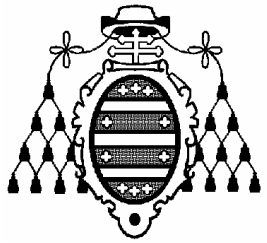
UNIVERSIDAD DE OVIEDO



ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA EN INFORMÁTICA DE OVIEDO

INFORMÁTICA MÓVIL

Práctica 2:
RMS (2ª parte) y WMA

(Java Micro Edition)

CURSO 2007-2008

TABLA DE CONTENIDOS

1. OBJETIVOS

Esta sesión práctica pretende avanzar en el uso del API RMS, a partir de la aplicación de ejemplo creada en las sesiones anteriores. En esta ocasión, se trabajará con el uso de las clases *RecordFilter* y *RecordComparator* para efectuar búsquedas y ordenar registros. Finalmente, se introducirá la capacidad de enviar/recibir información mediante el uso de mensajes SMS.

2. ORDENACIÓN DE REGISTROS

Para implementar un criterio de ordenación de los registros presentes en un *RecordStore*, es necesario crear una clase que implemente la **interfaz *RecordComparator***. Para ello, añadiremos en nuestra suite de midlets "*Contactos*" una nueva *Java Class* llamada *ComparaPorOrdAlfNombre*, importaremos las clases presentes en los paquetes *javax.microedition.rms* y *java.io*, y añadiremos en la definición de la clase que ésta implementa la interfaz *RecordComparator*. Para ello, la clase debe implementar el método *compare()*, y ofrecerá la posibilidad de insertar automáticamente ese método. Si se acepta esa opción, el código de *ComparaPorOrdAlfNombre.java* tendrá el siguiente aspecto:

```
package Contactos;

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

class ComparaPorOrdAlfNombre implements RecordComparator {

   public int compare(byte[] values, byte[] values1) {

   }

}
```

El método *compare()* recibe dos vectores de bytes (con el contenido de dos registros) y deberá retornar un entero que indique el resultado de la comparación entre los contenidos de ambos registros, según un criterio establecido en el código de este método:

- Si los dos registros se consideran iguales, *compare()* debe retornar la constante *RecordComparator.EQUIVALENT*.

- Si se considera que el primer registro es anterior en la ordenación al segundo, *compare()* debe retornar la constante *RecordComparator.PRECEDES*.

- Si se considera que el primer registro es posterior en la ordenación al segundo, *compare()* debe retornar la constante *RecordComparator.FOLLOWS*.

Para esta clase, que pretende que los registros estén ordenados por nombre en orden alfabético, será necesario extraer el nombre de los dos contactos (de *values* y de *values1*, respectivamente) mediante la utilización de las clases *ByteArrayInputStream* y *DataInputStream*. Después, se compararán ambos *String*s en función del criterio de orden alfabético. Para ello se podrá utilizar el método *compareTo()* de la clase *String*. Se recomienda pasar los dos nombres a minúsuculas (con el método *String.toLowerCase()*; o los

dos a mayúsculas, con *String.toUpperCase()*) para que la ordenación alfabética sea correcta.

El método *String.compareTo()* retorna:

- **Cero**, si dos *String*s son idénticos.

- Un valor **menor que cero**, si el *String* que invoca al método *compareTo()* precede alfabéticamente al *String* que le es pasado al método como parámetro.

- Un valor **mayor que cero**, en el caso contrario al anterior.

Si no se capturan excepciones en este método, el IDE mostrará errores al compilar la clase. Una gestión básica de excepciones sería la de incluir todo el código en un bloque *try* que capture todas las excepciones (*java.lang.Exception*) y que, por ejemplo, el método *ComparaPorOrdAlfNombre.compare()* retorne *RecordComparator.EQUIVALENT* ante una excepción.

Una vez implementada la clase, y antes de ver cómo usarla, crearemos otras dos clases que servirán para crear otros dos criterios de ordenación de registros.

2.1. ORDENACIÓN ALFABÉTICA POR APELLIDOS

Agrega una nueva clase a la suite, con nombre *ComparaPorOrdAlfApellidos*, que será utilizada para ordenar alfabéticamente por apellidos los contactos almacenados en el *RecordStore*. Esta clase es prácticamente idéntica a *ComparaPorOrdAlfNombre*, pero las dos cadenas a comparar en el método *compare()* serán la concatenación del primer y segundo apellido de cada uno de los contactos. Para la concatenación, puedes utilizar el método *concatTo* de la clase String o el operador +. Implementa una gestión de excepciones igual a la utilizada en la clase anterior.

