Informática Móvil

Diseño de aplicaciones con J2ME

Almacenamiento persistente de datos con RMS (Record Management System)

José Ramón Arias García
Ignacio Marín Prendes

UNIVERSIDAD DE OVIEDO
Área de Arquitectura y Tecnología de Computadores

Curso 2004/2005

Almacenamiento persistente de datos

- RMS (Record Management System)
  - API que proporciona a las aplicaciones MIDP persistencia de datos local (en el propio dispositivo)
    - Clases e interfaces RMS: definidas en *javax.microedition.rms*
  - Generalmente, única posibilidad de almacenamiento local de datos (pocos dispositivos usan sistemas de ficheros)
  - Conceptos claves básicos para el uso de RMS:
    - Registros (*Records*)
    - Almacenes de registros (*Record Stores*)

Registro

- Elemento individual de datos
- No hay restricciones acerca del contenido del registro
  - Secuencia de bytes (número, string, imagen, cualquier cosa representable como un "chorro" de bytes) con un tamaño máximo dependiente del sistema
- No existe el concepto de campos del registro
  - Es responsabilidad de quién escribe y lee registros interpretar la secuencia de bytes de un registro
  - Complica la tarea de crear aplicaciones
  - Permite que RMS sea simple y sencillo
- A nivel de API, registro = array de bytes

Almacenes de registros (I)

- Colección ordenada de cero ó más registros
- Todo registro pertenece a un almacen de registros
- Estructura de un almacen de registros
  - Nombre
  - Número de versión
  - Estampa de tiempo
  - Área de almacenamiento de registros

Almacenes de registros (II)

- **Nombre**
  - Identifica al almacen dentro de una suite de MIDlets
  - Cadena de 1-32 caracteres Unicode
  - Dentro de una suite, no puede haber más de dos almacenes con el mismo nombre
- **Nº de versión**
  - Valor entero actualizado automáticamente al insertar, modificar o borrar datos en el almacen de registros
- **Estampa de tiempo**
  - Entero de tipo *long*: representa el nº de milisegundos desde el 1 de enero de 1970 hasta el instante de la última modificación
- **Área de almacenamiento de registros**
  - Secuencia de registros (compuestos por dos elementos)
  - Identificador de registro + información del registro

Almacenes de registros (III)

- Al crear un registro en un almacen, se asigna un identificador único de registro (*record ID*)
  - Se asignan consecutivamente según se crean (1, 2, …)
  - Un *record ID* NO es un índice (si se borra un registro, no se "renumeran" los IDs de los demás)
- Estructura de un almacen de registros

CABECERA	
Nombre: miRecordStore	
Versión: 12	
Estampa de tiempo: 3106463320109	
RecordID	**Dato**
1	byte[]
2	byte[]
3	byte[]
...	...

Almacenes de registros (IV)

- Compartición de almacenes entre suites
  - MIDP 1.0: No pueden ser compartidos
  - MIDP 2.0: Se pueden compartir, siendo identificados por nombre_suite+creador(*vendor*)+nombre_del_almacen
- Mantienen estampa de tiempos e información de versión
  - Una aplicación puede saber cuándo se modificó por última vez un almacen
  - Una aplicación puede registrar un "listener" que le avise cuando un almacen concreto sea modificado
- A nivel de API, almacen de registros = instancia de *javax.microedition.rms.**RecordStore***

Límites de almacenamiento

- Variable entre dispositivos
  - MIDP: al menos 8 KB de memoria no volátil para almacenamiento persistente de datos
- RMS incluye métodos para determinar:
  - Tamaño de un registro individual
  - Tamaño total de un almacen de registros
  - Memoria disponible para almacenamiento de datos
- Uso del atributo MIDlet-Data-Size
  - Tanto en el manifiesto del JAR como en el JAD
  - Especifica el nº mín. de bytes necesario para almacenamiento de datos
  - Si el valor es demasiado grande, el AMS rechazará la instalación
  - Si el atributo no existe, el AMS supone que no almacenará datos
  - Algunas implementaciones MIDP obligan a definir atributos adicionales acerca de requisitos de almacenamiento

