

# Almacenamiento persistente con RMS

## Índice

1 Almacenes de registros.....	2
1.1 Abrir el almacén de registros.....	3
1.2 Listar los almacenes de registros.....	4
1.3 Eliminar un almacén de registros.....	4
1.4 Propiedades de los almacenes de registros.....	4
2 Registros.....	5
2.1 Almacenar información.....	5
2.2 Leer información.....	6
2.3 Borrar Registros.....	6
2.4 Almacenar y recuperar objetos.....	7
3 Navegar en el almacén de registros.....	7
3.1 Ordenación de registros.....	9
3.2 Filtrado de registros.....	9
4 Notificación de cambios.....	10
5 Optimización de consultas.....	11
6 Patrón de diseño adaptador.....	12

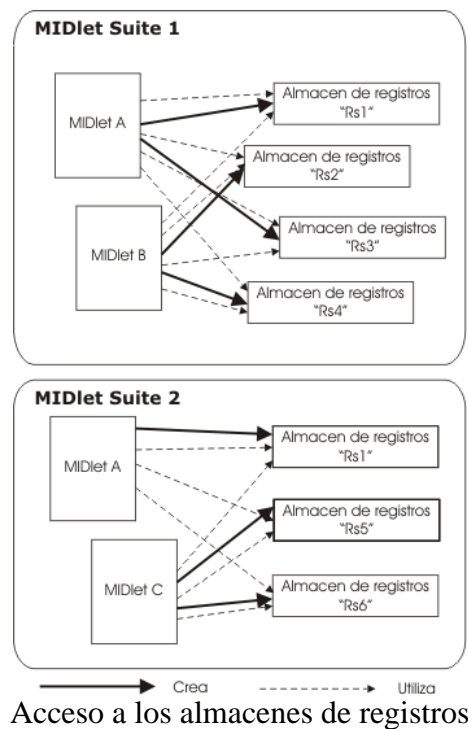
Muchas veces las aplicaciones necesitan almacenar datos de forma persistente. Cuando realizamos aplicaciones para PCs de sobremesa o servidores podemos almacenar esta información en algún fichero en el disco o bien en una base de datos. Lo más sencillo será almacenarla en ficheros, pero en los dispositivos móviles no podemos contar ni tan solo con esta característica. Aunque los móviles normalmente tienen su propio sistema de ficheros, por cuestiones de seguridad MIDP no nos dejará acceder directamente a él. Es posible que en alguna implementación podamos acceder a ficheros en el dispositivo, pero esto no es requerido por la especificación, por lo que si queremos que nuestra aplicación sea portable no deberemos confiar en esta característica.

Para almacenar datos de forma persistente en el móvil utilizaremos RMS (Record Management System). Se trata de un sistema de almacenamiento que nos permitirá almacenar registros con información de forma persistente en los dispositivos móviles. No se especifica ninguna forma determinada en la que se deba almacenar esta información, cada implementación deberá guardar estos datos de la mejor forma posible para cada dispositivo concreto, utilizando memoria no volátil, de forma que no se pierda la información aunque reiniciemos el dispositivo o cambiemos las baterías. Por ejemplo, algunas implementaciones podrán utilizar el sistema de ficheros del dispositivo para almacenar la información de RMS, o bien cualquier otro dispositivo de memoria no volátil que contenga el móvil. La forma de almacenamiento real de la información en el dispositivo será transparente para los MIDlets, éstos sólo podrán acceder a la información utilizando la API de RMS. Esta API se encuentra en el paquete `javax.microedition.rms`.

## 1. Almacenes de registros

La información se almacena en almacenes de registros (Record Stores), que serán identificados con un nombre que deberemos asignar nosotros. Cada aplicación podrá crear y utilizar tantos almacenes de registros como quiera. Cada almacén de registros contendrá una serie de registros con la información que queramos almacenar en ellos.

Los almacenes de registros son propios de la suite. Es decir, los almacenes de registro creados por un MIDlet dentro de una suite, serán compartidos por todos los MIDlets de esa suite, pero no podrán acceder a ellos los MIDlets de suites distintas. Por seguridad, no se permite acceder a recursos ni a almacenes de registros de suites distintas a la nuestra.



