Bases de datos orientadas a objetos y objeto-relacionales

1 Introducción.

Las Bases de Datos Relacionales (BDR) son ideales para aplicaciones tradicionales que soportan tareas administrativas y de gestión, y que trabajan con datos de estructuras simples y poco cambiantes, incluso cuando la aplicación pueda estar desarrolla en un lenguaje Orientado a Objetos y sea necesario un Mapeo Objeto Relacional (ORM)

Pero cuando la aplicación requiere otras necesidades, como por ejemplo, soporte multimedia, almacenar objetos muy cambiantes y complejos en estructura y relaciones, este tipo de bases de datos no es el más adecuado. Hablamos entonces de bases de datos avanzadas.

Si queremos representar un objeto y su relaciones en una BDR esto implica que:

- Los objetos deben ser descompuestos en diferentes tablas.
- A mayor complejidad, mayor número de tablas, de manera que se requieren muchos enlaces (joins) para recuperar un objeto, lo cual disminuye dramáticamente el rendimiento.

Las **Bases de Datos Orientadas a Objetos** (**BDOO**) o Bases de Objetos se integran directamente y sin problemas con las aplicaciones desarrolladas en lenguajes orientados a objetos, ya que **soportan un modelo de objetos puro y** son ideales para almacenar y recuperar datos complejos permitiendo a los usuarios su navegación directa (sin un mapeo entre distintas representaciones).

Las **Bases de Objetos** aparecieron a finales de los años 80 **motivadas fundamentalmente** por dos razones:

- Las necesidades de los lenguajes de Programación Orientados a Objetos (POO), como la necesidad de persistir objetos.
- Las limitaciones de las bases de datos relacionales, como el hecho de que sólo manejan estructuras muy simples (tablas) y tienen poca riqueza semántica para atender nuevos tipos de aplicaciones: Diseño y fabricación en ingeniería, aplicaciones multimedia, sistemas de información geográfica o aplicaciones CAD, por ejemplo.

Pero como las BDOO no terminaban de asentarse, debido fundamentalmente a la inexistencia de un estándar, y las BDR gozaban y gozan en la actualidad de una gran aceptación, experiencia y difusión, debido fundamentalmente a su gran robustez y al lenguaje SQL, los fabricantes de bases de datos comenzaron a implementar nuevas funcionalidades orientadas a objetos en las BDR existentes. Así surgieron, las bases de datos objeto-relacionales.

Las **Bases de Datos Objeto-Relacionales** (BDOR) son bases de datos relacionales que han evolucionado hacia una base de datos más extensa y compleja, incorporando conceptos del modelo orientado a objetos. Pero en estas bases de datos **aún existe un mapeo de objetos subyacente**, que es costoso y poco flexible, cuando los objetos y sus interacciones son complejos.

PARA SABER MÁS

Si consultas las páginas 5-9 del documento de este enlace, verás la propuesta de STONEBREAKER sobre la idoneidad de uno u otro sistema de bases de datos, en función de la aplicación que vayamos a desarrollar.

Información sobre Bases de Datos que almacenan Objetos.

http://eisc.univalle.edu.co/cursos/web/material/750283/1/Modelo_objeto_relacional_y_Procedimientos_almacenados.pdf

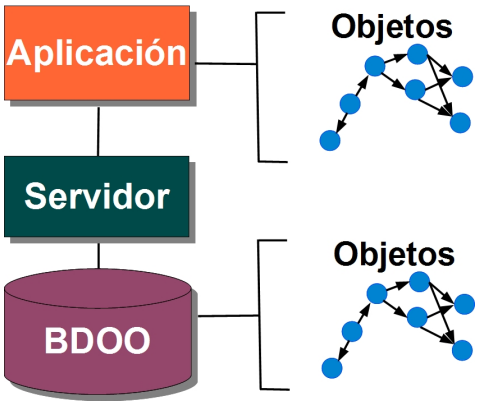
En este enlace, encontrarás bastante información sobre las BDOO .

Información sobre necesidad de las BDOO

http://kuainasi.ciens.ucv.ve/db4o/DB4o-P1.htm#Necesidad

2 Características de las bases de datos orientadas a objetos

En una BDOO, los datos se almacenan como objetos. Un **objeto** es, al igual que en POO, una entidad que se puede identificar unívocamente y que describe tanto el estado como el comportamiento de una entidad del 'mundo real'. El estado de un objeto se describe mediante atributos y su comportamiento es definido mediante procedimientos o métodos.



La principal característica de las BDOO es que **soportan un modelo de objetos puro y** que **el lenguaje de programación y el esquema de la base de datos utilizan las mismas definiciones de tipos.**

Otras características importantes de las BDOO son las siguientes:

- Los datos se **almacenan** como **objetos**.
- **Soportan las características propias de la Orientación a Objetos** como agregación, encapsulamiento, polimorfismo y herencia. La herencia se mantiene en la propia base de datos.
- **Identificador de objeto** (**OID**). Cada objeto tiene un identificador, generado por el sistema (no modificable por el usuario), que es único para cada objeto, lo que supone que cada vez que se necesite modificar un objeto, habrá que recuperarlo de la base de datos, hacer los cambios y almacenarlo nuevamente. Los OID son independientes del contenido del objeto, esto es, si cambia su información, el objeto sigue teniendo el mismo OID. Dos objetos serán equivalentes si tienen la misma información pero diferentes OID.
- **Jerarquía y extensión de tipos**. Se pueden definir nuevos tipos basándose en otros tipos predefinidos, cargándolos en una jerarquía de tipos (o jerarquía de clases).
- **Objetos complejos**. Los objetos pueden tener una estructura de objeto de complejidad arbitraria, a fin de contener toda la información necesaria que describe el objeto.
- **Acceso navegacional de datos.** Cuando los datos se almacenan en una estructura de red densa y probablemente con una estructura de diferentes niveles de profundidad, el acceso a datos se hace principalmente navegando la estructura de objetos y se expresa de forma natural utilizando las construcciones nativas del lenguaje, sin necesidad de uniones o joins típicas en las BDR.
- **Gestión de versiones.** El mismo objeto puede estar representado por múltiples versiones. Muchas aplicaciones de bases de datos que usan orientación a objetos requieren la existencia de varias versiones del mismo objeto, ya que si estando la aplicación en funcionamiento es necesario modificar alguno de sus módulos, el diseñador deberá crear una nueva versión de cada uno de ellos para efectuar cambios.

En 1989 se hizo el **manifiesto Malcom Atkinson** que propone **13 características obligatorias** para los sistemas de bases de datos orientados a objetos basado en dos criterios: *debe ser un sistema orientado a objetos y debe ser un sistema gestor de bases de datos (SGBD)*

PARA SABER MÁS
Existen diferentes manifiestos sobre las características que debe cumplir un sistema de bases de datos orientado a objetos. En el siguiente enlace puedes consultar estos manifiestos.
(Están a partir de la página 4).

Manifiestos sobre las bases de datos orientadas a objetos
http://zarza.fis.usal.es/~fgarcia/docencia/poo/99-00/bdoo.pdf

The Object-Oriented Database System Manifesto
http://www.cs.cmu.edu/afs/cs.cmu.edu/user/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html

2.1 Ventajas e inconvenientes

El uso de una BDOO puede ser ventajoso frente a una BDR relacional si nuestra aplicación requiere alguno de estos elementos:

- Un gran número de tipos de datos diferentes.
- Un gran número de relaciones entre los objetos.
- Objetos con comportamientos complejos.

Una de las principales ventajas de una BDOO es la **transparencia** (manipulación directa de datos utilizando un entorno de programación basado en objetos), por lo que el programador, solo se debe preocupar de los objetos de su aplicación, en lugar de cómo los debe almacenar y recuperar de un medio físico.

Otras ventajas de un sistema de bases de datos orientado a objetos son las siguientes:

- **Gran capacidad de modelado**. La utilización de objetos permite modelar el 'mundo real' de una manera óptima gracias al encapsulamiento y la herencia. Es una representación más natural.
- **Flexibilidad**. Permiten una estructura cambiante con solo añadir subclases.
- **Extensibilidad**. Se pueden construir nuevos tipos de datos a partir de los ya existentes, agrupar propiedades comunes de diversas clases e incluirlas en una superclase, lo que reduce la redundancia.
- **Adecuación a aplicaciones avanzadas de bases de datos, (como CASE, CAD, y multimedia)**
- **Interfaz única entre el LMD (lenguaje de manipulación de datos) y el lenguaje de programación.** Esto elimina el tener que incrustar un lenguaje declarativo como es el SQL en un lenguaje imperativo como Java.
- **Lenguaje de consultas más expresivo.** El lenguaje de consultas es navegacional de un objeto al siguiente, en contraste con el lenguaje declarativo SQL.
- **Manipula de forma rápida y ágil objetos complejos**, ya que la estructura de la base de datos está dada por referencias (apuntadores lógicos) entre objetos.
- **Alta velocidad de procesamiento**. Como el resultado de las consultas son objetos, no hay que reensamblar los objetos cada vez que se accede a la base de objetos.
- **Mejora los costes de desarrollo**, ya que es posible la reutilización de código, una de las características de los lenguajes de programación orientados a objetos.

Entre los inconvenientes o **desventajas** hay que destacar**:**

- **Carencia de un modelo de datos universal**. No hay ningún modelo de datos aceptado universalmente, y la mayor parte de los modelos carecen de una base teórica.
- **Falta de estándares**. Existe una carencia de estándares general para los sistemas de BDOO.
- **Falta de experiencia**. El uso de las BDOO es aún limitado.
- **Competencia de otros modelos**. Las bases de datos relacionales y objeto-relacionales están muy asentadas y extendidas, siendo un duro competidor.
- **Difícil optimización de consultas**. La optimización de consultas requiere conocer la implementación de los objetos, para poder acceder a la base de datos de manera eficiente. Sin embargo, esto compromete el concepto de encapsulación.
- **Complejidad**. El incremento de funcionalidad provisto por un sistema de BDOO lo hace más complejo que un sistema de BDR. Esto conlleva productos más caros y difíciles de usar.

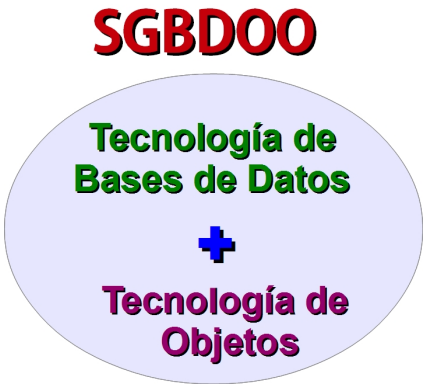
CITA PARA PENSAR.
Albert Einstein: "Cada día sabemos más y entendemos menos".

3 Gestores de Bases de Datos Orientadas a Objetos

Un **Sistema Gestor de Bases de Datos Orientada a Objetos (SGBDOO**) y en inglés ODBMS, Object Database Management System) es un software específico, dedicado a servir de interfaz entre la base de objetos, el usuario y las aplicaciones que la utilizan.

Un SGBDOO incorpora el paradigma de Orientación a Objetos y permite el almacenamiento de objetos en soporte secundario:

- Por ser SGBD debe incluir mecanismos para optimizar el acceso, gestionar el control de concurrencia, la seguridad y la gestión de usuarios, así como facilidar la consulta y recuperación ante fallos.
- Por ser OO incorpora características de identidad, encapsulación, herencia, polimorfismo y control de tipos.



Cuando aparecieron las bases de datos orientadas a objetos, un grupo formado por desarrolladores y usuarios de bases de objetos, denominado ODMG (Object-Oriented Database Management Group), propuso un estándar que se conoce como **estándar ODMG-93** y que se ha ido revisando con el tiempo, pero que en realidad **no ha tenido mucho éxito**, aunque es un punto de partida. La última versión es el estándar ODMG 3.0.

3.1 Estándar ODMG.

El **estándar ODMG** (Object Database Management Group) trata de estandarizar conceptos fundamentales de los Sistemas Gestores de Bases de Datos Orientados a Objetos (SGBDOO) e intenta definir un **SGBDOO como un sistema que integra las capacidades de las bases de datos con las capacidades de los lenguajes de programación orientados a objetos**, de manera que los objetos de la base de datos aparezcan como objetos del lenguaje de programación.

Fué desarrollado entre los años 1993 y 1994 por representantes de un amplio conjunto de empresas relacionadas con el desarrollo de software y sistemas orientados a objetos.

Arquitectura del estándar ODMG 3.0

La arquitectura propuesta por ODMG consta de:

- Un **modelo de objetos** que permite que tanto los diseños, como las implementaciones, sean portables entre los sistemas que lo soportan.
- Un **sistema de gestión** que soporta un lenguaje de bases de datos orientado a objetos, con una sintaxis similar a un lenguaje de programación también orientado a objetos.
- Un **lenguaje de base de datos** que es especificado mediante:
  - Un Lenguaje de Definición de Objetos (**ODL**)
  - Un Lenguaje de Manipulación de Objetos (**OML**)
  - Un Lenguaje de Consulta (**OQL**)

  siendo todos ellos portables a otros sistemas con el fin de conseguir la portabilidad de la aplicación completa.
- **Enlaces con lenguajes Orientados a Objetos** como C++, Java, Smaltalk.

El **modelo de objetos ODMG** epecifica las características de los objetos, cómo se relacionan, cómo se identifican, construcciones soportadas, proporciona los tipos de datos, los constructores de tipos y otros conceptos que pueden utilizarse en el ODL para especificar el esquema de la base de datos de objetos.

Vamos a destacar algunas de las **características más relevantes** del estándar ODMG:

- Las primitivas básicas de modelado son los **objetos** y los **literales**.
- Un objeto tiene un **Identificador de Objeto** (**OID**- Object IDentifier) y un estado (valor actual) que puede cambiar y tener una estructura compleja.
- **Un literal no tiene OID**, pero si un valor actual, que es constante (entre ellos el NULL).
- El estado está definido por los valores que el objeto toma para un conjunto de propiedades. Una propiedad puede ser:
  - Un atributo del objeto.
  - Una interrelación entre el objeto y otro u otros objetos.
- Objetos y literales están organizados en **tipos.** Todos los objetos y literales de un mismo tipo tienen un comportamiento y estado común (mismas propiedades).
- Un **objeto queda descrito por cuatro características**: ***identificador*** (único en BD), ***nombre*** (también identifica objeto), ***tiempo de vida*** (transitorio/persistente) y ***estructura*** (atómico, colección o estructurado).
- Los **tipos de objeto**s se descomponen en atómicos, colecciones y tipos estructurados.
  - **Tipos atómicos** o básicos: constan de un único elemento o valor (l*ong, double, char, float,* ...).
  - **Tipos estructurados**: compuestos por un número fijo de elementos que pueden ser de distinto tipo (*date, time*, ..).
  - **Tipos colección**: número variable de elementos(literal/objeto) del mismo tipo. Entre ellos:
    - Set<tipo>: grupo desordenado de elementos y sin duplicados.
    - Bag<tipo>: grupo desordenado de elementos que permite duplicados.
    - List<tipo>: grupo ordenado de elementos que permite duplicados.
    - Array<tipo> : grupo ordenado de elementos que permite el acceso por posición.
    - Dicctionary<clave, valor>: grupo de elementos donde cada 'valor' está asociado a una 'clave'

Algunos fabricantes sólo ofrecen vinculaciones de lenguajes específicos, sin ofrecer capacidades completas de ODL y OQL.

¿Qué **estrategias** o enfoques se siguen para el **desarrollo de SGBD que soporten objetos?**
Básicamente, las siguientes:

- Ampliar un lenguaje de programación OO existente con capacidades de BD (Ejemplo: *GemStone*).
- Proporcionar bibliotecas de clases con las capacidades tradicionales de las bases de datos, como persistencia, transacciones, concurrencia, etc., (Ejemplo: *ObjectStore y Db4o de Versant*)
- Ampliar un lenguaje de BD con capacidades OO, caso de SQL 2003 y Object SQL (OQL, propuesto por ODMG)

Tal y como estarás pensando, la carencia de un estándar real hace difícil el soporte para la portabilidad de aplicaciones y su interoperabilidad, y es en parte por ello, que a diferencia de las bases de datos relacionales donde hay muchos productos donde elegir, la variedad de sistemas de bases de datos orientadas a objetos es mucho menor. En la actualidad hay diferentes productos de este tipo, tanto con licencia libre como propietaria.

