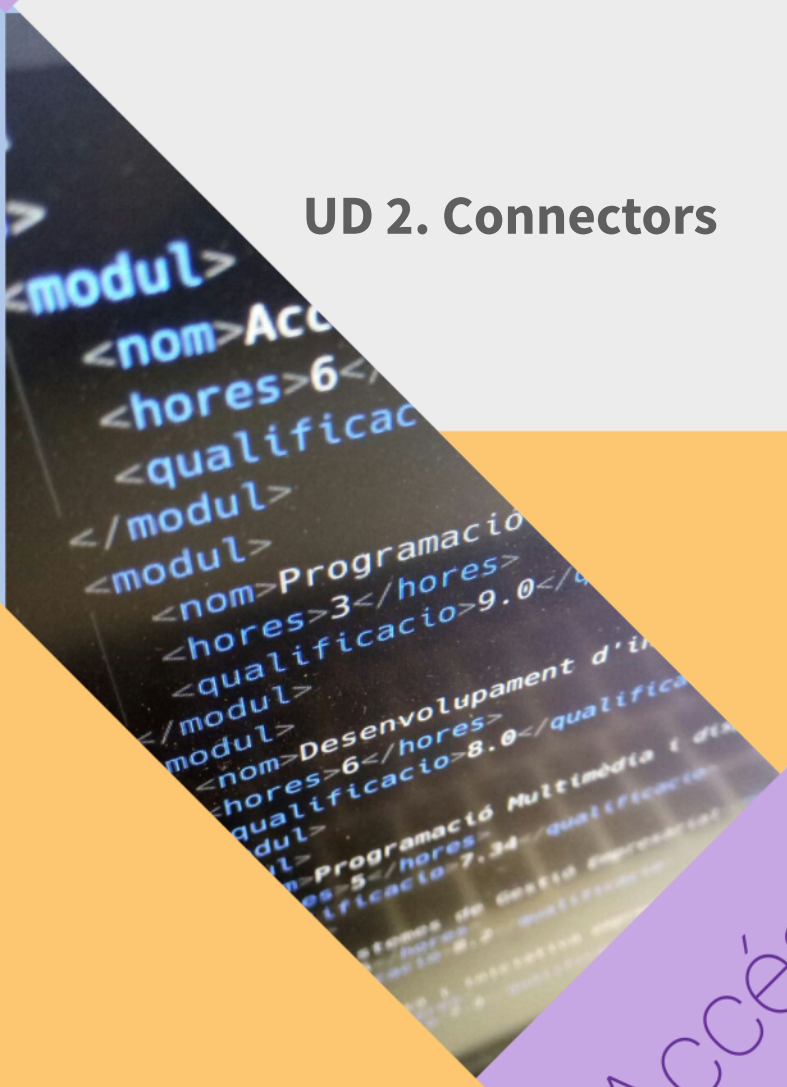
Curs 2019-2020



UD 2. Connectors

Accés a Dades

IES Jaume II El Just
Tavernes de la Valldigna

Continguts

1. El desfasament Objecte Relacional

Els Sistemes de Gestió de Bases de Dades Relacionals es basen en el Model Relacional, on la informació s'emmagatzema en diverses taules relacionades entre elles. Es tracta d'una tecnologia senzilla i eficient, que ha perdurat amb els anys, i que segueix sent el model emprat per la majoria de bases de dades i de SGBDs actuals.

Tot i l'èxit, el model té algunes limitacions, com la representació d'informació poc estructurada o complexa.

1.1. Què entenem per desfasament objecte-relacional?

Els models conceptuals ens ajuden a modelar una realitat complexa, i es basen en un procés d'abstracció de la realitat. Cada model té una forma de plasmar aquesta realitat, però totes elles són més properes a la mentalitat humana que a la memòria d'un ordinador.

Quan modelem una base de dades fem ús del model conceptual Entitat-Relació, i posteriorment, fem un procés de pas a taules i normalització d'aquest model, per tindre un model relacional de dades.

En el cas de la programació orientada a objectes, intentem representar la realitat mitjançant objectes i les relacions entre ells. Es tracta d'un altre tipus de model conceptual, però que pretén representar la mateixa realitat que el relacional.

Així doncs, tenim dues aproximacions diferents per tal de representar la realitat d'un problema: el Model Relacional de la base de dades i el model Orientat a Objectes de les nostres aplicacions.

1.2. Representació de la informació amb el model relacional

El model relacional es basa en taules i la relació entre elles.

- Cada taula té tantes columnes com **atributs** volem representar, i tantes files com **registres** o elements d'eixe tipus continga.
- Les taules tenen una **clau principal**, que identifica cadascun dels registres, i pot estar formada per un o diversos atributs.
- La relació entre taules es representa mitjançant **claus externes**, que consisteixen a incloure en una taula la clau principal d'una altra taula, com a «referència» a aquesta. Quan s'elimina un registre d'una taula, la clau primària del qual està referenciada per una altra, cal assegurar-se que mantenim la integritat referencial de la base de dades. Aleshores, davant aquest esborrat d'una clau primària, podem:
  - **No permetre l'esborrat** (*NO ACTION*),

  - Realitzar l'**esborrat en cascada**, esborrant també tots els registres que feren referència a la clau primària del registre esborrat (*CASCADE*),
  - **Establir a nuls** (*SET NULL*), de manera que la clau externa que es referia a la clau primària de l'altra taula pren el valor de NUL.

- Els diferents camps de les taules poden tenir també certes restriccions associades, com poden ser:

  - Restricció de **valor no nul**, de manera que el camp no pot ser nul en cap cas,
  - Restricció d'**unicitat** sobre un o diversos camps, de manera que el valor ha de ser únic en tota la taula. Les claus primàries, posseeixen ambdues propietats: valor no nul i unicitat.
  - Restricció **de domini**, o el que és el mateix, poden tenir un conjunt de valors possibles predeterminat,

Exemple

Veiem un breu exemple de model relacional creat amb MySQL Workbench, on es mostren tres Entitats (Joc, Genere i Jugadors) i les seues relacions:

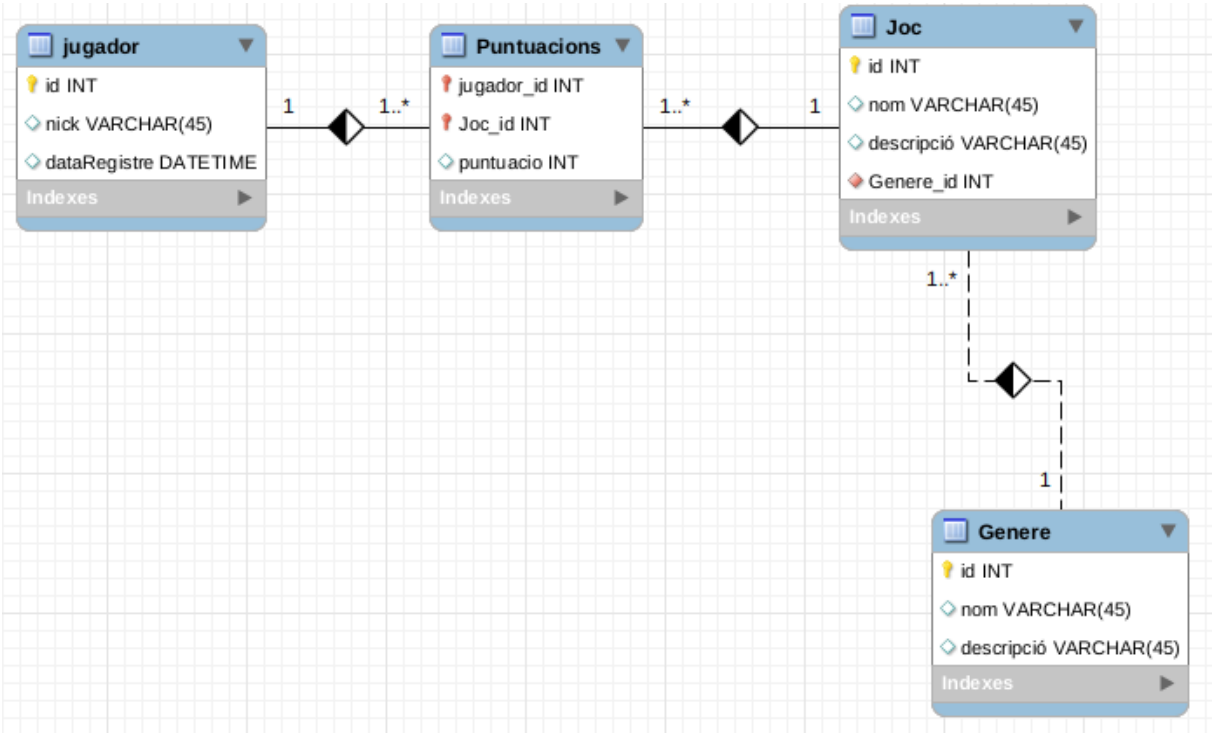


Figura 1: Entitat Relació de jocs

Com veiem, el que sería una relació molts a molts en el model conceptual entitat-relació, es converteix

en una nova taula en el model relacional (taula Puntuacions), i que relaciona un jugador amb la puntuació que té a un joc. Aquesta nova taula, tindrà una clau principal composta per dos camps: jugador_id i Joc_id, que al seu temps, seran claus externes a jugador i a joc, respectivament. Per altra banda, per indicar que un joc pertany a un gènere, aquest contindrà una clau externa (Genere_id) a la taula Genere.

Si passem el model a taules, quedaría una cosa així:

```
1
2  CREATE SCHEMA IF NOT EXISTS `BDJocs` DEFAULT CHARACTER SET utf8 ;
3  USE `BDJocs` ;
4
5  CREATE TABLE IF NOT EXISTS `BDJocs`.`jugador` (
6    `id` INT NOT NULL,
7    `nick` VARCHAR(45) NULL,
8    `dataRegistre` DATETIME NULL,
9    PRIMARY KEY (`id`)) ENGINE = InnoDB;
10
11 CREATE TABLE IF NOT EXISTS `BDJocs`.`Genere` (
12   `id` INT NOT NULL,
13   `nom` VARCHAR(45) NULL,
14   `descripció` VARCHAR(45) NULL,
15   PRIMARY KEY (`id`)) ENGINE = InnoDB;
16
17 CREATE TABLE IF NOT EXISTS `BDJocs`.`Joc` (
18   `id` INT NOT NULL,
19   `nom` VARCHAR(45) NULL,
20   `descripció` VARCHAR(45) NULL,
21   `Genere_id` INT NOT NULL,
22   PRIMARY KEY (`id`),
23   INDEX `fk_Joc_Genere1_idx` (`Genere_id` ASC),
24   CONSTRAINT `fk_Joc_Genere1`
25     FOREIGN KEY (`Genere_id`)
26     REFERENCES `BDJocs`.`Genere` (`id`)
27     ON DELETE NO ACTION
28     ON UPDATE NO ACTION) ENGINE = InnoDB;
29
30 CREATE TABLE IF NOT EXISTS `BDJocs`.`Puntuacions` (
31   `jugador_id` INT NOT NULL,
32   `Joc_id` INT NOT NULL,
33   `puntuacio` INT NULL,
34   PRIMARY KEY (`jugador_id`, `Joc_id`),
35   INDEX `fk_jugador_has_Joc_Joc1_idx` (`Joc_id` ASC),
36   INDEX `fk_jugador_has_Joc_jugador1_idx` (`jugador_id` ASC),
37   CONSTRAINT `fk_jugador_has_Joc_jugador1`
38     FOREIGN KEY (`jugador_id`)
39     REFERENCES `BDJocs`.`jugador` (`id`)
40     ON DELETE NO ACTION
41     ON UPDATE NO ACTION,
42   CONSTRAINT `fk_jugador_has_Joc_Joc1`
```

```
43      FOREIGN KEY (`Joc_id`)
44      REFERENCES `BDJocs`.`Joc` (`id`)
45      ON DELETE NO ACTION
46      ON UPDATE NO ACTION) ENGINE = InnoDB;
```

1.3. Representació de la informació amb el model orientat a objectes

A l'igual que el model Entitat-Relació, el model Orientat a Objectes és un model de dades conceptual, però que centra la importància en el modelat d'objectes.

Un objecte pot representar qualsevol element conceptual: entitats, processos, accions… Un objecte no representa únicament les característiques o propietats, sinò que se centra també en els processos que aquests sofreixen. En termes de del model orientat a objectes, diem que un objecte són dades més operacions o comportament.