Cada suite define su propio espacio de nombres. Es decir, los nombres de los almacenes de registros deben ser únicos para cada suite, pero pueden estar repetidos en diferentes suites. Como hemos dicho antes, nunca podremos acceder a un almacén de registros perteneciente a otra suite.

### 1.1. Abrir el almacén de registros

Lo primero que deberemos hacer es abrir o crear el almacén de registros. Para ello utilizaremos el siguiente método:

```
RecordStore rs = RecordStore.open(nombre, true);
```

Con el segundo parámetro a `true` estamos diciendo que si el almacén de registros con nombre `nombre` no existiese en nuestra suite lo crearía. Si por el contrario estuviese a `false`, sólo intentaría abrir un almacén de registros existente, y si éste no existe se producirá una excepción `RecordStoreNotFoundException`.

El nombre que especificamos para el almacén de registros deberá ser un nombre de como mucho 32 caracteres codificado en Unicode.

Una vez hayamos terminado de trabajar con el almacén de registros, podremos cerrarlo con:

```
rs.close();
```

## 1.2. Listar los almacenes de registros

Si queremos ver la lista completa de almacenes de registros creados dentro de nuestra suite, podemos utilizar el siguiente método:

```
String [] nombres = RecordStore.listRecordStores();
```

Esto nos devolverá una lista con los nombres de los almacenes de registros que hayan sido creados. Teniendo estos nombres podremos abrirlos como hemos visto anteriormente para consultarlos, o bien eliminarlos.

## 1.3. Eliminar un almacén de registros

Podemos eliminar un almacén de registros existente proporcionando su nombre, con:

```
RecordStore.deleteRecordStore(nombre);
```

## 1.4. Propiedades de los almacenes de registros

Los almacenes de registros tienen una serie de propiedades que podemos obtener con información sobre ellos. Una vez hayamos abierto el almacén de registros para trabajar con él, podremos obtener los valores de las siguientes propiedades:

- **Nombre:** El nombre con el que hemos identificado el almacén de registros.

```
String nombre = rs.getName();
```

- **Estampa de tiempo:** El almacén de registros contiene una estampa de tiempo, que nos indicará el momento de la última modificación que se ha realizado en los datos que almacena. Este instante de tiempo se mide en milisegundos desde el 1 de enero de 1970 a las 0:00, y podemos obtenerlo con:

```
long timestamp = rs.getLastModified();
```

- **Versión:** También tenemos una versión del almacén de registros. La versión será un número que se incrementará cuando se produzca cualquier modificación en el almacén de registros. Esta propiedad, junto a la anterior, nos será útil para tareas de sincronización de datos.

```
int version = rs.getVersion();
```

- **Tamaño:** Nos dice el espacio en bytes que ocupa el almacén de registros actualmente.

```
int tam = rs.getSize();
```

- **Tamaño disponible:** Nos dice el espacio máximo que podrá crecer este almacén de registros. El dispositivo limitará el espacio asignado a cada almacén de registros, y con este método podremos saber el espacio restante que nos queda.

```
int libre = rs.getSizeAvailable();
```

## 2. Registros

El almacén de registros contendrá una serie de registros donde podemos almacenar la información. Podemos ver el almacén de registros como una tabla en la que cada fila corresponde a un registro. Los registros tienen un identificador y un array de datos.

Identificador	Datos
1	array de datos ...
2	array de datos ...
3	array de datos ...
...	...

Estos datos de cada registro se almacenan como un array de bytes. Podremos acceder a estos registros mediante su identificador o bien recorriendo todos los registros de la tabla.

Cuando añadamos un nuevo registro al almacén se le asignará un identificador una unidad superior al identificador del último registro que tengamos. Es decir, si añadimos dos registros y al primero se le asigna un identificador  $n$ , el segundo tendrá un identificador  $n+1$ .

