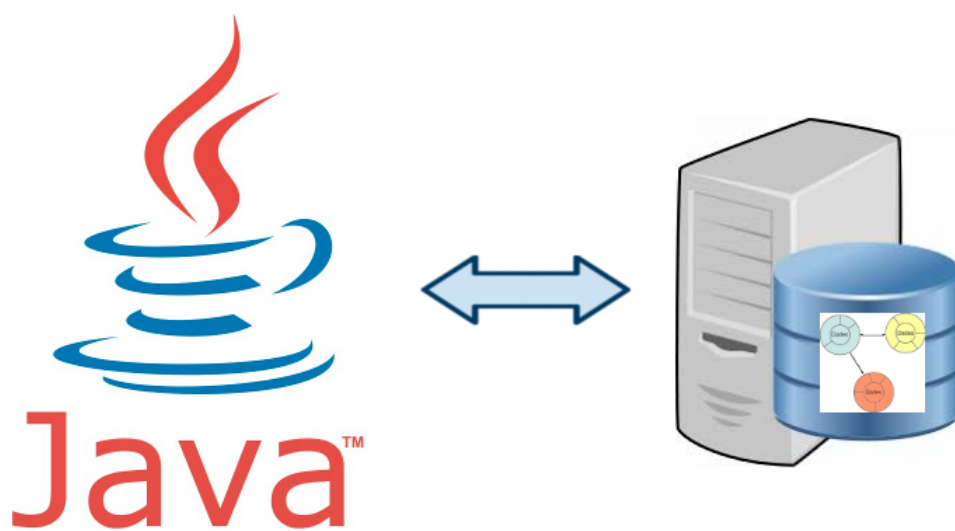

Accés a Dades

Tema 6: Bases de Dades Objecte-Relacionals i Orientades a Objectes



1. Introducció

En aquest tema veurem encara dues alternatives més que intenten minimitzar el desfasament objecte-relacional.

En la primera de les alternatives utilitzarem els mateixos SGBD relacionals. Com que en les aplicacions actuals s'utilitza quasi sempre Programació Orientada a Objectes, això ha obligat a aquests SGBD Relacionals a incorporar característiques orientades a objectes, com poden ser tipus de dades diferents (per exemple Arrays) i en definitiva poder guardar una cosa pareguda a un objecte. És el que s'ha batejat com a Sistemes Gestors de Bases de Dades **Objecte-Relacionals**, i bons exemples són Oracle, Informix o PostgreSQL.

La segona alternativa consisteix a utilitzar directament un Sistema Gestor de Bases de Dades **Orientat a Objectes**. Evidentment, es tracta de l'única alternativa que **no presenta desfasament Objecte-Relacional**, ja que les dades es tracten sempre, fins i tot durant l'emmagatzematge, com si foren objectes i no taules.

2 - Bases de Dades Objecte-Relacionals

Com s'ha comentat en el punt anterior, les **Bases de Dades Objecte-Relacionals** (SGBDOR, o ORDBMS en anglès) són Sistemes Gestors de Bases de Dades Relacionals que incorporen conceptes d'objectes per a poder donar servei a la programació orientada a objectes, que cada vegada s'està imposant més. D'aquesta manera podrem guardar objectes (o una cosa semblant) en les taules.

Com hem comentat en temes anteriors, les Bases de Dades Relacionals continuen utilitzant-se de forma massiva, per la seua solvència i robustesa. No obstant això, les més avançades intenten incorporar els objectes en els tipus de dades, de manera que els programes fets en Programació Orientada a Objectes (POO) ho tenen més fàcil per continuar utilitzant-les. És el cas per exemple dels SGBD Oracle, Informix o PostgreSQL.

En aquestos apunts utilitzarem **PostgreSQL**, per a poder donar continuïtat al SGBD utilitzat en el mòdul de Bases de Dades, de primer de DAM. Només volem connectar-nos com a clients, això sí, com a usuaris diferents, per a poder crear cadascú els seus objectes sense interferències d'uns als altres. D'aquesta manera se'ns plantegen dues possibilitats:

- Instal·lar el servidor PostgreSQL per a no interferir amb ningú,
- No instal·lar el servidor, i utilitzar un usuari i Bases de Dades diferent per a no interferir (el professor us en proporcionarà un).

Tant en un cas com en l'altre, en principi tindriem prou com a clients amb la perspectiva **Database Development** d'Eclipse, connectant-nos a la Base e Dades i usuari propis. Podrem consultar les coses que hi ha i també executar sentències SQL. Tanmateix, per a més comoditat i control sobre la Base de Dades de PostgreSQL, utilitzarem el **pgAdmin III**, el client més utilitzat de PostgreSQL.

Si voleu instal·lar-vos el servidor PostgreSQL:

En cas que vulgueu instal·lar-vos el vostre propi servidor PostgreSQL en el vostre equip, per a no tenir interferències de ningú.

Com a guia d'instal·lació i utilització de **PostgreSQL**, teniu uns apunts en l'últim tema, el dels annexos. Concretament, els punts que més interessaran seran:

- **2.2 Instal·lació automàtica en Windows**
- **3.2 PgAdmin III**, que és el programa que utilitzarem per a crear usuaris, crear Bases de Dades i per a fer consultes.
- **6.1 Gestió de rols: utilització com a usuaris**, per a poder crear usuaris
- **6.4 Autenticació d'usuaris**, per a poder connectar des del client (des de Java, en el nostre cas)
- **7.1 Bases de Dades: Creació, Modificació i Eliminació**, per crear Bases de Dades

Si heu instal·lat PostgreSQL en una màquina virtual, per a connectar des de qualsevol client, haurem de saber **l'adreça IP** d'aquesta màquina virtual. En canvi si heu instal·lat PostgreSQL en la màquina real, els accessos des del client (pgAdmin III i des de Java) es faran al **localhost**.

L'objectiu final serà accedir des de Java, però per a poder fer proves ens faria falta algun programa per accedir directament a PostgreSQL, veure els usuaris, veure directament les taules, i provar sentències SQL. El més còmode serà **PgAdmin III** (com s'ha comentat més amunt). Per a veure com s'utilitza, podeu consultar els apunts abans esmentats (pregunta 3.2). Però recordeu que únicament l'utilitzarem per a accedir de forma còmoda i poder fer proves ràpidament.

Haureu de crear-vos una connexió com al superusuari **postgres**, amb la contrasenya proporcionada en el moment de la instal·lació

Servidor localhost

Propietats SSL Tunnel SSH Avançat

Nom localhost

Ordinador central localhost

Port 5432

Servei

BBDD de mantenim. postgres

Nom d'usuari postgres

Contrasenya

Desa la contrasenya ☐

Color

Grup Servidors

Ajuda D'acord Cancel·la

A continuació haureu de crear un usuari per a fer les pràctiques del tema. Li podeu posar el nom **rx**, amb contrasenya **rx**. Posteriorment haureu de crear la Base de Dades **rx**, el propietari de la qual ha de ser **rx**.

Per últim, us aniria bé una connexió com a usuari **rx** a la BD **rx**. Aquesta és la que hauríeu d'utilitzar per a fer les pràctiques del tema.

Nou Registre de Servidor

Propietats SSL Tunnel SSH Avançat

Nom: localhost - rxx

Ordinador central: localhost

Port: 5432

Servei:

BBDD de mantenim.: rxx

Nom d'usuari: rxx

Contrasenya: ...

Desa la contrasenya: ☐

Color:

Grup: Servidors

Ajuda D'acord Cancel·la

Si NO voleu instal·lar-vos el servidor PostgreSQL:

Utilitzarem com a mínim el programa **PgAdmin III**, que ens permet connectar de forma còmoda amb el servidor. En els apunts penjats al final, en la secció d'annexos, teniu la manera d'instal·lar **PgAdmin III**, en la pregunta **2.5**, tant per a Windows com per a Ubuntu. Són versions endarrerides, però la manera de funcionar serà la mateixa.

Posteriorment ens haurem de crear una connexió amb el servidor (adreça **89.36.214.106**) a la Base de Dades i usuari que us especificarà el professor. En la imatge teniu un exemple, però reordeu que heu de canviar **rxx** per l'usuari i BD que us dirà el professor

Nou Registre de Servidor

Propietats SSL Tunnel SSH Avançat

Nom institut - rxx

Ordinador central 89.36.214.106

Port 5432

Servei

BBDD de mantenim. rxx ▼

Nom d'usuari rxx

Contrasenya ...|

Desa la contrasenya ☐

Color

Grup Servidors ▼

Ajuda D'acord Cancel·la

2.1 - Característiques

Les característiques del **Model Relacional Orientat a Objectes** (o **Objecte-Relacional**) es van definir en l'estàndard SQL de 1999, també anomenat SQL99. Aquestos són els aspectes més importants:

- Suport per a **tipus de dades bàsics i complexes**. L'usuari podrà crear els seus propis tipus de dades.
- **Arrays i/o col·leccions** en una columna. Es pot guardar, per tant, més d'un valor en una columna.
- Suport per a definir **mètodes** per a aquests tipus de dades.
- **Gestió** de tipus de dades **complexes** amb un mínim esforç.
- **Herència**.
- **Taules anidades**

I tot això mantenint la compatibilitat amb les Bases de Dades Relacionals tradicionals.

L'inconvenient és que augmenta la complexitat del sistema.

2.2 - Tipus de dades

Es tracta d'una revisió profunda que afegeix un nombre considerable de tipus poc convencionals, a més de permetre també la definició de tipus compostos o estructurats de dades. Els principals tipus que incorpora són:

- **Boolean.** Fins aleshores, els valors lògics se solien indicar utilitzant el tipus BIT.
- **Grans objectes.** SQL99 defineix dos tipus d'objectes grans, l'anomenat **BLOB** (Binary Large Object), adequat per a guardar dades binàries com ara imatges, vídeos, música, certificats digitals, etc. L'altre tipus s'anomena **CLOB** (Character Large Object), ideal per a dades extenses de tipus text com ara informes, pàgines web, articles, etc.
- **Col·leccions.** Permet guardar de forma directa col·leccions senceres de dades tant de tipus bàsic com de tipus estructurat.
- **Tipus compostos o estructurats.** Gràcies a la incorporació d'aquests tipus de dades és possible arribar als tipus de dades definits per l'usuari.
- **Referències a tipus estructurats.** Es tracta d'un tipus especial que actua com a punter de tipus compostos. Són útils perquè permeten fer una abstracció del lloc (taula) on realment es guardaran aquests tipus. El sistema està preparat per realitzar un emmagatzematge per defecte, de manera que aquests tipus són l'única manera de referenciar les dades guardades.

La incorporació de tots aquests tipus de dades aporta molta més flexibilitat a l'hora de fer el mapatge de les classes del model fent servir directament el llenguatge de definició DDL. A més, SQL amplia la sintaxi del llenguatge de consulta per poder utilitzar directament els nous tipus, de manera semblant a la manipulació dels atributs dels objectes.

A pesar de que no es tracta d'una revisió recent, la profunditat del canvi necessari per incorporar aquests nous tipus fa que a dia d'avui encara hi ha SGBD que no aconsegueixen tota l'especificació de l'any 1999. Per exemple, PostgreSQL no suporta encara l'ús de referències a tipus estructurats, cosa que és un gran inconvenient. A més, la sintaxi utilitzada pels diferents SGBD Objecte-Relacionals és menys estandarditzada que les revisions anteriors. Són elements a tenir en compte a l'hora d'escollir aquesta solució, ja que ens podem trobar amb aplicacions difícilment portables.

Anem a veure l'aplicació a **PostgreSQL** de cadascun dels tipus anteriors definits per l'estàndard SQL99, ja que PostgreSQL no segueix al 100% aquest estàndard (com ningú).

Totes les proves les podeu fer des de l'usuari i Base de Dades que us proporcionarà el professor, que han de ser diferents uns dels altres per a no interferir.

BOOLEAN

PostgreSQL suporta el tipus BOOLEAN tal i com ve marcat per l'estàndard. Per exemple, una taula on tenim un camp que marca si una persona és major d'edat:

```
CREATE TABLE PERSONA (  
    nif VARCHAR(10) PRIMARY KEY,  
    nom VARCHAR(25) ,  
    major_edat BOOLEAN )
```

Grans objectes

El tipus de dades que PostgreSQL destina per emmagatzemar dades binàries grans (BLOB) és **BYTEA**, i el substitut de CLOB és **TEXT**. Ambdós tipus s'adapten de forma dinàmica a la quantitat de dades que es guarden. No es pot indicar la mida. Per exemple:

```
CREATE TABLE PERSONA2 (  
    nif VARCHAR(10) PRIMARY KEY,
```



```
nom VARCHAR(25),  
major_edat BOOLEAN ,  
foto BYTEA,  
curriculum TEXT )
```

Col·leccions

PostgreSQL només admet **ARRAY** per a fer col·leccions, és a dir, columnes que poden contenir més d'un valor. En l'exemple ens definim un camp per a guardar més d'un telèfon (l'alternativa que ens donaria la normalització del Model Relacional és posar els telèfons en una altra taula).

```
CREATE TABLE PERSONA3 (  
    nif VARCHAR(10) PRIMARY KEY,  
    nom VARCHAR(25),  
    major_edat BOOLEAN ,  
    foto BYTEA,  
    curriculum TEXT,  
    telefons VARCHAR(12) ARRAY )
```

O de forma alternativa també podríem haver fet:

```
telefons VARCHAR(12) []
```

Tipus compostos o estructurats

Es fa amb la sentència **CREATE TYPE**, on es posen entre parèntesi i separats per comes, els camps que formen el tipus compost.

```
CREATE TYPE t_adreca AS(  
    carrer VARCHAR(255),  
    codipostal VARCHAR(7),  
    poblacio VARCHAR(100)  
);
```

```
CREATE TYPE t_telefon AS(  
    mobil BOOLEAN,  
    numero VARCHAR(20)  
);
```

```
CREATE TABLE PERSONA4 (  
    nif VARCHAR(10) PRIMARY KEY,  
    nom VARCHAR(25),  
    major_edat BOOLEAN ,  
    foto BYTEA,  
    curriculum TEXT,  
    adreca t_adreca,  
    correus_e VARCHAR(50) ARRAY,  
    telefons t_telefon ARRAY  
);
```

Observa com els tipus es defineixen com les taules, però sense restriccions ni de clau principal, ni de clau externa de no nul, ni de res. Després ens podrem definir camps de les taules d'aquests tipus. Fins i tot podem crear arrays d'aquests tipus, com en el cas dels telèfons.

En les últimes versions de PostgreSQL els camps de tipus compostos i els arrays poden ser clau principal, i clau

externa sense problemes.

2.3 - Manipulació de dades

Anem primer a veure com accedir a les dades dels tipus nous des de SQL, que en el cas de arrays i tipus estructurats no ho hem vist encara. Ens referim més concretament a les sentències DML, és a dir un INSERT, on hem d'especificar les dades concretes, o un UPDATE.

Posteriorment veurem com accedir des de Java a través de JDBC.

2.3.1 - Sentències SQL per a tipus de dades nous

Anem a comentar breument com especificarem les dades per als tipus de dades nous, posant diferents exemples en cada cas. De moment ens deixem els tipus BLOB i CLOB (BYTEA i TEXT), perquè des de PgAdmin no podem donar contingut. Ens els deixem per a accés des de JDBC.

Boolean

El millor és utilitzar els valors **TRUE** i **FALSE**, en majúscules o minúscules, i sense cometes.

```
INSERT INTO PERSONA4(nif,nom,major_edat)
VALUES ('11111111a','Albert',TRUE);
```

PostgreSQL pot fer conversions de text a boolean, i en aquest cas pot ser un text que siga igual a **TRUE** o **FALSE** o que comence per alguns d'aquests caràcters. També valdria posar 1 o 0 entre cometes.

```
INSERT INTO PERSONA4(nif,nom,major_edat)
VALUES ('22222222b','Bernat',false);
```

Observeu com ara per a seleccionar els que són majors d'edat comparem amb 'T'. Per tant fa una conversió de text a boolean. I val la inicial. Igual valdria en minúscula.

```
SELECT * FROM PERSONA4
WHERE major_edat='T';
```

Aquest seria el resultat:

	nif character varying(10)	nom character varying(25)	major_edat boolean	foto bytea	curriculum text	adreca t_adreca	correus_e character varying(50)[telefons t_telefon[
1	11111111a	Albert	t					

Array

Per a poder especificar el conjunt de valors, utilitzarem el constructor ARRAY, i posarem els diferents valors entre claudàtors ([]) i separats per comes.

```
INSERT INTO PERSONA4(nif,nom,major_edat,correus_e)
VALUES
('33333333c','Clàudia',TRUE,ARRAY['alu33333333c@ieselcaminas.org','claudia@gmail.co
```

En algunes ocasions és suficient amb posar-lo entre claus ({ }), sense haver d'utilitzar el constructor **ARRAY**. És com si indroduïrem els valor directament en la taula, des del PgAdmin, sense utilitzar sentència **INSERT**, encara que en realitat **PgAdmin** el que fa és construir ell la sentència SQL.

