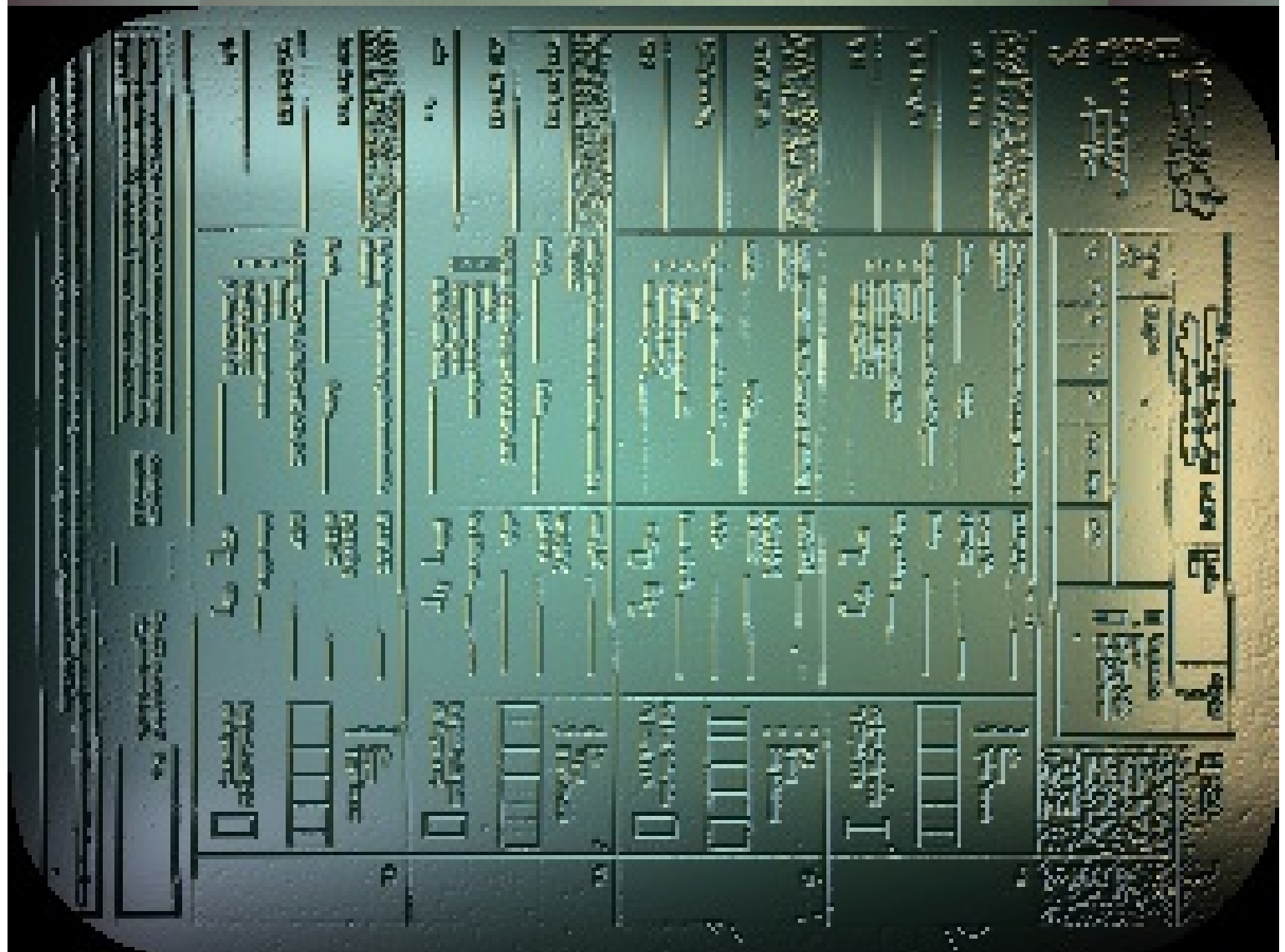


Java y Bases de Datos (Oracle). Una síntesis.

Sergio Gálvez Rojas
Miguel Ángel Mora Mata



JAVA Y BASES DE DATOS (ORACLE). UNA SÍNTESIS.
EDICIÓN ELECTRÓNICA

AUTOR: SERGIO GÁLVEZ ROJAS
 MIGUEL ÁNGEL MORA MATA

ILUSTRACIÓN
DE PORTADA: MIGUEL ÁNGEL MORA MATA

Sun, el logotipo de Sun, Sun Microsystems y Java son marcas o marcas registradas de Sun Microsystems Inc. en los EE.UU. y otros países.
Oracle es una marca registrada y Oracle9i/Oracle10g son marcas comerciales o marcas registradas de Oracle Corporation

Depósito Legal: MA 125-2010
I.S.B.N.: 978-84-693-0176-0

Java y Bases de Datos (Oracle)



Una síntesis

Sergio Gálvez Rojas
Doctor Ingeniero en Informática

Miguel Ángel Mora Mata
Ingeniero en Informática

Dpto. de Lenguajes y Ciencias de la Computación
E.T.S. de Ingeniería Informática
Universidad de Málaga



Java y Bases de Datos (Oracle).

Una síntesis.

Índice

Capítulo 1:

Introducción.....	<u>1</u>
Procedimientos almacenados.....	<u>2</u>
JDBC.....	<u>3</u>
SQLJ.....	<u>4</u>
BC4J.....	<u>5</u>
JDeveloper.....	<u>6</u>

Capítulo 2:

Procedimientos almacenados

en Java.....	<u>7</u>
Ejemplo.....	<u>8</u>
loadjava.....	<u>12</u>
Consideraciones sobre loadjava.....	<u>13</u>
Publicación.....	<u>14</u>
Correspondencia entre tipos.....	<u>15</u>
Tipos de parámetros.....	<u>16</u>
Creación de Tipos en Oracle.....	<u>17</u>
Otras invocaciones.....	<u>20</u>

Capítulo 3:

<i>Java Database Connectivity.....</i>	<u>21</u>
Cómo acceder a una base de datos.....	<u>22</u>
Drivers JDBC de Oracle.....	<u>23</u>
Ejemplo preliminar.....	<u>24</u>
JDBC 2.0.....	<u>25</u>
JDBC 3.0 y 4.0.....	<u>26</u>
Importación de paquetes y Carga del driver.....	<u>27</u>
Conexión a la base de datos.....	<u>28</u>
Consulta a la base de datos.....	<u>29</u>
Procesamiento del ResultSet.....	<u>30</u>
Cierres.....	<u>31</u>
Sentencias preparadas.....	<u>32</u>
Ejemplo de sentencia preparada.....	<u>33</u>
Acceso a la base de datos en procedimientos almacenados.....	<u>34</u>
Acceso a proc. almacenados.....	<u>35</u>
Ejemplo de llamada a DBMS_OUTPUT.....	<u>36</u>
Procesamiento básico de transacciones.....	<u>37</u>

Gestión de ResultSet.	38
ResultSet con <i>scroll</i>	39
ResultSet actualizable.	41
Metadatos de ResultSet.	43
Metadatos de la base de datos.	45
Gestión de <i>Large Objects</i>	46
Procesamiento <i>batch</i>	49
JDBC 4.0.	50
Anotaciones.	51
La interfaz BaseQuery.	52
Claves automáticas.	54
La interfaz DataSet.	55
Recorrido de excepciones SQL.	56
El tipo RowId.	57

Capítulo 4:

SQLJ.	58
Ejecución de un fichero SQLJ.	59
Declaraciones SQLJ.	60
Establecimiento de la conexión.	61
Establecimiento de la conexión.	62
Primer programa.	63
SELECT con resultado simple.	64
SELECT con resultado múltiple.	65
Iteradores <i>scrollable</i>	68
Ejemplo de iteradores <i>scrollable</i>	69
Ejemplo con <i>scrollable</i> anónimo.	70
Sensibilidad y Control de transacciones.	71
DDL y DML.	72
Cursores anidados.	74
Procedimientos PL/SQL.	76
SQLJ dentro de procedimientos PL/SQL.	77
VARRAY.	78
Consultas a VARRAY.	80
Inserción en VARRAY.	81

Capítulo 5:

JDeveloper.	82
Instalación de versiones antiguas.	83
Ejecución.	84
Filosofía estructural.	85
Ejecución de un proyecto.	86
Despliegue.	87
Caso práctico.	88
JPublisher.	94
Filosofía BC4J.	95
Capa de negocio.	96
Capa de manipulación.	97

Creación de una aplicación BC4J.	<u>98</u>
Módulo de aplicación BC4J.....	<u>99</u>
Test básico de un módulo de aplicación BC4J.	<u>100</u>
Creación de un cliente.	<u>101</u>
Ejecución de un proyecto.....	<u>102</u>
Despliegues de una aplicación BC4J.....	<u>103</u>
El servidor J2EE.	<u>104</u>
Despliegue en forma de <i>Enterprise JavaBeans</i>	<u>105</u>
Despliegue en JSP.	<u>107</u>
Propiedades de un objeto entidad.	<u>108</u>
Propiedades de un objeto vista.....	<u>109</u>

Prólogo

El presente texto constituye una síntesis completa que constituye el punto de partida para cualquier programador novel que desee aprender de manera rápida las distintas formas de intercomunicar un programa Java con una base de datos cualquiera.

Dado lo emocionante y extenso del tema, se ha decidido utilizar como Sistema Gestor de Bases de Datos a Oracle 9i, aunque los ejemplos propuestos funcionan igualmente (o con mínimas variaciones) con la versión 10g e incluso la 11i. Esta decisión se debe a que Oracle proporciona una de las bases de datos más completas y fáciles de instalar lo que permite comenzar en poco tiempo la codificación y ejecución de los distintos ejemplos que componen el curso.

Como complemento a las capacidades del sistema JDBC (*Java DataBase Connectivity*) también se proporcionan ejemplos que sólo tienen cabida en el entorno Oracle, como es la forma de incorporar procedimientos almacenados en Java o trabajar con tipos de datos creados por el usuario. Es más, el último capítulo se centra en el *framework* de desarrollo proporcionado por JDeveloper para realizar aplicaciones Java fácilmente mantenibles que accedan a Oracle.

Cada vez más la filosofía de Oracle se inclina por incorporar todo el código interno en lenguaje Java, en detrimento de PL/SQL (aunque no está previsto que éste vaya a desaparecer, ni mucho menos), ya que Java es un lenguaje ampliamente consolidado y reconocido por millones de programadores en todo el mundo y permite transferir fácilmente una base de datos de un sistema a otro, ya que también es multiplataforma. De hecho, el motor de la base de datos Oracle incorpora su propia máquina virtual específicamente adaptada a sus necesidades y con un rendimiento optimizado: la Oracle Java Virtual Machine (OJVM). Este hecho tiene, por contra, el inconveniente de que la OJVM suele ir un paso por detrás de las novedades incorporadas por la última versión de Java.

Confiamos en que el esfuerzo dedicado a la redacción de este documento sea de utilidad a todo aquel programador que desee introducirse de manera autodidacta en el fascinante mundo de Java y las Bases de Datos Relacionales.

Capítulo 1: Introducción

☆ Hay varias formas de interactuar con una base de datos:

☆ **Procedimiento almacenado**. Propio de cada SGBD (Sistema Gestor de Bases de Datos).

☆ **JDBC** (*Java Database Connectivity*). Estándar. Y ampliamente extendido.

☆ **SQLJ** (SQL para Java). Estándar pero no totalmente extendido.

☆ **BC4J** (*Business Components for Java*). No estándar pero de una gran versatilidad.

☆ Para todo esto utilizaremos **JDeveloper**, que es un entorno de desarrollo para Oracle. Pero antes de empezar con JDeveloper es bueno saber trabajar con las tecnologías correspondientes sin necesidad de entorno.

Procedimientos almacenados

☆ Un procedimiento almacenado en Java (PAJ) es un método Java convenientemente publicado y almacenado en la base de datos.

☆ Requiere una especificación de llamada que establece una correspondencia entre:

- ▶ Nombre de método Java y el nombre en SQL.
- ▶ Tipos de los parámetros en Java y los correspondientes en SQL.
- ▶ Tipo del valor retornado en Java y el que corresponda en SQL.

☆ Un **PROCEDURE** es una función Java que devuelve **void**.

☆ En general pueden ser invocadas de igual forma que un bloque PL/SQL (en Oracle).

☆ **Ventajas:**

- ▶ Potencia expresiva de la que carece PL/SQL
- ▶ Fuerte sistema de tipos
- ▶ Portabilidad
- ▶ Capacidad absoluta de interacción con el SGBD
- ▶ Se ejecutan con los permisos del que invoca
- ▶ Es ejecutado por una JVM independiente del SGBD

JDBC

- ★ JDBC (*Java Database Connectivity*) es el mecanismo estándar de acceso a bases de datos relacionales.
- ★ JDBC permite al programador conectarse con distintas bases de datos y conocer “al vuelo” sus diferencias con respecto al SQL estándar.
- ★ JDBC permite realizar consultas, actualizaciones, inserciones y borrados de registros.
- ★ JDBC permite la ejecución de sentencias SQL pre-escritas (entrecomilladas), por lo que es propenso a errores.
- ★ Las sentencias pueden ser simples o con variables de enlace (*bind*), lo que permite reutilizar una misma sentencia varias veces con distintos parámetros.
- ★ A través de JDBC también es posible invocar a procedimientos almacenados, así como a funciones.
- ★ También es posible conectar la salida estándar de Java con la salida de SQL*Plus.

SQLJ

★ SQLJ es un mecanismo para programar en JDBC sin utilizar cadenas entrecomilladas, por lo que no es tan propenso a errores en ejecución.

★ SQLJ no sólo permite ejecutar sentencias escritas en SQL nativo, sino que también permite gestionar los resultados de las consultas mediante un pseudolenguaje parecido al PL/SQL.

★ SQLJ es un estándar aún no soportado por la mayoría de SGBDR.

★ SQLJ consiste en escribir sentencias SQL dentro del propio código Java. Posteriormente, mediante un pre-compilador estas sentencias se traducen a JDBC puro.

★ Tanto JDBC como SQLJ permiten trabajar incluso con tipos definidos por el propio usuario.

BC4J

☆ BC4J (*Business Components For Java*) es una tecnología que establece un acceso multicapa a la base de datos.

☆ Una capa de negocio permite acceder a las tablas a través de objetos Java, así como validar la información de las tablas.

☆ Una capa de presentación permite suministrar los datos a las aplicaciones clientes. En esta capa se pueden definir datos calculados.

☆ Las capas clientes pueden acceder a los datos a través de las vistas de presentación y trabajar con ellos como se quiera.

☆ Los componentes BC4J se pueden depositar en la máquina, pero también se pueden depositar en contenedores J2EE remotos como OC4J (*Oracle Container For Java*).

☆ Esto permite gestionar los mismos BC4J con clientes de diversa índole: aplicaciones Java, JSP, formularios ADF, etc.

JDeveloper

★ JDeveloper es el entorno de desarrollo de Oracle para componentes BC4J, procedimientos almacenados, SQLJ, aplicaciones clientes, páginas JSP, etc.

★ Con JDeveloper resulta mucho más sencillo el desarrollo de aplicaciones, pero es un entorno que requiere un equipo potente.

★ Posee una gran cantidad de asistentes que facilitan la creación de componentes de cualquiera de las capas mencionadas: negocios, presentación y clientes.

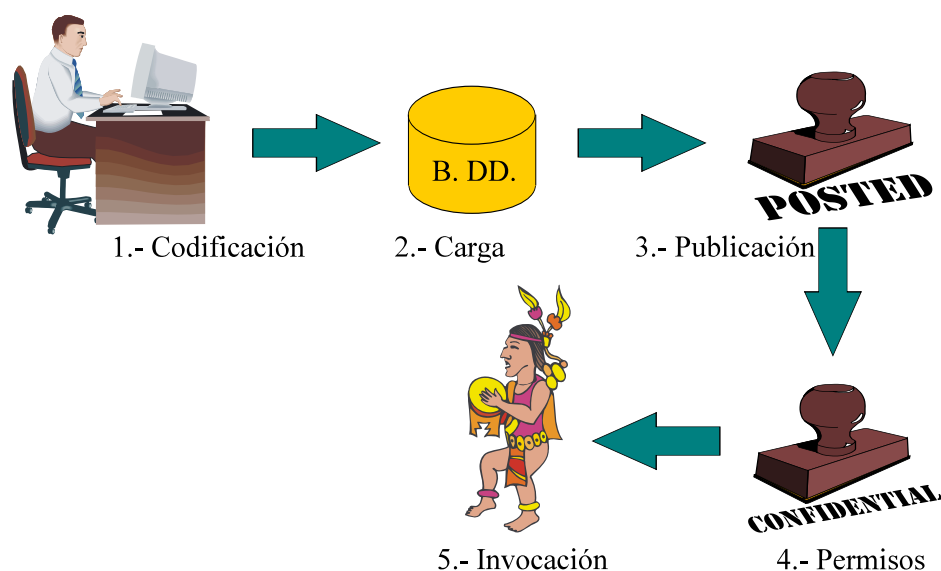
★ También permite trabajar con contenedores J2EE, y Oracle9iAS. Para J2EE incorpora su propio contenedor de pruebas OC4J, aunque en producción se aconseja usar Oracle9iAS.

★ Permite realizar diferentes despliegues de los mismos componentes, ya sean procedimientos almacenados, BC4J local o en contenedor J2EE, aplicaciones, etc.

Capítulo 2: Procedimientos almacenados en Java

★ Los pasos para almacenar un procedimiento Java en Oracle son:

- 1.- Codificar una clase Java con el método deseado.
- 2.- Cargar el método Java en el SGBD y resolver las referencias externas.
- 3.- Hacer público el método Java en el SGBD, mediante una especificación PL/SQL.
- 4.- Asignar los permisos necesarios para invocar al procedimiento.
- 5.- Invocar al procedimiento, bien desde SQL o desde PL/SQL.



Ejemplo. Paso 1: codificación

☆ Como ejemplo, se creará una función que calcula la longitud de una cadena SQL.

☆ Estática pues no se ejecuta en el contexto de un objeto:

```
// (c) Bulusu
public class SLength{
    public static int computeLength(String s){
        int retorno = s.length();
        return retorno;
    }
}
```

☆ Nótese que a PL/SQL no le es posible invocar directamente a `String.length()`; de ahí la necesidad de esta clase Java.

☆ Si es posible, es conveniente probar la clase Java antes de meterla en la base de datos.

```
public class Prueba{
    public static void main(String[] args){
        if (args.length == 0)
            System.out.println("Pásame un texto como
            parámetro");
        else
            System.out.println(SLength.computeLength(args[0]));
    }
}
```


Ejemplo. Paso 2: carga

☆ La utilidad **loadjava** permite meter en la base de datos:

- ▶ Fuentes .java
- ▶ Clases .class
- ▶ Ficheros .jar y .zip
- ▶ Ficheros de recursos Java .res
- ▶ Ficheros SQLJ .sqlj

```
loadjava -user SCOTT/TIGER  
         -oci8  
         -resolve  
         SLength.class
```

- user: Carga los ficheros como el usuario indicado.
- oci8: Se comunica con la base de datos a través del driver OCI.
- resolve: Resuelve las referencias externas

Ejemplo. Paso 3: publicación

★ La publicación construye una función o procedimiento PL/SQL asociándolo con el código Java.

★ PL/SQL hace homogéneo el acceso a los métodos mediante una única sintaxis, da igual que el método subyacente sea Java o PL/SQL.

```
CREATE OR REPLACE FUNCTION F_SLENGTH(s IN VARCHAR2)
RETURN NUMBER
AS LANGUAGE JAVA NAME
'SLength.computeLength(java.lang.String) return int';
```

★ A la función Java se le da un alias que será aquél por el que se la pueda invocar desde SQL o PL/SQL: **F_SLENGTH**.

★ Nótese que el tipo de los parámetros (y resultado) de la especificación Java debe estar calificado por completo (incluye el *package*).

★ La publicación la debe hacer quien haya incorporado la clase Java a la base de datos.