Otros aspectos de RMS

- Velocidad
  - Acceso a memoria persistente más lento que a memoria volátil
    - En algunas plataformas, escritura extremadamente lenta
  - Mejora de rendimiento
    - Cache de los datos más frecuentemente accedidos en memoria volátil
    - No hacer operaciones RMS en el hilo de eventos del MIDlet para mantener la respuesta de la IU
- Seguridad ante hilos
  - Las operaciones RMS son seguras ante hilos
  - Evidentemente, coordinar acceso entre hilos a un mismo almacen
  - La coordinación se refiere a hilos de distintos MIDlets, pues los almacenes de registros se comparten dentro del mismo MIDlet suite

Excepciones

- En general, los métodos del API RMS lanzan una o más excepciones particulares además de las excepciones estándar (como *java.lang.IllegalArgumentException*, p.e.)
- Las excepciones RMS forman parte de javax.microedition.rms:
  - **RecordStoreException:** superclase de las otras cuatro, lanzada cuando ocurren errores que estas otras no cubren
    - **InvalidRecordIDException**: una operación no puede ser realizada debido a que el *record ID* es inválido
    - **RecordStoreFullException**: no hay espacio disponible en memoria para el almacen
    - **RecordStoreNotFoundException**: la aplicación intenta abrir un almacen que no existe
    - **RecordStoreNotOpenException**: una aplicación intenta acceder a un almacen cuyo acceso había sido previamente cerrado

Operaciones con *RecordStore*: abrir

- No hay constructor
- Para abrir un *RecordStore*:

```
static RecordStore openRecordStore(String name, boolean createIfNecessary)
```

  - Hay otras 2 versiones sobrecargadas:

```
static RecordStore openRecordStore(String name, boolean createIfNec,
                                   int auth, boolean writeable)
```

auth=*AUTHMODE_PRIVATE*, *AUTHMODE_ANY*

```
static RecordStore openRecordStore(String name, String vendorName,
                                   String suiteName)
```

Operaciones con *RecordStore*: cerrar

- Para cerrar un *RecordStore*:

```
public void closeRecordStore() throws RecordStoreNotFoundException,
                                      RecordStoreException
```

Clases e interfaces auxiliares de *RecordStore*

- Clase ***RecordEnumeration***
  - Permite conocer el nº de registros de un *RecordStore*
  - Facilita el recorrido de un *RecordStore*:
    - acceder al reg. anterior/posterior
    - conocer el ID del registro siguiente o del actual
    - saber si el registro actual tiene un registro anterior/posterior
  - Permite recibir notificación de un cambio en un Record Store
  - Permite gestionar IDs (actualizarlos o devolverlos al estado inicial, p.e.)
  - Permite liberar los recursos ocupados por el objeto *RecordEnumeration*
- Interfaz ***RecordFilter***:
  - Facilita crear búsquedas eficientes en los registros
  - Crea objetos *RecordEnumeration* que incluyen registros de un RecordStore que cumplan una condición concreta
- Interfaz ***RecordComparator***:
  - Utilidad similar a RecordFilter, pero efectuando comparaciones entre campos de un registro

Métodos generales de *RecordStore*

MÉTODOS	DESCRIPCIÓN
String getName()	Devuelve el nombre del Record Store
int getVersion()	Devuelve la versión del Record Store
long getLastModified()	Devuelve la estampa de tiempos del Record Store
int getNumRecords()	Devuelve el número de registros del Record Store
int getSize()	Devuelve el número de bytes ocupado por el Record Store
int getSizeAvailable()	Devuelve el tamaño disponible para añadir registros
String[] listRecordStores()	Devuelve una lista con los nombres de los Record Stores que existen en la MIDlet suite
void deleteRecordStore(String name)	Elimina del dispositivo al Record Store cuo nombre es name
RecordEnumeration enumerateRecords(RecordFilter filter, RecordComparator comparator, boolean updated)	Crea un objeto RecordEnumeration que contiene todos los registros que cumplan una serie de condiciones
void addRecordListener (RecordListener listener)	Añade un 'listener'
void removeRecordListener (RecordListener listener)	Elimina un 'listener'