En les sentències SQL hem de posar-lo tot entre cometes simples. Com en el cas dels correus electrònics els elements són de text, haurem de jugar amb les cometes simples i dobles.

```
UPDATE PERSONA4
SET correus_e = '{"alu22222222b@ieselcaminas.org"}'
WHERE nom = 'Bernat';
```

En aquest moment el contingut de la taula hauria de ser:

	nif [PK] character	nom character	major_edat boolean	foto bytea	curri text	adreca t_adre	correus_e character varying(50)[telefons t_telefon[
1	11111111a	Albert	TRUE	<dades binàries>				
2	22222222b	Bernat	FALSE	<dades binàries>			{alu22222222b@ieselcaminas.org}	
3	33333333c	Clàudia	TRUE	<dades binàries>			{alu33333333c@ieselcaminas.org,claudia@gmail.com}	
*								

Per a fer referència a un determinat element de l'array (el primer, el segon, ...) posarem entre claudàtors l'índex, després del nom del camp. Aquest índex comença per 1.

```
SELECT nif,nom,major_edat,correus_e[1]
FROM PERSONA4;
```

	nif character varying(10)	nom character varying(25)	major_edat boolean	correus_e character varying(50)
1	11111111a	Albert	t	
2	33333333c	Clàudia	t	alu33333333c@ieselcaminas.org
3	22222222b	Bernat	f	alu22222222b@ieselcaminas.org

Per a poder buscar un element dins d'un array, no cal comparar amb cadascun dels elements, sinó que podem utilitzar **ANY**:

```
SELECT nif,nom,major_edat,correus_e
FROM PERSONA4
WHERE 'claudia@gmail.com' = ANY (correus_e);
```

	nif character varying(10)	nom character varying(25)	major_edat boolean	correus_e character varying(50)[]
1	33333333c	Clàudia	t	{alu33333333c@ieselcaminas.org,claudia@gmail.com}

Si volem afegir un element a un determinat array, des de SQL podem utilitzar funcions de manipulació d'arrays, com per exemple **array_append(array,element)**, que afegeix un element al final. En el següent exemple modifiquem els correus d'una persona, afegint-li un al final.

```
UPDATE PERSONA4
SET correus_e = array_append(correus_e,'bernat@gmail.com')
WHERE nom = 'Bernat';
```

Ara el contingut de la taula ja contindrà aquest segon correu de Bernat:

	nif [PK] character varying(10)	nom character varying(25)	major_edat boolean	foto bytea	curriculum text	adreca t_adreca	correus_e character varying(50)[]	telefons t_telefon[]
1	11111111a	Albert	TRUE	<dades binàries>				
2	22222222b	Bernat	FALSE	<dades binàries>			{alu22222222b@ieselcaminas.org,bernat@gmail.com}	
3	33333333c	Clàudia	TRUE	<dades binàries>			{alu33333333c@ieselcaminas.org,claudia@gmail.com}	
*								

Estructurat

Els valors es representaran entre parèntesis i separats per comes. En el nostre exemple tenim el tipus estructurat **t_adreca**, format per **carrer**, **codipostal** i **poblacio**.

```
UPDATE PERSONA4
SET adreca=('C/Major, 7','12001','Castelló')
WHERE nif='11111111a';
```

El contingut de la taula serà ara:

	nif [PK] character varying(10)	nom character varying(25)	major_edat boolean	foto bytea	curriculum text	adreca t_adreca	correus_e character varying(50)[]	telefons t_telefon[]
1	11111111a	Albert	TRUE	<dades binàries>		('C/Major, 7',12001,Castelló)		
2	22222222b	Bernat	FALSE	<dades binàries>			{alu22222222b@ieselcaminas.org,bernat@gmail.com}	
3	33333333c	Clàudia	TRUE	<dades binàries>			{alu33333333c@ieselcaminas.org,claudia@gmail.com}	
*								

Per a accedir a un camp en concret del tipus estructurat, posarem el nom de la columna i després, separat per un punt, el nom del camp estructurat. Tindrem la dificultat, però, de que amb aquesta sintaxi, PostgreSQL es pensa que la columna ha de ser una taula. Per a que no es duga a engany, posarem el nom de la columna (que és de tipus estructurat) entre parèntesis:

```
SELECT nif, nom, (adreca).poblacio
FROM PERSONA4;
```

Aquest serà el resultat de la sentència

	nif character varying(10)	nom character varying(25)	poblacio character varying(100)
1	33333333c	Clàudia	
2	22222222b	Bernat	
3	11111111a	Albert	Castelló

El problema anterior només el tenim en el SELECT, que és on pot haver confusió. En canvi en aquest cas no fa falta posar el nom de la columna entre parèntesis, perquè no hi ha confusió possible: adreca ha de ser un camp de la taula PERSONA4.

```
UPDATE PERSONA4
SET adreca.poblacio = 'Castelló';
```

El contingut de la taula serà ara:

	nif [PK] character	nom character	major_edat/ boolean	foto bytea	curriculum text	adreca t_adreca	correu_e character varying(50)[]	telefon t_telefon[
1	11111111a	Albert	TRUE	<dades binàries>		("C/Major, 7",12001,Castelló)		
2	22222222b	Bernat	FALSE	<dades binàries>		(,,Castelló)	{alu22222222b@ieselcaminas.org,bernat@gmail.com}	
3	33333333c	Clàudia	TRUE	<dades binàries>		(,,Castelló)	{alu33333333c@ieselcaminas.org,claudia@gmail.com}	
*								

Mirem aquest últim exemple, on es barregen els arrays i tipus estructurats. És l'exemple dels telèfons, que és un **array** de **t_telefon**, el qual és un tipus estructurat amb els camps **mobil** (booleà) i **número** (de text).

```
INSERT INTO PERSONA4(nif,nom,telefon)
VALUES ('44444444d','David', CAST (ARRAY[ (false, '964112233') , (true, '666777888') ]
AS t_telefon ARRAY) );
```

Ens hem vist obligats a utilitzar la funció CAST per a dir a PostgreSQL que el tipus del que se li passa a continuació és un array de t_telefon: **CAST (.... AS t_telefon ARRAY)**

El contingut de la taula serà ara:

	nif [PK] character	nom character	major_edat/ boolean	foto bytea	curriculum text	adreca t_adreca	correu_e character varying(50)[]	telefon t_telefon[
1	11111111a	Albert	TRUE	<dades binàries>		("C/Major, 7",12001,Castelló)		
2	22222222b	Bernat	FALSE	<dades binàries>		(,,Castelló)	{alu22222222b@ieselcaminas.org,bernat@gmail.com}	
3	33333333c	Clàudia	TRUE	<dades binàries>		(,,Castelló)	{alu33333333c@ieselcaminas.org,claudia@gmail.com}	
4	44444444d	David		<dades binàries>				{("f,964112233"), ("t,666777888")}
*								

2.3.2 - Accés a través de JDBC

Una vegada ens hem familiaritzat amb els nous tipus de dades fent sentències SQL de forma còmoda, anem al que ens interessa, que és accedir des de Java a través de JDBC. També veurem l'accés als camps de tipus **BYTEA** (BLOB) i **TEXT** (CLOB). JDBC es va haver d'adaptar a aquests nous tipus de dades, incorporant classes i mètodes, a partir de la versió 2.0.

Tots aquests exemples els farem en un projecte nou anomenat **Tema6_1**, dins d'un paquet anomenat **Exemples**. Al projecte, evidentment, li haurem d'afegir el driver de PostgreSQL.

Recordeu que heu de substituir **rx** pel vostre usuari i Base de Dades en el servidor de l'institut, i si treballeu sobre el vostre servidor, haureu de canviar l'adreça IP per **localhost**.

2.3.2.1 Boolean

Boolean

No ofereix cap dificultat. Com és un boolean, que és un tipus vàlid de Java, utilitzarem el mètode del ResultSet **getBoolean(index)** (setBoolean(index,valor) per a posar un valor en un PreparedStatement) i directament el tindrem disponible en Java. En el següent exemple es fa la connexió, i s'agafa el **nom** i el camp **major_edat** de totes les persones (**persona4**), i es fa una comprovació sobre si és major d'edat.

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

public class Proval {

    public static void main(String[] args) throws SQLException {

        String url = "jdbc:postgresql://89.36.214.106:5432/rxx";

        Connection con = DriverManager.getConnection(url, "rxx", "rxx");

        ResultSet rs = con.createStatement().executeQuery("select
nom,major_edat from persona4 order by nom");

        while (rs.next()) {
            if (rs.getBoolean(2))
                System.out.println(rs.getString(1) + " és major d'edat");
            else
                System.out.println(rs.getString(1) + " és menor d'edat");
        }
        rs.close();
        con.close();
    }
}
```

Recordeu que quan agafem del ResultSet, el primer camp correspon a l'índex 1 (no 0). I observeu també com per a l'última persona, David, no havíem introduït el camp major_edat, i en la pràctica és equivalent a false.

2.3.2.2 BLOB i CLOB

BLOB i CLOB

Comencem per intentar introduir informació, i després la intentarem recuperar. Intentarem introduir un nou registre en **PERSONA4**, amb un currículum provinent d'un fitxer i també amb una foto. El fitxer el podeu crear amb un editor de textos, per a més comoditat. La foto, podeu agafar qualsevol. Respectivament s'hauran d'anomenar **c_eva.txt** i **foto_eva.jpg**

Per a poder introduir les dades de tipus complexos, ens serà molt més còmoda la classe **PreparedStatement**, ja que podem posar paràmetres per als valors a introduir, i els paràmetres els podem posar del tipus que ens convinga. Concretament, per a BLOB (BYTEA) i CLOB (TEXT) utilitzarem els mètodes de **PreparedStatement** **setBinaryStream(index, inputStream, grandària)** i **setCharacterStream(index, reader, grandària)**. També existeix el mètode **setAsciiStream(index, inputStream, grandària)**, però no ens assegura agafar bé tots els caràcters, únicament els caràcters ASCII no especials.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class Prova2 {

    public static void main(String[] args) throws SQLException, IOException {
        String url = "jdbc:postgresql://89.36.214.106:5432/rxx";

        Connection con = DriverManager.getConnection(url, "rxx", "rxx");

        File f1 = new File("c_eva.txt");
        FileReader in1 = new FileReader(f1);
        File f2 = new File("foto_eva.png");
        FileInputStream in2 = new FileInputStream(f2);

        PreparedStatement st = con.prepareStatement("INSERT INTO persona4
(nif,nom,curriculum,foto) VALUES(?,?,?,?)");
        st.setString(1, "55555555e");
        st.setString(2, "Eva");
        st.setCharacterStream(3, in1, (int) f1.length());
        st.setBinaryStream(4, in2, (int) f2.length());
        st.executeUpdate();
        in1.close();
        in2.close();
        con.close();
    }
}
```

Hem creat la referència als fitxers en 2 passos, primer el **File** i després el **FileInputStream**, per a poder disposar de la llargària del fitxer (mètode de **File**). Observeu com per a posar el paràmetre corresponent al fitxer de text hem utilitzat el mètode **setCharacterStream()**, mentre que per al fitxer binari (per al camp **foto**) hem utilitzat **setBinaryStream()**.

Si mirem el contingut de la taula des del PgAdmin III, podem veure el contingut del camp de tipus TEXT, però no el de tipus BYTEA

nif	nom	major_edat	foto	curriculum	adreca	correus_e	telefons
[PK] character	character	boolean	bytea	text	t_adreca	character varying(50)	t_telefon
1	11111111a	Albert	TRUE	<dades binàries>	("C/Major, 7",12001,Castelló)		
2	22222222b	Bernat	FALSE	<dades binàries>	(,,Castelló)	{alu22222222b@ieselcaminas.org,bernat@gmail.com}	
3	33333333c	Clàudia	TRUE	<dades binàries>	(,,Castelló)	{alu33333333c@ieselcaminas.org,claudia@gmail.com}	
4	44444444d	David		<dades binàries>			{("f,964112233)","(t,666777888)"}
5	55555555e	Eva		<dades binàries>	CURRÍCULUM DE E		

Per a poder llegir el camp de grans dimensions, utilitzarem els mètodes de **ResultSet** **getBinaryStream(index)** i **getCharacterStream(index)**, que tornen un **InputStream** i un **Reader** respectivament, que els podem gestionar com qualsevol **InputStream** o **Reader** utilitzat en el tema 2. De moment només llegirem el fitxer de text:

```
import java.io.BufferedReader;
```

```
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;

public class Prova3 {

    public static void main(String[] args) throws SQLException, IOException {
        String url = "jdbc:postgresql://89.36.214.106:5432/rxx";

        Connection con = DriverManager.getConnection(url, "rxx", "rxx");

        ResultSet rs = con.createStatement().executeQuery(
            "SELECT nom,curriculum FROM persona4 WHERE nom='Eva'");
        while (rs.next()) {
            System.out.println(rs.getString(1));
            BufferedReader br = new BufferedReader(rs.getCharacterStream(2));
            String s = br.readLine();
            while (s != null) {
                System.out.println(s);
                s = br.readLine();
            }
            rs.close();
            con.close();
        }
    }
}
```

Per a procedir amb un fitxer binari, ho fariem de forma absolutament similar, però amb el mètode **getBinaryStream(index)**.

En realitat, si el camp de text no és molt gran, de manera que cap en un String, podem utilitzar sense problemes **getString(index)**. El que hem vist anteriorment s'utilitzaria per a textos molt grans.

2.3.2.3 Arrays

Arrays

Havíem vist que el millor per a poder introduir els valors d'un array en una sentència SQL era utilitzar el constructor ARRAY, posant entre claudàtors i separats per comes els distints valors. Evidentment, des de Java a través de JDBC també podem utilitzar aquesta manera. Però anem a veure una forma alternativa utilitzant el **PreparedStatement**, amb paràmetres, i per a posar el valor de l'array gastarem el mètode **setArray(index,array)**, on el segon paràmetre és del tipus **java.sql.Array**, i per tant no és un array habitual de Java, però que podem convertir-lo fàcilment, tal i com es veu en el següent exemple:

```
import java.sql.Array;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class Prova4 {

    public static void main(String[] args) throws SQLException {
        String url="jdbc:postgresql://89.36.214.106:5432/rxx";

        Connection con = DriverManager.getConnection(url, "rxx", "rxx");

        PreparedStatement st = con.prepareStatement("INSERT INTO
        persona4(nif,nom,correus_e) VALUES(?,?,?)");
        st.setString(1,"66666666f");
        st.setString(2, "Ferran");
        String[] correus =
        {"alu66666666f@ieselcaminas.org","ferran@gmail.com","f_66@hotmail.com"};
        Array c = con.createArrayOf("varchar", correus);
        st.setArray(3, c);
        st.executeUpdate();
        con.close();
    }
}
```

Aquest serà ara el contingut de la taula:

	nif [PK] charac	nom characte	major boole	foto bytea	curricu text	adreca t_adreca	correus_e character varying(50)[]	telefons t_telefon[]
1	111111111a	Albert	TRUE	<binary data>		('C/Major, 7",1		
2	222222222b	Bernat	FALSE	<binary data>		(,,Castelló)	{alu22222222b@ieselcaminas.org,bernat@gmail.com}	
3	333333333c	Clàudia	TRUE	<binary data>		(,,Castelló)	{alu333333333c@ieselcaminas.org,claudia@gmail.com}	
4	444444444d	David		<binary data>				{"(f,964112233)","(t,666777888)"}
5	555555555e	Eva		<binary data>	CURRIC			
6	666666666f	Ferran		<binary data>			{alu666666666f@ieselcaminas.org,ferran@gmail.com,f_66@hotmail.com}	
*								

Com es pot observar, a partir d'un array de Strings (**correus**) hem construït l'array de tipus **java.sql.Array** amb el mètode **createArrayOf(tipus,array)** de la connexió (per tant de JDBC). Després només cal inicialitzar el paràmetre amb **setArray(index,array)** del statement. Ho podríem haver fet tot en una línia:

```
st.setArray(3, con.createArrayOf("varchar", new String[]
{"alu66666666f@ieselcaminas.org", "ferran@gmail.com", "f_66@hotmail.com"}));
```

Per a fer el procés invers, és a dir, pera poder llegir un array i utilitzar-lo en Java, utilitzarem el mètode **getArray(index)** del ResultSet, que torna un **java.sql.Array**. Per a passar aquest a un array normal de Java, utilitzarem el mètode **getArray()**, que encara que es diu igual que l'anterior, ara és un mètode de **java.sql.Array**, i no té paràmetres.

