Debreceni Egyetem
Informatika Kar

Mobiltelefon programozás és mobil adatbázis-kezelés

Témavezető:
Dr. Fazekas Gábor
egyetemi docens

Készítette:
Toldi Gábor
informatika tanár

Debrecen
2008

Tartalomjegyzék

I. Bevezetés

A mobiltelefon korunk egyik legfontosabb napi használati eszköze. Szinte el se tudjuk képzelni már életünket a zsebünkben vagy a táskánkban lapuló, különféle hangjelzéseket adó (néha rezgő) „elektronikus társunk” nélkül. Olyan természetessé vált, hogy csak akkor érezzük igazán a hiányát, ha lemerült vagy ha lába kélt.

Személyessé válásának egyik legfőbb oka, hogy egyre inkább már nemcsak a kommunikációban vesszük hasznát, hanem egyéb más feladatok ellátására, sőt szórakoztatásunkra is alkalmas. Sok készülék alkalmazható számológépként, diktafonként, MP3 és MP4 lejátszóként, fényképezőgépként sőt videókameraként; némelyiken játszhatunk és internetezhetünk is, valamint megnézhetjük a számítógépen elkészített dokumentumainkat: így nem csoda, ha egyesek korunk „modern svájci bicskájának” titulálják.

Egy másik dolog, amely a XXI. századi ember életét szintén átszövi, akármerre megy: ez a dolog pedig az adat fogalma. Az ember normális igénye, hogy a számára fontos információkat a szelektálás után, egybegyűjtse és valamilyen rendszerben elraktározza. Az adatok és információk szervezett formában történő tárolásával nap mint nap találkozhatunk: gondoljunk csak a különféle hivatalok adatbázisaira vagy akár az e-mail fiókunkra.

Mindezek függvényében, evidens volt hát az az igény, hogy adatbázisok kezelésének lehetősége mobiltelefonokon is megvalósulhasson. Ezen törekvések „úttörőjeként” jelent meg a Java platformon a perzisztens adattárolás, a Record Management System (RMS), amely egy kezdeti megvalósítása az adatok adatbázisokba való szervezésének. Segítségével létrehozhatóak hasznos és érdekes alkalmazások, pl.: szótár program vagy egy mobilbiblia.

II. Ismerkedés a rekordkezeléssel

A Mobile Information Device Profile (MIDP) egyik fő alrendszere a Record Management System (RMS), ami egy alkalmazói programozási-interfész (API), amely a MIDP-nek nyújt szolgáltatásokat, eszközöket az adatok helyi tárolásához. A legtöbb MIDP-t támogató eszközön, ma az RMS-en keresztül oldja meg a helyi adattárolást. Kevés az az eszköz, amelyek csak a hagyományos állományrendszert támogatja.
Kezdjük az áttekintést azzal, hogy mi szerepel az RMS ajánlásában, specifikációjában és vegyük át a Record Management System fő fogalmait.

1. Rekordok

Ahogy a nevéből is következtethetünk, az RMS egy olyan rendszer, amely rekordok kezelését végzi. Egy *rekord* egy önálló adattétel. Az RMS nem korlátozza azt, hogy mi kerül egy rekordba: a rekord tartalmazhat számot, karakterláncot, tömböt, képet; bármit, amit bájtok sorozatából létrehozható. Ha tudod binárisan kódolni az adataidat és megvan hozzá a megfelelő dekódoló, akkor el tudod azt raktározni egy rekordban. Fontos persze az előírásokban meghatározott, méretbeli megszorítások betartása.
Sokan, akik csak most ismerkednek a perzisztens adattárolással, összekeverik RMS *rekord* fogalmát. Hol vannak a mezők? - kérdezik miközben azon töprengenek, hogy a rendszer hogyan bontja alosztályokra egyéni rekordokat, különálló adatsorozatokra. A válasz egyszerű: *RMS-ben egy rekordnak nincsenek semmilyen mezői.* Vagy, pontosabban fogalmazva, egy rekord, egy változó méretű különálló bináris mezőt tartalmaz. *Hogy értelmezze a rekord tartalmát az már nem tartozik az alkalmazás hatókörébe.* Az RMS elvégzi a raktározási műveletet és egy egyedi azonosítót biztosít, mást nem tesz. A munkának ez a fajta felosztása teszi az RMS-t egyszerűvé, de mégis rugalmassá az alkalmazásokban, ami fontos sajátossága a MIDP alrendszerének.
Az API szinten a rekordok egyszerűen bájttömbök

2. Rekordtárolók

Egy *rekordtároló* a rekordok egy rendezett gyűjteménye. A rekordok nem független entitások: mindegyiknek egy rekordtárolóhoz kell tartoznia, és minden rekordhozzáférés a rekordtárolón

keresztül történik. Tulajdonképpen, a rekordtároló garantálja, hogy a rekordok olvasása és írása megtörténjen, adat-meghibásodás nélkül.

Amikor egy rekordot létrehozunk, a rekordtároló kijelöl hozzá egy egyedi azonosító, egy egész számot, amit *ID-nek* nevezünk. Az első rekordnak, amit hozzáadunk a rekordtárolóhoz, az ID-je 1 lesz, a másodiknak 2 és így tovább. A rekord ID nem egy index: a rekordtörlések után nem kaphatják újra ugyanazt az ID-t az újonnan felvitt rekordok és nem hatnak a következő rekord értékére sem.

A rekordtárolókat nevekkel azonosítják a MIDlet programban. Egy rekordtároló neve maximum 32 karakter hosszú, Unicode karakterekből állhat, és egyedinek kell lennie a MIDlet programon belül, amely a rekordtároló létrehozásáért felelős. A MIDP 1.0 verzióban a rekordtárolókat nem oszthatják meg különböző MIDlet programok. A MIDP 2.0 opcionálisan megengedi egy MIDlet programnak, hogy megosszon egy rekordtárolót más programokkal, mely esetben a rekordtárolót magával a rekordtároló névvel egyetemben azonosítják a MIDlet program nevei.

A rekordtárolók szintén idő-bélyeget és verzióinformációt tartanak fenn, úgyhogy az alkalmazások lekérdezhetik, hogy egy rekordtárolót mikor módosítottak utoljára. Bezárásig nyomon követhető, egy figyelő segítségével, hogy mikor módosították a rekordtárolót.

Az API szinten egy rekordtárolót a *javax.microedition.rms.RecordStore* osztály egy példánya képviseli. Minden RMS osztályt és interfészt a *javax.microedition.rms* csomagban definiálnak.

A kódok megtekintése előtt átnézzük az RMS-sel kapcsolatos legfontosabb információkat.

3. Tárolási limit

A rendelkezésre álló memóriák mérete, ami a rekordalapú tárolóshoz szükséges, az eszköztől függően változnak

A MIDP részletes leírása megköveteli eszközöktől, hogy legalább 8 Kbyte nem felejtő memóriát foglaljanak le az állandó adattárolásért. A specifikáció nem tesz semmilyen megkötést a rekord méretére, de a tárhelyre vonatkozó megszorítások eszközről-eszközre változnak. Az RMS szolgáltatásokat nyújt arra nézve, hogy meghatározzuk egy egyéni rekord méretét, egy rekordtároló teljes méretét, és mennyi az adattároláshoz szükséges fennmaradó memória mérete. Nem szabad elfelejteni, hogy az állandó memória megosztott, szűkös erőforrás, így érdemes takarékosnak lenni a használatával.

Bármely MIDlet programnak, ami használja RMS-t, az adattárolásnak azt a minimális számú byte-ját kellene specifikálnia, amit azáltal igényel, hogy beállítja a *MIDlet-Data-Size* tulajdonságát a JAR fájlban és az alkalmazás leírójában is. Nem állít be nagyobb értéket, mint amennyi szükséges, az eszköz visszautasíthatja az olyan MIDlet program installálását, amelynél az adattárolás-követelményei felülmúlják a rendelkezésre álló helyet. Ha ez az attribútum hiányzik, akkor az eszköz feltételezi, hogy a MIDlet program nem igényel helyet az adattárolásnak. A gyakorlatban a legtöbb eszköz megengedi az alkalmazásoknak, hogy túllépjék a meghatározott tárhelyet.

Észrevehető, hogy néhány MIDP implementáció megköveteli a programozótól, hogy további attribútumokat határozzanak meg, amelyek a tároló-követelmények részleteivel kapcsolatosak. Ezekkel kapcsolatos további részletek az eszköz dokumentációjából kikereshetőek.

4. Sebesség

Az állandó memórián végrehajtott műveletek rendszerint több időt vesznek igénybe, mint az ezekkel ekvivalens felejtő (nem állandó) memóriákban végzett műveletek. Az adatok írása különösen néhány platformon kerülhetnek túl sok időbe. A jobb teljesítményért, gyorsító-tárban kerül elhelyezésre a felejtő memória gyakran használt adatai, hogy frissen tartsa a felhasználói interfészt, ezért itt nem hajt végre RMS műveleteket, nem foglal le a MIDlet eseményszálaiból.

5. Szál biztonság

Az RMS műveletek szálvédettek, de a szálaknak mindig ugyanahhoz a rekordtárolóhoz kapcsolódóan kell az adatok olvasását és írását végezniük, mint amelynél a megosztott erőforrások kezelését bonyolítják. Ezek az összefüggő feladatok azokra a szálakra vonatkoznak, amelyek futtatják a különböző MIDlet-eket, és melyek rekordtárolókat osztanak meg ugyanazon a MIDlet programon belül.

6. Kivételek

Általában a módszerek a RMS API-ban találhatóak és egy vagy több ellenőrzött kivételt dobnak, hozzáadódva általános futásidőjű kivételekhez, mint például a *java.lang.IllegalArgumentException*. A RMS kivételek mind a *javax.microedition.rms* csomag részét képzik:

- *InvalidRecordIDExceptiont* dob egy kivételt, amikor egy műveletet nem hajthatnak végre mert a rekord ID érvénytelen.
- *RecordStoreFullExceptiont* dob egy kivételt, amikor több hely nem elérhető a rekordtárolóban.
- *RecordStoreNotFoundExceptiont* kivételt dobják, amikor az alkalmazás megpróbál megnyitni egy nemlétező rekordtárolóot.
- *RecordStoreNotOpenExceptiont* dob, amikor egy alkalmazás megpróbál hozzáférni egy olyan rekordtárolóhoz, ami már zárva van.
- *RecordStoreException* a másik négy szuperosztálya, ez olyan általános hibák esetén kerül meghívásra, amelyek nincsenek lefedve.

Az egyszerűség kedvéért, itt nem kerül minden kivétel lekezelésre.

7. Az RMS használata

A továbbiakban az alapvető rekordműveletek leírása következik, amik a RMS API-t használják. A műveletek közül néhány bemutatásra kerül egy segédosztály fejlesztésén keresztül (*RMSAnalyzer*), melyen rekordtárolói működését elemezhetjük. Saját projektjeidben is fel tudod használni a *RMSAnalyzert* hibaelhárítási segédeszközként.

8. Rekordtároló vizsgálata

A *RecordStore.listRecordStores()* függvény használható arra, hogy lekérdezzük a rekordtárolók listáját egy MIDlet programban. Ez a statikus metódus karaktersorozatok tömbjével tér vissza, ahol mindegyik sztring egy rekordtároló nevét képviseli, ami a MIDlet program tulajdona. Ha nincsenek rekordtárolók, a visszatérés értéke *null*.

Az *RMSAnalyzer analyzeAll()* metódus arra használja *listRecordStores()*gyűjteményt, hogy listázza a rekordtárolók neveit az *analyze()* metódust meghívva.

```
public void analyzeAll(){
   String[] names = RecordStore.listRecordStores();

   for( int i = 0;
        names != null && i < names.length;
        ++i ){
       analyze( names[i] );
   }
}
```

Vegyük észre, hogy a tömb a *MIDlet program* csak a *saját* rekordtárolóinak az azonosítóit adja vissza; vagyis, amely létrehozta azokat. A MIDP specifikációk nem adnak semmilyen módot arra, hogy felsorolják a másik MIDlet programok rekordtárolóit. MIDP 1.0 rekordtároló nem látható az őt tartalmazó programon kívül egyáltalán. MIDP 2.0 programban megoszthatóként jelölhet ki egy rekordtárolót, de más MIDlet programok csak akkor tudják használni, ha ismerik a nevét.

9. Rekordtárolók nyitása, zárása

RecordStore.openRecordStore() függvény a rekordtároló megnyitására, valamint opcionálisan a létrehozására szolgál. Ez a statikus metódus visszaadja a RecordStore objektum egy példányát, amint az látható *RMSAnalyzer.analyze()* metódusban:

```
public void analyze( String rsName ){
   RecordStore rs = null;

   try {
      rs = RecordStore.openRecordStore( rsName, false );
      analyze( rs ); // túlcsordulást ellenőrző metódus meghívása
   } catch( RecordStoreException e ){
      logger.exception( rsName, e );
   } finally {
      try {
         rs.closeRecordStore();
      } catch( RecordStoreException e ){
         // kivétel elvetése
      }
   }
}
```

Az *openRecordStore()* második paramétere jelzi, hogy a rekordtárolót létre kell-e hozni vagy sem, abban az esetben, ha az még nem létezik. A MIDP 2.0 használatánál, az egyik MIDlet

program megnyithatja egy másik rekordtárolóját, ekkor az *openRecordStore()* egy másik fajtájára lesz szükség:

```
...
String name = "mySharedRS";
String vendor = "toldigabor";
String suite = "Tesztprogram";
RecordStore rs =
     RecordStore.openRecordStore( name, vendor, suite );
...
```

A vendor- és a programneve egyben határozzák meg a MIDlet programot.