A la unitat d'introducció ja vam fer un repàs de la programació orientada a objectes, pel que ens limitarem ara a fer un breu repàs dels principals conceptes:

- Un **objecte** és una entitat amb certes propietats i determinat comportament.
- En termes de POO, les propietats es coneixen com **atributs**, i el conjunt de valors d'aquestes determinen l'**estat** de l'objecte en un determinat moment.
- El comportament, ve determinat per una sèrie de funcions i procediments que anomenem **mètodes**, i que modifiquen l'estat de l'objecte.
- Un **objecte** tindrà a més un nom pel que s'identifica.
- Una **classe** és una abstracció d'un conjunt d'objectes, i un objecte ha de pertànyer neccessàriament a alguna classe.
- Les **classes defineixen els atributs i mètodes** que posseïran els objectes d'aquesta classe.
- Un objecte es diu que és una **instància** d'una classe.

Exemple

El mateix exemple que abans, amb una representació orientada a objectes, i un tant simplificada ens quedaría:

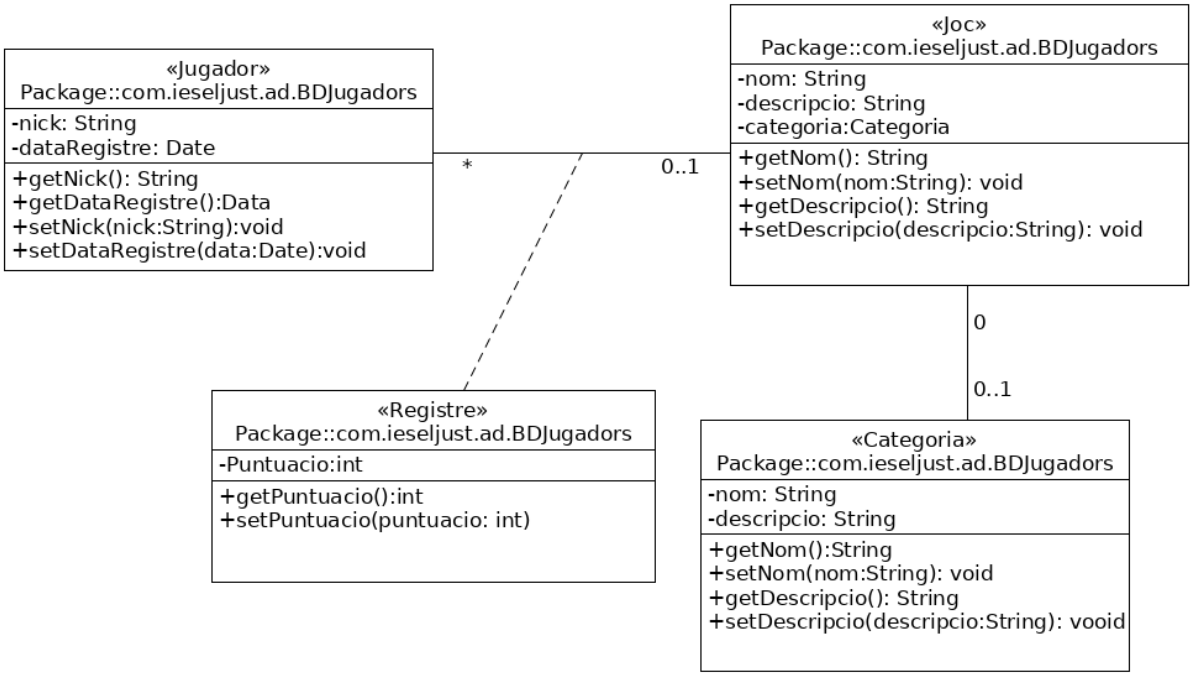


Figura 2: Diagrama de classes Jocs

Com veiem, guarda una estructura similar, a la que hem afegit també alguns mètodes com els getters i els setters. A més, les diferents classes no posseeïxen un atribut identificador, ja que cada objecte s'identifica per ell mateixa.

Veiem una xicoteta aproximació a com implementaríem aquesta jerarquía en Java.

- La classe `Genere` simplement emmagatzema el nom i la descripció i implementa els getters i setters, tal qual apareix al diagrama UML:

```
1
2  /*
3      Classe com.ieseljust.ad.BDJugadors.Genere
4  */
5
6  package com.ieseljust.ad.BDJugadors;
7
8  public class Genere{
9      protected String nom;
10     protected String descripcio;
11
12     public Genere(String nom, String descripcio){
13         this.nom=nom;
14         this.descripcio=descripcio;
15     };
```

```
16
17      public String getNom(){ return this.nom; }
18      public void setNom(String nom){this.nom=nom; }
19
20      public String getDescripcio(){ return this.descripcio; }
21      public void setDescripcio(String descripcio){ this.descripcio=
            descripcio;}
22
23  }
```

- La classe `Joc` emmagatzema el nom i la descripció d'aquest, i a més, un objecte de classe Genere, per indicar el seu gènere. A diferència del model relacional, on el que s'emmagatzema sería una clau externa a gènere, com que aci no tenim claus externes, emmagatzemem un objecte en sí.

```
1   /*
2       Classe com.ieseljust.ad.BDJugadors.Joc
3   */
4   package com.ieseljust.ad.BDJugadors;
5
6
7   public class Joc{
8       protected String nom;
9       protected String descripcio;
10      protected Genere genere;
11
12      public Joc(String nom, String descripcio, Genere genere)
13      {
14          this.nom=nom;
15          this.descripcio=descripcio;
16          this.genere=genere;
17
18      }
19
20      public String getNom(){return this.nom;}
21      public void setNom(String nom){this.nom=nom;}
22
23      public String getDescripcio(){return this.descripcio;}
24      public void setDescripcio(String descripcio){this.descripcio=
            descripcio;}
25
26      public Genere getGenere(){return this.genere;}
27      public void setGenere(Genere genere){this.genere=genere;}
28
29  }
```

- La classe `Registre` representarà la relació entre el jugador i el joc, i emmagatzema una puntuació i una referència a un objecte de tipus joc.

```java
/*
    Classe com.ieseljust.ad.BDJugadors.Registre
*/

package com.ieseljust.ad.BDJugadors;

import java.util.Date;
import java.util.HashSet;
import java.util.Set;

class Registre{
    private int puntuacio;
    private Joc joc;

    public Registre(Joc joc, int puntuacio){
        this.joc=joc;
        this.puntuacio=puntuacio;
    }

    public int getPuntuacio(){ return this.puntuacio;}
    public void setPuntuacio(int puntuacio){ this.puntuacio=puntuacio;}

    public Joc getJoc(){ return this.joc;}
    public void setJoc(Joc joc){ this.joc=joc;}

}
```

- I finalment, la classe `Jugador`, guardarà el seu nick i la data de registre, i a més, un conjunt d'objectes de tipus `Registre`, que emmagatzemarà les puntuacions del jugador en els diferents jocs en què ha jugat.

```java
/*
    Classe com.ieseljust.ad.BDJugadors.Jugador
*/


public class Jugador{
    private String nick;
    private Date dataRegistre;
    private Set<Registre> puntuacions;

    public Jugador(String nick, Date dataRegistre){
        this.nick=nick;
        this.dataRegistre=dataRegistre;
        puntuacions=new HashSet<Registre>();
    }

    public String getNick(){ return this.nick; }
    public void setNick(String nick){ this.nick=nick;}
```

```
19
20      public Date getDataRegistre(){ return this.dataRegistre; }
21      public void setDataRegistre(Date dataRegistre){this.dataRegistre=
            dataRegistre;}
22
23      public Set getPuntuacions(){return this.puntuacions;}
24      public void setPuntuacio(Joc joc, int puntuacio){
25          Registre registre=new Registre(joc, puntuacio);
26          this.puntuacions.add(registre);
27      }
28  }
```

> **La interfície Set i la classe HashSet**
>
> `Set` és una interfície del paquet java.util que tracta amb una col·lecció o conjunt d'elements desordenat i sense duplicats.
>
> Per la seua banda, `HashSet` és una classe que implementa la interfície `Set`, i que es basa en una taula hash, una estructura de dades que permet localitzar objectes en base a una clau que indica la posició en la taula, permetent l'accés directe a l'element, el que les fa ideals per a recerques, insercions i esborrats.

1.4. Model Relacional vs Model OO

Conceptualment, el model orientat a objecte és un model dinàmic, que se centra en els objectes i en els processos que aquestes sofreixen, però que no té en compte, de partida, la seua persistència. Hem d'aconseguir, doncs, poder guardar els estats dels objectes de forma permanent, i carregar-los quan els necessite l'aplicació, així com mantenir la consistència entre aquestes dades emmagatzemades i els objectes que les representen a l'aplicació.

Una forma d'oferir aquesta persistència als objectes sería fer ús d'un SGBD Relacional, però ens trobariem amb algunes complicacions. La primera, des del punt de vista conceptual, és que el model entitat-relació se centra en les dades, mentre que el model orientat a objectes, se centra en els objectes, entesos com a agrupacions de dades, i les operacions que s'hi realitzen sobre ells.

Una altra diferència, bastant important és la vinculació d'elements entre un i altre model. Per una banda, el model relacional afig informació extra a les taules en forma de clau externa, mentre que en el model orientat a objectes, no necessitem aquestes dades externes, sinò que la vinculació entre objectes es fa a través de referències entre ells. Un objecte, per exemple, tampoc necessitarà una clau primària, ja que l'objecte s'identifica per ell mateix.

Tal i com hem vist a l'exemple dels apartats anteriors, les taules al model relacional tenían una clau primària, per identificar els objectes i claus externes per expressar les relacions, mentre que en el model orientat a objectes, aquestes desapareixen, expressant les relacions entre objectes mitjançant

referències. A més, la forma d'expressar aquestes relacions també és diferent. Al model relacional, per exemple, el registre de *Puntuacions* és una taula que enllaça la taula *Jugador* amb la taula *Joc*, i afig a aquesta la puntuació del jugador al joc. En canvi, en la implementació en Java que hem fet, disposem d'objectes de tipus *Registre* que emmagatzemen una puntuació i una referència a *Joc*, però és la classe *Jugador*, qui manté el conjunt de *Registres* de les seues puntuacions.

Per altra banda, a l'hora de manipular les dades, cal tindre en compte que el model relacional disposa de llenguatges (SQL principalment) pensat exclussivament per a aquesta finalitat, mentre que en un llenguatge orientat a objectes es treballa de forma diferent, pel que caldrà incorporar mecanismes que permeten realitzar des del llenguatge de programació aquestes consultes. A més, quan obtenim els resultats de la consulta, ens trobem també amb altre problema, i és la conversió de resultats. Una consulta a una base de dades sempre torna un resultat en forma de taula, pel que caldrà transformar aquestes en estats dels objectes de l'aplicació.

Totes aquestes diferències suposen el que es coneix com «**desfasament objecte-relacional**», i que ens obligarà a fer determinades conversions entre objectes i taules quan volem guardar la informació en un SGBDR. En aquesta unitat i les següents, veurem com superar aquest desfasament des de diferents aproximacions.

2. Connectors

2.1. L'arquitectura client-servidor en els SGBDR

Una vegada som coneixedors del desfasament objecte-relacional, anem a centrar-nos en com accedir a bases de dades relacionals des dels llenguatges de programació.

Els SGBDR, es van popularitzar als anys 80, i són els més estesos actualment. Llevat d'algunes excepcions, treballen seguint una arquitectura client-servidor, pel que disposem d'un servidor on es troba el SGBDR i diversos clients que es connecten al servidor i fan les corresponents peticions.

Els SGBDR oferíen llenguatges de programació, però estàven molt vinculats a aquests, i el manteniment de les aplicacions resultava molt costós. Per aquest motiu, la tendència va ser desvincular el SGBDR del llenguatge de programació, i fer ús d'estàndards de connexió entre ells.

Gràcies a l'arquitectura client-servidor, els SGBDR van poder separar dades per una banda, i els programes d'accés a elles per altra. Aquesta versatilitat va tindre un xicotet inconvenient, i és que calia desenvolupar per una banda el servidor, però per altra, també la part client per tal de poder-se connectar als servidors. Aquestes connexions entre clients i servidors requeriran de protocols i llenguatges específics. Naix aci el concepte de *middleware*, entès coma una capa intermèdia de persistència, formada per llibreríes, llenguatges i protocols situats en client i servidor i que permeten connectar la base de dades amb les aplicacions.

