



Desarrollo de Aplicaciones Informáticas

CICLO FORMATIVO DE GRADO SUPERIOR

FORMACIÓN PROFESIONAL A DISTANCIA

Unidad **10**

Programación WEB - Acceso a datos

MÓDULO

Desarrollo de Aplicaciones en Entornos de Cuarta Generación y con Herramientas CASE



FORMACIÓN PROFESIONAL

Principado de Asturias

Título del Ciclo: DESARROLLO DE APLICACIONES INFORMATICAS

Título del Módulo: DESARROLLO DE APLICACIONES EN ENTORNOS DE CUARTA GENERACIÓN Y CON HERRAMIENTAS CASE

Unidad 10: Programación WEB - Acceso a datos

Dirección: Dirección General de Políticas Educativas, Ordenación Académica y Formación Profesional
Servicio de Formación Profesional Inicial y Aprendizaje Permanente

Dirección de la obra:

Alfonso Careaga Herrera
Antonio Reguera García
Arturo García Fernández
Ascensión Solís Fernández
Juan Carlos Quirós Quirós
Luís M^a Palacio Junquera
Yolanda Álvarez Granda

Coordinador de los contenidos:

Juan Manuel Fernández Gutiérrez

Autores:

Juan Manuel Fernández Gutiérrez

Colección:

Materiales didácticos de aula

Serie:

Formación profesional específica

Edita:

Consejería de Educación y Ciencia

ISBN:

978-84-692-3443-2

Deposito Legal:

AS-3564-2009

Copyright:

2009. Consejería de Educación y Ciencia

Esta publicación tiene fines exclusivamente educativos. Queda prohibida la venta, así como la reproducción total o parcial de sus contenidos sin autorización expresa de los autores y del Copyright

Introducción

En esta unidad se avanza en la programación con JSP introduciendo el acceso a datos. El sistema gestor es el de Oracle, no obstante todo lo que se expone en esta unidad funciona con cualquier base de datos excepto el método de conexión.

Otro aspecto que se toca y complementa el acceso a datos es la gestión de procedimientos almacenados, que en el caso de Oracle se utilizan con mucha frecuencia y es donde podemos hacer uso del lenguaje PL/SQL visto en unidades anteriores.

Objetivos

- Conocer los métodos de conexión y acceso a una base de datos
- Crear las clases para la manipulación de los datos
- Conocer distintas formas de acceso y recuperación de la información
- Gestión de transacciones
- Trabajar con procedimientos almacenados

Contenidos Generales

1. ACCESO A LAS BASES DE DATOS.....	3
2. CONEXIÓN CON LA BASE DE DATOS.....	3
3. CREAR SENTENCIAS	4
4. RECUPERAR VALORES. (EJECUTAR CONSULTAS).....	4
5. SENTENCIAS PREPARADAS	6
5.1. Suministrar valores para los parámetros de un <i>PreparedStatement</i>	7
6. TRANSACCIONES.....	9
6.1. <i>Deshacer una transacción</i>	10
7. INSERCIONES Y CONSULTAS A TRAVÉS DE FORMULARIOS.....	10
7.1. <i>Inserciones</i>	11
7.2. <i>Consultas</i>	15
7.3. <i>Consultar un departamento</i>	17
8. ACTUALIZACIONES.....	19
9. ELIMINAR DEPARTAMENTOS	22
10. PROCEDIMIENTOS ALMACENADOS. <i>CALLABLESTATEMENT</i>	26
10.1. <i>Eliminar tipos de objetos del esquema de usuario</i>	31
10.2. <i>CallableStatement. Procedimiento que devuelve valores</i>	33
11. MÉTODOS DE CONEXIÓN Y ACCESO.....	37



1. Acceso a las Bases de Datos

Desde que aparecen las bases de datos, se puede decir que la mayoría de las aplicaciones utilizan esta forma de almacenamiento, independientemente de que las aplicaciones sean web o cliente/servidor.

Ciñéndonos a las aplicaciones web, para poder utilizar bases de datos con Java emplearemos la API JDBC creada por Sun para el acceso a la bases de datos, lo que permite establecer una conexión con el motor de la base de datos independientemente de la que se maneje. Para más información sobre la API se puede acceder a java.sun.com/jdbc/.

Al utilizar un driver JDBC las funciones son las mismas independientemente de la base que se utilice. Una vez establecida la conexión se puede utilizar la base de datos independientemente de que sea mySQL, SQLserver, Oracle etc. Si se cambia la base de datos, en la aplicación no hay que modificar nada, salvo la conexión.

NOTA

Para que funcione *tomcat* con *oracle* hay que copiar la librería de oracle

C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\ojdbc14.jar

en el *bin* del *tomcat* o en el *webcontent* de la aplicación

2. Conexión con la base de datos

Para conectar con la base de datos lo primero que tenemos que hacer es establecer una conexión con el controlador de base de datos que queremos utilizar. Esto implica cargar el driver y hacer la conexión.

Cargar el driver que queremos utilizar sólo implica una línea de código. Si, por ejemplo, queremos utilizar el de oracle.

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Una vez cargado el driver, es posible hacer una conexión con un controlador de base de datos.

En nuestro caso conectamos con la *oracle10g* cuyo nombre se XE, el usuario y contraseña son *rasty/rasty*. 127.0.0.1 es nuestro equipo y el puerto 1521

```
Connection conecta =  
DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe",  
"rasty", "rasty");
```

Como estamos utilizando un puente desarrollado por oracle se pone *jdbc:oracle*. Si se utiliza el puente ODBC la URL empezaría por *jdbc:ODBC*

Si uno de los drivers que hemos cargado reconoce la URL suministrada por el método **DriverManager.getConnection**, dicho driver establecerá una conexión con el controlador de base de datos especificado en la URL del JDBC. La clase **DriverManager**, como su nombre indica, maneja todos los detalles del establecimiento de la conexión.

La conexión devuelta por el método **DriverManager.getConnection** es una conexión abierta que se puede utilizar para crear sentencias JDBC que pasen nuestras sentencias SQL al controlador de la base de datos. En el ejemplo anterior, **conecta** es una conexión abierta.

3. Crear sentencias

EL objeto **Statement** es el que envía nuestras sentencias SQL al controlador de la base de datos. Simplemente creamos un objeto **Statement** y lo ejecutamos, suministrando el método SQL apropiado con la sentencia SQL que queremos enviar. Para una sentencia **SELECT**, el método a ejecutar es **executeQuery**. Para sentencias que crean o modifican tablas, el método a utilizar es **executeUpdate**.

Se toma un ejemplar de una conexión activa para crear un objeto **Statement**. En el siguiente ejemplo, utilizamos nuestro objeto **Connection (conecta)** para crear el objeto **Statement (consulta)**.

```
Statement consulta = conecta.createStatement();
```

En este momento **consulta** existe, pero no tiene ninguna sentencia SQL que pasarle al controlador de la base de datos. Necesitamos suministrarle el método que utilizaremos para ejecutar **consulta**. El método más utilizado para ejecutar sentencias SQL es **executeQuery**. Este método se utiliza para ejecutar sentencias **SELECT**, que comprenden la amplia mayoría de las sentencias SQL.

4. Recuperar valores. (ejecutar consultas)

Ahora veremos como ejecutar una consulta y como obtener los resultados.

JDBC devuelve los resultados en un objeto **ResultSet**, por eso necesitamos declarar un ejemplar de la clase **ResultSet** para contener los resultados. El siguiente código presenta el objeto **ResultSet** denominado **resul** y le asigna el resultado de una consulta.

```
ResultSet resul = consulta.executeQuery("select deptno,dname from dept");
```

La variable **resul**, que es un ejemplar de **ResultSet**, contiene las filas de la consulta. Para acceder a los nombres y los códigos, iremos a la fila y recuperaremos los valores de acuerdo con sus tipos.

El método **next** mueve el cursor (una especie de puntero, es parecido a los cursores vistos en PL/SQL) a la siguiente fila y hace que esa fila (llamada fila actual) sea con la que podamos operar. Como el cursor inicialmente se posiciona justo encima de la primera fila de un objeto **ResultSet**, primero debemos llamar al método **next** para mover el cursor a la primera fila y convertirla en la fila actual. Sucesivas invocaciones del método **next** moverán el cursor de línea en línea. Dependiendo de la versión JDBC se puede mover el cursor hacia atrás, hacia posiciones específicas y a posiciones relativas a la fila actual además de mover el cursor hacia adelante.