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

public class Prova5 {

    public static void main(String[] args) throws SQLException {
        String url = "jdbc:postgresql://89.36.214.106:5432/rxx";

        Connection con = DriverManager.getConnection(url, "rxx", "rxx");
```

```
        ResultSet rs = con.createStatement().executeQuery("SELECT  
nom,correus_e FROM persona4 WHERE nom='Ferran'");  
        while (rs.next()) {  
            System.out.println("Correus de " + rs.getString(1));  
            String[] correus = (String[]) rs.getArray(2).getArray();  
            for (String c : correus) {  
                System.out.println(c);  
            }  
        }  
        rs.close();  
        con.close();  
    }  
}
```

Evidentment, si de la sentència SQL obtenim només un element de l'array, el mètode del ResultSet que hem d'utilitzar és el **get** del tipus elemental.

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.ResultSet;  
import java.sql.SQLException;  
  
public class Prova6 {  
  
    public static void main(String[] args) throws SQLException {  
        String url = "jdbc:postgresql://89.36.214.106:5432/rxx";  
  
        Connection con = DriverManager.getConnection(url, "rxx", "rxx");  
  
        ResultSet rs = con.createStatement().executeQuery("SELECT  
nom,correus_e[1] FROM persona4 ORDER BY nom");  
        while (rs.next()) {  
            System.out.print("Primer correu de " + rs.getString(1) + ": ");  
            System.out.println(rs.getString(2));  
        }  
        rs.close();  
        con.close();  
    }  
}
```

2.3.2.4 Estructurat

Estructurat

En el cas dels tipus definits per l'usuari en PostgreSQL, lamentablement les coses no funcionen tan bé. I és una pena, perquè si poguérem ens estalviariem molta de la feina de mapatge fent la correspondència entre els tipus definits en PostgreSQL i classes definides en Java.

Els mètodes que s'utilitzaran ara són **getObject(index)** del ResultSet, i per a poder guardar el **setObject(index,objecte)** del PreparedStatement.

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

public class Prova7_1 {
    public static void main(String[] args) throws SQLException {
        String url="jdbc:postgresql://89.36.214.106:5432/rxx";

        Connection con = DriverManager.getConnection(url, "rxx", "rxx");

        ResultSet rs = con.createStatement().executeQuery("SELECT nom,adreca
FROM persona4 ORDER BY nom");
        while (rs.next()){
            System.out.println("Adreça de " + rs.getString(1) + ": ");
            System.out.println(rs.getObject(2));
        }
    }
}
```

Podem accedir a l'adreça que és del tipus estructurat definit per nosaltres (amb carrer, codi postal i població).

Per tant sembla que va bé la cosa, però no. El problema és accedir a les dades internes de l'objecte. Si poguérem fer una conversió entre el tipus de dades definit en PostgreSQL (**t_adreca**) i una classe definida en Java que continga 3 propietats per al carrer, el codi postal i la població, ja ho tindríem. Però hi ha documentació per Internet que diu que no es pot fer. És a dir, el següent programa (suposem que tenim una classe anomenada **Adreca** que consta de 3 strings) :

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

public class Prova7_2 {
    public static void main(String[] args) throws SQLException {
        String url = "jdbc:postgresql://89.36.214.106:5432/rxx";

        Connection con = DriverManager.getConnection(url, "rxx", "rxx");

        ResultSet rs = con.createStatement().executeQuery("SELECT nom,adreca
FROM persona4 ORDER BY nom");
        while (rs.next()) {
            System.out.println("Adreça de " + rs.getString(1) + ": ");
            Adreca adr = (Adreca) rs.getObject(2);
            System.out.println(adr.getCarrer() + ". " + adr.getCodipostal() +
" (" + adr.getPoblacio() + ")");
        }
        rs.close();
        con.close();
    }
}
```

dóna el següent error:

```
Exception in thread "main" java.lang.ClassCastException:
org.postgresql.util.PGobject cannot be cast to Exemples.Adreca
at Exemples.Prova7_2.main(Prova7_2.java:16)
```

És a dir, no s'ha pogut convertir l'objecte agafat des de PostgreSQL a la classe definida en Java (recordeu que suposem que tenim una classe anomenada **Adreca** que consta de 3 strings).

Es pot arreglar, però l'esforç segurament no valdrà la pena.

La manera que he tingut de solucionar-lo ha estat buscar un altre driver JDBC que sí que ens permet fer una conversió entre els tipus que venen des de PostgreSQL. Aquest nou driver s'anomena **pgjdbc-ng** i el podeu trobar a la següent adreça:

<https://impossibl.github.io/pgjdbc-ng>

El següent programa sí que funcionarà. Observeu com la manera de connectar és lleugerament diferent (mreu la URL). Hem utilitzat un objecte de tipus **Struct** per a poder arreplegar el resultat.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Struct;

public class Prova7_3 {
    public static void main(String[] args) throws SQLException,
        ClassNotFoundException {
        String url = "jdbc:postgresql://89.36.214.106:5432/rxx";

        Connection con = DriverManager.getConnection(url, "rxx", "rxx");

        ResultSet rs = con.createStatement().executeQuery("SELECT nom,adreca
FROM persona4 ORDER BY nom");
        while (rs.next()) {
            System.out.println("Adreça de " + rs.getString(1) + ": ");

            Struct adr = (Struct)rs.getObject(2);
            if (adr != null)
                System.out.println(adr.getAttributes()[0] + ". " +
adr.getAttributes()[1] + " (" + adr.getAttributes()[2] + ")");
            }
        rs.close();
        con.close();
    }
}
```

I produiria el següent resultat:

```
Adreça de Albert:
C/Major, 7. 12001 (Castelló)
Adreça de Bernat:
null. null (Castelló)
Adreça de Clàudia:
null. null (Castelló)
Adreça de David:
Adreça de Eva:
Adreça de Ferran:
```

En aquest exemple fem la reconversió completa de **t_adreca** a **Adreca**. En aquesta classe, **Adreca**, implementem un constructor que agafa com a paràmetre un **Struct** (que ha estat la manera de salvar l'inconvenient en l'exemple anterior), i inicialitza les propietats a partir d'ell;

```
import java.sql.SQLException;
import java.sql.Struct;

public class Adreca {

    private String carrer = null;
    private String codipostal = null;
    private String poblacio = null;

    public Adreca() {
    }

    public Adreca(String c, String cp, String p) {
        this.carrer = c;
        this.codipostal = cp;
        this.poblacio = p;
    }

    public Adreca(Struct t_adr) throws SQLException {
        if (t_adr != null) {
            this.carrer = (String) t_adr.getAttributes()[0];
        }
    }
}
```

```

        this.codipostal = (String) t_adr.getAttributes()[1];
        this.poblacio = (String) t_adr.getAttributes()[2];
    }

    public String getCarrer() {
        return carrer;
    }

    public void setCarrer(String carrer) {
        this.carrer = carrer;
    }

    public String getCodipostal() {
        return codipostal;
    }

    public void setCodipostal(String codipostal) {
        this.codipostal = codipostal;
    }

    public String getPoblacio() {
        return poblacio;
    }

    public void setPoblacio(String poblacio) {
        this.poblacio = poblacio;
    }
}

```

Això ens permet fer el següent programa que ara sí que ens funcionarà. En compte de fer un **cast** utilitzem el constructor, i igual queda còmode. Hem aprofitat per traure només aquells que tenen alguna cosa en l'adreça.

```

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

public class Prova7_4 {
    public static void main(String[] args) throws SQLException,
        ClassNotFoundException {
        String url = "jdbc:pgsql://89.36.214.106:5432/rxx";

        Connection con = DriverManager.getConnection(url, "rxx", "rxx");

        ResultSet rs = con.createStatement().executeQuery("SELECT nom, adreca
FROM persona4 ORDER BY nom");
        while (rs.next()) {
            if (rs.getObject(2) != null) {
                System.out.println("Adreça de " + rs.getString(1) + ": ");
                Adreca adr = new Adreca((Struct) rs.getObject(2));
                System.out.println(adr.getCarrer() + ". " +
adr.getCodipostal() + " (" + adr.getPoblacio() + ")");
            }
        }
        rs.close();
        con.close();
    }
}

```

Recordeu que heu d'incorporar el driver **pgjdbc-ng**

El que sí que podem fer sense problemes des del driver normal és accedir als camps del tipus estructurat des de SQL. Però açò ens obliga en certa manera a dur el mapatge manual.

```

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

public class Prova7_5 {

    public static void main(String[] args) throws SQLException {
        String url = "jdbc:postgresql://89.36.214.106:5432/rxx";

        Connection con = DriverManager.getConnection(url, "rxx", "rxx");

        ResultSet rs = con.createStatement().executeQuery(
            "SELECT nom, (adreca).carrer, (adreca).codipostal,
(adreca).poblacio FROM persona4 ORDER BY nom");
    }
}

```

```
        while (rs.next()) {
            System.out.println("Adreça de " + rs.getString(1) + ": ");
            System.out.println(rs.getString(2) + ". CP: " + rs.getString(3) +
" (" + rs.getString(4) + ")");
        }
    }
}
```

Mirem un altre exemple per a accedir als telèfons. Recordem que **telefonos** és un array de **t_telefon**, per tant estem barrejant un array amb un tipus estructurat. Podem agafar bé l'array (col·loquem l'array en un ResultSet per mig de **getResultSet**, que ens dóna tants registres com elements de l'array. En cada registre tenim en el primer camp el número d'ordre, i en el segon el valor (que en el nostre cas és **t_telefon**)

```
import java.sql.Array;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;

public class Prova8 {

    public static void main(String[] args) throws SQLException {
        String url="jdbc:postgresql://89.36.214.106:5432/rxx";

        Connection con = DriverManager.getConnection(url, "rxx", "rxx");

        ResultSet rs = con.createStatement().executeQuery("SELECT nom,telefonos
FROM persona4 order by nom");
        while (rs.next()) {
            System.out.println("Telèfons de " + rs.getString(1));
            Array tels = rs.getArray(2);
            if (tels != null) {
                ResultSet rs2 = tels.getResultSet();
                while (rs2.next())
                    System.out.println(" " + rs2.getString(2));
            }
            else
                System.out.println(" No en té");
        }
    }
}
```


2.3.2.5 Exemple de tot junt

Per últim intentarem posar totes les dades, incloent la **foto** en una aplicació gràfica. Només mostrarem les dades de Eva, i així podrem veure la foto.

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.Image;
import java.io.BufferedReader;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.imageio.ImageIO;
import javax.swing.ImageIcon;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class Prova9_Pantalla extends JFrame{
    JTextField nif = new JTextField(9);
    JTextField nom = new JTextField(9);
    JCheckBox major = new JCheckBox();
    JLabel foto = null;
    JTextArea curric = new JTextArea();
    JTextField adreca = new JTextField(20);
    JTextArea correus = new JTextArea();
    JTextArea telefons = new JTextArea();

    JLabel et_nif = new JLabel("Nif");
    JLabel et_nom = new JLabel("Nom");
    JLabel et_major = new JLabel("Major d'edat");
    JLabel et_adr = new JLabel("Adreça");
    JLabel et_correus = new JLabel("Correus");
    JLabel et_telefons = new JLabel("Telèfons");

    JPanel pan1 = new JPanel(new GridLayout(1,2));
    JPanel pan1_1 = new JPanel(new GridLayout(3,1));
    JPanel pan1_1_1 = new JPanel(new FlowLayout());
    JPanel pan1_1_2 = new JPanel(new FlowLayout());
    JPanel pan1_1_3 = new JPanel(new FlowLayout());
    JPanel pan1_2 = new JPanel(new FlowLayout());
    JPanel pan2 = new JPanel(new BorderLayout());
    JPanel pan2_1 = new JPanel(new FlowLayout());
    JPanel pan2_2 = new JPanel(new GridLayout(2,2));
    JPanel pan3 = new JPanel(new BorderLayout());
    JPanel pan4 = new JPanel();

    public void iniciar() throws SQLException, IOException,
    NoSuchMethodException, SecurityException, IllegalAccessException,
    IllegalArgumentException, InvocationTargetException {
        this.setLayout(new GridLayout(3,1));
        this.setBounds(100, 100, 300, 300);

        this.getContentPane().add(pan1);
        this.getContentPane().add(pan2);
        this.getContentPane().add(pan3);

        pan1.add(pan1_1);
        pan1.add(pan1_2);
        pan1_1.add(pan1_1_1);
        pan1_1.add(pan1_1_2);
        pan1_1.add(pan1_1_3);
        pan1_1_1.add(et_nif);
        pan1_1_1.add(nif);
        pan1_1_2.add(et_nom);
        pan1_1_2.add(nom);
        pan1_1_3.add(et_major);
        pan1_1_3.add(major);

        pan2.add(pan2_1, BorderLayout.NORTH);
```

```

        pan2.add(pan2_2, BorderLayout.CENTER);
        pan2_1.add(et_adr);
        pan2_1.add(adreca);
        pan2_2.add(et_correus);
        pan2_2.add(et_telefons);
        pan2_2.add(correus);
        pan2_2.add(telefons);

        pan3.add(curric);

        String url = "jdbc:postgresql://89.36.214.106:5432/rxx";

        Connection con = DriverManager.getConnection(url, "rxx", "rxx");

        ResultSet rs = con.createStatement().executeQuery("SELECT * FROM
persona4 WHERE nom='Eva'");

        if (rs.next()){
            nif.setText(rs.getString(1));
            nom.setText(rs.getString(2));
            major.setSelected(rs.getBoolean(3));
            if (rs.getBinaryStream(4)!=null){
                Image img = ImageIO.read(rs.getBinaryStream(4));
                foto = new JLabel(new ImageIcon(img));
                pan1_2.add(foto);
            }
            if (rs.getCharacterStream(5)!=null){
                BufferedReader bf = new
BufferedReader(rs.getCharacterStream(5));
                String s;
                String tot="";
                while ((s = bf.readLine()) != null) {
                    tot += s + "\n";
                }
                curric.setText(tot);
            }
            if (rs.getObject(6)!=null){
                adreca.setText(rs.getObject(6).toString());
            }

            if (rs.getArray(7)!=null){
                String[] corr = (String[]) rs.getArray(7).getArray();
                for (String c : corr)
                    correus.append(c+"\n");
            }
            if (rs.getArray(8)!=null){
                ResultSet tels = rs.getArray(8).getResultSet();
                while (tels.next())
                    telefons.append(tels.getString(2)+"\n");
            }
        }

        rs.close();
        con.close();
        this.setVisible(true);
    }
}

```

I el programa principal, amb main() que el cridaria seria:

```

import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.sql.SQLException;

public class Prova9 {

    public static void main(String[] args) throws SQLException, IOException,
NoSuchMethodException, SecurityException, IllegalAccessException,
IllegalArgumentException, InvocationTargetException {
        Prova9_Pantalla finestra = new Prova9_Pantalla();
        finestra.iniciar();
    }
}

```

Aquest seria el resultat:

Nif 55555555e

Nom Eva

Major d'edat ☐

Adreça

Correus Telèfons

CURRÍCULUM DE EVA

=====

Molt treballadora i competent.

Ha estudiat:

- Primària
- ESO

I com a exemple final, fem una variació de l'anterior, però en aquesta ocasió més completa, per a mostrar com podem passar de la taula a objectes, utilitzant la tècnica vista en la pregunta 2.3.2.5, és a dir, utilitzant l'altre driver que ens permetia accedir bé al contingut dels tipus estructurats.