Amikor a rekordműveletek befejeződtek, az adatok eltárolása a rekordtároló bezárásával realizálódik. Ezt az eljárást a *RecordStore.closeRecordStore()* metódus meghívásával kell elvégezni, mint ahogy az *analyze()* függvényben is látható.

A *RecordStore* példánya egyedi egy MIDlet programon belül: az *openRecordStore()* által, ugyanazzal a névvel nyitott objektum esetében, következő hívásnál, ugyanazt az objektumreferenciát küldi vissza. Ezt a példányt osztja meg minden MIDlet a MIDlet programban.

Mindegyik rekordtároló figyelemmel kíséri, a program futási ideje alatt, hogy az objektumpéldányt hányszor nyitották meg. A rekordtároló nincs ténylegesen zárva, amíg *closeRecordStore()* függvény meghívásra nem kerül. A rekordtároló lezárása után *RecordStoreNotOpenExceptiont* kivételt dob.

10.Rekordtároló létrehozása

Egy saját rekordtárolót létrehozni, *openRecordStore()* metódus második paraméterének az igaz értékre történő állításával lehet.

```
...
// Rekordtároló létrehozása
RecordStore rs = null;

try {
   rs = RecordStore.openRecordStore( "myrs", true );
} catch( RecordStoreException e ){
   // megnyitás sikertelensége esetén
}
...
```

A rekordtároló inicializálásakor egy időben kerülhet végrehajtásra az a vizsgálat, amely ellenőrzi, hogy *getNextRecordID()* értéke egyenlő-e 1-gyel. Ez közvetlenül a rekordtároló megnyitás után történhet:

```
if( rs.getNextRecordID() == 1 ){
   // inicializálás egyszeri végrehajtása
}
```

Másik megoldás az inicializáláskor, hogy a rekordtároló üres voltát vizsgáljuk a *getNumRecords()* függvény segítségével:

```
if( rs.getNumRecords() == 0 ){
   // rekordtároló üres állapota esetén újra inicializálás
}
```

Létrehozható egy megosztott rekordtároló (csak MIDP 2.0-nál), az *openRecordStore()* négy-paraméteres változatát használva:

```
int authMode = RecordStore.AUTHMODE_ANY;
boolean writable = true;

rs = RecordStore.openRecordStore( "myrs", true, authMode, writable );
```

Amikor a második paraméter igaz és a rekordtároló már nem létezik, az utolsó két paraméter határozza meg az engedélyeket és az írhatóságát. Az engedélyeztetési mód dönti el, hogy más MIDlet programoknak lesz-e hozzáférésük a rekordtárolóhoz. A két lehetséges mód *RecordStore AUTHMODE_PRIVATE* (csak a tulajdonos MIDlet program férhet hozzá) és *RecordStore AUTHMODE_ANY* (bármilyen MIDlet program hozzáférhet). Az írhatóság jelző dönti el, hogy más MIDlet programok eszközölhetnek-e változásokat a rekordtárolón -- ha hamis, csak olvasni tudják a rekordtárolót.
A *RecordStore.setMode* futás közben is meg tudja változtatni a rekordtároló engedélyeit:

```
rs.setMode( RecordStore.AUTHMODE_ANY, false );
```

Érdemes a rekordtárolót *AUTHMODE_PRIVATE* módban létrehozni és csak aztán megosztani, miután azt inicializálták.

11.Rekordok hozzáadása és módosítása

Emlékezhetünk, hogy a rekordok tulajdonképpen bájttömbök. A *RecordStore.addRecord()* függvényt alkalmazható arra, hogy hozzáadjunk egy új rekordot egy nyitott rekordtárolóhoz:

```
...
byte[] data = new byte[]{ 0, 1, 2, 3 };
int recordID;

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

Egy üres rekordot úgy tudunk hozzáadni egy rekordtárolóhoz, hogy a metódus első paraméterének értékét null értékre állítjuk. A második és harmadik paraméterek leírják az eltolás kezdetet a tömbben és az összbájtok számát az eltolástól számítva a tárolóban. Sikeres tárolós esetén visszatér az új rekord azonosítójával, különben egy *RecordStoreFullException* kivételt dob.

A *RecordStore.setRecord()* használatával bármikor módosítható egy rekord.

```
...
int    recordID = ...; // néhány rekordazonosító
byte[] data = new byte[]{ 0, 10, 20, 30 };

rs.setRecord( recordID, data, 1, 2 );
   // összes adatrekord elhelyezése
...
```

Nem lehet egy rekordot hozzáadni vagy frissíteni egy az egyben: először egy bájttömbben az teljes rekordot a memóriában kell elhelyezni, aztán pedig hozzáadni vagy frissíteni a fenti függvényhívásokkal.

Kitalálható, hogy milyen rekordazonosítót fog visszaadni a *RecordStore.getNextRecordID()* függvény az *addRecord()* metódus meghívása után. Minden aktuális rekordazonosító ennél az értéknél kisebb lesz.

A következőkben megnézzük, hogy hogyan történik az objektumok bájttömbökké alakítása.

12.Rekordok olvasása

Egy rekord olvasásához, használd *RecordStore.getRecord()* két fajtájának az egyikét. Az első lefoglalja a szükséges méretet a bájttömb számára és belemásolja rekordadatokat:

```
...
int recordID = .... // néhány rekordazonosító
byte[] data = rs.getRecord( recordID );
...
```

A második alak bemásolja az adatokat egy előre lefoglalt tömbbe, egy jól meghatározott eltolási ponttól kezdve és visszatér a másolt bájtok számával:

```
...
int recordID = ...; // néhány rekordazonosító
byte[] data = ...; // tömb definiálása
int offset = ...; // kiindulási offset (eltolás)

int numCopied = rs.getRecord( recordID, data, offset );
...
```

A tömbnek elég nagynak kell lennie, hogy el tudja tárolni az adatokat, különben *java.lang.ArrayIndexOutOfBoundsExceptiont* kivételt fog dobni. A *RecordStore.getRecordSize()* által visszaadott érték használható arra, hogy egy olyan tömböt foglaljon le, amely elégendően nagy. Tulajdonképpen, *getRecord()* első változatával ekvivalens:

```
...
byte[] data = new byte[ rs.getRecordSize( recordID ) ];
rs.getRecord( recordID, data, 0 );
...
```

A második alak a memóriafoglalások minimalizálásához hasznosak, mikor cikluson keresztül állítunk be rekordértékeket. Például a *getNextRecordID()* és *getRecordSize()* metódusokat együtt használva teljes körű keresést lehet alkalmazni, minden rekordra a rekordtárolóban:

```
...
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 ); // feldolgozás
    } catch( InvalidRecordIDException e ){
        // hiba esetén, következő rekord
    } catch( RecordStoreException e ){
        handleError( rs, id, e ); // hiba rutin meghívása
    }
}
...
```

Egy jobb megközelítése a problémának azonban a *RecordStore.enumerateRecords()* iterátor használatával feldolgozni a rekordokat. Az *enumerateRecords()* használatára a következőkben láthatunk majd példát.

13.Rekordok, rekordtárolók törlése

Rekordot törölni a *RecordStore.deleteRecord()* metódussal lehet

```
...
int recordID = ...; // néhány rekordazonosító
rs.deleteRecord( recordID );
...
```

Bármilyen rekordtörlési kísérlet esetén *InvalidRecordIDException* kivétel dobása történik, ha nemlétező rekordazonosítóra hivatkozunk.
Magának a rekordtárolónak a törlése a *RecordStore.deleteRecordStore()* kiadásával végezhető:

```
...
try {
    RecordStore.deleteRecordStore( "myrs" );
} catch( RecordStoreNotFoundException e ){
    // nincs ilyen rekordtároló
} catch( RecordStoreException e ){
    // megnyitott állapotban van
}
...
```

Egy rekordtárolót csak akkor törölhetnek, ha az jelenleg nem nyitott, és csak egyetlen MIDlet program tulajdona.

14.Egyéb műveletek

Maradt még néhány RMS művelet, amelyek mindegyike a *RecordStore* osztály módszerei:

- *getLastModified()* visszaadja a rekordtároló utolsó módosításának idejét ugyanabban a formátumban, mint ahogy azt a *System.currentTimeMillis()* metódus adja vissza.
- *getName()* visszaadja a rekordtároló nevét.
- *getNumRecords()*a rekordok számát adja vissza a rekordtárolóban.
- *getSize()* bájtokban küldi vissza a rekordtároló teljes méretét. A teljes méret tartalmazza az összes rekord méretét, valamint azon felül a rendszer által megkövetelt szükséges rekordtároló méretet.
- *getSizeAvailable()* visszaküldi azon bájtok számát, amelyek rendelkezésre állnak a rekordtároló méretének növeléséhez. Vegyük észre, hogy a rendelkezésre álló aktuális méret kevesebb is lehet mint szükséges.
- *getVersion()* visszaküldi a rekordtároló verziószámát. A verziószám egy pozitív egész, 0-tól nagyobb szám, ami növekszik minden alkalommal, amikor rekordtárolót megváltoztatják.

Egy MIDlet program szintén tudja követni a rekordtárolót ért változásokat, figyelő beiktatásával: *addRecordListener()*, amit később kiiktathatunk: *removeRecordListener()* . A figyelőkről még a későbbiekben szó lesz.

15.Az RMSAnalyzer osztály

Most pedig nézzük meg a *RMSAnalyzer* osztályt, amelyben a rekordtárolónkat elemezhetjük:

```
...
RecordStore rs = ...; // rekordtároló megnyitása
RMSAnalyzer analyzer = new RMSAnalyzer();
analyzer.analyze( rs );
...
```

Alapértelmezésben az elemzés eredménye a *System.out* kimenetre megy.

Ezt a formációt akkor érdemes használni, amikor J2ME Wireless Toolkittal tesztelsz. Egy konkrét készüléken való tesztelés során az elemzést elküldheted egy soros portra vagy a hálózaton keresztül egy szervletnek. Definiálhatsz egy saját osztályt, amely implementálja a

RMSAnalyzer.Logger interfészt, és az RMSAnalyzer osztály példányának a konstruktorát kicseréled.

A programot a J2ME Wireless Toolkit segítségével tekinthetjük meg, az abban létrehozott RMSAnalyzerTest projekttel, amely demonstrálja az elemző használatát:

A forráskódot az 1. sz. melléklet tartalmazza.

III. Adatleképzés

Mint ahogy az eddigiekben láthattuk, a Mobile Information Device Profile (MIDP) gondoskodik adattárolásról a Record Management Systemen (RMS) keresztül. A MIDP által nyújtott tárolási támogatás egyszerű bájttömbökre van korlátozva, így az olvasási és írási műveletek a teljes rekordra értelmezettek, azaz mezőnként ezeket nem lehet végezni. Az RMS alkalmazási programozási-interfésze (API) nagyon egyszerű, így az alkalmazásaiban is, egy egyszerű bináris formátumot használ adattárolásra.
Ezek után, olyan adattárolási stratégiákkal ismerkedünk meg, amelyekkel alacsony szintű raktározási műveleteket építhetünk be a programunkba, úgyhogy valóban hatékony műveletvégzést biztosíthatunk.

1. Alapvető adatmanipulációs osztályok

A rekordtárolóba írott adatok és azon adatcsomagok között, amelyet egy hálózaton keresztül küldünk egy szervernek, valójában nincs különbség. A Connected Limited Device Configuration (CLDC), amelyen a MIDP is alapul, tartalmaz általános adatmanipulációs osztályokat, amelyek a Java 2 Platform alapkönyvtárából származnak - Standard Edition (J2SE) – ezek jól használhatóak az RMS műveleteknél. Egyik nagy előnye annak a közös interfészek használatának, hogy a MIDlet-ek könnyebben tudnak műveletmegosztásokat végezni egymás között, mind a standard, mind a vállalati java platformokon.

2. Bájt-tömb folyamok

Egy *ByteArrayInputStream* objektum áttranszformál egy bájttömböt egy bemeneti folyammá, ahogy ez a triviális példa is mutatja:

```
...
byte[] data = new byte[]{ 1, 2, 3 };
ByteArrayInputStream bin = new ByteArrayInputStream( data );

int b;

while( ( b = bin.read() ) != -1 ){
   System.out.println( b );
}
```

```
try {
   bin.close();
} catch( IOException e ){
   // sohasem dobja ezt a kivételt
}
...
```

A bemeneti folyam sorozatosan visszaadja mindegyik bájtot a tömbben, amíg eléri a tömb végét. Használva *mark()* és *reset()* metódusokat, bármikor újra tudunk pozícionálni a bájttömbön belül.

Egy *ByteArrayOutputStream* objektum elhelyezi a memóriapuffer adatait egy bájttömbbe, későbbi átalakítás céljából:

```
...
ByteArrayOutputStream bout = new ByteArrayOutputStream();

bout.write( 1 );
bout.write( 2 );
bout.write( 3 );

byte[] data = bout.toByteArray();

for ( int i = 0; i < data.length; ++i ){
   System.out.println( data[i] );
}

try {
   bout.close();
} catch( IOException e ){
   // sohasem dobja ezt a kivételt
}
...
```

A *ByteArrayOutputStream* puffere automatikusan nő az adatok adatfolyamon történő írása közben. A *toByteArray()* metódus a fogadott bájtokat bemásolja bájttömbbe. Ismét felhasználhatjuk a belső puffert a későbbiekben a *reset()* kiadásával.

3. Adatfolyamok

DataInputStream áttranszformál egy nyers bemeneti folyamot primitív adattípusú karakterláncokká:

```
...
InputStream in = ... // beviteli folyam
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 ){
   // hiba lekezelése
}
...
```

Az adatot csak akkor olvashatjuk ilyen módon, ha hardverfüggetlen formában írták a folyamra, úgy ahogyan arra a *DataInputStream* számít. Az osztálynak beépített metódusai vannak arra, hogy legegyszerűbb java adattípusokat képes legyen olvasni: *readBoolean(), readByte(), readChar(), readShort(), readInt()*, és *readLong()* . A CLDC 1.1 implementációk támogatják továbbá *readFloat()* és *readDouble()* módszereket. Vannak tovább metódusok az bájttömbök olvasására és előjelnélküli értékekhez: *readFully(), readUnsignedByte()* és *readUnsignedShort().*

A *readUTF()* módszer képes a több mint 65,535 karakterből álló sztringeket olvasni, ami UTF-8 formátum kódolású. Egy olyan karaktersorozat olvasásánál, amit 2 bájtonként lett megírva, az alkalmazásnak több menetben kell azt feldolgoznia a *readChar()* függvénnyel, ami egy határoló-jelet keres a karaktersorozat végén, mivel annak pontos hosszát nem tudja máshogy megállapítani. A hossz lehet egy rögzített érték vagy be lehet írva közvetlenül a sztring elé.

Az alkalmazásnak, ami az adatokat olvassa, ismernie kell azokat a primitíveket, amelyek a helyes módszereket használatához szükségesek.

DataOutputStream sztringeket és egyéb primitív adattípusokat ír ki egy kimeneti folyamra:

```
...
OutputStream out = ... // kiviteli folyam
DataOutputStream dout = new DataOutputStream( out );

try {
   dout.writeInt( 100 );
   dout.writeUTF( "Alig" );
```

```
   dout.writeUTF( "Lehel" );
   dout.writeLong( System.currentTimeMillis() );

   dout.close();
}
catch( IOException e ){
   // hiba lekezelése
}
...
```

Az adatokat ugyanúgy platform-független formában várja, mint a *DataInputStream*. Az osztálynak vannak metódusai legegyszerűbb java adattípusokat írására: *writeBoolean(), writeByte(), writeChar(), writeShort(), writeInt()* , és *writeLong()* . A CLDC 1.1 implementációk továbbá támogatják még *writeFloat()* és *writeDouble()* metódusokat is.
A karakterláncok írásához két módszer is használható. A több mint 65,535 karakterből álló sztringekhez a *writeUTF()* függvény alkalmazandó, ami UTF-8 formátum kódolású, vagy hívható még a *writeChars()* függvény, amellyel sorozatonként 2 bájtos karaktereket lehet olvasni.

4. Alapvető adatmegfeleltetések

Az általános adatmanipulációs osztályok egyszerűvé tesznek elemi adatmegfeleltetéseket. Adatok írása egy rekordtárolóba egyszerűen megoldható egy *DataOutputStream* és egy *ByteArrayOutputStreammel* folyamok kombinálásával, amely során a tárolás eredménye egy bájttömbbe kerül:

```
...
RecordStore rs = ... // rekordtároló definiálása
ByteArrayOutputStream bout = new ByteArrayOutputStream();
DataOutputStream dout = new DataOutputStream( bout );

try {
   dout.writeInt( 100 );
   dout.writeUTF( "Alig" );
   dout.writeUTF( "Lehel" );
   dout.writeLong( System.currentTimeMillis() );
   dout.close();

   byte[] data = bout.toByteArray();
   rs.addRecord( data, 0, data.length );
}
catch( RecordStoreException e ){
```

```
   // RMS kivétel lekezelése
}
catch( IOException e ){
   // IO hiba lekezelése
}
...
```

Az adatok olvasásánál pont fordított lépéseket kell tennünk, használva a *DataInputStreamet* és a *ByteArrayInputStreamet* folyamokat:

```
...
RecordStore rs = ... // rekordtároló definiálása
int recordID = ... // rekordazonosító definiálása rekordolvasáshoz
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();

   ... // adatfeldolgozás
}
catch( RecordStoreException e ){
   // RMS hiba lekezelése
}
catch( IOException e ){
   // IO hiba lekezelése
}
...
```

Figyeld meg a null értékeket karakterláncok írásánál. Sokszor egy üres sztringet használnak helyette. Ha nem, akkor jelzőbájtokra van szükség, hogy meg tudj különböztetni egy üres karaktersorozatot egy üres karakterlánctól.

5. Névszói objektum megfeleltetések

A gyakorlatban a legtöbb esetben a mentésre váró adatot beágyazzák egy objektumpéldányba. Nézzünk egy példát: a *Contact* osztály, amely információkat tart nyilván egy személyről:

```
// Információk egy ismerős személyről.

public class Contact {
   private String _firstName;
   private String _lastName;
   private String _phoneNumber;

   public Contact(){
   }

   public Contact( String firstName, String lastName,
                   String phoneNumber )
   {
      _firstName = firstName;
      _lastName = lastName;
      _phoneNumber = phoneNumber;
   }

   public String getFirstName(){
      return _firstName != null ? _firstName : "";
   }

   public String getLastName(){
      return _lastName != null ? _lastName : "";
   }

   public String getPhoneNumber(){
      return _phoneNumber != null ? _phoneNumber : "";
   }

   public void setFirstName( String name ){
      _firstName = name;
   }

   public void setLastName( String name ){
      _lastName = name;
   }

   public void setPhoneNumber( String number ){
      _phoneNumber = number;
   }
}
```

Ha az osztály módosítható, akkor a legegyszerűbb útja az objektum leképezésének és a bájttömbadatok megtekintésének, ha az osztályhoz metódusokat adunk:

```
public void fromByteArray( byte[] data ) throws IOException {
   ByteArrayInputStream bin = new ByteArrayInputStream(data);
   DataInputStream din = new DataInputStream( bin );

   _firstName = din.readUTF();
   _lastName = din.readUTF();
   _phoneNumber = din.readUTF();

   din.close();
}

public byte[] toByteArray() throws IOException {
   ByteArrayOutputStream bout = new ByteArrayOutputStream();
   DataOutputStream dout = new DataOutputStream( bout );

   dout.writeUTF( getFirstName() );
   dout.writeUTF( getLastName() );
   dout.writeUTF( getPhoneNumber() );

   dout.close();

   return bout.toByteArray();
}
```

Az objektum tárolása így egyszerű:

```
...
Contact contact = ... // Ismerős adatait hordozó objektum
RecordStore rs = ... // és az azt hordozó tároló definiálása.

try {
   byte[] data = contact.toByteArray();
   rs.addRecord( data, 0, data.length );
}
catch( RecordStoreException e ){
   // RMS hiba lekezelése
}
catch( IOException e ){
   // IO hiba lekezelése
}
...
```

Az objektum visszanyerése is igazán könnyű:

```
...
RecordStore rs = ... // rekordtároló definiálása
int recordID = ... // rekordazonosító definiálása olvasáshoz
Contact contact = new Contact();

try {
   byte[] data = rs.getRecord( recordID );
   contact.fromByteArray( data );
}
catch( RecordStoreException e ){
   // RMS hiba lekezelése
}
catch( IOException e ){
   // IO hiba lekezelése
}
...
```

Ha az osztály nem módosítható, mint például a standard *Vector* vagy *Hashtable* osztály, úgy neked kell egy segítő osztályt írnod. Például itt van egy olyan segítőosztály, ami feltérképezi a nem null hosszúságú sztringeket és egy listát készít róluk egy bájttömbbe:

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

// Segítő osztály, amely konverziót végez, sztringvektorokat
// (vektorok, amelyeknek elemei sztringek vagy sztringbufferek)
// transzformálva bájttömbökké és vissza.

public class StringVectorHelper {

   // Átalakít vektorrá egy bájttömböt.

   public Vector fromByteArray( byte[] data )
                        throws IOException {
      ByteArrayInputStream bin =
            new ByteArrayInputStream( data );
      DataInputStream din = new DataInputStream( bin );

      int count = din.readInt();
      Vector v = new Vector( count );

      while( count-- > 0 ){
         v.addElement( din.readUTF() );
      }

      din.close();
```

```
      return v;
   }

   // Transzformál egy vektort bájttömbe.

   public byte[] toByteArray( Vector v )
                        throws IOException {
      ByteArrayOutputStream bout =
               new ByteArrayOutputStream();
      DataOutputStream dout = new DataOutputStream( bout );

      dout.writeInt( v.size() );

      Enumeration e = v.elements();
      while( e.hasMoreElements() ){
         Object o = e.nextElement();
         dout.writeUTF( o != null ? o.toString() : "" );
      }

      dout.close();

      return bout.toByteArray();
   }
}
```

Nem szükséges a fent említettekkel foglalkoznod, ha csak primitív Java adattípusokat vagy egyszerű, önálló objektumokat akarsz tárolni. A fromByteArray() és toByteArray() módszerek, amik eddig bemutatásra kerültek, egyszerű, de hatásos eszközök az objektumtárolásra.

Szintén használhatóak arra ezek a technikák, hogy objektumokat másoljunk hálózaton keresztül: Amint ismert a célgép az objektumot bájttömbbé konvertálva egyszerűen el lehet küldeni a másik eszköznek vagy egy külső szervernek egy hálózati összeköttetésen keresztül, majd a megérkezett bájtsorozatot ismét objektumba lehet szervezni a célállomáson.

Az, hogy ugyanaz a technika használható két sok mindenben különböző területen, annak köszönhető, hogy például, azok az adatosztályok, amiket J2ME alkalmazásokra fejlesztettek, azok könnyen használhatóak szervletek készítésénél is, azaz a Java 2 - Enterprise Edition (J2EE) platformon is.

6. Adatfolyamok használata

Az eddigi példák közvetlenül bájttömbökre vonatkoztak. Kényelmesen el lehetett elvégezni egyszerű adathalmazok mentését és visszaállítását, azonban a nyers bájttömbök nehezen-kezelhetővé válnak, amint szekvenciális adathalmazokat akarsz tárolni, például objektumok egy listáját. Egy jobb megközelítésben érdemes elválasztani a bájttömbök kezelését, az adatok olvasásától és írásától. Például hozzá tudjuk adni ezeket a módszereket a Contact osztályunkhoz:

```
public void fromDataStream( DataInputStream din )
                    throws IOException {
   _firstName = din.readUTF();
   _lastName = din.readUTF();
   _phoneNumber = din.readUTF();
}

public void toDataStream( DataOutputStream dout )
                    throws IOException {
   dout.writeUTF( getFirstName() );
   dout.writeUTF( getLastName() );
   dout.writeUTF( getPhoneNumber() );
}
```

Megtarthatjuk a már létező fromByteArray és toByteArray metódusokat, de felül is írhatjuk őket új folyam-orientált metódusokkal:

```
public void fromByteArray( byte[] data ) throws IOException {
   ByteArrayInputStream bin = new ByteArrayInputStream(data);
   DataInputStream din = new DataInputStream( bin );

   fromDataStream( din );
   din.close();
}

public byte[] toByteArray() throws IOException {
   ByteArrayOutputStream bout = new ByteArrayOutputStream();
   DataOutputStream dout = new DataOutputStream( bout );

   toDataStream( dout );
   dout.close();

   return bout.toByteArray();
}
```

Összpontosítva az adatok írási és olvasási műveleteire, biztosított az állandó konzisztencia.

Ebben a sorozatrészben megtanulhattunk néhány adatmegfeleltetési technikát. A következőkben megismerkedünk több kifinomultabb megközelítéssel, a perzisztens adatkezelés területén. Ezek olyan objektumok tárolását és visszanyerését teszik lehetővé, amelyek változó típusú, összetett mezőkből állnak és amik bele is építhetők alkalmazásainkba.

IV. Adatleképzés kidolgozása

Az előzőekben megnéztük, hogy hogyan működik a primitív típusok adatainak leképzése. Megtanulhattuk ezek értékeinek bájttömbökben való tárolását. Megismerkedhettünk az objektumfolyamok tárolásával és azok visszakeresésével, és hogy hogyan tudjuk előhívni a tárolt értékeket a bájttömbből. A következőkben megvizsgáljuk, hogy hogyan lehet elhatárolni az alkalmazásainkban a magasabb szintű műveleteket az alacsonyabb szintű műveletektől és belehelyezni a kiterjesztett alaposztályba, amely tartalmazza az olvasási és írási műveleteket és mezőlistákat létrehozó metódusokat, valamint azokat a metódusokat, amelyek az objektumok eltárolását és visszanyerését végzik.