Los métodos **getXXX (getInt, getFloat, getString, etc)** del tipo apropiado se utilizan para recuperar el valor de cada columna. Por ejemplo, la primera columna de cada fila de **resul** es **deptno**, que almacena un valor del tipo **NUMBER** de SQL. El método para recuperar un valor de ese tipo es **getInt**. La segunda columna de cada fila almacena un valor del tipo **VARCHAR2** de SQL, y el método para recuperar valores de ese tipo es **getString**. El siguiente código accede a los valores almacenados en la fila actual de **resul** e imprime una línea con el *código* y *el nombre*. Cada vez que se llama al método **next**, la siguiente fila se convierte en la actual, y el bucle continúa hasta que no haya más filas en **resul**.

```
while (resul.next()){
    int    cod = resul.getInt("deptno");
    String nom = resul.getString("dname");
    System.out.println(cod + "    " + nom);
}
```

Veamos el programa completo

```
package beans;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class EjemploBd {

    public static void main(String[] args)

throws SQLException, ClassNotFoundException {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection
conecta=DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "rasty", "rasty");
```

```
Statement consulta =conecta.createStatement();
ResultSet resul = consulta.executeQuery("select deptno,dname from
dept");
while (resul.next()){
    int cod = resul.getInt("deptno");
    String nom = resul.getString("dname");
    System.out.println(cod + "    " + nom);
}
consulta.close();
}
```

5. Sentencias preparadas

Algunas veces es más conveniente o eficiente utilizar objetos **PreparedStatement** para enviar sentencias SQL a la base de datos. Este tipo especial de sentencias se deriva de una clase más general, **Statement**, que ya conocemos.

Si queremos ejecutar muchas veces un objeto **Statement** reduciremos el tiempo de ejecución si utilizamos un objeto **PreparedStatement** en su lugar.

La característica principal de un objeto **PreparedStatement** es que, al contrario de un objeto **Statement**, se le entrega una sentencia SQL cuando se crea. La ventaja de esto es que en la mayoría de los casos, esta sentencia SQL se enviará al controlador de la base de datos inmediatamente, donde será compilada. Como resultado, el objeto **PreparedStatement** no sólo contiene una sentencia SQL, sino una sentencia SQL que ha sido precompilada. Esto significa que cuando se ejecuta la **PreparedStatement**, el controlador de base de datos puede ejecutarla sin tener que compilarla primero.

Aunque los objetos **PreparedStatement** se pueden utilizar con sentencias SQL sin parámetros, probablemente nosotros utilizaremos más frecuentemente sentencias con parámetros. La ventaja de utilizar sentencias SQL que utilizan parámetros es que podemos utilizar la misma sentencia y suministrar distintos valores cada vez que la ejecutemos.

Al igual que los objetos **Statement**, creamos un objeto **PreparedStatement** con un objeto **Connection**. Utilizando nuestra conexión **conecta** abierta en ejemplos anteriores, podríamos escribir lo siguiente para crear un objeto **PreparedStatement** que tome dos parámetros de entrada.

```
PreparedStatement actualiza = conecta.prepareStatement(
    "UPDATE dept SET dname = ? WHERE deptno LIKE ?");
```


La variable **actualiza** contiene la sentencia SQL, "**UPDATE dept SET dname = ? WHERE deptno LIKE ?**"; que también ha sido enviada al controlador de la base de datos, y ha sido precompilada.

5.1. Suministrar valores para los parámetros de un *PreparedStatement*.

Necesitamos suministrar los valores que se utilizarán en los lugares donde están las marcas de interrogación antes de ejecutar un objeto **PreparedStatement**. Podemos hacer esto llamando a uno de los métodos **setXXX** definidos en la clase **PreparedStatement**. Si el valor que queremos sustituir por una marca de interrogación es un **int** de Java, podemos llamar al método **setInt**. Si el valor que queremos sustituir es un **String** de Java, podemos llamar al método **setString**, etc. En general, hay un método **setXXX** para cada tipo Java.

Utilizando el objeto **actualiza** del ejemplo anterior, la siguiente línea de código selecciona la primera marca de interrogación para un **string** de Java, con un valor de "VENTAS".

```
actualiza.setString(1, "VENTAS");
```

El primer argumento de un método **setXXX** indica la marca de interrogación que queremos seleccionar, y el segundo argumento el valor que queremos ponerle. El siguiente ejemplo selecciona la segunda marca de interrogación con el **int** "10".

```
actualiza.setInt(2, 10);
```

Finalmente el código sería el siguiente:

```
PreparedStatement actualiza = conecta.prepareStatement  
    ("UPDATE dept SET dname = ? WHERE deptno LIKE ?");  
actualiza.setString(1, "VENTAS");  
actualiza.setInt(2, 10);  
actualiza.executeUpdate();
```

El siguiente código es equivalente al anterior:

```
String updateString = "UPDATE detp SET dname = VENTAS " +  
    "WHERE deptno LIKE '10'";  
actualiza.executeUpdate(updateString);
```

Utilizamos el método **executeUpdate** para ejecutar ambas sentencias **actualiza**.

Mirando estos ejemplos podríamos preguntarnos por qué utilizar un objeto **PreparedStatement** con parámetros en vez de una simple sentencia, ya que la sentencia simple implica menos pasos. Si la actualización de las columnas es poco frecuente no sería necesario utilizar una sentencia SQL con parámetros. Si se realiza frecuentemente podría ser más eficiente utilizar un objeto **PreparedStatement**.

El siguiente código devuelve el número de líneas actualizadas en la variable **n**

```
int n = actualiza.executeUpdate();
```

El siguiente ejemplo muestra un programa completo con lo comentado en el epígrafe anterior

```
package beans;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class VerPrepaState {

    public static void main(String[] args)
        throws SQLException, ClassNotFoundException {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection
conecta=DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe","rasty","rasty");
        PreparedStatement actualiza = conecta.prepareStatement
("UPDATE dept SET dname = ? WHERE deptno LIKE ?");
        actualiza.setString(1, "VENTAS");
        actualiza.setInt(2, 10);
        int n = actualiza.executeUpdate();
        System.out.println("actualizadas" + " " + n + " filas");
        actualiza.close();
    }
}
```

6. Transacciones

Retomamos el concepto de transacción visto en la unidad de SQL. Una transacción es un conjunto de una o más sentencias que se ejecutan como una unidad, por eso o se ejecutan todas o no se ejecuta ninguna. Cuando se crea una conexión, está en modo auto-commit. Esto significa que cada sentencia SQL individual es tratada como una transacción y será automáticamente entregada justo después de ser ejecutada. La forma de permitir que dos o más sentencias sean agrupadas en una transacción es desactivar el modo auto-commit. Esto se muestra en el siguiente código, donde **conecta** es una conexión activa.

```
conecta.setAutoCommit(false);
```

Una vez que se ha desactivado la auto-commit, no se entregará ninguna sentencia SQL hasta que llamemos explícitamente al método **commit**. Todas las sentencias ejecutadas antes del método **commit** serán incluidas en la transacción actual y serán entregadas juntas como una unidad. El siguiente código, en el que **conecta** es una conexión activa, ilustra una transacción.

```
conecta.setAutoCommit(false);
PreparedStatement updateDep = conecta.prepareStatement(
    "UPDATE dept SET dname = ? WHERE deptno LIKE ?");
updateDep.setInt(1, "VENTAS");
updateDep.setString(2, 10);
updateDep.executeUpdate();
PreparedStatement updateEmp = conecta.prepareStatement(
    "UPDATE emp SET sal = sal + ? WHERE empno LIKE ?");
updateEmp.setInt(1, 100);
updateEmp.setString(2, 7321);
updateEmp.executeUpdate();
conecta.commit();
conecta.setAutoCommit(true);
```

En este ejemplo, el modo auto-commit se desactiva para la conexión **conecta**, lo que significa que las dos sentencias prepared: la **updateDep** y **updateEmp** serán entregadas juntas cuando se llame al método **commit**.

La línea final del ejemplo anterior activa el modo auto-commit, lo que significa que cada sentencia será de nuevo entregada automáticamente cuando esté completa. Volvemos por lo tanto al estado por defecto, en el que no tenemos que llamar al método **commit**. Es bueno

desactivar el modo auto-commit sólo mientras queramos estar en modo transacción. De esta forma, evitamos bloquear la base de datos durante varias sentencias.