2.2. ORDENACIÓN POR EDAD

Añade otra clase a la suite, cuyo nombre será *ComparaPorEdad*. En esta clase, el método *compare()* se encargará de extraer la edad de cada uno de los dos *byte[]* que le son pasados como parámetros y retornará la constante apropiada. De nuevo, por defecto, ante cualquier error se retornará *RecordComparator.EQUIVALENT*.

2.3. ORDENACIÓN DE LOS CONTACTOS POR NOMBRE, APELLIDOS O EDAD

Para obtener una ordenación de los contactos en base a cualquiera de los criterios implementados mediante las tres clases desarrolladas hasta el momento en esta sesión, se debe crear un objeto de la clase que se desee utilizar y crear un *RecordEnumeration* a partir del *RecordStore* mediante el método *enumerateRecords*, pasando como **segundo parámetro** el objeto que implementa la interfaz *RecordComparator* (en las transparencias de teoría, aparece un ejemplo).

Para que la presentación ordenada de registros se integre con nuestra aplicación, habrá que programar lo siguiente:

- Añadir un nuevo *Command* al menú del MIDlet *ContactosMIDlet* (con texto “Ordenar

contactos").

- Al seleccionar ese *Command*, se mostrará un nuevo diálogo *List*[1] de tipo *List.EXCLUSIVE*. Ese diálogo tendrá tres opciones (ordenación alfabética por nombre, por apellidos, y ordenación por edad). Además, el diálogo *List* incorporará dos *Command* más para aceptar o cancelar.

- Si se elige "aceptar", se comprobará cuál de las tres opciones del List se seleccionó (utilizando *this.menuModoPresentacion.getSelectedIndex()*, en el supuesto de que el objeto *List* se llamase *menuModoPresentacion*); a continuación, **se destruirá el *RecordEnumerator* que almacenaba nuestros contactos y se volverá a crear a partir del *RecordStore*** (pero aplicando la clase apropiada como segundo parámetro de *enumerateRecords()*). De esta forma, el **resto de código seguirá viendo el mismo RecordEnumeration, pero su contenido serán los registros ordenados** en base al criterio seleccionado en el *List*.

3. BÚSQUEDA DE REGISTROS

Para efectuar una búsqueda de registros, se debe crear una nueva clase MIDP que implemente, en esta ocasión, la **interfaz *RecordFilter***. Crea una nueva clase de este tipo, con el nombre *BuscaProfesion*, en la suite de MIDlets *Contactos*. Esta clase permitirá crear un *RecordEnumeration* con todos los contactos que ejercen una determinada profesión; es decir, seleccionará todos los contactos cuyo miembro *String profesion* sea igual a un *String* dado. Antes de implementar el código de la clase, vamos a ver cómo se utilizaría:

- En primer lugar, crearíamos un objeto *BuscaProfesion*, pasándole como único parámetro del constructor el *String* con la profesión a buscar.

- Finalmente, crearíamos un objeto *RecordEnumeration* utilizando como segundo parámetro de *enumerateRecords()* el objeto *BuscaProfesion* creado en el paso anterior.

En las transparencias de teoría aparece un ejemplo que clarifica lo expuesto anteriormente. En nuestro caso, habrá que crear dentro de la clase *BuscaProfesion*:

- Un miembro privado *String* que almacene la profesión a buscar.

- Un constructor con un parámetro tipo *String* para asignar a ese miembro privado un valor cuando creamos una instancia de *BuscaProfesion*.

- El método público *matches()*, que recibe como parámetro un *byte[]* (el contenido de un registro) y retornará un *boolean* según se considere que los datos del registro cumplen (*true*) o no (*false*) el criterio de búsqueda.

Para establecer que el contenido de un registro (*byte[] values*) casa con nuestro criterio de búsqueda, habrá que comparar (con *String.equals()*) el miembro privado *String* de *BuscaProfesion* con el valor de la cadena *profesion* dentro del *byte[]*. Obviamente, para leer ese dato, habrá que usar las clases *ByteArrayInputStream* y *DataInputStream* del paquete

[1] Un objeto List debe ser tratado igual que un Form para atender a la selección de un Command por parte del usuario.

java.io. Antes de comparar los dos *String*s, deberían haber sido convertidos ambos a la misma capitalidad (mayúsculas o minúsculas).