Mirem primer les classes necessàries. La classe **Adreca** ja l'havíem vista:

```
import java.sql.SQLException;
import java.sql.Struct;

public class Adreca {

    private String carrer = null;
    private String codipostal = null;
    private String poblacio = null;

    public Adreca() {
    }

    public Adreca(String c, String cp, String p) {
        this.carrer = c;
        this.codipostal = cp;
        this.poblacio = p;
    }

    public Adreca(Struct t_adr) throws SQLException {
        if (t_adr != null) {
            this.carrer = (String) t_adr.getAttributes()[0];
            this.codipostal = (String) t_adr.getAttributes()[1];
            this.poblacio = (String) t_adr.getAttributes()[2];
        }
    }

    public String getCarrer() {
        return carrer;
    }

    public void setCarrer(String carrer) {
        this.carrer = carrer;
    }

    public String getCodipostal() {
        return codipostal;
    }

    public void setCodipostal(String codipostal) {
        this.codipostal = codipostal;
    }

    public String getPoblacio() {
        return poblacio;
    }

    public void setPoblacio(String poblacio) {
        this.poblacio = poblacio;
    }
}
```

Ara ens farà falta també la classe **Telefon**. Observeu que també tenim el constructor al qual se li passa un **Struct**, que és el que utilitzarem.

```

import java.sql.SQLException;
import java.sql.Struct;

public class Telefon {
    private boolean mobil;
    private String numero;

    public Telefon(boolean m,String n){
        this.mobil=m;
        this.numero=n;
    }

    public Telefon(Struct t_tel){
        try {
            this.mobil=(boolean) t_tel.getAttributes()[0];
            this.numero=(String) t_tel.getAttributes()[1];
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public boolean isMobil() {
        return mobil;
    }

    public void setMobil(boolean mobil) {
        this.mobil = mobil;
    }

    public String getNumero() {
        return numero;
    }

    public void setNumero(String numero) {
        this.numero = numero;
    }
}

```

Per últim **Persona**. Hem utilitzat una tècnica un poc estranya: al constructor li passem un **ResultSet**, que suposem que estarà apuntant al contingut d'una fila de la taula **PERSONA4**.

```

import java.awt.Image;
import java.io.BufferedReader;
import java.io.IOException;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Struct;
import java.util.ArrayList;
import java.util.Arrays;

import javax.imageio.ImageIO;

public class Persona {
    private String nif;
    private String nom;
    private boolean majorEdat;
    private Image foto;
    private String curriculum;
    private Adreca adreca;
    private ArrayList<String> correusE;
    private ArrayList<Telefon> telefons;

    public Persona(String nif, String nom, boolean majorEdat, Image foto,
String curriculum, Adreca adreca,
        ArrayList<String> correusE, ArrayList<Telefon> telefons) {
        super();
        this.nif = nif;
        this.nom = nom;
        this.majorEdat = majorEdat;
        this.foto = foto;
        this.curriculum = curriculum;
        this.adreca = adreca;
        this.correusE = correusE;
        this.telefons = telefons;
    }

    public Persona(ResultSet rs) {
        super();
        try {
            this.nif = rs.getString(1);
            this.nom = rs.getString(2);
            this.majorEdat = rs.getBoolean(3);
            if (rs.getBinaryStream(4) != null)
                this.foto = ImageIO.read(rs.getBinaryStream(4));
            else
                this.foto=null;

```

```
        String tot = "";
        if (rs.getCharacterStream(5) != null) {
            BufferedReader bf = new
BufferedReader(rs.getCharacterStream(5));
            String s=null;

            while ((s = bf.readLine()) != null) {
                tot += s + "\n";
            }
        }
        this.curriculum = tot;

        this.adreca = new Adreca((Struct) rs.getObject(6));

        if (rs.getArray(7) != null)
            this.correusE = new ArrayList<String>
(Arrays.asList((String[]) rs.getArray(7).getArray()));
        else
            this.correusE = null;

        if (rs.getArray(8) != null){
            telefons = new ArrayList<Telefon>();
            ResultSet tels = rs.getArray(8).getResultSet();
            while (tels.next())
                telefons.add(new Telefon((Struct)tels.getObject(2)));
        }
        else
            this.telefons = null;

    } catch (SQLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public String getNif() {
    return nif;
}

public void setNif(String nif) {
    this.nif = nif;
}

public String getNom() {
    return nom;
}

public void setNom(String nom) {
    this.nom = nom;
}

public boolean isMajorEdat() {
    return majorEdat;
}

public void setMajorEdat(boolean majorEdat) {
    this.majorEdat = majorEdat;
}

public Image getFoto() {
    return foto;
}

public void setFoto(Image foto) {
    this.foto = foto;
}

public String getCurriculum() {
    return curriculum;
}

public void setCurriculum(String curriculum) {
    this.curriculum = curriculum;
}

public Adreca getAdreca() {
    return adreca;
}

public void setAdreca(Adreca adreca) {
    this.adreca = adreca;
}
```

```

    public ArrayList<String> getCorreusE() {
        return correusE;
    }

    public void setCorreusE(ArrayList<String> correusE) {
        this.correusE = correusE;
    }

    public ArrayList<Telefon> getTelefons() {
        return telefons;
    }

    public void setTelefons(ArrayList<Telefon> telefons) {
        this.telefons = telefons;
    }
}

```

Ara ja aniria el programa. Observeu què fàcil és des d'ací bolcar les dades des de la taula fins a un **ArrayList** d'objectes **Persona**. Hem fet també uns botons de moviment, per anar al registre següent i anterior (però en aquesta ocasió no els desactivem en arribar al principi o final; senzillament no funcionaran per a no eixir-nos del rang).

Com en el programa anterior, s'ha optat per construir molts panells, i anar col·locant tots els components en panells i subpanells. D'aquesta manera és més fàcil fer una redistribució de les coses. Però s'ha completat, i té un aspecte més acabat.

```

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;

import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingConstants;

public class Proval0_Pantalla extends JFrame implements ActionListener {
    JTextField nif = new JTextField(9);
    JTextField nom = new JTextField(9);
    JCheckBox major = new JCheckBox();
    JLabel foto = new JLabel("");
    JTextArea curric = new JTextArea();
    JTextField adreca = new JTextField(20);
    JTextArea correus = new JTextArea();
    JTextArea telefons = new JTextArea();

    ArrayList<Persona> llista = new ArrayList<Persona>();
    int indActual = 0;

    JLabel et_inicial = new JLabel("Visualització de les persones");
    JLabel et_nif = new JLabel("Nif");
    JLabel et_nom = new JLabel("Nom");
    JLabel et_major = new JLabel("Major d'edat");
    JLabel et_adr = new JLabel("Adreça");
    JLabel et_correus = new JLabel("Correus");
    JLabel et_telefons = new JLabel("Telèfons");

    JPanel panInicial = new JPanel(new FlowLayout());
    JPanel panCentral = new JPanel(new GridLayout(3, 1));
    JPanel pan1 = new JPanel(new GridLayout(1, 2));
    JPanel pan1_1 = new JPanel(new GridLayout(6, 2, 10, 10));
    JPanel pan1_1_1 = new JPanel(new FlowLayout());
    JPanel pan1_1_2 = new JPanel(new FlowLayout());
    JPanel pan1_1_3 = new JPanel(new FlowLayout());
    JPanel pan1_2 = new JPanel(new FlowLayout());
    JPanel pan2 = new JPanel(new BorderLayout());
    JPanel pan2_1 = new JPanel(new FlowLayout());

```

```

JPanel pan2_2 = new JPanel(new BorderLayout());
JPanel pan2_2_1 = new JPanel(new GridLayout(1,1,2, 2));
JPanel pan2_2_2 = new JPanel(new GridLayout(1,1,2, 2));
JPanel pan3 = new JPanel(new BorderLayout());
JPanel pan4 = new JPanel();

JPanel panInferior = new JPanel(new FlowLayout());
JButton ant = new JButton("<<");
JButton post = new JButton(">>");

JLabel buida = new JLabel("");

public void iniciar() throws SQLException, IOException,
NoSuchMethodException, SecurityException,
    IllegalAccessException, IllegalArgumentException,
InvocationTargetException {
    this.setLayout(new BorderLayout());
    this.setBounds(100, 100, 500, 675);

    this.getContentPane().add(panInicial, BorderLayout.NORTH);
    this.getContentPane().add(panCentral, BorderLayout.CENTER);
    panInicial.add(et_inicial);
    panCentral.add(pan1);
    panCentral.add(pan2);
    panCentral.add(pan3);
    panCentral.setBorder(BorderFactory.createEtchedBorder());

    pan1_1.setBorder(BorderFactory.createEmptyBorder(2,2,2,2));
    pan1.add(pan1_1);
    pan1.add(pan1_2);
    //pan1_1.add(pan1_1_1);
    //pan1_1.add(pan1_1_2);
    //pan1_1.add(pan1_1_3);
    et_nif.setHorizontalAlignment(SwingConstants.RIGHT);
    pan1_1.add(et_nif);
    pan1_1.add(nif);
    et_nom.setHorizontalAlignment(SwingConstants.RIGHT);
    pan1_1.add(et_nom);
    pan1_1.add(nom);
    major.setHorizontalAlignment(SwingConstants.RIGHT);
    pan1_1.add(major);
    pan1_1.add(et_major);
    pan1_1.add(buida);
    pan1.setBorder(BorderFactory.createEtchedBorder());
    pan1_2.add(foto);

    pan2.add(pan2_1, BorderLayout.NORTH);
    pan2.add(pan2_2, BorderLayout.CENTER);
    pan2_1.add(et_adr);
    pan2_1.add(adreca);
    pan2_2_1.add(et_correus);
    pan2_2_1.add(et_telefons);
    pan2_2_2.add(correus);
    pan2_2_2.add(telefons);
    pan2_2.add(pan2_2_1, BorderLayout.NORTH);
    pan2_2.add(pan2_2_2, BorderLayout.CENTER);
    pan2.setBorder(BorderFactory.createEtchedBorder());
    pan2_1.setBorder(BorderFactory.createLineBorder(Color.GRAY));
    //pan2_2.setBorder(new MatteBorder(2, 2, 2, 2, Color.BLACK));
    pan2_2_1.setBorder(BorderFactory.createLineBorder(Color.GRAY));
    pan2_2_1.setBackground( Color.GRAY );
    pan2_2_2.setBorder(BorderFactory.createLineBorder(Color.GRAY));
    pan2_2_2.setBackground( Color.GRAY );
    et_correus.setOpaque(true);
    et_telefons.setOpaque(true);
    correus.setOpaque(true);
    telefons.setOpaque(true);

    pan3.add(curric);
    pan3.setBorder(BorderFactory.createEtchedBorder());

    this.getContentPane().add(panInferior, BorderLayout.SOUTH);
    panInferior.add(ant);
    panInferior.add(post);

    String url = "jdbc:pgsql://89.36.214.106:5432/rxx";

    Connection con = DriverManager.getConnection(url, "rxx", "rxx");

    ResultSet rs = con.createStatement().executeQuery("SELECT * FROM
PERSONA4 ORDER BY 1");

    while (rs.next())
        llista.add(new Persona(rs));

```

```

        System.out.println(llista.size());
        mostraPersona(indActual);

        rs.close();
        con.close();
        this.setVisible(true);
        ant.addActionListener(this);
        post.addActionListener(this);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private void mostraPersona(int i) {
        Persona p = llista.get(i);

        nif.setText(p.getNif());
        nom.setText(p.getNom());
        major.setSelected(p.isMajorEdat());

        if (p.getFoto() != null) {
            foto.setIcon(new ImageIcon(p.getFoto()));
            //foto = new JLabel(new ImageIcon(p.getFoto()));
            foto.setBorder(BorderFactory.createEtchedBorder());
            panl_2.setVisible(true);
        } else
            panl_2.setVisible(false);

        curric.setText(p.getCurriculum());

        if (p.getAdreca() != null) {
            adreca.setText(p.getAdreca().getCarrer() + "-->" +
                p.getAdreca().getCodiPostal() + "-->"
                + p.getAdreca().getPoblacio());
        } else
            adreca.setText("");

        correus.setText("");
        if (p.getCorreusE() != null) {
            for (String c : p.getCorreusE())
                correus.append(c + "\n");
        }

        telefons.setText("");
        if (p.getTelefons() != null) {
            for (Telefon t : p.getTelefons()) {
                telefons.append(t.getNumero() + " (");
                if (t.isMobil())
                    telefons.append("mòbil)\n");
                else
                    telefons.append("fixe)\n");
            }
        }
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == ant) {
            if (indActual > 0)
                indActual--;
        }
        if (e.getSource() == post) {
            if (indActual < llista.size() - 1)
                indActual++;
        }
        mostraPersona(indActual);
    }
}

```

I aquest és el programa principal.

```

import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.sql.SQLException;

public class Proval0 {

    public static void main(String[] args) throws SQLException, IOException,
        NoSuchMethodException, SecurityException, IllegalAccessException,
        IllegalArgumentException, InvocationTargetException {
        Proval0_Pantalla finestra = new Proval0_Pantalla();
        finestra.iniciar();
    }
}

```


Aquest seria el seu aspecte:

Visualització de les persones

Nif


55555555e

Nom

Eva

☐

Major d'edat



Adreça

null-->null-->null

Correus

Telèfons

CURRÍCULUM DE EVA

=====

Molt treballadora i competent.

Ha estudiat:

- Primària

- ESO

- ...

<<

>>

2.4 - Altres aportacions

Les Bases de Dades Objecte-Relacionals ens poden oferir més coses que ens ajuden a definir les Bases de Dades més aproximades als objectes. Encara que s'escapen dels objectius del present curs, els comentem a continuació per veure el seu potencial.

Els veurem en SQL, i podem executar totes aquestes sentències des del PgAdmin III. No intentarem, però, utilitzar aquestes aportacions des de Java. Només les veurem per a poder mostrar el seu potencial.

2.4.1 - Polimorfisme i sobrecàrrega d'operadors

Polimorfisme

PostgreSQL permet el **polimorfisme de sobrecàrrega**, és a dir, construir una funció o procediment més d'una vegada amb diferents paràmetres. PostgreSQL sabrà quin s'ha d'executar pels paràmetres.

El curs passat vam crear funcions amb PL/pgSQL, el llenguatge de programació de PostgreSQL. Anem a veure uns exemples en els quals s'observa que podem crear 2 funcions amb el mateix nom però diferents paràmetres (en quantitat o en tipus dels paràmetres), és a dir **polimorfisme de sobrecàrrega**.

En aquest primer exemple la diferència entre les 2 funcions que es diuen igual és sobre el número de paràmetres.

```
CREATE OR REPLACE FUNCTION
MAX(a numeric,b numeric)
RETURNS numeric as '
BEGIN
    IF (a >= b)
        THEN RETURN a;
        ELSE RETURN b;
    END IF;
END; ' LANGUAGE plpgsql
```

```
CREATE OR REPLACE FUNCTION MAX(a
numeric,b numeric,c numeric)
RETURNS numeric as '
BEGIN
    RETURN MAX(MAX(a,b),c);
END; ' LANGUAGE plpgsql
```

Aquest segon exemple és sobre el tipus de dades dels paràmetres. Ens definim una funció anomenada **enganxa** sobre dos paràmetres de tipus text, que el que fa és senzillament concatenar. Però a continuació ens definim una altra funció **enganxa** sobre un text i un enter, que el que fa és repetir el text tantes vegades com diu l'enter.

```
CREATE OR REPLACE FUNCTION
ENGANXA(a text, b text)
RETURNS text AS '
BEGIN
    RETURN a || b;
END; ' LANGUAGE plpgsql
```

```
CREATE OR REPLACE FUNCTION
ENGANXA(a text, n integer)
RETURNS text AS '
DECLARE
    r text := '';
BEGIN
    FOR i IN 1..n LOOP
        r := r || a;
    END LOOP;
    RETURN r;
END; ' LANGUAGE plpgsql
```

Ens funcionarà tant **SELECT ENGANXA('Hola','Adéu)** com també **ENGANXA('Hola',4)**, fent coses molt diferents. Ara bé, el que no funcionarà és **SELECT ENGANXA(4,'Hola')**, ja que no tenim definida la funció **ENGANXA(integer,text)**.

Sobrecàrrega d'operadors

Nosaltres podem definir operadors. Per a definir un nou operador especificarem el símbol que utilitzarem, el o els operands i la funció que l'implementa.

La sintaxi és:

```
CREATE OPERATOR name (
    PROCEDURE = funcname
    [, LEFTARG = lefttype ] [, RIGHTARG = righttype ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    ...
)
```

Hi ha més opcions, que per a la utilitat d'aquest curs obviarem.

- En el nom de l'operador posarem un o més d'un caràcters de la següent llista:

+ - * / < > = ~ ! @ # % ^ & | ` ?

Hi ha algunes limitacions, que podem veure en la documentació.

- Sempre hem de posar la funció que implementa l'operador.

- Si l'operador és d'un únic operand, l'haurém d'especificar (el de la dreta o de l'esquerra, el que preferim). Si és de dos operands s'haurén d'especificar els dos.
- **COMMUTATOR** indica un altre operador que funciona igual canviant l'ordre dels paràmetres (en numèrics el commutador de < és >)
- **NEGATOR** indica un altre operador equivalent a negar aquest (en numèrics el negador de < és >=)

Per exemple anem a crear l'operador **MÀXIM (/|)** que calcula el màxim entre dos números. Utilitzarem la funció **MAX** que ja tenim creada.

```
CREATE OPERATOR /| (  
    PROCEDURE = MAX, LEFTARG = numeric, RIGHTARG = numeric);
```

Podem comprovar el seu funcionament amb la sentència:

```
SELECT 23 /| 15;
```

Anem a veure ara la sobrecàrrega d'operadors. Evidenlment estan definit els operador + (suma) i * (multiplicació) per a vaolrs numèrics. Però no ho estan per a valors de text.

- Anem a sobrecarregar l'operador suma (+) per a operands de text, de manera que es faça una concatenació (funció **ENGANXA(text,text)**)
- Anem a sobrecarregar l'operador producte (*) per a operand de text i enter, de manera que es repetesca el text tantes vegades com indica l'enter, és a dir, utilitzarem la funcio **ENGANXA(text,integer)**.

```
CREATE OPERATOR + (  
    PROCEDURE = ENGANXA, LEFTARG =  
    text, RIGHTARG = text);
```

```
CREATE OPERATOR * (  
    PROCEDURE = ENGANXA, LEFTARG =  
    text, RIGHTARG = integer);
```

2.4.2 - Herència

És com en l'herència d'objectes. Permet heretar les propietats d'un tipus o taula en un altre tipus a una altra taula. El cas del tipus ja el coneixem de Java. El cas de la taula pot ser menys familiar.