Métodos de *RecordStore* para acceso a registros

MÉTODOS	DESCRIPCIÓN
int addRecord(byte[] data, int offset, int numBytes)	Añade un registro al Record Store
void deleteRecord(int id)	Borra un registro del Record Store, identificado por su ID
int getNextRecordID()	Devuelve el ID del siguiente registro a insertar
byte[] getRecord(int id)	Devuelve el dato contenido en el registro cuyo ID es *id*
int getRecord(int id, byte[] buffer, int offset)	Devuelve en el parámetro *buffer* (a partir del byte *offset*) el dato contenido en un registro referenciado a través de su ID
int getRecordSize(int id)	Devuelve el tamaño (en bytes) de los datos contenidos en un registro
void setRecord(int id, byte[] newData, int offset, int size)	Sustituye los datos de un registro (identificado por su ID) por los *size* datos contenidos en *newData* (a partir del byte en la posición *offset*)

Ejemplo: Recorrido de un *RecordStore*

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

RecordStore rs=RecordStore.openRecordStore("nombre", false);

int tamEnBytes;

byte[] registroALeer = new byte[TAM_MAX_DE_DATO_EN_UN_REGISTRO];

for (int i=1;i<=rs.getNumRecords();i++){
    tamEnBytes = rs.getRecordSize(i);
    registroALeer = rs.getRecord(i);
    System.out.println("Registro "+i+": "+ new String(registroALeer, 0, tamEnBytes));
}

rs.closeRecordStore();
```

Métodos de *RecordEnumeration*

Método	Descripción
int numRecords()	Devuelve el número de registros
Byte[] nextRecord()	Devuelve el siguiente registro
int nextRecordId()	Devuelve el siguiente 'id' a devolver
Byte[] previousRecord()	Devuelve el registro anterior
int previousRecordId()	Devuelve el 'id' del registro anterior
boolean hasNextElement()	Devuelve true si existen más registros en adelante
boolean hasPreviousElement()	Devuelve true si existen más registros anteriores
void keepUpdated()	Provoca que los indices se actualicen cuándo se produzca algún cambio en el Record Store
boolean isKeptUpdated()	Devuelve true si los indices se actualizan al producirse algún cambio en el Record Store
void rebuild()	Actualiza los índices del RecordEnumeration
void reset()	Actualiza los índices a su estado inicial
void destroy()	Libera todos los recursos ocupados

Ejemplo: Recorrido con *RecordEnumeration*

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

RecordStore rs=RecordStore.openRecordStore("nombre", false);

int tamEnBytes;
byte[] registroALeer = new byte[TAM_MAX_DE_DATO_EN_UN_REGISTRO];
RecordEnumeration re = rs.enumerateRecords(null,null,false);

while (re.hasNextElement()){
   registroALeer = re.nextRecord();
   System.out.println("Registro "+i+": "+ new String(registroALeer));
}

rs.closeRecordStore();
```

Ejemplo: Uso de excepciones

- Es necesario utilizar excepciones para hacer el software robusto ante posibles errores
- Los ejemplos no suelen utilizarlas, por brevedad
- Ejemplo, apertura de acceso a un almacen de registros:

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

RecordStore rs;

try{
    rs=RecordStore.openRecordStore("nombre",true);
}
catch (RecordStoreNotFoundException e)
{
  System.out.println("No existe el Record Store \"nombre\" : "+e.toString());
}
catch (RecordStoreException e)
{
  System.out.println("Error al abrir el Record Store: "+e.toString());
}
```

Ejemplo: Escritura de un registro

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

String nuevoDato;
byte[] nuevoRegistro;
RecordStore rs=RecordStore.openRecordStore("nombre", false);

nuevoRegistro = nuevoDato.getBytes();
try{
    rs.addRecord(nuevoRegistro, 0, nuevoRegistro.length);
}
catch (RecordStoreException e){
   System.out.println("Error al insertar registro");
}

rs.closeRecordStore();
```

Uso práctico de registros

- Hasta ahora, registro = byte[]
- A partir de ahora, uso análogo al de una BD
- Solución inicial: campos de tamaño fijo
  - Problema: Desperdicio de espacio
- Mejor solución: Uso de flujos (*streams*) de bytes
  - Clases ***ByteArrayInputStream*** y ***DataInputStream***, (para lectura de datos)
  - Clases ***ByteArrayOutputStream*** y ***DataOutputStream***, (para escritura de datos)