A continuación te indicamos algunos **ejemplos de SGBOO**:

- **Db4o de Versant**. Es una BDOO Open Source para Java y .NET. Se distribuye bajo licencia GPL.
- **Matisse**. Es un SGBOO basado en la especificación ODMG, proporciona lenguajes para definición y manipulación de objetos, así como interfaces de programación para C, C++, Eiffel y Java.
- **NeoDatis.** Es una BDOO de código abierto que funciona con Java, .Net, Groovy y Android.
- **ObjectDB**. Es una BDOO que ofrece soporte para Java, C++, y Python entre otros lenguajes. No es un producto libre, aunque ofrecen versiones de prueba durante un periodo determinado.
- **EyeDB**. Es un SGBOO basado en la especificación ODMG, proporciona lenguajes para definición y manipulación de objetos, e interfaces de programación para C++ y Java. Se distribuye bajo licencia GNU y es software libre.
- **ObjectStore y GemStone**. Son otros SGBDOO.

3.2 Instalación del gestor de objetos Db4o.

En los siguientes apartados trabajaremos con **Db4objects de Versant o** simplemente **Db4o,** una **base de datos orientada a objetos nativa de Java**, disponible para Java y .Net, y utilizada en la actualidad por diversas compañías para el desarrollo de aplicaciones de dispositivos móviles, dispositivos médicos y biotecnología, aplicaciones Web, software enlatado y aplicaciones en tiempo real.

Tres **características importantes de Db4o** son las siguientes:

- El modelo de clases es el propio esquema de la base de datos, por lo que se elimina el proceso de diseño, implementación y mantenimiento de la base de datos.
- Está diseñada bajo la estrategia de proporcionar bibliotecas de clases con las capacidades tradicionales de las bases de datos, y con el objetivo de cero administración
- Puede trabajar como **base de datos embebida**, lo que significa que se puede distribuir con la aplicación, y solo la aplicación que lanza la base de datos embebida puede acceder a ella, siendo ésta invisible para el usuario final.

Una de las ventajas de db4o es que es un simple fichero .jar distribuible con cualquier aplicación sin necesidad de tener que instalar nada. Tampoco necesita drivers de tipo JDBC o similar.

La **instalación** de db4o consistirá en:

- Instalar el motor de base de datos, que son las clases necesarias para hacer que funcione la API en toda su extensión.
- Instalar alguna aplicación para visualizar los datos con los que se está trabajando. Esto último es necesario, pues en otro caso se trabajaría a ciegas con los datos.

Te puedes descargar Db4o desde su web oficial:

Web oficial de Db4o

http://www.db4o.com/espanol/

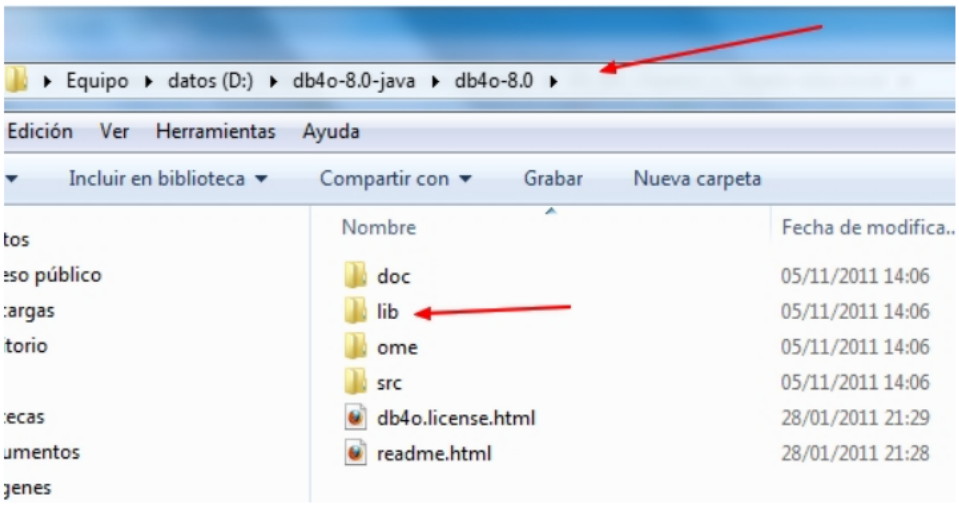
Los pasos para **instalar e integrar Db4o con el IDE NetBeans** son los siguientes:

- Accedemos a la web oficial de Db4o http://www.db4o.com/
- Pulsamos el botón **Downloads Now**
- En la siguiente página seleccionamos el producto a descargar **(db4o.8.0 for Java)** y pulsamos el botón **Download**.

db4objects BY VERSANT

DOWNLOAD NOW

- Hay que **descomprimir** el archivo **db4o-8.0-java.zip**.
- La carpeta descomprimida tendrá la siguiente estructura de directorios y archivos:
  - **doc**: incluye documentación y un tutorial en formato Javdoc y PDF.
  - **lib**: librerías en formato JAR
  - **ome**: object manager enterprise. Visor de objetos.
  - **src**: código fuente de las librerías



Db4o tiene varias configuraciones para su versión 8.0.

- **db4o-8.0-core*.jar**. Archivos core que contienen el núcleo del motor
- **db4o-8.0-cs*.jar.** Archivos cs que contiene la versión cliente/servidor
- **db4o-8.0-optional*.jar**. Archivos que añaden funcionalidad avanzada al motor de base de datos.
- **db4o-8.0-all*.jar**. Contiene todas las características anteriores. **Instalación completa**

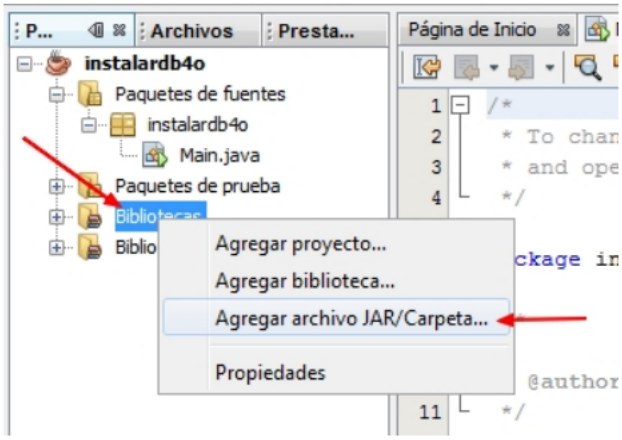
Db4o también está disponible para diferentes versiones de JDK.

Para la instalación completa existen las siguientes versiones:

- **db4o-8.0-all-java1.1.jar.** Compatible con los JDK que proporcinen compatibilidad con JDK 1.1.x
- **db4o-8.0-all-java1.2.jar.** Desarrollado para los JDK entre versiones 1.2 y 1.4
- **db4o-8.0-all-java5.jar.** Desarrollo para los JDK 5 y 6.

- Dentro del **directorio lib** se encuentran las librerías en formato . Jar
- El archivo que nos interesa es **db4o-8.0*-all-java5.jar**

- Creamos o Abrimos un **proyecto en el que trabajaremos con db4o**
- Nos situamos sobre **Bibliotecas** y desde su menú contextual (click derecho)
- Seleccionamos la opción **Agregar archivo JAR/Carpeta**



- Añadimos el JAR que nos interesa, en nuestro caso **db4o-8.0*-all-java5.jar**
- Ya podemos utilizar en nuestro proyecto: una base de datos db4o y toda su Interfaz de Programación para Aplicaciones (API).

PARA SABER MÁS
En el siguiente enlace, en inglés, encontrarás un extenso y completo tutorial sobre db4o

Tutorial oficial de Db4o
http://community.versant.com/Documentation/Reference/db4o-8.0/java/tutorial/

Artículo interesante sobre Db4o
http://www.db4o.com/espanol/db4o%20Whitepaper%20-%20Bases%20de%20Objetos.pdf

4 La API de la Base de Objetos

Todos los SGBDOO, independientemente de su estrategia de diseño, proporcionan una API (Interfaz de Programación de Aplicaciones), más o menos extenso, disponible para ciertos lenguajes OO. En el caso de Db4o, la API está disponible para Java y .Net.

Los **principales paquetes** del API de Db4o para Java son los siguientes:

- **com.db4o**. Paquete principal (core) de la Base de Objetos. Las interfaces y clases más importantes que incluye son:
  - **ObjectContainer**. Es el interfaz que permite realizar las principales tareas con la base de objetos. Un ObjectContainer puede representar una **base de datos independiente** (stand-alone) o una **conexión a un servidor** (en línea cliente-servidor). Este interfaz proporciona métodos para almacenar store(), consultar queryByExample() y eliminar delete() objetos de la base de datos, así como cerrar la conexión a ésta close(). También permite confirmar commit y deshacer rollback transacciones.
  - **EmbeddedObjectContainer**. Es un interfaz que extiende a ObjectContainer y representa un ObjectContainer local atacando a la base de datos.
  - **Db4oEmbedded.** Es una clase que proporciona métodos estáticos para conectar con la base de datos en modo embebido.
  - **ObjectServer.** Es el interfaz que permite trabajar con una base de datos db4o en modo cliente-servidor.
  - **ObjectSet.** Es un interfaz que representa el conjunto de objetos devueltos por una consulta.
- **com.db4o.query**. Paquete con funcionalidades de consulta. Proporciona interfaces que permiten establecer las condiciones y criterios de un consulta y una clase para realizar consultas mediante Native Query (Consultas nativas).
- **com.db4o.config**. Paquete con funcionalidades de configuración. Contiene interfaces y clases que nos permiten configurar y/o personalizar la base de objetos según necesiades. La configuración de la base de objetos se hace por norma general antes de abrir la sesión en la misma.
  - **EmbeddedConfiguration**. Es la interface de configuración para el uso en modo embebido.

Siempre que trabajemos con bases de objetos Db4o utilizaremos el interface ObjectContainer, puesto que es quien representará a la base de objetos, sea embebida o no.

DESTACADO
La documentación del API viene en formato JavaDoc y la puedes encontrar en el directorio /doc/api del fichero .zip descargado y descomprimido.

PARA SABER MÁS.
En el siguiente enlace puedes consultar la documentación oficial del API de db4o para Java

Documentación oficial del API de db4o
http://community.versant.com/Documentation/Reference/db4o-8.0/java/api/

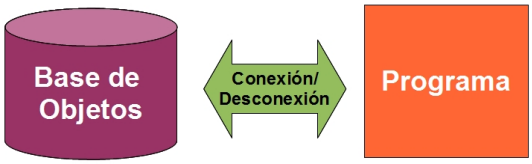
4.1 Apertura y cierre de conexiones.

En general, la conexión de una aplicación Java con una BDOO se podrá realizar vía:

- JDBC
- El API proporcionado por el propio gestor de objetos.

En el caso de Db4o, el paquete com.db4o proporciona las clases e interfaces que permiten abrir conexiones a una base de objetos db4o, así como el cierre de la misma. Estas son:

- **Abrir conexión.** Podemos utilizar las siguientes clases:
  - Db4oEmbedded**.** Es una clase que hereda de java.lang.Object y proporciona métodos estáticos como openFile() para abrir una instancia de la base de datos en **modo embebido**. En modo embebido tiene la limitación de que sólo se puede utilizar en la base de datos una conexión.
  - ObjectServer. Es un interfaz que permite trabajar con una base de datos db4o en **modo cliente-servidor.** Una vez abierta la base de datos como servidor mediante Db4o.openServer(), el método openClient() del interfaz ObjectServer permitirá **abrir conexiones cliente directamente en memoria** o bien **mediante TCP/IP.**
- **Cerrar conexión.** Para ello utilizaremos el método close() de la interfaz ObjectContainer.

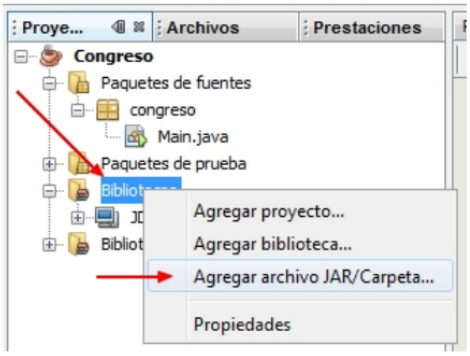


El **esquema de trabajo para operar con la base de objetos** será el siguiente:

- Declarar un ObjectContainer.
- Abrir una conexión a la base de objetos (openFile()). Si al abrir la BDOO esta no existe, se creará.
- Realizar operaciones con la base de objetos como consultas, borrados y modificaciones (en un bloque try{ } catch (){ })
- Cierre o desconexión de la base de objetos (close())

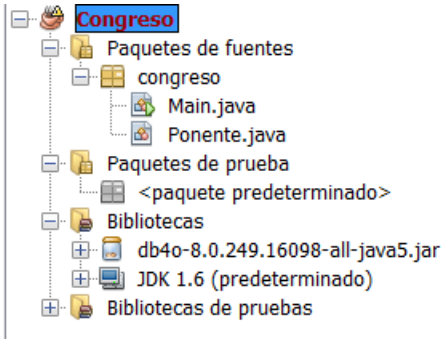
EJEMPLO. [AD05_Congreso.zip]. Este ejemplo muestra la creación de una base de objetos en modo embebido. En la BDOO creada se almacenan 4 objetos mediante el método store(). Físicamente, la BDOO será un **fichero de nombre** congreso.db4o almacenado **en el directorio raíz del proyecto**.

- Desde el IDE NetBeans creamos un nuevo **proyecto de nombre Congreso**
- Nos situamos sobre el proyecto, en el **nodo Bibliotecas**.
- Hacemos un click derecho sobre Bibliotecas
- Ejecutamos la **opción 'Agregar archivo JAR/Carpeta**...



- Añadimos el JAR que nos interesa, en nuestro caso **db4o-8.0*-all-java5.jar**

Creamos la clase Ponente y la clase Main

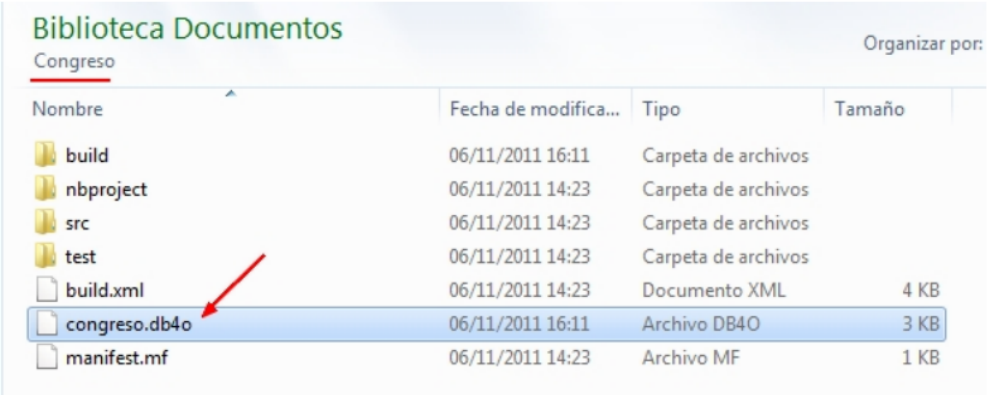



```
package congreso;
//paquetes necesarios del API Db4o para Java
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
public class Main {
   public static void main(String[] args) {
    ObjectContainer db = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),
            "congreso.db4o");
    //La base de datos física es el fichero "congreso.db4o" almacenado en la
    //carpeta raíz del proyecto creado
    try {
      almacenarPonentes(db);
    }catch(Exception e){
        System.out.println(e.getMessage());
    } finally {
      db.close(); //cerrar la conexión a la base de datos
    }
  }
  //Método para almacenar datos en la Base de Objetos.
  public static void almacenarPonentes(ObjectContainer db) {
    //se crean cuatro objetos tipo alumno con valores asignados
    Ponente p1 = new Ponente("11A", "Antonio Camaco", "acamacho@gmail.es", 300);
    Ponente p2 = new Ponente("22B","Isabel Pérez", "iperez@hotmail.es", 100);
    Ponente p3 = new Ponente("33C","Ana Navarro", "anavarro@yahoo.com", 200);
    Ponente p4 = new Ponente("44D","Pedro Sánchez", "psanchez@mixmail.com", 90);
    //Persistir Objetos: almacenamos los objetos con el método store()
    db.store(p1);
    db.store(p2);
    db.store(p3);
    db.store(p4);

  }
}
```

```
package congreso;
//clase que implementa la entidad Ponente
public class Ponente {
  private String nif;
  private String nombre;
  private String email;
  private float cache;
  //constructores
  public Ponente() {...}
  public Ponente(String ni, String n, String e) {...}
  public Ponente(String ni, String no, String e, float c) {...}
  //métodos básicos para asignar y obtener valores de atributos
  public void setNif(String n) {...}
  public String getNif() {...}
  public void setNombre(String n) {...}
  public String getNombre() {...}
  public void setEmail(String e) {...}
  public String getEmail() {...}
  public void setCache(float c) {...}
  public float getCache() {...}
  @Override
  //comportamiento del método toString heredado de la superclase Objet
  //Devuelve los atributos de un objeto Ponente
  public String toString() {
    if (this.cache != -1) {
      return this. nif+" "+this.nombre+" "+this.email+" Caché:"+this.cache;
    } else {
      return this.nif+" "+this.nombre +" "+this.email;
    }
  }
}
```

- La ejecución del proyecto, en nuestro caso, creará un fichero de nombre **congreso.db4o** en el raíz de la carpeta de nuestro proyecto **Congreso**, que es la BDOO creada y que contiene los datos de 4 objetos tipo ponente.