Tot i que inicialment cada SGBD implementava solucions específiques, es van imposar estàndards, entre els que trobem el llenguatge de consulta SQL (Structured Query Language), i que va suposar un gran avanç, ja que unificava la forma d'accedir a les bases de dades, tot i que les aplicacions seguíen requerint una API per poder fer ús de SQL.

2.2. Prootocols d'accés a bases de dades

Quan parlem de protocols d'accés a BD, ens trobem amb dues normes principals de connexió:

- **ODBC (Open Data Base Connectivity)**: Es tracta d'una API (Application Program Interface) desenvolupada per Microsoft per a sistemes Windows que permet afegir diferents connectors a diverses bases de dades relacionals basaden en SQL, de forma senzilla i transparent. Mitjançant ODBC les aplicacions poden obrir connexions amb la base de dades, enviar consultes, actualitacions i gestionar els resultats.
- **JDBC (Java Database Connectivity)**, que defineix una API multiplataforma, que poden utilitzar els programes en Java per tal de connectar-se als SGBDR.

2.3. JDBC

Com hem dit, JDBC és una API per a la connexió a base de dades específica de Java. El seu mode de funcionament és el següent:

- S'ofereix una API, encapsulada en classes, que garanteix uniformitat en la forma en què les aplicacions es connecten a la base de dades, independentment del SGBDR subjacent.
- Necessitarem un controlador per a cada base de dades a la que ens volguem connectar.

Java no disposa de cap biblioteca específica ODBC, però sí, per tal de no perdre el potencial d'aquestes connexions, es van incorporar uns drivers especials que actúen d'adaptadors entre JDBC i ODBC, de manera que es permet, mitjançant aquest *pont*, connectar qualsevol aplicació Java amb qualsevol connexió ODBC. Actualment, quasi tots els SGBD disposen de drivers JDBC, però en cas que no se'n disopsen, es pot fer ús d'aquest pont ODBC-JDBC.

2.3.1. Arquitectura de JDBC

La biblioteca estàndard de JDBC ofereix un conjunt d'interfícies sense implementació. D'aquesta implementació s'encarregaran els controladors o drivers de cada SGBD en qüestió. Les aplicacions, per tal d'accedir a la base de dades, hauran d'utilitzar les interfícies de JDBC, de manera que siga per a l'aplicació totalment transparent la implementació que realitza cada SGBD.

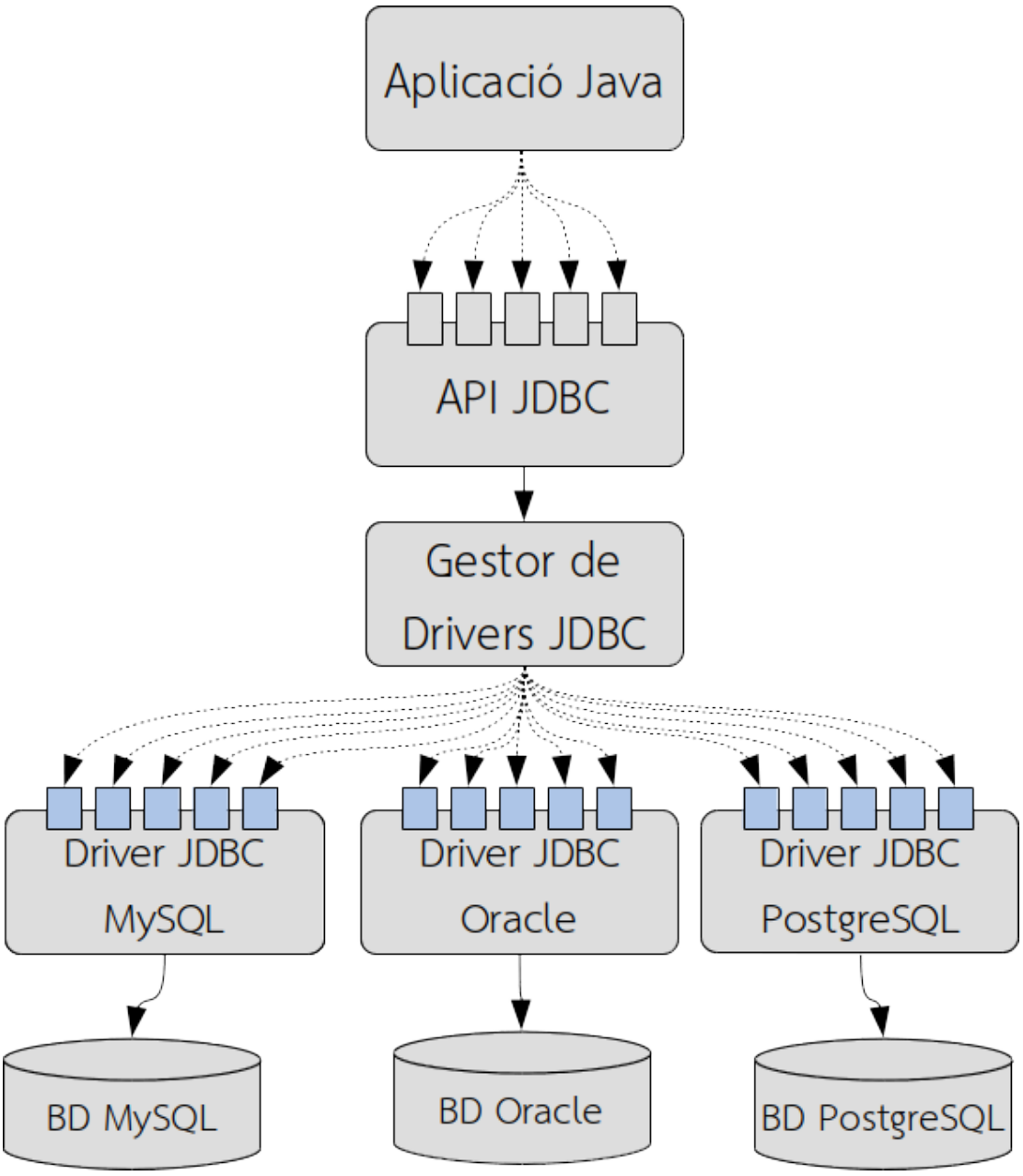


Figura 3: Arquitectura JDBC

Com veiem, Les aplicacions Java accedeixen als diferents mètodes que especifica l'API en forma d'interfícies, però són els controladors els qui accedeixen a la base de dades.

Cal dir, que les apliccions poden utilitzar diversos controladors JDBC de forma simultània, i accedir, per tant, a diverses bases de dades.L'aplicació, especifica un controlador JDBC mitjançant una URL (Localitzador Universal de Recursos) al gestor de Drivers, i aquest és qui s'encarrega d'establir de

forma correcta les connexions amb les bases de dades a través dels controladors.

Els controladors poden ser de diferents tipus:

- **Tipus I o Controladors pont**, caracteritzats per fer ús de tecnologia externa a JDBC i actuar d'adaptador entre JDBC i la tecnología concreta utilitzada. Un exemple és el pont JDBC-ODBC.
- **Tipus II o controladors amb API parcialment nativa,** o controladors natius. Estan formats per una part Java i d'altra que fa ús de biblioteques del sistema operatiu. El seu ús es deu a alguns SGBD que incorporen connectors propietaris que no segueixen cap estàndard (solen ser anteriors a ODBC/JDBC).
- **Tipus III o controladors Java vía protocol de xarxa**, que són controladors desenvolupats en Java que tradueixen les crides JDBC a un protocol de xarxa contra un servidor intermedi. Es tracta d'un sistema molt flexible, ja que els canvis en la implementació de la base de dades no afecten les aplicacions.
- **Tipus IV o Java purs/100%**, també anomenats de protocol natiu, i es tracta de controladors escrits totalment en Java. Les peticions al SGBD es fan a través del protocol de xarxa que utilitza el propi SGVD, pel que no es necessita codi natiu al client ni un servidor intermediari.Es l'alternativa que ha acabat imposant-se, ja que no requereix cap tipus d'instal·lació.

2.4. MySQL

2.4.1. Instal·lació i creació de la BD

Per tal de començar a treballar amb connectors, ho farem connectant-nos a MySQL. Per a això ens fa falta un servidor amb el paquet `mysql-server` instal·lat. Tenim l'opció de fer-ho en local, amb una màquina virtual, o mitjançant un contenidor amb *Docker*, que serà l'opció escollida.

A la documentació de la unitat disposeu d'un document sobre Docker (*Docker per a desenvolupadors*) i com crear un contenidor amb aquest servei, de manera que no ens faça falta crear el servidor en la nostra màquina, sinò que disposem d'un contenidor específica per al SGBD que posem en marxa només quan el necessitem

Així doncs, i per a la resta de document, anem a suposar que ja disposeu de la imatge Docker de MySQL i d'un contenidor funcionant al port **3308** (MySQL utilitza per defecte el 3306, però al nostre contenidor exposarem el servei pel 3308, per si ja teniu un servidor MySQL en local funcionant pel 3306).

Recordeu, que per crear el contenidor amb MySQL haviem de fer:

```
1 $ docker run --name mysql-srv \
```

```
2 >                 -p 3308:3306 \
3 >                 -v /srv/mysql:/var/lib/mysql \
4 >                 -e MYSQL_ROOT_PASSWORD="root" \
5 >                 -d mysql
```

Mentre que si ja tenim el contenidor creat, només l'hem d'iniciar amb:

```
1 $ docker start mysql-srv
```

Instal·lació del client MySQL Workbench El que sí que utilitzarem per treballar amb la base de dades des del nostre equip és el *MySQL Workbench*, que ens servirà per tal de crear nous esquemes i realitzar consultes. Per instal·lar-lo farem:

```
1 $ sudo apt-get install mysql-workbench
```

Una vegada instal·lat, tindrem el MySQL Workbench disponible dins la categoría de Programació en el menú d'inici.

Quan accedim a ell, ens trobem amb la següent finestra:

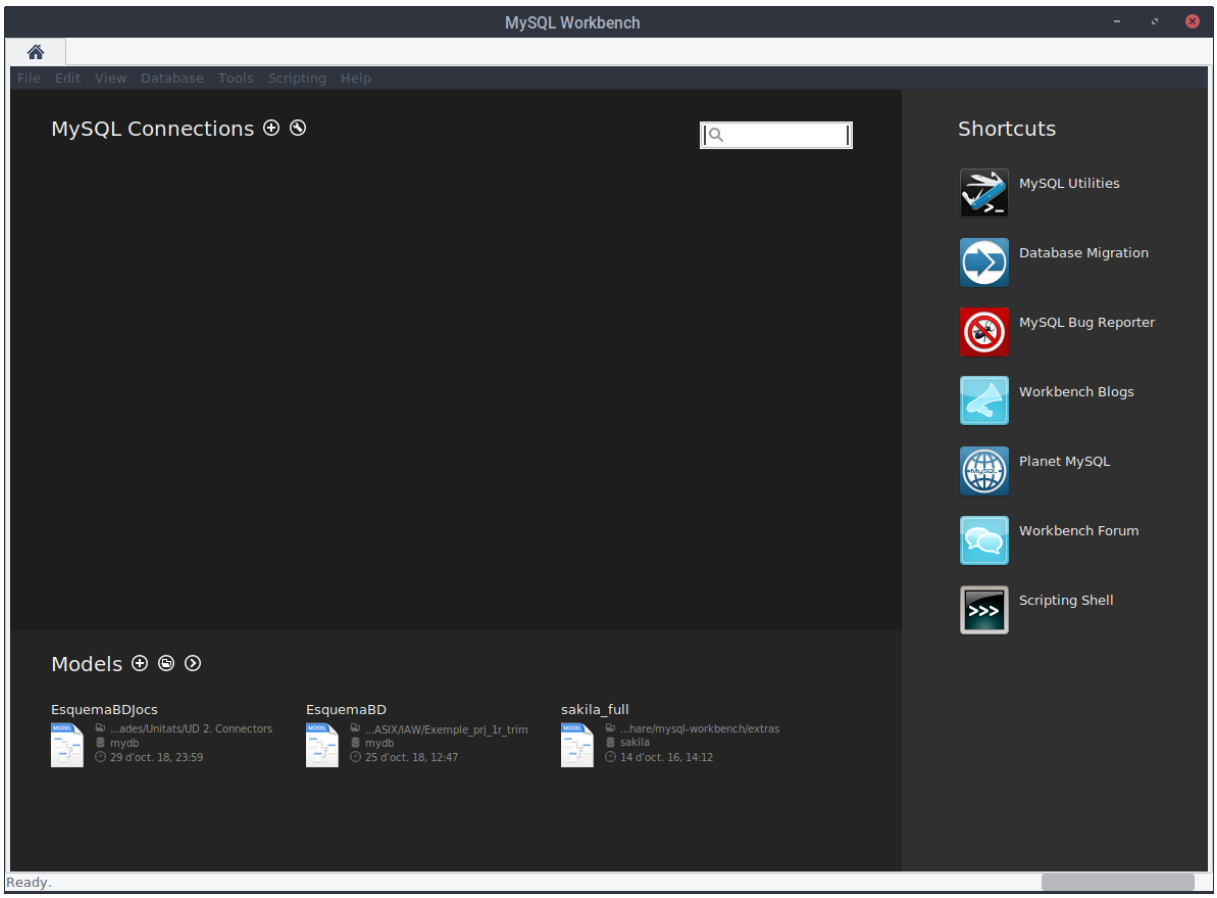


Figura 4: Entrada a MySQL Workbench

En aquesta finestara, farem clic al (+) que hi ha al costat de «MySQL Connections», per indicar que volem afegir una nova connexió. En aquesta finestra de configuració de la nova connexió, donarem nom a aquesta, especificarem l'adreça IP (127.0.0.1 si es tracta del nostre ordinador), i el port al que ens connectem. En cas de connectar-nos a un servei MySQL en local, fariem:

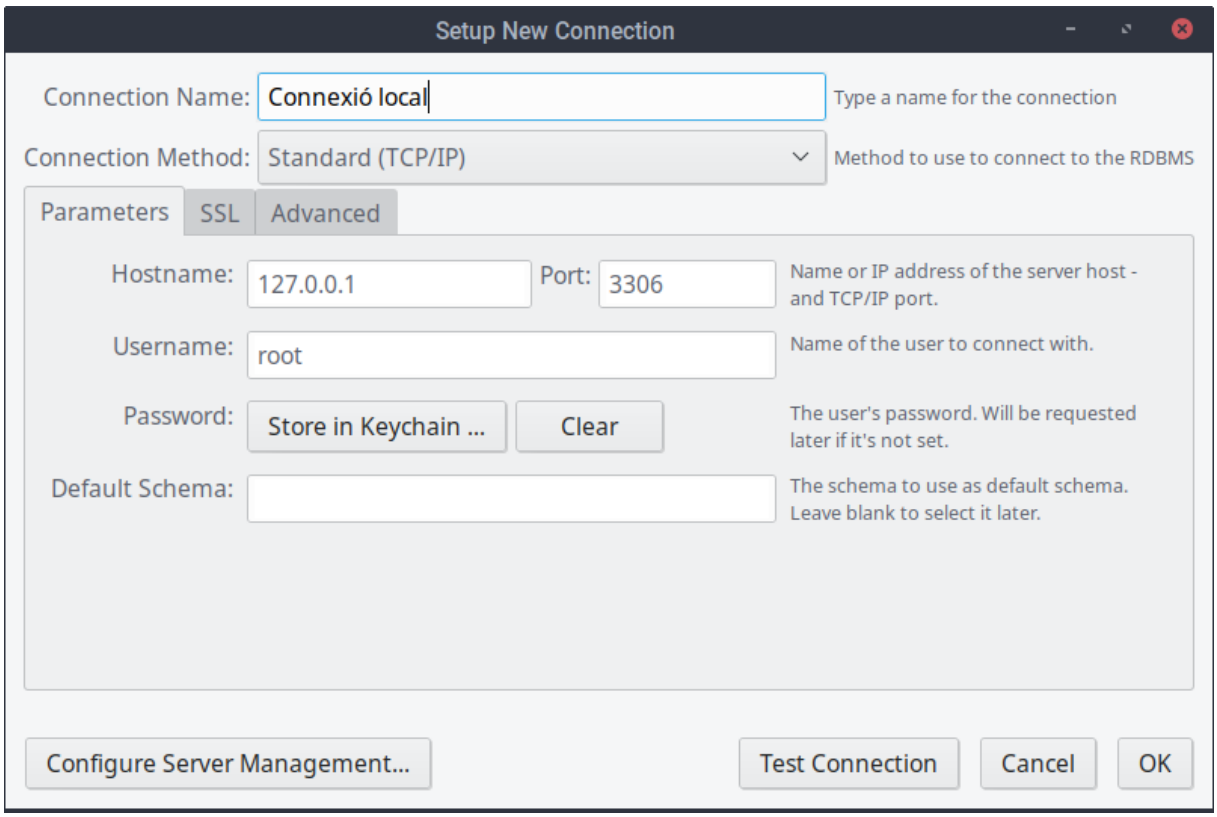


Figura 5: Entrada a MySQL Workbench

I si ens connectem al servei MySQL oferit pel contenidor Docker que hem creat, farem:

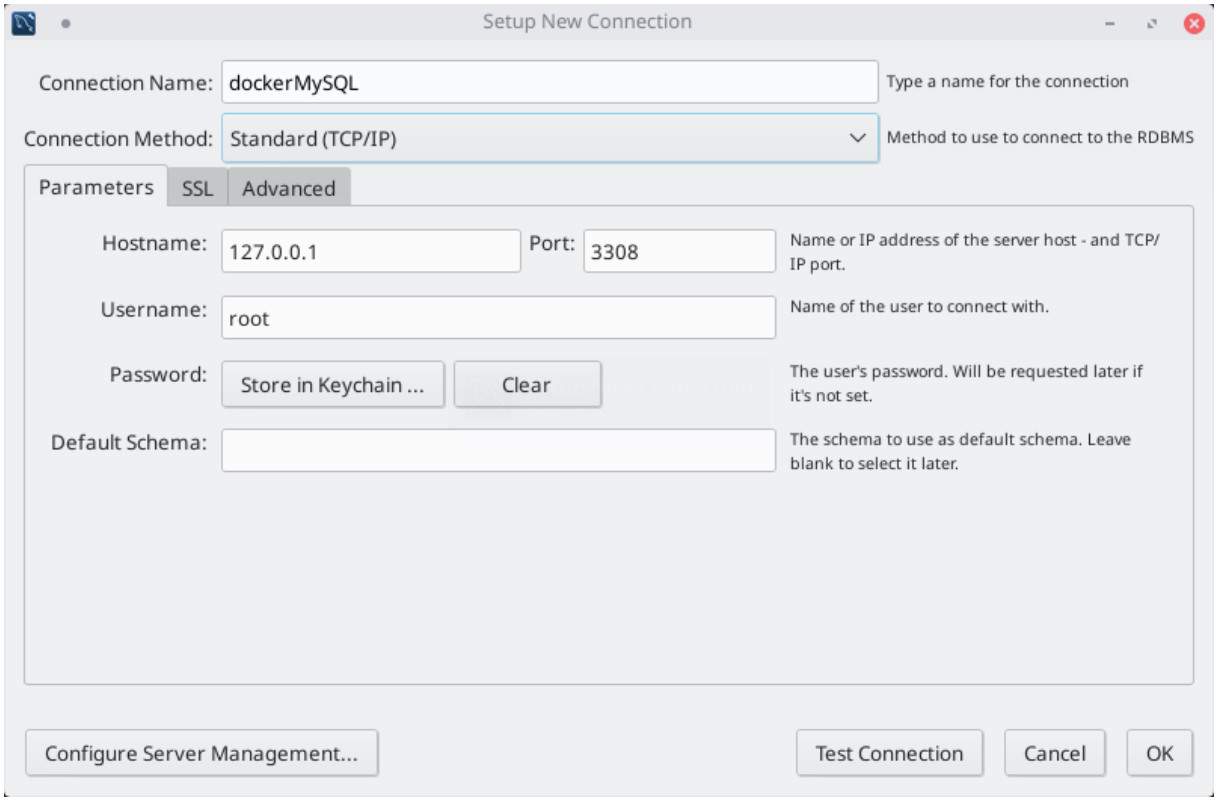


Figura 6: Entrada a MySQL Workbench

En aquesta finestra també caldrà indicar l'usuari i la contrassenya que vam especificar en crear el servei.

Una vegada creada i testejada la connexió, podem entrar en aquesta, i podem importar (*File* > *Open SQL Script*) un fitxer .sql, per tal de crear el nostre esquema. Indiquem l'esquema `esquemaBDJocs.sql` que disposeu al Moodle d'aules:

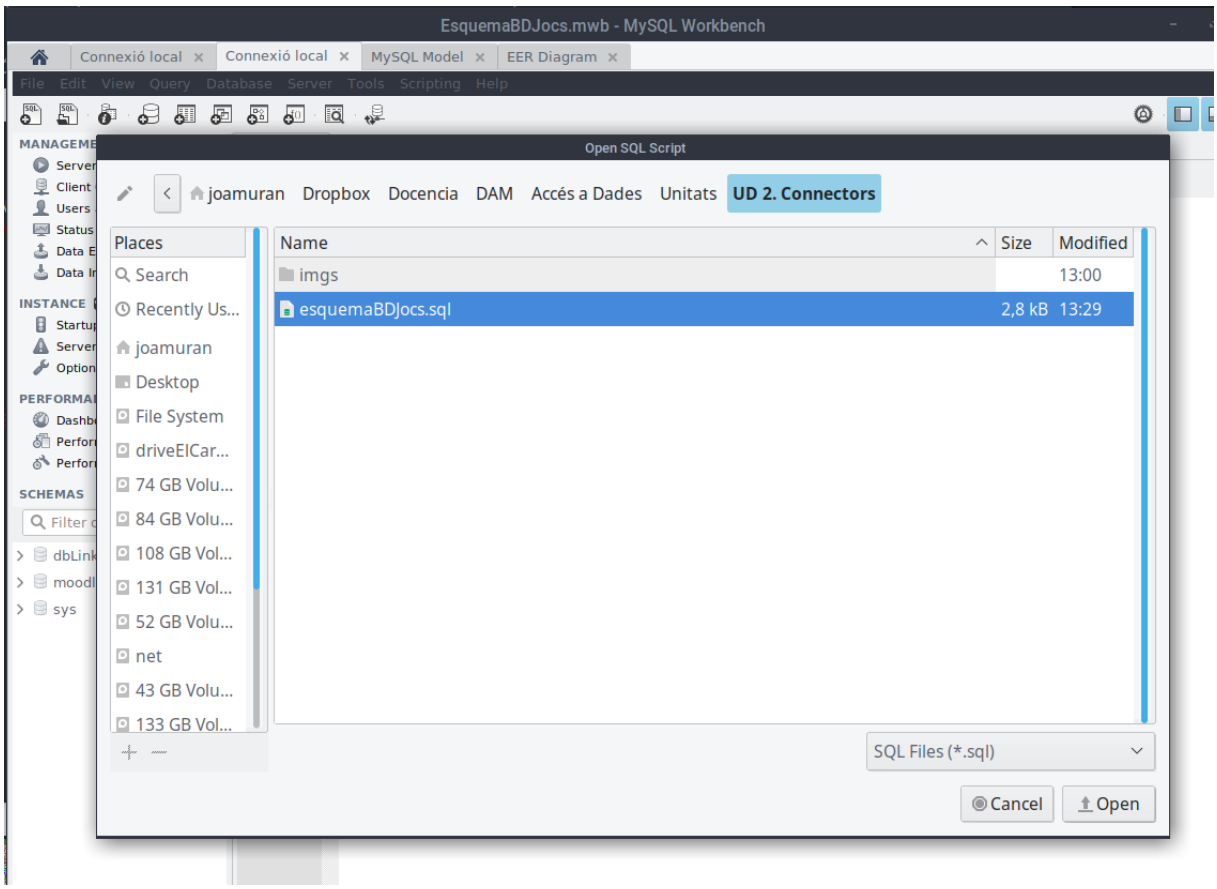


Figura 7: Entrada a MySQL Workbench

Una vegada tenim l'script carregat, fem clic a la icona d'executar la consulta, i tindrem creada la nostra base de dades. A través de *File > Run SQL Script*, també podríem haver executat directament l'script.