Ejemplo: Escritura de un registro con dos campos

```
import java.io.*;                              // Necesario para poder utilizar streams de bytes
import javax.microedition.rms:

 // Escritura de un registro con dos  campos (nombre y teléfono)
String nuevoDato;
byte[] nuevoRegistro;
ByteArrayOutputStream flujoBytesSalida; DataOutputStream flujoSalida;
String nombre=”Manolo”;
long telefono=012345678;

RecordStore rs=RecordStore.openRecordStore(“nombre”, true);

flujoBytesSalida = new ByteArrayOutputStream();
flujoSalida = new DataOutputStream(flujoBytesSalida);

flujoSalida.writeUTF(nombre);
flujoSalida.writeLong(tel);
flujoSalida.flush();          // asegurar que se escribe físicamente en el RecordStore

nuevoRegistro = flujoBytesSalida.toByteArray();
rs.addRecord(nuevoRegistro, 0, nuevoRegistro.length);

flujoBytesSalida.close(); flujoSalida.close();

rs.closeRecordStore();
```

Ejemplo: Lectura de registros con dos campos

```
import java.io.*;                          // Necesario para poder utilizar streams de bytes
import javax.microedition.rms:

 // Lectura de un registro con dos  campos (nombre y teléfono)
byte[] registroALeer = new byte[TAM_MAX_DE_DATO_EN_UN_REGISTRO]
int tamanoEnBytes;
ByteArrayInputStream flujoBytesEntrada; DataInputStream flujoEntrada;

RecordStore rs=RecordStore.openRecordStore("nombre", false);

 flujoBytesEntrada = new ByteArrayInputStream(registroALeer);
 flujoEntrada = new DataInputStream(flujoBytesEntrada);
 for (int i=1;i<=rs.getNumRecords();i++){
     rs.getRecord(i, registroALeer, 0);
     System.out.println("Registro "+i);
     System.out.println("Nombre: "+flujoEntrada.readUTF()+" Telefono:"+flujoEntrada.readLong());
     flujoBytesEntrada.reset();
 }
 flujoBytesEntrada.close(); flujoEntrada.close();

rs.closeRecordStore();
```

Búsqueda de registros

- Mediante la interfaz *RecordFilter*
  - Devuelve un *RecordEnumeration* con los registros que coincidan con un patrón de búsqueda
- Cómo usar esta interfaz
  - Implementando *public boolean matches(byte [] candidato)*
  - Compara el registro candidato pasado como parámetro con el valor a buscar, devolviendo *true* en caso de que coincida

Ejemplo: Implementación de *RecordFilter*

```
public class Filtro implements RecordFilter{
   private String cadenaabuscar = null;
   public Filtro(String cadena){
     this.cadenaabuscar = cadena.toLowerCase();
   }
   public boolean matches(byte[] candidato){
     boolean resultado = false;
     String cadenacandidata;
     ByteArrayInputStream bais;
     DataInputStream dis;
     try{
       bais = new ByteArrayInputStream(candidato);
      dis = new DataInputStream(bais);
      cadenacandidata = dis.readUTF().toLowerCase();
     }
     catch (Exception e){
      return false;
     }
    if ((cadenacandidata != null) && (cadenacandidata.indexOf(cadenaabuscar)!= -1))
       return true;
    else
       return false;
  }
}
```

CREACIÓN DE UNA CLASE QUE IMPLEMENTA LA INTERFAZ *RecordFilter*

Ejemplo: Uso de *RecordFilter*

```
// Creación de un objeto de tipo Filtro que implementa la interfaz RecordFilter
Filtro buscar = new Filtro("UniOvi");
// Obtención del RecordEnumeration usando el filtro anterior
RecordEnumeration re = rs.enumerateRecords(buscar,null,false);
// Si algún registro cumple el patrón de búsqueda, imprimir un mensaje
if (re.numRecords() > 0)
    System.out.println("Patron 'UniOvi' encontrado");
```