4.2 Consultas a la base de objetos

A una BDOO se podrán realizar consultas mediante:

- Un lenguaje de consultas como OQL, si el gestor está basado en el estandar ODMG e incluye sentencias del tipo SQL.
- El API proporcionado por el propio sistema gestor de bases de datos orientadas a objetos.

Los tres **sistemas de consulta que proporciona Db4o basados en el API del propio gestor,** son los siguientes:

- **Consultas por ejemplo. Query By Example (QBE**). Es la forma más sencilla y básica de realizar cosultas, pero tienen bastantes limitaciones.
- **Consultas nativas. Native Queries** (**NQ**). Es la interface principal de consultas de la base de objetos. Permiten realizar un filtro contra todas las instancias de la base de objetos.
- **Consultas SODA. Simple Object Data Access** (**SODA**). Permite generar consultas dinámicas. Es más potente que las anteriores y más rápida puesto que las anteriores (QBE y NQ) tienen que ser traducidas a SODA para ejecutarse.

¿**En qué consiste cada uno de estos sistemas de consulta**?

Consultas QBE

Las consultas **QBE (Query By Example)** se basan en suministrar a db4o un objeto que sirva de plantilla o prototipo de consulta.

- Mediante el método queryByExample() de la interface ObjectContainer Db4o retornará todos los objetos que coincidan con la plantilla.
- El resultado será una instancia de un ObjectSet.

Limitaciones:

- Hay que **proporcionar un ejemplo**, esto es, un objeto prototipo.
- **No se pueden realizar expresiones avanzadas**, como por ejemplo usar los operadores AND, OR y NOT.
- No se puede preguntar por ciertos objetos, en concreto aquellos cuyo valor de un campo numérico sea 0, strings vacíos o algún campo que sea null. (Éstos son los valores que se utilizan en el prototipo para seleccionar cualquier objeto)
- **Se necesita un constructor** para crear los objetos con valores no inicializados

EJEMPLO 1

Para recuperar todos los objetos, tipo ponente, de la base de objetos Congreso, mediante QBE, habrá que pasar un objeto prototipo vacío al método queryByExample(), lo que indica al sistema que se deben recuperar todos los objetos.

```
//consulta de todos los ponentes
public static void consultarPonentes(ObjectContainer db) {
  Ponente p = new Ponente(null, null, null, 0); //prototipo de búsqueda
  ObjectSet res = db.queryByExample(p); //realización de consulta
  mostrarConsulta(res);//obtención de resultados
}
```

Donde mostrarConsulta() es un método el cual tomará como parámetro un ObjectSet (conjunto de objetos) que irá recorriéndolo y mostrando uno a uno los objetos recuperados. El código del método es el siguiente

```
//Método para mostrar objetos recuperados de la Base de Objetos
public static void mostrarConsulta(ObjectSet resul) {
  System.out.println("Recuperados " + resul.size() + " Objetos");
  while (resul.hasNext()) {
    System.out.println(resul.next());
  }
}
```

EJEMPLO 2

Para consultar un ponente o ponentes en concreto, por ejemplo el o los de cierto caché, por ejemplo caché 200, se le pasa a queryByExample() el objeto prototipo con valor 200 en el campo caché.

```
//consulta de un Ponente en concreto. Consultar ponentes de cache 200.
public static void consultarPonente200(ObjectContainer db) {
  Ponente p = new Ponente(null, null, null, 200);
  ObjectSet res = db.queryByExample(p);
  mostrarConsulta(res);
}
```

EJEMPLO 3

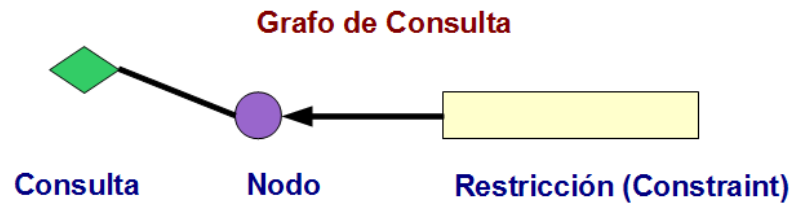
Para consultar un ponente o ponentes en concreto, por nombre, se le pasa a queryByExample() el objeto prototipo con parámetro nomb que le indicará el nombre.

```
//consulta de ponentes por nombre. Al método se le pasa el parámetro nomb
public static void consultarPonentePorNombre(ObjectContainer db, String nomb){
  Ponente p = new Ponente(null, nomb, null, 0);//prototipo de búsqueda
  ObjectSet res = db.queryByExample(p);
  mostrarConsulta(res);
}
```

Consultas SODA

Las **consultas SODA** utilizan la API SODA (Simple Object Data Access) de db4o para realizar consultas, permitiendo redactar consultas en tiempo de ejecución.

- La consulta se representa como un grafo con nodos.
- La clase Query es la interfaz del **query-graph**. Se necesitará un objeto tipo Query
- constrain() agrega una condición de restricción o Constraint a un nodo.
- descend() mueve de un nodo a otro (navega por los nodos)



- Se necesita el paquete com.db4o.query
- Al objeto Query creado se le añaden las Constraints que modelan la consulta que se desea obtener.
- Una vez construída la consulta, ésta se ejecuta mediante el método execute() del objeto tipo Query.
- Son las **consultas más rápidas y potentes**, ya que db4o transforma cualquier consulta en una consulta SODA para ser ejecutada.

EJEMPLO 1

Para consultar todos los ponentes, la única restricción es seleccionar la clase *ponente*, mediante query.constrain(ponente.class).

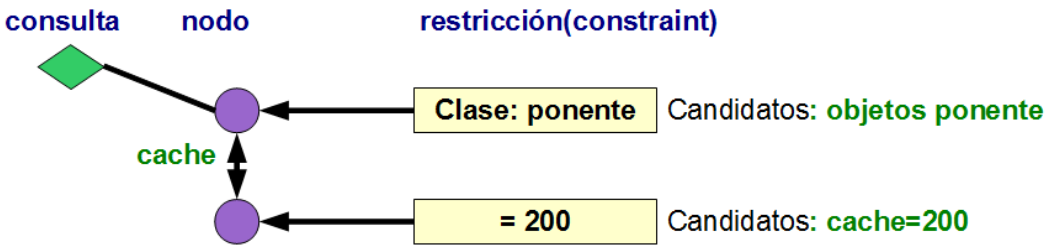
```
//Método para consultar todos los ponentes
public static void consultaSODAponentes(ObjectContainer db) {
  Query query = db.query();//declara un objeto Query
  //indica la clase a la que se aplicarán restricciones
  query.constrain(Ponente.class);
  ObjectSet result = query.execute(); //Ejecuta la consulta
  mostrarConsulta(result);//muestra los resultados de la consulta
}
```

Donde mostrarConsulta() es un método el cual tomará como parámetro un ObjectSet (conjunto de objetos) que irá recorriéndolo y mostrando uno a uno los objetos recuperados. El código del método es el siguiente:

```
//Método para mostrar objetos recuperados de la Base de Objetos
public static void mostrarConsulta(ObjectSet resul) {
  System.out.println("Recuperados " + resul.size() + " Objetos");
  while (resul.hasNext()) {
    System.out.println(resul.next());
  }
}
```

EJEMPLO 2

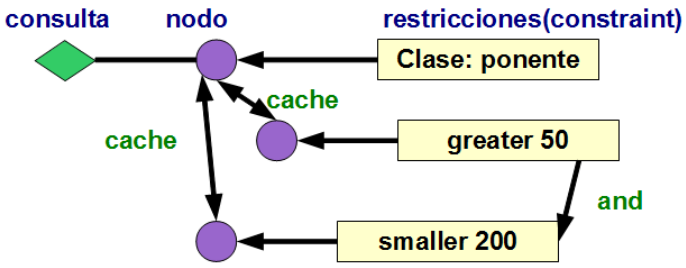
Para consultar ponentes de caché 200, se indicará la restricción constrain(200). Se construirá una consulta con una serie de nodos en los que se irá descendiendo para comprobar las restricciones o *constraints* establecidos y así incluir o excluir los posibles candidatos de ese nodo.




```
//Consulta SODA de todos los ponentes con caché 200
public static void consultaSODAcache200(ObjectContainer db) {
  Query query = db.query(); //se declara objeto tipo Query()
  query.constrain(Ponente.class); //clase a la que se aplican restricciones
  //establece restricción del valor 200 para cache
  query.descend("cache").constrain(200);
  ObjectSet result = query.execute(); //ejecuta consulta
  mostrarConsulta(result);//muestra resultados de consulta
}
```

EJEMPLO 3

Para consultar los ponentes con caché entre 50 y 200, habrá que enlazar dos restricciones. Para ello se necesitrá declarar un objeto Constraint que haga una de las restricciones y después enlazar con la otra restricción.




```
//consulta SODA de ponentes con cache entre 50 y 200
public static void consultaSODAcacheEntre50_200(ObjectContainer db) {
  Query query = db.query();
  query.constrain(Ponente.class);
  //se declara una de las restricciones con Constraint
  Constraint constra1 = query.descend("cache").constrain(200).smaller();
  //se enlazan las dos restricciones a aplicar
  query.descend("cache").constrain(50).greater().and(constra1);
  ObjectSet result = query.execute();
  mostrarConsulta(result);
}
```

Consultas Nativas- NQ

Las **consultas NQ (Native Query)** son la interfaz principal de consultas y se basan en el uso de la semántica del lenguaje de programación.

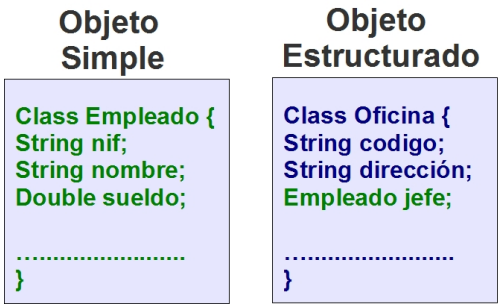
- Permiten realizar un filtro contra todas las instancias de una clase.
- Deberán retornar 'verdadero' para incluir determinadas instancias dentro del conjunto de resultados.
- Db4o realiza una optimización de las expresiones utilizadas mediante un procesador de consultas que intenta resolverlas usando índices, sin necesidad de instanciar los objetos reales donde sea posible.

EJEMPLO 1

Para consultar los ponentes con un caché igual a 200, mediante una consulta Nativa NQ, escribiríamos el siguiente código.

```
//consultar NQ los ponentes de caché 200
public static void consultarPonenteNQcache200(ObjectContainer db) {
  List res = db.query(new com.db4o.query.Predicate() {
    public boolean match(Ponente p) {
      return p.getCache() == 200;
    }
    //método abstracto
    @Override
    public boolean match(Object et) {
      throw new UnsupportedOperationException("Not supported yet.");
    }
  });
  mostrarConsulta((ObjectSet) res);
}
```

4.3 Objetos simples y objetos estructurados



En las BDOO los objetos se encuentran interrelacionados por referencias entre ellos de manera similar a como los objetos se referencian entre sí en memoria.

Un **objeto de tipo simple** u objeto simple es aquel que no contiene a otros objetos y por tanto posee una estructura de un solo nivel de profundidad en este sentido.

Un **objeto de tipo estructurado** u objeto estructurado incluye entre sus componentes a otros objetos y se define aplicando los constructores de tipos disponibles por el SGBDOO recursivamente a varios niveles de profundidad.

Entre un objeto y sus componentes de cada nivel, existen dos tipos de referencia:

- **Referencia de propiedad**. Se aplica cuando los componentes de un objeto se encapsulan dentro del propio objeto y se consideran, por tanto, parte de ese objeto. **Relación es-parte-de**. No necesitan tener identificadores de objeto y sólo los métodos de ese objeto pueden acceder a ellos. Desaparecen si el propio objeto se elimina.
- **Referencia de asociación**. Se aplica cuando entre los componentes del objeto estructurado existen objetos independientes, pero es posible hacer referencia a ellos desde el objeto estructurado. **Relación está-asociado-con**. Cuando un objeto estructurado tiene que acceder a sus componentes referenciados, lo hace invocando los métodos apropiados de los componentes, ya que no están encapsulados dentro del objeto estructurado.

Por ejemplo, si observas la figura superior derecha, en un objeto tipo Oficina los componentes codigo y dirección son_parte_del objeto, mientras que el componente jefe, es un objeto independiente que está_asociado_con el objeto oficina.

Entonces, ¿**las referencias de asociación son como las relaciones del modelo relacional**? Así es:

- La **referencia de asociación** representa las relaciones o interrelaciones entre objetos independientes, dando la posibilidad de que los objetos puedan detectarse mutuamente en una o dos direcciones, (lo que en el modelo relacional representamos mediante claves ajenas o foráneas que provienen de relaciones uno a uno, uno a muchos y muchos a muchos).
- Una **relación uno a muchos** se representa mediante un objeto tipo colección (List, Set, etc.). En una BDOO la colección se maneja como cualquier otro objeto (aunque potencialmente profundo) y normalmente se podrá recuperar y almacenar el objeto padre junto con la colección asociada y su contenido en una sola llamada.

Además, un objeto miembro referenciado puede ser referenciado por más de un objeto estructurado y, no se elimina automáticamente cuando se elimina el objeto del nivel superior.

4.4 Actualización de objetos simples

Recuerda que un **objeto simple** es un objeto que no contiene a otros objetos, como por ejemplo los objetos de la clase Ponente. Un segmento de la definición de esta clase, es la siguiente:

```
public class Ponente {
  private String nif;
  private String nombre;
  private String email;
  private float cache;
  //constructores
  public Ponente() {...}
```

Para **consultar objetos simples** se pueden utilizar cualquiera de los tres sistemas de consulta proporcionados por Db4o, tal y como has podido ver en el apartado anterior.

Para **modificar objetos** almacenados debes seguir los siguientes pasos:

- Cambiar los valores del objeto con los nuevos valores.
- Almacenar de nuevo el objeto con el método store() de la interfaz ObjectContainer.

Por ejemplo, el siguiente método permitirá modificar objetos ponente, en concreto actualizar el e-mail de ponentes por nif de ponente:

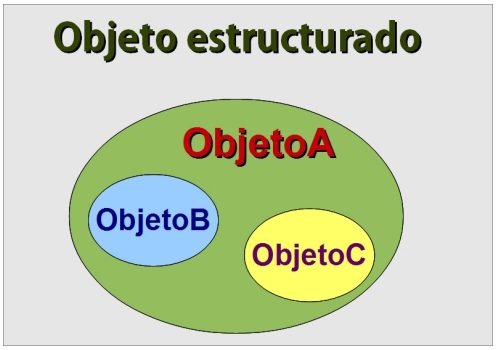
```
//método que modifica el e-mail de un Ponente cuyo nif se pasa como parámetro
// y almacena en la base de objetos los nuevos valores
public static void actualizarEmailPonente(ObjectContainer db, String nif, String em) {
  //se consulta a la base de objetos por el Ponente del nif indicado
  ObjectSet res = db.queryByExample(new Ponente(nif, null, null, 0));
  Ponente p = (Ponente) res.next(); //se obtiene el objeto consultado en p
  p.setEmail(em); //se cambia el email del objeto
  db.store(p); //se alamcena de nuevo el objeto poenente
}
```

DESTACADO
Db4o necesita conocer previamente un objeto para poder actualizarlo. Esto significa que para poder ser actualizados los objetos, éstos deben de haber sido insertados o recuperados en la misma sesión; en otro caso se añadirá otro objeto en vez de actualizarse.