Quan una taula hereta d'una altra, contindrà tots els camps d'aquesta, a més de poder posar nous camps.

Fins i tot podem fer que les files que introduïm en les taules que han heretat, puguin aparèixer en la taula "principal".

Si tenim una taula creada:

```
create table t1 (  
    c1 int2 primary key,  
    c2 text);
```

Podem crear una altra que herete les característiques d'aquesta.

```
create table t2 (  
    c3 text) INHERITS (t1);
```

La taula t2 tindrà, a banda de l'atribut definit, en tindrà 2 més, c1 i c2, heretats de t1.

Açò ho podem comprovar fàcilment si obrim ara el PgAdmin i mirem les definicions de les taules acabades de crear.

Fins i tot podem comprovar que l'atribut c1 és no nul en t2.

<pre>CREATE TABLE t1 (c1 smallint NOT NULL, c2 text, CONSTRAINT t1_pkey PRIMARY KEY (c1)) WITH (OIDS=FALSE); ALTER TABLE t1 OWNER TO geo;</pre>	<pre>CREATE TABLE t2 (-- Heretat from table t1: c1 smallint NOT NULL, -- Heretat from table t1: c2 text, c3 text) INHERITS (t1) WITH (OIDS=FALSE); ALTER TABLE t2 OWNER TO geo;</pre>
---	---

També ho podem comprovar si intentem inserir en t2:

```
insert into t2  
values (1, 'Hola', 'Adéu');
```

Lamentablement no conserva la clau primària, i si la volem també en t2 haurem de modificar la taula per definir-la. Ho podem comprovar amb aquesta sentència:

```
insert into t2  
values (1, 'Hola2', 'Adéu2');
```

No dóna cap problema, perquè no hi ha clau primària. Observeu com fins i tot **PgAdmin** si anem a veure les dades (**Vista de Dades**) avisa que no hi ha clau principal i per tant no podrà editar-la, ja que no tindrà manera d'identificar la fila (distint seria si l'haguérem definida amb **OIDS**, **object identifier**, que posa una columna amb un número que identifica la fila).

Ara fixem-nos en una altra cosa. Si ara volem veure el contingut de t1 (on de moment no havíem introduït res), veurem que ja té dues files: (1,'Hola') i (1,'Hola2'). Bé realment aquestes dues files no estan introduïdes en t1, sinó que quan PostgreSQL trau el contingut d'una taula, trau també, per defecte, les seues descendents. La prova que realment no estan introduïdes en t1 és que no ha donat problemes la clau primària (que en t1 sí que està definida). I fins i tot una

prova més concloent: anem a inserir en **t1**.

```
insert into t1
values (1,'Hola3');
```

No dóna cap problema, però si que el donaria si intentàrem:

```
insert into t1
values (1,'Hola4');
```

Si per alguna raó volem traure únicament aquelles files que realment pertanyen a **t1** posaríem l'atribut **ONLY** davant del nom de la taula

```
select * from only t1;
```

Una característica molt interessant és que els possibles canvis en l'estructura de la taula inicial es veuran reflectits en les taules descendents.

Així, si afegim una columna o la suprimim, o la canviem de tipus, o fins i tot si la posem no nula, el mateix passarà amb les taules descendents.

```
alter table t1 add column c4 numeric(5,2);
```

```
CREATE TABLE t2
(
-- Heretat from table t1:  c1 smallint NOT NULL,
-- Heretat from table t1:  c2 text,
  c3 text
-- Heretat from table t1:  c4 numeric(5,2)
)
INHERITS (t1)
WITH (
  OIDS=FALSE
);
ALTER TABLE t2 OWNER TO geo;
```

Nota

En PgAdmin haurem de "refrescar" les taules per poder comprovar que s'han afegit les modificacions.

No s'hereten, en canvi, les claus principals, ni restriccions UNIQUE, ni claus externes, encara que aquestes últimes en versions anteriors sí que s'heretaven; ja veurem en versions posteriors...

Fins i tot podem esborrar la taula i fer que s'esborren les descendents. Ho haurem de fer amb l'opció CASCADE:

```
drop table t1 cascade;
```

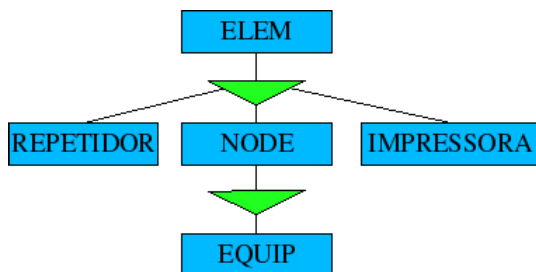
Sembla que la utilitat més gran de l'herència, és quan partim d'una especialització des del Model Entitat-Relació. Ens basarem en un exemple que no siga massa complicat, però en el qual es puguin veure unes quantes especialitzacions, i a dos nivells.

Nota

Els següents exemples els farem sobre l'usuari i Base de Dades propi de cadascun **rx**.

Suposem que volem documentar els tallers d'informàtica d'un Institut. De tots els elements voldrem el nom, la descripció i el lloc (el taller) on estan. Els elements són de tres tipus: nodes (ordinadors, servidors d'impressió, routers), que tenen una adreça IP, repetidors (switch i hubs), i impressores. Lògicament ens guardarem coses diferents de cadascun. Dels nodes, en principi, només ens interessin els ordinadors.

Aquest seria l'esquema Entitat-Relació:



Aquest esquema ens donaria les següents taules:

```

elem (nom,descripcio,taller)
repetidor (nom)
node (nom,adreca)
impressora (nom,tipus,velocitat)
equip (nom,cpu,ram,hd)
  
```

Anem a crear les taules

```

create table elem (
  nom varchar(15) constraint cp_ele primary key,
  descripcio varchar(50),
  taller varchar(15));
  
```

```

create table repetidor () inherits (elem);
  
```

```

create table node (
  adreca inet) inherits (elem);
  
```

```

create table impressora (
  tipus varchar(15),
  velocitat varchar(15)) inherits (elem);
  
```

```

create table equip (
  cpu varchar(25),
  ram numeric(4,0),
  hd varchar(25)) inherits (node);
  
```

Ara procedirem a crear les claus primàries de **repetidor**, **node**, **impressora** i **equip**

```

alter table repetidor
  add constraint cp_rep primary key(nom);
  
```

```

alter table node
  add constraint cp_nod primary key(nom);
  
```

```

alter table impressora
  add constraint cp_imp primary key(nom);
  
```

```

alter table equip
  add constraint cp_equ primary key(nom);
  
```

Introduïm algunes dades de mostra. Normalment introduïrem en les taules descendents.

```

insert into repetidor
  values ('S-TI1','Switch del Taller 1','TI1');
  
```

```

insert into equip values
  
```

```
( 'TI1-01', 'Ord 1 Taller 1', 'TI1', '192.168.201.101', 'AMD Athlon 64 X2', 2048, '250 Gb' ),
( 'TI1-02', 'Ord 2 Taller 1', 'TI1', '192.168.201.102', 'AMD Athlon 64 X2', 2048, '250 Gb' ),
( 'TI1-03', 'Ord 3 Taller 1', 'TI1', '192.168.201.103', 'AMD Athlon 64 X2', 2048, '250 Gb' ),
( 'TI4-01', 'Ord 1 Taller 4', 'TI4', '192.168.204.101', 'Celeron', 1024, '80 Gb' );
```

Si ara llistem **ELEM** veurem tots els element introduïts.

Query - xarxa sobre xarxa@172.16.1.2:5432 *

Fitxer Edita Consulta Favorits Macros Vista Ajuda

Editor SQL Constructor de Consultes Gràfic

SELECT * FROM elem;

Subfinestra de sortida

Sortida de dades Explain Missatges Historial

	nom character varying(15)	descripcio character varying(50)	taller character varying(15)	
1	S-TI1	Switch del Taller 1	TI1	
2	TI1-01	Ord 1 Taller 1	TI1	
3	TI1-02	Ord 2 Taller 1	TI1	
4	TI1-03	Ord 3 Taller 1	TI1	
5	TI4-01	Ord 1 Taller 4	TI4	

OK. Unix Ln 1 Col 20 Ch 20 5 registres 17 ms

Les taules poden heretar els atributs de més d'una taula.

Per a provar-lo anem a modificar el disseny anterior, suposant que tenim dues classes d'impressora, les que es connecten a la xarxa, i les que es connecten a un equip. Però la característica que volem de les que es connecten en xarxa és la de **node**. Per tant, per una banda volem les característiques de **node**, i per una altra les de **impressora**. Anem a veure com quedaria aquesta impressora en xarxa, deixant l'altra per després.

```
create table imp_x
( ) inherits (impressora, node);
```

En el moment de crear la taula ens ha avisat que s'heretaven columnes amb el mateix nom, però no ha hagut cap problema. Només els col·loca una vegada, encara que en comprovar en l'estructura que ens mostra **PgAdmin** sembla que les columnes estiguen més d'una vegada:


```
CREATE TABLE imp_x
(
  -- Heretat from table impressora: nom character varying(15) NOT NULL,
  -- Heretat from table node: nom character varying(15) NOT NULL,
  -- Heretat from table node: descripcio character varying(50),
  -- Heretat from table impressora: descripcio character varying(50),
  -- Heretat from table node: taller character varying(15),
  -- Heretat from table impressora: taller character varying(15),
  -- Heretat from table node: tipus character varying(15),
  -- Heretat from table impressora: tipus character varying(15),
  -- Heretat from table impressora: velocitat character varying(15),
  -- Heretat from table node: velocitat character varying(15),
  -- Heretat from table node: adreca inet,
  -- Heretat from table impressora: adreca inet
)
INHERITS (impressora, node)
WITH (
  OIDS=FALSE
);
ALTER TABLE imp_x OWNER TO xarxa;
```

Ho veurem clar en inserir una fila i visualitzar-la.

Però si ens fixem, no tindrem clau primària. Haurem de modificar, tal i com vam fer amb les altres taules:

```
alter table imp_x
  add constraint cp_imp_x primary key(nom);
```

Introduïm alguna dada per veure el resultat.

```
insert into imp_x
  values ('IMP-01', 'Imp 1 Taller 1', 'TI1', 'Laser', '20 ppm', '192.168.201.201');
```

Podem comprovar que si llistem la taula **IMPRESSORA**, apareixerà, i si llistem la taula **NODE**, també eixirà acompanyada de tots els equips.

Left Screenshot: Query - xarxa sobre xarxa@172.16.1.2:5432 *

Editor SQL: `SELECT * FROM impressora;`

Subfinestra de sortida:

	nom character varying(15)	descripcio character varying(50)	taller character varying(15)	tipus character varying(15)	velocitat character varying(15)
1	IMP-01	Imp 1 Taller 1	TI1	Laser	20 ppm

OK. Unix |Ln 1 Col 25 Ch 25 | 1 registres | 18 ms

Right Screenshot: Query - xarxa sobre xarxa@172.16.1.2:5432 *

Editor SQL: `SELECT * FROM node;`

Subfinestra de sortida:

	nom character varying(15)	descripcio character varying(50)	taller character varying(15)	adreca inet
1	TI1-01	Ord 1 Taller 1	TI1	192.168.201.101
2	TI1-02	Ord 2 Taller 1	TI1	192.168.201.102
3	TI1-03	Ord 3 Taller 1	TI1	192.168.201.103
4	TI4-01	Ord 1 Taller 4	TI4	192.168.204.101
5	IMP-01	Imp 1 Taller 1	TI1	192.168.201.201

OK. Unix |Ln 1 Col 19 Ch 19 | 5 registres | 15 ms

Per últim si llistem la pròpia taula **IMP_X**, veurem com els camps no estan repetits.

Query - xarxa sobre xarxa@172.16.1.2:5432 *

Fitxer Edita Consulta Favorits Macros Vista Ajuda

Editor SQL Constructor de Consultes Gràfic

SELECT * FROM imp_x

Subfinestra de sortida

Sortida de dades Explain Missatges Historial

	nom character	descripcio character varying(50)	taller character	tipus character varying(15)	velocitat character varying(15)	adreca inet
1	IMP-01	Imp 1 Taller 1	TI1	Laser	20 ppm	192.168.201.201

OK. Unix Ln 1 Col 20 Ch 20 1 registres. 16 ms

3 - Bases de Dades Orientades a Objectes

L'última alternativa que veurem per a guardar els objectes de Java serà fer servir directament una **Base de Dades Orientada a Objectes**. En principi semblaria que es tracta de la millor solució ja que no hi ha cap tipus de desfasament, i es podran guardar directament els objectes. Encara així, aquesta modalitat de bases de dades presenta certes peculiaritats que cal tenir molt en compte.

Actualment trobem al mercat dos tipus de Bases de Dades Orientades a Objecte.

- Les que compleixen l'estàndard proposat per l'**ODMG** (Object Database Management Group) a finals de l'any 2000
- Les anomenades bases de dades de tipus **NoSQL**, iniciatives tecnològiques posteriors. En realitat el terme **NoSQL** (Not Only SQL) es refereix a totes aquelles que no segueixen el Model Relacional, i s'inclourien BD Orientades a Objectes, BD XML, i també altres tipus com les documentals, les BD clau-valor, i algun altre tipus.

L'ODMG es va desfer l'any 2001, després de culminar l'estàndard anomenat ODMG 3.0. Encara que ja ha passat molt de temps, es tracta d'un estàndard molt **infrautilitzat**. Els llenguatges que especifica (ODL i OQL), encara que tenen una potència molt gran i permeten una adaptació real a la sintaxi habitualment utilitzada per la majoria de llenguatges Orientats a Objectes, presenta una **enorme influència del paradigma relacional**. Aquesta és segurament la causa de que hi hagen pocs productes actuals que segueixen aquest model. I és una pena, perquè a banda de guardar còmodament els objectes, el OQL ens permetria buscar la informació de forma fàcil i potent.

En els últims anys han aparegut algunes iniciatives noves que proposen un canvi de plantejament en la gestió dels objectes: les Bases de Dades **NoSQL**. Es tracta d'un plantejament trencador que vol aprofitar-se de l'estructura interna pròpia dels objectes per organitzar la persistència a mode de punters o referències internes, de manera que una vegada guardats els objectes, no es perdi la potencialitat que suposa aquesta organització.

És una iniciativa molt jove que sembla cridada a ser la següent generació en els Sistemes Gestors de Bases de Dades perquè dona una resposta molt adequada a l'hora de representar escenaris complexos en els quals les bases de dades clàssiques (relacionals) han fracassat. Ens referim a escenaris amb components gràfics o documentals, o a situacions amb informació extremadament complexa que cal analitzar (*business intelligence*, per exemple).

DB4O (*Database for Objects*) és una de les iniciatives actuals amb llicència GPL més actives, que implementa una base de dades 100% Orientada a Objectes i basada en el concepte **NoSQL**. Es tracta d'una base de dades lleugera amb una **versió integrada en el propi llenguatge Java (Embedded)**. Això fa que la seua instal·lació siga nul·la, ja que només cal incorporar a la nostra aplicació les biblioteques que podem trobar a la pàgina de l'empresa **Versant**, per disposar d'una base de dades operativa. I pel seu petit volum i extrema eficiència, s'utilitza també molt en aplicacions mòbils i serveis web.

Existeix també un versió distribuïda per instal·lar en un servidor, però la versió integrada serà suficient per agafar les nocions bàsiques que ens cal explicar.

L'ús de **DB4O** no precisa crear cap model de dades específic per configurar l'estructura que suporti les dades en SGBDOO. De fet, utilitza la informació interna dels objectes per construir de forma dinàmica aquesta estructura. L'extracció d'aquesta informació és possible gràcies a les característiques reflexives dels llenguatges de programació als quals dona suport (Java i C#).

DB4O no guarda els objectes basant-se en els seus valors, sinó en les seues referències en memòria. Dos objectes ubicats en llocs diferents de la memòria RAM, encara que representen el mateix, si es guarden en DB4O es tractaran com a objectes diferents, encara que hi haja una coincidència amb tots i cadascun dels seus valors. És per això que es recomana utilitzar només una única instància per cada objecte que l'aplicació necessite. Si una instància ja està guardada, caldrà recuperar-la de la Base de Dades, ja que les successives recuperacions d'un mateix objecte retornen sempre la mateixa referència en memòria. Si la instància no estiguera guardada, caldria instanciar-la una única vegada fent servir qualsevol forma d'instanciació suportada pel llenguatge (utilitzant, per exemple, una sentència *new* seguida del constructor).

A banda de la instanciació d'objectes i del concepte d'identitat utilitzat, DB4O afegeix un nou concepte que caldrà tenir en compte a l'hora d'implementar aplicacions que la utilitzen. Es tracta del concepte d'activació. Per tal d'evitar un

temps de procés i consum de memòria excessius a l'hora de treballar amb estructures de dades molt complexes i ramificades, DB4O ofereix diverses formes de recuperar un objecte sense haver d'instanciar tots i cadascun dels seus objectes interns. DB4O ofereix la possibilitat d'indicar la profunditat a la qual es vol fer la recuperació. Per defecte, la profunditat utilitzada és 5, però podem canviar-ho a voluntat. Una vegada instanciat un objecte podem canviar la seua profunditat d'activació de forma dinàmica en qualsevol moment, de manera que disposem d'un mecanisme per accedir a tots els objectes que l'aplicació requereix en el moment que siga necessari sense haver de malgastar recursos.

