Opciones avanzadas en el acceso a una BD

Índice

1. Fuentes de datos (DataSources)

Hasta el momento, cuando queríamos conectarnos a una base de datos, proporcionábamos su dirección URL. Si cambiábamos la dirección de la BD o la propia BD debíamos recompilar la aplicación. Si tenemos muchas aplicaciones en nuestro sistema la actualización puede ser costosa, ya que deberiamos cambiar todas las direcciones. Las fuentes de datos en JDBC permiten una utilización más flexible de las conexiones. Un objeto **DataSource** se gestiona de manera independiente de nuestra aplicación. Contendrá los datos necesarios para la conexión al sistema y proporcionará los métodos apropiados para consultar a la BD. La clase **DataSource**es una interfaz, por lo que no podemos instanciarla directamente. Debemos disponer de una clase que implemente dicha interfaz, usualmente desarrollada por el fabricante de la BD. Esto suele venir implementado en los servidores de aplicaciones.

2. Uso de las fuentes de datos

Para hacer uso de las fuentes de datos debemos seguir dos fases. En la primera, haciendo uso de JNDI debemos asociar un objeto **DataSource** con una base de datos. El objeto **DataSource** contendrá los datos necesarios para la conexión a la BD: dirección URL, datos de usuarios, etc. Este objeto lo almacenaremos en un servicio de directorios con JNDI. JNDI nos permite asociar un nombre lógico ("MiDS") a cualquier objeto. En principio podemos tener tantos objetos **DataSource** como queramos en nuestro sistema. Esta primera fase la suele llevar a cabo el servidor de aplicaciones.

La segunda fase consiste en la conexión de un cliente a la base de datos. Cuando un cliente necesite conectarse a la base de datos, utilizará JNDI para buscar el objeto **DataSource** asociado y éste a su vez devolverá una conexión, tal como hacía la clase **DriverManager** anteriormente. En el código que se muestra a continuación se puede observar el código que implementará el cliente cuando quiera obtener una conexión.

```
// Se crea un contexto inicial (JNDI)
Context contexto = new InitialContex();
// JNDI nos proporciona el objeto DataSource
DataSource ds = (DataSource) context.lookup("jdbc/MySQL");
// Obtenemos la conexión del objeto DataSource
conexion = ds.getConnection();
```

Con esta abstracción en la conexión hemos conseguido que el cliente sólo conozca el nombre "lógico" de la BD, no su dirección URL. El cliente tampoco conoce los detalles del driver utilizado. Todo esto lo encapsula el objeto **DataSource**. Si queremos realizar algún cambio en nuestra BD (cambio de dominio, actualización, etc.) sólo tendremos que cambiar el objeto **DataSource**, no la aplicación cliente. A partir de este punto ya podemos hacer uso del API

de JDBC estándar para realizar consultas a la BD. Como ya hemos comentado las fuentes de datos las suele gestionar el servidor de aplicaciones.

3. Reserva de conexiones (ConnectionPool)

Cuando accedemos a una base de datos, el proceso de conexión es el más costoso temporalmente. En una aplicación de servidor, el número de conexiones puede llegar a ser enorme. Es por ello que se deba buscar un método para optimizar el coste temporal. Una técnica muy utilizada es la reutilización de los recursos costosos. En este caso el recurso costoso es la conexión, por lo que resulta lógico que se intente reducir el coste de conexión manteniendo un número predeterminado de conexiones creadas e ir sirviendo a las clientes que las soliciten. Cuando una aplicación termina con una conexión la libera y queda disponible para otra aplicación. También en este caso, al igual que en el caso de **DataSource**, es el servidor de aplicaciones el que se encarga de administrar y definir cuántas reservas tendremos en funcionamiento.

A pesar de ser una característica muy útil, la reserva de conexiones está aconsejada en el caso de que se cumpla lo siguiente:

- Las aplicaciones acceden a la base de datos mediante un conjunto muy reducido de cuentas de usuario. Normalmente en la conexión a una base de datos especificamos el usuario con el que accedemos. La reserva de conexiones necesita que todas las conexiones sean para el mismo usuario, por lo que para aplicaciones donde se utilicen diversas cuentas de usuario para acceder y el número de accesos por cada cuenta es reducido es desaconsejable el uso de la reserva de conexiones.
- El acceso a la base de datos se realiza durante una única sesión. Veamos el siguiente ejemplo: en una transacción estamos constantemente accediendo a la base de datos durante mucho tiempo, por ejemplo, un carrito de la compra donde por cada elemento hacemos una actualización de una tabla de la BD. En este caso está desaconsejado el uso de la reserva de conexiones, es preferible el uso de una conexión dedicada.