Las operaciones para acceder a los datos de los registros son atómicas, por lo que no tendremos problemas cuando se acceda concurrentemente al almacén de registros.

### 2.1. Almacenar información

Tenemos dos formas de almacenar información en el almacén de registros. Lo primero que deberemos hacer en ambos casos es construir un array de bytes con la información que queramos añadir. Para hacer esto podemos utilizar un flujo `DataOutputStream`,

como se muestra en el siguiente ejemplo:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);

dos.writeUTF(nombre);
dos.writeInt(edad);

byte [] datos = baos.toByteArray();
```

Una vez tenemos el array de datos que queremos almacenar, podremos utilizar uno de los siguientes métodos del objeto almacén de datos:

```
int id = rs.addRecord(datos, 0, datos.length);
rs.setRecord(id, datos, 0, datos.length);
```

En el caso de `addRecord`, lo que se hace es añadir un nuevo registro al almacén con la información que hemos proporcionado, devolviéndonos el identificador `id` asignado al registro que acabamos de añadir.

Con `setRecord` lo que se hace es sobrescribir el registro correspondiente al identificador `id` indicado con los datos proporcionados. En este caso no se añade ningún registro nuevo, sólo se almacenan los datos en un registro ya existente.

## 2.2. Leer información

Si tenemos el identificador del registro que queremos leer, podemos obtener su contenido como array de bytes directamente utilizando el método:

```
byte [] datos = rs.getRecord(id);
```

Si hemos codificado la información dentro de este registro utilizando un flujo `DataOutputStream`, podemos decodificarlo realizando el proceso inverso con un flujo `DataInputStream`:

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);
DataInputStream dis = DataInputStream(bais);

String nombre = dis.readUTF();
String edad = dis.readInt();

dis.close();
```

## 2.3. Borrar Registros

Podremos borrar un registro del almacén a partir de su identificador con el siguiente método:

```
rs.deleteRecord(id);
```

## 2.4. Almacenar y recuperar objetos

Si hemos definido una forma de serializar los objetos, podemos aprovechar esta serialización para almacenar los objetos de forma persistente en RMS y posteriormente poder recuperarlos.

Imaginemos que en nuestra clase `MisDatos` hemos definido los siguientes métodos para serializar y deserializar tal como vimos en el apartado de entrada/salida:

```
public void serialize(OutputStream out)
public static MisDatos deserialize(InputStream in)
```

Podemos serializar el objeto en un array de bytes utilizando estos métodos para almacenarlo en RMS de la siguiente forma:

```
MisDatos md = new MisDatos();
...
ByteArrayOutputStream baos = new ByteArrayOutputStream();
md.serialize(baos);

byte [] datos = baos.toByteArray();
```

Una vez tenemos este array de bytes podremos almacenarlo en RMS. Cuando queramos recuperar el objeto original, leeremos el array de bytes de RMS y deserializaremos el objeto de la siguiente forma:

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);
MisDatos md = MisDatos.deserialize(bais);
```

## 3. Navegar en el almacén de registros

Si no conocemos el identificador del registro al que queremos acceder, podremos recorrer todos los registros del almacén utilizando un objeto `RecordEnumeration`. Para obtener la enumeración de registros del almacén podemos utilizar el siguiente método:

```
RecordEnumeration re = rs.enumerateRecords(null, null, false);
```

Con los dos primeros parámetros podremos establecer la ordenación y el filtrado de los registros que se enumeren como veremos más adelante. Por ahora vamos a dejarlo a `null` para obtener la enumeración con todos los registros y en un orden arbitrario. Esta es la forma más eficiente de acceder a los registros.

El tercer parámetro nos dice si la enumeración debe mantenerse actualizada con los registros que hay realmente almacenados, o si por el contrario los cambios que se realicen en el almacén después de haber obtenido la enumeración no afectarán a dicha enumeración. Será más eficiente establecer el valor a `false` para evitar que se tenga que mantener actualizado, pero esto tendrá el inconveniente de que puede que alguno de los registros de la enumeración se haya borrado o que se hayan añadido nuevos registros que no constan en la enumeración. En el caso de que especifiquemos `false` para que no actualice automáticamente la enumeración, podremos forzar manualmente a que se actualice invocando el método `rebuild` de la misma, que la reconstruirá utilizando los nuevos datos.

Recorreremos la enumeración de registros de forma similar a como recorreremos los objetos `Enumeration`. Tendremos un cursor que en cada momento estará en uno de los elementos de la enumeración. En este caso podremos recorrer la enumeración de forma bidireccional.

Para pasar al siguiente registro de la enumeración y obtener sus datos utilizaremos el método `nextRecord`. Podremos saber si existe un siguiente registro llamando a `hasNextElement`. Nada más crear la enumeración el cursor no se encontrará en ninguno de los registros. Cuando llamemos a `nextRecord` por primera vez se situará en el primer registro y nos devolverá su array de datos. De esta forma podremos seguir recorriendo la enumeración mientras haya más registros. Un bucle típico para hacer este recorrido es el siguiente:

```
while(re.hasNextElement()) {  
    byte [] datos = re.nextRecord();  
    // Procesar datos obtenidos  
    ...  
}
```

Hemos dicho que el recorrido puede ser bidireccional. Por lo tanto, tenemos un método `previousRecord` que moverá el cursor al registro anterior devolviéndonos su contenido. De la misma forma, tenemos un método `hasPreviousElement` que nos dirá si existe un registro anterior. Si invocamos `previousRecord` nada más crear la enumeración, cuando el cursor todavía no se ha posicionado en ningún registro, moverá el cursor al último registro de la enumeración devolviéndonos su resultado. Podemos también volver al estado inicial de la enumeración en el que el cursor no apunta a ningún registro llamando a su método `reset`.

En lugar de obtener el contenido de los registros puede que nos interese obtener su identificador, de forma que podamos eliminarlos o hacer otras operaciones con ellos. Para



ello tenemos los métodos `nextRecordId` y `previousRecordId`, que tendrán el mismo comportamiento que `nextRecord` y `previousRecord` respectivamente, salvo porque devuelven el identificador de los registros recorridos, y no su contenido.

### 3.1. Ordenación de registros

Puede que nos interese que la enumeración nos ofrezca los registros en un orden determinado. Podemos hacer que se ordenen proporcionando nosotros el criterio de ordenación. Para ello deberemos crear un comparador de registros que nos diga cuando un registros es mayor, menor o igual que otro registro. Para crear este comparador deberemos crear una clase que implemente la interfaz `RecordComparator`:

```
public class MiComparador implements RecordComparator {
    public int compare(byte [] reg1, byte [] reg2) {
        if( /* reg1 es anterior a reg2 */ ) {
            return RecordComparator.PRECEDES;
        } else if( /* reg1 es posterior a reg2 */ ) {
            return RecordComparator.FOLLOWS;
        } else if( /* reg1 es igual a reg2 */ ) {
            return RecordComparator.EQUIVALENT;
        }
    }
}
```

De esta manera, dentro del código de esta clase deberemos decir cuando un registro va antes, después o es equivalente a otro registro, para que el enumerador sepa cómo ordenarlos. Ahora, cuando creemos el enumerador deberemos proporcionarle un objeto de la clase que hemos creado para que realice la ordenación tal como lo hayamos especificado en el método `compare`:

```
RecordEnumeration re =
    rs.enumerateRecords(new MiComparador(), null, false);
```

Una vez hecho esto, podremos recorrer los registros del enumerador como hemos visto anteriormente, con la diferencia de que ahora obtendremos los registros en el orden indicado.

### 3.2. Filtrado de registros

Es posible que no queramos que el enumerador nos devuelva todos los registros, sino sólo los que cumplan unas determinadas características. Es posible realizar un filtrado para que el enumerador sólo nos devuelva los registros que nos interesan. Para que esto sea posible deberemos definir qué características cumplen los registros que nos interesan.

Esto lo haremos creando una clase que implemente la interfaz `RecordFilter`:

```
public class MiFiltro implements RecordFilter {
    public boolean matches(byte [] reg) {
        if( /* reg nos interesa */ ) {
            return true;
        } else {
            return false;
        }
    }
}
```

De esta forma dentro del método `matches` diremos si un determinado registro nos interesa, o si por lo contrario debe ser filtrado para que no aparezca en la enumeración. Ahora podremos proporcionar este filtro al crear la enumeración para que filtre los registros según el criterio que hayamos especificado en el método `matches`:

```
RecordEnumeration re =
    rs.enumerateRecords(null, new MiFiltro(), false);
```

Ahora cuando recorramos la enumeración, sólo veremos los registros que cumplan los criterios impuestos en el filtro.

## 4. Notificación de cambios

Es posible que queramos que en cuanto haya un cambio en el almacén de registros se nos notifique. Esto ocurrirá por ejemplo cuando estemos trabajando con la copia de los valores de un conjunto de registros en memoria, y queramos que esta información se mantenga actualizada con los últimos cambios que se hayan producido en el almacén.

Para estar al tanto de estos cambios deberemos utilizar un listener, que escuche los cambios en el almacén de registros. Este listener lo crearemos implementando la interfaz `RecordListener`, como se muestra a continuación:

```
public class MiListener implements RecordListener {
    public void recordAdded(RecordStore rs, int id) {
        // Se ha añadido un registro con identificador id a rs
    }

    public void recordChanged(RecordStore rs, int id) {
        // Se ha modificado el registro con identificador id en rs
    }

    public void recordDeleted(RecordStore rs, int id) {
        // Se ha eliminado el registro con identificador id de rs
    }
}
```

```
}  
}
```

De esta forma dentro de estos métodos podremos indicar qué hacer cuando se produzca uno de estos cambios en el almacén de registros. Para que cuando se produzca un cambio en el almacén de registros se le notifique a este listener, deberemos añadir el listener en el correspondiente almacén de registros de la siguiente forma:

```
rs.addRecordListener(new MiListener());
```

De esta forma cada vez que se realice alguna operación en la que se añadan, eliminen o modifiquen registros del almacén se le notificará a nuestro listener para que éste pueda realizar la operación que sea necesaria.

Por ejemplo, cuando creamos una enumeración con registros poniendo a `true` el parámetro para que mantenga en todo momento actualizados los datos de la enumeración, lo que hará será utilizar un listener para ser notificada de los cambios que se produzcan en el almacén. Cada vez que se produzca un cambio, el listener hará que los datos de la enumeración se actualicen.

## 5. Optimización de consultas

Hemos visto que podemos realizar consultas en el almacén utilizando filtrado. Con el objeto `RecordFilter` podemos obtener un conjunto de registros que cumpla ciertas condiciones.

Por ejemplo, imaginemos una aplicación de agenda en la que tengamos almacenadas una serie de citas. Para cada cita tenemos fecha y hora de la cita, asunto, descripción, lugar de la reunión, nombre de la persona de contacto, y la posibilidad de activar una alarma para que el móvil nos avise cuando llegue la hora de la reunión.

```
public class Cita {  
    Date fecha;  
    String asunto;  
    String descripcion;  
    String lugar;  
    String contacto;  
    boolean alarma;  
}
```

En esta aplicación nos interesará obtener aquellas citas que tengan programada una alarma todavía pendiente, es decir, con una fecha posterior a la actual, para que la aplicación pueda activar estas alarmas cuando sea necesario. Esta será una consulta que se hará muy frecuentemente.

Como los datos están almacenados codificados en binario, para buscar aquellos registros que cumplan los criterios de búsqueda deseados tendremos que recorrer todo el conjunto de registros y descodificarlos para comprobar si cumplen estos criterios. Esto nos obligará a leer todos los datos almacenados cada vez que queramos obtener un subconjunto de ellos.

Podemos optimizar esta consulta creando un índice para el conjunto de registros. Para ello crearemos un nuevo almacén de registros donde almacenaremos los índices. De esta forma tendremos un almacén de datos y un almacén de índices. En el almacén de índices tendremos un registro por cada registro existente en el almacén de datos. Cada índice contendrá como datos el identificador del registro al que representa, y además aquellos campos de este registro que utilizamos frecuentemente para realizar las búsquedas.

De esta forma, podremos realizar búsquedas en el almacén de índices, en lugar de hacerlas directamente en el almacén de datos. Buscaremos aquellos índices que cumplan los criterios de búsqueda, y obtendremos los identificadores almacenados en estos índices. Con estos identificadores podremos acceder directamente a los registros buscados en el almacén de datos, habiendo evitado de esta forma tener que recorrer este almacén completo.

Por ejemplo, en el caso de nuestra aplicación de agenda, los índices contendrán para cada registro el fecha de la cita y el flag que nos indica si la alarma está activada. De esta forma, las búsquedas que realizamos más frecuentemente se podrán realizar de forma optimizada.