3.1 - Instal·lació de DB4O

DB4O va ser desenvolupat per la companyia **db4objects, Inc.** En 2008 va ser adquirida per l'empresa **Versant**, que va continuar desenvolupant-lo com a Open Source.

Em 2012, Versant va ser adquirida per la companyia **Actian**, que ja no ha continuat desenvolupant-lo com a Open Source, optant per un altre producte de pagament.

A pesar de l'anterior, instal·larem DB4O, encara que siga una versió creada ja fa un temps.

La instal·lació de DB4O és molt senzilla, ja que només és descomprimir, per a després utilitzar les llibreries en el projecte. Podem obtenir DB4O des de la pàgina que Actian manté per a les versions anteriors:

<http://supportservices.actian.com/versant/default.html>

A banda de descomprimir el fitxer, també muntarem una perspectiva per poder "veure" el contingut de les Bases de Dades d'objectes. El següent vídeo il·lustra tot el procés:

3.2 - Funcionalitat bàsica

Treballarem sobre un exemple anterior, el dels empleats, però incorporant més dades, com els telèfons, els correus electrònics, etc, per veure que podem guardar una classe un poc més complicada en la BD Orientada a Objectes.

En un projecte nou, anomenat **Tema6_2**, anem a incorporar les llibreries de **DB4O**. Aquestes es troben en el subdirectori **lib** del lloc on havíem col·locat DB4O.

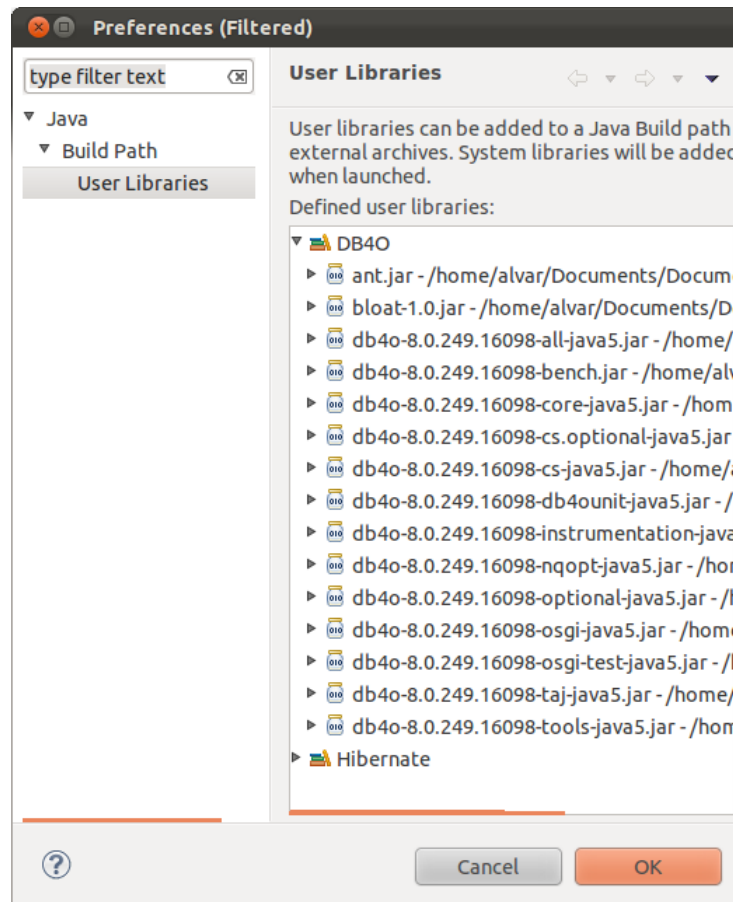
Segurament serà suficient amb incorporar el jar

dg4o-8.0.249.16098-core-java5.jar

Si no tinguérem prou amb aquest, un candidat molt bo seria

dg4o-8.0.249.16098-all-java5.jar

I si en alguna ocasió tenim problemes, doncs sempre podem incorporar tots els jar en una llibreria d'usuari (per exemple amb el nom **DB4O**) com ja vam fer amb Hibernate. La imatge mostra el procés, però recordeu que segurament no farà falta:



Per organitzar-lo millor creem un paquet anomenat **classesEmpleat**, que ens servirà per a fer tots els exemples. Ens crearem la classe **Empleat**, i les classes **Adreca** i **Telefon** que utilitzarà aquella. Construïm les classes amb **constructor** i mètodes **get** i **set**.

```
public class Adreca {
    private String carrer;
    private String codipostal;
    private String poblacio;

    public Adreca() {
    }
}
```

```

public Adreca(String c, String cp, String p) {
    setCarrer(c);
    setCodipostal(cp);
    setPoblacio(p);
}

public String getCarrer() {
    return carrer;
}

public void setCarrer(String carrer) {
    this.carrer = carrer;
}

public String getCodipostal() {
    return codipostal;
}

public void setCodipostal(String codipostal) {
    this.codipostal = codipostal;
}

public String getPoblacio() {
    return poblacio;
}

public void setPoblacio(String poblacio) {
    this.poblacio = poblacio;
}
}

```

```

public class Telefon {
    private boolean mobil;
    private String numero;

    public Telefon() {
    }

    public Telefon(boolean m, String num) {
        setMobil(m);
        setNumero(num);
    }

    public boolean isMobil() {
        return mobil;
    }

    public void setMobil(boolean mobil) {
        this.mobil = mobil;
    }

    public String getNumero() {
        return numero;
    }

    public void setNumero(String numero) {
        this.numero = numero;
    }
}

```

Ara ja va **Empleat**:

```

public class Empleat {
    private String nif;
    private String nom;
    private int departament;
    private int edat=0;
    private double sou=0.0;
    private byte[] foto;
    private char[] curriculum;
    private Adreca adreca;
    private String[] correus_e;
    private Telefon[] telefons;

    public Empleat(){
    }

    public Empleat(String nif){
        this.setNif(nif);
    }
}

```

```

    }

    public Empleat(String nif,String nom,int dep,int edat,double sou,byte[]
    foto,char[] curr,Adreca adr,String[] corr,Telefon[] tels){
        this.setNif(nif);
        this.setNom(nom);
        this.setDepartament(dep);
        this.setEdat(edat);
        this.setSou(sou);
        this.setFoto(foto);
        this.setCurriculum(curr);
        this.setAdreca(adr);
        this.setCorreus_e(corr);
        this.setTelefons(tels);
    }

    ... //mètodes get i set
}

```

No hem posat tots els mètodes get i set, per a que no quede tan llarg. Es poden generar molt fàcilment amb: **Source**
--> Generate Getters and Setters

Com comentàvem anteriorment, hi ha una versió servidor, però que nosaltres ens conformarem amb la versió integrada (*embedded*). En el cas de la versió servidor utilitzariem la classe **Db4o** per a fer la connexió. Com que nosaltres farem servir la versió integrada, utilitzarem la classe **Db4oEmbedded**.

Des de l'aplicació indicarem el nom del fitxer on es guardaran les dades cridant el mètode estàtic de Db4oEmbedded **openFile**. A partir d'aquest moment, es mantindrà oberta una transacció que continuarà activa fins que tanquem amb el mètode **close**.

Les proves que farem a continuació les podem posar en un paquet anomenat **Exemples**

Inserció

Per a guardar un objecte utilitzem el mètode **store(objecte)**

```

import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;

import classesEmpleat.Adreca;
import classesEmpleat.Empleat;
import classesEmpleat.Telefon;

public class Proval {
    public static void main(String[] args) {
        ObjectContainer bd = Db4oEmbedded.openFile("Empleats.db4o");

        Empleat e = new Empleat("11111111a", "Albert", 10, 45, 1000, null,
        null, null, null, null);

        // les dades més complicades les introduïm de forma especial
        e.setAdreca(new Adreca("C/ Major, 7", "12001", "Castelló"));
        String[] corr = { "alul1111111a@ieselcaminas.org" };
        e.setCorreus_e(corr);
        Telefon[] tels = { new Telefon(true, "666777888"), new Telefon(false,
        "964112233") };
        e.setTelefons(tels);

        bd.store(e);

        bd.close();
    }
}

```

Sí que ha guardat l'objecte. Mirem-lo des de la perspectiva OME amb un vídeo il·lustratiu:

En finalitzar d'observar les dades des de la perspectiva OME, és convenient **tancar la connexió**. Si no la tanquem, quan anem a executar el següent programa, ens donarà error, avisant que la Base de Dades està bloquejada (*com.db4o.ext.DatabaseFileLockedException*).

Per tant, haurem de tenir especial atenció a tancar la connexió a la Base de Dades. Podria passar que ens donara un error el programa, i la connexió s'haja quedat oberta. Segurament el més oportú serà intentar tancar el programa, o tancar Eclipse, i d'aquesta manera desbloquejarem la Base de Dades.

El mètode **commit** obliga a guardar les dades cap al contenidor i activa de nou una transacció per a les properes operacions, per tant és convenient anar utilitzant-lo després d'una sèrie d'actualitzacions.

Anem a posar algunes dades més, per a tenir un poc més de joc. Concretament seran dues empleades més.

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;

import classesEmpleat.Adreca;
import classesEmpleat.Empleat;
import classesEmpleat.Telefon;

public class Proval_1 {
    public static void main(String[] args) {
        ObjectContainer bd = Db4oEmbedded.openFile("Empleats.db4o");
        Empleat e = new
Empleat("22222222b", "Berta", 10, 35, 1700, null, null, null, null, null);
        Empleat f = new
```

```

Empleat("33333333c", "Clàudia", 20, 37, 1500, null, null, null, null, null);

//les dades més complicades les introduïm de forma especial
e.setAdreca(new Adreca("C/ Enmig, 7", "12001", "Castelló"));
String[] corr = {"alu22222222b@ieselcaminas.org", "berta@gmail.com"};
e.setCorreus_e(corr);
Telefon[] tels = {new Telefon(true, "666555444"), new
Telefon(false, "964223344")};
e.setTelefons(tels);

f.setAdreca(new Adreca("C/ de Dalt, 7", null, "Borriana"));
String[] corr2 = {"alu33333333c@ieselcaminas.org"};
f.setCorreus_e(corr2);

bd.store(e);
bd.store(f);

bd.close();
}
}

```

Consulta bàsica

En la següent pregunta, veurem les maneres de fer una consulta, però ara anem a veure la forma més senzilla, que és la realitzada a través del mètode anomenat **queryByExample**. Aquest mètode rep per paràmetre un objecte del tipus a cercar, que farà d'exemple o patró per trobar totes aquelles instàncies emmagatzemades coincidents amb les dades del patró. El patró que es passe per paràmetre no haurà de tenir totes les dades complimentades, sinó només aquelles de les quals se'n desitja la coincidència. Així, per exemple, si hi passem un objecte comercial amb un únic atribut complet (el *nif*), **queryByExample** retornarà totes aquelles instàncies que tinguin per *nif* el valor entrat. El retorn es fa en un objecte de tipus **ObjectSet**, una classe que implementa la interfície **List** de Java i també la interfície **Iterable**, de manera que siga possible recórrer el contingut usant els mètodes **next** i **hasNext**. També podem utilitzar el bucle **for** (el del **foreach**).

En el següent exemple es veu com una vegada obtingut l'objecte, es pot accedir molt fàcilment a tota la informació:

```

import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

import classesEmpleat.Empleat;

public class Prova2 {
    public static void main(String[] args) {
        ObjectContainer bd = Db4oEmbedded.openFile("Empleats.db4o");
        Empleat e = null;

        ObjectSet<Empleat> llista = bd.queryByExample(new
Empleat("11111111a"));

        if (llista.hasNext()) {
            e = llista.next();
            System.out.println("Nif: " + e.getNif() + ". Nom: " + e.getNom() +
". Població: " + e.getAdreca().getPoblacio());
            System.out.println("Primer correu: " + e.getCorreus_e()[0] + ".
Primer telèfon: " + e.getTelefons()[0].getNumero());
        }
        bd.close();
    }
}

```

Observeu com no hem utilitzat un bucle per a recórrer la llista, sinó un if. Això és perquè en aquest cas concret sabem a priori que en cas de trobar alguna instància, només serà una. Aquest seria el resultat:

```

Nif: 11111111a. Nom: Albert. Població: Castelló
Primer correu: alu11111111a@ieselcaminas.org. Primer telèfon: 666777888

```

Esborrat

Per a poder fer una actualització o esborrat d'algun objecte de la Base de Dades, aquest s'ha de correspondre amb

algun objecte del programa Java. Aquesta correspondència pot ser perquè un objecte nou l'hem guardat amb **store()** (i continua "viu"), o perquè l'hem llegit de la BD (millor dit, hem llegit una llista i després hem fet l'assignació a un objecte).

L'eliminació dels objectes s'aconsegueix amb el mètode **delete**. Per defecte, DB4O elimina només l'objecte que es passa com a paràmetre, però no els objectes que aquest puga contenir. Si un objecte conté un altre objecte, com succeeix amb les instàncies **Empleat** i **Adreca** i **Telefon**, això pot convertir-se en un gran problema, ja que són objectes que normalment no es manipularan per separat i, en cas que no s'esborren amb el seu propietari, continuaran indefinidament en la Base de Dades. Per evitar-lo hauríem de configurar per a que **esborre en cascada**.

Mirem un exemple en el qual esborrem un empleat. En el comentari teniu el moment en que encara no es corresponen, i per tant no es pot esborrar.

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

import classesEmpleat.Eempleat;

public class Prova3 {

    public static void main(String[] args) {

        ObjectContainer bd = Db4oEmbedded.openFile("Empleats.db4o");
        Empleat e = new Empleat("2222222b");

        // Si posàrem ací db.delete(e) no tindria efecte, perquè e no es
        // correspon amb cap instància de la BD

        ObjectSet<Empleat> llista = bd.queryByExample(e);
        if (llista.hasNext()) {
            e = llista.next();
            bd.delete(e);
        }
        bd.close();
    }
}
```

Com que no hem esborrat en cascada, si després mirem des de la perspectiva OME, comprovarem que encara existeixen els objectes adreça i telèfon, i que ara no correspondran a cap empleat. Hauríem d'aprofitar el moment per a esborrar des de la perspectiva OME les instàncies de Adreca i Telefon que corresponien a l'empleat que hem esborrat, per deixar-lo consistent. En les següents imatges es mostra aquest fet:

classesEmpleat.Eempleat ☒										
Row Id	nif ▼	nom	departament	edat	sou	foto	curriculum	adreca	correus_e	telefons
1	11111111a	Albert	10	45	1000.0	null	null	(G) classesEm	1 items	2 items
2	33333333c	Clàudia	20	37	1500.0	null	null	(G) classesEm	1 items	null

Ja no existeix l'objecte Empleat corresponent a Berta, però:

classesEmpleat.Eempleat				classesEmpleat.Adreca ☒		classesEmpleat.Telefon ☒	
Row Id	carrer	codipostal	població				
1	C/ Major, 7	12001	Castelló				
2	C/ Enmig, 7	12001	Castelló				
3	C/ de Dalt, 7	null	Borriana				

classesEmpleat.Eempleat		classesEmpleat.Adreca		classesEmpleat.Telefon ☒	
Row Id	mobil				numero
1	true				666777888
2	false				964112233
3	true				666555444
4	false				964223344

encara existeix la seua adreça (C/Enmig, 7 de Castelló) i els seua telàfons (666555444 i 964223344)

Per a poder esborrar en cascada, en el moment d'obrir el fitxer contenidor haurem d'especificar-lo posant-li una configuració com veurem a continuació. No és possible modificar la configuració de forma dinàmica. A més, malauradament, la configuració no es guarda amb el fitxer contenidor, sinó que cada vegada que obrim, haurem d'especificar-li la configuració desitjada. En aquesta configuració li direm que la classe **Empleat** esborra en cascada, és a dir, que quan esborrem un objecte, els objectes "subordinats" (de les classes **Adreca** i **Telefon**) també s'esborraran.