Ejemplo. Pasos 4 y 5: permisos e invocación

★ En este caso, si nos conectamos como SCOTT/TIGER no necesitaremos asignar permisos de acceso, ya que somos el propietario.

★ Ahora es posible invocar la función F_LENGTH de PL/SQL como si de cualquier otra se tratara:

```
select f_length('hola') from dual;
```

o bien

```
DECLARE
    RET_VAL NUMBER;
BEGIN
    RET_VAL := F_LENGTH('Ejemplo de prueba');
    dbms_output.put_line(to_char(ret_val));
END;
```

No olvidar el: `set serveroutput on`

loadjava

★ Algunos parámetros de **loadjava** son:

Parámetro	Descripción
-debug	Genera y visualiza información de depuración
-force	Carga las clases aunque ya hayan sido previamente cargadas
-grant	Les da permiso de EXECUTE a los usuarios indicados
-help	Visualiza la ayuda
-oci8	Se comunica con Oracle a través del driver OCI
-thin	Se comunica con Oracle a través del driver thin 100% Java
-resolve	Resuelve las referencias externas
-resolver	Permite indicar paquetes de otros esquemas con los que se debe realizar la resolución de referencias externas
-schema	Carga los ficheros en el esquema indicado. Por defecto es el de -user
-user	Se conecta a la base de datos con el usuario y clave indicados
-verbose	Visualiza información sobre la ejecución de loadjava

Consideraciones sobre loadjava

- ☆ Esta utilidad carga los ficheros en una tabla temporal con columnas BLOB y luego invoca a CREATE JAVA ...
- ☆ Una vez que todos los ficheros han sido cargados en la base de datos, loadjava lanza una sentencia ALTER JAVA CLASS ... RESOLVE para asegurarse de que las clases Java son completas, esto es, ninguna hace uso de una clase que no esté en el sistema.
- ☆ Es posible cargar un fichero fuente. En tal caso no se pueden cargar sus .class. Además, para recargar el fuente primero hay que descargarlo con **dropjava**.
- ☆ Es conveniente compilar fuera del SGBD y cargar las clases en la base de Datos.
- ☆ Es posible compilar dentro de Oracle: para ello se cargan sólo los fuentes que serán automáticamente compilados con la opción -resolve.
- ☆ Es posible dar opciones de compilación con el paquete DBMS_JAVA.
- ☆ Si hay errores de compilación, se almacenarán en USER_ERRORS.
- ☆ USER_OBJECTS guarda la información sobre los componentes Java cargados

Publicación

- ★ Establece una correspondencia entre la nomenclatura PL/SQL y la nomenclatura Java.
- ★ No supone una capa más en ejecución, por lo que es bastante eficiente.
- ★ Sólo se pueden publicar los métodos *public static*. No obstante los métodos públicos de instancia también se pueden publicar, pero sólo como miembros de un objeto SQL.
- ★ Se permite la sobrecarga.
- ★ Se deben tener en cuenta tres cosas:
 - ▶ Correspondencia entre tipos Java y SQL.
 - ▶ Tipos de parámetros: entrada, salida y ent/sal.
 - ▶ Privilegios de ejecución.
- ★ Si la función Java retorna `void` entonces se debe declarar un PROCEDURE PL/SQL en vez de una FUNCTION.
- ★ La declaración PL/SQL puede ser a nivel del sistema o a nivel de un paquete. En este último caso, la asociación entre PL/SQL y Java se hace en el PACKAGE BODY.
- ★ También es posible declarar un objeto SQL y añadirle métodos en Java

Correspondencia entre tipos

★ Es la siguiente:

SQL	Java
CHAR, NCHAR, LONG, VARCHAR2, NVARCHAR2	oracle.sql.CHAR java.lang.String java.lang.Byte java.lang.Short java.lang.Integer java.lang.Long java.lang.Float java.lang.Double java.math.BigDecimal java.sql.Date java.sql.Time byte, short, int, long, float, double
DATE	oracle.sql.DATE java.sql.Date java.sql.Time java.sql.Timestamp
NUMBER	oracle.sql.NUMBER java.lang.Byte java.lang.Short java.lang.Integer java.lang.Long java.lang.Float java.lang.Double java.math.BigDecimal byte, short, int, long, float, double
RAW, LONG RAW	oracle.SQL.RAW byte[]
ROWID	oracle.SQL.ROWID java.lang.String
Otros tipos	Propios de Oracle

Tipos de parámetros

- ★ Para poder pasar valores nulos no se pueden usar los tipos primitivos de Java: `int`, etc.
- ★ Los subtipos de SQL (`INTEGER`, `REAL` y `FLOAT`) carecen de correspondencia.
- ★ El tipo `boolean` de Java no tiene correspondiente en PL/SQL.
- ★ Por defecto los parámetros son todos de tipo `IN`, que se corresponde con el paso de parámetros por valor de Java.
- ★ Para pasar parámetros de forma `IN` y `OUT` se emplea un array de un solo elemento; p.ej. un parámetro `number` de `IN` `OUT` se corresponde en Java con `int io[]`;
- ★ Cada parámetro de la función Java se debe corresponder con uno de PL/SQL. La única excepción es una función `main` cuyo parámetro `String[]` se puede corresponder con varios parámetros PL/SQL.
- ★ Se recuerda que si la función Java no tiene parámetros, ni siquiera es necesario indicar la lista vacía en la declaración PL/SQL.

Creación de Tipos en Oracle

★ En Oracle es posible crear tipos de datos nuevos con `CREATE TYPE ... AS OBJECT`, así como asignarle métodos, ya sean de clase (`STATIC`) o de instancia (`MEMBER`).

★ En Oracle El objeto Oracle puede estar soportado realmente por un objeto Java. Para ello, la clase de Java debe implementar la interface `SQLData`, que es estándar (paquete `java.sql.*`), y posee las funciones:

- ▶ `getSQLTypeName`, que debe devolver el nombre con que se construyó el objeto Java con **`readSQL`**.
- ▶ `readSQL`, que permite construir el objeto Java a partir de uno en Oracle, al que se conecta un canal de entrada
- ▶ `writeSQL`, que permite cargar un objeto Oracle mediante un canal de salida.

que permiten que el programador establezca una correspondencia entre los componentes del objeto Java y del objeto Oracle.

★ Las funciones de la interfaz `SQLData` son invocadas automáticamente por JDBC durante su relación con el SGBD de Oracle.

★ Hay que tener cuidado con el paso de parámetros `NULL` a objetos, ya que ello exige que se haga por referencia.

Creación de Tipos en Oracle

★ El siguiente código se asociará más adelante a un tipo de Oracle.

```
import java.sql.*;
public class DireccionJefe implements SQLData{
    private int id;
    private String descripcion;

    public void crearDireccion(int id, String descripcion){
        this.id = id;
        this.descripcion = descripcion;
    }

    public void cambiarDireccion(String descripcion){
        this.descripcion = descripcion;
    }

    //Implementación de la interface.
    String SQLType;
    public String getSQLTypeName()
        throws SQLException{
        return SQLType;
    }
    public void readSQL(SQLInput canal, String typeName)
        throws SQLException{
        SQLType = typeName;
        id = canal.readInt();
        descripcion = new String(canal.readString());
    }
    public void writeSQL(SQLOutput canal)
        throws SQLException{
        canal.writeInt(id);
        canal.writeString(descripcion);
    }
}
```

Creación de Tipos en Oracle

★ Y ahora en Oracle se puede escribir algo como:

- Crea el tipo DIRECCION_JEFE:

```
CREATE OR REPLACE TYPE DIRECCION_JEFE AS OBJECT (  
    ID NUMBER(10),  
    DESCRIPCION VARCHAR2(45),  
    MEMBER PROCEDURE CREAR_DIRECCION(ID NUMBER, DESCRIPCION  
    VARCHAR2)  
    IS LANGUAGE JAVA NAME  
        'DireccionJefe.crearDireccion(int,  
java.lang.String)',  
    MEMBER PROCEDURE CAMBIAR_DIRECCION(DESCRIPCION VARCHAR2)  
    IS LANGUAGE JAVA NAME  
        'DireccionJefe.cambiarDireccion(java.lang.String)'  
);
```

- Crea una tabla con un campo de tipo DIRECCION_JEFE:

```
CREATE TABLE DIRECCIONES OF DIRECCION_JEFE;
```

- Mete un registro en la tabla anterior:

```
DECLARE  
    DIR DIRECCION_JEFE := DIRECCION_JEFE(null, 'hola');  
    texto varchar2(50);  
BEGIN  
    DIR.CREAR_DIRECCION(1, 'DIRE DEL JEFE');  
    INSERT INTO DIRECCIONES VALUES (DIR);  
END;
```

- Visualiza el contenido de la tabla anterior:

```
SELECT * FROM DIRECCIONES;
```

Otras invocaciones

★ También se puede invocar una función Java desde DML, con las siguientes restricciones:

- El método no puede tener parámetros OUT o IN-OUT.
- No puede modificar ninguna tabla.
- Tampoco puede consultar la tabla consultada o modificada.
- No puede ejecutar sentencias DDL, ni de transacción, ni de sesión ni ALTER SYSTEM.

★ También se puede invocar desde un trigger, con las mismas restricciones que cualquier procedimiento escrito en PL/SQL.

★ También se puede invocar desde un bloque PL/SQL de cualquier nivel, incluido el superior. Ej.:

```
VARIABLE lon NUMBER;  
CALL F_FLENGTH('Pablito clavó un clavito') INTO :lon;  
PRINT lon;
```

★ Más adelante estudiaremos cómo conectarnos a la base de datos actual:

```
Connection con =  
    OracleDriver().defaultConnection();
```

★ Más adelante veremos como efectuar llamadas a PL/SQL desde el interior de funciones Java.

Capítulo 3: *Java Database Connectivity*

★ JDBC proporciona una interfaz estándar de acceso a bases de datos relacionales.

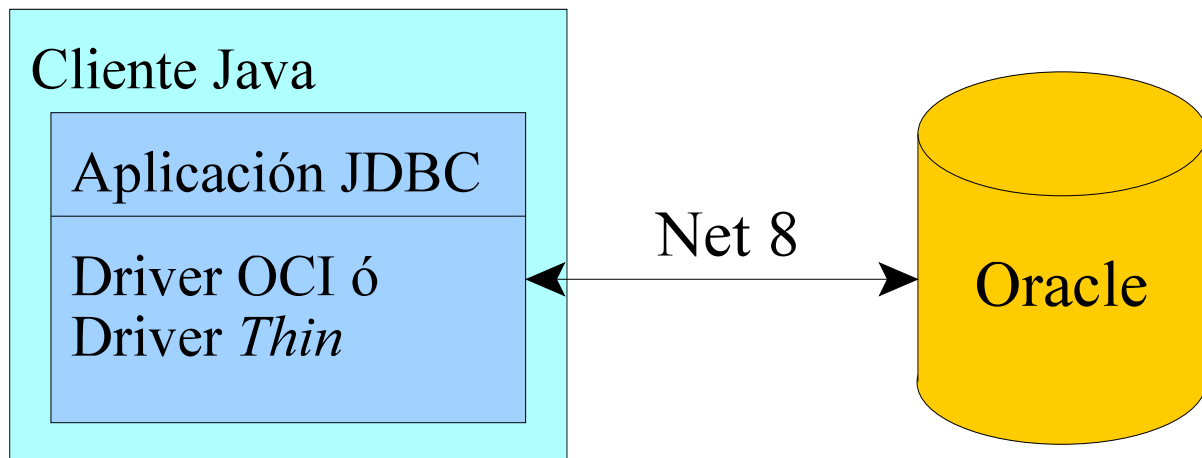
★ JDBC es independiente de dónde se encuentre el cliente y dónde esté el servidor.

★ JDBC consiste en una API de alto nivel y diferentes *drivers* cada uno para conectarse a una base de datos distinta.

★ Estos *drivers* pueden ser de cuatro tipos fundamentalmente:

- ▶ Tipo 1 (JDBC/ODBC): Necesita que la base de datos esté dada de alta en ODBC. Supone un puente de comunicaciones entre JDBC y ODBC.
- ▶ Tipo 2 (API Nativa): Es un tipo de driver totalmente propietario y que explota al máximo las características de una base de datos concreta.
- ▶ Tipo 3 (Protocolo de red abierto): A diferencia de las anteriores, permite accesos por red. Estos accesos usan protocolos independientes de los que proporcione la base de datos concreta a la que se conecta.
- ▶ Tipo 4 (Protocolo de red propietario): Igual que el anterior, pero usando protocolos propios del SGBD.

Cómo acceder a una base de datos



★ Un programa Java se conecta a una base de datos con JDBC realizando las siguientes operaciones:

1. Importación de paquetes
2. Carga del driver JDBC
3. Conexión con la base de datos
4. (Opcional) Averiguar las capacidades de la base de datos
5. (Opcional) Obtención de metadatos propios del esquema al que nos hemos conectado
6. Construcción de la sentencia SQL y ejecución
7. Procesamiento de resultados, si los hay
8. Cierre de la sentencia y del cursor, si lo hay
9. Cierre de la conexión

Drivers JDBC de Oracle

☆ Oracle proporciona los siguientes drivers de conexión:

Thin cliente: driver 100% Java, tipo 4. Da la máxima portabilidad, ya que simula TTC sobre TCP/IP.

OCI (Oracle Call Interface) cliente: driver nativo, tipo 2. Convierte las llamadas JDBC en llamadas OCI.

Thin servidor: igual que el Thin cliente pero interno a un servidor. Permite el acceso a bases de datos remotas. Este es el driver que se utiliza dentro de los procedimientos almacenados.

Interno al servidor: igual que el anterior pero sólo accede a la base de datos actual.

Ejemplo preliminar

★ Ejemplo con driver Thin y sentencia SELECT:

```
import java.sql.*;
public class EjemploConsulta{
    public static void main(String args[]){
        try{
            DriverManager.registerDriver(
                new oracle.jdbc.driver.OracleDriver());
            Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:oracle",
                "scott", "tiger");
            Statement stmt = conn.createStatement();
            ResultSet rset = stmt.executeQuery(
                "SELECT empno, ename, sal "+
                "FROM emp order by ename");
            while (rset.next()){
                System.out.println(
                    rset.getInt(1) + "-" +
                    rset.getString("ename") + "-" +
                    rset.getFloat("sal")
                );
            }
            rset.close();
            stmt.close();
            conn.close();
        }catch(SQLException x){
            x.printStackTrace();
        }
    }
}
```


JDBC 2.0

★ JDBC 1.0 permite accesos más o menos básicos a una base de datos.

★ JDBC 2.0 permite muchas más cosas:

- ▶ Los cursores (llamados **resultset**), se pueden recorrer de arriba abajo o viceversa, incluso cambiando el sentido.
- ▶ Se admite un mayor número de tipos de datos.
- ▶ Pool de conexiones. Para no estar conectándose y desconectándose continuamente (lo que consume recursos), los drivers OCI y Thin proporcionan un pool de conexiones permanentemente abiertas.
- ▶ Transacciones distribuidas. Paquetes `javax.sql.*` en el lado cliente. En el caso de Oracle, usar el paquete `oracle.jdbc.xa.client.*`.

★ El JDK 1.2 es compatible con JDBC 2.0 a través del paquete `java.sql.*`. Desde Oracle, lo que se usan son los paquetes `oracle.sql.*` y `oracle.jdbc.driver.*`, que se encuentran en `classes12.jar` del directorio `E:\oracle\ora92\jdbc\lib`, que hay que incluir en el `CLASSPATH`.

JDBC 3.0 y 4.0

★ JDBC 3.0 permite algunas cosas más:

- ▶ Al llamar a un procedimiento almacenado, los parámetros pueden pasarse por nombre en vez de por posición.
- ▶ Si una tabla tiene un campo autonumérico, permite averiguar qué valor se le ha dado tras una inserción.
- ▶ Se proporciona un pool de conexiones automático.
- ▶ Se permite el uso de SAVEPOINT.

★ JDBC 4.0 aún añade más cosas:

- ▶ No es necesario cargar el driver para realizar la conexión. El **DriverManager** lo hace de manera inteligente y automática.
- ▶ Soporta el tipo ROWID mediante la clase **RowId**.
- ▶ Facilita las sentencias SQL mediante el uso de anotaciones (**cancelado en la versión final de JSE 6.0**).
- ▶ Mejora en la gestión de excepciones: **SQLException** tiene nuevas subclases y las excepciones múltiples pueden recorrerse mediante un bucle.
- ▶ Soporte para caracteres nacionales.

Importación de paquetes y Carga del driver

☆ El paquete fundamental que debe cargarse es **java.sql.***

☆ Si va a hacerse uso de las características avanzadas de Oracle, habrá que importar también **oracle.jdbc.driver.*** y **oracle.sql.***

☆ La carga del driver se hace una sola vez en toda la aplicación. El mismo driver permite múltiples conexiones, incluso simultáneas.

☆ El driver es una clase java. En nuestro caso:
oracle.jdbc.driver.OracleDriver

☆ El driver puede cargarse de dos formas:

```
DriverManager.registerDriver(  
    new oracle.jdbc.driver.OracleDriver());
```

ó
Class.forName("oracle.jdbc.driver.OracleDriver");
pero se recomienda el 1^{er} método por ser más explícito y no propenso a errores en ejecución.

☆ Como puede verse, todos los drivers son gestionados por la clase **DriverManager**.

Conexión a la base de datos

★ Una conexión a la base de datos produce un objeto de tipo **Connection**, que se le pide al gestor de drivers (**DriverManager**), mediante el mensaje **getConnection()**, que toma tres parámetros de tipo String: cadena de conexión, usuario y contraseña.

★ La cadena de conexión tiene un formato muy concreto y propenso a errores ortográficos:

jdbc:oracle:oci8:@*sid*
jdbc:oracle:thin:@*host:puerto:sid*

★ Con el driver Thin no se usa la entrada TNSNAMES, por lo que hay que indicar el host y el puerto directamente, lo que también pueden especificarse, si se desea, con el driver **oci8**.

★ En un procedimiento almacenado, la conexión con la base de datos actual se consigue mediante:

```
Connection conn = new OracleDriver().defaultConnection();
```

★ La conexión también puede conseguirse sin DriverManager, de la forma:

```
ds = new oracle.jdbc.pool.OracleDataSource();
```

```
ds.setURL(myURL);
```

```
Connection conn = ds.getConnection(user, password);
```

método recomendado por Oracle.

Consulta a la base de datos

★ Toda sentencia está asociada a una conexión.

★ Por ello, las sentencias se le solicitan a las conexiones con:

conn.createStatement()

ó

conn.prepareStatement(TextoSQL)

que devuelven un objeto de tipo **Statement**.

★ Una sentencia no preparada se ejecuta con:

stmt.executeQuery(TextoSQL)

ó

stmt.executeUpdate(TextoSQL)

según se trate de una consulta o de un INSERT, UPDATE o DELETE.

★ Un **executeQuery** devuelve un cursor, implementado con un objeto de tipo **ResultSet**.

★ Las sentencias preparadas permiten utilizar valores de enlace (*bind*). Las veremos más adelante.

Procesamiento del ResultSet

☆ El procesamiento se divide en dos partes:

- 1.- Acceso a los registros obtenidos, de uno en uno.
- 2.- Obtención de los valores de las columnas, de una en una.

☆ El acceso a los registros se hace en base al método **next()** de los **ResultSet**., que devuelve un valor *booleano* que indica si hay o no siguiente registro.

☆ Las columnas pueden ser accedidas por nombre (**String**), o por posición (en caso de que haya campos calculados).

☆ Para ello se usa un método que depende del tipo de la columna obtenida: **getInt**, **getString**, **getFloat**, etc.