Para **eliminar objetos** almacenados utilizaremos el método delete() de la interface ObjectContainer .
Por ejemplo, el siguiente método elimina un objeto ponente por su nif:

```
// Método que elimina de la base de objetos [con delete()] el Ponente cuyo
//nif se pasa como parámetro
public static void borrarPonenteporNif(ObjectContainer db, String nif) {
  //se consulta a la base de objetos por el Ponente del nif indicado
  ObjectSet res = db.queryByExample(new Ponente(nif, null, null, 0));
  Ponente p = (Ponente) res.next(); //se obtiene el objeto consultado en p
  db.delete(p); //se elimina el objeto poenente de la base de objetos
}
```

4.5 Actualización de objetos estructurados



Los **objetos estructurados** son objetos que contienen a su vez a otros objetos (objetos hijo u objetos miembro).

En el caso de objetos estructurados se habla de diferentes **niveles de profundidad del objeto.** El nivel más alto, nivel 1, será el que corresponde a la definición del objeto estrucutrado (objeto padre), el siguiente nivel, nivel 2, corresponderá a la definición del objeto hijo y así sucesivamente podrá haber un nivel 3, 4... dependiendo de que los objetos hijos a su vez incluyan en su definición a otro u otros objetos miembro.

En el siguiente ejemplo, definimos la clase charla (objeto estructurado padre) que incorpora a un objeto ponente (objeto miembro). El nivel más alto de profundidad o nivel 1 es el que corresponde a la definición de charla y el nivel 2 corresponderá a la definición del objeto ponente.

```
//clase que implementa un Charla. Cada Charla tiene un título y está asignada
//a un Ponente. CLASE ESTRUCTURADA: contiene un objeto Ponente
public class Charla {
  private String titulo;
  private float duracion;
  private Ponente pl;
  //constructor
  public Charla(String ti, float h) {
    this.titulo = ti;
    this.pl = null;
    this.duracion=h;
  }
//Método para obtener el Ponente de una Charla
  public Ponente getPonente() {
    return pl;
  }
//Método para asignar el Ponente de una Charla
  public void setPonente(Ponente p) {
    this.pl = p;
  }
```

¿**Cómo se almacenan, consultan y actualizan los objetos estructurados en Db4o**?

- Los objetos estructurados se almacenan asignando valores con set() y después persistiendo el objeto con store(). Al almacenar un objeto estructurado del nivel más alto, se almacenarán de forma implícita todos los objetos hijo.
- Las consultas se realizan por cualquiera de los sistemas soportados por el gestor y se podrá ir descendiendo por los diferentes niveles de profundidad.
- La eliminación o borrado de un objeto estructurado se realiza mediante el método delete(). Por defecto, no se eliminarán los objetos miembro. Para eliminar objetos estructurados en cascada o de forma recursiva, eliminando los objetos miembro, habrá que configurar de modo apropiado la base de objetos antes de abrirla, mediante el paquete com.db4o.config. En el caso de modo embebido, se hará mediante la interface EmbeddedConfiguration. En la nueva configuración se debe indicar cascadeOnDelete(true).
- La modificación se realizará actualizando los nuevos valores mediante el método set(). Por defecto las modificaciones solo afectan al nivel más alto. Para actualizar de forma recursiva todos los objeto miembro habrá que indicar en la configuración cascadeOnUpdate(true).

EJEMPLOS. [ActualizarCongresoEstructurados.zip]

Clase estructurada

```
//clase que implementa un Charla.
//a un Ponente. CLASE ESTRUCTURADA
public class Charla {
  private String titulo;
  private float duracion;
  private Ponente pl;
```

```
public class Ponente {
  private String nif;
  private String nombre;
  private String email;
  private float cache;
  //constructores
  public Ponente() {...}
```

- Al almacenar un objeto padre (nivel 1), se almacenan automáticamente los objetos hijo (nivel 2)

```
  //Almacenar objetos estructurados
  //Método para insertar charlas en la Base de Objetos y almacenarlas
  public static void almacenarCharlas(ObjectContainer db) {
    //se crean 4 objetos tipo Charla
    Charla c1 = new Charla("Bases de Datos Orientadas a Objetos", 2);
    Charla c2 = new Charla("MySQL y PostGreSQL", 3);
    Charla c3 = new Charla("XML", 2);
    Charla c4 = new Charla("Db4o", 3);
    //se crean 4 objetos Ponente
    Ponente p1 = new Ponente("11A", "Antonio Camaco", "acamacho@gmail.es", 300);
    Ponente p2 = new Ponente("22B", "Isabel Pérez", "iperez@hotmail.es", 100);
    Ponente p3 = new Ponente("33C", "Ana Navarro", "anavarro@yahoo.com", 200);
    //se le asigna un Ponente a cada Charla
    c1.setPonente(p1);
    c2.setPonente(p2);
    c3.setPonente(p3);
    c4.setPonente(p1);
//Persistir Objetos: almacenamos los objetos con el método store()
    db.store(c1);
    db.store(c2);
    db.store(c3);
    db.store(c4);
  }
```

- Método que muestra objetos recuperados de una consulta

```
  //Método para mostrar objetos recuperados de la Base de Objetos
  public static void mostrarConsulta(ObjectSet resul) {
    //mensaje indicando el total de objetos recuperados
    System.out.println("Recuperados " + resul.size() + " Objetos");
    while (resul.hasNext()) {//bucle que obtiene objeto a objeto
      System.out.println(resul.next());
    }
```

- **Consulta QBE** de un objeto estructurado. Consulta de todas las charlas

```
//Consulta QBE de objetos estructurados. Consulta de todas las charlas.
  public static void mostrarCharlasQBE(ObjectContainer db) {
    //se crea el objeto Charla patrón de búsqueda
    Charla c = new Charla(null, 0);
    //Consulta las charlas con patrones indicados
    ObjectSet resul = db.queryByExample(c);
    mostrarConsulta(resul);//método que muestra los objetos recuperados de BDOO
  }
```

- **Consulta QBE** objeto estructurado. Consulta de charlas de cierto ponente

```
//Consulta de las charlas del ponenete Anotnio Camaco
public static void mostrarCharlasCamacoQBE(ObjectContainer db) {
  //se crea objeto Ponente con patrón de búsqueda (el ejemplo)
  Ponente p = new Ponente(null, "Antonio Camaco", null, 0);
  //se crea el objeto Charla con patrón de búsqueda
  Charla c = new Charla(null, 0);
  c.setPonente(p); //se asocia el Ponente de búsqueda a la Charla
  //Consulta las charlas con patrones indicados
  ObjectSet resul = db.queryByExample(c);
  mostrarConsulta(resul);//método que muestra los objetos recuperados de BDOO
}
```

- **Consulta SODA** de un objeto estructurado. Se consulta la charla de un determinado título.

```
//Consulta SODA de objetos estructurados
//Se consulta la Charla cuyo título se pasa en parámetro tit
public static void consultaSODACharla_concreta(ObjectContainer db,String tit){
  Query query = db.query(); //declara un objeto query
  //establece la clase a la que se aplicará la restricción
  query.constrain(Charla.class);
  query.descend("titulo").constrain(tit);//establece restricción de búsqueda
  ObjectSet resul = query.execute();//ejecuta consulta
  mostrarConsulta(resul);//muestra los objetos recuperados de la BDOO
}
```

NOTA.

Para evitar en los ejemplos que se carguen de nuevo los datos en cada ejecución, puedes poner al principio

```
public static void main(String[] args) {
  //borra fichero con BD antes de comenzar
  new File("congreso.db4o").delete();
```

De esta forma la BD siempre se inicializa al principio de la ejecución y no conserva los datos de ejecuciones anteriores

Suponiendo la siguiente apertura a la base de objetos, con una configuración básica o por defecto, no se realizarán operaciones de actualización en cascada

```
//Conexión a la base de objetos y apertura de la base de objetos congreso.db4o
  ObjectContainer db = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),
          "congreso.db4o");
```

- **Eliminación** de objetos estructurados

```
//Borrado de objetos estructurados. Se utiliza Consulta SODA
//Se elimina la Charla de título tit sin eliminar al Ponente asociado
public static void borrarCharlaporTitulo(ObjectContainer db, String tit) {
  Query query = db.query(); //declaración de un objeto query().
  //establece la clase a la que se aplicará la restricción
  query.constrain(Charla.class);
  //establece la restricción de búsqueda
  query.descend("titulo").constrain(tit);
  //ejecuta consulta con restricción búsqueda
  ObjectSet resul = query.execute();
  //bucle que recupera los objetos Charla y los elimina de la BDOO
  while (resul.hasNext()) {
    Charla c = (Charla) resul.next();
    System.out.println("Eliminando: " + c);
    db.delete(c);
  }
```

- **Modificación** Objeto estructurado. Cambia la duración de una charla

```
//Modificación de Objetos estructurados. Con consulta QBE
//Actualiza la duración de la Charla de título tit en d horas
public static void actualizarHorasCharla(ObjectContainer db,
        String tit, float d) {
  //consulta la Charla de patrón Charla(tit,0). Consulta QBE
  ObjectSet res = db.queryByExample(new Charla(tit, 0));
  Charla c = (Charla) res.next(); //obtiene la Charla consultada
  c.setDuracion(d); //asigna la nueva duración
  db.store(c); //almacena la Charla modificada
}
```

- Para realizar **eliminación en cascada** de charlas, la conexión y apertura a la base de objetos sería la siguiente:

```
//Se indica la configuración de conexión, con borrado y modificación en cascada
    EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
    config.common().objectClass(charla.class).cascadeOnDelete(true);
    config.common().objectClass(charla.class).cascadeOnUpdate(true);
//Se abre la conexión a la base de objetos congreso.db4o
    ObjectContainer db = Db4oEmbedded.openFile(config, "congreso.db4o");
```

.

- **Eliminación en cascada** de una charla y ponente asociado

```
//Borrado en cascada de objetos estructurados. Se utiliza Consulta SODA
//Se elimina la charla de título tit y el ponente asociado
public static void borrarCharlaporTitulo(ObjectContainer db, String tit) {
  Query query = db.query(); //declaración de un objeto query().
  query.constrain(charla.class);//clase a la que se aplicará la restricción
  query.descend("titulo").constrain(tit);//restricción de búsqueda
  ObjectSet resul = query.execute();//ejecuta consulta

  charla c=null;
  if(resul.hasNext()){ //si se obtiene resultado
    c = (charla) resul.next(); //recuperamos la charla
    System.out.println("Eliminando: " + c);
    db.delete(c); //elimina la charla y el poneente asociado de la BDOO
  }else
    System.out.println("No existe charla: " + tit);
}
```

- **Modificación en cascada** de una charla y ponente asociado

```
//Modificación de Objetos estructurados. Con consulta QBE
//Actualiza la duración de la Charla de título tit en d horas
public static void actualizarHorasCharlayCachePonente(ObjectContainer db,
        String tit, float d, float ca) {
  //consulta la Charla de patrón Charla(tit,0). Consulta QBE
  ObjectSet res = db.queryByExample(new charla(tit, 0));
  charla c = (charla) res.next(); //obtiene la Charla consultada
  c.setDuracion(d); //asigna la nueva duración
  c.getPonente().setCache(ca);
  db.store(c); //almacena la Charla modificada y el caché del ponente
}
```

5 El lenguaje de consulta de objetos OQL

OQL (Object Query Languaje) es el lenguaje de consulta de objetos propuesto en el estándar ODMG.

Las siguientes son algunas de las c**aracterísticas más relevantes de OQL**:

- Es un **lenguaje declarativo** del tipo de SQL que permite realizar consultas de modo eficiente sobre bases de datos orientadas a objetos.
- Su **sintaxis es similar a la de SQL**, proporcionando un superconjunto de la sintaxis de la sentencia SELECT, con algunas características añadidas para los conceptos ODMG, como la identidad del objeto, los objetos complejos, las operaciones, la herencia, el polimorfismo y las relaciones.
- **No posee primitivas para modificar el estado de los objetos** ya que las modificaciones se pueden realizar mediante los métodos que estos poseen.
- Puede ser usado como un **lenguaje autónomo o incrustado** dentro de otros lenguajes como C++, Smalltalk y Java.
- Una **consulta OQL incrustada** en uno de estos lenguajes de programación puede devolver objetos que coincidan con el sistema de tipos de ese lenguaje.
- Desde OQL se pueden invocar operaciones escritas en estos lenguajes.
- Permite **acceso tanto asociativo como navegacional**:
  - Una **consulta asociativa** devuelve una colección de objetos.
  - Una **consulta navegacional** accede a objetos individuales y las interrelaciones entre objetos sirven para navegar entre objetos.

PARA SABER MÁS

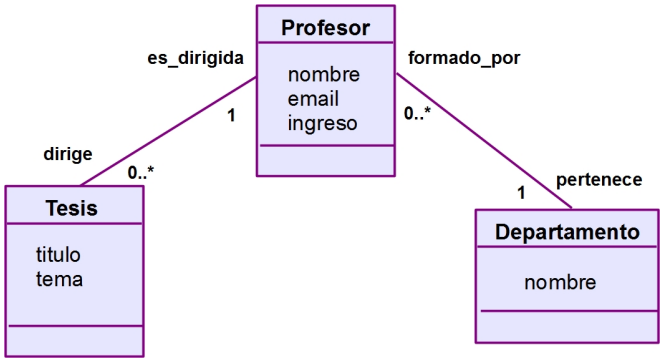
En el siguiente enlace puedes ampliar información sobre el estándar ODMG y en particular sobre el lenguaje OQL, así como ver diferentes ejemplos de consultas en la propuesta del estándar.

Estándar ODMG
http://alarcos.inf-cr.uclm.es/doc/bbddavanzadas/08-09/ODMG.pdf

5.1 Sintaxis, expresiones y operadores

La sintaxis básica y resumida de una sentencia SELECT del OQL estándar es la siguien te:

SELECT [DISTINCT] <expresión, ...>
FROM <lista from>
[WHERE <condición>]
[ORDER BY <expresión>]



Por ejemplo, suponiendo el esquema de base de objetos que puedes ver en la figura ampliable de la derecha, la siguiente sentencia select recupera de la base de objetos los atributos nombre y el correo de objetos tipo profesor cuyo año de ingreso es anterior al 1990, y ordenados alfabéticamente por nombre:

SELECT p.nombre, p.email FROM Profesor p WHERE p.ingreso <= 1990 ORDER BY p.nombre;

Las siguientes, son algunas **consideraciones a tener en cuenta**:

- En las consultas se necesita un punto de entrada, que suele ser el nombre de una clase.
- El resultado de una consulta es una colección que puede ser tipo bag (si hay valores repetidos) o tipo set (no hay valores repetidos). En este último caso habrá que especificar SELECT DISTINCT.
- En general, una consulta OQL puede devolver un resultado con una estructura compleja especificada en la misma consulta utilizando struct.
- Una vez que se establece un punto de entrada, se pueden utilizar expresiones de caminos para especificar un camino a atributos y objetos relacionados. Una **expresión de camino** empieza normalmente con un nombre de objeto persistente o una variable iterador, seguida de ninguno o varios nombres de relaciones o de atributos conectados mediante un punto.
- Es posible crear objetos mutables (no literales) formados por el resultado de una consulta.

Además, en una consulta OQL se pueden utilizar, entre otros, los siguientes **operadores y expresiones:**

- Operadores de acceso: "." / "->" aplicados a un atributo, una expresión o una relación. FIRST / LAST (primero / último elemento de una lista o un vector).
- Operadores aritméticos: +, -, *, /, -(unario), MOD, ABS para formar expresiones aritméticas.
- Operadores relacionales: >, <, >=, <=, <>, = que permiten comparaciones y construir expresiones lógicas.
- Operadores lógicos: NOT, AND, OR que permiten enlazar otras expresiones.