```
public class IndiceCita {  
    int id;  
    Date fecha;  
    boolean alarma;  
}
```

## 6. Patrón de diseño adaptador

Para implementar el acceso a RMS en nuestra aplicación es conveniente utilizar el patrón de diseño adaptador.

Un adaptador es una interfaz adaptada a las necesidades concretas de nuestra aplicación, que encapsula el acceso a una API genérica y nos aísla de ella.

En este caso, el adaptador será una clase que encapsulará todo el acceso a RMS, ofreciéndonos una serie de métodos para acceder a los tipos de datos concretos utilizados en nuestra aplicación. Por ejemplo, mientras en la API de RMS tenemos un método genérico `getRecord` para acceder a un registro, en el adaptador de nuestra aplicación tendremos un método `getCita` que leerá una cita de RMS.

En el caso de la agenda el adaptador podría ofrecernos los siguientes métodos:

```
Cita [] listaCitas();
int addCita(Cita cita);
void updateCita(Cita cita);
void removeCita(int id);
Cita getCita(id);
```

De esta forma aislaremos el resto del código de la forma en la que se encuentran almacenados los datos. Todo el código RMS estará dentro del adaptador. Por ejemplo, podemos tener un adaptador como el siguiente:

```
public class AdaptadorRMS {

    // Nombres de los almacenes
    public final static String RS_INDICE = "rs_indice";
    public final static String RS_DATOS = "rs_datos";

    // Almacenes de registros
    RecordStore rsIndice;
    RecordStore rsDatos;

    public AdaptadorRMS() throws RecordStoreException {
        rsIndice = RecordStore.openRecordStore(RS_INDICE, true);
        rsDatos = RecordStore.openRecordStore(RS_DATOS, true);
    }

    /*
     * Obtiene todas las citas
     */
    public Cita[] listaCitas() throws RecordStoreException,
    IOException {
        RecordEnumeration re = rsDatos.enumerateRecords(null, null,
        false);

        Vector citas = new Vector();

        while (re.hasNextElement()) {
            int id = re.nextRecordId();
            byte[] datos = rsDatos.getRecord(id);

            ByteArrayInputStream bais = new ByteArrayInputStream(datos);
            DataInputStream dis = new DataInputStream(bais);

            Cita cita = Cita.deserialize(dis);
            cita.setRmsID(id);
            citas.addElement(cita);
        }

        Cita[] result = new Cita[citas.size()];
        citas.copyInto(result);

        return result;
    }
}
```

```

/*
 * Agrega una cita
 */
public int addCita(Cita cita) throws IOException,
RecordStoreException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);

    cita.serialize(dos);
    byte[] datos = baos.toByteArray();

    int id = rsDatos.addRecord(datos, 0, datos.length);
    cita.setRmsID(id);
    return id;
}

/*
 * Elimina una cita
 */
public void removeCita(int id) throws RecordStoreException {
    rsDatos.deleteRecord(id);
}

/*
 * Actualiza una cita
 */
public void updateCita(Cita cita) throws IOException,
RecordStoreException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);
    cita.serialize(dos);

    byte[] datos = baos.toByteArray();
    rsDatos.setRecord(cita.getRmsID(), datos, 0, datos.length);
}

/*
 * Obtiene una cita
 */
public Cita getCita(int id) throws RecordStoreException,
IOException {
    byte[] datos = rsDatos.getRecord(id);

    ByteArrayInputStream bais = new ByteArrayInputStream(datos);
    DataInputStream dis = new DataInputStream(bais);

    Cita cita = Cita.deserialize(dis);
    cita.setRmsID(id);
    return cita;
}

...