USO DE LA CLASE *Filtro*

Ordenación de registros

- Mediante la interfaz *RecordComparator*
  - Devuelve un *RecordEnumeration* con los registros ordenados según un criterio de comparación
- Cómo usar esta interfaz
  - Implementando *public int compare(byte [] reg1, byte [] reg2)*
  - Compara entre los campos que se desee de los dos registros y el entero devuelto indica si reg1 va antes o después que reg2
  - El valor devuelto puede ser:
    - RecordComparator.*EQUIVALENT*
    - RecordComparator.*FOLLOWS* (reg1 sigue a reg2)
    - RecordComparator.*PRECEDES* (reg1 precede a reg2)

Ejemplo: Implementación de *RecordComparator*

```
public class Compara implements RecordComparator{
  public boolean compare(byte[] reg1, byte[] reg2){
    ByteArrayInputStream bais;
    DataInputStream dis;
    String cad1, cad2;
    try{
      bais = new ByteArrayInputStream(reg1);
      dis = new DataInputStream(bais);
      cad1 = dis.readUTF();
      bais = new ByteArrayInputStream(reg2);
      dis = new DataInputStream(bais);
      cad2 = dis.readUTF();
      int resultado = cad1.compareTo(cad2);
      if (resultado == 0)
        return RecordComparator.EQUIVALENT;
      else if (result < 0)
                return RecordComparator.PRECEDES;
            else
                return RecordComparator.FOLLOWS;
    }
    catch (Exception e){
      return RecordComparator.EQUIVALENT;
    }
  }
}
```

CREACIÓN DE UNA CLASE QUE IMPLEMENTA LA INTERFAZ *RecordComparator*

Ejemplo: Funcionamiento de *RecordComparator*

```
//Crear un objeto de tipo RecordComparator
Compara comp = new Compara();
//Obtengo el RecordEnumeration usando el objeto anterior, y con los registros ordenados
RecordEnumeration re = rs.enumerateRecords(null,comp,false);
```

USO DE LA CLASE *Compara*

- Las clases que implementan la interfaz *RecordComparator* pueden usarse para crear un *RecordEnumeration* cuyos registros están ordenados según un criterio

Gestión de eventos en un *RecordStore*

- Mediante la interfaz *RecordListener*
  - Funciona como los listener vistos en la API de alto nivel de IU
  - Cuando ocurre algún evento, se llama al método indicado para notificar un cambio

- Métodos de *RecordListener*

Método	Descripción
void recordAdded(RecordStore rs, int id)	Invocado cuándo un registro es añadido
void recordChanged(RecordStore rs, int id)	Invocado cuándo un registro es modificado
void recordDeleted(RecordStore rs, int id)	Invocado cuándo un registro es borrado

Ejemplo: Implementación de *RecordListener*

```
public class EjemploListener implements RecordListener{

  public void recordAdded(RecordStore rs, int id){
    try{
      String nombre = rs.getName();
      System.out.println("Registro "+id+" añadido al Record Store: "+nombre);
    }
    catch (Exception e){
      System.err.println(e);
    }
  }

  public void recordDeleted(RecordStore rs, int id){
    try{
      String nombre = rs.getName();
      System.out.println("Registro "+id+" borrado del Record Store: "+ nombre);
    }
    catch (Exception e){
      System.err.println(e);
    }
  }

  public void recordChanged(RecordStore rs, int id){
    try{
      String nombre = rs.getName();
      System.out.println("Registro "+id+" modificado del Record Store: "+nombre);
    }
    catch (Exception e){
      System.err.println(e);
    }
  }
}
```

CREACIÓN DE UNA CLASE QUE IMPLEMENTA LA INTERFAZ *RecordListener*

Ejemplo: Funcionamiento de *RecordListener*

```
// Abir acceso al RecordStore
RecordStore rs = openRecordStore("mi_almacen",true);

// Asociar al RecordStore una instancia de la clase EjemploListener, que implementa RecordListener
rs.addRecordListener( new PruebaListener());
```

USO DE LA CLASE *PruebaListener*

Informática Móvil

Diseño de aplicaciones con J2ME

Almacenamiento persistente de datos con RMS (Record Management System)

José Ramón Arias García
Ignacio Marín Prendes

UNIVERSIDAD DE OVIEDO
Área de Arquitectura y Tecnología de Computadores

Curso 2004/2005