A continuación te indicamos de manera resumida, **algunas otras caracterísiticas**, del estándar OQL:

- Definición de vistas, es decir, es posible dar nombre a una consulta y utilizarlo en otras consultas.
- Extracción de elementos sencillos de colecciones set, bag, o list.
- Operadores de colecciones como funciones de agregacións (MAX(), MIN(), COUNT(), SUM() y AVG()) y cuantificadores (FOR ALL, EXISTS).
- Realización de agrupaciones mediante GROUP BY y filtro de los grupos mediante HAVING.
- Combinación de consultas mediante JOINs
- Unión, intersección y resta de colecciones mediante los operadores UNION, INTERSEC y EXCEPT.

En los siguientes apartados nos centraremos en el OQL concreto de un SGBDOO y que por supuesto, incorpora sus propias particularidades y diferencias respecto a la propuesta OQL de ODMG.

PARA SABER MÁS
En el siguiente enlace puedes consultar una extensa guía de referencia y con ejemplos, en inglés, del lenguaje OQL propuesto por el ODMG.

Guía de referencia del lenguaje OQL
http://tech.novosoft-us.com/products/oql_book.htm

5.2 Matisse, un gestor de objetos que incorpora OQL.

Para practicar y trabajar con OQL utilizaremos Matisse, un gestor orientado a objetos que incorpora características del estándar ODMG, como los lenguajes ODL y OQL, y que tiene soporte para Java. Aunque Matisse llama SQL a su lenguaje de consultas, nosotros nos referiremos a él como OQL.

El propio gestor proporciona el driver matisse.jar para interactuar con aplicaciones escritas en Java.

Dentro de los **paquetes del API** destacamos:

- com.matisse. Proporciona las clases e interfaces básicos para trabajar con Java y una base de datos de objetos Matisse.
- MtDatabase. Clase que proporciona todos los métodos para realizar las conexiones y transacciones en la base de objetos.
- com.matisse.sql. Proporciona clases que permiten interactuar con la base de objetos vía JDBC.

DESCATADO
Todas las interacciones entre una aplicación Java y la base de objetos Matisse se realizan en el contexto de una transacción (implícita o explícita).

Te puedes descargar Matisse desde su web oficial:

web oficial de Matisse
http://www.matisse.com/

Los pasos para descargar el software Matisse y realizar su instalación e integración en NetBeans a partir de del driver matisse.jar son los siguientes:

- Accede a la dirección: http://www.matisse.com/
- Pulsa en el botón Download. Habrá que registrarse para hacer la desacarga.
- Seleccionamos el ejecutable apropiado para windows, de 32 o 64 bits, según sea nuestro sistema y al hacer clic, descargará un archivo .exe.
- Una vez descargado, lo ejecutamos y un asistente nos guiará por la instalación, en la cual dejaremos los valores por defecto seleccionados y simplemente pulsaremos Next.

Para i**ntegrar Matisse en NetBeans**:

- Abrimos un proyecto y desde el menú contextual de la carpeta Bibliotecas seleccionamos la opción Agregar archivo JAR/Carpeta.
- Buscamos y elegimos el fichero comentado anteriormente, el .jar que nos interesa. Se pulsa Abrir y hemos acabado.

EJEMPLO. Creación de una base de objetos matisse en NetBeans
En el recurso didáctico **CrearBDooMatisse.pdf** verás cómo crear la base de objetos para practicar con OQL.

El proyecto completo de creación de una base de objetos en Matisse, así como el esquema ODL de dicha base de objetos lo tienes en **Crear_BO_Doctorado.zip**

PARA SABER MÁS
Desde el siguiente enlace podrás descargar numerosos manuales y abundante documentación sobre Matisse.

Documentación, manuales y guías de Matisse
http://www.matisse.com/developers/documentation/

5.3 Ejecución de sentencias OQL

La **sintaxis básica del OQL** de Matisse es una sentencia SELECT de la forma: **SELECT... FROM... WHERE**...;

Por ejemplo, para recuperar el valor de todos los atributos de los objetos tipo Profesor, escribiríamos la siguiente sentencia OQL:
SELECT * FROM Profesor;

Y para recuperar del atributo nombre de los objetos tipo Profesor cuyo año de ingreso es anterior al 1990, escribiríamos la siguiente sentencia :
SELECT nombre FROM Profesor WHERE ingreso <= 1990;

Las siguientes son algunas **consideraciones y ejemplos sobre las consultas con SELECT**:

- Toda sentencia SELECT finaliza en punto y coma.

- Además de la cláusula WHERE, que permite establecer un criterio de selección se pueden utilizar las cláusulas de agrupamiento GROUP BY, y de ordenación ORDER BY, entre otras. También se pueden asignar alias mediante AS, realizar búsquedas por patrones de caracteres con LIKE.
  Ejemplo: título y tema de las tesis cuyo tema contien la palabra Objeto, ordenadas por tema.
  SELECT titulo AS "Tesis", tema FROM Tesis WHERE tema LIKE '%Objeto%' ORDER BY tema;

- Al recupear todos los atributos de una clase mediante SELECT *, la consulta retornará el OID de cada objeto recuperado, así como las interrelaciones o relaciones definidas para esa clase. El OID y la interrelación son del tipo string y se representan mediante un número hexadecimal. Realmente el identificador recuperado para la interrelación, hace referencia al primer objeto relacionado, incluso aunque la interrelación incluya a más de un objeto.

- Se pueden hacer JOIN de clases, y por ejemplo esto puede permitir obtener todos los objetos relacionados con otro objeto en una ***consulta asociativa***.
  Ejemplo: nombre de cada profesor y título y tema de las tesis que dirige con JOIN
  SELECT p.nombre, t.titulo, t.tema FROM Profesor p JOIN Tesis t ON t.es_dirigida = p.OID;

- Mediante SELECT se pueden realizar ***consultas navegacionales*** haciendo referencia a las interrelacionaes entre objetos.
  Ejemplo: nombre de cada profesor y título y tema de las tesis que dirige, navegacional
  SELECT t.titulo AS "Tesis", tema, t.es_dirigida.nombre AS "Profesor " FROM Tesis t;

- También se pueden realizar ***consultas navegacionales*** a través de la referencia de los objetos (REF()).
  Ejemplo: tesis que dirigen los profesores con ingreso el 1990 o posterior
  SELECT REF(p.dirige) FROM Profesor p WHERE p.ingreso >= 1990;

EJEMPLO. En el recurso didáctico **Consultas_OQL_Matisse.pdf** encontrarás diferentes ejemplos de consultas utilizando de forma autónoma o conversacional el lenguaje OQL de Matisse. Se explica como realizar consultas en Matisse mediante el SQL Query Analyzer y se ven diferentes ejemplos de consultas OQL realizadas a la base de objetos doctorado.

PARA SABER MÁS
En el siguiente enlace dispones de la guía completa, en inglés, sobre el OQL de Matisse:

Guía oficial del OQL de Matisse
http://www.matisse.com/pdf/developers/sql_pg.pdf

5.4 Ejecución de sentencias OQL vía JDBC

Veremos ahora cómo realizar consultas a una base de objetos Matisse mediante sentencias OQL embebidas en Java. Para ello, la conexión a la base de objetos se realizará vía JDBC. Matisse proporciona dos formas de manipular objetos vía JDBC: mediante JDB puro o mediante una mezcla de JDBC y Matisse.

1. Para crear una conexión vía JDBC puro utilizaremos java.sql.* y com.matisse.sql.MtDriver, no siendo necesario en este caso el paquete com.matisse.
   - La cadena de conexión será de la forma: String url = "jdbc:mt://" + hostname + "/" + dbname;
   - La conexión se realizará mediante Connection jcon = DriverManager.getConnection(url);

2. Para crear una conexión vía JDBC y Matisse (a través de MtDatabase) se necesita java.sql.* y com.matisse.MtDatabase.

- La cadena de conexión será de la forma: MtDatabase db = new MtDatabase(hostname, dbname);
- La conexión se realizará mediante: db.open(); y Connection jcon = db.getJDBCConnection();

Una vez realizada la conexión por uno u otro método, ya podremos ejecutar sentencias OQL vía JDBC, (tal y como si de una sentencia SQL se tratara) en la aplicación java. El esquema de consulta es el siguiente:

```
Statement stmt = jcon.createStatement();
String query = "SELECT atributo FROM Clase";
ResultSet rs = stmt.executeQuery(query);
System.out.println("Resultado de la consulta: " + query);
while (rs.next()) {
System.out.println(rs.getString("nombre"));
}
```

EJEMPLO. **ConsultasOQL_JDBC_Matisse.zip**. Este proyecto contiene ejemplos de consultas OQL vía JDBC, que puedes probar en tu equipo.

CITA PARA PENSAR
Blaise Pascal: "Vale más saber alguna cosa de todo, que saberlo todo de una sola cosa".

PARA SABER MÁS
En el siguiente enlace puedes consultar la Guía Matisse para programadores Java y ver diferentes ejemplos de consultas OQL y en modo nativo.

Guía oficial de Matisse para programadores Java
http://www.matisse.com/pdf/developers/java_pg.pdf

6 Características de las bases de datos objeto-relacionales

Las **bases de datos objeto-relacionales, BDOR,** son una extensión de las bases de datos relacionales tradicionales a las que se les ha añadido conceptos del modelo orientado a objetos. Las podemos ver como un híbrido de las BDR y las BDOO que intenta aunar los beneficios de ambos modelos.

- Las **clases, objetos,** y **herencia** son directamente soportados en los esquemas de la base de datos y el lenguaje de consulta y manipulación de datos.
- Además dan soporte a una extensión del modelo de datos con la creación personalizada de **tipos de datos y métodos.**

El modelo objeto-relacional ofrece las **ventajas** de las técnicas orientadas a objetos en cuanto a mejorar la reutilización y el uso intuitivo de los objetos, a la vez que se mantiene la alta capacidad de concurrencia y el rendimiento de las bases de datos relacionales.

Por debajo de la capa de objetos, **los datos seguirán estando almacenados en tablas**, pero se puede trabajar con ellos de manera más parecida a las entidades de la vida real, dando así más significado a los datos. En vez de pensar en términos de columnas y tablas cuando realices consultas a la base de datos, simplemente deberás seleccionar entidades que habrás creado, por ejemplo clientes o pedidos.

En una BDOR las columnas de las tablas no están restringidas a contener escalares o valores atómicos, sino que **una columna puede almacenar tipos estructurados** (objetos o colecciones).

Y eso, ¿cómo es posible?. Pues porque internamente tanto las tablas como las columnas son tratados como objetos, esto es, se realiza un mapeo objeto-relacional de manera transparente.

Como consecuencia de incorporar conceptos del modelo orientado a objetos a las bases de datos realcionales, aparecen nuevas características en las BDOR, como son **un sistema de tipos más rico**, incluyendo tipos de datos complejos, objetos de gran tamaño, tipos definidos por el usuario, tablas en tablas, arrays, colecciones, etc; y **un lenguaje de consulta, SQL extendido**, que permite trabajar con estos nuevos tipos de datos.

En definitiva las **características más importantes de las BDOR** y por tanto de los sistemas gestores de bases de datos objeto-relacionales, SGBDOR, son las siguientes:

- **Soporte de tipos complejos.**
  - **Tipos definidos por el usuario**. (**UDT** – User Definedy Type).
  - **Tipos colección**
- **Reusabilidad**. Posibilidad de guardar esos tipos en el gestor de la BDOR, para reutilizarlos en tantas tablas como sea necesario.
- **Soporte para crear métodos o funciones**. Posibilidad de definir funciones y almacenarlas en el gestor. Las funciones pueden modelar el comportamiento de un tipo objeto, en este caso se llaman métodos.
- **Se pueden almacenar múliples valores en una columna de una misma fila.**
- **Tablas anidadas**. Se pueden definir columnas como arrays o vectores multidimensionales, tanto de tipos básicos como de tipos estructurados o colecciones, esto es, se pueden anidar tablas
- **Herencia** de tipos y tablas.
- **Compatibilidad con las bases de datos relacionales** tradicionales. Es decir, se pueden pasar las aplicaciones sobre bases de datos relacionales al nuevo modelo sin tener que rescribirlas.

Estas y otras características de las bases de datos objeto-relacionales vienen recogidas en el estándar SQL: 1999 y el estándard SQL 2003.

PARA SABER MÁS.
En este enlace dispones de una presentación que detalla las caracterísiticas de las BDOR
http://personales.unican.es/zorrillm/BDAvanzadas/Teoria/bda-t3-trans-ObjetoRelacional.pdf

6.1 El estándar SQL99

La **norma ANSI SQL: 1999** (abreviadamente, **SQL99**) extiende el estándar SQL92 de las Bases de Datos Relacionales, y da cabida a nuevas características orientadas a objetos preservando los fundamentos relacionales.

Algunas extensiones que contempla este estándar y que están relacionadas directamente con la orientación a objetos son las siguientes:

- **Extensión de tipos de datos.**
  - **Tipo estructurado definido por el usuario (UDT** – User Definedy Type).
    - Se defienen como un tipo de dato con un nombre dado y definido por el usuario diseñador de la BD. un tipo del interés del usuario y a la medida de ciertas aplicaciones
    - Se define mediante una sentencia DDL .
    - Cada UDT se define en términos de otros tipos del lenguaje SQL y/o otros tipos definidos previamente por el usuario. Por tanto, los atributos que componen al nuevo tipo de dato estructurado (UDT) pueden ser:
      - tipo básico (int, char, ..)
      - tipo compuesto por varios elementos del mismo tipo (int array(100))
      - tipo estructurado o UDT.
  - **Tipo colección** (ARRAYS, SET, BAG y LIST). Conjuntos de elementos.
  - **Tipo fila** (ROW) y **Tipo referencia** (REF). Un apuntador o referencia a un objeto.
  - **Tipo objeto grande**. Nuevos tipos de datos básicos para datos de caracteres de gran tamaño, y datos binarios de gran tamaño (Large Objects), como LOB
- **Extensión de la semántica de datos**
  - Disparadores o *triggers*
  - *Procedimientos y funciones* definidos por el usuario, y almacenados en el gestor de bases de datos.
  - Un tipo estructurado(UDT) puede tener métodos definidos sobre él. Un ***método*** es una función SQL ligada a un Tipo Definido por el Usuario
  - **Herencia**. De tablas y de tipos. La herencia de tipos incluye también métodos, es decir, tanto las propiedades como el comportamiento.

EJEMPLO.
El siguiente segmento SQL crea un nuevo tipo de dato, un tipo estructurado definido por el usuario (UDT) de nombre *profesor* y que incluye en su definición un método, el método *sueldo().*

- El método se define en el esquema y la signatura se define separada de la especificación del cuerpo.

```
CREATE TYPE profesor AS (
        id INTEGER,
        nombre VARCHAR (20),
        sueldo_base DECIMAL (9,2),
        complementos DECIMAL (9,2),
        INSTANTIABLE NOT FINAL
        METHOD sueldo() RETURNS DECIMAL (9,2)
);
CREATE METHOD sueldo() FOR profesor
BEGIN
.....
END;
```

PARA SABER MÁS. ***Extensiones que incorpora tanto la norma SQL99 como SQL2003***
http://sinbad.dit.upm.es/docencia/grado/curso1213/3%20Slices-%20BSDT%202012-13%20BDOR%20Parte%202.pdf

7 Gestores de Bases de Datos Objeto-Relacionales

Un Sistema Gestor de Bases de Datos Objeto-Relacional (SGBDOR) contiene dos tecnologías; la tecnología relacional y la tecnología de objetos, pero con ciertas restricciones.

A continuación te indicamos algunos ejemplos de **gestores de bases de datos objeto-relacionales**, tanto de código abierto como propietario, todos ellos con soporte para Java:

• De código abierto:
  • **PostgreSQL**
  • **Apache Derby**
	• De código propietario
	  • **Oracle**
	  • **MS SQL Server**
	  • **DB2 de IBM**
	  • **Informix**
	  • **Sybase**

Las normativas SQL99 y SQL2003 son el estándar base que siguen los SGBDOR en la actualidad, aunque como siempre ocurre, cada gestor incorpora sus propias particularidades y diferencias respecto al estándar.