Una aplicación que utilice el enfoque de reserva de conexiones debe seguir el siguiente orden:

- Obtener una referencia a la reserva o a un objeto que gestione la reserva.
- Consigue una conexión de una reserva (**getConnection**)
- Utiliza la conexión. Aquí se realizarán todas las consultas y actualizaciones en la base de datos. Un aspecto muy importante aquí son las transacciones. Si una aplicación empieza una transacción y no la termina o la deshace, puede que se produzca una pérdida de consistencia de datos.
- Devuelve la conexión a la reserva. Es muy importante que la aplicación que solicita la reserva no cierre la conexión.

4. Transacciones

Muchas veces, cuando tengamos que realizar una serie de acciones, queremos que todas se hayan realizado correctamente, o bien que no se realice ninguna de ellas, pero no que se realicen algunas y otras no.

Podemos ver esto mediante un ejemplo, en el que se va a hacer una reserva de vuelos para ir desde Alicante a Osaka. Para hacer esto tendremos que hacer trasbordo en dos aeropuertos, por lo que tenemos que reservar un vuelo Alicante-Madrid, un vuelo Madrid-Amsterdam y un vuelo Amsterdam-Osaka. Si cualquiera de estos tres vuelos estuviese lleno y no pudiésemos reservar, no queremos reservar ninguno de los otros dos porque no nos serviría de nada. Por lo tanto, sólo nos interesa que la reserva se lleve a cabo si podemos reservar los tres vuelos.

Una transacción es un conjunto de sentencias que deben ser ejecutadas como una unidad, de forma que si una de ellas no puede realizarse, no se llevará a cabo ninguna. Dicho de otra manera, las transacciones hacen que la BD pase de un estado consistente al siguiente.

Pero para hacer esto encontramos un problema. Pensemos en nuestro ejemplo de la reserva de vuelos, en la que necesitaremos realizar las siguientes inserciones (reservas):

```
try {
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, destino)
        VALUES('Paquito', 'Alicante', 'Madrid')");
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, destino)
        VALUES('Paquito', 'Madrid', 'Amsterdam')");
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, destino)
        VALUES('Paquito', 'Amsterdam', 'Osaka')");
}
catch(SQLException e) {
    // ¿Dónde ha fallado? ¿Qué hacemos ahora?
}
```

En este caso, vemos que si falla la reserva de uno de los tres vuelos obtendremos una excepción, pero en ese caso, ¿cómo podremos saber dónde se ha producido el fallo y hasta qué acción debemos deshacer? Con la excepción lo único que sabemos es que algo ha fallado, pero no sabremos dónde ha sido, por lo que de esta forma no podremos saber hasta qué acción debemos deshacer.

Para hacer esto de una forma limpia asegurando la consistencia de los datos, utilizaremos las operaciones de *commit* y *rollback*.

Cuando realicemos cambios en la base de datos, estos cambios se harán efectivos en ella de forma persistente cuando realicemos la operación *commit*. En el modo de operación que

hemos visto hasta ahora, por defecto tenemos activado el modo *auto-commit*, de forma que siempre que ejecutamos alguna sentencia se realiza *commit* automáticamente. Sin embargo, en el caso de las transacciones con múltiples sentencias, no nos interesará hacer estos cambios persistentes hasta haber comprobado que todos los cambios se pueden hacer de forma correcta. Para ello desactivaremos este modo con:

```
con.setAutoCommit(false);
```

Al desactivar este modo, una vez hayamos hecho las modificaciones de forma correcta, deberemos hacerlas persistentes mediante la operación *commit* llamando de forma explícita a:

```
con.commit();
```

Si por el contrario hemos obtenido algún error, no queremos que esas modificaciones se lleven a cabo finalmente en la BD, por lo que podremos deshacerlas llamando a:

```
con.rollback();
```

Por lo tanto, la operación *rollback* deshará todos los cambios que hayamos realizado para los que todavía no hubiésemos hecho *commit* para hacerlos persistentes, permitiéndonos de esta forma implementar estas transacciones de forma atómica.