1. Az alaposztályok kiterjesztése

J2ME fejlesztőként célod kell, hogy legyen a memória-felhasználás minimalizálása. Ideális esetben egy adott időben csak egy adatrészletnek kellene lennie a memóriában. Amikor adatok kerülnek átadásra írás céljából a *ByteArrayOutputStream*-nek, ilyenkor nincs hozzáférési lehetőség a mögötte lévő bájttömbhöz, így meg kell hívni a *toByteArray()* metódust, amely a tömb egy másolatával tér vissza. Ez egy fölösleges szemét lesz a memóriában, ha eltárolod a rekordtárolóba. A tömbhöz való közvetlen hozzáférést és az abba való írást végzi a *ByteArrayOutputStream* kiterjesztéseként létrehozott *DirectByteArrayOutputStreamnek* osztály:

```
import java.io.*;

// ByteArrayOutputStream egy verziója, amely direkt hozzáférést
// biztosít számodra egy bájttömbhöz, ha szükséges

public class DirectByteArrayOutputStream
                    extends ByteArrayOutputStream {

   // Létrehoz egy kimeneti bájttömböt alapértelmezett mérettel.

   public DirectByteArrayOutputStream(){
      super();
   }

   // Létrehoz egy kimeneti bájttömböt megadott mérettel.

   public DirectByteArrayOutputStream( int size ){
```

```
        super( size );
    }

    // Visszatér a bájttömb hivatkozásával.

    public synchronized byte[] getByteArray(){
        return buf;
    }

    // Kicseréli az új bájttömbbel a régit
    // és visszaállítja a helyes darabszámot

    public synchronized byte[] swap( byte[] newBuf ){
        byte[] oldBuf = buf;
        buf = newBuf;
        reset();
        return oldBuf;
    }
}
```

Ügyelni kell a *size()* metódus hívásakor, hogy egyértelmű legyen, hogy ténylegesen került-e elraktározásra adat a bájttömbben:

```
...
DirectByteArrayOutputStream bout = ...
RecordStore rs = ...

int numBytes = bout.size();
byte[] data = bout.getByteArray();

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

A következetesség kedvéért, hasonlóan ki lehet terjeszteni a *ByteArrayInputStream* osztályt is, bár ez nem lesz olyan nélkülözhetetlen osztály, mint a másik:

```
import java.io.*;

// ByteArrayInputStream egy verziója, amely direkt hozzáférést
// biztosít számodra egy bájttömbhöz, ha szükséges.

public class DirectByteArrayInputStream
                extends ByteArrayInputStream {

    // Létrehoz egy kimeneti folyamot egy megadott tömbből.

    public DirectByteArrayInputStream( byte buf[] ){
```

```
      super( buf );
   }

   // Létrehoz egy kimeneti folyamot egy megadott résztömbből.

   public DirectByteArrayInputStream( byte buf[], int offset, int length ){
      super( buf, offset, length );
   }

   // Visszaállítja a folyamot a tömbbel.

   public synchronized void setByteArray( byte[] buf ){
      this.buf = buf;
      this.pos = 0;
      this.count = buf.length;
      this.mark = 0;
   }

   // Visszaállítja a folyamot a tömbbel.

   public synchronized void setByteArray( byte[] buf, int offset, int length ){
      this.buf = buf;
      this.pos = offset;
      this.count = Math.min( offset + length, buf.length );
      this.mark = offset;
   }
}
```

Vedd észre, hogy *ByteArrayInputStream* és *ByteArrayOutputStream* osztályok kiterjesztéseiben, szerepelnek metódusok, amelyek a szinkronizálást segítik elő, ezzel ügyelve a szálak biztonságára. Azonban a legtöbb esetben, a folyamok csak egyetlen szálat használnak, így végső soron a szinkronizálás fölösleges. Ha az alkalmazásodban sok írási és olvasási művelet történik, megfontolandó egy szinkronizálatlan osztály létrehozása, amely egy kevés sebességnövekedéshez vezet.

Könnyen létre lehet hozni más, további osztályokat is a *DataInputStream* és a *DataOutputStream* kiterjesztésével. Ha például gyakran kerül sor egész számok írására, akkor létrehozható *DataOutputStreamnek* egy ilyenfajta kiterjesztése:

```
import java.io.*;

public class ExtendedDataOutputStream extends DataOutputStream {
   public ExtendedDataOutputStream( OutputStream out ){
      super( out );
   }
```

```
    public final void writeIntArray( int[] arr )
                                throws IOException {
        int size = arr.length;
        writeInt( size );
        for( int i = 0; i < size; ++i ){
            writeInt( arr[i] );
        }
    }
}
```

2. Rekordmezők létrehozása

Így most már megvan minden eszközöd egy mezőközpontú rekordtároló létrehozására, amelyben minden rekord meghatározott típusú és nevű mezők halmazából épül fel. Két osztály segítségével kezelheted a rekordtárolót. Az első a *FieldList,* amely magukra a mezőkre vonatkozó információk felügyeletét végzi:

A forráskódot a 2. sz. melléklet tartalmazza.

A *FieldList* osztály két központi eleme, két tömb. Az első tömb tárolja a mezők típusait, a második pedig a mezők neveit. A nevek szabadon választhatóak; a típusok azonban nem, mivel ezek fogják meghatározni az adatok olvasásának és írásának módját. Minden szabványos java primitív adattípust támogat, plusz a *String* típust. Íme a mezőobjektumok listájának szervezett létrehozása:

```
...
FieldList depts = new FieldList( 3 );
depts.setFieldType( 0, FieldList.TYPE_SHORT );
depts.setFieldName( 0, "Azonosító" );
depts.setFieldType( 1, FieldList.TYPE_STRING );
depts.setFieldName( 1, "Név" );
depts.setFieldType( 2, FieldList.TYPE_INT );
depts.setFieldName( 2, "Vezető azonosító" );
...
```

A *FieldList* egy példányát elraktározhatjuk egy adatfolyamban, egy bájttömbben, vagy egy rekordtárolóban. A mezőlistát el lehet menteni az első rekordként a rekordtárolóban, így bármilyen kód, ami megnyitja a rekordtárolót, kiolvashatja belőle a rekordszerkezetet. Ehhez a rekordtárolási stratégiához, tehát el kell tárolni a mezőlistát, valahogy így:

```
...
FieldList list = ... // mezőlista definíció
RecordStore rs = RecordStore.openRecordStore( "foo", true );
```

```
if( rs.getNumRecords() == 0 ){ // üres tároló
   list.toRecordStore( rs, -1 );
}
...
```

A *toRecordStore()* módszer második paramétere azonosítja be az adatszerkezetet tároló rekordot. A negatív érték azt mutatja, hogy ez a leírórekord.

A másik szükséges osztály egy mezőalapú rekordtároló kezelő osztály. A *FieldBasedStore* osztályba csomagoljuk a tárolással kapcsolatos műveleteket, az olvasást és az írást:

A forráskódot a 3. sz. melléklet tartalmazza.

A *FieldBasedStore* létrehozásához szükség lesz egy nyitott *RecordStore* példányra és egy *FieldList* példányra. A *FieldList* később kiolvasható magából a rekordtárolóból impliciten. A *FieldBasedStore* létrehozási folyamata:

```
...
RecordStore rs = ... // nyitott rekordtároló
FieldBasedStore fstore = new FieldBasedStore( rs );
...
```

Vagy meghatározható expliciten is:

```
...
RecordStore rs = ... // nyitott rekordtároló
FieldList list = ... // mezőlista
FieldBasedStore fstore = new FieldBasedStore( rs, list );
...
```

A *FieldBasedStore* mindegyik rekordot objektumtömbként kezeli. A típusok a tömbben összeillesztésre kerülnek a mezőlistában leírt típusokkal. A korábban bemutatott lista osztály tartalmazott egy osztály azonosítót, egy osztálynevet és egy kezelő azonosítót. Így lehet hozzá adni egy rekordot:

```
...
Object[] fields = new Object[]{
   new Short( 1 ), "Accounting", new Integer( 100 )
};

int recordID = fstore.addRecord( fields );
...
```

Mivel a *FieldBasedStore* rekordok írását végző kódja elég intelligens ahhoz, hogy a megfelelő adat-átalakításokat elvégezze, így az objektumok létrehozása egyszerűbben is megoldható.

```
...
Object[] fields = new Object[]{ "1", "Accounting", "100" };
int recordID = fstore.addRecord( fields );
...
```

A rekordok olvasása triviális:

```
...
Object[] fields = fstore.readRecord( recordID );
for( int i = 0; i < fields.length; ++i ){
   System.out.println( "Field: " +
      fstore.getFieldList().getFieldName( i ) +
      " Value: " + fields[i] );
}
...
```

Egy rekord módosítása a tömb megfelelő elemének átírásával történik:

```
...
Object[] fields = fstore.readRecord( recordID );
fields[2] = "134"; // mezőmódosítás
fstore.writeRecord( recordID, fields );
...
```

Íme egy MIDlet példa, ami mezőközpontú rekordtárolókat használ, emberek és részlegek adatait tárolva. Ebben a példában használatra kerül a dolgozat elején létrehozott RMSAnalyzer osztály is.

A forráskódot a 4. sz. melléklet tartalmazza.

A *FieldBasedStore* és *FieldList* osztályok jelen állapotukban nem tökéletesek: némi fejlesztések még eszközölhetőek bennük. A mezőközpontú tárolók, mint például kurzor alapú JDBC eredményhalmaza, egyik rekordról a másik rekordra lépkedve saját mezőértékeket állíthat be. A tárolóba eleve mezőkből álló rekordokat lehet elmenteni. A mezőlista vissza tudja adni egy mező indexét névre történő hivatkozásnál. Ezeket és egyéb más fejlesztéseket bele lehetne még építeni az osztályokba, amelyek megvalósítása természetesen nem egyszerűek, így meg kell fontolni, hogy ezekből a szolgáltatásokból melyekre lesz szükség programunkban. Egy általános mezőközpontú megközelítés a legjobb eleinte, amíg nem tudjuk az adatok szerkezetét pontos struktúráját. Általában a legjobb megoldás, ha becsomagoljuk az ismert objektumtípusokat perzisztens objektumokba.

V. Szűrési és bejárási módszerek

Az előzőekben megtárgyaltuk az alapvető adatmegfeleltetési módokat a rendezett bájttömb-tárolóktól, a Record Management System (RMS) által működtetett rekordtároló kezelőkig. Az adatok olvasásának és írásának helyes megszervezése csak az első akadály legyőzését jelenti. Az adatok megtalálása is épp oly fontos tényező, amelyhez szükséges a rekordtárolóban való navigáció valamint a rekordok egy előre meghatározott szempont szerint való rendezése és különböző szűrők létrehozása, amelyek a kívánt rekordok kiválasztását eredményezik. Ez a rész, ezen utóbbi problémákra igyekszik megoldást nyújtani, míg a következőkben keresésére, kiválasztására összpontosítunk.

1. A rekordazonosító nem ugyanaz, mint az index

Egy rekord beolvasásához szükséges tudni annak rekordazonosítóját (ID). Az első fejezetekben láthattuk, hogy a rekordazonosító egy olyan egész szám, ami egyértelműen beazonosít egy rekordot a rekordtárolóban, azonban ez nem egy index. Ennek a különbségnek van néhány alapvető következménye.

Ha egy tárolóban N darabszámú rekord volt indexelve, akkor mindegyik felvesz egy értéket a 0 és N-1 tartományban vagy 1 és N intervallumban, attól függően, hogy 0-val vagy 1-gyel indult a sorozat. Minden olyan alkalomkor, amikor egy rekord beillesztése vagy törlése történik, a rekordindexek a módosításnak megfelelően változnak. A tartomány bővülhet, illetve szőkülhet, de folytonos marad.

Az indextől eltérően, a rekordazonosító nem változik meg, nem számít, hogy hány rekord kerül beillesztésre, vagy hány rekord kerül törlésre. Az első rekord beillesztésekor kap a rekordtárolótól egy 1-gyes értékű rekordazonosítót, a következőkben egy 2-es értékű rekordazonosítót és így tovább. Ha törlődik egy rekord, akkor annak rekordazonosítója érvénytelenítésre kerül és bármilyen rekord-hozzáférési kísérlet esetén, egy *InvalidRecordIDExceptiont* kivétel dobódik. Az érvényes rekordazonosítók nem maradnak folytonosak.

Mivel egyértelműen beazonosítják a rekordokat, ezért használhatóak arra a rekordazonosítók, hogy két vagy több rekordot összekapcsolva, mint adatértékként tárolható el a rekord

azonosítója. Szintén alkalmazhatóak a rekordazonosítók külső programokkal való adatszinkronizálásra.

A rekordazonosítók használatának fő hátránya, hogy bonyolult a rekordtároló bejárása: nem lehet egyszerűen „végiglépkedni” az indexhalmazon, mint ahogy azt egy tömbön keresztül megtehető. Két bejárási technika alkalmazható: a *burte-force,* illetve a *felsorolás.*

2. Brute-Force bejárás

A Brute-Force bejárásnál egyszerűen lekérdezzük a rekordokat egytől-egyig, kezdve az első rekorddal, átugorva az érvénytelen rekordokat. Ezt mindaddig folytatva, amíg maradtak még rekordok:

```
...
RecordStore rs = ... // rekordtároló megnyitása
int lastID = rs.getNextRecordID();
int numRecords = rs.getNumRecords();
int count = 0;

for( int id = 1; id < lastID && count < numRecords; ++id ) {
   try {
      byte[] data = rs.getRecord( id );
      ... // adatfeldolgozás
      ++count;
   }
   catch( InvalidRecordIDException e ){
      // csak elvetjük és rátérünk a következő rekordra
   }
   catch( RecordStoreException e ){
      // több általános hiba kezelhető itt
      break;
   }
}
...
```

A *getNextRecordID()* szolgáltatja azon rekord rekordazonosítóját, amely a rekordtárolóban a soron következő. Használható ez a függvény egy felső értékhatárolóként is, a lehetséges rekordazonosítók vizsgálatára. A gyűjtemény számlálója minden alkalommal növekszik, amint egy érvényes rekord beolvasása történik, így a bejárás rögtön megáll, amint az összes rekord feldolgozása végbement. Megjegyzendő, hogy a rekordtároló nem kerül zárolásra a bejárás ideje alatt, így amennyiben szükséges, beépíthető egy *synchronized* blokk, ami más programszálaktól jövő, a rekordtárolót ért változásokat figyeli és szinkronizálja.
A brute-force algoritmust egyszerű megérteni és könnyen lehet alkalmazni néhány rekordot tartalmazó tárolónál, azonban előnyösebb használni a felsorolást.

3. Rekordok felsorolása

Az előbbi példában látott megoldás helyett - ahol ciklusból való kilépés történt a kivétel keletkezésének hatására – végignézhetjük az összes érvényes rekordazonosítót, lényegében brute-force módszer szerint. Erre azonban felkínál az RMS egy speciális programozási eszközt, a felsorolást – használva a RecordEnumeration interfészt -, ami visszatér a felsorolt rekordok azonosítóival. Ez az interfész nem terjeszti ki a szabványos *java.util.Enumeration* interfészt, de hasonló módon működik. Tulajdonképpen, ez egy sokoldalúbb interfész: oda-vissza lehet rekordfelsorolást végezni és opcionálisan használható a rekordtárolóban történt változások nyomon követése.
Létrehozható egy felsorolás, úgy hogy meghívod az enumerateRecord() metódusát annak a rekordtárolónak, amelynek elemeinek listájára szükség van, mint itt a példában:

```
...
RecordStore rs = ... // rekordtároló megnyitása
RecordEnumeration enum =
        rs.enumerateRecords( null, null, false );

... // felsorolás használata itt

enum.destroy(); // mindent kitöröl!
...
```

Az egyszerűség kedvéért, a kivételkezelés kimaradt ebből a kódrészletből, de nem árt körültekintőnek lenni, hogy *enumerateRecords()* dobhat egy *RecordStoreNotOpenExceptiont* kivételt.

Az *enumerateRecords()* első két paramétere a rekordok szűrésére és rendezésére vonatkozik (ezután kerül tárgyalásra). A harmadik paraméterrel beállíthatjuk, hogy a felsorolás következtében a rekordtárolóban végbement változások nyomon legyenek-e követve. A változások nyomon követésére csak különleges esetekben lehet szükség, ami most itt nem szükséges, így minden további példában false-ra állítható.

Amikor egy felsorolás elkészült, meg kell hívni a *destroy()* metódust, hogy felszabadítsa az erőforrásokat, amiket a rendszer lefoglalt. A fel nem számolt felsorolások az alkalmazásban, memóriahiányt eredményezhetnek.

A *hasNextElement()* és *nextRecordId()*, a felsorolásban előre felé haladnak:

```
...
RecordStore rs = ...
RecordEnumeration enum = ...

try {
   while( enum.hasNextElement() ){
      int id = enum.nextRecordId();
      byte[] data = rs.getRecord( id );
      ... // tetszőleges művelet az adattal
   }
}
catch( RecordStoreException e ){
   // hiba lekezelése
}
...
```

Visszafelé haladni a *hasPreviousElement()* és a *previousRecordId()* használatával lehet. Észrevehető, hogy *nextRecordId()* és *previousRecordId()* is *InvalidRecordIDExceptiont* kivétel dob, ha egy rekord törlődik a rekordtárolóból, amíg a felsorolás aktív és nincs a rekordtároló változásaira nyomkövetés beállítva.

A módszer, amely visszatér a rendezetlen felsorolás rekordjaival implementációfüggő.

A kényelem kedvéért arra is van mód, hogy a rekordazonosító helyett magát a rekordot érjük el (a következőt vagy az előzőt) a *nextRecord()* és a *previousRecord()* használatával:

```
...
try {
   while( enum.hasNextElement() ){
      byte[] data = enum.nextRecord();
      ... // tetszőleges művelet az adattal
   }
}
catch( RecordStoreException e ){
```

```
   // hiba lekezelése
}
...
```

Mindazonáltal a rekord azonosítóját így nem fogod megkapni, de megtehető az, hogy az adatokat pontosan a memóriafoglalások sorrendjében helyezed el a bájttömbben.
A *numRecords()* metódus visszaadja a felsorolásban szereplő rekordazonosítók számát, de ez a hívás, mivel egyből elvégzi a felsorolást, sok memóriát használ fel, így a feldolgozásban okozhat egy viszonylag hosszabb szünetet.
A *reset()* újraindítja a felsorolást és kezdőállapotába helyezi azt. A *rebuild()* hatására a rekordtároló aktuális állapota alapján frissíti magát.
Az *isKeptUpdated()* használatával ellenőrizhető, hogy a felsorolás során nyomkövetési változások történtek-e a rekordtárolóban. A *keepUpdated()* metódussal pedig meg lehet adni a változások nyomon követésének státuszát.

4. Felsorolt rekordok szűrése

Egy rekordtároló egy alhalmazán értelmezett. Használatához szükséges egy felsorolás, amely kihagyja a keresetlen rekordokat, egy szűrő segítségével. Ez egy objektum, ami implementálja a RecordFilter interfészt. A szűrő *matches()* metódusának kell jeleznie, hogy a vizsgált rekord beletartozik-e a felsorolásba:

```
public class MyFilter implements RecordFilter {
   public boolean matches( byte[] recordData ){
      ... // illesztést végző kód
   }
}
```

A módszer egyetlen paramétere a bájttömbben tárolt rekord, amely a felsorolás során felveszi a rekordtárolóban tárolt értékeket, amelyekhez pedig illesztjük a keresett értéket, így végezve el a szűrést. A keresett értéket az implementáló osztály konstruktorának paraméterében adjuk meg. A szűrő objektum az *enumerateRecords()* első paramétereként szerepel:

```
...
enum = rs.enumerateRecords( new MyFilter(), null, false );
...
```

A rekordazonosító nem kerül átadásra a szűrőnek, csak a rekordadatot tartalmazó bájttömb. A feldolgozás során szükség lehet a rekordazonosítóra is, így azt el lehet tárolni a rekord valamely mezőjében, így az azonosíthatóvá válik. Az előzőekben arra használtuk az első rekordot, hogy a mezőkkel kapcsolatos információkat tároljon - nyilvánvalóan ki kell szűrni ezt a segédrekordot bármilyen felsorolásból.

5. Felsorolt rekordok rendezése

A rekordokat érdemes valamilyen következetes szempont szerint rendezni a felsorolásban. Ehhez a felsorolás el kell látni egy összehasonlító függvénnyel. Ezen összehasonlító módszert, egy olyan objektum tartalmazza, amely a *RecordComparator* interfészt implementálja. Az összehasonlító *compare()* metódus egy konstans értékkel tér vissza, amely -1, 0 vagy 1 lehet, attól függően, hogy a függvény paraméterében megadott két rekord közül, az első megelőzi-e, követi-e a másodikat vagy egyelőek-e:

```
public class MyComparator implements RecordComparator {
   public int compare( byte[] r1, byte[] r2 ){
      int salary1 = getSalary( r1 );
      int salary2 = getSalary( r2 );

      if( salary1 < salary2 ){
         return PRECEDES;
      } else if( salary1 == salary2 ){
         return EQUIVALENT;
      } else {
         return FOLLOWS;
      }
      }

      private int getSalary( byte[] data ){
         ... // fizetési információk kezelése
      }
   }
```

Az állandók a következők *PRECEDES* , *EQUIVALENT* , és *FOLLOWS* . Ezeket a *RecordComparator* interfész definiálja. Az összehasonlító objektumok *enumerateRecords()* második paraméterében kell megadni:

```
...
enum = rs.enumerateRecords( null, new MyComparator(), false );
...
```

Ahogy szűrőnél, a rekordazonosítók itt sem kerülnek átadásra az összehasonlító objektumnak, csak a nyers rekordadatok.

Mindkét objektum átadható - a szűrő és az összehasonlító is -, a felsorolásnak. Ilyenkor a szűrés előbb történik meg, mint az adatok rendezése.

VI. Keresés a rekordtárolóban

Az előzőekben megnéztük, hogy hogyan lehet kilistázni egy rekordtároló tartalmát, hogyan lehet rekordokat rendezni és a szűrők használatával egyszerű lekérdezési listát létrehozni. A továbbiakban feltárjuk az egy vagy több rekord keresésére vonatkozó különböző stratégiákat.

1. Stratégiák keresése

Nyilvánvalóan, a legegyszerűbb módja rekordok vagy rekordhalmazok keresésének, a szűrők használata. A szűrőnek ismernie kell a rekordadatok tárolási módját. Érdemes az adattárolásnak, adatmegfeleltetésnek olyan módját használni, hogy az más alkalmazásoknál is újrafelhasználható legyen. Az adatmegfeleltetések tárgyalásánál például úgy definiáltuk *FieldListet* és *a FieldBasedStore* osztályokat, hogy tetszőleges adatok olvasását és írását kezeljék a rekordtárolóban. Egy kevés kódújraírással, ezek mintájára létrehozható egy új, teljesen általános alaposztály, a *FieldBasedRecordMapper*:
A forráskódot az 5. sz. melléklet tartalmazza.
Most kiterjesztjük *FieldBasedRecordMapper* osztályt úgy, hogy létrehozzunk egy másik absztrakt osztályt, a *FieldBasedFilter*-t, a szűrő alaposztályunk számára:

```
import javax.microedition.rms.*;

// Rekord-filter a rekordok szűréséhez, amelyek adatai
// le vannak képezve mezőlistákra. Az aktuális szűrő
// ki lesz terjesztve és implementálva a matchFields
// metódus által.

public abstract class FieldBasedFilter
                extends FieldBasedRecordMapper
                implements RecordFilter {

   // Konstruktor a filterhez. Az opcionális bájttömböt
   // – ami nem vesz részt az adatfeldolgozásban -
   // általában első rekordként helyezzük el a rekordtárolóban,
   // ahol raktározzuk a mezőkre vonatkozó információkat.

   protected FieldBasedFilter(){
      this( null );
   }

   protected FieldBasedFilter( byte[] ignore ){
```

```
        _ignore = ignore;
    }

    // Két bájttömb összehasonlítása

    private boolean equal( byte[] a1, byte[] a2 ){
        int len = a1.length;

        if( len != a2.length ) return false;

        for( int i = 0; i < len; ++i ){
            if( a1[i] != a2[i] ) return false;
        }

        return true;
    }

    // Szűrő meghívása egy rekordhoz

    public boolean matches( byte[] data ){
        if( _ignore != null ){
            if( equal( _ignore, data ) ) return false;
        }

        prepareForInput( data );
        return matchFields();
    }

    // Az aktuális szűrő fogja majd implementálni ezt a metódust.

    protected abstract boolean matchFields();

    private byte[] _ignore;
}
```

A *FieldList* egy példányát valahogy így hozhatjuk létre:

```
...
FieldList empFields = new FieldList( 5 );

empFields.setFieldType( 0, FieldList.TYPE_INT );
empFields.setFieldName( 0, " Azonosító" );
empFields.setFieldType( 1, FieldList.TYPE_STRING );
empFields.setFieldName( 1, " Keresztnév" );
empFields.setFieldType( 2, FieldList.TYPE_STRING );
empFields.setFieldName( 2, " Vezetéknév" );
empFields.setFieldType( 3, FieldList.TYPE_BOOLEAN );
empFields.setFieldName( 3, " Tevékenység" );
```

```
empFields.setFieldType( 4, FieldList.TYPE_CHAR );
empFields.setFieldName( 4, " Nem" );
...
```

Íme egy példa, egy osztályra, ami egy megadott utónévre keres rá:

```
import java.io.IOException;

// Szűrő, amely illesztést végez egy megadott vezetéknévre,
// az alkalmazottak osztály rekordjában.

public class MatchLastName extends FieldBasedFilter {
   public MatchLastName( String name ){
      this( name, null );
   }

   public MatchLastName( String name, byte[] ignore ){
      super( ignore );
      _name = name;
   }

   protected boolean matchFields(){
      try {
         readField( FieldList.TYPE_INT );
         readField( FieldList.TYPE_STRING );

         String ln = (String)
                  readField( FieldList.TYPE_STRING );

         return ln.equals( _name );
      }
      catch( IOException e ){
         return false;
      }
   }

   private String _name;
}
```

A rekordkeresés megadása nagyon egyszerű:

```
...
RecordStore employees = ... // alkalmazottak listája
RecordFilter lname = new MatchLastName( "Smith" );
RecordEnumeration enum =
     employees.enumerateRecords( lname, null, false );

while( enum.hasNextElement() ){
```

```
   int id = enum.nextRecordId();
   ... // stb. stb./
}

enum.destroy();
...
```

Körültekintően kell eljárni a kód használata során, hogy a rekordok ne legyenek gyakrabban feldolgozás alatt, mint szükséges. Egyik megközelítése a problémának, hogy eltároljuk a memóriában a nem túl régi hozzáférésű rekordokat. Például minden alkalommal, amikor egy szűrésben résztvevő rekord felhasználásra kerül és ebben a formában tároljuk, ez az objektum egy egyedi entitást reprezentál. A rekordazonosítója kulcsként funkcionálhat, azonban ehhez, természetesen, el kell tárolni mezőként a rekord azonosítóját.

Egyszerűbb útja lehet a felsorolás használatával az illeszkedő rekordokat összegyűjteni és kicsomagolni egy különálló listában. A már korábban létrehozott Contact osztály *toByteArray()* és *fromByteArray()* adatleképző módszerek így módosulnak:

```
import java.io.*;

// Kapcsolati információk egy személyhez

public class Contact {
   private String _firstName;
   private String _lastName;
   private String _phoneNumber;

   public Contact(){
   }

   public Contact( String firstName, String lastName,
                   String phoneNumber )
   {
      _firstName = firstName;
      _lastName = lastName;
      _phoneNumber = phoneNumber;
   }

   public String getFirstName(){
      return _firstName != null ? _firstName : "";
   }

   public String getLastName(){
      return _lastName != null ? _lastName : "";
   }
```

