Flujos de E/S y serialización de objetos. RMS

Índice

1 Flujos de datos de entrada/salida....2
2 Entrada, salida y salida de error estándar....3
3 Acceso a ficheros....4
4 Acceso a los recursos....5
5 Codificación de datos....6
6 Serialización de objetos en Java....6
7 Serialización manual en JavaME....8
8 Almacenamiento de registros RMS en JavaME....8
8.1 Almacenes de registros....9
8.2 Registros....12
8.3 Navegar en el almacén de registros....14
8.4 Notificación de cambios....17

Una de las posibilidades que ofrecen los flujos de entrada y salida es la de transportar objetos Java codificados en binario (serializados). En esta sesión se verá cómo hacer esto en JavaME, ya que no soporta la serialización automática. Otro mecanismo de persistencia muy usado en JavaME son los registros RMS.

1. Flujos de datos de entrada/salida

Existen varios objetos que hacen de flujos de datos, y que se distinguen por la finalidad del flujo de datos y por el tipo de datos que viajen a través de ellos. Según el tipo de datos que transporten podemos distinguir:

- Flujos de caracteres
- Flujos de *bytes*

Dentro de cada uno de estos grupos tenemos varios pares de objetos, de los cuales uno nos servirá para leer del flujo y el otro para escribir en él. Cada par de objetos será utilizado para comunicarse con distintos elementos (memoria, ficheros, red u otros programas). Estas clases, según sean de entrada o salida y según sean de caracteres o de *bytes* llevarán distintos sufijos, según se muestra en la siguiente tabla:

	Flujo de entrada / lector	Flujo de salida / escritor
Caractéres	`_Reader`	`_Writer`
Bytes	`_InputStream`	`_OutputStream`

Donde el prefijo se referirá a la fuente o sumidero de los datos que puede tomar valores como los que se muestran a continuación:

`File_`	Acceso a ficheros
`Piped_`	Comunicación entre programas mediante tuberías (*pipes*)
`String_`	Acceso a una cadena en memoria (solo caracteres)
`CharArray_`	Acceso a un *array* de caracteres en memoria (solo caracteres)
`ByteArray_`	Acceso a un *array* de *bytes* en memoria (solo *bytes*)

Además podemos distinguir los flujos de datos según su propósito, pudiendo ser:

- Canales de datos, simplemente para leer o escribir datos directamente en una fuente o sumidero externo.
- Flujos de procesamiento, que además de enviar o recibir datos realizan algún procesamiento con ellos. Tenemos por ejemplo flujos que realizan un filtrado de los datos que viajan a través de ellos (con prefijo `Filter`), conversores datos (con prefijo `Data`), *bufferes* de datos (con prefijo `Buffered`), preparados para la impresión de elementos (con prefijo `Print`), etc.

Un tipo de filtros de procesamiento a destacar son aquellos que nos permiten convertir un flujo de *bytes* a flujo de caracteres. Estos objetos son `InputStreamReader` y `OutputStreamWriter`. Como podemos ver en su sufijo, son flujos de caracteres, pero se construyen a partir de flujos de *bytes*, permitiendo de esta manera acceder a nuestro flujo de *bytes* como si fuese un flujo de caracteres.

Para cada uno de los tipos básicos de flujo que hemos visto existe una superclase, de la que heredaran todos sus subtipos, y que contienen una serie de métodos que serán comunes a todos ellos. Entre estos métodos encontramos los métodos básicos para leer o escribir caracteres o *bytes* en el flujo a bajo nivel. En la siguiente tabla se muestran los métodos más importantes de cada objeto:

InputStream	**read**(), **reset**(), **available**(), **close**()
OutputStream	**write**(int b), **flush**(), **close**()
Reader	**read**(), **reset**(), **close**()
Writer	**write**(int c), **flush**(), **close**()

A parte de estos métodos podemos encontrar variantes de los métodos de lectura y escritura, otros métodos, y además cada tipo específico de flujo contendrá sus propios métodos. Todas estas clases se encuentran en el paquete `java.io`. Para más detalles sobre ellas se puede consultar la especificación de la API de Java.