Nuestro ejemplo de la reserva de vuelos debería hacerse de la siguiente forma:

```
try {
    con.setAutoCommit(false);
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, destino)
        VALUES('Paquito', 'Alicante', 'Madrid')");
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, destino)
        VALUES('Paquito', 'Madrid', 'Amsterdam')");
    stmt.executeUpdate("INSERT INTO RESERVAS(pasajero, origen, destino)
        VALUES('Paquito', 'Amsterdam', 'Osaka')");
    // Hemos podido reservar los tres vuelos correctamente
    con.commit();
}
catch(SQLException e) {
    // Alguno de los tres ha fallado, deshacemos los cambios
    try {
        con.rollback();
    }
    catch(SQLException e) {};
}
```

Una característica relacionada con las transacciones es la concurrencia en el acceso a la BD. Dicho de otra forma, qué sucede cuando varios usuarios se encuentran accediendo a la vez a los mismos datos y pretenden modificarlos. Un ejemplo sencillo: tenemos una tienda y dos usuarios están accediendo al mismo disco, del cual sólo queda una unidad. El primero de los usuarios consulta el disponible, comprueba que existe una unidad y lo introduce en su cesta

de la compra. El otro usuario en ese preciso momento también está consultando el disponible, también le aparece una unidad y también intenta introducirlo en su cesta de la compra. Al segundo usuario el sistema no debería dejarle actualizar los datos que está manejando el primero.

La concurrencia es manejada por los distintos SGBD de manera distinta. PostGres permite nivel de lectura confirmada. En este nivel, si una transacción lee una fila de la BD y otra transacción cambia la misma fila, en el momento que la primera transacción intente leer de nuevo se actualiza el valor. Si las dos intentan modificar la misma fila la segunda se bloqueará hasta que la primera finalice la transacción. Esto último hay que tenerlo en cuenta para evitar bloqueos. Para saber el nivel concurrencia permitido por el SGBD podemos utilizar el siguiente método de la clase **Connection:**

```
int con.getTransactionIsolation();
```

Los valores más comunes que puede devolver este método son: Connection.**TRANSACTION_NONE** (no soporta transacciones), Connection.**TRANSACTION_READ_UNCOMMITTED** (soporta transacciones en su nivel más bajo), Connection.**TRANSACTION_READ_COMMITTED** (el nivel de PostGres antes comentado). MySQL incorpora transacciones en sus últimas versiones. Sin embargo, su funcionamiento es diferente a PostGres. Si una aplicación actualiza una fila y dentro de su transacción otra aplicación intenta modificarla, MySQL permite la actualización y no se produce ningún problema.

Un posible problema en las transacciones es el interbloqueo. Un interbloqueo se produce en la siguiente situación: una aplicación tiene que modificar dos registros. Otra aplicación modifica los mismos, pero en orden inverso. Se empiezan a ejecutar las dos aplicaciones a la vez y al haber modificado un registro no dejan que la otra lo modifique. Sin embargo, ninguna de las dos terminan porque están esperando que se desbloquee su registro. MySQL no detecta esta situación. Es más, permite cambiar los datos que una transacción sin finalizar haya realizado. PostGres, al bloquear los registros sí que provoca un interbloqueo que detecta y lanza una excepción.

5. Puntos de almacenamiento (savepoint)

Los puntos de almacenamiento permiten definir puntos de control en el flujo de una transacción. Comentaremos aquí la definición y cómo podemos utilizarlos. Sin embargo, ni MySQL ni PostGres los tienen implementados. Los puntos de almacenamiento los genera un objeto **Connection**. Permiten marcar uno o más lugares de una transacción para, si algo falla, poder deshacer las acciones realizadas a partir de uno de esos puntos marcados. Los métodos a utilizar son los siguientes:

```
Savepoint con.setSavepoint("Punto de control");
Savepoint con.setSavepoint();
```

Este método permite marcar un punto de almacenamiento. También disponemos de una forma de deshacer los cambios producidos a partir de un punto de almacenamiento:

```
con.rollback (Savepoint sp);
```