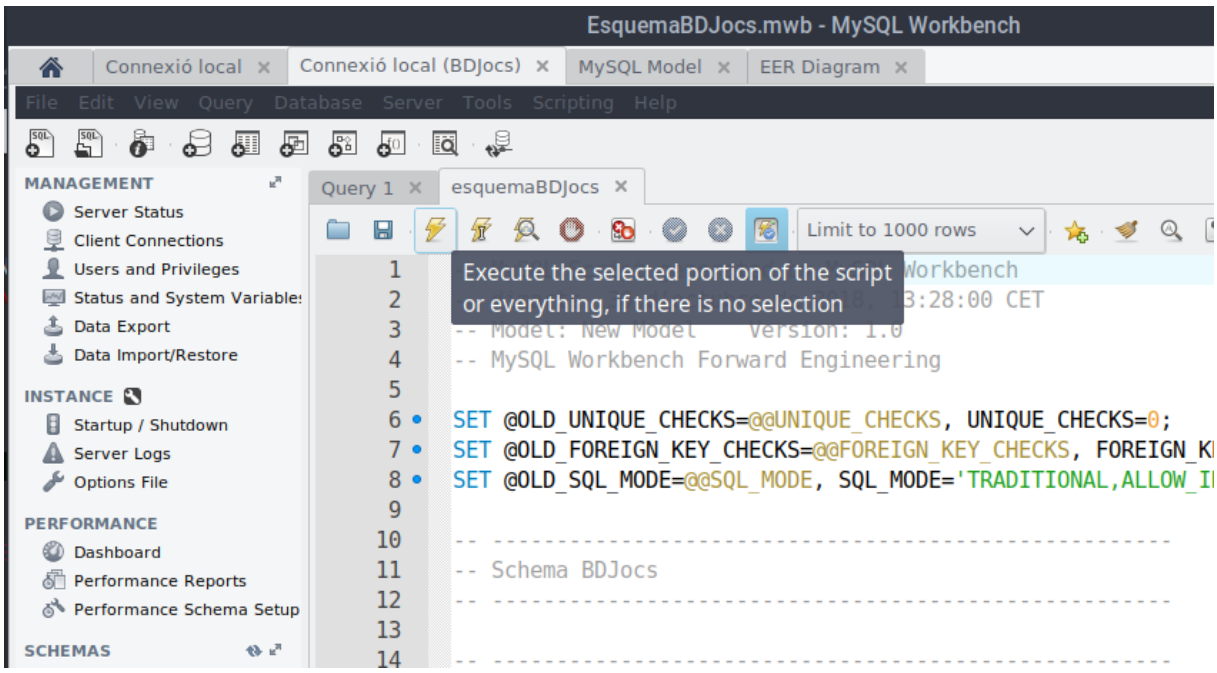


Figura 8: Entrada a MySQL Workbench

Una vegada hajam executat l'script, d'una o altra manera, podem refrescar els esquemes disponibles, i veurem la nostra base de dades acabada de crear:

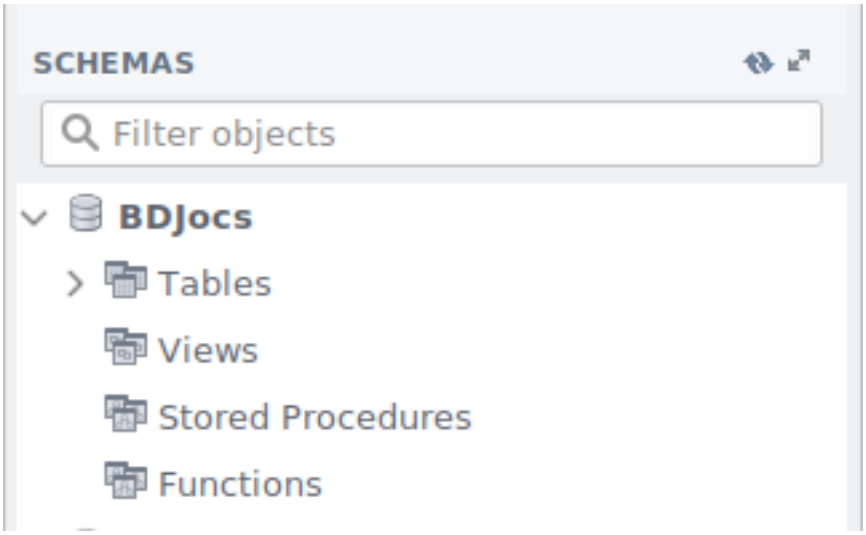


Figura 9: Entrada a MySQL Workbench

Ara, anem a afegir alguns registres (de moment de prova), per tal de tindre alguna taula poblada. En una nova consulta, executarem:

```
INSERT INTO `BDJocs`.`Genere` (`id`, `nom`, `descripció`) VALUES ('1', 'Arcade', 'Es caracteritzen per la jugabilitat simple, repetitiva i d\'acció ràpida.');
INSERT INTO `BDJocs`.`Genere` (`id`, `nom`, `descripció`) VALUES ('2', 'Plataformes', 'Es controla a un personatge que ha d\'avançar per un escenari evitant obstacles');
INSERT INTO `BDJocs`.`Genere` (`id`, `nom`, `descripció`) VALUES ('3', 'Shoot\'em up', 'Jocs de dispars amb perspectiva en dues dimensions, caracteritzats per l\'ús continu de dispars, la millora de les armes, l\'avança automàtic i l\'enfrontament amb enemics de gran tamany al final de cada missió.');
INSERT INTO `BDJocs`.`Genere` (`id`, `nom`, `descripció`) VALUES ('4', 'Agilitat mental', 'Posen a prova la intel·ligéncia del jugador per a la resolució de problemes, bé de caràcter matemàtic, espacial o lògic');
```

2.4.2. Controladors JDBC de MySQL

Anem a crear un nou projecte Gradle en el que inclourem com a dependència el connector JDBC per a mysql.

Per a això, inicialitzem el projecte (recordeu fer-ho en una carpeta ja creada per a ell, al nostre cas, li hem dit *bdjocs*):

```
$ gradle init
Starting a Gradle Daemon (subsequent builds will be faster)

Select type of project to generate:
  1: basic
  2: groovy-application
  3: groovy-library
  4: java-application
  5: java-library
  6: kotlin-application
  7: kotlin-library
  8: scala-library
Enter selection (default: basic) [1..8] 4

Select build script DSL:
  1: groovy
  2: kotlin
Enter selection (default: groovy) [1..2] 1

Select test framework:
  1: junit
  2: testng
  3: spock
Enter selection (default: junit) [1..3] 1
```

```
25
26 Project name (default: bdjocs):
27 Source package (default: bdjocs):
28
29 BUILD SUCCESSFUL in 2m 1s
30 2 actionable tasks: 2 executed
```

Com veieu, hem creat una aplicació Java, amb l'script `build.gradle` en format *groovy*, framework de proves *junit* i com a nom de projecte *bdjocs* que és el nom de la carpeta on ens trobàvem.

Ara, editem el fitxer `build.gradle`, i afegim una nova dependència. El connector que anem a utilitzar és el connector *MySQL Connector/J*, versió *8.0.18*, que es tracta d'un driver de tipus 4 per a MySQL (https://mvnrepository.com/artifact/mysql/mysql-connector-java/8.0.18).

El contingut de `build.gradle` quedaría de la següent manera (llevem els comentaris per defecte per tal que quede més clar:

```
1  plugins {
2      id 'java'
3      id 'application'
4  }
5
6  repositories {
7      jcenter()
8      // Afegim els repositoris de Maven
9      mavenCentral()
10 }
11
12 dependencies {
13     implementation 'com.google.guava:guava:26.0-jre'
14     testImplementation 'junit:junit:4.12'
15     // https://mvnrepository.com/artifact/mysql/mysql-connector-java
16     compile group: 'mysql', name: 'mysql-connector-java', version: '
           8.0.18'
17 }
18 mainClassName = 'com.ieseljust.ad.BDJugadors.mySqlConnect'
```

Com veiem, hem afegit la línia que indica la dependència del controlador `mysql-connector-java` .

Ara creem la carpeta de fonts en dins el projecte, amb la següent estructura:

```
1 src/
2 `-- main
3     `-- java
4         `-- com
5             `-- ieseljust
6                 `-- ad
7                     `-- BDJugadors
8                         `-- mySqlConnect.java
```

Per a això, des de la carpeta del projecte (*bdjocs*), farem:

```
/bdjocs$ mkdir -p src/main/java/com/ieseljust/ad/BDJugadors
```

A més, **caldrà eliminar la carpeta de tests, que no anem a utilitzar per evitar errors de compilació**:

```
$ rm -r src/test
```

Finalment, en la carpeta `src/java/main/com/ieseljust/ad/BDJugadors` crearem el fitxer `mySqlConnect.java` amb el següent contingut:

```
1
2  package com.ieseljust.ad.BDJugadors;
3
4  import java.sql.Connection;
5  import java.sql.DriverManager;
6  import java.sql.ResultSet;
7  import java.sql.SQLException;
8
9  public class mySqlConnect {
10
11
12     public static void main (String [] args ) throws
           ClassNotFoundException, SQLException{
13         Class.forName("com.mysql.cj.jdbc.Driver");
14
15         String connectionUrl = "jdbc:mysql://localhost:3308/BDJocs?
               useUnicode=true&characterEncoding=UTF-8&user=root&password=
               root";
16         Connection conn = DriverManager.getConnection(connectionUrl);
17
18         ResultSet rs = conn.prepareStatement("show tables").
               executeQuery();
19
20         System.out.println("\nTaules de la base de dades:\n");
21
22         while(rs.next()){
23             String s = rs.getString(1);
24             System.out.println(s);
25         }
26
27         System.out.println("\nRegistres de la taula Genere:\n");
28
29         rs = conn.prepareStatement("select * from Genere").executeQuery
               ();
30
31         while(rs.next()){
32             String id = rs.getString(1);
```

```
33              String nom = rs.getString(2);
34              String desc = rs.getString(3);
35              System.out.println(id+" "+nom+" "+desc);
36          }
37
38      }
39
40  }
```

Analitzem un poc el codi:

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

Amb açò hem carregat el driver. El mètode Class.forName accepta com a paràmetre una cadena de text, amb el nom complet de la classe a carregar en eixe moment. Els controladors JDBC disposen d'una classe, anomenada generalment Driver, que s'encarrega d'establir la connexió amb el SGBD. El nom de cada classe dependrà del fabricant, pel que caldrà consultar la documentació corresponent a cada driver.

```
1  String connectionUrl = "jdbc:mysql://localhost:3308/BDJocs?useUnicode=
       true&characterEncoding=UTF-8&user=root&password=root";
2  Connection conn = DriverManager.getConnection(connectionUrl);
```

Amb aquestes línies hem realitzat la connexió a la base de dades. La primera d'elles estableix la URL de connexió, i indica:

- El driver: jdbc:mysql,
- El servidor i el port (localhost:3308),
- La Base de dades a utilitzar, i els paràmetres de connexió, amb la codificació, l'usuari i la contrassenya.

I finalment, fem dues consultes, amb el següent esquema:

1. Crear un «preparedStatement» en la connexió, executar-lo i guardar-lo en un resultset:

```
1  ResultSet rs = conn.prepareStatement("show tables").executeQuery();
```

2. Mostrar els resultats mentre hi haja informació al resultset:

```
1  System.out.println("\nTaules de la base de dades:\n");
2
3  while(rs.next()){
4      String s = rs.getString(1);
5      System.out.println(s);
6  }
```

Finalment, construim el projecte i l'executem per veure el resultat:

```
$ gradle build

BUILD SUCCESSFUL in 2s
6 actionable tasks: 1 executed, 5 up-to-date
```

```
$ gradle run

> Task :run

Taules de la base de dades:

Genere
Joc
Puntuacions
jugador

Registres de la taula Genere:

1 Arcade Es caracteritzen per la jugabilitat simple, repetitiva i d'acció ràpida.
2 Plataformes Es controla a un personatge que ha d'avançar per un escenari evitant obstacles
3 Shoot'em up Jocs de dispars amb perspectiva en dues dimensions, caracteritzats per l'ús continu de dispars, la millora de les armes, l'avança automàtic i l'enfrontament amb enemics de gran tamany al final de cada missió.
4 Agilitat mental Posen a prova la intel·ligéncia del jugador per a la resolució de problemes, bé de caràcter matemàtic, espacial o lògic

BUILD SUCCESSFUL in 6s
2 actionable tasks: 2 executed
```

3. Peticions a les bases de dades

A l'apartat anterior hem vist com connectar-nos a una base de dades, executar una consulta i obtenir uns resultats. Recordem que per tal de realitzar la connexió a la base de dades, calía:

1. Carregar del driver, amb `Class.forName("com.mysql.cj.jdbc.Driver");`
2. Crear la URL de connexió i indicar-li al DriverManager que ens volem connectar:

```
String connectionUrl = "jdbc:mysql://localhost:3308/BDJocs?useUnicode=true&characterEncoding=UTF-8&user=root&password=root";
Connection conn = DriverManager.getConnection(connectionUrl);
```

En aquest apartat, anem a veure com realitzar peticions a la base de dades amb una miqueta més de detall.