2. Entrada, salida y salida de error estándar

Al igual que en C, en Java también existen los conceptos de entrada, salida, y salida de error estándar. La entrada estándar normalmente se refiere a lo que el usuario escribe en la consola, aunque el sistema operativo puede hacer que se tome de otra fuente. De la misma forma la salida y la salida de error estándar lo que hacen normalmente es mostrar los mensajes y los errores del programa respectivamente en la consola, aunque el sistema operativo también podrá redirigirlas a otro destino.

En Java esta entrada, salida y salida de error estándar se tratan de la misma forma que cualquier otro flujo de datos, estando estos tres elementos encapsulados en tres objetos de flujo de datos que se encuentran como propiedades estáticas de la clase `System`:

	Tipo	**Objeto**
Entrada estándar	`InputStream`	`System.`**`in`**
Salida estándar	`PrintStream`	`System.`**`out`**
Salida de error estándar	`PrintStream`	`System.`**`err`**

Para la entrada estándar vemos que se utiliza un objeto `InputStream` básico, sin embargo para la salida se utilizan objetos `PrintWriter` que facilitan la impresión de texto

ofreciendo a parte del método común de bajo nivel `write` para escribir *bytes*, dos métodos más: `print` y `println`. Estas funciones nos permitirán escribir cualquier cadena, tipo básico, o bien cualquier objeto que defina el método `toString` que devuelva una representación del objeto en forma de cadena. La única diferencia entre los dos métodos es que el segundo añade automáticamente un salto de línea al final del texto impreso, mientras que en el primero deberemos especificar explícitamente este salto.

Para escribir texto en la consola normalmente utilizaremos:

```
System.out.println("Hola mundo");
```

En el caso de la impresión de errores por la salida de error de estándar, deberemos utilizar:

```
System.err.println("Error: Se ha producido un error");
```

Además la clase `System` nos permite sustituir estos flujos por defecto por otros flujos, cambiando de esta forma la entrada, salida y salida de error estándar.

Truco

Podemos ahorrar tiempo si en Eclipse en lugar de escribir `System.out.println` escribimos simplemente `sysout` y tras esto pulsamos *Ctrl + Espacio*.

3. Acceso a ficheros

Podremos acceder a ficheros bien por caracteres, o bien de forma binaria (por *bytes*). Las clases que utilizaremos en cada caso son:

	Lectura	Escritura
Caracteres	`FileReader`	`FileWriter`
Binarios	`FileInputStream`	`FileOutputStream`

Para crear un lector o escritor de ficheros deberemos proporcionar al constructor el fichero del que queremos leer o en el que queramos escribir. Podremos proporcionar esta información bien como una cadena de texto con el nombre del fichero, o bien construyendo un objeto `File` representando al fichero al que queremos acceder. Este objeto nos permitirá obtener información adicional sobre el fichero, a parte de permitirnos realizar operaciones sobre el sistema de ficheros.

A continuación vemos un ejemplo simple de la copia de un fichero carácter a carácter:

```
public void copia_fichero() {
  int c;
  try {
    FileReader in = new FileReader("fuente.txt");
    FileWriter out = new FileWriter("destino.txt");

    while( (c = in.read()) != -1) {
```

```
      out.write(c);
    }

    in.close();
    out.close();

  } catch(FileNotFoundException e1) {
    System.err.println("Error: No se encuentra el fichero");
  } catch(IOException e2) {
    System.err.println("Error leyendo/escribiendo fichero");
  }
}
```

En el ejemplo podemos ver que para el acceso a un fichero es necesario capturar dos excepciones, para el caso de que no exista el fichero al que queramos acceder y por si se produce un error en la E/S.

Para la escritura podemos utilizar el método anterior, aunque muchas veces nos resultará mucho más cómodo utilizar un objeto `PrintWriter` con el que podamos escribir directamente líneas de texto:

```
public void escribe_fichero() {
  FileWriter out = null;
  PrintWriter p_out = null;

  try {
    out = new FileWriter("result.txt");
    p_out = new PrintWriter(out);
    p_out.println(
      "Este texto será escrito en el fichero de salida");

  } catch(IOException e) {
    System.err.println("Error al escribir en el fichero");
  } finally {
    p_out.close();
  }
}
```

4. Acceso a los recursos

Hemos visto como leer y escribir ficheros, pero cuando ejecutamos una aplicación contenida en un fichero JAR, puede que necesitemos leer recursos contenidos dentro de este JAR.