☆ Cada una de estas funciones toma como parámetro el nombre de la columna o bien la posición que ocupa.

```
while (rset.next()) {  
    System.out.println(  
        rset.getInt(1) + "-" +  
        rset.getString("ename") + "-" +  
        rset.getFloat("sal")  
    );  
}
```

Cierres

- ★ Una vez procesador un **ResultSet** hay que cerrarlo, así como la sentencia que lo originó.
- ★ Caso de no hacerlo se estarán ocupando cursores y bloques de memoria innecesarios, pudiendo producirse un error al superar el número máximo de cursores abiertos.
- ★ Una vez que se haya acabado todo el procesamiento con la base de datos, se debe cerrar la conexión.
- ★ La conexión se abre una sola vez al comienzo de la sesión y se cierra una vez acaba la sesión. Una misma sesión puede ejecutar múltiples sentencias.
- ★ Tanto los **ResultSet**, las **Statement** y las **Connection** se cierran con **close()**.
- ★ Por último, indicar que los métodos relacionados con SQL pueden lanzar la excepción **SQLException**, por lo que todo ha de enmarcarse en un try-catch.
- ★ A efectos de depuración es conveniente indicar **e.printStackTrace()** en la parte catch.

Sentencias preparadas

★ Una sentencia preparada es una sentencia procedente de un texto SQL con “huecos”.

★ Una sentencia preparada se obtiene mediante:

PreparedStatement stmt = conn.prepareStatement(TextoSQL);
donde el TextoSQL coloca un ? en cada lugar donde se espera un literal. Ej:

”INSERT INTO emp VALUES (?, ?, ?, ?, ?, ?, ?, ?)”

★ Cada hueco debe sustituirse por un valor literal.

★ Los huecos se referencian por posición. En función del tipo de datos que cada hueco vaya a albergar, se utiliza una determinada función para cargar su valor: **setInt**, **setString**, **setFloat**, etc. Ej.:

stmt.setInt(posición, valor)

★ Una vez cargados todos los “huecos” de una sentencia preparada, ésta se ejecuta mediante **executeQuery()**, si es un **SELECT**, o **executeUpdate()** si es un **INSERT**, **UPDATE** o **DELETE**.

★ Nótese que **executeQuery** y **executeUpdate** no tienen parámetros en las sentencias preparadas.

Ejemplo de sentencia preparada

★ A continuación se muestra el ejemplo anterior con sentencia preparada.

```
import java.sql.*;
public class EjemploConsultaPreparada{
    public static void main(String args[]){
        try{
            DriverManager.registerDriver(
                new oracle.jdbc.driver.OracleDriver());
            Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:oracle",
                "scott",
                "tiger");
            PreparedStatement stmt = conn.prepareStatement(
                "SELECT empno, ename, sal FROM emp " +
                "WHERE sal >= ? ORDER BY ename");
            double prueba[] = {1300.00, 2000.00, 3000.00};
            for(int i=0; i<prueba.length; i++){
                stmt.setDouble(1, prueba[i]);
                ResultSet rset = stmt.executeQuery();
                System.out.println("Probando con "+prueba[i]);
                while (rset.next()){
                    System.out.println(
                        rset.getInt(1) + "-" +
                        rset.getString("ename") + "-" +
                        rset.getFloat("sal")
                    );
                }
                rset.close();
            }
            stmt.close();
            conn.close();
        } catch(SQLException x){
            x.printStackTrace();
        }
    }
}
```

★ Nótese como una sentencia se puede reutilizar, pero cerrando los cursores que ya no hagan falta.

Acceso a la base de datos en procedimientos almacenados

☆ Hasta ahora hemos visto procedimientos almacenados que no acceden a la base de datos, lo cual no es de mucha utilidad.

☆ Un procedimiento almacenado accede a la base de datos actual cogiendo la conexión con

Connection conn = new OracleDriver().defaultConnection();
y sin que sea necesario registrar el driver. Tampoco es necesario cerrar la conexión. También se puede hacer:

Connection conn=

DriverManager.getConnection("jdbc:default:connection");

```
import java.sql.*;
import oracle.jdbc.driver.*;
public class ConsultaProcedimientoAlmacenado{
    public static int countConsulta(double sal)
                                throws SQLException {
        int retorno;
        Connection conn =
            new OracleDriver().defaultConnection();
        PreparedStatement stmt =
            conn.prepareStatement(
                "SELECT empno, ename, sal "+
                "FROM emp WHERE sal >= ? ORDER BY ename");
        stmt.setDouble(1, sal);
        ResultSet rset = stmt.executeQuery();
        retorno = 0;
        while (rset.next())
            retorno ++;
        rset.close();
        stmt.close();
        return retorno;
    }
}
```

```
c:\>loadjava -user scott/tiger@oracle -oci8 -resolve ConsultaProcedimientoAlmacenado.class
SQL>CREATE OR REPLACE FUNCTION CONSULTA_SUELDO(SAL NUMBER)
RETURN NUMBER IS LANGUAGE JAVA
NAME 'ConsultaProcedimientoAlmacenado.countConsulta(double) return int';
```

Acceso a proc. almacenados

★ Las llamadas a procedimientos almacenados son gestionadas por un tipo especial de sentencia: **CallableStatement**.

★ Se obtienen enviando a la conexión el mensaje **prepareCall**("{call nombre_proc(?, ?, ?)}")

★ Los parámetros de entrada se gestionan igual que en una **PreparedStatement**.

★ Los parámetros de salida deben registrarse antes de la llamada, con: **registerOutParameter(pos, tipo)**.

★ Los tipos de los parámetros se toman de la clase **java.sql.Types**

★ La invocación se hace con **executeUpdate()**.

★ Una vez ejecutada la llamada, los parámetros de salida se obtienen con **getTipo(pos)**, tal y como si se tratara de un **ResultSet** (pero sin **next()**).

★ Caso de llamar a una función, se hace:

prepareCall("{ ? = call nombre_proc(?, ?, ?)}")

donde el 1^{er} ? se trata como un parámetro de salida.

Ejemplo de llamada a DBMS_OUTPUT

☆ El siguiente ejemplo ilustra cómo visualizar datos por pantalla.

```
import java.sql.*;
import oracle.jdbc.driver.*;
public class ConsultaProcedimientoAlmacenado{
    public static void consulta(double sal)
                                throws SQLException {
        int retorno;
        Connection conn =
            new OracleDriver().defaultConnection();
        PreparedStatement stmt =
            conn.prepareStatement(
                "SELECT empno, ename, sal "+
                "FROM emp WHERE sal >= ? ORDER BY ename");
        stmt.setDouble(1, sal);
        ResultSet rset = stmt.executeQuery();
        CallableStatement cStmt = conn.prepareCall(
            "{call DBMS_OUTPUT.PUT_LINE("+
            "to_char(?)||' '||' '||to_char(?))}");
        while (rset.next()){
            cStmt.setInt(1, rset.getInt(1));
            cStmt.setString(2, rset.getString(2));
            cStmt.setDouble(3, rset.getDouble(3));
            cStmt.executeUpdate();
        }
        cStmt.close();
        rset.close();
        stmt.close();
    }
}
```

☆ Se puede conseguir que los **System.out.println** de Java se vean en la consola SQL, ejecutando en dicha consola: **CALL dbms_java.set_output(2000);**

Procesamiento básico de transacciones

☆ Por defecto, JDBC hace un COMMIT automático tras cada INSERT, UPDATE, o DELETE (excepto si se trata de un procedimiento almacenado).

☆ Este comportamiento se puede cambiar mediante el método **setAutoCommit(boolean)** de la clase **Connection**.

☆ Para hacer un COMMIT, la clase **Connection** posee el método **commit()**.

☆ Para hacer un ROLLBACK, la clase **Connection** posee el método **rollback()**.

☆ Si se cierra la conexión sin hacer **commit()** o **rollback()** explícitamente, se hace un COMMIT automático, aunque el AUTO COMMIT esté a false.

☆ También se permite el establecimiento/recuperación de/a puntos de salvaguarda con (propio de JDBC 3.0):
Savepoint setSavepoint(String nombre)

y
rollback(Savepoint sv)

Gestión de ResultSet

★ En JDBC 1.0 los ResultSet sólo se pueden recorrer de arriba abajo. Además, son de sólo lectura.

★ JDBC 2.0 permite que los ResultSet se puedan recorrer en ambos sentidos, permite posicionarse en un registro concreto e introduce posibilidades de actualización (conurrencia).

★ Un ResultSet puede ser de tres tipos:

ResultSet.TYPE_FORWARD_ONLY. Sólo se puede recorrer de arriba abajo. Es el tipo por defecto.

ResultSet.TYPE_SCROLL_INSENSITIVE. Se puede recorrer en cualquier sentido. Pero es insensible a los cambios efectuados por la misma transacción, o por otras transacciones finalizadas con éxito (COMMIT).

ResultSet.TYPE_SCROLL_SENSITIVE. Igual que el anterior, pero sensible a cambios.

★ Para obtener un ResultSet de alguno de estos tipos se especifica su tipo en los métodos:

`createStatement(int resultSetType, int resultSetConcurrency);`

`prepareStatement(String sql, int resultSetType, int resultSetConcurrency);`

`prepareCall(String sql, int resultSetType, int resultSetConcurrency);`

ResultSet con *scroll*

★ En un ResultSet con scroll se pueden usar los siguientes métodos (si el **ResultSet** es **TYPE_FORWARD_ONLY** se producirá una excepción SQL):

<code>boolean first()</code>	//Se pone al comienzo del ResultSet. Si éste está vacío retorna false , si no true .
<code>boolean last()</code>	// Igual que el anterior, pero al final.
<code>boolean previous()</code>	// Pasa al registro anterior al actual. Si no hay registro anterior devuelve false , si no true .
<code>boolean next()</code>	// Igual que el anterior, pero al posterior.
<code>boolean absolute(int rowNum)</code>	// Se va al registro indicado. Si se pasa, se queda en <code>beforeFirst</code> o <code>afterLast</code> . Devuelve true si el número de registro es válido.
<code>boolean relative(int offset)</code>	// Igual que el anterior pero de forma relativa.
<code>int getRow()</code>	// Devuelve el número de registro actual
<code>void beforeFirst()</code>	// Se posiciona antes del primer registro.
<code>boolean isBeforeFirst()</code>	// true si se está antes del primer registro.
<code>void afterLast()</code>	// Se posiciona tras el último registro.
<code>boolean isAfterLast()</code>	// true si se está tras el último registro.
<code>boolean isFirst()</code>	// true si se está en el primer registro.
<code>boolean isLast()</code>	// true si se está en el último registro.

Ejemplo de ResultSet con *scroll*

☆ Nótese cómo el siguiente ejemplo funciona incluso cambiando el 1300 por un 7300, de manera que el ResultSet generado no posea ningún registro.

```
import java.sql.*;
import java.sql.*;
public class ResultSetScrollable{
    public static void main(String args[]){
        try{
            DriverManager.registerDriver(
                new oracle.jdbc.driver.OracleDriver());
            Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:oracle",
                "scott",
                "tiger");
            Statement stmt = conn.createStatement(
                ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
            ResultSet rset = stmt.executeQuery(
                "SELECT empno, ename, sal FROM emp "+
                "WHERE sal >= 1300");
            rset.last();
            System.out.println(rset.getRow());
            rset.close();
            stmt.close();
            conn.close();
        }catch(SQLException x){
            x.printStackTrace();
        }
    }
}
```


ResultSet actualizable

★ Para indicar si un ResultSet se puede actualizar o no, se usan las constantes:

ResultSet.CONCUR_READ_ONLY. No permite acciones DML.

ResultSet.CONCUR_UPDATABLE. Permite acciones DML, tanto UPDATE como INSERT y DELETE.

★ Para hacer estas operaciones, la consulta debe ser considerada UPDATABLE por Oracle.

★ UPDATE. Una vez posicionado en el registro a modificar, ejecutar **updateTipo(pos, nuevoValor)**, donde pos representa la columna por posición (esto implica que especificar * en el SELECT no funcionará). Una vez actualizados todos los campos oportunos, ejecutar **updateRow()**. Las actualizaciones realizadas pueden cancelarse (si el AUTO COMMIT está a **false**), con **cancelRowUpdates()**.

★ INSERT. Ejecutar **moveToInsertRow()**, que se posiciona en un registro “fantasma”. A continuación, actualizar los campos como si fuera un UPDATE, y por último ejecutar un **insertRow()**.

★ DELETE. Una vez posicionados en el registro a borrar, ejecutar **deleteRow()**.

Ejemplo de ResultSet actualizable

★ Vamos a subirle el sueldo a todo el mundo en un 5% más 100€.

```
import java.sql.*;
public class SubidaSueldo{
    public static void main(String args[]){
        try{
            DriverManager.registerDriver(
                new oracle.jdbc.driver.OracleDriver());
            Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:oracle",
                "scott",
                "tiger");
            Statement stmt = conn.createStatement(
                ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_UPDATABLE);
            ResultSet rset = stmt.executeQuery(
                "SELECT empno, ename, sal FROM emp");
            while(rset.next()){
                rset.updateDouble("sal", rset.getDouble("sal")*1.5+100.0);
                rset.updateRow();
            }
            rset.close();
            stmt.close();
            conn.close();
        } catch(SQLException x){
            x.printStackTrace();
        }
    }
}
```

Metadatos de ResultSet

★ Cuando un **ResultSet** recoge datos de una consulta desconocida (p.ej., introducida por teclado), no se sabe qué campos obtener.

★ Para saber la estructura de una consulta se utiliza la clase **ResultSetMetaData**.

★ Sus funciones más interesantes son:

```
int getColumnCount();           // Número de campos por registro.  
String getColumnName(int i)     // Nombre de la columna i-ésima.  
Int getColumnType(int i)        // Tipo de la columna i-ésima.
```

★ Los tipos de las columnas vienen definidos por las constantes de `java.sql.Types`.

Ejemplo de Metadatos de ResultSet

★ Probar el siguiente ejemplo.

```
import java.sql.*;
public class SelectDesconocido{
    public static void main(String args[]){
        if (args.length == 0)
            System.out.println("Necesito un argumento.");
        else try{
            DriverManager.registerDriver(
                new oracle.jdbc.driver.OracleDriver());
            Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:oracle",
                "scott", "tiger");
            Statement stmt = conn.createStatement();
            ResultSet rset = stmt.executeQuery(args[0]);
            ResultSetMetaData rsetMD = rset.getMetaData();
            for(int i=1; i<=rsetMD.getColumnCount(); i++)
                System.out.print(rsetMD.getColumnName(i)+" ");
            System.out.println("");
            while(rset.next()){
                for(int i=1; i<=rsetMD.getColumnCount(); i++)
                    if (rsetMD.getColumnType(i) == Types.VARCHAR)
                        System.out.print(rset.getString(i)+" ");
                    else
                        System.out.print(""+rset.getDouble(i)+" ");
                System.out.println("");
            }
            rset.close();
            stmt.close();
            conn.close();
        } catch(SQLException x){
            x.printStackTrace();
        }
    }
}
```

Metadatos de la base de datos

- ☆ Los metadatos accesibles por la sesión actual pueden obtenerse a través de la conexión.
- ☆ Para ello se emplea el método **getMetaData()** que devuelve un objeto de tipo **DatabaseMetaData**.
- ☆ Este es uno de los objetos más complejos de todo Java.
- ☆ Por supuesto, es posible acceder a las tablas de metadatos de Oracle sin hacer uso de este objeto (como una consulta cualquiera), pero con ello se perdería portabilidad.
- ☆ El siguiente ejemplo muestra las tablas de las que es propietario el usuario actual.

```
DatabaseMetaData dbaseMD = conn.getMetaData();
ResultSet rset = dbaseMD.getTables(null, "SCOTT", null, null);
while(rset.next()){
    System.out.print(rset.getString("TABLE_CAT")+" ");
    System.out.print(rset.getString("TABLE_SCHEM")+" ");
    System.out.print(rset.getString("TABLE_NAME")+" ");
    System.out.println(rset.getString("TABLE_TYPE")+" ");
}
```

Gestión de *Large Objects*

★ Los “objetos grandes” que permite JDBC son los BLOBS y CLOBS. Los BFILE se gestionan mediante una extensión de Oracle.

★ Ejemplo de creación de tabla e inserción de datos:

```
CREATE TABLE lob_table(  
    id NUMBER PRIMARY KEY,  
    blob_data BLOB,  
    clob_data CLOB  
);
```

```
INSERT INTO lob_table VALUES (1, empty_blob(), empty_clob());
```

★ Realmente, lo que se almacena en la tabla no son los *bytes* que contienen los objetos grandes sino un localizador a los mismos.

★ Un objeto grande se gestiona obteniendo, a partir de su localizador, canales de entrada o salida para leerlo o escribirlo. Por tanto, se puede actualizar un objeto grande a través de un simple `SELECT...FOR UPDATE`. Evidentemente, las actualizaciones de un objeto grande no están sujetas a `COMMIT` y `ROLLBACK`.

★ Los localizadores no se pueden crear en JDBC, sólo en SQL.

★ Oracle no soporta **`java.sql.Blob`**, por lo que hay que usar **`oracle.sql.BLOB`**

★ Con **`java.sql.Blob`** se puede trabajar directamente con los bytes del BLOB, sin necesidad de canales de E/S.

Ejemplo de escritura de BLOB

★ El siguiente ejemplo mete el fichero que se indique como BLOB en la tabla anterior.

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.driver.*;
import oracle.sql.*;
public class MeteBLOB{
    public static void main(String args[]){
        if (args.length != 2) System.out.println("Necesito 2 argumentos.");
        else try{
            DriverManager.registerDriver(new OracleDriver());
            Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:oracle",
                "scott", "tiger");
            conn.setAutoCommit(false);
            Statement stmt = conn.createStatement();
            stmt.executeUpdate(
                "INSERT INTO lob_table VALUES (" +
                args[0] + ", empty_blob(), empty_clob())");
            ResultSet rset = stmt.executeQuery(
                "SELECT blob_data FROM lob_table where id = " +
                args[0] + " FOR UPDATE");
            if (rset.next()) try {
                BLOB blob_data = ((OracleResultSet)rset).getBLOB(1);
                int tamañoBuffer = blob_data.getBufferSize();
                OutputStream out = blob_data.getBinaryOutputStream();
                BufferedInputStream in = new BufferedInputStream(
                    new FileInputStream(args[1]),
                    tamañoBuffer
                );
                byte[] chunk = new byte[tamañoBuffer];
                while (true){
                    int numBytes = in.read(chunk, 0, tamañoBuffer);
                    if (numBytes == -1) break;
                    out.write(chunk, 0, numBytes);
                }
                in.close();
                out.close();
            } catch (IOException x) { x.printStackTrace(); }
            rset.close();
            stmt.close();
            conn.close();
        } catch (SQLException x) { x.printStackTrace(); }
    }
}
```

Ejemplo de lectura de BLOB

★ El siguiente ejemplo lee un BLOB y lo guarda en el fichero que se indique.

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.driver.*;
import oracle.sql.*;
public class SacarBLOB{
    public static void main(String args[]){
        if (args.length != 2)
            System.out.println("Necesito 2 argumentos.");
        else try{
            DriverManager.registerDriver(new OracleDriver());
            Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:oracle",
                "scott", "tiger");
            conn.setAutoCommit(false);
            Statement stmt = conn.createStatement();
            ResultSet rset = stmt.executeQuery(
                "SELECT blob_data FROM lob_table where id = "+args[0]);
            if (rset.next()) try {
                BLOB blob_data = ((OracleResultSet)rset).getBLOB(1);
                int tamañoBuffer = blob_data.getBufferSize();
                InputStream in = blob_data.getBinaryStream();
                BufferedOutputStream out = new BufferedOutputStream(
                    new FileOutputStream(args[1]),
                    tamañoBuffer
                );
                byte[] chunk = new byte[tamañoBuffer];
                while (true){
                    int numBytes = in.read(chunk, 0, tamañoBuffer);
                    if (numBytes == -1) break;
                    out.write(chunk, 0, numBytes);
                }
                in.close();
                out.close();
            } catch (IOException x){
                x.printStackTrace();
            }
            rset.close();
            stmt.close();
            conn.close();
        } catch (SQLException x){ x.printStackTrace();
        }
    }
}
```


Procesamiento *batch*

★ Es posible dejar pendientes una secuencia de sentencias y ejecutarlas todas de golpe como si fuera un *script* completo.

★ Esto sólo tiene sentido con operaciones idénticas que sólo varían en los valores de enlace. Deben ser de DML, y no SELECT. También se pueden invocar procedimientos almacenados que no tengan parámetros de salida.