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.config.EmbeddedConfiguration;
```

```
import classesEmpleat.Eempleat;

public class Prova4 {

    public static void main(String[] args) {
        EmbeddedConfiguration conf = Db4oEmbedded.newConfiguration();
        conf.common().objectClass(Eempleat.class).cascadeOnDelete(true);

        ObjectContainer bd = Db4oEmbedded.openFile(conf, "Empleats.db4o");

        Eempleat e = new Eempleat("33333333c");
        ObjectSet<Eempleat> llista = bd.queryByExample(e);
        if (llista.hasNext()) {
            e = llista.next();
            bd.delete(e);
        }
        bd.close();
    }
}
```

En les següents imatges es mostra com ara sí que ha esborrat en cascada:

classesEmpleat.Eempleat ☒										
Row Id	nif	nom	departament	edat	sou ▲	foto	curriculum	adreca	correus_e	telefonos
1	11111111a	Albert	10	45	1000.0	null	null	(G) classesEm	1 items	2 items

Hem esborrat a Clàudia

classesEmpleat.Eempleat				classesEmpleat.Adreca ☒		classesEmpleat.Telefon ☒	
Row Id	carrer	codipostal	poblacio				
1	C/ Major, 7	12001	Castelló				
2	C/ Enmig, 7	12001	Castelló				

classesEmpleat.Eempleat		classesEmpleat.Adreca	classesEmpleat.Telefon ☒
Row Id	mobili	numero	
1	true	666777888	
2	false	964112233	
3	true	666555444	
4	false	964223344	

I també ha desaparegut la seua adreça (C/ de Dalt de Borriana). Com que no tenia telèfons, continuen els mateixos d'abans

Modificació

Per a modificar un objecte de la Base de Dades primer haurem de tenir un objecte de Java que es corresponga amb ell (igual que en l'esborrat). Després de modificar-lo, només l'haurem de guardar amb **store()**. Hem de parar atenció a que si el que volem modificar és d'una subclasse, haurem de **modificar en cascada**, sinó no tindrà efecte. Ho farem indicant **cascadeOnUpdate(true)** a la configuració amb què obrirem el fitxer :

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.config.EmbeddedConfiguration;

import classesEmpleat.Adreca;
import classesEmpleat.Eempleat;

public class Prova5 {

    public static void main(String[] args) {
        EmbeddedConfiguration conf = Db4oEmbedded.newConfiguration();
        conf.common().objectClass(Eempleat.class).cascadeOnUpdate(true);

        ObjectContainer bd = Db4oEmbedded.openFile(conf, "Empleats.db4o");

        Eempleat e = new Eempleat("11111111a");
        ObjectSet<Eempleat> llista = bd.queryByExample(e);
        if (llista.hasNext()) {
            e = llista.next();
            e.setSou(e.getSou() + 200);
            Adreca adr = e.getAdreca();
            adr.setCarrer("Pl. Rei en Jaume, 15");
            adr.setCodipostal("12002");
            e.setAdreca(adr);
            bd.store(e);
        }
        bd.close();
    }
}
```

En la imatge es veu que en fer l'actualització en cascada sí que s'han guardat els canvis, i la primera adreça (que és la corresponent a Albert) s'ha modificat.

classesEmpleat.Adreca ☒			
Row Id	carrer ▲	codipostal	població
1	Pl. Rei en Jaume, 15	12002	Castelló
2	C/ Enmig, 7	12001	Castelló

La restricció que hem comentat abans de que hem de tenir un objecte de Java que es corresponga amb ell (que ocupa els casos d'esborrat i modificació), l'hem de tenir molt present. I hem d'anar amb compte, perquè quan es tanca la BD es perd tota correspondència.

El següent exemple és idèntic a l'anterior, però es tanca i es torna a obrir la BD després d'haver assignat a **e** l'objecte, i abans de guardar-lo. En principi el que voldríem és modificar les dades de l'empleat existent, però en realitat hem introduït un nou empleat (amb el mateix nif, nom, ...), i per tant molt perillós perquè estem duplicant la informació. Observeu que, com que només es vol modificar el sou, no cal actualitzar en cascada.

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

import classesEmpleat.Eempleat;

public class Prova6 {

    public static void main(String[] args) {
        ObjectContainer bd = Db4oEmbedded.openFile("Empleats.db4o");

        Eempleat e = new Eempleat("11111111a");
        ObjectSet<Eempleat> llista = bd.queryByExample(e);
        if (llista.hasNext()) {
            e = llista.next();
            e.setSou(e.getSou() + 200);

            bd.close(); // Tanquem i tornem a obrir la BD, per veure que hem
                        // perdut la correspondència de e amb l'objecte de la
BD
            bd = Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),
"Empleats.db4o");

            bd.store(e);
            bd.close();
        }
    }
}
```

Ara el contingut de la Base de Dade és aquest:

classesEmpleat.Eempleat ☒										
Row Id	nif	nom	departament	edat	sou	foto	curriculum	adreca	correus_e	telefonos
1	11111111a	Albert	10	45	1400.0	null	null	(G) classesEm	1 items	2 items
2	11111111a	Albert	10	45	1600.0	null	null	(G) classesEm	1 items	2 items

On es veu que hem creat un nou objecte, en compte de modificar el que ja existia. I el mateix amb l'adreça i els telèfons

classesEmpleat.Eempleat				classesEmpleat.Adreca ☒		classesEmpleat.Telefon ☒	
Row Id	carrer	codipostal	població	mobilitat	numero		
1	Pl. Rei en Jaume, 15	12002	Castelló	true	666777888		
2	C/ Enmig, 7	12001	Castelló	false	964112233		
3	Pl. Rei en Jaume, 15	12002	Castelló	true	666555444		
4				false	964223344		
5				true	666777888		
6				false	964112233		

En cas que tanquem la BD i volem modificar o esborrar un objecte haurem de tornar a connectar amb ell.

I en el cas de la inserció, abans d'inserir, podríem comprovar que no existeix (per exemple que no existeix cap empleat amb aquest nif).

3.3 - Consultes

DB4O disposa de tres formes de realitzar consultes. Totes elles són de tipus NoSQL.

Nota

Per a poder tenir un poc més de joc, torneu a inserir les 2 empleades de la modificació de la classe **Prova1**, i així en tindrem un total de 3. Si hàviem fet tots els exemples anteriors, potser siga millor esborrar **Empleats.db4o** i tornar a executar **Prova1** i **Prova1_1** per a crear-les de nou.

Mètode Query By Example

La primera forma ja s'ha comentat, és la que s'anomena consulta basada en un exemple o "query by example". Consisteix, com ja hem vist, en trobar totes les instàncies guardades que coincideixen amb els valors no nuls i diferents de zero (en cas que siguin numèrics) d'un patró o exemple passat per paràmetre.

Si, per exemple, volem traure els empleats del departament 10 que són de Castelló, n'hi hauria prou amb crear el patró següent:

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

import classesEmpleat.Adreca;
import classesEmpleat.Empleat;

public class Prova1 {

    public static void main(String[] args) {
        ObjectContainer bd =
        Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), "Empleats.db4o");

        Empleat f = new Empleat();
        f.setDepartament(10);
        f.setAdreca(new Adreca(null, null, "Castelló"));

        ObjectSet<Empleat> llista = bd.queryByExample(f);
        for (Empleat e : llista) {
            System.out.println("Nif: " + e.getNif() + ". Nom: " + e.getNom() +
            ". Departament: " + e.getDepartament()
            + ". Població: " + e.getAdreca().getPoblació());
        }
        bd.close();
    }
}
```

cosa que donarà com a resultat el següent, que es pot comprovar que són del departament 10 i de Castelló:

```
Nif: 11111111a. Nom: Albert. Departament: 10. Població: Castelló
Nif: 22222222b. Nom: Berta. Departament: 10. Població: Castelló
```

Seguint aquest raonament, per obtenir tots els empleats de l'aplicació caldrà passar un patró empleat sense valors (**bd.queryByExample(new Empleat())**), i si el que desitgem és obtenir tots els objectes emmagatzemats a la base de dades, el que haurem de passar com a paràmetre és un valor null (**bd.queryByExample(null)**).

Com podeu veure, resulta un sistema molt simple. Ara bé, també té moltes limitacions en consultes més complexes, i fins i tot poden resultar impossibles. Posem alguns exemples en els quals no funciona aquest tipus de consulta:

- És impossible trobar tots els empleats que no tinguin algun camp assignat encara (és a dir, null) a causa del mecanisme utilitzat: només s'avaluen els camps no nuls.
- Tampoc podríem trobar aquells empleats que cobren més de 1300€ . En aquest tipus de consulta només podem buscar igualtats.
- Com es basa en la coincidència, no podem fer consultes que puguin agafar un de dos o més valors determinats. Per exemple, agafar els empleats que són de Castelló o Borriana.

Mètode *Native Queries*

DB4O disposa d'un sistema molt més potent anomenat **Native Queries**. És fàcil deduir que es tracta d'un sistema vinculat directament al mateix llenguatge de programació. De fet, es tracta de construir un procediment en el qual s'avaluen els objectes i es decideix quins objectes compleixen la condició i quins no.

Per a fer la consulta haurem de crear una classe que implemente una interfície anomenada **Predicate**. Aquesta interfície consta d'un únic mètode declarat anomenat **match**. La classe nostra que implementarà Predicate haurà de sobreescrivre el mètode match(), i en aquest mètode podrem posar una sèrie de sentències Java i dir si cada objecte de la Base de Dades compleix o no la condició tornant respectivament true o false.

En el següent exemple creem una classe anomenada **EmpleatsPerPoblacio** (que implementa **Predicate**), a la qual se li pot passar en el constructor un vector de cadenes de caràcters amb els noms de les poblacions de les quals volem els empleats. En la implementació del mètode **match** tornarem cert si l'empleat és d'alguna de les poblacions, i fals en cas contrari. Com que utilitzem el mateix llenguatge de programació, la potència és molt elevada i la corba d'aprenentatge d'aquesta tècnica esdevé pràcticament nul·la.

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.Predicate;

import classesEmpleat.Empleat;

public class Prova12 {

    public static void main(String[] args) {
        ObjectContainer bd =
        Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), "Empleats.db4o");
        String[] pobl = { "Castelló", "Borriana" };

        ObjectSet<Empleat> llista = bd.query(new EmpleatsPerPoblacio(pobl));

        for (Empleat e : llista) {
            System.out.println(e.getNom() + " (" + e.getAdreca().getPoblacio()
+ ")");
        }
        bd.close();
    }

    public static class EmpleatsPerPoblacio extends Predicate<Empleat> {
        String[] poblacions;

        public EmpleatsPerPoblacio(String[] poblacions) {
            this.poblacions = poblacions;
        }

        @Override
        public boolean match(Empleat emp) {
            boolean ret = false;
            for (int i = 0; !ret && i < poblacions.length; i++) {
                if
                (emp.getAdreca().getPoblacio().equalsIgnoreCase(poblacions[i]))
                    ret = true;
            }
            return ret;
        }
    }
}
```

Observeu que una vegada definida la classe, podem fer-la servir en una **Query** per realitzar una consulta específica. En l'exemple, s'obtenen tots els empleats que són de Castelló o de Borriana. En variar la llista de poblacions obtindrem uns objectes empleat o uns altres. En el mètode match, que és qui diu si un element Empleat compleix la condició, es comprova si la població de l'empleat (que està dins d'adreça, i per tant s'accedeix amb **emp.getAdreca().getPoblacio()**) és igual a alguna de les de l'array de poblacions. Per a fer-lo més general, es fa amb el mètode **equalsIgnoreCase()**, que no distingeix entre majúscules i minúscules. En quant es troba una població coincident, s'eixirà del bucle, ja que s'ha posat també com a condició del bucle **!ret** (que no s'ha trobat), a banda del comptador del bucle.

Com que es tracta d'una interfície amb un únic mètode a implementar, no caldrà que implementem sempre noves classes per a cada consulta diferent, sinó que podem fer servir **classes anidades anònimes** (anonymous nested class), per a fer-lo molt més curt. Recordeu que les classes anidades poden treballar directament amb tots els atributs (tinguen l'àmbit que tinguen) de la classe que les continga i que les classes anònimes es defineixen a l'interior d'un mètode qualsevol.

Mirem-ho en un altre exemple, en el qual es buscaran els empleats que tinguen el sou entre dos valors determinats. Construïm la classe **Predicate** en el mateix lloc on s'utilitza, en el **query()**, i no abulta molt perquè només té el mètode **match()**. En el mètode **match()** és on es comprova la condició:

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.Predicate;

import classesEmpleat.Eempleat;

public class Prova13 {

    public static void main(String[] args) {
        ObjectContainer bd =
        Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(), "Empleats.db4o");
        final int max = 1500;
        final int min = 1000;
        ObjectSet<Empleat> llista = bd.query(new Predicate<Empleat>() {
            @Override
            public boolean match(Empleat emp) {
                if (emp.getSou() <= max && emp.getSou() >= min)
                    return true;
                else
                    return false;
            }
        });

        for (Empleat e : llista) {
            System.out.println(e.getNom() + " (" + e.getSou() + ")");
        }
        bd.close();
    }
}
```

Mètode SODA

Existeix encara una altra forma de definir consultes. DB4O l'anomena **SODA** (*Simple Object Database Access*), i es pot considerar com la forma d'accedir a l'estructura interna de la base de dades a baix nivell per tal de seleccionar els nodes de dades que complesquen uns determinats requisits i que acabaran determinant el resultat de la consulta. De fet, segons indiquen els autors, és la forma de consulta més ràpida de les tres.

La idea fonamental de SODA és construir les consultes com un recorregut d'una xarxa de nodes enllaçats. Els nodes de la consulta s'estructuren de forma semblant a les classes emmagatzemades a la base de dades, de manera que el camí seguit en avaluar la consulta, node a node, es repeteix en les instàncies emmagatzemades, la qual cosa permet accedir als valors per avaluar de forma ràpida.

El camí s'especifica utilitzant el mètode **descend()** per mitjà del qual seleccionem la branca de l'estructura de classes que vulguem fer referència. Per exemple, si ens trobem en el node de la classe **Empleat** i volguérem fer referència al nom de la població que en l'estructura de classes es troba a **empleat.getAdreca().getPoblacio()**, hauríem de fer

```
node.descend("adreca").descend("poblacio")
```

El resultat de la sentència anterior és un node focalitzat a l'atribut població continguda a l'adreça de l'empleat.

Cada node pot estar afectat per una restricció, per una ordenació i/o per una operació amb una altre node. Les restriccions permeten seleccionar o desestimar les instàncies que es vagen comprovant. Les ordenacions, com és natural, forcen l'ordre de les instàncies seleccionades d'acord amb els valors de l'atribut representat pel node afectat.

Finalment, les operacions marquen quin serà el següent node a avaluar, el qual actuarà també com a filtre dels objectes de la selecció.

Les restriccions es veuran afectades per una o més relacions que permetran modificar la comparació i sentenciar en favor o en contra de la selecció d'una instància. Per defecte, la relació avaluada és la d'igualtat. Per exemple, si partim d'un node que representa el NIF d'un empleat, podem definir la relació d'igualtat següent:

```
node.constrain("11111111a")
```

Mirem com quedaria el programa que selecciona únicament l'empleat amb el nif anterior:

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.Query;

public class Prova14 {

    public static void main(String[] args) {
        ObjectContainer bd =
            Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),
                "Empleats.db4o");
        Query q = bd.query(); //node arrel.
        q.constrain(Empleat.class); //limitem la cerca als Empleats (pot
            haver guardada més d'una classe)

        Query node = q.descend("nif"); //arribem a l'altura del nif, que és on
            posem la restricció
        node.constrain("11111111a");

        ObjectSet<Empleat> llista = q.execute();

        for(Empleat e: llista){
            System.out.println("Nif: " + e.getNif() + ". Nom: " + e.getNom() +
                " (" + e.getSou() + ")");
        }
        bd.close();
    }
}
```

Però si la relació ha de ser una comparació de tipus **major que**, **menor o igual que**, ... , caldrà especificar-les expressament. La manera serà especificant un mètode de la restricció. Les possibilitats seran:

- Major: **greater()**

Si suposem que partim d'un node focalitzat al **sou** d'un empleat i volem la condició que el sou siga major estrictament que 1300. S'indicaria d'aquesta manera:

```
node.constrain(1300).greater();
```

- Menor: **smaller()**

Si volem que el sou siga estrictament menor que 1500:

```
node.constrain(1500).smaller();
```

- Major o igual, menor o igual: **equal()** (després del greater o smaller)

Si ara volem que el sou siga menor o igual que 1500:

```
node.constrain(1300).smaller().equal();
```

- Que comence per: **startsWith(boolean)**

Si partim d'un node focalitzat al **nom** de l'empleat i volem els que comencen per **A**:

```
node.constrain("A").startsWith(true);
```

Si en el paràmetre booleà posem true, haurà de coincidir exactament el principi. Si posem false, no distingirà entre majúscules i minúscules.

- Per a unir restriccions: **or(restricció) and(restricció)**. Per a negar **not()**

Per exemple, si partim d'un node focalitzat al **nom** de l'empleat, podem seleccionar tots els que comencen per **A** o per **B**, fent:

```
Constraint constr1 = node.constrain("A").startsWith();
```

```
Constraint constr2 = node.constrain("B").startsWith();
constr1.or(constr2);
```

- Si posem més d'una restricció (més d'un constrain), s'hauran de complir totes, i per tant actua com un **and**

A banda de les restriccions, si volem ordenar de forma ascendent o descendent, ho indicarem amb els mètode **orderAscending()** o **orderDescending()** del node pel mig del qual volem ordenar .

Mirem un parell d'exemples per veure com es posa tot en joc. Anem a construir la sentència que permeti seleccionar tots els empleats amb un sou que oscil·le entre un rang de valors definits (estrictament major que 1000, i menor o igual que 1500, per exemple) ordenats de forma descendent per sou:

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.Query;

public class Prova15 {

    public static void main(String[] args) {

        ObjectContainer bd =
            Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),
                "Empleats.db4o");

        Query q = bd.query();           //node arrel.
        q.constrain(Empleat.class);      //limitem la cerca als Empleats (pot
        haver guardada més d'una classe)
        Query node = q.descend("sou");   //arribem a l'altura del sou, que és on
        posem restriccions
        node.constrain(1000).greater().and(node.constrain(1500).smaller().equal());
        node.orderDescending();

        ObjectSet<Empleat> llista = q.execute();

        for(Empleat e: llista){
            System.out.println(e.getNom() + " (" + e.getSou() + ")");
        }
        bd.close();
    }
}
```

I ara els empleats del departament 10 que són de Castelló. Podem utilitzar el mateix objecte node per anar afegint restriccions, però haurem de cuidar de localitzar-lo al lloc oportú:

```
import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.Query;

public class Prova16 {

    public static void main(String[] args) {

        ObjectContainer bd =
            Db4oEmbedded.openFile(Db4oEmbedded.newConfiguration(),
                "Empleats.db4o");

        Query q = bd.query();
        q.constrain(Empleat.class);

        Query node = q.descend("departament");
        node.constrain(10);

        node = q.descend("adreca").descend("poblacio");
        node.constrain("Castelló");

        ObjectSet<Empleat> llista = q.execute();

        for(Empleat e: llista){
            System.out.println("Nom: " + e.getNom() + ". Població: " +
                e.getAdreca().getPoblacio() + ". Departament: " +
                e.getDepartament());
        }
        bd.close();
    }
}
```

Tot i que cal reconèixer la potència del sistema, de moment encara no és capaç de tenir tota l'expressivitat d'un llenguatge com OQL. No disposa de funcions d'agregació (SUM, AVG, MAX, MIN, ...), ni es poden expressar relacions entre instàncies emmagatzemades. De moment és l'aplicació que haurà de fer-se responsable que això siga possible. Es tracta, però, d'una tècnica molt jove, que de ben segur anirà evolucionant. Caldrà estar atents per veure fins on arriba.