Para acceder a estos recursos deberemos abrir un flujo de entrada que se encargue de leer su contenido. Para ello utilizaremos el método `getResourceAsStream` de la clase `Class`:

```
InputStream in = getClass().getResourceAsStream("/datos.txt");
```

De esta forma podremos utilizar el flujo de entrada obtenido para leer el contenido del fichero que hayamos indicado. Este fichero deberá estar contenido en el JAR de la aplicación.

Especificamos el carácter '`/`' delante del nombre del recurso para referenciarlo de forma relativa al directorio raíz del JAR. Si no lo especificásemos de esta forma se buscaría de

forma relativa al directorio correspondiente al paquete de la clase actual.

5. Codificación de datos

Si queremos guardar datos en un fichero binario, enviarlos a través de la red, o en general transferirlos mediante cualquier flujo de E/S, deberemos codificar estos datos en forma de *array* de *bytes*. Los flujos de procesamiento `DataInputStream` y `DataOutputStream` nos permitirán codificar y descodificar respectivamente los tipos de datos simples en forma de *array* de *bytes* para ser enviados a través de un flujo de datos.

Por ejemplo, podemos codificar datos en un *array* en memoria (`ByteArrayOutputStream`) de la siguiente forma:

```
String nombre = "Jose";
String edad = 25;
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);

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

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

Podremos descodificar este *array* de *bytes* realizando el procedimiento inverso, con un flujo que lea un *array* de *bytes* de memoria (`ByteArrayInputStream`):

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);
DataInputStream dis = new DataInputStream(bais);
String nombre = dis.readUTF();
int edad = dis.readInt();
```

Si en lugar de almacenar estos datos codificados en una *array* en memoria queremos guardarlos codificados en un fichero, haremos lo mismo simplemente sustituyendo el flujo canal de datos `ByteArrayOutputStream` por un `FileOutputStream`. De esta forma podremos utilizar cualquier canal de datos para enviar estos datos codificados a través de él.

6. Serialización de objetos en Java

Si queremos enviar un objeto a través de un flujo de datos, deberemos convertirlo en una serie de *bytes*. Esto es lo que se conoce como serialización de objetos, que nos permitirá leer y escribir objetos directamente.

Para leer o escribir objetos podemos utilizar los objetos `ObjectInputStream` y `ObjectOutputStream` que incorporan los métodos `readObject` y `writeObject` respectivamente. Los objetos que escribamos en dicho flujo deben tener la capacidad de ser *serializables*.

Serán *serializables* aquellos objetos que implementan la interfaz `Serializable`. Cuando queramos hacer que una clase definida por nosotros sea *serializable* deberemos implementar dicho interfaz, que no define ninguna función, sólo se utiliza para identificar las clases que son *serializables*. Para que nuestra clase pueda ser *serializable*, todas sus propiedades deberán ser de tipos de datos básicos o bien objetos que también sean *serializables*.

Un uso común de la serialización se realiza en los *Transfer Objetcs*. Este tipo de objetos deben ser serializables para así poderse intercambiar entre todas las capas de la aplicación, aunque se encuentren en máquinas diferentes.

Por ejemplo, si tenemos un objeto como el siguiente:

```
public class Punto2D implements Serializable {
        private int x;
        private int y;

        public int getX() {
                return x;
        }
        public void setX(int x) {
                this.x = x;
        }
        public int getY() {
                return y;
        }
        public void setY(int y) {
                this.y = y;
        }
}
```

Podríamos enviarlo a través de un flujo, independientemente de su destino, de la siguiente forma:

```
Punto2D p = crearPunto();
FileOutputStream fos = new FileOutputStream(FICHERO_DATOS);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(p);
oos.close();
```

En este caso hemos utilizado como canal de datos un flujo con destino a un fichero, pero se podría haber utilizado cualquier otro tipo de canal (por ejemplo para enviar un objeto Java desde un servidor web hasta una máquina cliente). En aplicaciones distribuidas los objetos *serializables* nos permitirán mover estructuras de datos entre diferentes máquinas sin que el desarrollador tenga que preocuparse de la codificación y transmisión de los datos.