★ Se supone que cada sentencia del proceso *batch* devuelve el número de registros que ha actualizado.

★ Para esto, en lugar de hacer **executeQuery()**, se hace **addBatch()**.

★ Para ejecutar el proceso batch, se le envía a la sentencia un **executeBatch()**.

★ Ejemplo:

```
conn.setAutoCommit(false);
PreparedStatement stmt = conn.prepareStatement(
    "INSERT INTO temp VALUES (?)");
stmt.setInt(1, 123);
stmt.addBatch();
stmt.setInt(1, 746);
stmt.addBatch();
stmt.setInt(1, 837);
stmt.addBatch();
stmt.setInt(1, 998);
stmt.addBatch();
stmt.setInt(1, 388);
stmt.addBatch();
int[] numRegs = stmt.executeBatch();
for(int i=0; i<numRegs.length; i++)
    System.out.println(numRegs[i]);
stmt.close();
```

JDBC 4.0

- ★ Java 6.0 incorpora JDBC 4.0, definida mediante la JSR 221.
- ★ El principal añadido de JDBC 4.0 es que permite hacer consultas y operaciones mediante anotaciones, lo que facilita enormemente el trabajo del desarrollador. Por desgracia, en el último momento, esta capacidad fue eliminada de la versión JDK 6.0 Final, lo que dejó con la miel en los labios a la comunidad Java. A pesar de ello veremos su funcionamiento sobre el papel.
- ★ Otra característica importante de JDBC 4.0 es que permite crear una conexión sin cargar explícitamente el driver con anterioridad, esto es, basta con hacer directamente:

```
DriverManager.registerDriver(new OracleDriver());  
Connection conn =  
    DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:oracle", "scott", "tiger");
```
- ★ Todos sabemos que cuando se produce un error en SQL, éste puede resultar críptico al deberse a una cadena de problemas acaecidos. Por ello en JDBC 4.0 la clase **SQLException** hereda de **Iterable**, lo que permite recorrer dicha cadena de fallos.
- ★ Por último, dado que el tipo ROWID se está haciendo cada vez más estándar en todas las bases de datos, JDBC 4.0 incorpora las clases necesarias para su gestión.

Anotaciones

★ Las anotaciones fueron introducidas en Java 5.0 y permiten especificar metadatos sobre variables, parámetros, campos, métodos, etc. El desarrollador puede definir sus propias anotaciones al igual que lo hace con clases e interfaces.

★ JDK 5.0 incluye varias anotaciones predefinidas; por ejemplo, para indicar que una método está obsoleto se codifica como:

```
@Deprecated  
public void unMetodo() {  
    ...  
}
```

★ Las anotaciones pueden ser procesadas por **apt.exe** (*Annotation Processing Tool*) e incluso puede tomar parámetros. **apt.exe** cuando procesa una anotación tiene acceso a los parámetros de la misma y al componente a que se aplica (clase, campo, variable, etc.), lo que le permite generar, por ejemplo, ficheros adicionales, modificar la clase en sí misma, etc.

★ Las anotaciones se han aplicado con éxito en la generación de *proxies* o *stubs* de acceso a servicios web: el desarrollador marca una clase como **@WebService** y algunos de sus métodos como **@WebMethod** y **apt.exe** se encarga de generar el WSDL correspondiente y la clase *stub* para invocarlo (desde Java).

★ Las versiones preliminares de JDK 6.0 introducían anotaciones para facilitar la construcción del patrón DAO al acceder a bases de datos con SQL. Éstas desaparecieron en la versión final.

La interfaz BaseQuery

★ La idea es englobar en una clase todas las operaciones funcionalmente relacionadas que deseen realizarse sobre una BD relacional.

```
import java.sql.BaseQuery;
import java.sql.DataSet;
import java.sql.Select;
public interface UserQueries extends BaseQuery {

    // Select all users
    @Select(sql = "SELECT userId, firstName, lastName FROM Users",
            readOnly=false, connected=false, tableName="Users")
    DataSet getAllUsers();

    // Select user by name */
    @Select(sql = "SELECT userId, firstName, lastName FROM Users "
            + "WHERE userName=?", readOnly=false, connected=false,
            tableName = "Users")
    DataSet getUserByName(String userName);

    // Delete user
    @Update("DELETE Users WHERE firstName={firstName} " +
            "AND lastName={lastName}")
    int deleteUser(String firstName, String lastName);
}
```

★ Aunque **BaseQuery** es una interfaz, no es necesario implementar explícitamente sus métodos, ya que las anotaciones lo hacen por nosotros.

★ Estas anotaciones (**@Select** y **@Update**) poseen diversos parámetros (readOnly, connected, scrollable, etc.) que permiten afinar su funcionamiento.

La interfaz BaseQuery

★ Los parámetros de la anotación **@Select** son:

Parámetro	Tipo	Propósito
sql ó value	String	Sentencia SQL a ejecutar
tableName	String	Nombre de la tabla a actualizar cuando se invoque al método DataSet.sync()
readOnly	boolean	Indica que el DataSet retornado es de sólo lectura
connected	boolean	Indica si el DataSet está conectado o no a la fuente de datos (<i>data source</i>)
allColumnsMapped	boolean	Indica si existe una biyección entre los nombres de las columnas de la sentencia SQL y los campos del DataSet retornado
scrollable	boolean	Indica si el DataSet retornado es <i>scrollable</i> o no. Este parámetro sólo se tiene en cuenta en modo conectado

★ Los parámetros de la anotación **@Update** son:

Parámetro	Tipo	Propósito
sql ó value	String	Sentencia SQL a ejecutar
keys	GeneratedKeys	Indica si se retornan o no las claves autogenerated. El valor por defecto es GeneratedKeys.NO_KEYS_RETURNED

Claves automáticas

★ **@AutoGeneratedKeys** se usa para insertar un registro nuevo en una tabla con campos autonuméricos. El objetivo es obtener los valores autogenerados por el SGBD en el momento de la inserción. Esta anotación es otra forma de invocar la función: **Statement.executeUpdate(java.lang.String, java.lang.String [])**

★ Por ejemplo:

```
@AutoGeneratedKeys
public class misClaves {
    public String col1;
    public String col2;
}
public interface MisConsultas{
    @Update(sql="insert into tabName(?1, ?2)",
        keys=GeneratedKeys.ALL_KEYS)
    DataSet<misClaves> añadirPersona(String nombre, int edad);
}
```

★ Cuando se invoca al método anotado **añadirPersona(...)**, las claves autogeneradas se almacenan en el **DataSet** que se devuelve.

★ Los parámetros de la anotación **@ResultColumn** son:

Elemento	Tipo	Propósito
name o value	String	Nombre de la columna de la sentencia SQL que se desea mapear en un campo de una clase
uniqueIdentifier	boolean	Indica si el campo es o no clave primaria

★ Un ejemplo de utilización puede ser:

```
public class User {
    @ResultColumn(name="userId") public String id;
    public String nombre;
    public String apellidos;
}
```

La interfaz DataSet

★ La interfaz **DataSet** hereda de **java.util.List** y puede operar en dos modos:

✎ *connected*: se comporta como un **ResultSet**

✎ *disconnected*: se comporta como un **CachedRowSet**

★ Un ejemplo completo de sus posibilidades puede ser:

```
public interface MisConsultas extends BaseQuery {
    @Select(sql = "SELECT userId, nombre, apellidos FROM Users",
        ReadOnly=false, connected=false, tableName="Users")
    DataSet<User> getUsers();
}
```

☞ Para comenzar a trabajar:

```
MisConsultas q = conn.createQueryObject(MisConsultas.class);
```

☞ Para obtener el **DataSet** y recorrerlo:

```
DataSet<User> dataSet = q.getUsers();
for (User user : dataSet) {
    System.out.println(user.getName());
}
```

☞ También es posible insertar, eliminar y modificar:

```
// Ejemplo de inserción
User newUser = new User("Federico", "Engels");
dataSet.insert(newUser);
// Ejemplo de modificación
for (User user : dataSet)
    if (user.getNombre().equals("Federico")) {
        user.setNombre("Friedrich");
        dataSet.modify();
    }
// Ejemplo de eliminación
for (User user : dataSet)
    if (user.getApellidos().equals("Engels")) {
        dataSet.delete();
    }
```

Recorrido de excepciones SQL

★ En JDBC 4.0 la clase **SQLException** implementa **java.lang.Iterable**. Esto quiere decir que es posible recorrer todos los objetos de tipo **Throwable** que haya en una excepción **SQLException**, de la siguiente manera:

```
try {  
    // do some data manipulation here  
} catch (SQLException e) {  
    for (Throwable t : e) {  
        System.out.println("Error: " + t);  
    }  
}
```

★ Además la clase **SQLException** posee 4 nuevos constructores:

- ▶ **public SQLException(java.lang.Throwable cause)**
- ▶ **public SQLException(java.lang.String reason,
 java.lang.Throwable cause)**
- ▶ **public SQLException(java.lang.String reason,
 java.lang.String sqlState,
 java.lang.Throwable cause)**
- ▶ **public SQLException(java.lang.String reason,
 java.lang.String sqlState,
 int vendorCode,
 java.lang.Throwable cause)**

El tipo RowId

★ JDBC 4.0 incorpora la interfaz **java.sql.RowId** para representar un identificador único para cada registro de un **ResultSet**, si es que lo soporta la conexión. Para comprobar esto último, la clase **DatabaseMetaData** dispone del método:

```
RowIdLifetime getRowIdLifetime() throws SQLException
```

donde **RowIdLifetime** es un enumerado que puede valer:

- ⇒ **ROWID_UNSUPPORTED**: la conexión no maneja ROWIDs.
- ⇒ **ROWID_VALID_FOREVER**: el objeto **RowId** de esta conexión es válido para siempre.
- ⇒ **ROWID_VALID_SESSION**: el objeto **RowId** de esta conexión es válido durante la conexión actual (sesión).
- ⇒ **ROWID_VALID_TRANSACTION**: el objeto **RowId** de esta conexión es válido, al menos, hasta que finalice la transacción actual.
- ⇒ **ROWID_VALID_OTHER**: indica que el **RowId** de la conexión es válido pero no se sabe por cuánto tiempo.

★ La interfaz **ResultSet** incluye dos métodos **getRowId**:

```
RowId getRowId(int columnIndex)
```

```
RowId getRowId(java.lang.String columnLabel)
```

★ Por ejemplo, ante un **SELECT** como:

```
SELECT ROWID, nombre FROM Users
```

se podría hacer un recorrido como:

```
while (rs.next()) {  
    System.out.println("Row Id: " + rs.getRowId(1));  
    System.out.println("Nombre: " + rs.getString(2));  
}
```

Capítulo 4: SQLJ

☆ SQLJ supone una interesante forma estándar de acceder a una base de datos.

☆ La ventaja sobre JDBC es que las sentencias SQL no se escriben como literales entrecomillados, sino que se precompilan, con lo que se evitan numerosos errores de ejecución.

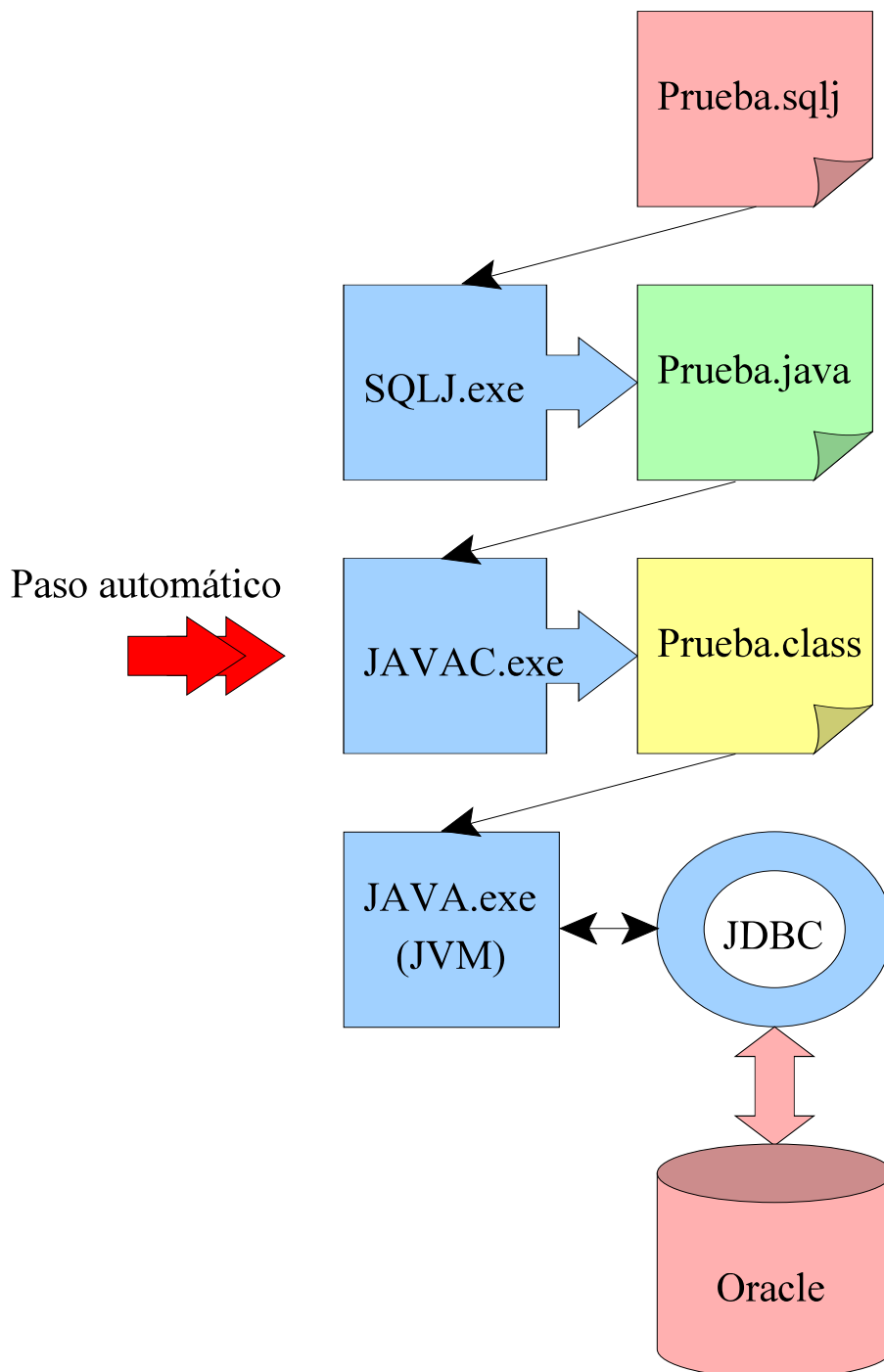
☆ Al ser precompilado es más rápido.

☆ La desventaja es que las sentencias SQL que se ejecutan deben ser estáticas (aunque pueden tener variables de enlace *-bind-*). No obstante, es posible simular un comportamiento dinámico.

☆ El proceso de trabajo con la base de datos es muy parecido al de JDBC: carga del *driver* y conexión, captura de excepciones y cierre de la conexión. Tan sólo varía un poco la recuperación de datos y la asignación de valores en sentencias preparadas.

Ejecución de un fichero SQLJ

★ Un fichero con extensión .sqlj es un fichero Java que tiene inmersas directivas SQLJ. El proceso para convertir un .sqlj en un .class es el siguiente:



Declaraciones SQLJ

★ SQLJ permite dos tipos de declaraciones: declaraciones puras y de variables Java.

★ Las declaraciones puras permiten declarar un **Iterator** (para recorrer el resultado de un SELECT) o bien la **Connection** por defecto con la que se van a hacer las operaciones (por si hay más de una conexión simultánea). Ejs.:

```
#sql public iterator EmpIter (int empno, int enmae, int sal);
```

ó

```
#sql public context miConnCtxt
```

★ Tras la directiva **#sql** se coloca la sentencia SQL a ejecutar entre { y }. Esta sentencia puede contener expresiones Java precedidas por dos puntos :. Ej.:

```
int hDeptno;
```

```
#sql {UPDATE emp SET sal=sal*1.5 WHERE deptno = :hDeptno};
```

ó

```
float hNuevoSal;
```

```
#sql {UPDATE emp SET sal=:(1.5*hNuevoSal) WHERE deptno =5};
```

★ También es posible ejecutar una función PL/SQL y guardar el resultado ejecutando:

```
int hSal;
```

```
#sql hSal = {VALUES (funcionPL_SQL(parámetros))};
```

Establecimiento de la conexión

★ Para realizar una conexión, es posible crear un fichero **connect.properties** que contiene la información de conexión. El contenido de este fichero puede ser:

```
# Ejemplo de fichero connect.properties  
sqlj.url=jdbc:oracle:thin:@localhost:1521:oracle  
sqlj.user=scott  
sqlj.password=tiger
```

★ Y la conexión se haría con:

```
Oracle.connect(getClass(), "connect.properties");
```

Nótese que no es necesario registrar el *driver*.

★ Para ello hay que importar la clase **oracle.sqlj.runtime.Oracle**

★ La conexión también puede hacerse de la forma:

```
Oracle.connect("jdbc:oracle:thin:@localhost:1521:oracle", "scott", "tiger");
```

★ Sea cual sea la forma en que se ha obtenido la conexión, ésta es la que se utilizará por defecto en cualquier sentencia SQLJ.

Establecimiento de la conexión

☆ Si nuestro programa realiza varias conexiones simultáneas, entonces hay que indicarle a las sentencias SQLJ cuál queremos utilizar.

☆ Lo primero que hay que hacer es almacenar las distintas conexiones en forma de objetos **DefaultContext**. Para ello utilizaremos **getConnection()** en lugar de **connect()** para conectarnos a la base de datos, ya que **getConnect()** devuelve un objeto de tipo **DefaultContext** que usaremos posteriormente para ejecutar la sentencia SQLJ.

☆ Desde Java se puede indicar la conexión por defecto ejecutando:

```
DefaultContext.setDefaultContext(contexto);
```

☆ En la propia directiva #sql se puede indicar el contexto de ejecución:

```
#sql contexto {sentencia SQL};
```

Primer programa

★ A continuación se muestra un sencillo programa que inserta un registro en la tabla EMP (no olvidar el COMMIT del final).

```
import java.sql.*;
import oracle.sqlj.runtime.*;
public class Primer{
    public static void main(String[] args)
        throws SQLException {
        if (args.length != 2)
            System.out.println("Necesito 2 argumentos.");
        else {
            Oracle.connect(Primer.class, "connect.properties");
            #sql {INSERT INTO EMP(EMPNO, ENAME)
                VALUES :(args[0]), :(args[1])};
            #sql {COMMIT};
            Oracle.close();
        }
    }
}
```

★ Este ejemplo hace uso del fichero connect.properties visto anteriormente.

★ Nótese cómo se cierra la conexión por defecto. En caso de disponer de un DefaultContext, deben cerrarse cada uno de ellos, con **close()**.

SELECT con resultado simple

★ Cuando una sentencia SELECT retorna una única tupla, los valores de ésta se pueden colocar en variables Java mediante la cláusula INTO. Si el SELECT no retorna nada o más de una fila, se produce un error.

```
import java.sql.*;
import oracle.sqlj.runtime.*;
public class SelectSimple{
    public static void main(String[] args)
        throws SQLException {
        if (args.length != 1)
            System.out.println("Necesito 2 argumentos.");
        else {
            Oracle.connect(SelectSimple.class, "connect.properties");
            String ename;
            double sal;
            #sql {SELECT ename, sal INTO :ename, :sal
                FROM emp WHERE empno = :(args[0])};
            System.out.println("Nombre: "+ename+". Salario: "+sal);
            Oracle.close();
        }
    }
}
```

★ Nótese cómo este ejemplo funciona aunque **args[0]** es de tipo **String** y **empno** es un **NUMBER**.

★ Nótese cómo no es necesario que la variable **ename** haya sido inicializada.

SELECT con resultado múltiple

★ Cuando el SELECT devuelve cero o más de una fila, debe asignarse a un iterador.

