieve a subset of the records in a record store

```
RecordStore rs = RecordStore.openRecordStore("store", true);
RecordEnumeration enumeration = rs.enumerateRecords
(null,null,true);

while(enumeration.hasNextElement()) {
  int recordID = enumeration.nextRecordId();
  byte[] myBytes = rs.getRecord(recordID);
}
```

- `enumerateRecords` can even be used with filters and comparators.
  - comperator: determine the order in which the records are returned.
  - filter: defines the subset to be retrieved

How to deal with data types

- The RMS require to deal with an unsophisticated binary format
- In most cases, we need to convert the bytes retrieved to something useful
  - e.g. Basic data types like Strings, Integers, etc.
  - or even complex data type like Classes
- Fortunately, MIDP includes standard data-manipulation classes drawn from the core J2SE library
  - ease of development
    - encapsulation of low level API
  - using these classes also help to use port J2SE applications and libraries

Reading: DataInputStream

- A `DataInputStream` transforms a raw input stream into primitive data types and strings

```
InputStream in = ... // an input stream
DataInputStream din = new DataInputStream( in );
try {
    int custID = din.readInt();
    String lastName = din.readUTF();
    String firstName = din.readUTF();
    long timestamp = din.readLong();
    din.close();
}
catch( IOException e ){
    // handle the error here
}
```

Writing: DataOutputStream

- A `DataOutputStream` writes strings and primitive data types to an output stream:

```
OutputStream out = ... // an output stream
DataOutputStream dout = new DataOutputStream( out );
try {
    dout.writeInt( 100 );
    dout.writeUTF( "Smith" );
    dout.writeUTF( "John" );
    dout.writeLong( System.currentTimeMillis() );
    dout.close();
}
catch( IOException e ){
    // handle the error here
}
```

Putting it together: Combine DataInputStream and a ByteArrayInputStream for reading

```
RecordStore rs = ... // a record store
int recordID = ... // the record to read
ByteArrayInputStream bin;
DataInputStream din;
try {
    byte[] data = rs.getRecord( recordID );
    bin = new ByteArrayInputStream( data );
    din = new DataInputStream( bin );
    int id = din.readInt();
    String lastName = din.readUTF();
    String firstName = din.readUTF();
    long timestamp = din.readLong();
    din.close();
    ... // process data here
}
catch( RecordStoreException e ){
    // handle RMS error here
}
catch( IOException e ){
    // handle IO error here
}
```

Putting it together: Combine DataOutputStream and a ByteArrayOutputStream for writing

```
RecordStore rs = ... // a record store
ByteArrayOutputStream bout = new ByteArrayOutputStream();
DataOutputStream dout = new DataOutputStream( bout );
try {
    dout.writeInt( 100 );
    dout.writeUTF( "Smith" );
    dout.writeUTF( "John" );
    dout.writeLong( System.currentTimeMillis() );
    dout.close();
    byte[] data = bout.toByteArray();
    rs.addRecord( data, 0, data.length );
}
catch( RecordStoreException e ){
    // handle RMS error here
}
catch( IOException e ){
    // handle IO error here
}
```

Get notified when the RecordStore changes

- Implement the RecordListener interface and register this object with the RecordStore

```
public void recordAdded(RecordStore recordStore, int recordId) {
   try{
      System.out.println(recordStore.getRecord(recordId)[0]+" is
   added.");
    }catch(Exception e){
    }
}
```

- Similarly, the methods recordDeleted and recordChanged are called

Homework :-)

- Some instances of the class Student need to be stored in a RecordStore „students“

```
public class Student{
   private String firstName;
   private String lastName; //unique
   private Date enrollment;
   private Image photo; //available as a Midlet resource (png)
}
```

- Extend the class with a method write(), such that `aStudent.write()` saves the data into the RS „students“. Also, provide a class Students with a public method `Student lookup(String lastName)`, such that the student named lastName will be retrieved from the RecordStore „students“.
- Use threads when writing data to the Record Store. Take care that resources will be released as soon as possible.

This Lecture

- Breymann, U., Mosemann, H.: Java ME – Anwendungsentwicklung für Handys, PDA und Co., Hanser, 2006, www.java-me.de
  - Chapter 6
- Threads in J2ME
  - http://developers.sun.com/techtopics/mobility/midp/articles/threading2/
- RMS
  - http://java.sun.com/javame/reference/apis/jsr118/
  - http://today.java.net/lpt/a/149
  - http://developers.sun.com/techtopics/mobility/midp/articles/databaserms

Thank you!

Dr. Thilo Horstmann

e-mail: thilo.horstmann@gmail.com
blog: http://www.das-zentralorgan.de