```
public String getPhoneNumber(){
   return _phoneNumber != null ? _phoneNumber : "";
}

public void setFirstName( String name ){
   _firstName = name;
}

public void setLastName( String name ){
   _lastName = name;
}

public void setPhoneNumber( String number ){
   _phoneNumber = number;
}

public void fromByteArray( byte[] data )
                        throws IOException {
   ByteArrayInputStream bin =
                new ByteArrayInputStream( data );
   DataInputStream din = new DataInputStream( bin );

   fromDataStream( din );
   din.close();
}

public byte[] toByteArray() throws IOException {
   ByteArrayOutputStream bout =
                new ByteArrayOutputStream();
   DataOutputStream dout = new DataOutputStream( bout );

   toDataStream( dout );
   dout.close();

   return bout.toByteArray();
}

public void fromDataStream( DataInputStream din )
                        throws IOException {
   _firstName = din.readUTF();
   _lastName = din.readUTF();
   _phoneNumber = din.readUTF();
}

public void toDataStream( DataOutputStream dout )
                    throws IOException {
   dout.writeUTF( getFirstName() );
```

```
        dout.writeUTF( getLastName() );
        dout.writeUTF( getPhoneNumber() );
    }
}
```

A következő osztály rekordok szűrését végzi, keresés céljából, valamint az illeszkedő rekordok eltárolását. Üres felsorolás jelzése esetén hamis értékkel tér vissza:

```
import java.io.*;
import java.util.*;
import javax.microedition.rms.*;

// Megkeresi a kapcsolati információkat egy megadott személy
// vezeték- és/vagy keresztnevéhez végzett illesztéssel.

public class FindContacts {

    // Létrehozz egy keresőt a megadott névhez.
    // Ha mindkét név meg van adva, akkor a vezeték- és a keresztnévre is keres,
    // különben csak a megadottra végez illesztést.

    public FindContacts( String fname, String lname ){
        _fname = normalize( fname );
        _lname = normalize( lname );
    }

    // Végighalad a rekordtároló adatain és
    // visszatér az illeszkedő kapcsolati objektum listájával.

    public Vector list( RecordStore rs )
                    throws RecordStoreException,
                        IOException {

        Vector v = new Vector();
        Filter f = new Filter( v );
        RecordEnumeration enum =
              rs.enumerateRecords( f, null, false );
        enum.hasNextElement();
        enum.destroy();

        return v;
    }

    // Létezik-e a megadott kapcsolati információ vagy sem

    public boolean matchesContact( Contact c ){
        boolean sameFirst = false;
        boolean sameLast = false;
```

```
    if( _fname != null ){
        sameFirst =
          c.getFirstName().toLowerCase().equals(_fname);
    }

    if( _lname != null ){
        sameLast =
          c.getLastName().toLowerCase().equals( _lname );
    }

    if( _fname != null && _lname != null ){
        return sameFirst && sameLast;
    }

    return sameFirst || sameLast;
}

// Normalizálja a megadott nevet

private static String normalize( String name ){
    return( name != null ?
            name.trim().toLowerCase() : null );
}

private String _fname;
private String _lname;


private class Filter implements RecordFilter {
    private Filter( Vector list ){
        _list = list;
    }

    public boolean matches( byte[] data ){
        try {
            Contact c = new Contact();
            c.fromByteArray( data );

            if( matchesContact( c ) ){
                _list.addElement( c );
            }
        }
        catch( IOException e ){
        }

        return false;
    }
```

```
        private Vector _list;
    }
}
```

Sajnos, memóriakorlátok megakadályozhatják több objektum eltárolását egy időben, mint amennyire szükség lehet némely esetben. Lehet nyerni egy kevés helyet az indexek használatával, lefoglalható egy erre a célra fenntartott táblázat, ahol rekordazonosító és kulcsérték párok tárolhatóak, úgymint itt a contacts nevei. Egy index általában elég kisméretű, így bent maradhat a memóriában, és megkímél attól, hogy újból és újból létre kelljen hozni egy idő rabló felsorolást, amikor egy konkrét rekordra rá akarsz keresni.

VII. Összefoglalás

Eddig tartott a mobiltelefonokra szánt Java alapú perzisztens adattárolás ismertetése, valamint az RMS (Record Management System) szerény eszközkészletének kiegészítése segédosztályokkal. A mobil adatbázis-kezelés lehetőségeit látva ezen segédosztályok további fejlesztésére, kibővítésére, csiszolására még nagy szükség van (legalább is a szakdolgozat beadásának idejében) és még akkor sem hasonlíthatóak az olyan profi adatbázis-kezelési technikákhoz, mint a szintén Java platformon használatos JDBC-ben már megvalósításra kerültek, azonban ez nem is csoda hiszen azt számítógépekre tervezték.

Fejlettebb mobiladatbázis-kezelési technológia létrehozására született már megoldás más platformon: az úgynevezett „okostelefonok" Symbian operációsrendszere SQL-t használ, élvezve a relációs adatbázis-kezelés nyújtotta előnyöket. Azonban a Symbian alá írni alkalmazásainkat mégsem tanácsos, mivel ez a fajta platform jelen pillanatban nem igazán elterjedt. Túlnyomórészt Nokia telefonokat működtetnek, bár ez a márka igencsak közkedvelt, de még így is a telefontulajdonosok viszonylag szűk rétege használhatná az általunk készített programokat.

Remélhetőleg a Sun cég Java fejlesztői is hamarosan követik a példát, hiszen a mobiltechnológia egy meredeken emelkedő iparág, így az a mobil-adatbáziskezelés területén sem maradhat le.

Addig sem szabad azonban elvetni az RMS-t, hiszen szorgalmas munkával és egy kis kreativitással hasznos alkalmazásokat készíthetnek a vállalkozó kedvű programozók.

VIII. Irodalomjegyzék

Databases and MIDP, Part 1 Understanding the Record Management System
http://developers.sun.com/techtopics/mobility/midp/articles/databaserms/

Databases and MIDP, Part 2 Data Mapping
http://developers.sun.com/techtopics/mobility/midp/articles/databasemap/

Databases and MIDP, Part 3 Putting Data Mapping to Work
http://developers.sun.com/techtopics/mobility/midp/articles/databasemapextend/

Databases and MIDP, Part 4 Filtering and Traversal Strategies
http://developers.sun.com/techtopics/mobility/midp/articles/databasefilter/

Databases and MIDP, Part 5 Searching a Record Store
http://developers.sun.com/techtopics/mobility/midp/articles/databaserecordstore/

J2ME record management store
http://www-128.ibm.com/developerworks/wireless/library/wi-rms/

J2ME Java ME MicroDevNet Working with the RMS
http://microjava.com/articles/techtalk/rms?content_id=2124

J2ME Java ME MicroDevNet Working with the RMS - Part II
http://www.microjava.com/articles/techtalk/rms2?PageNo=1

MIDP Database Programming Using RMS a Persistent Storage for MIDlets
http://developers.sun.com/techtopics/mobility/midp/articles/persist/

MIDP Programming with J2ME
http://www.developer.com/java/j2me/article.php/10934_1561591_1

Exploring the NetBeans Visual Mobile Designer
http://www.netbeans.org/kb/41/exploringmvd.html

java.net J2ME Tutorial, Part 1 Creating MIDlets
http://today.java.net/pub/a/today/2005/02/09/j2me1.html

java.net J2ME Tutorial, Part 2 User Interfaces with MIDP 2.0
http://today.java.net/pub/a/today/2005/05/03/midletUI.html

java.net J2ME Tutorial, Part 3 Exploring the Game API of MIDP 2.0
http://today.java.net/pub/a/today/2005/07/07/j2me3.html

java.net J2ME Tutorial, Part 4 Multimedia and MIDP 2.0
http://today.java.net/pub/a/today/2005/09/27/j2me4.html

java.net Mobile Memories The MIDP Record Management System
http://today.java.net/pub/a/today/2004/11/16/J2ME-3.html

IX. Függelék

1. sz. melléklet: Az RMSAnalyzer osztály

```
import java.io.*;
import javax.microedition.rms.*;

// Rekordtároló analizálása.
// Analizálás eredményének kimenete a System.out szabvány,
// de ez megváltoztatható az implementációban,
// hogy a naplózás eredménye hová kerüljön.

public class RMSAnalyzer {

   // naplózás interfész

   public interface Logger {
      void logEnd( RecordStore rs );
      void logException( String name, Throwable e );
      void logException( RecordStore rs, Throwable e );
      void logRecord( RecordStore rs, int id,
                  byte[] data, int size );
      void logStart( RecordStore rs );
   }

   private Logger logger;

   // Analizáló konstruktora a System.out kimenetre.

   public RMSAnalyzer(){
      this( null );
   }

   // Analizáló konstruktora egy megadott kimenetre.

   public RMSAnalyzer( Logger logger ){
      this.logger = ( logger != null ) ? logger :
                              new SystemLogger();
   }

   // Rekordtárolók megnyitása a MIDlet saját eszközeivel
   // és azok tartalmának elemzése.

   public void analyzeAll(){
      String[] names = RecordStore.listRecordStores();
```

```
    for( int i = 0;
         names != null && i < names.length;
         ++i ){
        analyze( names[i] );
    }
}

// Rekordtároló megnyitása név szerint és annak tartalmának elemzése.

public void analyze( String rsName ){
    RecordStore rs = null;

    try {
        rs = RecordStore.openRecordStore( rsName, false );
        analyze( rs );
    } catch( RecordStoreException e ){
        logger.logException( rsName, e );
    } finally {
        try {
            rs.closeRecordStore();
        } catch( RecordStoreException e ){
            // kivétel eldobása
        }
    }
}

// Nyitott rekordtároló tartalmának elemzése
// egy egyszerű brute force keresési eljárással.

public synchronized void analyze( RecordStore rs ){
    try {
        logger.logStart( rs );

        int    lastID = rs.getNextRecordID();
        int    numRecords = rs.getNumRecords();
        int    count = 0;
        byte[] data = null;

        for( int id = 0;
             id < lastID && count < numRecords;
             ++id ){
            try {
                int size = rs.getRecordSize( id );

                // Megbizonyosodni, hogy az adattömbben van-e elég hely,
                // plusz néhány további információ hozzáadása.
```

```
                if( data == null || data.length < size ){
                    data = new byte[ size + 20 ];
                }

                rs.getRecord( id, data, 0 );
                logger.logRecord( rs, id, data, size );

                ++count; // csak akkor növelni az értékét, ha a rekord létezik
            }
            catch( InvalidRecordIDException e ){
                // elvetni a rekordot és továbblépni a következőre
            }
            catch( RecordStoreException e ){
                logger.logException( rs, e );
            }
        }

    } catch( RecordStoreException e ){
        logger.logException( rs, e );
    } finally {
        logger.logEnd( rs );
    }
}

// Napló kimenete a PrintStream-re.

public static class PrintStreamLogger implements Logger {
    public static final int COLS_MIN = 10;
    public static final int COLS_DEFAULT = 20;

    private int         cols;
    private int         numBytes;
    private StringBuffer hBuf;
    private StringBuffer cBuf;
    private StringBuffer pBuf;
    private PrintStream  out;

    public PrintStreamLogger( PrintStream out ){
        this( out, COLS_DEFAULT );
    }

    public PrintStreamLogger( PrintStream out, int cols ){
        this.out = out;
        this.cols = ( cols > COLS_MIN ? cols : COLS_MIN );
    }

    private char convertChar( char ch ){
        if( ch < 0x20 ) return '.';
```

```
    return ch;
}

public void logEnd( RecordStore rs ){
    out.println( "\nA rekordok aktuális mérete = "
                + numBytes );
    printChar( '-', cols * 4 + 1 );

    hBuf = null;
    cBuf = null;
    pBuf = null;
}

public void logException( String name, Throwable e ){
    out.println( "Kivétel történt analizálás közben " +
                name + ": " + e );
}

public void logException( RecordStore rs, Throwable e ){
    String name;

    try {
        name = rs.getName();
    } catch( RecordStoreException rse ){
        name = "";
    }

    logException( name, e );
}

public void logRecord( RecordStore rs, int id,
                       byte[] data, int len ){
    if( len < 0 && data != null ){
        len = data.length;
    }

    hBuf.setLength( 0 );
    cBuf.setLength( 0 );

    numBytes += len;

    out.println( "Rekord #" + id + " hossza "
                + len + " bytes" );

    for( int i = 0; i < len; ++i ){
        int    b = Math.abs( data[i] );
        String hStr = Integer.toHexString( b );
```

```
            if( b < 0x10 ){
               hBuf.append( '0');
            }

            hBuf.append( hStr );
            hBuf.append( ' ' );

            cBuf.append( convertChar( (char) b ) );

            if( cBuf.length() == cols ){
               out.println( hBuf + " " + cBuf );

               hBuf.setLength( 0 );
               cBuf.setLength( 0 );
            }
         }

         len = cBuf.length();

         if( len > 0 ){
            while( len++ < cols ){
               hBuf.append( "   " );
               cBuf.append( ' ' );
            }

            out.println( hBuf + " " + cBuf );
         }
      }

      public void logStart( RecordStore rs ){
         hBuf = new StringBuffer( cols * 3 );
         cBuf = new StringBuffer( cols );
         pBuf = new StringBuffer();

         printChar( '=', cols * 4 + 1 );

         numBytes = 0;

         try {
            out.println( "Rekord tároló: "
                     + rs.getName() );
            out.println( "    Rekordszám = "
                     + rs.getNumRecords() );
            out.println( "    Teljes méret = "
                     + rs.getSize() );
            out.println( "    Verzió = "
                     + rs.getVersion() );
            out.println( "    Utolsó módosítás ideje = "
```