Por ejemplo en PostgreSQL solo se puede usar la herencia entre tablas, ya que implementa los objetos como tuplas y las clases como tablas, mientras que Oracle soporta herencia de tipos.

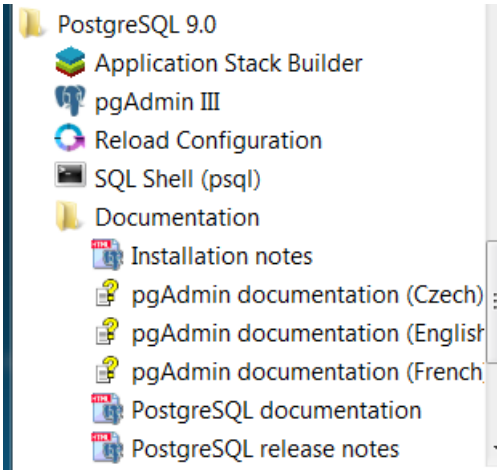
Para practicar con este tipo de gestores y algunas de las nuevas características que incorporan, hemos elegido **PostgreSQL**, considerado como el gestor de código abierto más avanzado del mundo.

El código fuente de PostgreSQL está disponible bajo la licencia BSD. Esta licencia te da libertad para usar, modificar y distribuir PostgreSQL en cualquier forma, ya sea junto a código abierto o cerrado. Además PostgreSQL incluye un API para diferentes lenguajes de programación, entre ellos Java y .NET

7.1 Instalación del Gestor PostgreSQL

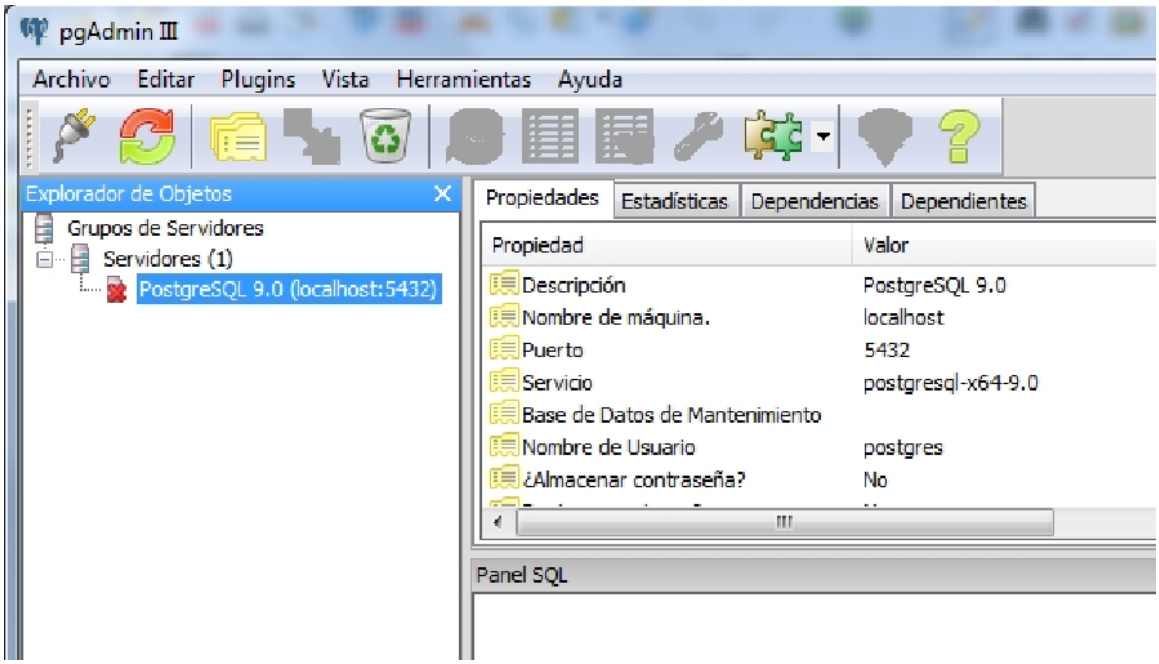
El **proceso de instalación del servidor PostgreSQL en Windows** consiste básicamente en los siguientes pasos:

- Acceder a la web http://www.postgresql.org y desde Downloads, página para descargas, descargar la versión , de 32 o 64 bits , de PostgreSQL para nuestro sistema operativo Windows.
- Una vez descargadado, ejecutamos el instalador. Aparece un asistente que guía el proceso de instalación, basta con ir dando a siguiente, hasta la pantalla donde habrá que introducir la contraseña del **superusuario de PostgreSQ**L, contraseña del **usuario postgres**.
- Se introduce contraseña y damos a siguiente hasta la pantalla de configuración regional que indicaremos idioma Spanish-Spain, le damos a siguiente , comenzará la instalación de ficheros y por último le damos a terminar.
- Desde *Herramientas Adminitrativas/Servicio**s*** podemos ver y cambiar la configuración de inicio del servicio, que por defecto aparece iniciado.
- Estando iniciado el servicio, desde menú *Inicio/Todos los Programas/ PostgreSQL* seleccionamos **pgAdmin III**, esto nos lleva al ***administrador gráfico de PostgreSQL.***

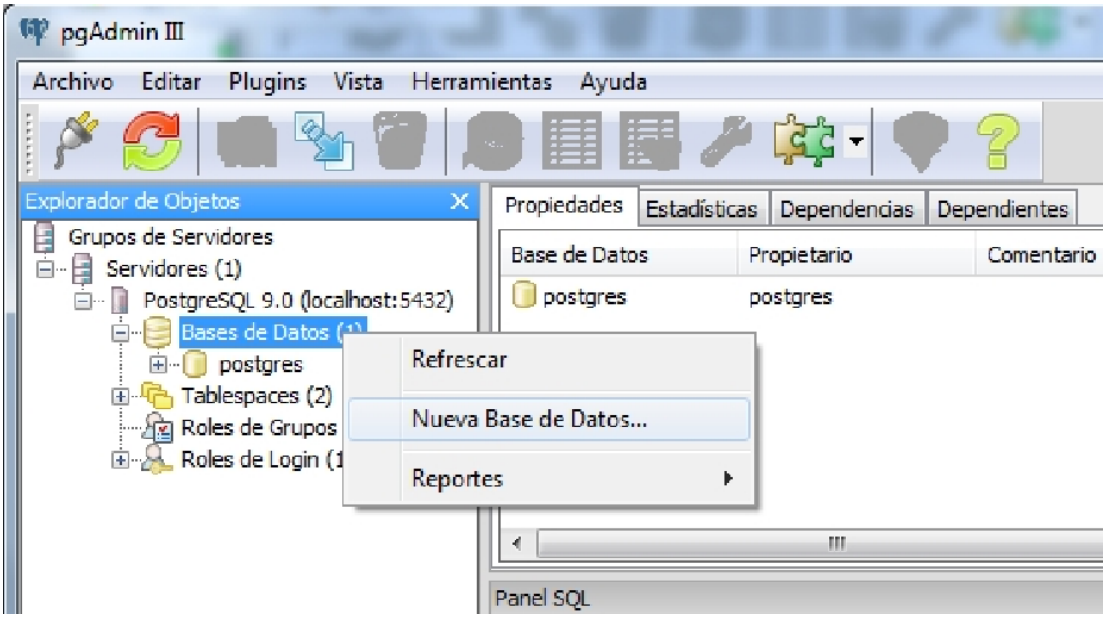


Trabajar desde pgAdmin III, administrador gráfico de PostgreSQL

- Hacemos doble click sobre el servidor PostgreSQL 9.0 (localhost:5432).



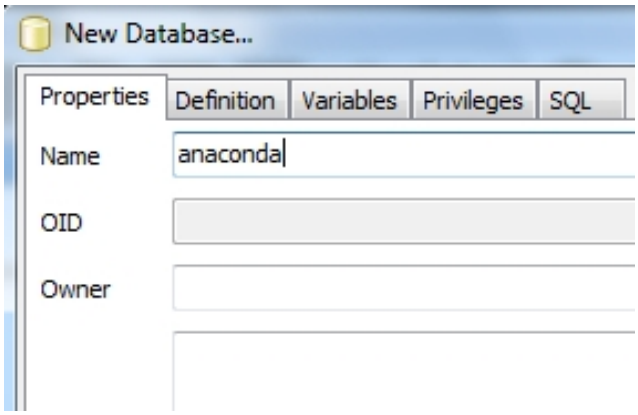
- Introducimos la contraseña del superusuario de PostgreSQL (*usuario postgres*) establecida en la instalación y pulsamos OK.
- Desde el grupo Base de Datos podemos obtener los detalles las bases de datos PostgreSQL, así como crear una nueva Base de Datos desde el menú contextual.



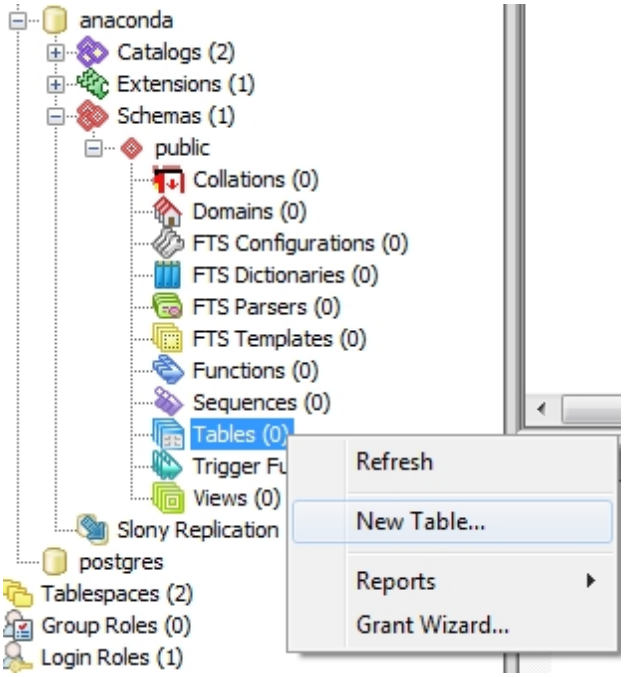
Creación de una base de datos

Con la ayuda de pgAdmin, esta tarea puede realizarse en unos cuantos clics de ratón:

- Desde **PgAdmin(III)** nos situamos **sobre localhost:5432** y hacemos doble clic para iniciar conexión con el servidor, introducimos la contraseña del usuario *postgres.*



- Nos situamos sobre el icono ***Databases***, pulsamos botón derecho del ratón y seleccionamos ***New Database***, introducimos como nombre anaconda.
- Si no se especifica lo contrario, la base de datos se creará con la codificación de caracteres UTF8. Esto puede cambiarse desde la ficha *Definition*, aunque siempre es una opción conveniente UTF8.
- Es conveniente especificar Spanish_Spain.1252 en los campos *Collation y Character type* de esa misma ficha, para asegurarnos de que la base de datos va a ordenar correctamente los campos de texto, y que no habrá caracteres erróneos.
- Finalmente, presionamos OK para crear la base de datos.



- Podemos ver haciendo doble clic sobre la BD, el árbol de objetos generado, entre esos objetos, en ***Schema/Public*** se crearán las tablas de la base de datos.
- De la BD anaconda cuelgan una serie de objetos, el más importante de los cuales es Schemas (Esquemas). Los esquemas son los verdaderos contenedores de la información.
- Una base de datos puede tener varios esquemas, aunque por defecto sólo se crea uno denominado public.
- Dentro de un esquema, la información se organiza en unidades denominadas Tables (Tablas).
- Aunque podemos crear nuevas tablas desde el menú contextual con un sencillo clic, nuestro objetivo es hacerlo con Java.

PARA SABER MÁS

En el siguiente enlace puedes consultar las **características generales de PostgreSQL**
http://www.postgresql.org.es/sobre_postgresql

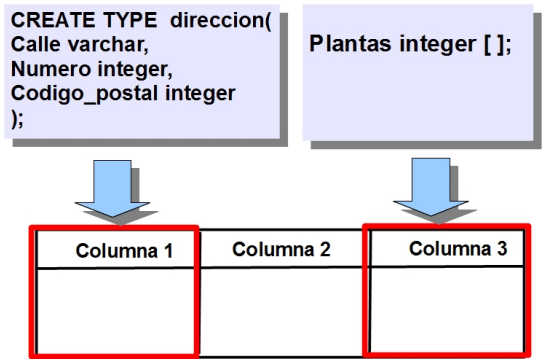
Desde este otro enlace puedes consultar la **documentación de PostgreSQL**
http://www.postgresql.org.es/documentacion

7.2 Tipos de datos: tipos básicos y tipos estructurados

Como ya sabes, los SGBDOR incorporan un conjunto muy rico de tipos de datos. PostgreSQL no soporta herencia de tipos pero permite **definir nuevos tipos de datos mediante los mecanismos de extensión.**

Te vamos a comentar tres categorías de tipos de datos que encontramos en PostgreSQL:

- **Tipos básicos**: el equivalente a los tipos de columna usados en cualquier Base de Datos Relacional.
- **Tipos compuestos**: un conjunto de valores definidos por el usuario con estructura de fila de tabla, y que como tal puede estar formada por tipos de datos distintos.
- **Tipos array**: un conjunto de valores distribuidos en un vector multidimensional, con la condición de que todos sean del mismo tipo de dato (básico o compuesto). Ofrece más funcionalidades que el array descrito en el estándar SQL99.



Los **tipos compuestos** de PostgreSQL son el equivalente a los **tipos estructurados** definidos por el usuario (UDT) del estándar SQL99. De hecho, son la base sobre la que se asienta el soporte a objetos.

El **tipo array** es el equivalente al **tipo colección**, permitiendo implementar las relaciones 1:N

Tpos de datos en PostgreSQL:
http://www.postgresql.org/docs/9.1/static/datatype.html

Tipos básicos.

Entre ellos podemos destacar:

- **Tipos numéricos**. Aparte de valores enteros, números de coma flotante, y números de precisión arbitraria, PostgreSQL incorpora también un tipo entero auto-incremental denominado *serial*.
- **Tipos de fecha y hora**. Además de los típicos valores de fecha, hora e instante, PostgreSQL incorpora el tipo **i**nterval para representar intervalos de tiempo.
- **Tipos de cadena de caracteres**. Prácticamente los mismos que en cualquier BDR.
- **Tipos largos**. Como por ejemplo, el tipo BLOB para representar objetos binarios. En la actualidad presentes en muchas BDR como MySQL.

Tipo compuesto o estructurado. (UDT)

- Los define el usuario mediante una sentencia DDL de la forma:

```
CREATE TYPE nombreTipo  AS (
        tipo miembro1,
        ……..
        tipo miembroN
);
```

pudiendo ser cada miembro de un tipo básico, otro tipo compuesto definido anteriormente o un tipo array.

EJEMPLO

Podemos crear **un nuevo tipo, el tipo dirección** de la siguiente forma:

```
CREATE TYPE direccion AS (
        calle varchar(40),
        numero integer,
        codigo_postal varchar(9)
);
```

y luego **definir una la tabla** de nombre afiliados, **con una columna basada en el nuevo tipo:**

```
CREATE TABLE  afiliados  (
        afiliado_id integer,
        nombre text,
        domicilio direccion
);
```

- Para insertar datos de un tipo compuesto se puede utilizar la siguiente notación. Observa que ROW() o tipo fila, es un constructor del nuevo tipo.

```
INSERT INTO afiliados VALUES (20, 'Pablo Rojo', ROW('Calle la Medina', 42, '04008'));
```

- Cada vez que se crea una tabla, se crea automáticamente **un tipo compuesto con el mismo nombre que la tabla**, para **representar el tipo fila de la tabla**. Pero las restricciones establecidas en la tabla, no se propagan al tipo de dato compuesto fuera de esa tabla.

EJEMPLO. Crea la tabla de nombre *direccion* y a la vez un nuevo tipo de nombre *direccion.* En este caso, la restricción CHECK() solo se aplica a la tabla direccion, no al tipo direccion.

```
CREATE TABLE direccion (
        calle varchar(40),
        numero integer CHECK (numero > 0),
        codigo_postal varchar(9)
);
```

Tipo array.

El **tipo array** permite especificar **vectores multidimensionales** como tipo de dato para las columnas de una tabla. La única condición es que **todos sus elementos sean del mismo tipo**.