Exercicis

**Exercici 6.1 (BDOR)**

Des de la perspectiva **Database Development** d'Eclipse, o des de **PgAdmin** si el tens instal·lat, entrant com a usuari **rx** en la Base de Dades **rx**:

- Crea un tipus de dades anomenat **coordenades**, amb dos camps: **lat** (numèric) i **lon** (numèric).
- Crea un tipus de dades anomenat **punt**, amb tres camps: **num_p** (enter), **nom_p** (text) i **coord** (de tipus coordenades)
- Crea un **taula** anomenada **ruta** amb la següent estructura: **num_r** (enter, clau principal), **nom_r** (text), **desn** (enter), **desn_acum** (enter), **llista_punts** (array de **punt**)

**Exercici 6.2 (BDOR)**

Crea un nou paquet anomenat **Exercicis** en el projecte **Tema6_1**. Inclou en el projecte les llibreries de JDBC per a PostgreSQL i per a SQLite, si no ho estan encara.

- Copia el paquet **util.bd** al projecte. Aquest paquet el vam fer en l'exercici **Ex 4.4** i inclou les classes **Coordenades.java**, **PuntGeo.java** i **Ruta.java**. També inclou la classe **GestionarRutesBD.java**, que ens permetia gestionar la BD **Rutes.sqlite**
- Copia també la BD **Rutes.sqlite**, creada en exercicis anteriors, però que l'última actualització és del mateix exercici.

A partir d'ací comença realment l'exercici, que consistirà en passar les dades des de **Rutes.sqlite** fins la Base de Dades de **PostgreSQL rx**, creada en l'exercici anterior, i posteriorment visualitzar-les, però de forma còmoda

- Crea la classe **PassarRutaSqlitePostgresql.java**, amb **main()** (és a dir, executable).
 - Heu d'agafar totes les rutes de **Rutes.sqlite** i deixar-les en un **ArrayList** de **Ruta** amb el mètode ja creat de **GestionarRutesBD.java** anomenat **llistat()**.
 - Utilitzeu el mètode **cadenaRuta(ruta)** que us passarà el professor que torna una cadena de text amb totes les dades de la ruta, que es pot utilitzar en un **INSERT** (per exemple el podeu col·locar en la mateixa classe **PassarRutaSqlitePostgresql**). Us recomane vivament que primer tragueu per pantalla la sentència **INSERT** abans d'executar-la. Observeu que **cadenaRuta()** no torna el número de la ruta.
 - Inserir totes les rutes en la BD de **PostgreSQL rx**.
- Crea la classe **VisRutaPostgreSQL**, amb **main()** (és a dir, executable) que visualitzi les rutes amb el següent format. Observa que sí que agafem element a element de la llista de punts, però després no intentem separar el contingut ni dels elements de tipus **punt**, ni molt menys dels de tipus **coordenades**.

```
Ruta número 1: Pujada a Penyalosa
Punts:
(1,"Sant Joan","(40.251036,-0.354223)")
(2,"Encreuament","(40.25156,-0.352507)")
(3,"Barranc de la Pregunta","(40.247318,-0.351713)")
(4,"El Corralico","(40.231708,-0.348859)")
(5,"Penyalosa","(40.222632,-0.350339)")
```

**Exercici 6.3 (BDOO)**

Crea un nou paquet anomenat **Exercicis** en el projecte **Tema6_2**. Incorpora en el projecte el driver per a **DB4O**.

- Incorpora en ell el paquet **util.bd**, ja utilitzades en l'exercici anterior. En ell han d'estar les classes **Coordenades.java**, **PuntGeo.java** i **Ruta.java**.
- Assegura't que la classe **Ruta.java** tenen els mètodes **getLlistaDePunts()** i **setLlistaDePunts()**, (o similar) que tornen o tenen com a paràmetre un **ArrayList** de **PuntGeo**. En cas que no els tingues, crea'ls de la forma ràpida, i si encara tens dubtes, senzillament agafa el paquet **util.bd** que et proporciona el professor
- En el paquet ha d'estar també la classe **GestionarRutesBD.java**, que ens permetia gestionar la BD **Rutes.sqlite**.
- Copia també la BD **Rutes.sqlite**.

A partir d'ací comença realment l'exercici, que consistirà en passar les dades des de **Rutes.sqlite** fins la Base de Dades de **DB4O Rutes.db4o**.

- Crea la classe **PassarRutaSqliteDB4O.java**, amb **main()** (és a dir, executable).
 - Has d'agafar totes les rutes de **Rutes.sqlite** i deixar-les en un **ArrayList** de **Ruta** amb el mètode ja creat de **GestionarRutesBD.java** anomenat **llistat()**.
 - Insereix totes les rutes en la BD **Rutes.db4o**.
 - Tanca la connexió.
- Crea la classe **VisRutaDB4O.java**, amb **main()** (és a dir, executable), que ha de connectar a la Base de Dades **Rutes.db4o**, ha de llegir totes les rutes (ves amb compte, perquè només has de llegir rutes; s'han guardat més objectes: punts i coordenades) i ha de traure per pantalla el nom de la ruta i el número de punts.

```
Pujada a Penyagolosa: 5 punts
La Magdalena: 8 punts
Pelegrins de Les Useres: 6 punts
```



Exercici 6.4 (BDOO)

En el mateix projecte i paquet anem a fer una aplicació amb interfície pràctica. De moment tindrà aquest aspecte:

Nom punt	Latitud	Longitud
Sant Joan	40.251036	-0.354223
Encreuament	40.25156	-0.352507
Barranc de la Pegunta	40.247318	-0.351713
El Corralico	40.231708	-0.348859
Penyagolosa	40.222632	-0.350339

en el qual, a banda de les etiquetes (**JLabel**) i quadres de text (**JTextField**) tenim una taula (**JTable**) on col·locarem tots els punts de la ruta (nom, latitud i longitud). Tots els controls són no editables, per a no poder introduir cap informació.

Haureu de tenir una classe **Main** com la següent:

```
public class Rutes_DB4O {

    public static void main(String[] args) {
        Rutes_DB4O_pantalla finestra = new Rutes_DB4O_pantalla();
        finestra.iniciar();
    }
}
```

I la classe **Pantalla**, que és realment és on es fa la feina. Us passe l'esquelet, amb la inclusió de tots els controls. Per cert, els controls estan deshabilitats per a no poder modificar les dades. Teniu ja construït un mètode anomenat **plenarTaula()** que accepta un paràmetre de tipus **ArrayList<PuntGeo>**, i s'encarrega de plenar correctament la taula

```
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;

import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextField;
import javax.swing.JTable;

import com.db4o.Db4oEmbedded;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

import util.bd.PuntGeo;
import util.bd.Ruta;

public class Rutes_DB4O_pantalla extends JFrame implements
ActionListener {
```

```

private static final long serialVersionUID = 1L;
ArrayList<Ruta> llista = new ArrayList<Ruta>();
int num_actual;

JLabel etiqueta = new JLabel("");
JLabel etNom = new JLabel("Ruta:");
JTextField qNom = new JTextField(15);
JLabel etDesn = new JLabel("Desnivell:");
JTextField qDesn = new JTextField(5);
JLabel etDesnAcum = new JLabel("Desnivell acumulat:");
JTextField qDesnAcum = new JTextField(5);
JLabel etPunts = new JLabel("Punts:");
JTable punts = new JTable(1,3);
JButton primer = new JButton(" << ");
JButton anterior = new JButton(" < ");
JButton seguent = new JButton(" > ");
JButton ultim = new JButton(" >> ");

// en iniciar posem un contenidor per als elements anteriors
public void iniciar() {
    getContentPane().setLayout(new GridLayout(0,1));
    JPanel p_prin = new JPanel();
    p_prin.setLayout(new BorderLayout(p_prin, BorderLayout.Y_AXIS));
    // contenidor per als elements
    JPanel panell1 = new JPanel(new GridLayout(0,2));
    panell1.add(etNom);
    qNom.setEditable(false);
    panell1.add(qNom);
    panell1.add(etDesn);
    qDesn.setEditable(false);
    panell1.add(qDesn);
    panell1.add(etDesnAcum);
    qDesnAcum.setEditable(false);
    panell1.add(qDesnAcum);
    panell1.add(etPunts);

    JPanel panell2 = new JPanel(new GridLayout(0,1));
    punts.setEnabled(false);
    JScrollPane scroll = new JScrollPane(punts);
    panell2.add(scroll, null);

    JPanel panell5 = new JPanel(new FlowLayout());
    panell5.add(primer);
    panell5.add(anterior);
    panell5.add(seguent);
    panell5.add(ultim);

    getContentPane().add(p_prin);
    p_prin.add(panell1);
    p_prin.add(panell2);
    p_prin.add(panell5);

    setVisible(true);
    pack();
    primer.addActionListener(this);
    anterior.addActionListener(this);
    seguent.addActionListener(this);
    ultim.addActionListener(this);

    inicialitzar();
    VisRuta();
}

private void plenarTaula(ArrayList<PuntGeo> ll_punts) {
    Object[][] ll = new Object[ll_punts.size()][3];
    for (int i=0; i<ll_punts.size(); i++) {
        ll[i][0]=ll_punts.get(i).getNom();
        ll[i][1]=ll_punts.get(i).getLatitud();
        ll[i][2]=ll_punts.get(i).getLongitud();
    }
    String[] caps = {"Nom punt", "Latitud", "Longitud"};
    punts.setModel(new
javax.swing.table.DefaultTableModel(ll, caps));
}

@Override
public void actionPerformed(ActionEvent e) {

}

private void inicialitzar() {
}

```

```
private void VisRuta() {
}
}
```

Vosaltres haureu de fer els 3 mètodes del final per a que apareguen les dades.



Exercici 6.5 (BDOO)

Modificar la classe anterior per a incorporar també la **distància total de la ruta**. Per a poder calcular-la ens ajudarem de d'una funció ja creada (que us passarà el professor) que calcula la distància en quilòmetres entre dos punts, donant les coordenades (latitud i longitud) dels dos punts: **Dist(lat1,long1,lat2,long2)**. Podeu incorporar-la a la classe on esteu fent l'exercici (**Rutes_DB4O_pantalla**).

Aquest seria un exemple:

Ruta:	Pujada a Penyagolosa	
Desnivell:	530	
Desnivell acumulat:	606	
Distància:	3.41 km	
Punts:		
Nom punt	Latitud	Longitud
Sant Joan	40.251036	-0.354223
Encreuament	40.25156	-0.352507
Barranc de la Pregunta	40.247318	-0.351713
El Corralico	40.231708	-0.348859
Penyagolosa	40.222632	-0.350339

<<
<
>
>>



Exercici 6.6 (ampliació - voluntari)

Modifica l'aplicació anterior per a que es puguin modificar, esborrar i inserir les rutes.

Ruta:	Catí - Sant Pere de Castellfort	
Desnivell:	611	
Desnivell acumulat:	1286	
Distància:	19.67 km	
Punts:		
Nom punt	Latitud	Longitud
Catí	40.47095	0.0222777778
L'Avella	40.501991667	7.75E-4
Salvassoria	40.5124638889	-0.0293944444
La Llacua	40.5009138889	-0.0552722222
Venta de la Ratlla	40.5103111111	-0.1438055556
Sant Pere	40.4978472222	-0.173075

<< < > >>

Editar Eliminar Nova Ruta

Tancar

- S'haurien de posar més botons: **Editar**, **Eliminar** i **Nova Ruta**.
- Estaria bé que en entrar a qualsevol de les opcions anteriors es desactivaren els botons de navegació, que desaparegueren els d'Editar, Eliminar i Nova Ruta, i que aparegueren els d'**Acceptar** i **Cancel·lar**.
- En tots els casos, si es cancel·la no es fa cap acció, però s'ha de tornar a l'estat anterior (primera imatge)
- **EDITAR:**
 - S'han d'"activar" els controls per a poder modificar les dades.
 - En cas d'acceptar s'ha de fer la modificació a partir del contingut de tots els controls (no cal detectar quins s'han modificat)
 - En cas de cancel·lar, no es fa la modificació, i senzillament s'ha de tornar a visualitzar la ruta actual (com no s'ha fet cap canvi, apareixeran les dades anteriors)
 - Per a afegir nous punts, es podria posar un botó per a afegir una nova línia al JTable, i un altre per a llevar una línia

Ruta:	Catí - Sant Pere de Castellfort	
Desnivell:	611	
Desnivell acumulat:	1286	
Distància:	19.67 km	
Punts:		
Nom punt	Latitud	Longitud
Catí	40.47095	0.0222777778
L'Avella	40.501991667	7.75E-4
Salvassoria	40.5124638889	-0.0293944444
La Llacua	40.5009138889	-0.0552722222
Venta de la Ratlla	40.5103111111	-0.1438055556
Sant Pere	40.4978472222	-0.173075

- **ELIMINAR:**

- Si s'accepta, s'haurà d'esborrar la ruta, sinó tornar a visualitzar-la

Ruta:	Catí - Sant Pere de Castellfort	
Desnivell:	611	
Desnivell acumulat:	1286	
Distància:	19.67 km	
Punts:		
Nom punt	Latitud	Longitud
Catí	40.47095	0.0222777778
L'Avella	40.501991667	7.75E-4
Salvassoria	40.5124638889	-0.0293944444
La Llacua	40.5009138889	-0.0552722222
Venta de la Ratlla	40.5103111111	-0.1438055556
Sant Pere	40.4978472222	-0.173075

Below the table is a large empty rectangular area. At the bottom of the window are buttons: '<<', '<', '>', '>>', 'Acceptar', 'Cancel·lar', and 'Tancar'.

- **INSERIR:**

- Haurà de mostrar tots els camps en blanc, i evidentment activats, per a poder introduir dades.
- En cas d'acceptar s'ha d'introduir la nova ruta.
- En cas de cancel·lar, estaria bé tornar a la que s'estava mostrant abans d'apretar el botó de nova ruta.
- Per a introduir nous punts, es podria posar un botó per a afegir una nova línia al JTable, i un altre per a llevar una línia

Nom punt	Latitud	Longitud
----------	---------	----------

Nota

EL JTable de vegades és engorros. Si s'està editant una casella, la informació no s'ha introduït encara, fins que no s'aprete enter, tab o amb el ratolí no s'aprete a algun altre lloc. Per a acabar la introducció de la informació que s'està editant, es podria executar el següent (per exemple quan s'ha apretat **Acceptar**):

```
if (punts.isEditing())  
    punts.getCellEditor().stopCellEditing();
```

on **punts** seria al JTable.

Llicenciat sota la [Llicència Creative Commons Reconeixement NoComercial CompartirIgual 2.5](#)