★ Existe una correspondencia unívoca entre cada tipo de iterador y cada tipo de tupla retornada por un SELECT. Un ejemplo de declaración de iterador sería:

```
#sql iterator EmpIter (int empno, String ename, double sal);
```

que será capaz de recorrer un ResultSet con tres campos de los tipos especificados, por ese orden.

★ Un iterador se asocia a una consulta de la forma:

```
EmpIter empRow = null;
```

```
#sql empRow = {SELECT empno, ename, sal FROM emp};
```

produciéndose una correspondencia de campos por nombre.

★ Un iterador posee el método **next()** (con idéntico comportamiento a un **ResultSet**) y funciones con los nombres de los campos con los que se declaró. Ej.:

```
while (empRow.next()){  
    System.out.print(empRow.empno()+"-");  
    System.out.print(empRow.ename()+"-");  
    System.out.println(empRow.sal()+"-");  
}
```

★ El iterador debe cerrarse con **close()**:

```
empRow.close();
```

Ejemplo de SELECT con resultado múltiple

```
import java.sql.*;
import oracle.sqlj.runtime.*;
public class SelectCompuesto{
    public static void main(String[] args) throws SQLException {
        Oracle.connect(SelectCompuesto.class, "connect.properties");
        #sql iterator EmpIter (int empno, String ename, double sal);
        EmpIter empRow = null;
        #sql empRow = {SELECT empno, ename, sal FROM emp};
        while (empRow.next()){
            System.out.print(empRow.empno()+"-");

            System.out.print(empRow.ename()+"-");

            System.out.println(empRow.sal()+"-");
        }
        empRow.close();
        Oracle.close();
    }
}
```

★ Nótese que si algún salario en NULL, esto da un error. Para evitar este problema, basta con sustituir el **double** por **Double**:

```
#sql iterator EmpIter (int empno, String ename, Double sal);
```

★ Si se desea gestionar la excepción **SQLException**, hay que importar el paquete **sqlj.runtime.***

Ejemplo de SELECT con resultado múltiple

☆ Si interesa obtener los registros resultantes de un SELECT en variables Java, se puede usar un iterador anónimo y la sentencia **FETCH...INTO**. Bastaría con poner el trozo de código:

```
#sql iterator EmpIter (int, String, Double);
EmpIter empRow = null;
int empno=0;
String ename=null;
Double sal=null;
#sql empRow = {SELECT empno, ename, sal FROM emp};
while (true){
    #sql {FETCH :empRow INTO :empno, :ename, :sal};
    if (empRow.endFetch()) break;
    System.out.print(empno+"-");
    System.out.print(ename+"-");
    System.out.println(sal+"-");
}
empRow.close();
```

☆ Una declaración de la forma:

```
#sql iterator EmpIter (int empno, String ename, Double sal);
```

crea una clase `EmpIter` que hereda de `sqlj.runtime.ResultSetIterator`, que incluye las funciones `empno()`, `ename()` y `sal()` para recuperar los datos. Pueden añadirse modificadores de clase:

```
#sql public static iterator EmpIter (int, String, Double);
```

Iteradores *scrollable*

★ Permiten recorrer aleatoriamente el resultado de un **SELECT**. Se declaran de la forma:

```
#sql iterator EmpIter implements sqlj.runtime.Scrollable (int empno,  
String ename, double sal);
```

★ Un iterador *scrollable* puede hacer uso de:

```
isBeforeFirst(), beforeFirst()
```

```
isFirst(), first(),
```

```
isAfterLast(), afterLast()
```

```
isLast(), last(),
```

```
next(),
```

```
previous(),
```

```
absolute(int),
```

```
relative(int),
```

al igual que un **ResultSet** *scrollable*. También puede usar **setFetchDirection(int)**, donde el parámetro se toma de una de las constantes de la clase **sqlj.runtime.ResultSetIterator**: **FETCH_FORWARD** y **FETCH_REVERSE**.

★ Si el **ResultSet** es anónimo se puede hacer uso de:

```
#sql { FETCH NEXT FROM :iter INTO :x, :y, :z };
```

```
#sql { FETCH PRIOR FROM :iter INTO :x, :y, :z };
```

```
#sql { FETCH FIRST FROM :iter INTO :x, :y, :z };
```

```
#sql { FETCH LAST FROM :iter INTO :x, :y, :z };
```

```
#sql { FETCH RELATIVE :(valor) FROM :iter INTO :x, :y, :z };
```

```
#sql { FETCH ABSOLUTE :(valor) FROM :iter INTO :x, :y, :z };
```

Ejemplo de iteradores *scrollable*

★ El siguiente ejemplo recorre los registros de atrás hacia adelante.

```
import java.sql.*;
import oracle.sqlj.runtime.*;
import sqlj.runtime.*;
public class SelectScrollable{
    public static void main(String[] args)
        throws SQLException {
        Oracle.connect(SelectScrollable.class, "connect.properties");
        #sql iterator EmpIter implements Scrollable
            (int empno, String ename, Double sal);
        EmpIter empRow = null;
        #sql empRow = {SELECT empno, ename, sal FROM emp};
        // La siguiente línea se puede quitar
        // Es sólo a efectos de eficiencia en el cursor creado
        empRow.setFetchDirection(
            ResultSetIterator.FETCH_REVERSE);
        empRow.afterLast();
        while (empRow.previous()){
            System.out.print(empRow.empno()+"-");

            System.out.print(empRow.ename()+"-");

            System.out.println(empRow.sal()+"-");
        }
        empRow.close();
        Oracle.close();
    }
}
```

Ejemplo con *scrollable* anónimo

★ El ejemplo anterior con *scrollable* anónimo (correspondencia de campos por posición) quedaría:

```
import java.sql.*;
import oracle.sqlj.runtime.*;
import sqlj.runtime.*;
public class SelectScrollableAnonimo{
    public static void main(String[] args) throws SQLException {
        Oracle.connect(SelectScrollableAnonimo.class, "connect.properties");
        #sql iterator EmpIter implements Scrollable
            (int, String, Double);
        EmpIter empRow = null;
        int empno=0;
        String ename=null;
        Double sal=null;
        #sql empRow = {SELECT empno, ename, sal FROM emp};
        // La siguiente línea se puede quitar
        // Es sólo a efectos de eficiencia en el cursor creado
        empRow.setFetchDirection(
            ResultSetIterator.FETCH_REVERSE);
        #sql {FETCH LAST FROM :empRow
            INTO :empno, :ename, :sal};
        while (!empRow.isBeforeFirst()){
            System.out.print(empno+"-");
            System.out.print(ename+"-");
            System.out.println(sal+"-");
            #sql {FETCH PRIOR FROM :empRow
                INTO :empno, :ename, :sal};
        }
        empRow.close();
        Oracle.close();
    }
}
```

Sensibilidad y Control de transacciones

☆ Por defecto, el resultado de un SELECT SQLJ es un iterador insensible a los cambios en los datos. Esto se puede cambiar declarando la clase del iterador de la forma:

```
#sql public static MyScrIter implements sqlj.runtime.Scrollable  
                                with (sensitivity=SENSITIVE)  
    (String ename, int empno);
```

☆ En SQLJ el auto commit está desactivado por defecto.

☆ Para activarlo es necesario obtener el contexto por defecto y, a partir de él, obtener un objeto **Connection** al que hacerle un **setAutoCommit(true)**:
cntxt.getConnection().setAutoCommit(true);

☆ Como ya se comentó, estando el auto commit a **false**, si se cierra el contexto sin hacer un COMMIT, entonces se hace un ROLLBACK de la última transacción.

DDL y DML

★ A continuación se muestra un ejemplo compuesto.

```
import java.sql.*;
import oracle.sqlj.runtime.*;
import sqlj.runtime.*;
public class DDLDML {
    public static void main(String[] args) throws SQLException
    {
        Object array[][]= {
            {new Integer(123), "Gustavo"},
            {new Integer(567), "Federico"},
            {new Integer(123), "Ismael"},
            {new Integer(186), "Iván"},
            {new Integer(633), "Tadeo"},
        };
        final int      ID = 0,
                    NOMBRE = 1;
        Oracle.connect(
            "jdbc:oracle:thin:@localhost:1521:oracle",
            "scott", "tiger");
        try {
            #sql { create table PRUEBA (
                        ID NUMBER(5) PRIMARY KEY,
                        NOMBRE VARCHAR2(35)) };
        } catch(SQLException x) {
            System.out.println("Tabla ya creada.");
        };
        for(int i=0; i<array.length; i++){
```



```
        int existe;
        #sql { select count(*) into :existe from PRUEBA
                where ID = :(array[i][ID]) };
        if (existe > 0) continue;
        #sql { insert into PRUEBA
                V A L U E S ( : ( a r r a y [ i ] [ I D ] ) ,
:(array[i][NOMBRE])) };
    }
    #sql { COMMIT };
    #sql iterator PruebaIt (int id, String nombre);
    PruebaIt pruebaRow=null;
    #sql pruebaRow = {select ID, NOMBRE from
PRUEBA};
    while (pruebaRow.next()){
        System.out.print(pruebaRow.id()+"-");

        System.out.println(pruebaRow.nombre()+"-");

    }
    pruebaRow.close();
    #sql {drop table PRUEBA};
    Oracle.close();
}
}
```

Cursores anidados

★ Como sabemos, una sentencia como:

```
SELECT nombre, CURSOR(  
    SELECT nombre, apellidos  
    FROM empleados  
    WHERE departamentos.codigo = empleados.codigo_departamento)  
FROM departamentos;  
produce un resultado maestro/detalle.
```

★ Esto puede conseguirse en SQLJ declarando un iterador para el detalle, y otro para el maestro. El iterador maestro contendrá un campo de tipo iterador detalle.

★ Para que el iterador maestro vea al detalle, los iteradores deben declararse fuera de la clase principal. Para ello se añaden los modificadores de clase convenientes: **private** y **public** donde convenga.

★ Nótese la cláusula AS para poder referencias al iterador detalle.

Ejemplo de cursores anidados

★ El ejemplo anterior quedaría:

```
import java.sql.*;
import oracle.sqlj.runtime.*;
public class MaestroDetalle{
#sql public static iterator EmpIter (String ename, Double sal);
#sql private static iterator DptIter (String dname, EmpIter empleados);
    public static void main(String[] args) throws SQLException {
        Oracle.connect(MaestroDetalle.class, "connect.properties");
        EmpIter empRow = null;
        DptIter dptRow = null;
        #sql dptRow = {      SELECT dname, CURSOR(
                                SELECT ename, sal
                                FROM emp
                                WHERE dept.deptno = emp.deptno)
                                AS empleados
                                FROM dept};
        while (dptRow.next()){
            System.out.println(dptRow.dname());
            empRow = dptRow.empleados();
            while (empRow.next()){
                System.out.print(empRow.ename()+"-");

                System.out.println(empRow.sal());
            }
            empRow.close();
        }
        dptRow.close();
        Oracle.close();
    }
}
```

Procedimientos PL/SQL

★ Un procedimiento PL/SQL se llama con CALL, de igual forma que desde PL/SQL. Ej.:

```
#sql {CALL DBMS_OUTPUT.PUT_LINE('Cursillo')};
```

★ Nótese cómo un PUT_LINE sólo funcionará en el caso de un programa SQLJ metido como procedimiento almacenado. Esto se verá más adelante.

★ Los parámetros OUT o IN OUT deben ser variables Java.

★ El resultado de una función se obtiene de la forma:

```
#sql hSal = {VALUES (funcionPL_SQL(parámetros))};
```

o bien:

```
#sql { SET :hSal = funcionPL_SQL(parámetros) };
```

★ La utilización de SET con respecto a VALUES tiene la ventaja de que permite evaluar expresiones completas (no sólo funciones simples), así como evaluar variables pre-construidas. Ej.:

```
import java.sql.*;
import oracle.sqlj.runtime.*;
public class LlamadaAFuncion{
    public static void main(String[] args) throws SQLException {
        Oracle.connect(LlamadaAFuncion.class, "connect.properties");
        Date fecha=null;
        #sql {SET :fecha = sysdate};
        System.out.println("La fecha de Oracles es " + fecha);
    }
}
```

SQLJ dentro de procedimientos PL/SQL

☆ El SQLJ puede ejecutarse dentro de una base de datos como cualquier procedimiento o función almacenada no SQLJ.

☆ La única diferencia es que no hay que crear ningún tipo de contexto ni conexión. Se trabaja directamente con la sesión del que ejecuta el procedimiento almacenado. Ej.:

```
import java.sql.*;
import oracle.sqlj.runtime.*;
public class PrimerProcedimientoAlmacenado{
    public static void imprimir() throws SQLException {
        #sql {CALL DBMS_OUTPUT.ENABLE(2000)};
        #sql {CALL DBMS_OUTPUT.PUT_LINE('Cursillo')};
    }
}
```

```
c:\>loadjava -user scott/tiger@oracle -oci8 -resolve
        PrimerProcedimientoAlmacenado*.class
```

```
SQL> CREATE OR REPLACE PROCEDURE IMPRIMIR AS
        LANGUAGE JAVA NAME
        'PrimerProcedimientoAlmacenado.imprimir()';
```

☆ Nótese la necesidad de hacer un SET SERVEROUTPUT ON para ver el texto en pantalla.

VARRAY

★ SQL permite trabajar con arrays de longitud limitada pero indefinida:

```
CREATE TYPE T_DATOS AS VARRAY(10) OF VARCHAR2(35);
```

★ Se pueden crear campos de este tipo de la forma:

```
CREATE TABLE INFO(  
ID NUMBER,  
DATOS T_DATOS);
```

★ Las inserciones serían:

```
INSERT INTO INFO  
VALUES(1, T_DATOS('UNO', 'DOS', 'TRES', 'CUATRO'));
```

★ Y la selección, tan sencilla como:

```
SELECT * FROM INFO;
```

VARRAY

★ Para trabajar con un VARRAY con SQLJ hay que crear una clase *wrapper*. Lo mejor es hacerlo con la utilidad **jpub.exe** (Jpublisher). Ej.:

```
jpub -sql=T_DATOS -user=scott/tiger@oracle
```

que genera T_DATOS.java

★ El fichero Java generado posee las funciones:

Constructor sin parámetros.

getArray() que en nuestro caso devuelve String[]

setArray(String[])

length()

getElement(int) devuelve el elemento indicado: String

setElement(String, int)

Todas las operaciones pueden generar SQLException.

★ En general, **jpub** permite crear clases Java wrapper de cualquier tipo definido en Oracle. Esto se verá más adelante, al estudiar Jdeveloper.

★ **jpub** posee los modificadores:

-usertypes

-numbertypes

-lobtypes

-builtintypes

que pueden valer **oracle** ó **jdbc** según se quiera que las clases generadas hereden en base a la biblioteca estándar (JDBC) o a la deOracle.

Consultas a VARRAY

★ Un ejemplo de consulta sería (una vez generado el *wrapper*):

```
import java.sql.*;
import oracle.sqlj.runtime.*;
public class ConsultaVARRAY {
    public static void main(String[] args)
        throws SQLException {
        Oracle.connect(Primer.class, "connect.properties");
        #sql iterator InfoItClass (Integer id, T_DATOS datos);
        InfoItClass infoIt=null;
        #sql infoIt = {SELECT * FROM INFO};
        while(infoIt.next()){
            System.out.println(infoIt.id());
            for(int i=0;i<infoIt.datos().length(); i++)
                System.out.println("\t"+infoIt.datos().getElement(i));
        }
    }
}
```

★ Mucho cuidado porque para que esto funcione es necesario incluir en el CLASSPATH la biblioteca \oracle\ora92\jdbc\lib\nls_charset12.jar. Si no se pone esa librería, los String salen en Hexadecimal debido a las conversiones de los juegos de caracteres nacionales.

Inserción en VARRAY

★ Un ejemplo de inserción sería:

```
import java.sql.*;
import oracle.sqlj.runtime.*;
public class InsercionVARRAY{
    public static void main(String[] args)
        throws SQLException {
        Oracle.connect(Primer.class, "connect.properties");
        T_DATOS datos = new T_DATOS();
        datos.setArray(new String[]{"Alfa", "Beta", "Gamma"});
        #sql {INSERT INTO INFO VALUES (2, :datos)};
        #sql {COMMIT};
    }
}
```

★ Nótese la existencia de un constructor sin parámetros de la clase T_DATOS.

Capítulo 5: JDeveloper

- ★ JDeveloper es un potente entorno gráfico que facilita el trabajo con Java y Oracle.
- ★ Permite construir programas SQLJ y guardarlos como procedimientos almacenados, con poco esfuerzo.
- ★ Permite la construcción de componentes BC4J que son algo así como envolturas a los componentes relacionales de la base de datos.
- ★ Permite utilizar estos componentes en distintos *frameworks*: OAS, J2EE, JSP, etc.
- ★ Un *framework* es una filosofía de construcción de programas Java que suele acceder a una base de datos.
- ★ Un *framework* se apoya en componentes propietarios, por lo que es conveniente hacer uso de herramientas visuales que los construyan automáticamente.

Instalación de versiones antiguas

★ JDeveloper se puede descomprimir (en esto consiste la instalación) en cualquier directorio que no sea ORACLE_HOME.

★ El fichero básico de configuración es:

`<jdev_install>\jdev\bin\jdev.conf`

★ Hay que asegurarse de que este fichero contiene dos entradas:

- ▶ Una que indica donde está el JDK. Ej.:
`SetJavaHome e:\j2sdk1.4.2`
- ▶ Y otra que indica qué JVM se va a usar. Es conveniente hacer uso de la JVM de Oracle (OJVM) que está optimizada para Jdeveloper. Ej.:

`SetJavaVM ojvm`

pero también se puede hacer:

`SetJavaVM hotspot`

★ Para hacer uso de la OJVM hay que configurar su acceso, indicándole dónde está el JDK:

`<jdev_install>\jdev\bin\InstallOJVM.bat e:\j2sdk1.4.2`

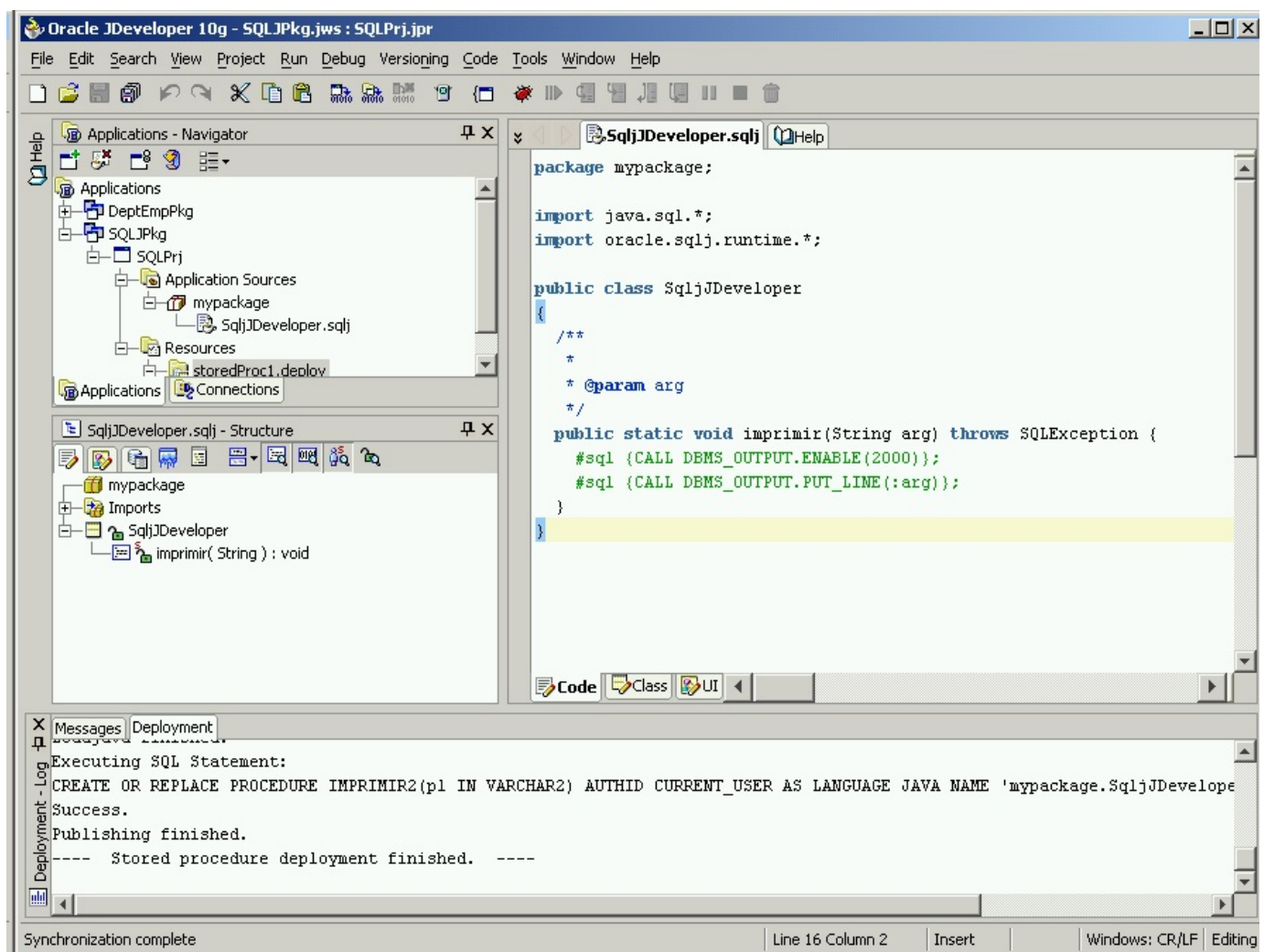
Ejecución

★ El fichero de ejecución de JDeveloper para Windows es:

<jdev_install>\jdev\bin\jdevw.exe



★ Una vez conectados a JDeveloper, éste tiene la siguiente apariencia:



Filosofía estructural

★ JDeveloper estructura el trabajo en aplicaciones completas, llamadas **workspaces**.