3.1. Consultes i actualitzacions de la base de dades

Per tal de realitzar una operació sobre la base de dades, **primer hem de preparar la consulta, i després executar-la.**

Per tal de crear consultes, JDBC ens ofereix les següents interfícies/classes:

Interfície	Descripció
Statement	S'utilitza de forma general, i és útil quan volem realitzar sentències SQL estàtiques, ja que no accepta paràmetres
PreparedStatement	S'utilitza quan volem llançar diverses peticions, i a més, es permet realitzar sentències parametritzables
CallableStatement	S'usa per tal d'accedir a procediments emmagatzemats a la base de dades, i també accepta paràmetres d'entrada

I per tal d'executar aquestes, tenim els mètodes:

Mètode	Descripció
executeQuery	Executa sentències de les que esperem que tornen dades (consultes SELECT)
executeUpdate	Exeuta sentències que no s'espera que retornen dades, sinò que serviran per modificar la base de dades connectada (consultes INSERT, DELETE, UPDATE, CREATE TABLE)

3.1.1. Peticions amb executeUpdate

Com hem dit, amb executeUpdate podem realitzar totes les sentències SQL per modificar tant l'estructura de la base de dades (CREATE TABLE, CREATE VIEW, ALTER TABLE, DROP TABLE), i per a modificar el contigut de les taules (INSERT, UPDATE, DELETE).

Per tal d'executar consultes d'aquest tipus haurem de:

1. Instanciar un Statement a partir d'una connexió activa,
2. Executar la sentència SQL amb executeUpdate
3. Tancar l'objecte Statement.

Per exemple, per inserir un nous registres en la taula «Jocs»:

```
1
2  package com.ieseljust.ad.BDJugadors;
3
4  import java.sql.Connection;
5  import java.sql.DriverManager;
6  import java.sql.SQLException;
7  import java.sql.Statement;
8
9  public class inserirJocs {
10
11     public static void main(String[] args) {
12         Connection con = null;
13         Statement st = null;
14         String sentSQL = null;
15
16         try {
17             Class.forName("com.mysql.cj.jdbc.Driver");
18
19             String connectionUrl = "jdbc:mysql://localhost:3308/BDJocs?
                   useUnicode=true&characterEncoding=UTF-8&user=root&
                   password=root";
20             con = DriverManager.getConnection(connectionUrl);
21
22             st = con.createStatement();
23
24             sentSQL = "INSERT INTO Joc VALUES (1, 'Double Dragon', 'Dos
                    germans bessons experts en arts marcials s`han de fer
                   camí en un escenari urbà on membres de bandes rivals
                   volen deixar-los fora de combat.', 1);";
25             st.executeUpdate(sentSQL);
26
27             sentSQL = "INSERT INTO Joc VALUES (2, 'Tetris', 'Tetris és
                   un videojoc de tipus trencaclosques inventat per l`
                   enginyer informàtic rus Aleksei Pàjitnov l`any 1985
                   mentre treballava a l`Académia de Ciéncies de Moscou',
                   4);";
28             st.executeUpdate(sentSQL);
29
30         } catch (SQLException ex) {
31             System.out.println("Error " + ex.getMessage());
32         } catch (ClassNotFoundException ex) {
33             System.out.println("No s'ha trobat el controlador JDBC (" +
                    ex.getMessage() + ")");
34
35         } finally {
```

```
36              try {
37                  if (st != null && !st.isClosed()) {
38                      st.close();
39                  }
40              } catch (SQLException ex) {
41                  System.out.println("No s'ha pogut tancar el Statement")
                        ;
42              }
43              try {
44                  if (con != null && !con.isClosed()) {
45                      con.close();
46                  }
47              } catch (SQLException ex) {
48                  System.out.println("No s'ha pogut tancar la connexió");
49              }
50          }
51      }
52  }
```

De la mateixa manera que hem fet amb l'exemple anterior, podrem fer consulter per a crear taules a la base de dades, modificar-les, o eliminar registres.

3.1.2. Peticions amb executeQuery

Com hem comentat, el mètode executeQuery ens retorna resultats. Aquests resultats es guarden en un objecte de la classe ResultSet, que representa un conjunt de resultats.

Així doncs, quan utilitzem executeQuery, ho farem de forma semblant a:

```
1  ResultSet rs = st.executeQuery(sentSQL);
```

En l'objecte `rs`, de tipus Resultset, tindrem el resultat de la consulta organitzat per files. En cada consulta que fem al *resultset* podrem accedir a una d'aquestes files. Per tal de recórrer totes les files, haurem de fer ús del mètode `next()` de `Resultset`, de manera que s'avance a la fila següent. Quan obtenim el `Resultset` després de la consulta, aquest estarà posicionat en la primera fila. Quan arribem a la última, `next()` ens retornarà *fals*.

Dins de cada fila, podrem accedir a les diferents columnes, indicant el número de columna (o el nom si el SGBD el suporta) i el tipus de dada a retornar. Els mètodes per obtenir cada tipus de dada començaran per `get`, més el tipus de dada: `getString` per obtenir dades de tipus `String`, `getInteger` per a enters, etc. **Cal tindre en compte que la primera columna és la 1 i no la 0, com passa en molts casos en programació**.

Errors freqüents Sovint és habitual en els programadors reutilitzar o reassignar variables o classes que teníem definides. Cal anar amb compte que la reutilització no és una bona opció quan treballem

amb `Statements` o `ResultSets`, ja que es tracta d'interfícies que poden ser implementades de diferents modes, i el seu comportament pot ser imprevisible. **La norma general és doncs, fer ús d'un `Statement` o `Resultset` per cada consulta que desitgem realitzar.**

Alliberament dels recursos Les connexions que realitzem amb la base de dades s'han de tancar només siga possible. Les instàncies de `Connection` i `Statement` guarden molta informació relacionada amb les consultes realitzades, i mantenen actives les sessions iniciades al SGBD, amb el conseqüent consum de recursos del servidor.

Quan al mateix mètode hem de tancar un `Statement` i el `Connection` a partir del qual l'hem creat, primer haurem de tancar l'statement i després la connexió. En cas contrari, quan intentem tancar l'statement, ens botarà una excepció del tipus SQLException, ja que en haver tancat la connexió, hem deixat aquest inaccessible.

A més, també cal incloure tot l'alliberament de recursos fins un bloc finally, de manera que encara que hi hagen errors en l'execució d'algun statement, poguem tancar les connexions. Per altra banda, en cas que hajam de tancar diversos objectes, és possible que es produisca una excepció que evite que es tanquen els objectes posteriors a l'excepció. El que fem en aquest cas, és encapsular cada tancament entre sentències try-catch dins el propi finally.

3.1.3. Ressum

A mode de ressum, recordem que els passos a seguir per tal d'accedir a una base de dades a través de JDBC són:

1. Importem les classes necessàries (java.sql.Connection, java.sql.DriverManager, java.sql.SQLException, java.sql.Statement.)
2. Carregar el driver JDBC (`Class.forName("com.mysql.cj.jdbc.Driver")`).
3. Identificar l'orige de les dades, per establir la URI de connexió,
4. Crear un objecte de tipus `Connection` a trevés del `DriverManager`.
5. Crear un objecte de tipus statement, a partir de la connexió,
6. Executar la consuta mitjançant l'statement,
7. Recuperar les dades de l'objecte amb un `Rsultset`, mitjançant un `execute` o `executeUpdate` .
8. Alliberar el Resultset (`rs.close()`).
9. Alliberar l'Statement. (`st.close()`)
10. Alliberar la connexió (`con.close()`).

3.2. Consultes a la BD sobre metadades

3.2.1. Metadades

Les metadades d'una base de dades descriuen l'estructura que aquesta té: taules de què es compon la base de dades, els camps que componen aquestes taules, els tipus d'aquests camps, etc. Tot i que normalment coneixem prèviament aquesta estructura, és possible que en alguna ocasió necessitem d'ella. Per a això disposem de les interfícies `DatabaseMetaData`, i `ResultsetMetaData`.

DatabaseMetaData La interfície `DatabaseMetaData` ens proporciona informació sobre les taules i vistes de la base de dades, així com la seua estructura.

A la següent taula tenim alguns dels mètodes més rellevants d'aquesta interfície.

Mètode	Descripció
`String getDatabaseProductName()`	Obté el nom del SGBD
`String getDriverName()`	Obté el nom del driver JDBC en ús
`String getURL()`	Obté la URL de la connexió
`String getUserName()`	Obté el nom de l'usuari connectat a la BD
`ResultSet getTables(String cataleg, String esquema, String patroNomTaula, String[] tipus)`	Obté informació de les taules disponibles al catàleg indicat
`ResultSet getColumns( String cataleg, String esquema, String patroNomTaula, patroNomColumna)`	Obté informació de les columnes de la taula especificada al catàleg i esquema indicats
`ResultSet getPrimaryKeys( String cataleg, String esquema, String patroNomTaula)`	Obté la llista de camps que formen la clau principal

Mètode	Descripció
`ResultSet getImportedKeys(String cataleg, String esquema, String patroNomTaula)`	Obté una llista amb les claus externes definides en la taula
`ResultSet getExportedKeys(String cataleg, String esquema, String patroNomTaula)`	Obté una llista amb les claus externes que apunten a aquesta taula

Als exemples disposeu de la classe `getMetaData`, que exemplifica com utilitzar alguns d'aquests mètodes. Anem a analitzar algunes de les seues parts:

1. **En primer lloc, carreguem els drivers i creem la connexió, com sempre que anem a accedir a la base de dades:**

```
1 // Carreguem el driver JDBC
2 Class.forName("com.mysql.cj.jdbc.Driver");
3 //  Creem la connexió a la base de dades
4 Connection con = DriverManager.getConnection("jdbc:mysql://localhost
      :3308/BDJocs", "root", "root");
```

Fixeu-se que en aquest cas, no hem indicat la cadena de connexió com en exemples anteriors, on passàvem un paràmetre amb tota la cadena, sinò que hem utilitzat tres paràmetres: la cadena, indicant el driver, ordinador, port i base de dades; i dos paràmetres més, representant el nom d'usuari i la contrassenya.

2. **Obtenim de les metadades del SGBD amb `getMetaData` i les mostrem en un format amigable:**

```
1
2 // Obtenim les metadades de la BD amb getMetaDada
3 DatabaseMetaData dbmd = con.getMetaData();
4
5 // Pintem la capçalera de la taula de resultats
6 System.out.println(ConsoleColors.CYAN_BOLD_BRIGHT+"\nInformació del
      SGBD");
7 System.out.println("-------------------\n"+ConsoleColors.RESET);
8 System.out.println(ConsoleColors.WHITE_BOLD+"SGBD:\t"+ConsoleColors.
      RESET + dbmd.getDatabaseProductName());
9 System.out.println(ConsoleColors.WHITE_BOLD+"Driver:\t"+ConsoleColors.
      RESET  + dbmd.getDriverName());
```

```
10 System.out.println(ConsoleColors.WHITE_BOLD+"URL:\t"+ConsoleColors.
       RESET  + dbmd.getURL());
11 System.out.println(ConsoleColors.WHITE_BOLD+"Usuari:\t"+ConsoleColors.
       RESET  + dbmd.getUserName());
12 System.out.println();
13 System.out.println(ConsoleColors.CYAN_BRIGHT+"\n----------------------
       Taules-------------------------\n"+ConsoleColors.RESET);
```

Com veiem, hem fet ús d'alguns dels mètodes indicats més amunt per obtenir en nom del SGBD, el Driver, l'URL de connexió i l'usuari. La classe estàtica `ConsoleColors` és simplement una classe que defineix les constants per mostrar l'eixida en colors.