Declaración de **tipos array:**

- Declaración de una **columna de tipo vector**: nombre_columna tipo_dato[n]
- Declaración de una **columna de tipo matriz multidimensional**: nombre_columna tipo_dato[f][c]

donde como se ve, sólo hay que agregar un '[ ]' al tipo de dato por cada dimensión.

- Se pueden dejar los [ ] sin valor. Aunque se ponga un valor de elementos para esa dimensión, PostgreSQL no restringe a ese tamaño especificado.

EJEMPLOS.

```
CREATE TABLE calificaciones(
            notas  real[6]
);
```

- Para Insertar valores, cada fila se inserta entre { } o usando el constructor ARRAY[ ]

```
INSERT INTO calificaciones  VALUES (ARRAY[5.5, 8, 9, 9, 6,7]);
```

```
CREATE TABLE marcas (
    puntos   integer[2][3]
);
```

- Para Insertar valores: INSERT INTO marcas VALUES ('{{1,2,3},{4,5,6}}');

7.3 Conexión mediante JDBC

La **conexión de una aplicación Java con PostgreSQL** se realiza mediante un **conector tipo JDBC.**

Para realizar la **integración del driver JDBC de PostgreSQL** seguiremos los pasos:

- Accedemos a la dirección web http://jdbc.postgresql.org/download.html y desde la sección Current Version, seleccionamos el driver más adecuado para el Java que tengamos instalada (normalmente, la última disponible para el JDBC4 Postgresql)
- Guardamos el fichero .jar en un directorio, y anotamos la ruta para referencia posterior. Desde Netbeans, clic con el botón derecho en Bibliotecas de nuestro proyecto, y ejecutar el Agregar archivo JAR/Carpeta, buscamos donde se guardó el fichero .jar descargado, aceptar y listo.

Una vez integrado el driver JDBC de PostgreSQL en un proyecto, podemos trabajar con la bases de datos desde Java importando el paquete java.sql.*: import java.sql.*

Recuerda que en JDBC, una base de datos está representada por una URL. La cadena correspondiente tiene una de las tres formas siguientes:

- jdbc: postgresql: base de datos
- jdbc: postgresql: //host/base de datos
- jdbc: postgresql: //host: puerto/base de datos

El nombre de host por defecto, del servidor PostgreSQL, será **localhost**, y el **puerto** por el que escucha el **5432**. Como ves, esta cadena es idéntica a la empleada por otros SGBDR.

- **Para conectar con la base de datos**, utilizaremos el método DriverManager.getConnection() que devuelve un objeto Connection (la conexión con la base de datos). Una de las posibles sintaxis de este método es:
  Connection conn = DriverManager.getConnection(url, username, password);

- Una vez abierta, **la conexión se mantendrá operativa hasta que se llame a su método close()** para efectuar la desconexión. Si al intentar conectar con la base de datos ésta no existe, se generará una excepción del tipo PSQLException "FATAL: no existe la base de datos ...". En cualquier caso se requiere un **bloque try-catch .**

```
//cadena url de la base de datos anaconda en el servidor local (no hay
//que indicar el puerto si es el por defecto)
String url = "jdbc:postgresql://localhost/anaconda";

//conexión con la base de datos
Connection conn = null;

try {
  //abre la conexión con la base de datos a la que apunta el url
  //mediante la contraseña del usuario postgres
  conn = DriverManager.getConnection(url, "postgres", "1234");

} catch (SQLException ex) {
  //imprime la excepción
  System.out.println(ex.toString());
} finally {

  //cierra la conexión
  conn.close();
}
```

En el siguiente enlace encontrarás información detallada sobre el **API JDBC de PostgreSQL**:
http://jdbc.postgresql.org/documentation/publicapi/index.html

7.4 Consulta y actualización de tipos básicos

PostgreSQL implementa los objetos como filas, las clases como tablas, y los atributos como columnas. Hablaremos por tanto de tablas, filas y columnas, tal y como lo hace PostgreSQL.

Para interactuar con **PostgreSQL desde Java, vía JDBC**, debemos enviar sentencias SQL a la base de datos mediante el uso de comandos.

Podemos utilizar los siguientes tipos de sentencias o comandos:

- Statement: para sentencias sencillas en SQL.
- PreparedStatement: para consultas preparadas, como por ejemplo las que tienen parámetros.
- CallableStatement: para ejecutar procedimientos almacenados en la base de datos.

Recuerda que el API JDBC distingue dos tipos de consultas:

- Consultas: SELECT
- Actualizaciones: INSERT, UPDATE, DELETE, sentencias DDL

Por tanto, si nuestra conexión es conn, para enviar un comando Statement haríamos lo siguiente:

- **Crear la sentencia Statement**, por ejemplo.: Statement sta = conn.createStatement();
- **Ejecutar la sentencia:**
  - **executeQuery().** Si es una consulta (SELECT).

    sta.executeQuery(string sentenciaSQL);

    Devuelve un objeto tipo **ResultSet,** apuntando al resultado de la consulta.
  - **executeUpdate().** Si es una actualización (INSERT, UPDATE O DELETE)

    sta.executeUpdate(string sentenciaSQL);

    Devuelve un **entero,** indicando el número de filas afectadas por la operación realizada
  - **execute().** Si es una sentencia DDL (CREATE, DROP, o un ALTER)

    sta.execute(string sentenciaSQL);

    Devuelve un **entero,** indicando el éxito o fracaso de la operación realizada

Como ves, en PostgreSQL se utilizan estos comandos como en cualquier otra BDR. La diferencia estará en que se pueden utilizar sentencias SQL que manejan tipos complejos.

EJEMPLO. Consulta de tipos básicos.

```
System.out.print("\nTemperatura media por mes de Murcia:\n");
//sentencia SQL
String consulta = "SELECT m.mes, "
        + "ROUND((dm.temp_max+dm.temp_min)/2,2) "
        + "FROM meses m  "
        + "INNER JOIN  datos_meteo dm ON m.mes_id=dm.mes_id "
        + "WHERE dm.provincia_id=3";
//comando SQL
Statement sta = conn.createStatement();
//ejecuta la sentSql para que devuelva un conjunto de registros
ResultSet res = sta.executeQuery(consulta);
//imprime el resultado
imprimir_ResultSet(res);
//cierra los objetos auxiliares
res.close();
sta.close();
```

Donde el método **imprimir_ResultSet()** es:

```
private static void imprimir_ResultSet(ResultSet resultSet) throws
        SQLException {
  //número de columnas del resultset
  ResultSetMetaData metaDatos = resultSet.getMetaData();
  int columnas = metaDatos.getColumnCount();
  //mientras quedan registros por leer en el ResultSet
  while (resultSet.next()) {
    for (int i = 1; i <= columnas; i++) {
      //imprime cada columna seguido de un tabulador
      System.out.print(resultSet.getString(i) + "\t");
    }
    //línea en blanco
    System.out.println();
  }
}
```

Ejemplo de **consulta de actualización**

```
System.out.print("\nCambia la clave principal de 'Valencia' en la tabla "
        + "'provincias'(de 4 a 5).");
//sentencia SQL
String consulta = "UPDATE provincias "
        + "SET provincia_id=5 "
        + "WHERE "
        + "provincia_id=4";
//objeto Statement
Statement sta = conn.createStatement();
//ejecuta la sentencia SQL
System.out.print("\nComo resultado, " + sta.executeUpdate(consulta)
        + " fila actualizada:\n");
```

El proyecto Java completo, **Ud5_PostgreBasico.zip,** te proporciona ejemplos de:

- creación de tablas en la base de datos *temperaturas* de PostreSQL
- inserción, consulta, modificación y borrado de tipos básicos.

Para probar el proyecto Java, crea la BD PostgreSQL de nombre *temperaturas* mediante PgAdminIII. La aplicación creará las tablas:

- meses
- provincias
- datos_meteo

7.5 Consulta y actualización de tipos estructurados

Imaginemos que tenemos el tipo estructurado *direccion*:

```
CREATE TYPE direccion AS (
        calle varchar (40),
        numero integer,
        codigo_postal varchar(9)
);
```

y la tabla *empleados*, con una columna basada en el nuevo tipo:

```
CREATE TABLE empleados (
        emple_id serial,
        nombre text,
        apellidos text,
        domicilio direccion
);
```

¿**Cómo insertamos valores en una tabla con un tipo estructurado**? Se puede hacer de dos formas:

- Pasando el valor del campo estructurado **entre comillas simples** (lo que obliga a encerrar entre comillas dobles cualquier valor de cadena dentro), y paréntesis para encerrar los subvalores separados por comas:

  ```
  INSERT INTO empleados (nombre, apellidos, direccion)
  VALUES ('Onorato', 'Maestre Toledo', '("Calle de Rufino", 56, "98080")');
  ```

- Mediante el constructor **ROW** que permite dar valor a un tipo compuesto o estructurado.

  ```
  INSERT INTO empleados (nombre, apellidos, direccion)
  VALUES ('Onorato', 'Maestre Toledo', ROW('Calle de Rufino', 56, '98080'));
  ```

¿Cómo se referencia una subcolumna de un tipo estructurado?

- Se emplea la notación punto, '.' , tras el nombre de la columna entre paréntesis, (tanto en consultas de selección, como de actualización) . Por ejemplo:

  ```
  SELECT (domicilio).calle FROM empleados WHERE (domicilio).codigo _postal='98080';
  ```

devolvería, entre otros, el nombre de la calle Maestre Toledo. Los paréntesis son necesarios para que el gestor no confunda el nombre del campo compuesto con el de una tabla.

Y ¿cómo se elimina el tipo estructurado?

- Se elimina con DROP TYPE, por ejemplo DROP TYPE direccion;

EJEMPLOS

En el siguiente enlace encontrarás ejemplos de cómo se modifican los valores de un tipo estructurado o compuesto, y más **ejemplos sobre consultas e inserciones en PostgreSQL**.

http://www.postgresql.org/docs/9.1/interactive/rowtypes.html

El proyecto Java **ud5_PostgreEstructurado.zip** incluye ejemplos para ilustrar cómo realizar estas tareas desde una aplicación Java.

A partir de la BD '*viviendas*' de PosgreSQL, este proyecto Java se encarga de:

- crear nuevos tipos de datos (UDT) en los que se basarán las tablas de la BDOR
- crear las tablas
- insertar datos y consultar, modificar y eliminar datos

Crea los **tipos (UDT) 'vivienda' y 'direccion'**

```
private static void crear_Tipos(Connection conn) throws SQLException {
  String sql = "CREATE TYPE vivienda AS("
          + "planta integer, "
          + "metros_2 integer, "
          + "num_habitaciones integer, "
          + "num_bannios integer, "
          + "promotor varchar(25));"

          + "CREATE TYPE direccion AS("
          + "calle varchar(40), "
          + "ciudad varchar(25), "
          + "codigo_postal varchar(9));";
  //comando auxiliar para ejecutar la sql
  Statement sta = conn.createStatement();
  //ejecuta la sql
  sta.execute(sql);
  //cierra el objeto auxiliar
  sta.close();
}
```

Crea la **tabla 'viviendas_nuevas'** basada en los tipos estructurados creados anterioremente

```
private static void crear_Tabla(Connection conn) throws SQLException {
  //sql SQL
  String sql = "CREATE TABLE viviendas_nuevas("
          + "vivienda_id serial,"
          + "caracteristicas vivienda, "
          + "ubicacion direccion, "
          + "CONSTRAINT vivienda_id PRIMARY KEY (vivienda_id)"
          + ")";
  //comando auxiliar para ejecutar la sql
  Statement sta = conn.createStatement();
  //ejecuta la sql
  sta.execute(sql);
  //cierra el objeto auxiliar
  sta.close();
}
```

Inserta algunos datos en la tabla. Observa el **uso de ROW()**

```
private static void insertar_Registros(Connection conn) throws SQLException {
  //sql SQL
  String sql = "INSERT INTO viviendas_nuevas("
          + "caracteristicas,ubicacion) VALUES("
          + "ROW(3,69,2,1,'Construcciones Perico'),"
          + "ROW('Pepito López','Madrid','28013'));"

          + "INSERT INTO viviendas_nuevas("
          + "caracteristicas,ubicacion) VALUES("
          + "ROW(2,101,3,2,'Construcciones Julian'),"
          + "ROW('Juanita Pedroches','Murcia','30300'));"

          + "INSERT INTO viviendas_nuevas("
          + "caracteristicas,ubicacion) VALUES("
          + "ROW(10,194,5,2,'Construcciones Rodrigo'),"
          + "ROW('Evaristo Torcuato','Madrid','28013')"
          + ")";
  //comando auxiliar para ejecutar la sql
  Statement sta = conn.createStatement();
  //ejecuta la sql
  sta.execute(sql);
  //cierra el objeto auxiliar
  sta.close();
}
```

Consulta mediante una sentencia Statement, las viviendas en Madrid. Observa **como se accede al tipo estructurado (UDT)**

```
private static void consulta2(Connection conn) throws SQLException {

  System.out.print("\nOferta de viviendas de nueva construcción "
          + "en Madrid:\n");
  String consulta = "SELECT * FROM viviendas_nuevas "
          + "WHERE (viviendas_nuevas.ubicacion).ciudad='Madrid'";
  Statement sta = conn.createStatement();
  ResultSet res = sta.executeQuery(consulta);
  imprimir_ResultSet(res);
  res.close();
  sta.close();
}
```

Consulta mediante el uso de una sentencia preparada con parámetros. **PreparedStatement**

```
private static void consulta3(Connection conn) throws SQLException {
  int n=3; //simula el valor introducido desde la interfaz gráfica
  System.out.print("\nPromotores en Madrid de viviendas nuevas "
          + "con "+n +" habitaciones o más:\n");
  String consulta = "SELECT "
          + "(viviendas_nuevas.caracteristicas).promotor AS prom "
          + "FROM viviendas_nuevas "
          + "WHERE (viviendas_nuevas.ubicacion).ciudad='Madrid' "
          + "AND (viviendas_nuevas.caracteristicas).num_habitaciones>?";
  //Sentencia preparada. Uso de parámetros
  PreparedStatement pstmt = conn.prepareStatement(consulta);
  //Asigna parámetro
  pstmt.setInt(1, n);
  //ejecuta consulta
  ResultSet res = pstmt.executeQuery();
  imprimir_ResultSet(res);
  res.close();
  pstmt.close();
}
```

Consulta de **actualización de un tipo estructurado**.
Observa como se accede al tipo estructurado (UDT)

```
System.out.print("\nCambia el código postal de la vivienda ofertada "
        + "en la calle 'Evaristo Torcuato'");
String consulta = "UPDATE viviendas_nuevas "
        + "SET ubicacion.codigo_postal='28006' "
        + "WHERE "
        + "(viviendas_nuevas.ubicacion).calle='Evaristo Torcuato'";
Statement sta = conn.createStatement();
//ejecuta la sql para que muestre las filas afectadas
System.out.print("\nComo resultado, " + sta.executeUpdate(consulta)
        + " fila actualizada:\n");
```

Una vez ejecutado el proyecto, observa desde PgAdminIII, que puedes utilizar los nuevos tipos de datos creados 'vivienda' y direccion' para otras columnas de tablas de esa BD.