6.1. Deshacer una transacción

Llamando al método **rollback**, como ya se vio en SQL, se aborta la transacción y devuelve cualquier valor que fuera modificado a sus valores anteriores. Si estamos intentando ejecutar una o más sentencias en una transacción y obtenemos una **SQLException**, deberíamos llamar al método **rollback** para abortar la transacción y empezarla de nuevo. Esta es la única forma para asegurarnos de cuál ha sido entregada y cuál no ha sido entregada. Capturar una **SQLException** nos dice que hay algo erróneo, pero no nos dice si fue o no fue entregada.

7. Inserciones y consultas a través de formularios

En este epígrafe vamos generar código para hacer inserciones en la tabla **dept** del usuario **rasty** y también poder realizar consultas sobre la misma tabla.

Para realizar este trabajo creamos un proyecto nuevo siguiendo las pautas marcadas en ejemplos anteriores. Se crea un paquete, para este ejemplo que se denomina *paquetes* y dentro de él una clase que en este caso se denomina *AccesoBdatos*. En dicha clase lo primero que se define es un método para establecer la conexión, lo denominamos *conectar*.

```
package paquetes;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Vector;
public class AccesoBdatos {
    Connection conecta;
    public void conectar()
        throws SQLException, ClassNotFoundException {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        conecta=DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:
                                         xe", "rasty", "rasty");
    }
}
```

En el método *conectar*

- se establecen que excepciones pueden producirse (throws),
- se cargan los drivers de oracle (Class.forName). Acordaros de que tienen que existir en la carpeta LIB del Tomcat
- se establece la conexión con la base de datos (DriverManager.getConnection) que se guarda en el objeto *conecta*

A partir de aquí ya se puede trabajar con la base de datos.

7.1. Inserciones

Vamos crear un método (en la clase anterior) que permita insertar datos en la tabla *dept* del usuario *rasty*

```
public void insertar(Integer clave, String nombre, String localidad){
    try {
        String sql="insert into dept values (?,?,?)";
        PreparedStatement inserta
            =conecta.prepareStatement(sql);

        inserta.setInt(1,clave);
        inserta.setString(2,nombre);
        inserta.setString(3,localidad);
        inserta.executeUpdate();
    } catch(SQLException e){
        System.out.println("error al insertar en
            dept"+e.getMessage());
    }
}
```

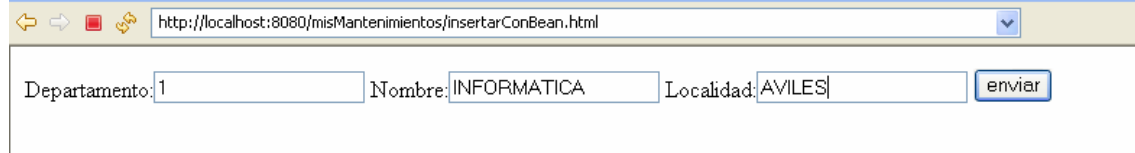
Seguimos dentro de la clase *AccesoBdatos*. En el método *insertar* se definen tres parámetros que van recibir los valores que se quieren insertar. Se crea una variable del tipo String que va contener la sentencia insert, se puede prescindir de la variable y colocar la sentencia directamente en *preparedStatement*

Las interrogaciones de la sentencia insert se sustituirán por los valores que se les pasan desde los métodos set del objeto *inserta*.

A través del siguiente formulario se introducen los datos que se insertaran en la tabla *dept*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Formulario para la introducción de los datos</title>
```

```
</head>
<body>
<form action="insertando.jsp">
  Departamento:<input type="text" name="cod" />
  Nombre:<input type="text" name="nom" />
  Localidad:<input type="text" name="loc">
  <input type="submit" value="enviar"/>
</body>
</html>
```



http://localhost:8080/misMantenimientos/insertarConBean.html

Departamento: 1 Nombre: INFORMATICA Localidad: AVILES enviar

El fichero JSP (*insertando*) se encarga de recoger los valores y hacer la inserción a través del método *insertar* definido en la clase *AccesoBdatos*

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="datos" scope="session" class="paquetes.AccesoBdatos" />
<%
datos.conectar();
try {
datos.insertar(Integer.parseInt(request.getParameter("cod")),
    request.getParameter("nom"),request.getParameter("loc"));
}
catch (Exception e) { out.println("error en datos " + e.getMessage());
}
response.sendRedirect("insertar.html");
%>
</body>
</html>
```

Otra versión del ejemplo anterior. Trabajando con Beans

Otra forma de trabajar es crear una clase con los métodos *set* y *get*, de esta forma se simplifica el código en el fichero JSP, ya que trabajamos con la estructura de la tabla definida en el JavaBean. La clase con el constructor es la siguiente

```
package paquetes;

public class Depto {
    Integer deptno;
    String dname;
    String loc;
    public Depto(Integer deptno, String dname, String loc) {
        super();
        this.deptno = deptno;
        this.dname = dname;
        this.loc = loc;
    }
    public Depto() {
    }
    public Integer getDeptno() {
        return deptno;
    }
    public void setDeptno(Integer deptno) {
        this.deptno = deptno;
    }
    public String getDname() {
        return dname;
    }
    public void setDname(String dname) {
        this.dname = dname;
    }
    public String getLoc() {
        return loc;
    }
    public void setLoc(String loc) {
        this.loc = loc;
    }
}
}
```

Después de definir los tipos de datos de la clase, los métodos *getters* y *setters* se generan de la siguiente forma:

Botón derecho → *source* → *generate constructor use fields* y luego repetir la operación con → *generate getters and setters*

Se crea un nuevo método en la clase *AccesoBdatos* que se utilizará en vez del *insertar* que denominaremos *insertarConBean* y que tiene el siguiente código:

```
public void insertarConBean(Depto registro){
    try {
        String sql="insert into dept values (?,?,?)";
        PreparedStatement inserta =conecta.prepareStatement(sql);
        inserta.setInt(1,registro.getDeptno());
        inserta.setString(2,registro.getDname());
        inserta.setString(3,registro.getLoc());
    }
}
```

```
        inserta.executeUpdate();

    } catch(SQLException e){
        System.out.println("error al insertar en dept"+e.getMessage());
    }
}
```

Con esta nueva versión el formulario y el fichero JSP contiene el siguiente código

El Fichero HTML es el siguiente

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>formulario para insertar con bean</title>
</head>
<body>
<form action="insertandoConBean.jsp">
  Departamento:<input type="text" name="deptno" />
  Nombre:<input type="text" name="dname" />
  Localidad:<input type="text" name="loc">
  <input type="submit" value="enviar"/>
</body>
</html>
```

Los campos del formulario tienen que llamarse igual que las propiedades del Bean, para que haya una correspondencia entre el formulario y la clase, de forma que al introducir los datos en el formulario automáticamente se actualicen las propiedades del objeto Bean con esos datos.

El fichero JSP es el siguiente (*insertandoConBean.jsp*)

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insertando con java bean</title>
</head>
<body>
<jsp:useBean id="datos" scope="session" class="paquetes.AccesoBdatos" />
<jsp:useBean id="datosTabla" scope="session" class="paquetes.Depto" />
<jsp:setProperty name="datosTabla" property="*" />
```



```
<%
datos.conectar();
try {
    datos.insertarConBean(datosTabla);
}
catch (Exception e) { out.println("error en datos " + e.getMessage());
}
response.sendRedirect("insertarConBean.html");
%>
</body>

</html>
```

Se puede volver a la página html con forward como ya vimos en epígrafes anteriores o con `response.sendRedirect("insertarConBean.html");`

7.2. Consultas

Vamos a consultar la tabla *dept* utilizando la clase *Depto* (JavaBean creado para realizar las inserciones del ejemplo anterior), esta clase no hay que modificarla. Se tiene que añadir un nuevo método a la clase *AccesoBdatos* que establece la conexión y accede a la tabla *dept* el código es el siguiente:

```
public Collection consultarConBean(){
    ArrayList deptos = new ArrayList ();
    try {

        PreparedStatement consulta = conecta.prepareStatement("SELECT
                                                                * FROM dept");

        ResultSet reg = consulta.executeQuery ();
        while (reg.next ()) {
            Depto departamento = new Depto(
                reg.getInt (1), reg.getString (2), reg.getString (3));
            deptos.add(departamento);
        }
        consulta.close ();
    }
    catch (SQLException e) {
        return null;
    }
    return deptos;
}
```

Se define una colección de tipo *ArrayList* denominada *deptos*. Como en las inserciones se define la sentencia sql que se quiere ejecutar, en este caso se trata de consultar toda la tabla `SELECT * FROM dept` la variable *reg* del objeto *ResultSet* contiene las filas que se recuperan

de la tabla, se define un bucle while para recuperar cada fila y guardarla en la *array* con la estructura del registro definido en la clase *Depto*. Por último, si se produce alguna excepción se ignora. Si no hay excepciones cuando termina el proceso se devuelve el control a quién llamó al método, en este caso, *consultarConBean* del fichero JSP del mismo nombre.