```
                    + rs.getLastModified() );
            out.println( "    Rendelkezésre álló méret = "
                    + rs.getSizeAvailable() );
            out.println( "" );
        } catch( RecordStoreException e ){
            logException( rs, e );
        }
    }

    private void printChar( char ch, int num ){
        pBuf.setLength( 0 );
        while( num-- > 0 ){
            pBuf.append( ch );
        }
        out.println( pBuf.toString() );
    }
}

// Napló kimenete a szabványos System.out-ra.

public static class SystemLogger
                extends PrintStreamLogger {
    public SystemLogger(){
        super( System.out );
    }

    public SystemLogger( int cols ){
        super( System.out, cols );
    }
}
```

2. sz. melléklet: A FieldList osztály

```
import java.io.*;
import javax.microedition.rms.*;

// Fentartott információk a mezőkről a rekordalapú rekordtárolóban.
// Általában csak egy mezőtípus lista és (opcionálisan) mezőnevek,
// de könnyen kiterjeszthető a tároló egyéb információkkal.

public class FieldList {

    private static final int VERSION = 1;

    // Alap mezőtípusok
```

```
public static final byte TYPE_BOOLEAN = 1;
public static final byte TYPE_BYTE = 2;
public static final byte TYPE_CHAR = 3;
public static final byte TYPE_SHORT = 4;
public static final byte TYPE_INT = 5;
public static final byte TYPE_LONG = 6;
public static final byte TYPE_STRING = 7;

// Üres lista létrehozása

public FieldList(){
}

// Megadott lista létrehozása

public FieldList( int numFields ){
   if( numFields < 0 || numFields > 255 ){
      throw new IllegalArgumentException(
            "Helytelen mezőszám" );
   }

   _types = new byte[ numFields ];
   _names = new String[ numFields ];
}

// Visszatér a mezők számával

public int getFieldCount(){
   return _types != null ? _types.length : 0;
}

// Visszatér a mező nevével

public String getFieldName( int index ){
   String name = _names[ index ];
   return name != null ? name : "";
}

// Visszatér a mező típusával

public byte getFieldType( int index ){
   return _types[ index ];
}

// Kiolvassa a mező listát a bájttömbből.

public void fromByteArray( byte[] data )
                  throws IOException {
```

```
    ByteArrayInputStream bin =
            new ByteArrayInputStream( data );
    fromDataStream( new DataInputStream( bin ) );
    bin.close();
}

// Kiolvassa a mezőlistát az adatfolyamból.

public void fromDataStream( DataInputStream din )
                        throws IOException {
    int version = din.readUnsignedByte();
    if( version != VERSION ){
        throw new IOException( "Incorrect version " +
            version + " for FieldList, expected " +
            VERSION );
    }

    int numFields = din.readUnsignedByte();

    _types = new byte[ numFields ];
    _names = new String[ numFields ];

    if( numFields > 0 ){
        din.readFully( _types );

        for( int i = 0; i < numFields; ++i ){
            _names[i] = din.readUTF();
        }
    }
}

// Kiolvassa a mezőlistát a rekordtárolóból.

public void fromRecordStore( RecordStore rs, int index )
                        throws IOException,
                               RecordStoreException {
    fromByteArray( rs.getRecord( index ) );
}

// Beállítja a mező nevét

public void setFieldName( int index, String name ){
    _names[ index ] = name;
}

// Beállítja a mező típusát

public void setFieldType( int index, byte type ){
```

```
    _types[ index ] = type;
}

// Eltárolja a mezőlistát egy bájttömbbe.

public byte[] toByteArray() throws IOException {
    ByteArrayOutputStream bout =
            new ByteArrayOutputStream();
    toDataStream( new DataOutputStream( bout ) );
    byte[] data = bout.toByteArray();
    bout.close();
    return data;
}

// Eltárolja a mezőlistát egy adatfolyamba.

public void toDataStream( DataOutputStream out )
                    throws IOException {
    out.writeByte( VERSION );

    int count = getFieldCount();

    out.writeByte( count );

    if( count > 0 ){
        out.write( _types, 0, count );

        for( int i = 0; i < count; ++i ){
            out.writeUTF( getFieldName( i ) );
        }
    }
}

// Kiírja a mezőlistát egy rekordtárolóba.

public int toRecordStore( RecordStore rs, int index )
                    throws IOException,
                        RecordStoreException {
    byte[]  data = toByteArray();
    boolean add = true;

    if( index > 0 ){
        try {
            rs.setRecord( index, data, 0, data.length );
            add = false;
        }
        catch( InvalidRecordIDException e ){
        }
```

```
        }

        // Ha a rekord valójában nem létezik még,
        // akkor előzetesen elkészti annak egy „álrekordját”.

        if( add ){
            synchronized( rs ){
                int nextID = rs.getNextRecordID();
                if( index <= 0 ) index = nextID;

                while( nextID < index ){
                    rs.addRecord( null, 0, 0 );
                }

                if( nextID == index ){
                    rs.addRecord( data, 0, data.length );
                }
            }
        }

        return index;
    }

    private String[] _names;
    private byte[]   _types;
}
```

3. sz. melléklet: A FieldBaseStore osztály

```
import java.io.*;
import javax.microedition.rms.*;
import j2me.io.*;

// Csomagoló osztály rekordtárolóhoz, amely engedélyezi
// a mezők halmazából álló rekordokhoz való hozzáférést.
// A meződefiníciókról a FieldList osztály gondoskodik, amely külön kezelünk és
// amelyet a rekordtároló egy elválasztott részében tárolunk.

public class FieldBasedStore {

    // Néhány hasznos konstans

    public static Boolean TRUE = new Boolean( true );
    public static Boolean FALSE = new Boolean( false );

    // Fenntartott sztring típusú jelzők
```

```
private static final byte NULL_STRING_MARKER = 0;
private static final byte UTF_STRING_MARKER = 1;

// Mezőtároló létrehozása, ahol a tároló lefoglalja magának
// az első rekordot a mezőlista számára.

public FieldBasedStore( RecordStore rs )
                    throws IOException,
                         RecordStoreException {
   this( rs, 1 );
}

// Mezőtároló létrehozása, ahol a
// mezőlista megadott rekordban van tárolva.

public FieldBasedStore( RecordStore rs, int fieldListID )
                    throws IOException,
                         RecordStoreException {
   this( rs, loadFieldList( rs, fieldListID ) );
}

// Mezőtároló létrehozása, megadott mezőlista alapján

public FieldBasedStore( RecordStore rs, FieldList list ){
   _rs = rs;
   _fieldList = list;
}

// Új rekord hozzáadása a tárolóhoz
// Visszatér az új rekord azonosítójával

public synchronized int addRecord( Object[] fields )
                        throws IOException,
                             RecordStoreException {
   writeStream( fields );
   byte[] data = _bout.getByteArray();
   return _rs.addRecord( data, 0, data.length );
}

// Visszatér a mezőlistával

public FieldList getFieldList(){
   return _fieldList;
}

// Visszatér a rekordtárolóval
```

```
public RecordStore getRecordStore(){
   return _rs;
}

// Betölti a mezőlistát a rekordtárolóból

private static FieldList loadFieldList( RecordStore rs,
                                        int fieldListID )
                                 throws IOException,
                                        RecordStoreException {
   FieldList list = new FieldList();
   list.fromRecordStore( rs, fieldListID );
   return list;
}

// Előkészíti a tárolót a bemenetre, azáltal, hogy az adat puffer
// méretét elég nagyra állítja.

private void prepareForInput( int size ){
   if( _buffer == null || _buffer.length < size ){
      _buffer = new byte[ size ];
   }

   if( _bin == null ){
      _bin = new DirectByteArrayInputStream( _buffer );
      _din = new DataInputStream( _bin );
   } else {
      _bin.setByteArray( _buffer );
   }
}

// Előkészíti a tárolót a kimenetre.

private void prepareForOutput(){
   if( _bout == null ){
      _bout = new DirectByteArrayOutputStream();
      _dout = new DataOutputStream( _bout );
   } else {
      _bout.reset();
   }
}

// Kiolvas egy mezőt a pufferből.

private Object readField( int type ) throws IOException {
   switch( type ){
      case FieldList.TYPE_BOOLEAN:
         return _din.readBoolean() ? TRUE : FALSE;
```

```
        case FieldList.TYPE_BYTE:
            return new Byte( _din.readByte() );
        case FieldList.TYPE_CHAR:
            return new Character( _din.readChar() );
        case FieldList.TYPE_SHORT:
            return new Short( _din.readShort() );
        case FieldList.TYPE_INT:
            return new Integer( _din.readInt() );
        case FieldList.TYPE_LONG:
            return new Long( _din.readLong() );
        case FieldList.TYPE_STRING: {
            byte marker = _din.readByte();
            if( marker == UTF_STRING_MARKER ){
                return _din.readUTF();
            }
        }
    }

    return null;
}

// Kiolvas egy rekordot egy megadott azonosító alapján
// és visszatér, mint egy objektumhalmazzal,
// amelynek elemei megegyeznek a mezőlista típusaival.

public synchronized Object[] readRecord( int recordID )
                        throws IOException,
                               RecordStoreException {
    prepareForInput( _rs.getRecordSize( recordID ) );
    _rs.getRecord( recordID, _buffer, 0 );

    int count = _fieldList.getFieldCount();
    Object[] fields = new Object[ count ];

    for( int i = 0; i < count; ++i ){
        fields[i] = readField(_fieldList.getFieldType(i));
    }

    return fields;
}

// Átkonvertál egy objektumot logikai típusú értékké

public static boolean toBoolean( Object value ){
    if( value instanceof Boolean ){
        return ((Boolean) value).booleanValue();
    } else if( value != null ){
        String str = value.toString().trim();
```

```
        if( str.equals( "true" ) ) return true;
        if( str.equals( "false" ) ) return false;

        return( toInt( value ) != 0 );
    }

    return false;
}

// Átkonvertál egy objektumot karakter típusú értékké

public static char toChar( Object value ){
    if( value instanceof Character ){
        return ((Character) value).charValue();
    } else if( value != null ){
        String s = value.toString();
        if( s.length() > 0 ){
            return s.charAt( 0 );
        }
    }

    return 0;
}

// Átkonvertál egy objektumot egész típusú értékké

public static int toInt( Object value ){
    if( value instanceof Integer ){
        return ((Integer) value).intValue();
    } else if( value instanceof Boolean ){
        return ((Boolean) value).booleanValue() ? 1 : 0;
    } else if( value instanceof Byte ){
        return ((Byte) value).byteValue();
    } else if( value instanceof Character ){
        return ((Character) value).charValue();
    } else if( value instanceof Short ){
        return ((Short) value).shortValue();
    } else if( value instanceof Long ){
        return (int) ((Long) value).longValue();
    } else if( value != null ){
        try {
            return Integer.parseInt( value.toString() );
        }
        catch( NumberFormatException e ){
        }
    }
```

```
    return 0;
}

// Átkonvertál egy objektumot hosszú egész típusú értékké

public static long toLong( Object value ){
    if( value instanceof Integer ){
        return ((Integer) value).longValue();
    } else if( value instanceof Boolean ){
        return ((Boolean) value).booleanValue() ? 1 : 0;
    } else if( value instanceof Byte ){
        return ((Byte) value).byteValue();
    } else if( value instanceof Character ){
        return ((Character) value).charValue();
    } else if( value instanceof Short ){
        return ((Short) value).shortValue();
    } else if( value instanceof Long ){
        return ((Long) value).longValue();
    } else if( value != null ){
        try {
            return Long.parseLong( value.toString() );
        }
        catch( NumberFormatException e ){
        }
    }

    return 0;
}

// Kiír egy mezőt a kimeneti pufferbe

private void writeField( int type, Object value )
                        throws IOException {
    switch( type ){
        case FieldList.TYPE_BOOLEAN:
            _dout.writeBoolean( toBoolean( value ) );
            break;
        case FieldList.TYPE_BYTE:
            _dout.write( (byte) toInt( value ) );
            break;
        case FieldList.TYPE_CHAR:
            _dout.writeChar( toChar( value ) );
            break;
        case FieldList.TYPE_SHORT:
            _dout.writeShort( (short) toInt( value ) );
            break;
        case FieldList.TYPE_INT:
            _dout.writeInt( toInt( value ) );
```

```
            break;
        case FieldList.TYPE_LONG:
            _dout.writeLong( toLong( value ) );
            break;
        case FieldList.TYPE_STRING:
            if( value != null ){
                String str = value.toString();
                _dout.writeByte( UTF_STRING_MARKER );
                _dout.writeUTF( str );
            } else {
                _dout.writeByte( NULL_STRING_MARKER );
            }
            break;
    }
}

// Kiírja egy adott rekord mezőinek halmazát.
// A mezők típusának kompatibilisnek kell lenniük
// a mezőlista típusaival.

public synchronized void writeRecord( int recordID,
                                      Object[] fields )
                         throws IOException,
                                RecordStoreException {
    writeStream( fields );
    byte[] data = _bout.getByteArray();
    _rs.setRecord( recordID, data, 0, data.length );
}