3.1. INTEGRACIÓN DE LA BÚSQUEDA POR PROFESIÓN EN LA SUITE DE MIDLETS CONTACTOS

- Añade un nuevo *Command* en el menú principal del MIDlet *ContactosMIDlet*. El texto que presentará en el menú será "Buscar profesión".

- Si se selecciona ese *Command*, se deberá mostrar un diálogo *TextBox*[2] que pida introducir la "Profesión a buscar".

- Ese *TextBox* ofrecerá las opciones *Cancelar* y *Aceptar* mediante sendos *Command*.

- Si se elige la opción *Aceptar*, se mostrará un nuevo formulario[3] que ofrecerá un elemento por cada contacto que ejerza la profesión especificada a través del *TextBox* anterior. Además, el formulario incluirá un *Command* para volver *Atrás*. Se puede gestionar este formulario de dos formas (elige la que prefieras):

    - Cada vez que se pulse *Aceptar* en el *TextBox* anterior, se crea el formulario, se crea el *RecordEnumeration* con los contactos que ejerzan la profesión especificada, y se añaden los elementos y el Command para volver atrás. Al pulsar sobre la opción *Atrás* se destruye el formulario, el *RecordEnumeration*, el *Command* y se muestra la pantalla de navegación.

    - Se crea el formulario vacío en el constructor del MIDlet y se le añade el *Command*; en *startApp()* se registra el formulario para que "escuche" a los Command. Cuando se pulse *Aceptar* en el *TextBox* se generará el *RecordEnumeration* y se recorrerá éste añadiendo un elemento al formulario por cada registro presente. Cuando se pulse *Atrás* en el formulario, se destruirá el RecordEnumeration y los elementos en el formulario (pero no el formulario).

La disyunción entre estas dos estrategias puede haber surgido varias veces en la confección de la aplicación y es el diseñador del software quien debe elegir la apropiada, según se desee minimizar el uso de memoria (primera opción) o de CPU (segunda opción).

IMPORTANTE: Este **RecordEnumeration** que se usará para mantener los contactos que ejercen una profesión determinada será un objeto **distinto al utilizado para navegar** en la pantalla principal.

Para **añadir un nuevo elemento al formulario**, se utilizará *Form.append(String)*, siendo el *String* pasado como parámetro una cadena con el nombre y apellidos del contacto separados por espacios.
Para **eliminar un elemento del formulario**, se utilizará *Form.delete(int)*. El parámetro *int*

[2] Un objeto TextBox debe ser tratado igual que un Form para atender a la selección de un Command por parte del usuario.
[3] Recuerda registrar también este formulario para que "escuche" a los Command.

indica el número de elemento a eliminar del formulario. Mediante un bucle *for* que llame a este método pasándole en las sucesivas llamadas valores entre *0* y *Form.size()-1*, se podrán eliminar los elementos del formulario.

4. ENVÍO DE MENSAJES SMS

Mediante el uso de *WMA (Wireless Messaging API)*, es posible enviar y recibir mensajes SMS en las aplicaciones J2ME. En las transparencias de teoría, aparece el código para enviar un SMS de texto. Añade un nuevo *Command* al MIDlet *ContactosMIDlet*, de forma que al seleccionarlo se llame a un método privado de ContactosMIDlet cuyo cuerpo sea el código que aparece en el ejemplo visto en teoría. A ese código habrá que hacerle las siguientes modificaciones:

- La dirección de destino del mensaje será "sms://+ 10000000".

- El texto enviado en el mensaje (cadena pasada como parámetro a *setPayloadText()*) será la concatenación de los distintos elementos de información de un contacto, separados por el carácter "\n" (salto de línea).

4.1. EMULACIÓN DE ENVÍO DE MENSAJES SMS

Para comprobar que la porción de código encargada del envío de un SMS funcione, será necesario configurar adecuadamente el emulador y utilizar las consolas WMA que proporciona NetBeans. Esta herramienta no hace sino utilizar la consola de Wireless Toolkit, la aplicación estándar utilizada para emular comunicaciones entre dispositivos MID. Habrá que configurar el número de teléfono que se va a asociar al emulador del móvil.