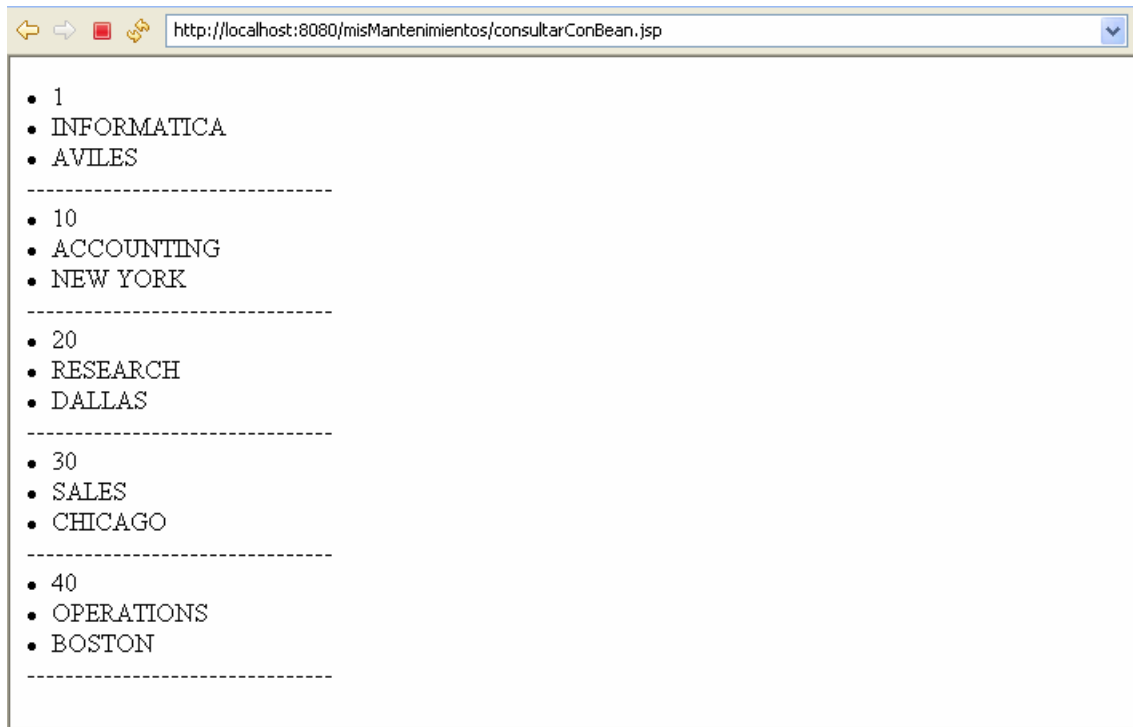
El siguiente código corresponde a la página JSP que llama al método y muestra el resultado.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" import="java.util.*,paquetes.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="datos" scope="session" class="paquetes.AccesoBdatos" />
<jsp:useBean id="datosTabla" scope="session" class="paquetes.Depto" />

<%
datos.conectar();
Collection departamentos = datos.consultarConBean();
if (departamentos != null)
{
    if (departamentos.size () > 0)
    {
        for (Iterator i = departamentos.iterator(); i.hasNext(); )
        {
            Depto departamento = (Depto) i.next ();
            %>
            <li> <%= departamento.getDeptno () %></li>
            <li> <%= departamento.getDname () %></li>
            <li> <%= departamento.getLoc() %></li>
            <br><a>-----</a>
            <%
            }
        }
    }
}
%>
</body>
</html>
```

Este es el fichero que hay que ejecutar para visualizar las filas de la tabla. Se define una variable del tipo colección donde se recuperan los datos que devuelve el método *consultarConBean*, recorriendo la estructura con un *for* se van visualizando cada una de las filas, en este caso se mezcla el código HTML

 con las expresiones del tipo <%= departamento.getDeptno () %>



7.3. Consultar un departamento

Veamos una variante más. El siguiente método se utiliza para recuperar un departamento concreto cuyo número se teclea. Lo mismo que en los casos anteriores, se incluye en la clase *AccesoBdatos*.

```

public Depto consultarUno(int numero){
    try {
        PreparedStatement consulta = conecta.prepareStatement("SELECT
                                                                * FROM dept WHERE deptno=?");
        consulta.setInt(1, numero);
        ResultSet reg = consulta.executeQuery ();
        Depto departamento = new Depto();
        if (reg.next ())
        {
            departamento.setDeptno(reg.getInt (1));
            departamento.setDname(reg.getString(2));
            departamento.setLoc(reg.getString(3));
        }
        consulta.close ();
        return departamento;
    }
    catch (SQLException e) {
        System.out.println("error en la consulta"+e.getMessage());
        return null;
    }
}

```

En este caso el método no es de tipo colección, es del tipo de la clase *Depto*, tiene un parámetro que recibe el número del departamento que se quiere recuperar, ya no se necesita una *while* ya que se recupera una única fila que se comprueba con la *if*.

La página JSP que se ejecuta es la siguiente:

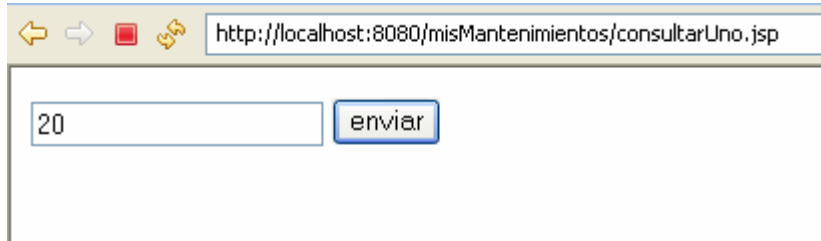
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" import="java.util.*,paquetes.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>página consultarUno que trabaja con el método del mismo
nombre</title>
</head>
<body>
<jsp:useBean id="datos" scope="session" class="paquetes.AccesoBdatos" />
<jsp:useBean id="datosTabla" scope="session" class="paquetes.Depto" />
<form >
<input type="text" name="num" >
<input type="submit" value="enviar">
</form>
<%
datos.conectar();
String valor = request.getParameter("num");
if (valor != null)
{
    int numero = Integer.parseInt(valor);
    Depto departamentos = datos.consultarUno(numero);
    if (departamentos != null)
    {
        %>
        <li> <%= departamentos.getDeptno () %></li>
        <li> <%= departamentos.getDname () %></li>
        <li> <%= departamentos.getLoc() %></li>
        <br><a>-----</a>
        %>
    }
}
%>
</body>
</html>
```

Como el número de departamento que se teclea es de tipo texto hay que convertirlo, se recupera con `request.getParameter("num")`, se comprueba si es nulo para pasarlo a entero `numero = Integer.parseInt(valor)`, este es el valor que se le pasa al método

y devuelve la estructura con los datos de la tabla correspondiente al número del departamento que se le pasó.