// Kiír egy mezőhalmazt a kimeneti folyamra

private void writeStream( Object[] fields )
                                throws IOException {
    int count = _fieldList.getFieldCount();
    int len = ( fields != null ? fields.length : 0 );

    prepareForOutput();

    for( int i = 0; i < count; ++i ){
        writeField( _fieldList.getFieldType( i ),
                    ( i < len ? fields[i] : null ) );
    }
}

private DirectByteArrayInputStream  _bin;
private DirectByteArrayOutputStream _bout;
private byte[]                      _buffer;
private DataInputStream             _din;
```

```
    private DataOutputStream          _dout;
    private FieldList                 _fieldList;
    private RecordStore               _rs;
}
```

4. sz. melléklet: A RMSMappings osztály

```
import java.io.*;
import java.util.*;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import javax.microedition.rms.*;
import j2me.rms.*;

// Egy egyszerű MIDlet az RMS leképzések teszteléséhez
// felhasználva FieldBasedStore osztályt.

public class RMSMappings extends MIDlet implements CommandListener {

    private Display display;

    public static final Command exitCommand =
                    new Command( "Exit",
                            Command.EXIT, 1 );
    public static final Command testCommand =
                    new Command( "Test",
                            Command.SCREEN, 1 );

    private static Object[][] empList = {
        new Object[]{ "1", "Mary", "CEO", "100", "F" },
        new Object[]{ "2", "John", "CFO", "200", "M" },
        new Object[]{ "3", "Pat", "Closem", "300", "F" },
        new Object[]{ "4", "PJ", "Admin", "100", "M" },
    };

    private static Object[][] deptList = {
        new Object[]{ "100", "Executive", "1" },
        new Object[]{ "200", "Operations", "2" },
        new Object[]{ "300", "Sales", "1" },
    };

    public RMSMappings(){
    }

    public void commandAction( Command c,
                        Displayable d ){
        if( c == exitCommand ){
            exitMIDlet();
```

```
    } else if( c == testCommand ){
        runTest();
    }
}

protected void destroyApp( boolean unconditional )
                throws MIDletStateChangeException {
    exitMIDlet();
}

public void exitMIDlet(){
    notifyDestroyed();
}

public Display getDisplay(){ return display; }

protected void initMIDlet(){
    display.setCurrent( new MainForm() );
}

protected void pauseApp(){
}

private void printRecord( FieldBasedStore store,
                    int recordID ){
    try {
        FieldList list = store.getFieldList();
        Object[]  fields = store.readRecord( recordID );

        if( fields.length != list.getFieldCount() ){
            System.out.println( "Hiba: helytelen érték" );
            return;
        }

        System.out.println( "Rekord " + recordID + ":" );

        for( int i = 0; i < fields.length; ++i ){
            System.out.println( "  " +
                list.getFieldName( i ) + ": " +
                fields[i] );
        }
    }
    catch( RecordStoreException e  ){
    }
    catch( IOException e ){
    }
}
```

```
private void runTest(){
   // Először töröljük a rekordtárolót...

   System.out.println( "Rekordtárolók törlése..." );

   String[] names = RecordStore.listRecordStores();

   for( int i = 0; i < names.length; ++i ){
      try {
         RecordStore.deleteRecordStore( names[i] );
      } catch( RecordStoreException e ){
         System.out.println( "Nem lehet törölni " +
                        names[i] );
      }
   }

   // Létrehozunk két rekordtárolót, egyik a mezőlistákat
   // tárolja az első rekordpozícióban,
   // míg a másik mezőlista el van választva.

   RecordStore     empRS = null;
   RecordStore     deptRS = null;
   FieldList       empFields = new FieldList( 5 );
   FieldList       deptFields = new FieldList( 3 );
   FieldBasedStore employees;
   FieldBasedStore departments;

   empFields.setFieldType( 0, FieldList.TYPE_INT );
   empFields.setFieldName( 0, "Azonosító" );
   empFields.setFieldType( 1, FieldList.TYPE_STRING );
   empFields.setFieldName( 1, "Keresztnév" );
   empFields.setFieldType( 2, FieldList.TYPE_STRING );
   empFields.setFieldName( 2, "Vezetéknév" );
   empFields.setFieldType( 3, FieldList.TYPE_BOOLEAN );
   empFields.setFieldName( 3, "Tevékenység" );
   empFields.setFieldType( 4, FieldList.TYPE_CHAR );
   empFields.setFieldName( 4, "Nem" );

   System.out.println( "Alkalmazottak inicializálása" );

   try {
      empRS = RecordStore.openRecordStore( "empRS",
                                    true );
      // most eltároljuk a mezőlistát a rekordtárolóban
      empFields.toRecordStore( empRS, -1 );
      employees = new FieldBasedStore( empRS );
   }
   catch( RecordStoreException e ){
```

```
    System.out.println( "Nem hozható létre az empRS" );
    return;
}
catch( IOException e ){
    System.out.println( "Hiba a mezőlista tárolásánál!" );
    return;
}

System.out.println( "Részleg inicializálása" );

deptFields.setFieldType( 0, FieldList.TYPE_INT );
deptFields.setFieldName( 0, "Azonosító" );
deptFields.setFieldType( 1, FieldList.TYPE_STRING );
deptFields.setFieldName( 1, "Név" );
deptFields.setFieldType( 2, FieldList.TYPE_INT );
deptFields.setFieldName( 2, "Vezető" );

try {
    deptRS = RecordStore.openRecordStore( "deptRS",
                                    true );
    departments = new FieldBasedStore( deptRS,
                                  deptFields );
}
catch( RecordStoreException e ){
    System.out.println( "Nem hozható létre a deptRS" );
    return;
}

int[] empRecordID;
int[] deptRecordID;
int   i;

// Adat hozzáadása...

try {
    empRecordID = new int[ empList.length ];

    for( i = 0; i < empList.length; ++i ){
        empRecordID[i] =
              employees.addRecord( empList[i] );
    }

    deptRecordID = new int[ deptList.length ];

    for( i = 0; i < deptList.length; ++i ){
        deptRecordID[i] =
            departments.addRecord( deptList[i] );
    }
```

```
}
catch( RecordStoreException e ){
   System.out.println( "Hiba a rekord hozzáadása során!" );
   return;
}
catch( IOException e ){
   System.out.println( "Hiba a mező írásakor!" );
   return;
}

// Most elérjük az adatokat és kiírjuk őket…

System.out.println( "---- Alkalmazott adatai ----" );

for( i = 0; i < empRecordID.length; ++i ){
   printRecord( employees, empRecordID[i] );
}

System.out.println( "---- Részleg adatai ----" );

for( i = 0; i < deptRecordID.length; ++i ){
   printRecord( departments, deptRecordID[i] );
}

System.out.println( "Az empRS bezárása" );

try {
   empRS.closeRecordStore();
}
catch( RecordStoreException e ){
   System.out.println( "Hiba az empRS bezárása közben!" );
}

System.out.println( "A deptRS bezárása" );

try {
   deptRS.closeRecordStore();
}
catch( RecordStoreException e ){
   System.out.println( "Hiba a deptRS bezárása közben!" );
}

System.out.println( "Rekordtároló analízis..." );

// Adatok analizálása...

RMSAnalyzer analyzer = new RMSAnalyzer(
          new RMSAnalyzer.SystemLogger( 10 ) );
```

```
      analyzer.analyzeAll();
   }

   protected void startApp()
                  throws MIDletStateChangeException {
      if( display == null ){
         display = Display.getDisplay( this );
         initMIDlet();
      }
   }

   public class MainForm extends Form {
      public MainForm(){
         super( "RMSMappings" );

         addCommand( exitCommand );
         addCommand( testCommand );

         setCommandListener( RMSMappings.this );
      }
   }
}
```

5. sz. melléklet: A FieldBasedRecordMapper osztály

```
import java.io.*;
import javax.microedition.rms.*;
import j2me.io.*;

// Alaposztály tetszőleges adatok olvasásához és írásához,
// amelyek FieldList-ként lettek definiálva.

public abstract class FieldBasedRecordMapper {

   // Néhány hasznos konstans

   public static Boolean TRUE = new Boolean( true );
   public static Boolean FALSE = new Boolean( false );

   // Fenntartott sztring típusú jelzők

   private static final byte NULL_STRING_MARKER = 0;
   private static final byte UTF_STRING_MARKER = 1;

   // A leképző konstruktora megadott lista alapján.
```

```
protected FieldBasedRecordMapper(){
}

// Bellítások előkészítése a bemeneti adatpuffer számára

protected void prepareForInput( byte[] data ){
   if( _bin == null ){
      _bin = new DirectByteArrayInputStream( data );
      _din = new DataInputStream( _bin );
   } else {
      _bin.setByteArray( data );
   }
}

// Tároló előkészítése a kimenet számára

protected void prepareForOutput(){
   if( _bout == null ){
      _bout = new DirectByteArrayOutputStream();
      _dout = new DataOutputStream( _bout );
   } else {
      _bout.reset();
   }
}

// Kiolvas egy mezőt a pufferből

protected Object readField( int type ) throws IOException {
   switch( type ){
      case FieldList.TYPE_BOOLEAN:
         return _din.readBoolean() ? TRUE : FALSE;
      case FieldList.TYPE_BYTE:
         return new Byte( _din.readByte() );
      case FieldList.TYPE_CHAR:
         return new Character( _din.readChar() );
      case FieldList.TYPE_SHORT:
         return new Short( _din.readShort() );
      case FieldList.TYPE_INT:
         return new Integer( _din.readInt() );
      case FieldList.TYPE_LONG:
         return new Long( _din.readLong() );
      case FieldList.TYPE_STRING: {
         byte marker = _din.readByte();
         if( marker == UTF_STRING_MARKER ){
            return _din.readUTF();
         }
      }
   }
```

```
    return null;
}

// Átkonvertál egy objektumot logikai típusú értékké

public static boolean toBoolean( Object value ){
    if( value instanceof Boolean ){
        return ((Boolean) value).booleanValue();
    } else if( value != null ){
        String str = value.toString().trim();

        if( str.equals( "true" ) ) return true;
        if( str.equals( "false" ) ) return false;

        return( toInt( value ) != 0 );
    }

    return false;
}

// Átkonvertál egy objektumot karakter típusú értékké

public static char toChar( Object value ){
    if( value instanceof Character ){
        return ((Character) value).charValue();
    } else if( value != null ){
        String s = value.toString();
        if( s.length() > 0 ){
            return s.charAt( 0 );
        }
    }

    return 0;
}

// Átkonvertál egy objektumot egész típusú értékké

public static int toInt( Object value ){
    if( value instanceof Integer ){
        return ((Integer) value).intValue();
    } else if( value instanceof Boolean ){
        return ((Boolean) value).booleanValue() ? 1 : 0;
    } else if( value instanceof Byte ){
        return ((Byte) value).byteValue();
    } else if( value instanceof Character ){
        return ((Character) value).charValue();
    } else if( value instanceof Short ){
```

```
        return ((Short) value).shortValue();
    } else if( value instanceof Long ){
        return (int) ((Long) value).longValue();
    } else if( value != null ){
        try {
            return Integer.parseInt( value.toString() );
        }
        catch( NumberFormatException e ){
        }
    }

    return 0;
}

// Átkonvertál egy objektumot hosszú egész típusú értékké

public static long toLong( Object value ){
    if( value instanceof Integer ){
        return ((Integer) value).longValue();
    } else if( value instanceof Boolean ){
        return ((Boolean) value).booleanValue() ? 1 : 0;
    } else if( value instanceof Byte ){
        return ((Byte) value).byteValue();
    } else if( value instanceof Character ){
        return ((Character) value).charValue();
    } else if( value instanceof Short ){
        return ((Short) value).shortValue();
    } else if( value instanceof Long ){
        return ((Long) value).longValue();
    } else if( value != null ){
        try {
            return Long.parseLong( value.toString() );
        }
        catch( NumberFormatException e ){
        }
    }

    return 0;
}

// Kiír egy mezőt a kimeneti pufferbe

protected void writeField( int type, Object value )
                        throws IOException {
    switch( type ){
        case FieldList.TYPE_BOOLEAN:
            _dout.writeBoolean( toBoolean( value ) );
            break;
```

```
        case FieldList.TYPE_BYTE:
           _dout.write( (byte) toInt( value ) );
           break;
        case FieldList.TYPE_CHAR:
           _dout.writeChar( toChar( value ) );
           break;
        case FieldList.TYPE_SHORT:
           _dout.writeShort( (short) toInt( value ) );
           break;
        case FieldList.TYPE_INT:
           _dout.writeInt( toInt( value ) );
           break;
        case FieldList.TYPE_LONG:
           _dout.writeLong( toLong( value ) );
           break;
        case FieldList.TYPE_STRING:
           if( value != null ){
              String str = value.toString();
              _dout.writeByte( UTF_STRING_MARKER );
              _dout.writeUTF( str );
           } else {
              _dout.writeByte( NULL_STRING_MARKER );
           }
           break;
     }
  }

  // Kiír egy mezőhalmazt a kimeneti folyamra

  protected byte[] writeStream( FieldList list,
                                Object[] fields )
                                    throws IOException {
     int count = list.getFieldCount();
     int len = ( fields != null ? fields.length : 0 );

     prepareForOutput();

     for( int i = 0; i < count; ++i ){
        writeField( list.getFieldType( i ),
                    ( i < len ? fields[i] : null ) );
     }

     return _bout.getByteArray();
  }

  private DirectByteArrayInputStream  _bin;
  private DirectByteArrayOutputStream _bout;
  private DataInputStream             _din;
```

```
    private DataOutputStream            _dout;
}
```