En el caso que de una excepción se produzca, podemos consultar el punto de almacenamiento para determinar si las acciones anteriores a la definición del punto de control se han realizado con éxito. El siguiente código muestra un ejemplo de utilización.

```
Savepoint sp = null;
boolean realizar_commit = false;
try {
    insertarTabla1 (datos);
    realizar_commit = true;
    sp = con.setSavepoint();
    insertarTabla2 (datos2);
}
catch {
    // Si sp es null es que falló la primera inserción
    if (sp==null) {
        con.rollback();
    }
    else {
        con.rollback(sp);
    }
}
finally {
    if (realizar_commit) {
        con.commit();
    }
}
```

Vamos a analizar con detalle el código. Se declaran dos variables. La primera es el punto de almacenamiento. La segunda es una variable de control que nos va a permitir saber si se realizó la primera acción en la base de datos. El código dentro de la sentencia **try** llama a dos métodos, **insertarTabla1** e **insertarTabla2**. Estos métodos se encargan de actualizar la base de datos, con la particularidad de que la primera inserción es más importante que la segunda. Además, si la primera no ha tenido éxito se deshace toda la transacción, pero si ha tenido éxito y la segunda no, podemos deshacer sólo la segunda. En este caso es cuando tiene utilidad los puntos de almacenamiento. Hemos colocado un punto de almacenamiento después de la llamada a la primera inserción. Si se ha producido una excepción, sabemos que si el punto de almacenamiento creado es nulo entonces la excepción se produjo en la primera inserción. Si no es nulo, la primera inserción finalizó con éxito y la segunda provocó la excepción. Por ello llamamos al método **roolback** con el punto de almacenamiento como parámetro. Esta acción provoca que se deshagan las acciones realizadas en la segunda

inserción. Por último, en el bloque **finally**, realizamos **commit** si se ha realizado la primera inserción con éxito.

6. RowSet

Un **RowSet** es un componente que se ajusta a JavaBean y que encapsula el acceso a bases de datos, incluído el resultado. Permite olvidarnos de los objetos *Connection*, *ResultSet*, *Statement*, etc. Todas las funcionalidades de estas clases las implementa RowSet. Rowset es un interfaz, por lo que debe ser implementada antes de instanciarse.

La clase mantiene los métodos utilizados por el *ResultSet* para acceder a los datos y a los registros. Disponemos de *next, previous, getXXX, updateXXX* tal como los usábamos en *ResultSet*. Los métodos descritos a continuación sirven para configurar el acceso a una base de datos:

```
//Utilizamos una de las implementaciones como ejemplo
JdbcRowSetImpl jrs = new JdbcRowSetImpl ();
// Asignamos la URL de nuestra base de datos
jrs.setUrl ("jdbc:mysql://localhost/bd");
// Asignamos el usuario y la contraseña
jrs.setUsername ("miguel");
jrs.setPassword ("cazorla");
```

A partir de este momento ya podemos utilizar el RowSet y realizar consultas a la BD. Debemos tener en cuenta que debemos cargar el driver de la base de datos tal como lo hacíamos anteriormente:

```
Class.forName ("com.mysql.jdbc.Driver");
```

Para ejecutar un comando, primero asignamos el comando y después lo ejecutamos.

```
crs.setCommand ("Select * from vuelo");
crs.execute ();
```

Ya podemos movernos tal como lo hacíamos en el *ResultSet*. También podemos utilizar los métodos *updateXXX* para actualizar los datos del *RowSet*. Una vez realizada la llamada a **updateRow** debemos realizar una llamada al método **acceptChanges**. Este método se encargará de actualizar los datos en la fuente de datos. También permite la preparación de sentencias, tal como se hacía con las sentencias preparadas. Sin embargo, aquí no es necesario el uso de objetos adicionales.

```
crs.setCommand ("select numero from vuelo where aero_inic=?");
crs.setString (1, "ALC");
```

La especificación define cinco clases de RowSets: JdbcRowSet, CachedRowSet, WebRowSet, JoinRowSet y FilteredRowSet. Todas ellas son interfaces que hay que

implementar. Sun ha desarrollado una implementación de todas ellas. Para instanciar las clases debemos usar el siguiente código de ejemplo:

```
JdbcRowSet jrs = new JdbcRowSetImpl();
```