/*
 * Cierra los almacenes de registros
 */
public void cerrar() throws RecordStoreException {
    rsIndice.closeRecordStore();
    rsDatos.closeRecordStore();
}

```

```
}
}
```

## Clave primaria

Para poder referenciar un determinado registro necesitaremos que tenga una clave primaria que lo identifique. Como clave primaria podemos utilizar el ID que RMS asigna a cada registro. Para utilizar este valor como clave primaria, deberemos añadirlo como atributo al objeto que encapsule nuestros datos. Por ejemplo, en el caso de las citas añadiremos un nuevo atributo `rmsID` a la clase `Cita` en el que almacenaremos esta clave primaria:

```
public class Cita {
    int rmsID;

    Date fecha;
    String asunto;
    String descripcion;
    String lugar;
    String contacto;
    boolean alarma;
}
```

En el ejemplo del adaptador anterior hemos visto como en cada operación en la que se obtienen citas, se almacena en ellas su identificador, por si posteriormente quisiéramos modificar o eliminar dicha cita.

## Gestión de índices

Además podemos añadir al adaptador el código necesario para gestionar los índices. Al igual que en el caso anterior, también deberemos definir una clave primaria para los índices de forma que puedan modificarse o eliminarse posteriormente. En este ejemplo utilizamos como clave primaria el ID asignado al registro por RMS.

```
/*
 * Busca citas con alarma posterior a la fecha indicada
 */
public IndiceCita[] buscaCitas(Date fecha, boolean alarma)
    throws RecordStoreException,
IOException {
    RecordEnumeration re = rsIndice.enumerateRecords(
        new FiltroIndice(fecha, alarma), new
OrdenIndice(), false);

    Vector indices = new Vector();

    while (re.hasNextElement()) {
        int id = re.nextRecordId();
        byte[] datos = rsIndice.getRecord(id);

        ByteArrayInputStream bais = new ByteArrayInputStream(datos);
```

```

        DataInputStream dis = new DataInputStream(bais);

        IndiceCita indice = IndiceCita.deserialize(dis);
        indice.setRmsID(id);
        indices.addElement(indice);
    }

    IndiceCita[] result = new IndiceCita[indices.size()];
    indices.copyInto(result);
    return result;
}

/*
 * Agrega un indice
 */
public int addIndice(IndiceCita indice)
    throws IOException,
RecordStoreException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);
    indice.serialize(dos);
    byte[] datos = baos.toByteArray();
    int id = rsIndice.addRecord(datos, 0, datos.length);
    return id;
}

```

Para realizar la búsqueda de índices de citas posteriores a una determinada fecha con alarma podemos utilizar un filtro de índices como el siguiente:

```

/*
 * Filtra fechas posteriores con alarma
 */
class FiltroIndice implements RecordFilter {
    Date fecha;
    boolean alarma;

    public FiltroIndice(Date fecha, boolean alarma) {
        this.fecha = fecha;
        this.alarma = alarma;
    }

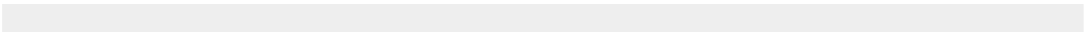
    public boolean matches(byte[] datos) {
        try {
            ByteArrayInputStream bais = new ByteArrayInputStream(datos);
            DataInputStream dis = new DataInputStream(bais);
            IndiceCita indice = IndiceCita.deserialize(dis);

            return indice.isAlarma() == this.alarma
                && indice.getFecha().getTime() >=
this.fecha.getTime();

        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }
    }
}

```





Este filtro podemos encapsularlo en el mismo adaptador.