Muchas clases de la API de Java son *serializables*, como por ejemplo las colecciones. Si tenemos una serie de elementos en una lista, podríamos serializar la lista completa, y de esa forma guardar todos nuestros objetos, con una única llamada a `writeObject`.

Cuando una clase implemente la interfaz `Serializable` veremos que Eclipse nos da un *warning* si no añadimos un campo `serialVersionUID`. Este es un código numérico que se utiliza para asegurarnos de que al recuperar un objeto serializado éste se asocie a la

misma clase con la que se creó. Así evitamos el problema que puede surgir al tener dos clases que puedan tener el mismo nombre, pero que no sean iguales (podría darse el caso que una de ellas esté en una máquina cliente, y la otra en el servidor). Si no tuviésemos ningún código para identificarlas, se podría intentar recuperar un objeto en una clase incorrecta.

Eclipse nos ofrece dos formas de generar este código pulsando sobre el icono del *warning*: con un valor por defecto, o con un valor generado automáticamente. Será recomendable utiliar esta segunda forma, que nos asegura que dos clases distintas tendrán códigos distintos.

7. Serialización manual en JavaME

CLDC no soporta la serialización de objetos, por tanto tendremos que serializar los objetos manualmente, definiendo los métodos `serialize()` y `deserialize()`.

En el ejemplo de un objeto que representa un punto en el plano, serializaríamos el objeto y lo deserializaríamos con:

```
public class Punto2D {
  int x;
  int y;
  String etiqueta;

  public void serialize(OutputStream out) throws IOException {
    DataOutputStream dos = new DataOutputStream( out );
    dos.writeInt(x);
    dos.writeInt(y);
    dos.writeUTF(etiqueta);
    dos.flush();
  }

  public static Punto2D deserialize(InputStream in)
                                      throws IOException {
    DataInputStream dis = new DataInputStream( in );
    Punto2D p = new Punto2D();
    p.x = dis.readInt();
    p.y = dis.readInt();
    p.etiqueta = dis.readUTF();
    return p;
  }

}
```

8. Almacenamiento de registros RMS en JavaME

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

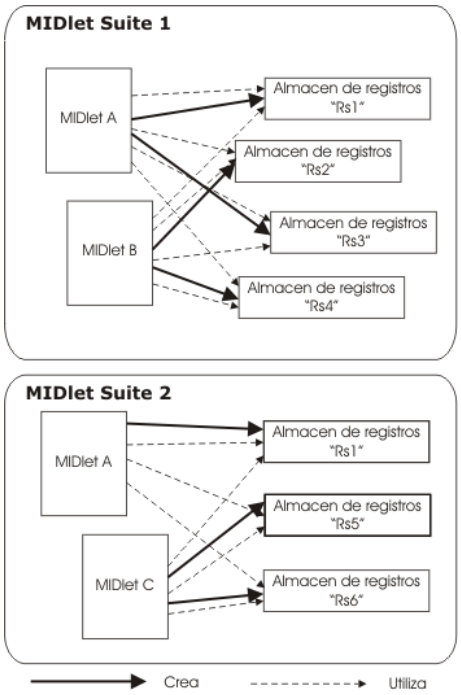
ficheros, por cuestiones de seguridad MIDP no nos dejará acceder directamente a él. Es posible que en alguna implementación podamos acceder a ficheros en el dispositivo, pero esto no es requerido por la especificación, por lo que si queremos que nuestra aplicación sea portable no deberemos confiar en 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`.

8.1. 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.



Acceso a los almacenes de registros

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.

8.1.1. 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 se 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();
```

8.1.2. 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.

8.1.3. Eliminar un almacén de registros

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

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

8.1.4. 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();
```

8.2. 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.

8.2.1. 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.

8.2.2. 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 descodificarlo 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();
```

8.2.3. Borrar Registros

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

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

8.2.4. 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);
```

8.3. 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 al 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 recorremos 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.

8.3.1. 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.

8.3.2. 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.

8.4. 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.