★ Un workspace puede estar formado por uno o más proyectos. Un proyecto es simplemente un conjunto de ficheros relacionados para un propósito concreto dentro de una aplicación.

★ Básicamente hay dos tipos de workspace: **Java** y **web**. Nosotros nos centraremos en los Java.

★ Una vez creado un workspace, es posible crear en su interior tantos proyectos como se quiera. Trabajaremos con dos tipos de proyectos: **Business Components** y **Java Application**.

★ Finalmente, a un proyecto se le añadirán los ficheros concretos que lo componen.

★ JDeveloper incorpora asistentes para la creación de numerosos componentes, aunque éstos también pueden construirse manualmente.

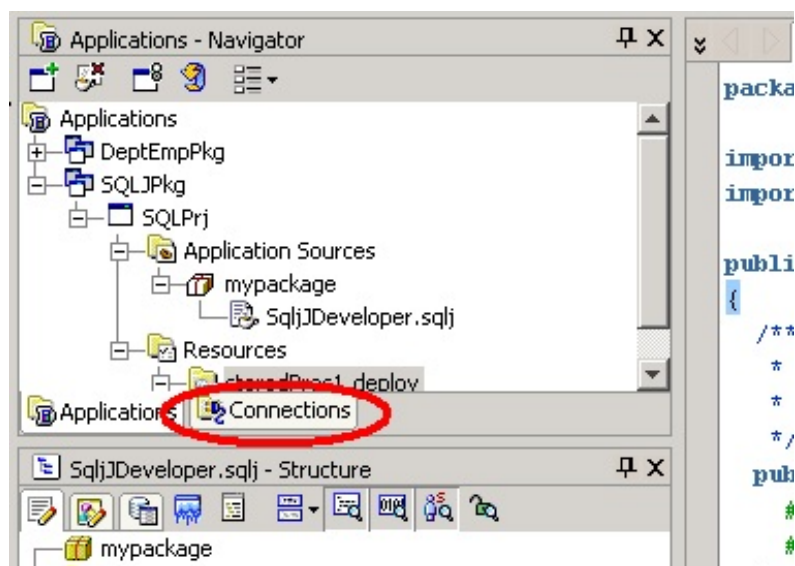
Ejecución de un proyecto

☆ Para ejecutar un proyecto es necesario desplegarlo en un destino (*deployment*).

☆ Esto es así, porque a estos niveles, la ejecución no tiene porqué ser en local, sino que puede ser remota haciendo uso de distintos esquemas de ejecución: J2EE, OAS, JSP, local, procedimientos almacenados, etc.

☆ Sea cual sea el destino, hay que establecer una conexión a él. Lo normal es crear, al menos, una conexión a la base de datos Oracle con la que se va a trabajar.

☆ Para ello, el panel de navegación tiene una pestaña **Connections** donde es posible crear los distintos tipos de conexiones.



Despliegue

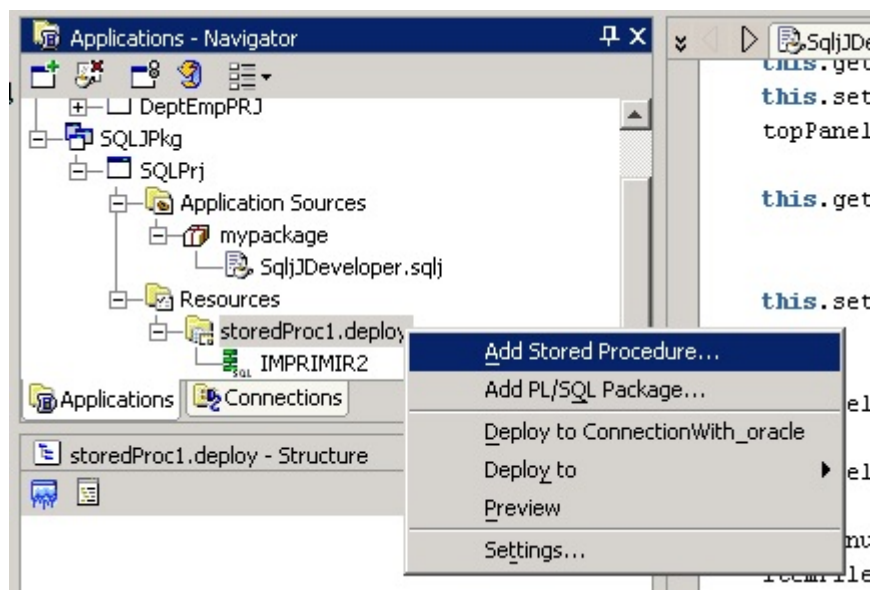
★ Si, por ejemplo, se va a crear un procedimiento almacenado, lo primero que hay que hacer es crear una conexión que acceda al esquema que los va albergar.

★ A continuación hay que seleccionar en el proyecto, la creación de un **New -> Deployment profile**. A continuación escogeríamos la opción **Loadjava...**

★ Por último, seleccionamos los ficheros a depositar.

★ Una vez finalizada esta fase, en el perfil de despliegue creado hay que seleccionar qué métodos se quieren cargar y con que *wrapper* PL/SQL. El *wrapper* lo crea automáticamente Jdeveloper.

★ Finalmente, se haría el despliegue con la conexión que se quiera.



Caso práctico

★ Vamos a colocar el siguiente código SQLJ dentro del esquema de scott.

```
import java.sql.*;
import oracle.sqlj.runtime.*;

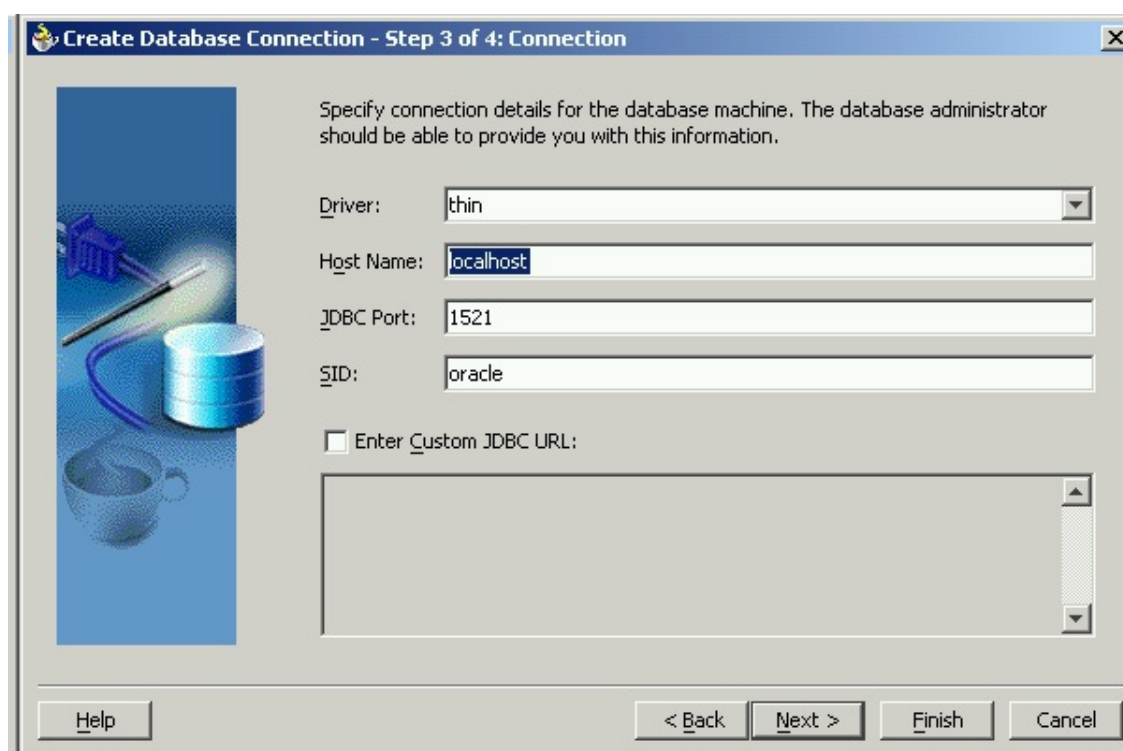
public class PrimerSqljJDeveloper {
    public static void imprimir(String arg) throws SQLException {
        #sql {CALL DBMS_OUTPUT.ENABLE(2000)};
        #sql {CALL DBMS_OUTPUT.PUT_LINE(:arg)};
    }
}
```

★ Los pasos exactos que hay que seguir son:

- 1.- Crear una conexión
- 2.- Crear un área de trabajo.
- 3.- Crear un proyecto.
- 4.- Indicar las librerías del proyecto.
- 5.- Añadir un fichero SQLJ.
- 6.- Crear un perfil de despliegue.
- 7.- Configurar los procedimientos PL/SQL.
- 8.- Ejecutar el despliegue.
- 9.- Comprobar el procedimiento desde SQL*Plus o SQL*Worksheet.

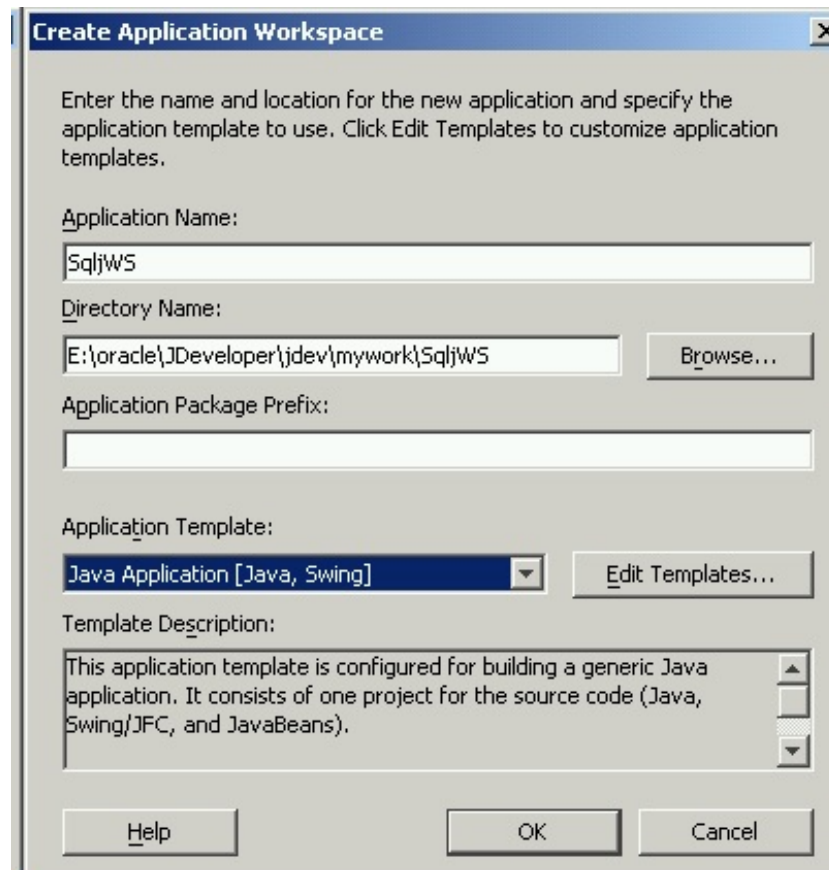
Pasos (I)

- ★ Para crear la conexión, pulsar la pestaña Connection del Navigator. Database->Botón derecho->New Connection.
- ★ La conexión debe ser Oracle(JDBC) y el nombre: OracleJDBC_*SID*.
- ★ Ahora hay que escoger el esquema de usuario, el tipo de driver a usar (Thin u OCI8), así como la base de datos a la que conectarse.
- ★ Comprobar que la conexión es satisfactoria. Si no, comprobar nombre de usuario, contraseña y sid.



Pasos (II)

☆ Para crear el área de trabajo, pulsar Applications -> Botón derecho->New Application Workspace.



☆ Automáticamente se ha creado el proyecto Client. Renombrar Client como SqljPRJ: Seleccionar Client. File -> Rename

☆ Antes de añadir el fuente SQLJ es necesario ajustar el CLASSPATH del proyecto.

Pasos (III)

★ Para añadir librerías a SqljPRJ: Seleccionar SqljPRJ -> Botón derecho -> Project properties -> Profiles -> Development -> Libraries.

★ Aquí crearemos dos librería nuevas:

GeneralJDBC que contendrá los JAR:

<ORAHOME>\jdbc\lib\classes12.jar

<ORAHOME>\jdbc\lib\ojdbc.jar

<ORAHOME>\jdbc\lib\nls_char12.jar

GeneralSQL que contendrá los JAR:

<ORAHOME>\sqlj\lib\translator.jar

<ORAHOME>\sqlj\lib\runtime12.jar

★ Estas librerías deberán estar añadidas a la lista de seleccionadas, junto con JDeveloper Runtime.

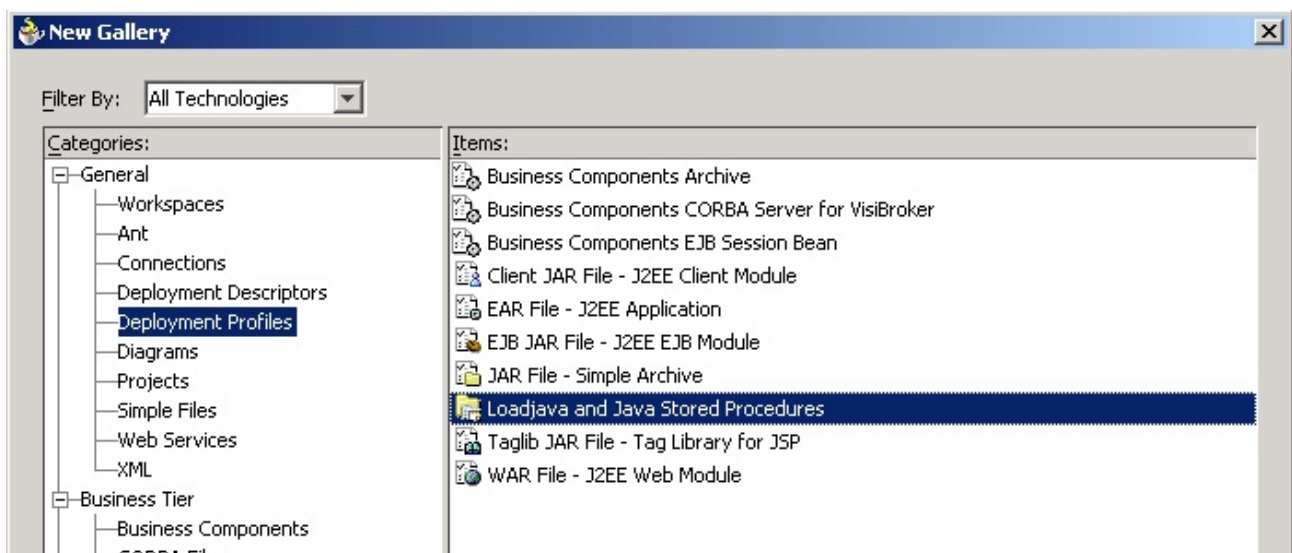
★ Para añadir ficheros a SqljPRJ: Seleccionar SqljPRJ -> Botón derecho -> New -> General -> Simple Files -> File.

★ Éste será el fichero SQLJ mostrado anteriormente. Su nombre debe coincidir con el de la clase que contiene, y debe tener extensión **sqlj**.

Pasos (IV)

★ Una vez escrito el código del fichero SQLJ, hay que crear el perfil de despliegue: Seleccionar SqljPRJ -> Botón derecho -> New -> General -> Deployment Profiles -> Loadjava and Java Stored Procedures.

★ Si no aparece Loadjava..., seleccionar Filter by: All technologies.



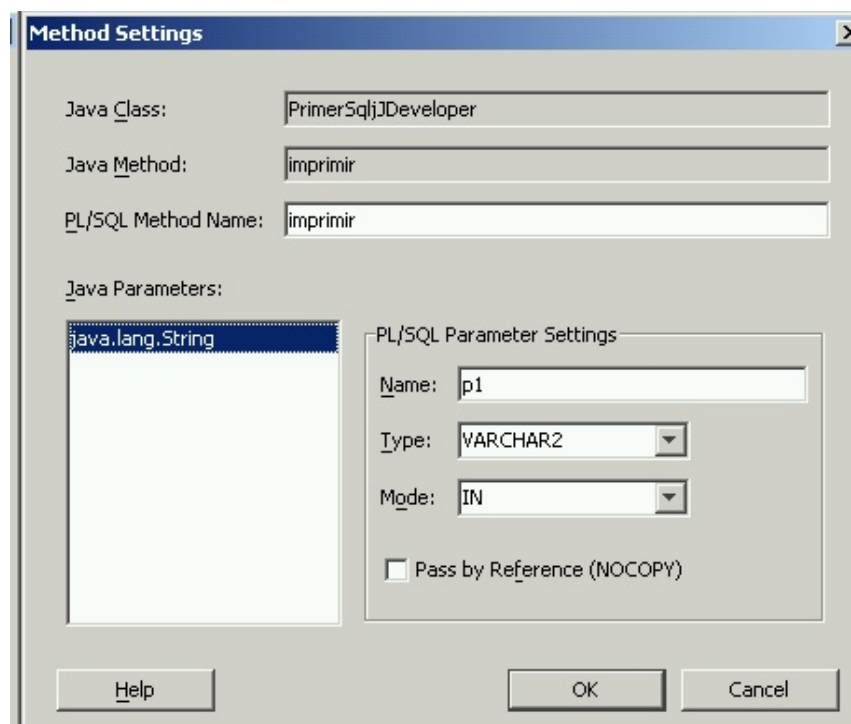
★ Aceptar las opciones que se proponen. Se le puede cambiar el nombre al perfil, si así se desea.

★ En el navegador se creará una pestaña Resources que contendrá al perfil de despliegue.