7.6 Consulta y actualización de tipos array

El tipo array de PostgreSQL, tal y como ya has visto, permite que en las columnas de las tablas de la BD se pueda almacenar un conjunto de valores (atributo multivaluado)

La declaración de **tipos array puede ser :**

- **tipo vector**: nombre_columna tipo_dato[n]
- **tipo matriz multidimensional**: nombre_columna tipo_dato[f][c]

Supongamos la siguiente defínción de tabla y columna de tipo array:

```
CREATE TABLE marcas (
      puntos   integer[2][3]
);
```

Para **acceder a valores array** se utiliza la siguiente notación:

- **Insertar valores:** cada fila se inserta entre { } o bien se utiliza el constructor ARRAY[ ]

```
INSERT INTO marcas  VALUES ('{{1,2,3},{4,5,6}}');
INSERT INTO marcas  VALUES (ARRAY[{1,2,3},{4,5,6}]);
```

- **Consultar:** una fila o un elemento concreto
  - para referirse a la fila f del array puntos se pondrá: puntos [f:f]

    ```
    SELECT  puntos[ 1:1] FROM marcas;    //fila 1 del array puntos
    ```
  - para referirse a un elemento concreto del array puntos será: puntos [f][c]

    ```
    SELECT puntos[1][3] FROM marcas;    //elemento 1,3 del array puntos
    ```

En el siguiente enlace encontrarás más ejemplos de cómo consultar y actualizar columnas de tipo array.
http://www.postgresql.org/docs/9.1/static/arrays.html

EJEMPLO. En el siguiente ejemplo puedes ver la creación de una tabla con una columna de tipo array de varchar y como se insertan y consultan valores.

```
sta.execute("CREATE TABLE tareas(comercial_id integer,"
        + "agenda varchar[][])");
//inserta un registro de dos tareas por día para el comercial número 3
//durante dos días (día 1: [1][1],[1][2]; día 2:[2][1],[2][2])
sta.executeUpdate("INSERT INTO tareas VALUES(3,"
        + "'{{\"reunión 9:30\",\"comida 14:30\"},"
        + "{\"reunión 8:30\",\"cena 22:30\"}}')");
//consulta todas las tareas del segundo día del comercial número 3
ResultSet rst = sta.executeQuery("SELECT agenda[2:2] "
        + "FROM tareas WHERE comercial_id=3");
//muestra el resultado
while (rst.next()) {
  System.out.println(rst.getString(1));
}
//consulta la segunda tarea del primer día del comercial número 3
rst = sta.executeQuery("SELECT agenda[1][2] "
        + "FROM tareas WHERE comercial_id=3");
while (rst.next()) {
System.out.println(rst.getString(1));
 }
```

7.7 Herencia de tablas

PostgreSQL **soporta herencia de tablas**, de manera que se puede crear una tabla que herede de otra.

EJEMPLO.

```
CREATE TABLE ciudades (
        nombre  text,
        poblacion float,
        altitud int
);

CREATE TABLE capitales (
        estado  char(2)
) INHERITS (ciudades);
```

En este caso, una fila (una instancia) de capitales hereda todos los atributos (nombre, poblacion y altitud) de su padre, ciudades.

La tabla (clase) capitales tiene un atributo extra, estado, que indica de que estado es capital.

En PostgreSQL:

- una tabla puede heredar de ninguna o varias tablas, y
- una consulta puede hacer referencia tanto a todas las filas de una tabla como a todas las filas de una tabla y sus descendientes. El comportamiento por defecto es el último.

Por **ejemplo**, la siguiente consulta encuentra todas aquellas ciudades, incluidas las capitales de estado, que están situadas a una altitud de 500 o más metros:

```
SELECT nombre, altitud
FROM ciudades
WHERE altitud > 500;
```

```
 nombre     |    altitud
------------+----------
 Las Vegas  |      2174
 Mariposa   |      1953
 Madison    |       845
```

Para mostrar tan **solo las ciudades que no son capitales**, se debe utilizar el predicado **ONLY**

```
SELECT nombre, altitud
   FROM ONLY ciudades
   WHERE altitud > 500;
```

```
 nombre   | altitud
------------+----------
 Las Vegas  |      2174
 Mariposa   |      1953
```

También se puede utilizar la **notación *** después de la tabla, para indicar de forma explícita que la consulta incluya a todos los descendientes en la jerarquía de herencia.

```
SELECT nombre, altitud
FROM ciudades*
WHERE altitud > 500;
```

Si no se cambia el comportamiento por defecto de la herencia de PostgreSQL, el * no es necesario. Si se cambia el comportamiento por defeto de la herencia en PostgreSQL, habrá que usar * para acceder a todos los descendientes y no será necesario el predicado ONLY cuando no se requiera accceder a los descendientes. El parámetro de configuración es `sql_inheritance` (`boolean`) que por defecto está a `true` u `on` desde la versión 7.1.

Consulta el siguiente enlace para más información sobre Herencia en PostgreSQL.

Herencia en PostgreSQL
http://www.postgresql.org/docs/9.2/interactive/ddl-inherit.html

7.8 Funciones del gestor desde Java

Recuerda que el estándar SQL99 introduce la posibilidad de crear funciones de usuario que puedan quedar almacenadas en el gestor de bases de datos, utilizando un Lenguaje Procedural (PL). Estas funciones pueden definir el comportamiento de ciertos objetos y entonces, se llaman métodos.

En PostgreSQL podemos crear funciones con prestaciones para los nuevos tipos de datos, tipos estructurados y array, y llamarlas desde una aplicación Java, pero no son propiamente métodos, tal y como los recoge el estándar SQL99.

En PostgreSQL se pueden construir funciones mediante diferentes lenguajes: SQL, PL/pgSQL, PL/Java, C o C++).
Veremos algún ejemplo de funciones en SQL. Sobre estas funciones debes saber que:

- Los **parámetros de la función** pueden ser cualquier tipo de dato admitido por el gestor (un tipo básico, un tipo compuesto, un array o alguna combinación de ellos).
- El **tipo devuelto** puede ser un tipo básico, un array o un tipo compuesto.
- Su **estructura general** tiene la forma:

```
CREATE FUNCION nombre_funcion (tipo_1,tipo_2,...)
  RETURN tipo AS $$sentencia_sql$$
  LANGUAJE SQL
```

Y debes **tener en cuenta que**:

- Los argumentos de la función SQL se pueden referenciar en las consultas usando la sintaxis $n:, donde $1 se refiere al primer argumento, $2 al segundo, y así sucesivamente. Si un argumento es un tipo compuesto, entonces se usará la notación '.' para acceder a sus subcolumnas.
- Por último, al final de la función hay que especificar que la función se ha escrito en lenguaje SQL mediante las palabras clave LANGUAJE SQL.

Por ejemplo, desde Java podemos crear una función que convierte un tipo estructurado, por ejemplo el tipo direccion, en una cadena. Para ello, ejecutaremos un comando Statement (sta) con el código SQL de creación de la función:

```
//crea el tipo estructurado
sta.execute("CREATE TYPE puesto AS(nombre varchar,"
        + "cargo varchar,sueldo numeric)");
//crea la función que transforma el tipo estructurado en una cadena
sta.execute("CREATE FUNCTION cadena_puesto(puesto) RETURNS varchar AS $$"
        + "SELECT $1.nombre||' como '||$1.cargo||' tiene un sueldo de '"
        + "||CAST(ROUND($1.sueldo,2) AS varchar)||'€';$$"
        + "LANGUAGE SQL");
//crea la tabla con una columna del tipo creado (columna ocupacion)
sta.execute("CREATE TABLE empleados(empleado_id serial,"
        + "ocupacion puesto)");

    //Ejecuta la función mediante una consulta SELECT
    ResultSet rst = sta.executeQuery("SELECT cadena_puesto(ocupacion) "
            + "FROM empleados");
```

Si una función tiene algún parámetro de tipo estructurado, no se podrá eliminar el tipo hasta que no se elimine la función. Se puede usar DROP TYPE nb_tipo CASCADE para eliminar de manera automática las funciones que utilizan ese tipo.

*Consulta los siguientes enlaces para más información sobre **funciones PostgreSQL***
http://www.postgresql.org/docs/9.2/static/xfunc-sql.html

http://www.postgresql.org/docs/9.2/static/sql-createfunction.html

8 Gestión de transacciones

Como en cualquier otro Sistema de Bases de datos, en un Sistema de bases de objetos u objeto relacional, **una transacción** es un conjunto de sentencias que se ejecutan formando una unidad de trabajo, esto es, en forma indivisible o atómica, o se ejecutan todas o no se ejecuta ninguna.

Mediante la gestión de transacciones, los sistemas gestores proporcionan un acceso concurrente a los datos almacenados, mantienen la integridad y seguridad de los datos, y proporcionan un mecanismo de recuperación de la base de datos ante fallos.

Un ejemplo habitual para motivar la necesidad de transacciones es el traspaso de una cantidad de dinero (digamos 10000€) entre dos cuentas bancarias.

Normalmente se realiza mediante dos operaciones distintas, una en la que se decrementa el saldo de la cuenta origen y otra en la que incrementamos el saldo de la cuenta destino.

Para garantizar la integridad del sistema (es decir, para que no aparezca o desaparezca dinero), las dos operaciones tienen que completarse por completo, o anularse íntegramente en caso de que una de ellas falle.

Las transacciones deben cumplir el criterio **ACID**.

- **Atomicidad**. Se deben cumplir todas las operaciones de la transacción o no se cumple ninguna; no puede quedar a medias.
- **Consistencia**.: La transacción solo termina si la base de datos queda en un estado consistente.
- **Isolation** (Aislamiento). Las transacciones sobre la misma información deben ser independientes, para que no interfieran sus operaciones y no se produzca ningún tipo de error.
- **Durabilidad**. Cuando la transacción termina el resultado de la misma perdura, y no se puede deshacer aunque falle el sistema.

Algunos sistemas proporcionan también **puntos de salvaguarda** (***savepoints***) que permiten descartar selectivamente partes de la transacción, justo antes de acometer el resto. Así, después de definir un punto como punto de salvaguarda, puede retrocederse al mismo. Entonces se descartan todos los cambios hechos por la transacción después del punto de salvaguarda, pero se mantienen todos los anteriores.

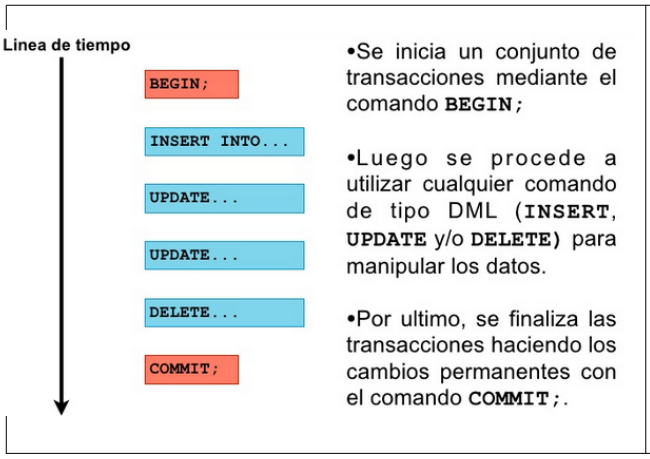
Originariamente, los puntos de salvaguarda fueron una aportación del estándar SQL99 de las Bases de Datos Objeto-Relacionales. Pero con el tiempo, se han ido incorporando también a muchas Bases de Datos Relaciones como MySQL (al menos cuando se utiliza la tecnología de almacenamiento InnoDB).

Transacciones en PostgreSQL
http://www.postgresql.org/docs/9.2/interactive/tutorial-transactions.html

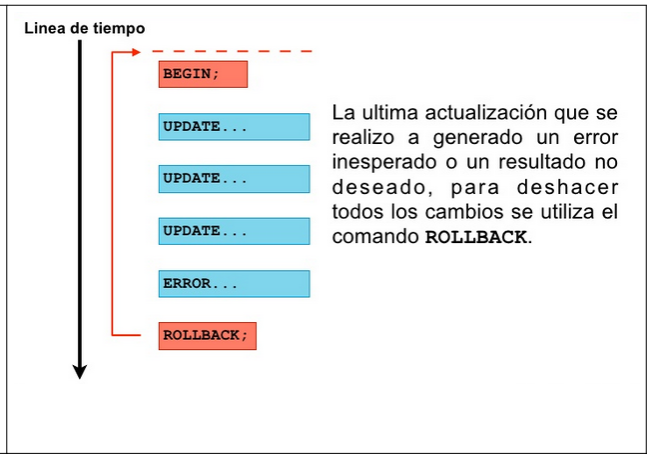
8.1 Transacciones en una base de datos objeto-relacional

Los SGBDOR gestionan transacciones mediante las sentencias COMMIT (confirmar transacción) y ROLLBACK (deshacer transacción).

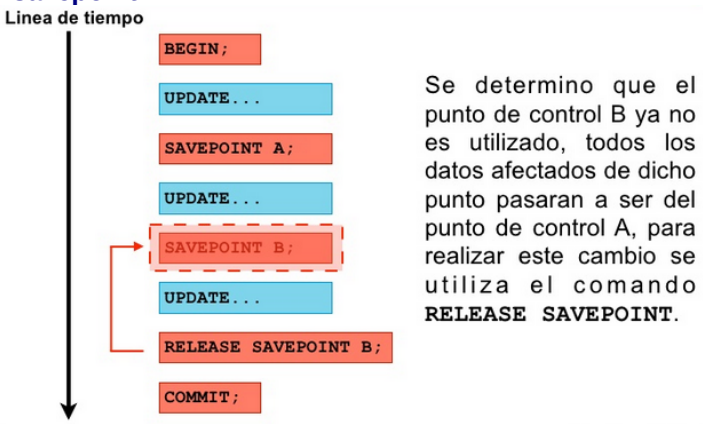
commit



rollback



savepoint



JDBC permite agrupar instrucciones SQL en una sola transacción. Así, podemos asegurar las propiedades ACID usando las facilidades transaccionales del JDBC.

El control de la transacción lo realiza el objeto Connection. Cuando se crea una conexión, por defecto es en modo AUTO COMMIT= TRUE. Esto significa que cada instrucción individual SQL se trata como una transacción en sí misma, y se confirmará en cuanto que la ejecución termine.

Por tanto, si queremos que un grupo de sentencias se ejecuten como una transacción, tras abrir una conexión Connection conn; habrá que:

- Poner autocommit=false de la siguiente manera:

```
if (conn.getAutoCommit() )
conn.setAutoCommit( false );
```

- Mediante un bloque try-catch controlar si se deshace conn.rollback o confirma conn.commit la transacción iniciada con esa conexión.

EJEMPLO

En el siguiente enlace puedes ver ejemplos de **transacción en PostgreSQL.**

http://balteus.blogspot.com.es/2012/04/transacciones-autonomas-en-postgresql.html

8.2 Transacciones en un gestor de objetos

Las transacciones en un sistema de objetos puro, se gestionan mediante COMMIT y ROLLBACK .

Un fallo causa un rollback de los datos, como normalmente sucede en las bases de datos relacionales. En cuanto a la concurrencia, las bases de objetos verifican en qué momento permiten el acceso en paralelo a los datos. Este acceso concurrente implica que más de una aplicación o hilo podrían estar leyendo o actualizando los mismos objetos a la vez. Esto se suele denominar **commit de dos fases** (dos procesos pueden trabajar sobre el mismo objeto concurrentemente). Para ello se utiliza normalmente bloqueo de datos para lecturas y escrituras.

Veamos un ejemplo sencillo con la base de objetos db4o**. En Db4o:**

- Siempre se trabaja dentro de una transacción.
- Al abrir un ObjectContainer se crea e inicia implícitamente una transacción
- Las transacciones se gestionan explícitamente mediante commit y rollback.
- La transacción actual hace commit implícitamente cuando se cierra el ObjectContainer.

Por ejemplo, podemos deshacer una transacción con rollback , restaurando la base de objetos al estado anterior, justo al estado después del último commit o rollback.

En el siguiente ejemplo, al ejecutar db.rollback después de db.store(p3) y db.store(p4), hará que los objetos p3 y p4 no se almacenen en la base de objetos.

```
//Persistir Objetos: almacenamos los objetos con el método store()
db.store(p1);
db.store(p2);
db.commit(); //confirmar explícitamente la transacción con commit
//se crean dos objetos ponente
ponente p3 = new ponente("33C", "Ana Navarro", "anavarro@yahoo.com", 200);
ponente p4 = new ponente("44D", "Pedro Sánchez", "psanchez@mixmail.com",90);
//se almacenan en la base de objetos
db.store(p3);
db.store(p4);
db.rollback();//se deshace la transacción (almacenado de p3 y p4 en la BDOO)
```

EJEMPLOS

En el siguiente enlace puede consultar más ejemplos sobre la gestión de transacciones en db4o y sobre el problema de los denominados objetos vivos.

Gestión de transacciones en db4o
http://kuainasi.ciens.ucv.ve/db4o/DB4o-P3.htm#Transac

9 Bibliografía

- Acceso a Datos , Editorial Garceta
- Java, cómo programar, Edición 9, Editorial DEITEL
- http://sinbad.dit.upm.es/docencia/grado/curso1213/bsdt1213.html
- http://www.slideshare.net/nicola51980/postgresql-leccin-8-manipulando-datos-y-transacciones
- http://www.db4o.com/espanol/
- http://www.matisse.com/developers/documentation/
- http://www.postgresql.org/docs/9.2/static/