3. **Obtenim les taules de la base de dades BDJocs amb getTables**

```
1  System.out.println(ConsoleColors.CYAN+String.format("%-15s %-15s %-15s"
       ,"Base de dades","Taula","Tipus"));
2  System.out.println("
       ------------------------------------------------------"+
       ConsoleColors.RESET);
3
4  // I recorrem els resultats amb un resultset:
5  ResultSet rsmd = dbmd.getTables("BDJocs", null, null, null);
6  while (rsmd.next()) {
7      System.out.println(String.format("%-15s %-15s %-15s",rsmd.getString
           (1),rsmd.getString(3),rsmd.getString(4)));
8  }
9
10 rsmd.close(); // Tanquem el resultset
```

Com veiem, hem mostrat una capçalera per tal de mostrar l'eixida tabulada, i per a això hem fet ús d'`String.format`, de manera que indiquem, com a primer paràmetre una cadena de format, a la qual, amb `%-15s` indiquem que mostre 15 caràcters de cadascuna de les cadenes que li passem en els arguments següents i a més, que els justifique a l'esquerra.

En aquesta capçalera, indiquem que anem a mostrar: la base de dades, el nom de la taula, i el tipus d'aquesta (si realment és una taula, o és una vista).

Després de mostrar la capçalera, invoquem el mètode `getTables`, que ens retorna un `ResultSet`. Si ens fixem en la capçalera d'aquesta funció `getTables` i la comparem amb la crida que fem:

```
1 ResultSet            getTables(String cataleg, String esquema, String
      patroNomTaula, String[] tipus)
2 ResultSet rsmd = dbmd.getTables("BDJocs",           null,
                  null,           null);
```

Estem indicant que ens mostre les taules del catàleg *BDJocs*, és a dir, la base de dades, i no indiquem cap esquema ni cap nom de taula o tipus. En aquest cas, si no indiquem cap d'aquests arguments, el mètode interpreta que els volem tots.

En aquest punt, cal incidir en què que els termes *catàleg* i *esquema* solen prestar-se a confusió. Segons els estàndards, un catàleg conté diversos esquemes, amb informació detallada del sistema, des de la forma d'emmagatzemament intern fins els esquemes conceptuals. En un catàleg, sembre hi ha un esquema anomenat INFORMATION_SCHEMA, amb les vistes i els dominis de l'esquema d'informació del sistema.

De totes formes, la majoría dels SGBDs fan correspondre el catàleg amb la base de dades. Sense anar més lluny, en aquesta consulta indiquem el nom de la base de dades com a catàleg, mentre que si obrim el MySQLWorkbench, la base de dades està representada com a «schema».

Podem trobar més informació al respecte en aquests enllaços: * https://stackoverflow.com/questions/7022755/whats-the-difference-between-a-catalog-and-a-schema-in-a-relational-database * https://www.quora.com/What-is-the-difference-between-system-catalog-and-database-schema-in-a-Database

Bé, una vegada vist açò, ens anem a centrar en la línia:

```
System.out.println(String.format("%-15s %-15s %-15s",rsmd.getString(1),
    rsmd.getString(3),rsmd.getString(4)));
```

Que és la que mostra els diferents camps d'interès. Si ens fixem, en ella accedim a les columnes 1, 3 i 4 del `ResultsetMetaData rsmd`. Quina informació contenen estes columnes? Per a això cal consultar l'API de Java pera DataBaseMetaData, a la qual se'ns indica que aquestes columnes corresponen a:

```
    1. TABLE_CAT String => table catalog (may be null)
       ..
    3. TABLE_NAME String => table name
    4. TABLE_TYPE String => table type. Typical types are "TABLE", "VIEW
       ", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS"
       , "SYNONYM".
```

Que com veiem, es correspon al catàleg (nom de la BD), el nom de la taula, i el seu tipus.

4. **Obtenim les columnes de la taula**

Finalment, l'exemple pregunta el nom d'una taula (emmagatzemat a la variable `taula`), i ens mostra els camps que aquesta conté. Per a això farem ús de `getColumns`, però abans, farem un parell de consultes, per esbrinar les claus primàries i externes de la base de dades:

```
// Calculem les claus primàries de la taula i les afegim a un ArrayList
ResultSet rspk = dbmd.getPrimaryKeys("BDJocs", null, taula);
ArrayList<String> pks = new ArrayList<String>();

while (rspk.next())
    pks.add(rspk.getString(4));

rspk.close(); // Tanquem el resultset
```

Com veiem, hem fet ús del mètode `getPrimaryKeys`, per obtenir les claus primàries de la base de dades «BDJocs», i la taula que se'ns haja indicat a la variable `taula`. El resultat és un resultset, del que ens interessa el camp 4 (una altra vegada, hem de consultar l'api de Java per al mètode getPrimaryKeys de la classe DatabaseMetaData. Aquest camp, com veurem a la documentació és el nom de la columna.

Així doncs, el que fem és crear un ArrayList que emmagatzemarà els noms de les columnes que formen la clau primària, per a utilitzar posteriorment.

Ara anem a calcular les claus externes, d'una forma molt semblant:

```
1  // Calculant les claus externes i les guardem en dos arraylist,
2  // Un per a la clau, i altre per a la taula a la que referencia
3  ResultSet rsfk = dbmd.getImportedKeys("BDJocs", null, taula);
4  ArrayList<String> fks = new ArrayList<String>();
5  ArrayList<String> fksExt = new ArrayList<String>();
6
7  while (rsfk.next()){
8      fks.add(rsfk.getString(8)); // Guarda el camp de la Clau Externa
9      fksExt.add(rsfk.getString(3)); // Guarda la taula a que fa referé
           ncia
10 }
11
12 rsfk.close(); // Tanquem el resultset
```

Com veiem, hem fet ús del mètode `getImportedKeys`, del qual tenim la descripció en l'API, i del què ens interessen dues columnes, la 8, que indica el nom de la columna que és clau externa, i la 3, que és el nom de la taula a la que fa referència. Per emmagatzemar aquesta informació, hem fet ús de dos `ArrayList`.

Finalment, obtenim ja el nom de les columnes de la taula demanada:

```
1  // Obtenim les columnes de la taula i juntem la informació obtinguda
2
3  ResultSet columnes = dbmd.getColumns("BDJocs",null , taula, null);
4
5  System.out.println(ConsoleColors.CYAN_BOLD_BRIGHT+"----------- TAULA "
       + taula+" ------------------------"+ConsoleColors.RESET);
6  System.out.println(ConsoleColors.CYAN_BRIGHT+String.format("%-25s %-15s
       %-15s","Atribut/Claus","Tipus","Pot ser nul?"));
7  System.out.println("
       ------------------------------------------------------"+
       ConsoleColors.RESET);
8
9  while (columnes.next()){
10     String columnName=columnes.getString(4);
11     if (pks.contains(columnName))
12         columnName=ConsoleColors.WHITE_UNDERLINED+ConsoleColors.
               WHITE_BOLD_BRIGHT+columnName+"(PK)"+ConsoleColors.RESET;
```

```
13      if (fks.contains(columnName))
14          columnName=ConsoleColors.CYAN_BOLD_BRIGHT+columnName+"(FK) -->
                "+fksExt.get(fks.indexOf(columnName))+ConsoleColors.RESET;
15
16      String tipus=columnes.getString(6);
17      String nullable=columnes.getString(18);
18
19      System.out.println(String.format("%-25s %-15s %15s",columnName,
            tipus,nullable));
20  }
```

Com podem veure, estem utilitzant de nou una eixida formatada amb `String.format`, de manera que mostrem el nom de l'atribut (la columna), el tipus de dada que conté, i si pot ser nul. Aquesta informació, tal i com indica de nou, l'API per a getColumns, es troba respectivament a les columnes 4, 6 i 18 del `ResultSet` resultant.

Com veiem, també hem fet un parell de comprovacions, per veure si la columna està als ArrayLists tant de les claus primàries (`if (pks.contains(columnName))`) o bé de les externes (`if (fks.contains(columnName))`). Si la columna que estem tractant és clau primària, ho ressaltarem i indicarem que és `(PK)`, i si és clau externa, ho indicarem també, `((FK))` i a més, indicarem la taula a la que fa referència. Per a això, obtenim de l'`ArrayList fksExt` l'element que es trobe en la mateixa posició que es trobe la columna que és clau externa a l'`ArrayList fks` (ja que hem anat emmagatzemant anteriorment de forma simultània les claus externes i les taules a les que feien referència en els dos `ArrayList`).

ResultSetMetaData Els `ResultSets` producte d'una consulta també disposen d'un conjunt de metadades. Aquestes metadades es poden obtindre mitjançant el `ResultSetMetaData`. Veiem alguns dels mètodes més rellevants d'aquest:

Mètode	Descripció
`int getColumnCount()`	Obté el nombre de columnes del ResultSet
`String getColumnName(index)`	Obté el nom de la columna indicada en l'índex (recordeu que la primera és 1)
`String getColumnTypeName(index)`	Obté el tipus de la columna

Per veure un xicotet exemple, afegirem a la classe anterior, després de consultar les metadades de la taula, una consulta de tipus `SELECT * FROM taula`, i obtindrem els resultats i les metadades d'aquest, que utilitzarem per tal de mostrar la capçalera i el continut de la taula:

```java
// Ara anem a fer una consulta i comprovar les seues metadades

ResultSet rs = con.createStatement().executeQuery("SELECT * FROM " +
    taula);
System.out.println(ConsoleColors.CYAN_BOLD_BRIGHT);
System.out.println("");
System.out.println("Contingut de " + taula);
System.out.println("*****************************");

ResultSetMetaData rsmdQuery = rs.getMetaData();
for (int i = 1; i <= rsmdQuery.getColumnCount(); i++)
    System.out.print(String.format("%-25.25s",rsmdQuery.getColumnName(i
        )));

System.out.println();
System.out.println(ConsoleColors.RESET);

while (rs.next()) {
    for (int i = 1; i <= rsmdQuery.getColumnCount(); i++)
        System.out.print(String.format("%-25.25s ",rs.getString(i)));
    System.out.println();
    }
```

Com podem veure, hem creat una consulta mitjançant un Statement, i hem guardat el resultat en un resultset. Posteriorment, hem creat un `ResultSetMetaData rsmdQuery` amb `rs.getMetaData()` , i hem recorregut les seues columnes, obtenint informació. Per a això, sabem la longitud del resultset gràcies a `getColumnCount()`, i després, amb `getColumnName` hem obtingut el nom de la columna en cada posició. Després de mostrar la capàlera, hem procedit de forma similar per tal d'obtindre el contingut de les columnes de la consulta.

3.2.2. Execució d'scripts

A l'apartat *3.1. Consultes i actualitzacions de la base de dades* hem vist com fer ús d'`executeUpdate` per tal de llançar peticions de modificació de la base de dades. En aquell apartat hem vist com realitzar peticions simples, però molts SGBDs permeten l'execució de múltiples sentències DDL i DML en una mateixa cadena (per exemple, crear una taula i diversos inserts en ella). Per a això, quan ens connectem a la base de dades, abans que res, cal indicar la propietat `allowMultipleQueries=true`.

Veiem un exemple per a MySQL. Anem a crear un script que elimine la taula «jugadors» de la base de dades «BDJocs», la cree de nou, i a més, inserisca alguns jugadors. L'script SQL per a això seria:

```sql
CREATE DATABASE  IF NOT EXISTS `BDJocs` /*!40100 DEFAULT CHARACTER SET
    utf8 */;
USE `BDJocs`;
```

```
3
4  --
5  -- Table structure for table `jugador2`
6  --
7
8  DROP TABLE IF EXISTS `jugador2`;
9  /*!40101 SET @saved_cs_client     = @@character_set_client */;
10 /*!40101 SET character_set_client = utf8 */;
11 CREATE TABLE `jugador2` (
12   `id` int(11) NOT NULL AUTO_INCREMENT,
13   `nick` varchar(45) DEFAULT NULL,
14   `dataRegistre` datetime DEFAULT NULL,
15   PRIMARY KEY (`id`)
16 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
17 /*!40101 SET character_set_client = @saved_cs_client */;
18
19 --
20 -- Dumping data for table `jugador2`
21 --
22
23 LOCK TABLES `jugador2` WRITE;
24 /*!40000 ALTER TABLE `jugador2` DISABLE KEYS */;
25 INSERT INTO `jugador2` VALUES (1,'Billy Mitchell','2018-11-13 00:00:00'
      ),(2,'Steve Wiebe','2018-11-13 00:00:00'),(3,'Hank Chien','
      2018-11-13 00:00:00');
26 /*!40000 ALTER TABLE `jugador2` ENABLE KEYS */;
27 UNLOCK TABLES;
```

L'script és un bolcat directament amb MySQLWorkbench de la taula jugadors, al qual hem modificat el nom de la taula jugador per jugador2, ja que ara per ara, jugador no es podría esborrar si no esborrem abans la taula *puntuacions*, ja que fa referència a aquesta. Per altra banda, i per a un exemple posterior, hem definit l'id de jugador com a `AUTO_INCREMENT`.