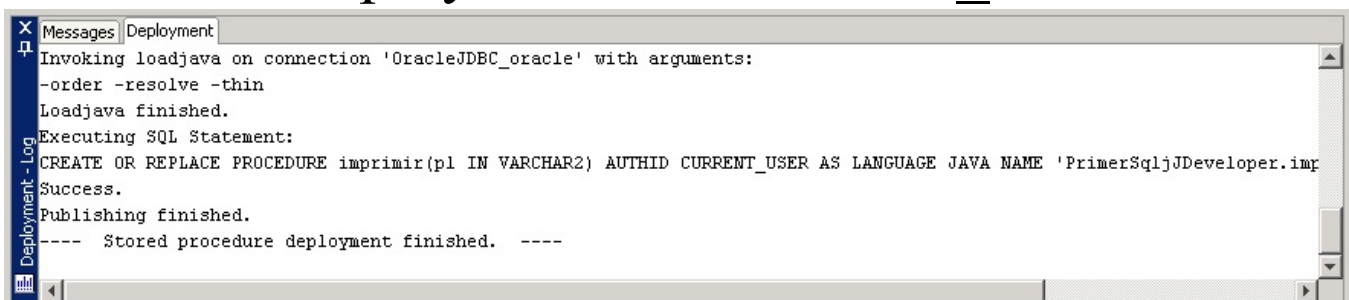
Pasos (V)

★ Sobre el perfil de despliegue: Botón derecho -> Add Stored Procedure -> [imprimir](#) -> Settings.

★ Con esto, estamos diciendo que vamos a crear un *wrapper* PL/SQL para la función imprimir del SQLJ.



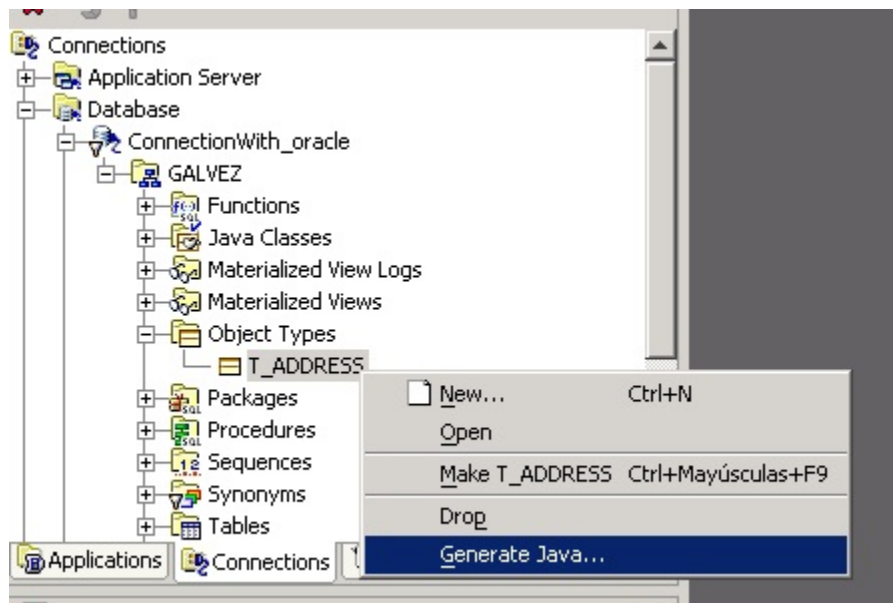
★ Una vez hecho esto, ya sólo queda realizar el despliegue. Sobre el perfil de despliegue-> Botón derecho -> Deploy to -> OracleJDBC_SID:



JPublisher

★ JDeveloper permite ejecutar JPublisher desde un asistente.

★ A través de una conexión es posible ver todos los elementos de un usuario.



★ Si el objeto sobre quien se ejecuta el JDeveloper es un registro, entonces se crean dos ficheros: uno .sqlj que representa al objeto en sí, y otro .java que representa a una referencia a dicho objeto.

Filosofía BC4J

★ JDeveloper permite el desarrollo de aplicaciones que pueden ser depositadas sin modificación en diversas plataformas: OC4J, J2EE, etc.

★ BC4J es una envoltura sobre las tablas y claves foráneas de la base de datos.

★ Sobre esta envoltura se construye una aplicación, a base de formularios.

★ Las aplicaciones se pueden ejecutar en diferentes entornos, según el despliegue que se haga. Nosotros veremos EJB sobre aplicaciones independientes.

Capa de negocio

★ Los componentes que forman la capa de negocio son:

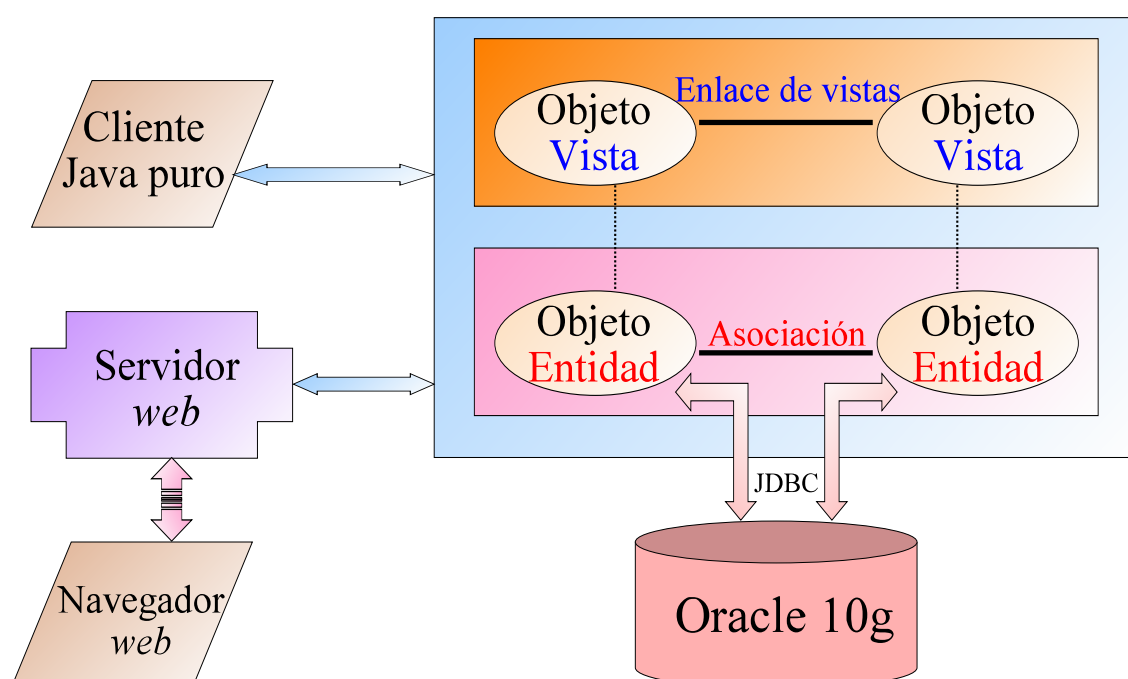
★ Objetos Entidad:

- ▶ Se corresponde con una tabla o vista de la capa de datos.
- ▶ Gestiona la interacción con la base de datos, incluidos los buffer necesarios.
- ▶ Están formados por atributos.
- ▶ Implementa la validación de datos. Para ello puede ser necesario crear **Dominios**.
- ▶ Realmente son clase de Java, luego las reglas de negocio se definen en forma de métodos.

★ Asociaciones:

- ▶ Se corresponde con una clave foránea
- ▶ Se utilizan para representar las relaciones maestro-detalle entre dos objetos entidad.
- ▶ Una asociación es bidireccional.

Framework BC4J



Capa de manipulación

★ Los componentes que forman la capa de manipulación son:

★ Objetos vista:

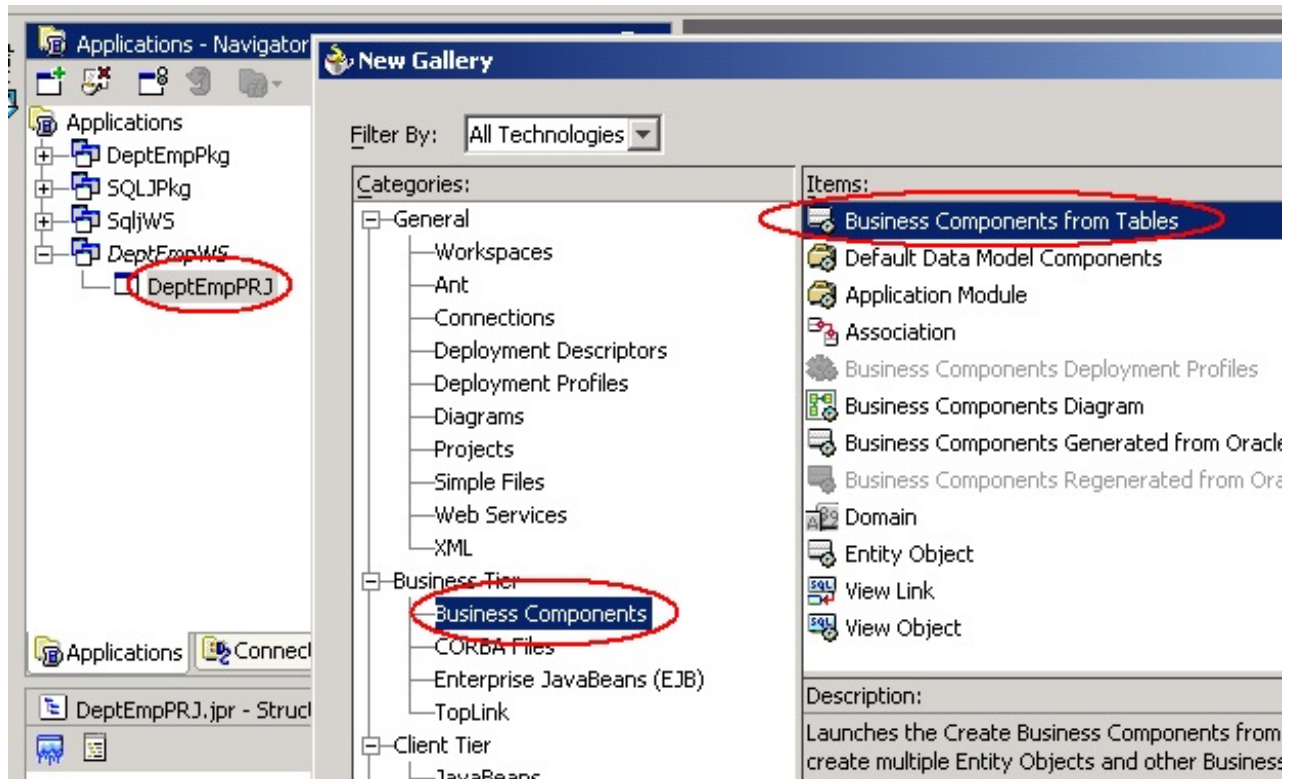
- ▶ Contiene un subconjunto de los datos referenciados por un objeto entidad.
- ▶ Visualiza los datos correspondientes a una consulta SQL.
- ▶ Permite hacer filtrados y ordenación de los datos.
- ▶ Puede hacer referencia a atributos de un objeto entidad, así como a atributos calculados.
- ▶ Permiten la manipulación: inserción, eliminación y actualización.
- ▶ Se implementa como un JavaBean.

★ Enlaces de vistas:

- ▶ Define un enlace maestro-detalle entre dos objetos vista.
- ▶ Es un objeto unidireccional que coordina la recuperación de los datos pertinentes a un maestro.

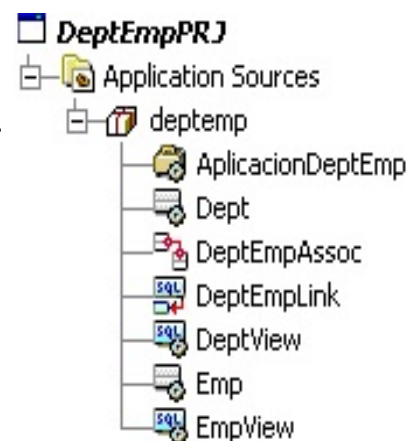
Creación de una aplicación BC4J

★ Una vez creado el correspondiente proyecto (DeptEmpPRJ), hay que crear la capa de negocio:



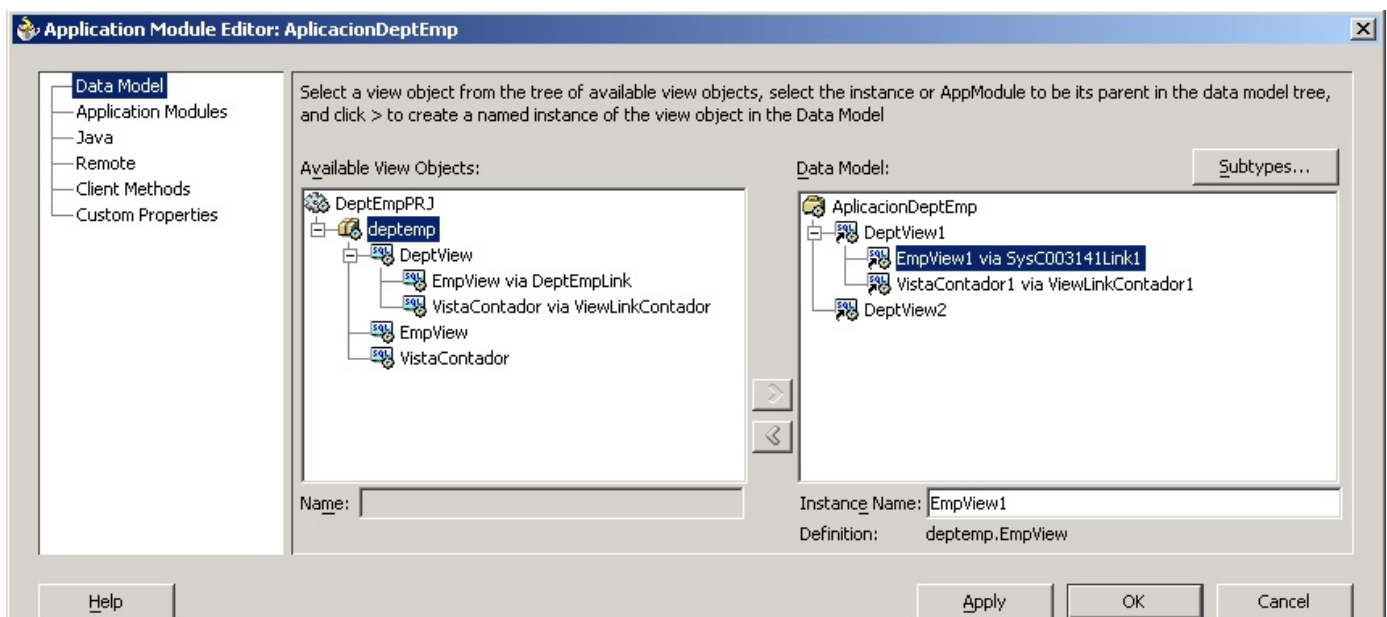
★ En este punto, un asistente nos ayudará a seleccionar las tablas origen para crear los objetos entidad, así como los objetos vista por defecto.

★ Las asociaciones y enlaces también se crean. Puede ser conveniente cambiarles el nombre.



Módulo de aplicación BC4J

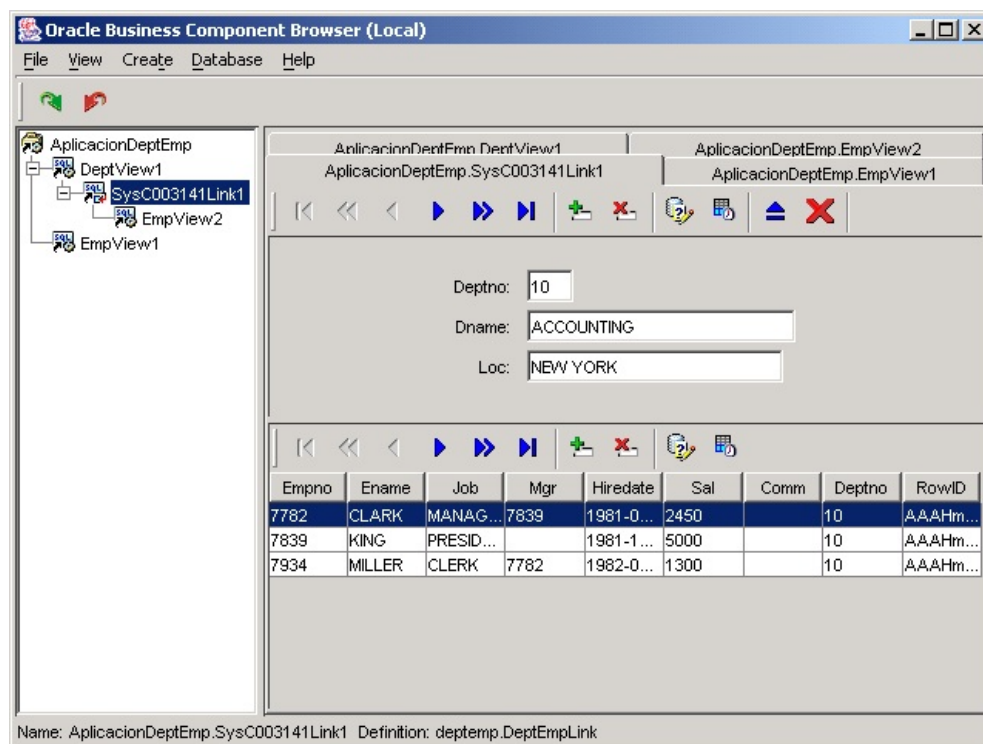
- ★ Con el asistente anterior también se crea un módulo de aplicación.
- ★ El módulo de aplicación es la última fase de la capa intermedia (*middle-tier*).
- ★ Representa un conjunto de relaciones entre objetos vista. Un mismo módulo de aplicación puede contener diferentes composiciones maestro-detalle. Cada composición hace uso de una instancia de un objeto vista concreto.



- ★ Las aplicaciones clientes (*Frames*, JSP, etc.) acceden a un módulo de aplicación a través de JNDI (ya sea en modo local o en modo EJB, según la configuración elegida).
- ★ A partir del módulo de aplicación, las aplicaciones cliente acceden a la instancia de objeto vista que se quiera: clase **ViewObject**.
- ★ La clase ViewObject es muy completa y permite recorrer los registros mediante, a su vez, la clase **Row**.

Test básico de un módulo de aplicación BC4J

☆ La aplicación puede ser testada con la configuración por defecto: Botón derecho sobre AplicacionDeptEmp -> Configurations ó Test:



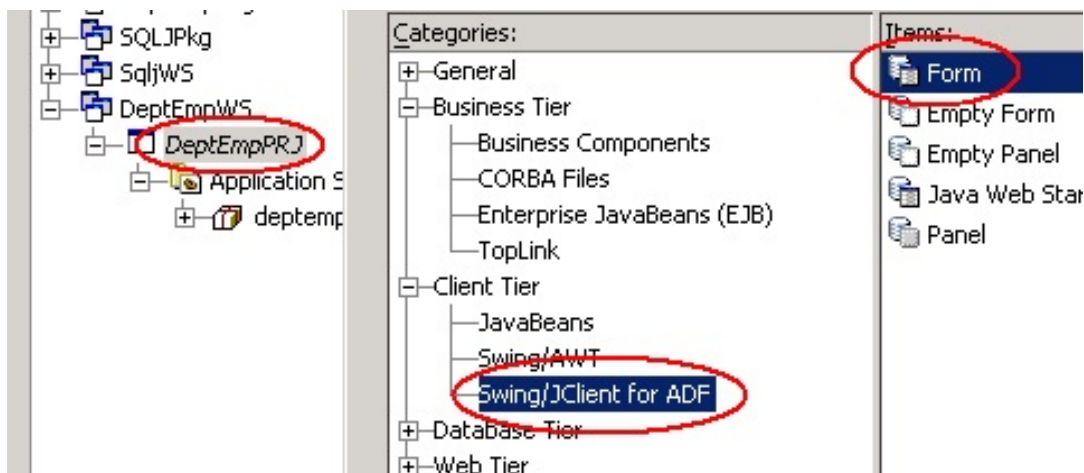
☆ Es conveniente establecer un paquete para la aplicación, y otro distinto para los clientes.

☆ Los clientes generados automáticamente hacen uso de unos ficheros XML para realizar la correspondencia entre los campos del Frame y los atributos de un objeto vista.