Se muestran las siguientes páginas:

Pide el número



Muestra los datos del departamento correspondientes al nº 20



8. Actualizaciones

Seguimos avanzando con más módulos. Ahora incluiremos un método para actualizar los datos de la tabla *dept* excepto el número del departamento que es clave primaria. El código se muestra en el siguiente recuadro. Lo mismo que los anteriores, se incluye en la clase *AccesoBdatos*.

```
public void actualizar(Depto registro){
    try {
        String sql="update dept set dname=?, loc=? where deptno=?";
        PreparedStatement actualiza = conecta.prepareStatement(sql);
        actualiza.setString(1,registro.getDname());
        actualiza.setString(2,registro.getLoc());
        actualiza.setInt(3,registro.getDeptno());
        actualiza.executeUpdate();

    } catch(SQLException e){
        System.out.println("error al actualizar en
                                dept "+e.getMessage());
    }
}
```

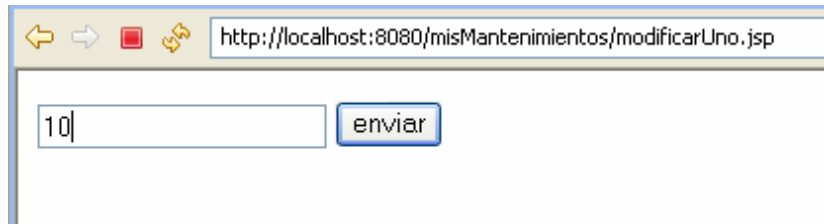
Como podéis observar es igual que el de *insertar* solo cambia el orden de los parámetros y el nombre de *inserta* por *actualiza*.

La página JSP (*modificarUno.jsp*) que pide el número del departamento y muestra los contenidos es la siguiente:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" import="paquetes.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>modificarUno</title>
</head>
<body>
<jsp:useBean id="datos" scope="session" class="paquetes.AccesoBdatos" />
<jsp:useBean id="datosTabla" scope="session" class="paquetes.Depto" />
<form >
<input type="text" name="num" >
<input type="submit" value="enviar">
</form>
<%
datos.conectar();
String valor = request.getParameter("num");
if (valor != null)
{
    int numero = Integer.parseInt(valor);
    Depto departamentos = datos.consultarUno(numero);
    if (departamentos != null)
    {
        %>
        <form action="actualizar.jsp">
        <input readonly type="text" name="deptno" value ="<%=
            departamentos.getDeptno () %>">
        <input type="text" name="dname" value ="<%=
            departamentos.getDname () %>">
        <input type="text" name="loc" value ="<%=
            departamentos.getLoc () %>">
        <input type="submit" value="enviar">
        </form>
        <%
    }
}
%>
</body>
</html>
```

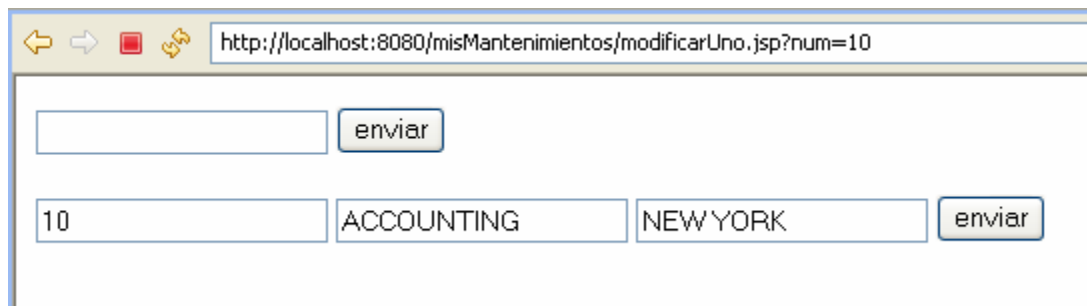
Se ve que es la misma que la de *consultarUno*, solo cambia la presentación de los datos que en este caso los presenta como formulario en vez de una lista, lo que permite cambiar el contenido, excepto el número de departamento *deptno* que es de solo lectura *readonly*. Una vez

presentados y modificados los datos se llama a la página *actualizar.jsp* que es la que lanza la actualización.



Se teclea el número de departamento que se quiere modificar, al pulsar en *enviar* se visualizan todos los datos del departamento.

Una vez presentados se pueden modificar todos excepto el código



Código de la página *actualizar.jsp*

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" import="paquetes.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>actualizar</title>
</head>
<body>
<jsp:useBean id="datos" scope="session" class="paquetes.AccesoBdatos" />
<jsp:useBean id="datosTabla" scope="session" class="paquetes.Depto" />
<jsp:setProperty name="datosTabla" property="*" />
<%
datos.conectar();
try {
    datos.actualizar(datosTabla);
```

```

    }
    catch (Exception e) { out.println("error en datos " +
                                     e.getMessage());
    }
    response.sendRedirect("modificarUno.jsp");
%>
</body>
</html>

```

Podéis observar que es idéntica a la de insertar solo cambia el método al que llama y la página a la que redirecciona una vez hecha la actualización.

Esta facilidad de reutilizar el código se debe a la utilización del JavaBean *Depto*, de lo contrario si se hace como el primer ejemplo de inserción el código sería diferente para cada una de estas funciones.

En la pantalla anterior seleccionando *enviar* actualizaría la localidad del departamento 10.

9. Eliminar departamentos

Otro módulo que se debe incluir en la aplicación es el de bajas de registros, en este caso borra departamentos en base al número que se teclea.

El método que se debe incluir en la clase de conexión y manipulación de la base de datos (*AccesoBdatos*) es el siguiente:

```

public void bajas(int numero){
    try {
        PreparedStatement consulta = conecta.prepareStatement("DELETE FROM
dept WHERE deptno=?");
        consulta.setInt(1, numero);
        ResultSet reg = consulta.executeQuery ();
        consulta.close ();
    }
    catch (SQLException e) {
        System.out.println("error en la consulta"+e.getMessage());
    }
}

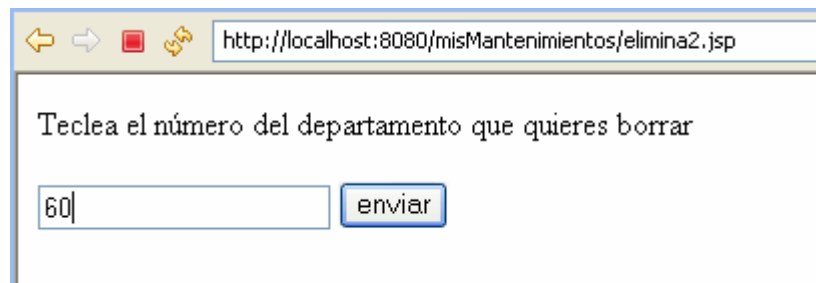
```

}

Código de la página desde la que se lanza la petición de bajas

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" import="java.util.*,paquetes.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="datos" scope="session" class="paquetes.AccesoBdatos" />
<jsp:useBean id="datosTabla" scope="session" class="paquetes.Depto" />
<form >
<p>Teclea el número del departamento que quieres borrar </p>
<input type="text" name="num" >
<input type="submit" value="enviar">
</form>
<%
datos.conectar();
String valor = request.getParameter("num");
if (valor != null)
{
    int numero = Integer.parseInt(valor);
    datos.bajas(numero);
}
%>
</body>
</html>
```

Visualización de la página



Se teclea el departamento y al enviar se elimina de la base de datos.

El siguiente código muestra otra versión para eliminar un departamento, en este caso el método devuelve el número de filas borradas.

El método que se incluye en la clase (*AccesoBdatos*) es el siguiente:

```
public int bajas2(int numero){
    int filas=0;
    try {
        PreparedStatement baja = conecta.prepareStatement("DELETE FROM
dept WHERE deptno=?");
        baja.setInt(1, numero);
        filas = baja.executeUpdate ();
        baja.close ();
        return filas;
    }
    catch (SQLException e) {
        System.out.println("error en la consulta"+e.getMessage());
        return 0;
    }
}
```

El código del fichero *jsp* es el que se muestra a continuación:

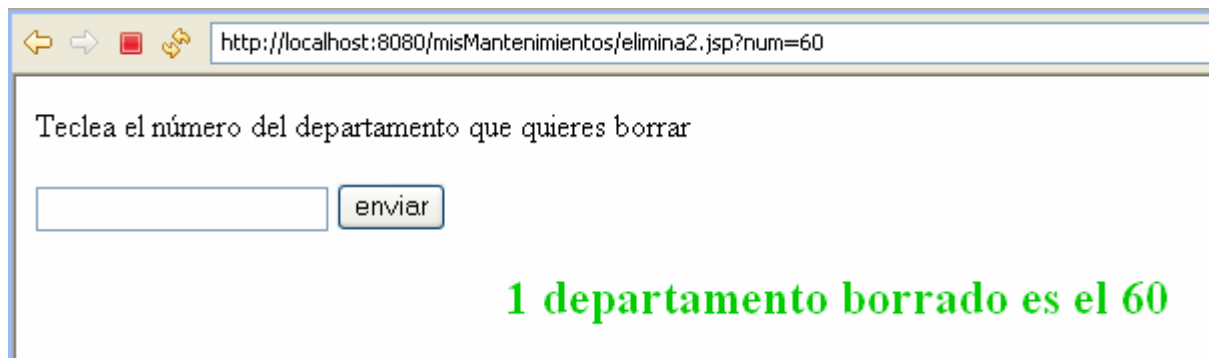
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="datos" scope="session" class="paquetes.AccesoBdatos" />
<jsp:useBean id="datosTabla" scope="session" class="paquetes.Depto" />
<form >
<p>Teclea el número del departamento que quieres borrar </p>
<input type="text" name="num" >
<input type="submit" value="enviar">
</form>
<%
datos.conectar();
String valor = request.getParameter("num");
if (valor != null)
{
    int numero = Integer.parseInt(valor);
    int fila = datos.bajas2(numero);

    if (fila==1) {
        %>
```

```
        <center>
        <h2><font color="#00cc00">Se ha borrado el
                                departamento <%= numero %>
        </font></h2>
        </center>
<%
    }
    else {
<%
        <center>
        <h2><font color="#cc0000">No se pudo borrar
                                el departamento <%=numero%>
        </font></h2>
        </center>
<%
    }
}
%>
</body>
</html>
```

Se puede ver que la diferencia fundamental es que al devolver el método un valor hay que recogerlo en algún sitio. `int fila = datos.bajas2(numero)`, en el caso anterior se llamaba sin más y no devolvía nada (void).

El resultado de la ejecución es la siguiente página



La pantalla se puede formatear separando el código HTML de java, normalmente se utilizaría código CSS que sustituiría el HTML, en este caso

```
<center>
  <h2><font color="#cc0000">No
    se pudo borrar el departamento
  </font></h2>
</center>
```

10. Procedimientos almacenados. *CallableStatement*

Ya hemos visto en unidad dedicada a PL/SQL en que consistía un procedimiento almacenado. Recordamos que un procedimiento almacenado es un grupo de sentencias SQL que forman una unidad lógica y que realizan una tarea particular. Los procedimientos almacenados se utilizan para encapsular un conjunto de operaciones o peticiones para ejecutar en un servidor de base de datos. Por ejemplo, las operaciones sobre una base de datos de empleados (salarios, comisiones, antigüedad) podrían ser codificadas como procedimientos almacenados ejecutados por el código de la aplicación. Los procedimientos almacenados pueden compilarse y ejecutarse con diferentes parámetros y resultados, y podrían tener cualquier combinación de parámetros de entrada/salida.

Desde el entorno SQL*Plus creamos el siguiente procedimiento almacenado.

```
CREATE PROCEDURE consultaEmp AS
BEGIN
    SELECT ename,sal, comm FROM emp WHERE sal > 1000;
END;
```

- Para llamar al procedimiento desde PL/SQL

```
BEGIN
    consultaEmp;
END;
```

- Para llamarlo desde Java

JDBC permite llamar a un procedimiento almacenado en la base de datos desde una aplicación escrita en Java. El primer paso es crear un objeto **CallableStatement**. Al igual que los objetos **Statement** y **PreparedStatement**, esto se hace con una conexión abierta, **Connection**. Un objeto **CallableStatement** contiene una llamada a un procedimiento almacenado; no contiene el propio procedimiento. La primera línea del código siguiente crea una llamada al procedimiento almacenado **consultaEmp** utilizando la conexión **conecta**. La parte que está encerrada entre corchetes es la sintaxis

de escape para los procedimientos almacenados. Cuando un controlador encuentra "{call consultaEmp}", traducirá esta sintaxis de escape al SQL nativo utilizado en la base de datos para llamar al procedimiento almacenado llamado **consultaEmp**.

```
CallableStatement consu = conecta.prepareCall("{call consultaEmp}");  
ResultSet resultado = consu.executeQuery();
```

En **resultado** tendremos las filas devueltas por la consulta del procedimiento.

Observa que el método utilizado para ejecutar **consu** es **executeQuery** porque **consu** llama a un procedimiento almacenado que contiene una consulta y esto produce una hoja de resultados. Si el procedimiento hubiera contenido una sentencia de actualización o una sentencia DDL, se hubiera utilizado el método **executeUpdate**. Sin embargo, en algunos casos, cuando el procedimiento almacenado contiene más de una sentencia SQL producirá más de una hoja de resultados, en estos casos, donde existen múltiples resultados, se debería utilizar el método **execute** para ejecutar **CallableStatement**.

La clase **CallableStatement** es una subclase de **PreparedStatement**, por eso un objeto **CallableStatement** puede tomar parámetros de entrada como lo haría un objeto **PreparedStatement**. Además, un objeto **CallableStatement** puede tomar parámetros de salida, o parámetros que son tanto de entrada como de salida.

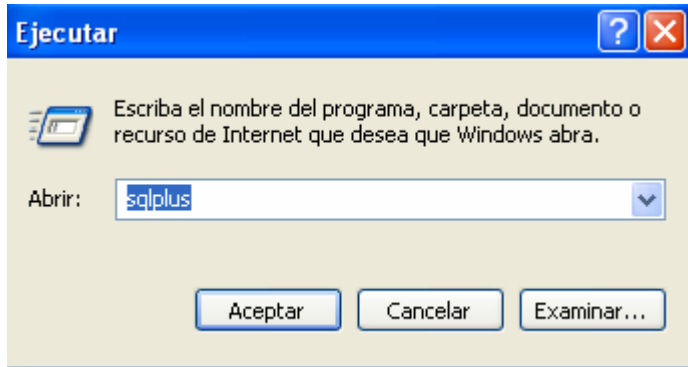
Vamos seguir un ejemplo que consistirá en crear y almacenar un procedimiento en el esquema de *rasty* que permita actualizar las localidades de los departamentos pasándole el código del departamento que se quiere actualizar y la nueva localidad. Los datos código y localidad se introducen desde un formulario.

Para seguir un orden lógico lo primero que haremos será crear el siguiente procedimiento escribiéndolo en el bloc de notas :

```
CREATE OR REPLACE PROCEDURE  actualizaDept(cod NUMBER,  
                                           localidad VARCHAR2) AS  
BEGIN  
  UPDATE  Dept SET loc=localidad WHERE deptno = cod;  
END;
```

Recordad que *oracle* no diferencia las mayúsculas de las minúsculas, en cambio con Java hay que tener cuidado ya que establece diferencias.

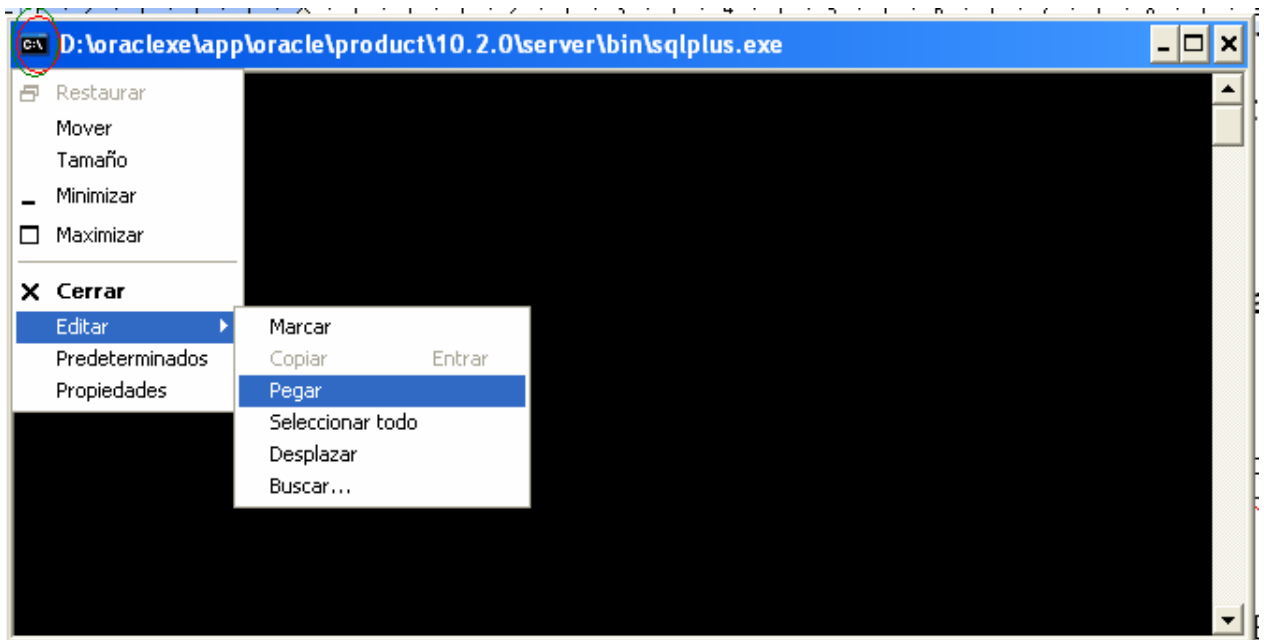
El fichero que contiene el código anterior se llama *actuali.sql* para crearlo en oracle entre otras opciones la más sencilla se abrir una ventana **SqlPlus** (inicio → ejecutar y el nombre del programa es *sqlplus*)