6.1. JdbcRowSet

Esta implementación es simplemente una capa por encima de *ResultSet* para que parezca y pueda ser utilizado como un JavaBean. Además, nos va a permitir visitar los registros de la consulta hacia delante y detrás y actualizar la consulta, aunque la conexión y/o el driver no lo permitan. Esta es la única implementación que mantiene la conexión con la BD. El resto realizan la conexión, consultan la BD, devuelven los datos y cierran la conexión. Podemos crear un objeto JdbcRowSet a partir de un objeto ResultSet. Por ejemplo:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery ("select numero from vuelo where
aero_inic=ALC2");
JdbcRowSet jrs = new JdbcRowSetImpl (rs);
```

JdbcRowSet es un *wrapper* (envolvente) para el ResultSet. Contiene exactamente los mismos datos que el ResultSet. Como vemos, como no hemos indicado ninguna opción para que sea *scrollable* ni *updatable* el ResultSet sólo se puede consultar hacia delante. Sin embargo, el JdbcRowSet sí que es *scrollable* y *updatable.*

6.2. CachedRowSet

Esta implementación permite operar con los datos sin estar conectado a su fuente de datos. También permite el *scrolling* y la actualización, a pesar de que el driver no los soporte. Esta es una de las ventajas fundamentales de esta implementación. Otra de sus funcionalidades es la posibilidad de recibir datos de fuentes distintas a las bases de datos (hojas de cálculo, ficheros de texto tabulados, etc.). Cuando invocamos al método **execute** se realiza una conexión a la base de datos, se recibe la información resultado de realizar la consulta y se cierra la conexión. Si realizamos una actualización de los datos (**acceptChanges**) el objeto realiza una nueva conexión y actualiza los datos. Otra característica muy interesante es el manejo de eventos (*Listeners*). Son detallados más abajo.

6.3. JoinRowSet

Esta implementación permite crear uniones de SQL (SQL join). Una unión es una consulta donde usamos más de una tabla. Por ejemplo, una típica sentencia SQL:

```
select * from vuelo,disponibilidad
       where vuelo.numero=disponibilidad.numVuelo
```

Si ya disponemos de, por ejemplo, un CachedRowSet con consulta a esas dos tablas, usuario y trayecto, podemos obtener el resultado de realizar la anterior consulta a la bD sin necesidad de conectarnos a ella. Para ello, simplemente tenemos que crear un objeto JoinRowSet y añadirle los dos objetos RowSet.

```
CachedRowSet crsu = new CachedRowSetImpl();
crsu.setUrl("jdbc:mysql://127.0.0.1/viajes");
crsu.setUsername("admin");
crsu.setPassword("admin");
crsu.setCommand("Select * from usuario");
crsu.execute();
CachedRowSet crst = new CachedRowSetImpl();
crst.setUrl("jdbc:mysql://127.0.0.1/viajes");
crst.setUsername("admin");
crst.setPassword("admin");
crst.setCommand("Select * from trayecto");
crst.execute();
JoinRowSet jrs = new JoinRowSetImpl();
jrs.addRowSet(crst,2);
jrs.addRowSet(crsu,"DNI");
```

Como podemos observar, hemos usado el método *addRowSet* para añadir los RowSets. El segundo parámetro de este método indica el campo que se usa para la correspondencia. La tabla usuario y trayecto tienen un campo de relación, el DNI. El campo se puede indicar mediante el nombre del campo o su número de orden. Se ha detectado problemas en la implementación de Sun, no funcionando correctamente.

6.4. FilteredRowSet

De forma parecida al anterior, un FilteredRowSet permite filtrar los datos de un RowSet, es decir, reducir el número de registros mediante algún criterio. Esta clase se gestiona de forma similar que las anteriores:

```
FilteredRowSet frs = new FilteredRowSetImpl();
frs.setUrl("jdbc:mysql://127.0.0.1/viajes");
frs.setUsername("admin");
frs.setPassword("admin");
frs.setCommand("Select * from usuario");
frs.execute();
```

Adicionalmente debemos crear un filtro para realizar el filtrado de los datos. Un filtro debe implementar la interfaz *Predicate.* Este filtro es el encargado de eliminar los datos que no cumplan un cierto criterio. Para crear un filtro podemos usar el siguiente código. Notar que hace falta implementar los tres métodos *evaluate*:

```
public class FiltroDisp implements Predicate{
    private double val;
    public FiltroDisp (double vali) {
        val=vali;
    }
    public boolean evaluate (Object obj, String str) {
        return true;
    }
    public boolean evaluate (Object obj, int i) {
        return true;
    }
    public boolean evaluate (RowSet rs) {
        try {
            return (rs.getDouble("Precio") <= val);
        }
        catch (SQLException e) {
            System.out.println(e); return false;
        }
    }
}
```

Este filtro comprueba si el precio de un vuelo es inferior a una cierta cantidad. Esa cantidad se la pasamos al filtro cuando lo creamos. Los métodos *evaluate* son los llamados por FilteredRowSet para comprobar si los registros cumplen el criterio. Si asignamos el filtro al objeto FilteredRowSet antes creado, los registros que no cumplan el criterio del filtro se ocultarán. No se borran, se ocultan. Si a continuación asignamos un filtro diferente, se vuelven a ocultar aquellos que no cumplan el nuevo criterio, mostrando los que sí que lo cumplan.

```
FiltroDisp fd = new FiltroDisp (200.00);
frs.setFilter(fd);
```

En la implementación de Sun, cuando visitas los elementos después de aplicar el filtro, en el último registro da una excepción.

6.5. WebRowSet

El objetivo principal de esta implementación es la lectura y escritura de datos en formato XML. En una aplicación cliente-servidor, la parte del servidor es la que se encarga de consultar la base de datos y crear un objeto **WebRowSet** para a continuación crear un fichero XML que es el que se le envía al cliente. El cliente recibe los datos en formato XML, pero los puede manejar como si fuera un resultado de una consulta. Cuando actualiza algún dato y llama al método *acceptChanges* la información se envía de vuelta al servidor en formato XML y el servidor es el que se encarga de escribir en la base de datos. Esta implementación tiene sentido cuando trabajamos detrás de cortafuegos y/o proxys, debido a que las conexiones directas a base de datos no son siempre posibles. Esta clase dispone de dos métodos para leer y escribir datos en XML, ambos asociados a objetos que nos permiten leer

y escribir.

```
java.io.FileWriter FW = new java.io.FileWriter("ejemplo.xml");
crs.writeXML (FW);
```

Cuando llamamos al método **writeXML** el contenido del RowSet se escribe en el fichero *ejemplo.xml* en formato XML. De la misma forma disponemos del método **readXML** asociado a un objeto de la clase *java.io.FileReader*. En la actual implementación se ha detectado un problema de lectura.

6.6. Manejo de eventos

Otro característica importante de los *RowSet* son los eventos. Podemos definir una clase que actúe de escuchante, para que cuando se produzca un evento se realicen una serie de acciones definidas en el escuchante. El API JDBC especifica la interfaz *javax.sql.RowSetListener*. La interfaz permite especificar tres métodos, asociados a tres eventos distintos:

```
public void cursorMoved (RowSetEvent evento)
public void rowChanged (RowSetEvent evento)
public void rowSetChanged (RowSetEvent evento)
```

El primer método se invocará cuando el cursor se mueva. El cursor es un elemento que indica al objeto *RowSet* en qué fila se encuentra dentro del conjunto de resultado. Dentro de este método podemos realizar las acciones que queramos: notificar el evento a otro objeto, realizar alguna comprobación, cambiar la base de datos, etc. El segundo método se invocará cuando se cambie una fila y el tercero cuando cambie el *RowSet*. A continuación se muestra un ejemplo de definición de uno de los métodos:

```
import javax.sql.*;
import java.io.*;
public class Escuchante implements RowSetListener {
    public void cursorMoved (RowSetEvent evento) {
        System.out.println("Se ha movido el cursor");
    }
```

Para asociar el escuchante a una clase RowSet debemos hacer uso de la anterior clase:

```
Escuchante escucha = new Escuchante();
//crs es un objeto de tipo RowSet
crs.addRowSetListener(escucha);
```

Podemos añadir tantos escuchantes como queramos.