En el diálogo de preferencias (propiedades del proyecto→Platform→Manage Emulators→Tools & Extensions→Open Preferences), se elegirá la categoría WMA, donde podrá ser configurado el número de teléfono del primer emulador (*First Assigned Phone Number*, al que asignaremos el número +10000000) y el del siguiente emulador (*Phone Number of Next Emulator*, al que asignaremos el número +10000001).

El siguiente paso consiste en acceder a la opción *Open Utilities* para abrir una consola WMA. De acuerdo con los parámetros introducidos en las preferencias, se habrá abierto una consola WMA, que monitorizará el tráfico de red de un móvil con número de teléfono +10000000.

Ahora bastará con ejecutar la aplicación para comprobar cómo la consola WMA del teléfono +10000000 recibe el mensaje. Para observar detenidamente cómo se ejecuta la aplicación, se aconseja ejecutar la aplicación utilizando el modo de depuración. Previamente, se debería haber activado un punto de ruptura en la línea en que se declara el *String url* y se inicializa éste con la dirección del móvil destinatario del mensaje.

La ejecución de la suite *Contactos* provocará la creación de un emulador que ejecutará nuestra aplicación; dicho emulador tendrá asignado el número de teléfono +10000001. Ejecutando instrucción a instrucción se podrá comprobar que:

- Al ejecutar *Connection.open*, el emulador pide permiso para recibir mensajes de texto. Esto se debe a que cuando hay una conexión abierta, ésta puede ser utilizada tanto para leer mensajes como para escribirlos (ver figura a la izquierda, en la Ilustración 1).

- Al ejecutar *Connection.send*, el emulador pide permiso para enviar el mensaje de texto (ver figura a la derecha, en la Ilustración 1).

Ilustración 1: Peticiones de autorización para recepción y envío de mensajes

- Tras ejecutar *Connection.send*, la consola WMA muestra un mensaje advirtiendo de la llegada de un SMS y muestra el contenido del mismo.

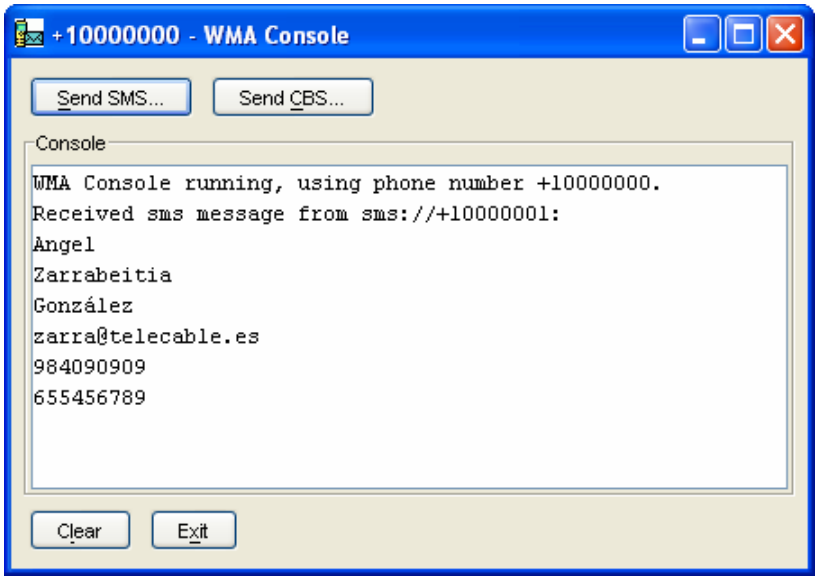


Ilustración 2: Consola WMA del teléfono destinatario del SMS

Se puede probar a no autorizar el envío y la recepción de mensajes para ver cómo funciona el mecanismo de gestión de errores mediante excepciones. Otra posible prueba interesante sería

enviar los datos en binario, para minimizar el tamaño de los SMS.
Es posible que la aplicación se bloquee al mostrar los diálogos que aparecen en la Ilustración 1. Estos diálogos chocan con el diseño de interfaz de usuario que hemos ido confeccionando. Para solucionar el problema, se puede hacer que el Wireless Toolkit no muestre los diálogos de confirmación de permisos para establecimiento de conexión y envío de mensaje. Para conseguir esto, se debe acceder a las preferencias del emulador y cambiar, en la pestaña *Security*, el dominio de seguridad del emulador a *trusted*.