Luego ejecutar con *START path\fichero.sql*. Si se produce algún error se puede visualizar con el comando SHOW ERRORS.

```
C:\ D:\oracle\app\oracle\product\10.2.0\server\bin\sqlplus.exe
SQL> START D:\J2EE\ACTUALI.SQL
Procedimiento creado.
SQL> _
```

También podéis copiarlo desde el block de notas y pegarlo en la ventana SQLPLUS siguiendo la secuencia que se muestra en la siguiente captura.



Por último también podéis crearlo desde el entorno web de oracle. Elegid la opción que consideréis oportuna.

Una vez creado el procedimiento creamos el método que lo accede y ejecuta. Con *CallableStatement* creamos el objeto que hace la llamada al procedimiento y en este caso al tratarse de una actualización lo ejecutamos con *executeUpdate()*. Este método se crea en la misma clase que los demás (*AccesoBdatos*). El código completo del mismo es el siguiente:

```
public void llamaProc(Integer clave, String localidad){
    try {
        CallableStatement actu = conecta.prepareCall("{call
                                                    actualizaDept(?,?)}");
        actu.setInt(1,clave);
        actu.setString(2,localidad);
        actu.executeUpdate();
    }
    catch (SQLException e) {
        System.out.println("error en la
                            actualización"+e.getMessage());
    }
}
```

El código de la página JSP que pide los datos y llama al método es la siguiente:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
```

```

"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="datos" scope="session" class="paquetes.AccesoBdatos" />
<form >
  Departamento:<input type="text" name="cod" />
  Localidad:<input type="text" name="loc">
  <input type="submit" value="enviar"/>
</form>
<%
datos.conectar();
if (request.getParameter("loc") != null) {
try {
    datos.llamaProc(Integer.parseInt(request.getParameter("cod")),re
                    quest.getParameter("loc"));
    }
    catch (Exception e) { out.println("error en datos " +
                                e.getMessage());
    }
}
%>
</body>
</html>

```

Para la siguiente entrada

Departamento: Localidad:

La tabla queda actualizada con el contenido que se muestra en la siguiente captura.

```

C:\D:\oracle\app\oracle\product\10.2.0\server\bin\sql
SQL> SELECT * FROM DEPT;

```

DEPTNO	DNAME	LOC
10	ACCOUNTING	AVILES
20	RESEARCH	luanco
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

```

SQL>

```


Nota: El código 20 se modificó en una actualización anterior.

10.1. Eliminar tipos de objetos del esquema de usuario

Vamos retomar el procedimiento de la unidad de PL/SQL que se utilizaba para eliminar objetos de un tipo que se pasaba como parámetro. Por ejemplo: VIEW, TABLE, PROCEDURE, etc.

El procedimiento almacenado es el siguiente:

```
CREATE OR REPLACE PROCEDURE BORRA_OBJETOS
    (TIPO USER_OBJECTS.OBJECT_TYPE%TYPE) AS
ID INTEGER;
CURSOR C1 IS SELECT OBJECT_NAME FROM USER_OBJECTS WHERE OBJECT_TYPE=TIPO;
BEGIN
FOR REG IN C1 LOOP
ID:=DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(ID,'DROP '|| TIPO || ' ' || REG.OBJECT_NAME, DBMS_SQL.V7);
DBMS_SQL.CLOSE_CURSOR(ID);
END LOOP;
END;
/
```

Si el procedimiento anterior no lo tenéis creado en vuestro usuario lo copiáis en un fichero de texto y lo guardáis con extensión SQL. Ejecutáis *SqlPlus* y lo cargáis con el comando *Start*.

También podéis copiarlo y pegarlo sin más en el entorno SQLPLUS.

Para hacer las pruebas tenéis que crear las vistas que se muestran en la siguiente captura.

```
SQL> create view vista1 as select dname from dept;
Vista creada.
SQL> create view vista2 as select dname from dept;
Vista creada.
SQL> select * from User_catalog;
TABLE_NAME          TABLE_TYPE
-----
VISTA2              VIEW
VISTA1              VIEW
DEPT                TABLE
EMP                 TABLE
```

En la imagen se ven los objetos que existen antes de que se ejecute el procedimiento .

El método que deberéis incluir en la clase *AccesoBdatos* es el siguiente

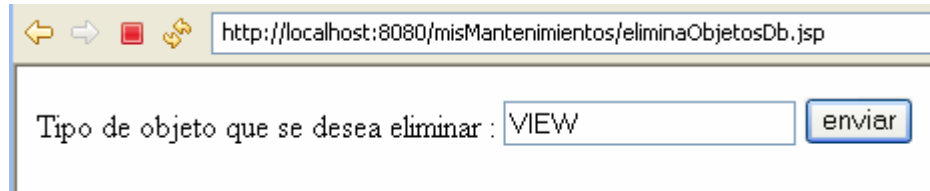
```
public void llamaProc2(String tipo){
    try {
        CallableStatement borra = conecta.prepareStatement("{call
                                                                borra_objetos(?)}");
        borra.setString(1,tipo);
        borra.executeUpdate();
    }
    catch (SQLException e) {
        System.out.println("error en la
                            actualización"+e.getMessage());
    }
}
```

Podéis ver cómo se crea el objeto *borra*, a través del cual se pasa el parámetro al procedimiento almacenado.

El fichero *jsp* que muestra la página que pide el tipo y que llama a este método es el que se muestra a continuación.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="datos" scope="session" class="paquetes.AccesoBdatos" />
<form >
    Tipo de objeto:<input type="text" name="tipos" />
    <input type="submit" value="enviar"/>
</form>
<%
datos.conectar();
String valor = request.getParameter("tipos");
if (valor != null) {
    datos.llamaProc2(valor);
}
%>
</body>
</html>
```

Aspecto de la página que pide el tipo



En la siguiente captura se pueden ver los objetos del esquema después de enviar el tipo VIEW como tipo de objetos que se quieren eliminar

```
SQL> select * from User_catalog;
TABLE_NAME                TABLE_TYPE
-----
DEPT                       TABLE
EMP                        TABLE
SQL>
```

10.2. CallableStatement. Procedimiento que devuelve valores.

Hemos visto cómo trabajar con procedimientos que recibían valores, en este caso (para parámetros IN) se llamaba al método `set<tipo>` en el caso de que se trabaje con parámetros OUT el método que los maneja es `registerOutParameter` y si se trabaja con los dos tipos de parámetros se debe llamar a los dos.

El siguiente método se utilizó para consultar un departamento, para ello utilizamos el objeto `PreparedStatement`.

```
public Depto consultarUno(int numero){
    try {
        PreparedStatement consulta = conecta.prepareStatement("SELECT * FROM
dept WHERE deptno=?");
        consulta.setInt(1, numero);
        ResultSet reg = consulta.executeQuery ();
        Depto departamento = new Depto();
        if (reg.next ())
        {
            departamento.setDeptno(reg.getInt (1));
            departamento.setDname(reg.getString(2));
            departamento.setLoc(reg.getString(3));
        }
        consulta.close ();
    }
}
```

```
        return departamento;
    }
    catch (SQLException e) {
        System.out.println("error en la consulta"+e.getMessage());
        return null;
    }
}
```

Ahora vamos sacar la consulta del método y pasarla a un procedimiento almacenado y que éste nos devuelva el nombre y la localidad del departamento cuyo código le pasamos.