Creación de un cliente

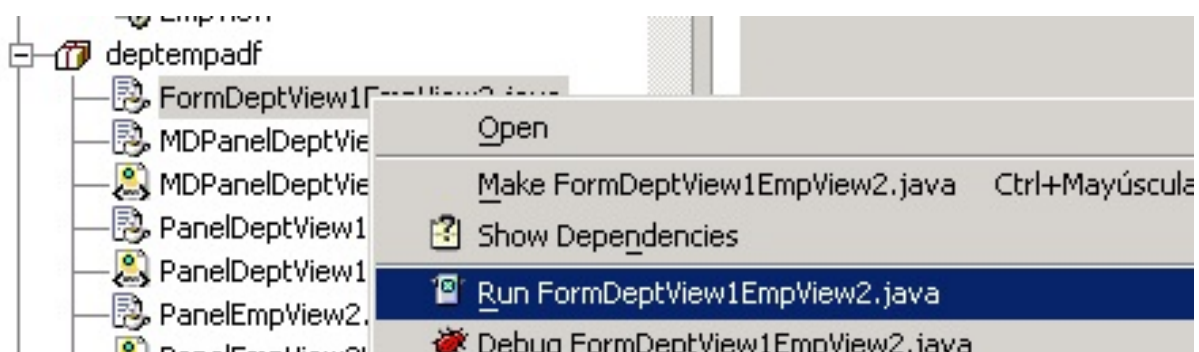
☆ Con el test anterior sólo se puede probar la aplicación desde dentro del JDeveloper.

☆ Para poder depositar todo el resultado en algún sitio y poderlo ejecutar, es necesario crear una capa cliente:



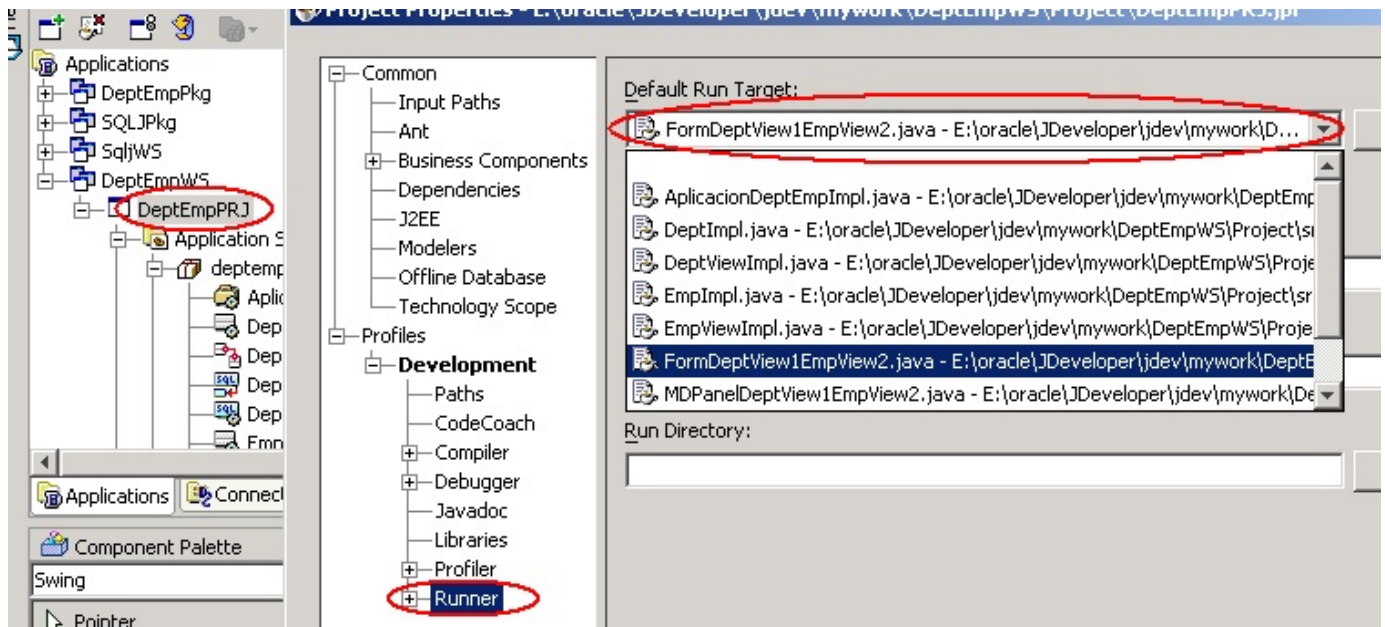
☆ Esto nos llevará a un asistente que nos permitirá crear un formulario parecido al que se generó en el test anterior.

☆ Previamente se creará un modelo de datos que accede a una configuración concreta de una aplicación. El formulario se puede modificar y ejecutar:




Ejecución de un proyecto

☆ Un proyecto se puede ejecutar modificando sus



propiedades y cambiando el punto de entrada:

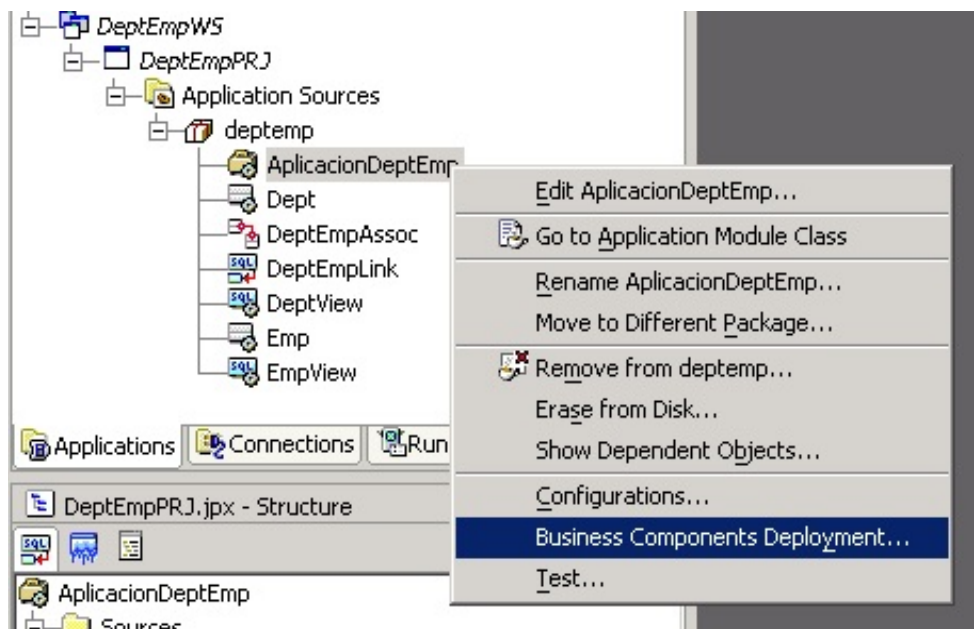
☆ Así, el botón de ejecución () ejecutará el fichero que se indique.

☆ Nótese la enorme cantidad de librerías referenciadas: Libraries.

☆ La sentencia que desencadena la ejecución puede verse en el log del JDeveloper. Esta sentencia puede usarse para lanzar el programa de forma independiente del JDeveloper.

Despliegues de una aplicación BC4J

★ Una vez creada una aplicación, y la capa cliente se pueden hacer distintos despliegues para ponerla en ejecución:



★ El despliegue Simple Archive Files permite generar ficheros JAR que aglutinan todo nuestro proyecto.

El servidor J2EE

★ JDeveloper incorpora un contenedor/servidor J2EE para suministrar *Enterprise Java Beans*.

★ Como fase de instalación, sólo hay que darle una clave al usuario **admin**:

<jdev_install>\j2ee\home>java -jar oc4j.jar -install
lo que pedirá clave y confirmación.

★ El servidor se arranca con:

<jdev_install>\jdev\bin>start_oc4j.bat

★ Y se para con:

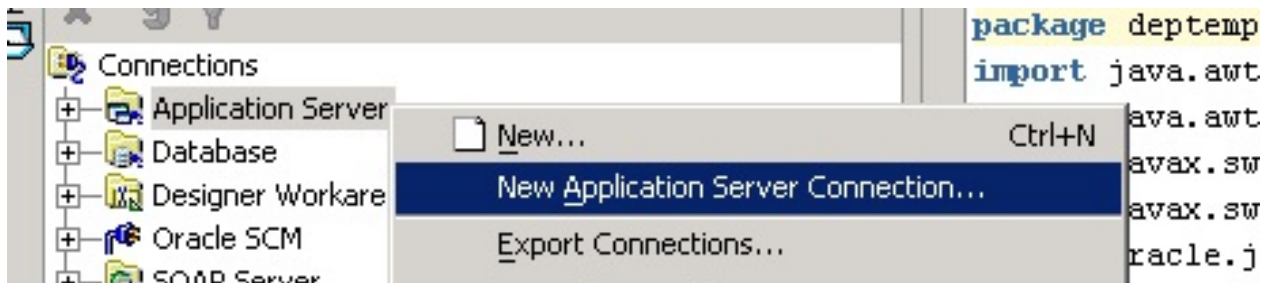
<jdev_install>\jdev\bin>stop_oc4j admin **manager**

★ En todo este proceso quizás sea necesario darle un valor a la variable de entorno JAVA_HOME.

★ Cuidado porque los puertos de conexión han cambiado de una versión a otra del servidor OC4J de J2EE.

Despliegue en forma de *Enterprise JavaBeans*

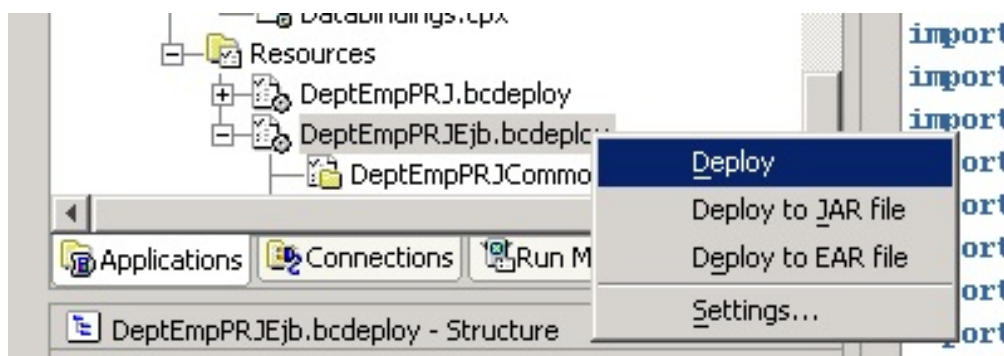
★ Una vez arrancado el servidor, se debe crear una conexión a un servidor de aplicaciones.



★ El tipo de conexión es Standalone OC4J.

★ Ahora podemos comenzar un despliegue EJB Session Beans. Podemos escoger todas las opciones por defecto. Haremos un despliegue del único módulo de aplicaciones que hemos creado.

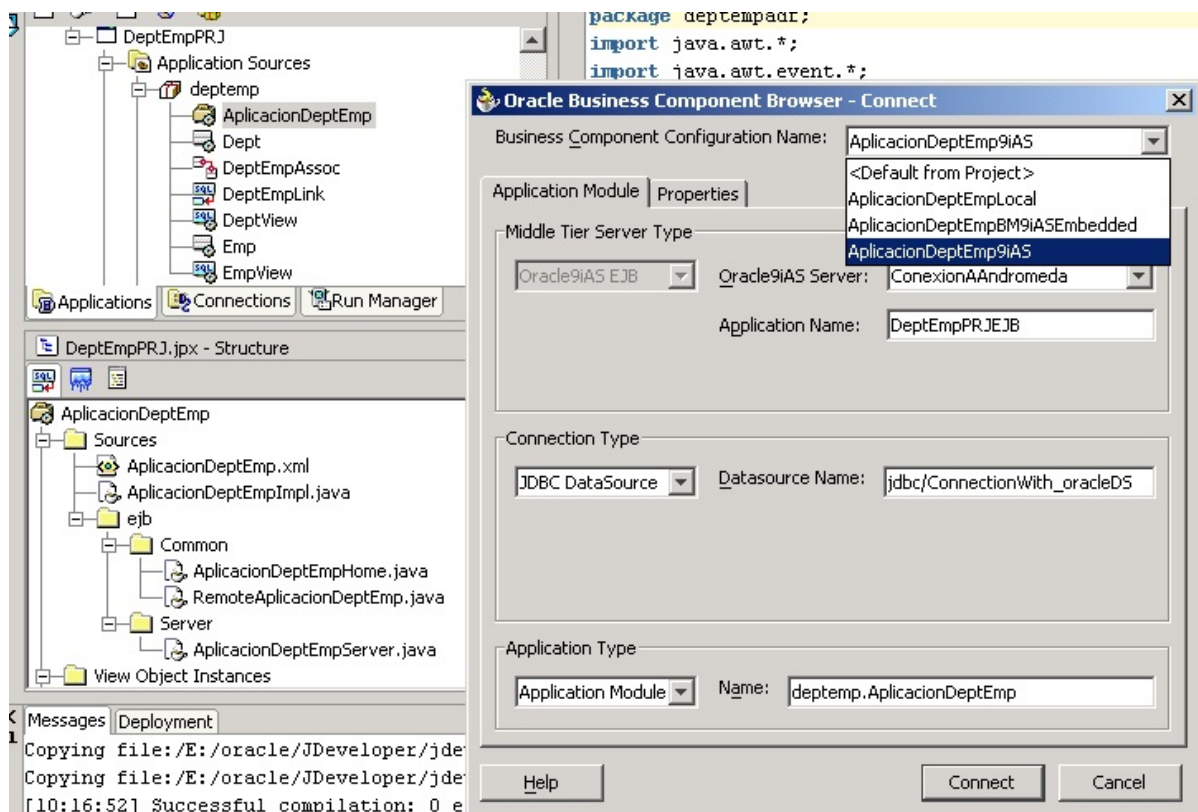
★ Una vez creado el perfil de despliegue, podemos ejecutarlo seleccionando la opción:



Despliegue en forma de *Enterprise JavaBeans*

★ Lo anterior crea una serie de ficheros JAR y los deposita en el contenedor J2EE que además los sirve.

★ Ahora es posible testar la aplicación, teniendo en cuenta que hay que seleccionar un nuevo tipo de conexión creada automáticamente al haber creado un nuevo perfil de despliegue.



★ Aparentemente el resultado es el mismo que en Local, pero se está trabajando con JavaBeans.

Despliegue en JSP

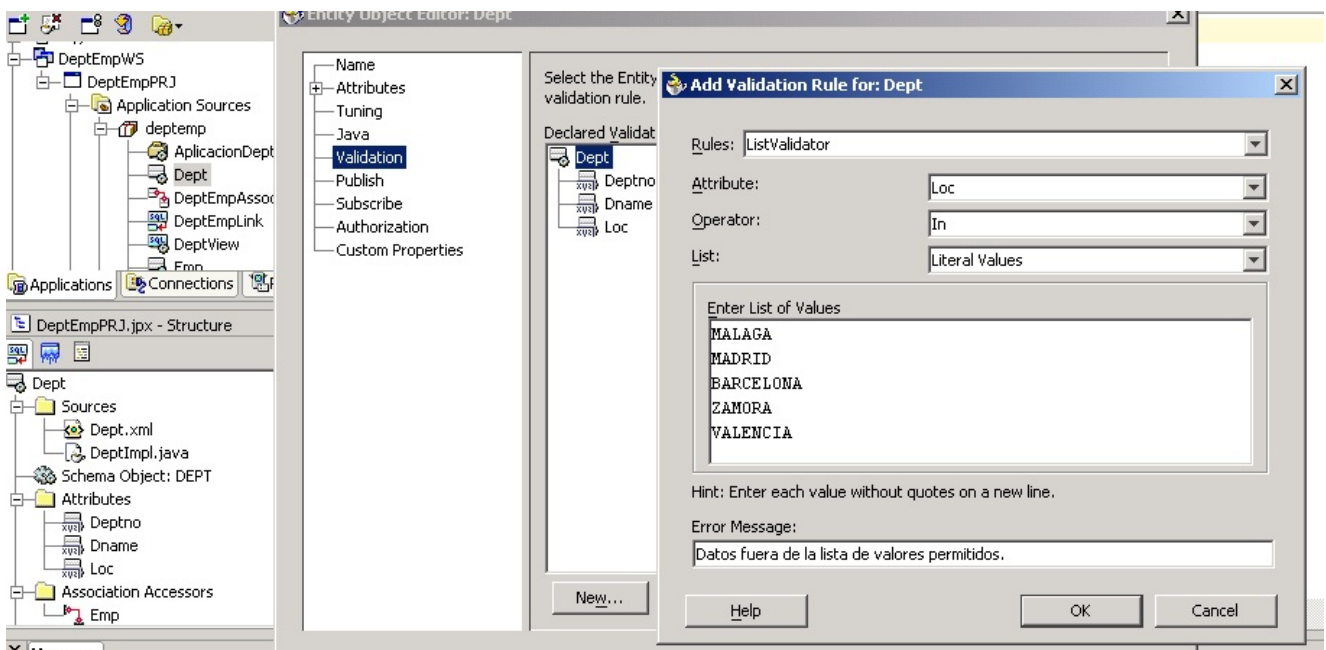
- ★ La creación de páginas JSP se ha modificado en la versión 10g de JDeveloper.
- ★ Hay que crear un esquema de flujo Struts que indica cómo se pasa de una página a otra y qué ejecución desencadena la pulsación de cada botón.
- ★ Seguir los pasos que se indican en cualquier documento de ayuda por internet.
- ★ Cuidado. En lugar de crear las páginas JSP desde el modelo Struts, es mejor crearlas fuera y arrastrarlas después al modelo Struts. También se puede modificar el fichero modelo visual que está en la carpeta *models*.
- ★ Cuidado ya que hay que meter la barra de navegación dentro del formulario de datos.

Propiedades de un objeto entidad

★ Cada objeto entidad tiene asociada una clase Java que permite gestionar cada registro. Además hay un fichero XML que indica sus características dentro de Jdeveloper.

★ Si es necesario modificar el comportamiento de la clase Java, es mejor hacerlo a través del editor de Jdeveloper, quien modifica la clase Java a la vez que actualiza su fichero XML de control.

★ Con este método se pueden añadir mecanismos de validación en la



capa de negocio, como puede verse en la figura. Cualquier vista sobre este objeto entidad se verá forzada a cumplir la validación, así como cualquier aplicación.

Propiedades de un objeto vista

- ☆ Un objeto vista está formado por una sentencia SELECT. Cada columna de dicha SELECT produce un campo de la vista.
- ☆ Además, es posible añadir más campos. Estos campos pueden tener valores calculados.
- ☆ Los campos se calculan una sola vez cuando se carga el cursor por la consulta.
- ☆ Para actualizar los campos calculados es necesario realizar una nueva consulta en tiempo de ejecución.

Java y Bases de Datos (Oracle)

Una síntesis

El presente volumen es el compendio de varios años de docencia en el campo de la programación. El programador actual debe enfrentarse a numerosas dificultades, incluida la integración de diversas tecnologías en una misma aplicación, y este texto se centra en explicar de manera real la integración entre el lenguaje Java y las bases de datos relacionales.

La forma realizar un mismo proceso suele tener, en Informática, un número ilimitado de posibilidades, y el caso que nos ocupa no es una excepción. Aquí se abordan los mecanismos básicos de cada uno de los paradigmas de integración e interconexión existentes; así, Java puede interactuar con una base de datos:

- ▶ Mediante *Java DataBase Connectivity*.
- ▶ Mediante SQLJ.
- ▶ Mediante un acceso interno por procedimientos almacenados.
- ▶ Mediante el uso de un *framework*.

Lejos de proporcionar una pormenorizada explicación de cada una de estas formas de comunicación, el presente volumen pretende ser una síntesis más o menos completa, esto es, un compendio de explicaciones sucintas junto con multitud de ejemplos que pueden ser ejecutados por el propio lector.

ISBN 978-84-693-0176-0



9 788469 301760 >