El procedimiento almacenado es el siguiente:

```
create or replace procedure consultaDepar(num in dept.deptno%TYPE,
        name out dept.dname%TYPE, local out dept.loc%TYPE)is
begin
    select dname, loc into name,local from dept where dept.deptno=num;
    exception when no_data_found then name:='***'; local:='***';
end consultaDepar;
```

en el procedimiento se observa que tiene un parámetro de entrada **num**, y dos de salida: **name** y **local**.

El método que sustituye al anterior (consultarUno) es el siguiente:

```
public Depto consultarUnoProc(int numero){
    try {
        CallableStatement consulta = conecta.prepareCall("{Call
consultaDepar(?,?,?)}");
        consulta.setInt(1, numero);
        consulta.registerOutParameter(2, Types.VARCHAR);
        consulta.registerOutParameter(3, Types.VARCHAR);
        consulta.execute();
        Depto departamento = new Depto();
        departamento.deptno=numero;
        departamento.dname=consulta.getString(2);
        departamento.loc=consulta.getString(3);
        consulta.close ();
        return departamento;
    }
    catch (SQLException e) {
        System.out.println("error en la consulta"+e.getMessage());
        return null;
    }
}
```

El objeto CallableStatement trabaja con tres parámetros, uno tipo IN

```
setInt(1, numero)
```

y los otros dos tipo OUT

```
registerOutParameter(2, Types.VARCHAR)
```

```
registerOutParameter(3, Types.VARCHAR)
```

Como no devuelve un objeto tipo ResultSet (varias filas), no se puede utilizar el método executeQuery (), en este caso hay que utilizar execute ()

La página JSP y la clase no cambian

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" import="java.util.*,paquetes.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>página consultarUno que trabaja con el método del mismo nombre</title>
</head>
<body>
<jsp:useBean id="datos" scope="session" class="paquetes.AccesoBdatos" />
<jsp:useBean id="datosTabla" scope="session" class="paquetes.Depto" />
<form >
<input type="text" name="num" >
<input type="submit" value="enviar">
</form>
<%
datos.conectar();
String valor = request.getParameter("num");
if (valor != null)
{
    int numero = Integer.parseInt(valor);
    Depto departamentos = datos.consultarUnoProc(numero);
    if (departamentos != null)
    {
        %>
        <li> <%= departamentos.getDeptno () %></li>
        <li> <%= departamentos.getDname () %></li>
        <li> <%= departamentos.getLoc() %></li>
        <br><a>-----</a>
        %>
    }
}
%>
</body>
</html>
```

```
package paquetes;

public class Depto {
    Integer deptno;
    String dname;
    String loc;
    public Depto(Integer deptno, String dname, String loc) {
        super();
        this.deptno = deptno;
        this.dname = dname;
        this.loc = loc;
    }
    public Depto() {
    }
    public Integer getDeptno() {
        return deptno;
    }
    public void setDeptno(Integer deptno) {
        this.deptno = deptno;
    }
    public String getDname() {
        return dname;
    }
    public void setDname(String dname) {
        this.dname = dname;
    }
    public String getLoc() {
        return loc;
    }
    public void setLoc(String loc) {
        this.loc = loc;
    }
}
}
```

11. Métodos de conexión y acceso

En el recuadro se muestra el contenido de la clase **AccesoBdatos** en su conjunto, son todos los métodos que se describieron por separado.

```

package paquetes;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Vector;
public class AccesoBdatos {
    Connection conecta;
    public void conectar()
        throws SQLException, ClassNotFoundException {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        conecta=DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "rasty", "rasty");
    }
    //método para insertar sin utilizar un JavaBean
    public void insertar(Integer clave, String nombre, String localidad){
        try {
            String sql="insert into dept values (?, ?, ?)";
            PreparedStatement inserta = conecta.prepareStatement(sql);
            inserta.setInt(1, clave);
            inserta.setString(2, nombre);
            inserta.setString(3, localidad);
            inserta.executeUpdate();
        } catch (SQLException e){
            System.out.println("error al insertar en
                                dept"+e.getMessage());
        }
    }
    //método para insertar utilizando un JavaBean
    public void insertarConBean(Depto registro){
        try {
            String sql="insert into dept values (?, ?, ?)";
            PreparedStatement inserta = conecta.prepareStatement(sql);
            inserta.setInt(1, registro.getDeptno());
            inserta.setString(2, registro.getDname());
            inserta.setString(3, registro.getLoc());
            inserta.executeUpdate();
        } catch (SQLException e){
            System.out.println("error al insertar en dept"+e.getMessage());
        }
    }
}

```

```

    }
}
//método para consultar utilizando un JavaBean
public Collection consultarConBean(){
    ArrayList deptos = new ArrayList ();
    try {

        PreparedStatement consulta = conecta.prepareStatement("SELECT *
                                                                FROM dept ORDER BY 1");

        ResultSet reg = consulta.executeQuery ();
        while (reg.next ()) {
            Depto departamento = new Depto(
                reg.getInt (1), reg.getString (2), reg.getString (3));
            deptos.add(departamento);
        }
        consulta.close ();
    }
    catch (SQLException e) {
        return null;
    }
    return deptos;
}
//consultar uno
public Depto consultarUno(int numero){
    try {
        PreparedStatement consulta = conecta.prepareStatement("SELECT *
                                                                FROM dept WHERE deptno=?");

        consulta.setInt(1, numero);
        ResultSet reg = consulta.executeQuery ();
        Depto departamento = new Depto();
        if (reg.next ())
        {
            departamento.setDeptno(reg.getInt (1));
            departamento.setDname(reg.getString(2));
            departamento.setLoc(reg.getString(3));
        }
        consulta.close ();
        return departamento;
    }
    catch (SQLException e) {
        System.out.println("error en la consulta"+e.getMessage());
        return null;
    }
}
//actualizar
public void actualizar(Depto registro){
    try {
        String sql="update dept set dname = ?, loc = ? where deptno = ?";
        PreparedStatement actualiza = conecta.prepareStatement(sql);
        actualiza.setString(1,registro.getDname());
        actualiza.setString(2,registro.getLoc());
        actualiza.setInt(3,registro.getDeptno());
        actualiza.executeUpdate();
    }
}

```



```

    } catch(SQLException e){
        System.out.println("error al actualizar en dept"+e.getMessage());
    }
}
//bajas
public void bajas(int numero){
    try {
        PreparedStatement consulta = conecta.prepareStatement("DELETE FROM
                                                                dept WHERE deptno=?");

        consulta.setInt(1, numero);
        ResultSet reg = consulta.executeQuery ();
        consulta.close ();

    }
    catch (SQLException e) {
        System.out.println("error en la consulta"+e.getMessage());
    }
}

public int bajas2(int numero){
    int filas=0;
    try {
        PreparedStatement baja = conecta.prepareStatement("DELETE FROM
                                                         dept WHERE deptno=?");

        baja.setInt(1, numero);
        filas = baja.executeUpdate ();
        baja.close ();
        return filas;
    }
    catch (SQLException e) {
        System.out.println("error en la consulta"+e.getMessage());
        return 0;
    }
}

//acceso a procedimientos almacenados
public void llamaProc(Integer clave, String localidad){
    try {
        CallableStatement actu = conecta.prepareCall("{call
                                                         actualizaDept(?,?)}");

        actu.setInt(1,clave);
        actu.setString(2,localidad);
        actu.executeUpdate();
    }
    catch (SQLException e) {
        System.out.println("error en la
                            actualización"+e.getMessage());
    }
}

public void llamaProc2(String tipo){
    try {
        CallableStatement borra =conecta.prepareCall("{call borra_objetos(?)}");
        borra.setString(1,tipo);
        borra.executeUpdate();
    }
}

```

```
catch (SQLException e) {  
    System.out.println("error en la  
                        actualización"+e.getMessage());  
}  
}
```

Desarrollo de Aplicaciones Informáticas

materiales didácticos de aula



UNIÓN EUROPEA
Fondo Social Europeo



Gobierno del Principado de Asturias
Consejería de Educación y Ciencia



FORMACIÓN PROFESIONAL
Principado de Asturias