Ara anem a veure el codi per tal d'executar l'script:

```
1  package com.ieseljust.ad.BDJugadors;
2
3  import java.io.BufferedReader;
4  import java.io.File;
5  import java.io.FileNotFoundException;
6  import java.io.FileReader;
7  import java.io.IOException;
8  import java.sql.*;
9
10 public class execScript {
11     public static void run() {
12         File script=new File("sql/jugadors.sql");
13         System.out.println("Executant l'script "+script.getName());
14
15         // Convertim el fitxer a una cadena
```

```
16          BufferedReader br=null;
17          try{
18              br=new BufferedReader(new FileReader(script));
19          } catch(FileNotFoundException e){
20              System.out.println("Error: L'script no existeix.");
21          };
22
23          String line=null;
24          StringBuilder sb= new StringBuilder();
25
26          // Obtenim el símbol del salt de línia segons el sistema
27          String breakLine=System.getProperty("line.separator");
28
29          try{
30              while ((line=br.readLine())!=null) {
31                  sb.append(line);
32                  sb.append(breakLine);
33              }
34           } catch (IOException e){
35              System.out.println ("ERROR d'E/s");
36          }
37
38          // Convertim el stringBuilder en cadena:
39          String query = sb.toString();
40
41          System.out.println("Executant consulta: \n"+query);
42
43          try{
44              Class.forName("com.mysql.cj.jdbc.Driver");
45          } catch (ClassNotFoundException e) {
46              System.out.println("Error en carregar el driver");
47          }
48
49          try{
50              Connection con=DriverManager.getConnection("jdbc:mysql://
                    localhost:3308/BDJocs2?allowMultiQueries=true", "root","
                    root");
51              Statement st=con.createStatement();
52              int result=st.executeUpdate(query);
53
54              System.out.println("Script Executat amb éxit. Eixida: "+
                    result);
55
56              st.close();
57              con.close();
58
59           } catch (SQLException e){
60              System.out.println("Error en l'script "+ e);
61          }
62
63      }
```

```
64
65  }
```

Vegem algunes de les seues peculiaritats:

1. **Ús d'`StringBuilder` per crear l'script**: El mètode `executeUpdate` necessita una cadena de text per executar la consulta. Aquesta cadena de text es crearà concatenent totes les línies del fitxer sql que estem llegint, separant-les amb un retorn de carro. La classe `StringBuilder` ens ofereix una manera d'anar «afegint línies» a un string (ja que un string és immutable), mitjançant el mètode `append`. A més, fem ús de la propietat del sistema `line.separator`, per obtindre quin és el *retorn de carro* del sistema.
2. **allowMultipleQueries**: La connexió s'ha realitzar, com hem anticipat, passant el paràmetre `allowMultipleQueries=true`:

```
1  Connection con=DriverManager.getConnection("jdbc:mysql://localhost
       :3308/BDJocs2?allowMultiQueries=true", "root","root");
```

La resta d'script és bastant intuïtiu: executem la consulta amb `executeUpdate`, i tanquem els resultsets i les connexions corresponents.

3.2.3. Sentències preparades

Anteriorment, hem comentat la diferència entre un `Statement` i un `PreparedStatement`, i no era més que la primera ens servía per executar sentències SQL estàtiques, mentre que la segona ens permetia realitzar sentències parametritzables. Açò vol dir que ens permet construir una cadena SQL amb marcadors de posició que representen les dades que s'assignaran posteriorment. Aquests marcadors, o *placeholders*, s'indiquen amb un `?`.

Anem a veure parts del codi de l'exemple `afigJugadors.java`, que ens deixa afegir jugadors a través de la terminal. Per a això, ens pregunta el nom del jugador (nick), i la data de registre, i l'afig a la base de dades, mitjançant sentències preparades.

1. En primer lloc, el que fem és demanar per teclar el nom del jugador i la data:

```
1  System.out.print(ConsoleColors.GREEN_BOLD_BRIGHT+"# Nom del jugador: "+
       ConsoleColors.RESET);
2  while(!keyboard.hasNext());
3  if (keyboard.hasNext()) nomJugador = keyboard.nextLine();
4
5  System.out.print(ConsoleColors.GREEN_BOLD_BRIGHT+"# Data de registre (
       format YYYY-MM-DD): "+ConsoleColors.RESET);
6  while(!keyboard.hasNext());
7  if (keyboard.hasNext()) DataRegistre = keyboard.nextLine();
```

2. El primer pas per inserir en la base de dades és crear una cadena amb la sentència, i indicant els placeholders (símbol ?) en la consulta:

```
// 1. Creem la senténcia
String sql="INSERT INTO jugador2 (`nick`, `dataRegistre`) VALUES (?, ?)
    ";
```

Fixeu-se que no hem indicat l'id del jugador, ja que, en la taula, tal i com l'hem definit abans, l'id del jugador s'ha definit com a `AUTO_INCREMENT`, de manera que no siga necessari agafar-lo.

3. Una vegada tenim la cadena creada, preparem el Statement amb `prepareStatement` des de la connexió, i donem valor als *placeholders*:

```
PreparedStatement pst=con.prepareStatement(sql, Statement.
    RETURN_GENERATED_KEYS);

// 3. Assignem els paràmetres
pst.setString(1, nomJugador);
pst.setDate(2, java.sql.Date.valueOf(DataRegistre));
```

Reparem en dos detalls:

- Hem afegit com a segon paràmetre del `prepareStatement` la constant predefinida `Statement.RETURN_GENERATED_KEYS`. Aquesta opció realment no és necessària, pel que podríem haver fet `con.prepareStatement(sql)` sense problemes. De tota manera, com qu el camp `id` l'hem definit com a `AUTO_INCREMENT`, açò ens servirà per obtindre l'id de la fila que hem inserit.
- Per altra banda, com que el segon *placeholder* fa referència a un obejcte de tipus `Date`, el que hem de fer és convertir l'string de `DataRegistre` en un tipus de dada compatible amb SQL, la qual cosa aconseguim amb `java.sql.valueOf()`.

A més, com veiem, en l'assignació de valors als placeholders, aci hem fet ús de `setString` i `setDate`, per fer referència a que una columna és de tipus `String` i l'altra de tipus `Date`, i que es corresponen als tipus `VARCHAR` i `DATE` de MySQL. A l'API de PreparedStatement, podem veure els diferents mètodes *set* per establir valor als diferents tipus de dada de MySQL des de les sentències preparades.

4. Finalment, executem la consulta i obtenim el número de files afegides i l'id del jugador que hem afgit:

```
// 4. I executem la consulta
int filesRet=pst.executeUpdate();
System.out.println(ConsoleColors.CYAN_BOLD_BRIGHT+"S'han afegit "+
    filesRet+" files"+ConsoleColors.RESET);

// 5. Obtenim l'id de la última fila inserida
```

```
6  int lastId=-1;
7  ResultSet generatedKeys = pst.getGeneratedKeys();
8  if (generatedKeys.next())
9     lastId=generatedKeys.getInt(1);
10
11 System.out.println(ConsoleColors.CYAN_BOLD_BRIGHT+"L'ID de la fila és "
       +lastId+ConsoleColors.RESET);
```

Com veiem, hem fet el executeUpdate, que ens retorna el número de files afectades per la consulta.

Per altre banda, el mètode `getGeneratedKeys` del `preparedStatement`, ens ha retornat un Resultset amb les claus generades en la última inserció.

3.2.4. Transaccions

Les transaccions són sentències SQL d'acualització (INSERTs, UPDATEs i DELETEs), que s'han d'executar baix el principi *tot o res*: o s'executen totes o no se n'executa cap. Així, es van realitzant les diferents operacions, però aquestes no són efectives fins que no es fa el *COMMIT*. Si alguna cosa falla i volem rebutjar totes les operacions, fem un *ROLLBACK*.

JDBC funciona de manera que cada `Statement` d'una connexió és en sí una transacció (mode autocommit). Per tal de treballar amb transaccions de més d'un `Statement`, cal desactivat aquest mètode, la qual cosa s'aconsegueix amb el mètode `setAutoCommit(false)` de la connexió.

Veiem una variant de l'exemple d'afegir jugadors, a la qual s'afigen tots els jugadors inserits dins una transacció, de manera que, o bé s'afigen tots o bé no se n'afig cap:

```
1  package com.ieseljust.ad.BDJugadors;
2
3  import java.io.BufferedReader;
4  import java.io.File;
5  import java.io.FileNotFoundException;
6  import java.io.FileReader;
7  import java.io.IOException;
8  import java.sql.*;
9  import java.util.Scanner;
10
11
12 public class afigJugadors2 {
13     // Anem a inserir informació a la taula Jugadors
14     public static void run() throws SQLException,
           ClassNotFoundException{
15
16         // Carreguem el driver JDBC
17         Class.forName("com.mysql.cj.jdbc.Driver");
18         //  Creem la connexió a la base de dades
19         Connection con = DriverManager.getConnection("jdbc:mysql://
               localhost:3308/BDJocs", "root", "root");
```

```
20
21          // Establim l'autocommit a fals:
22          con.setAutoCommit(false);
23
24          Scanner keyboard = new Scanner(System.in);
25          String opcio="";
26
27          String nomJugador="";
28          String DataRegistre="";
29          try{
30
31              do {
32                  System.out.print(ConsoleColors.GREEN_BOLD_BRIGHT+"# Nom
                        del jugador: "+ConsoleColors.RESET);
33                  while(!keyboard.hasNext());
34                  if (keyboard.hasNext()) nomJugador = keyboard.nextLine
                        ();
35
36                  System.out.print(ConsoleColors.GREEN_BOLD_BRIGHT+"#
                        Data de registre (format YYYY-MM-DD): "+
                        ConsoleColors.RESET);
37                  while(!keyboard.hasNext());
38                  if (keyboard.hasNext()) DataRegistre = keyboard.
                        nextLine();
39
40
41                  // Inserció en la BD amb un preparedStatement:
42
43                  try{
44                      // 1. Creem la senténcia
45                      String sql="INSERT INTO jugador2 (`nick`, `
                            dataRegistre`) VALUES (?, ?)";
46
47                      // 2. Creem el statement amb la connexió
48                      PreparedStatement pst=con.prepareStatement(sql,
                            Statement.RETURN_GENERATED_KEYS);
49
50                      // 3. Assignem els paràmetres
51                      pst.setString(1, nomJugador);
52                      pst.setDate(2, java.sql.Date.valueOf(DataRegistre))
                            ;
53
54                      // 4. I executem la consulta
55                      int filesRet=pst.executeUpdate();
56                      System.out.println(ConsoleColors.CYAN_BOLD_BRIGHT+"
                            S'han afegit "+filesRet+" files"+ConsoleColors.
                            RESET);
57
58                      // 5. Obtenim l'id de la última fila inserida
59                      int lastId=-1;
60                      ResultSet generatedKeys = pst.getGeneratedKeys();
```

```
61                      if (generatedKeys.next())
62                          lastId=generatedKeys.getInt(1);
63
64                      System.out.println(ConsoleColors.CYAN_BOLD_BRIGHT+"
                            L'ID de la fila és "+lastId+ConsoleColors.RESET)
                            ;
65
66
67                  } catch (SQLException e){
68                      System.out.println(ConsoleColors.RED_BOLD_BRIGHT+"
                            Error en la BD: "+ConsoleColors.RESET);
69                      con.rollback();
70                  } catch(Exception e){
71                      System.out.println("Error: "+e.getMessage());
72                      con.rollback();
73                  };
74
75                  System.out.print(ConsoleColors.GREEN_BOLD_BRIGHT+"
                        Desitges afegir un nou jugador? (s/n) "+
                        ConsoleColors.RESET);
76                  while(!keyboard.hasNext());
77                  if (keyboard.hasNext()) opcio = keyboard.nextLine();
78
79              } while(opcio.equals("s"));
80
81              // Aci al final fem el commit i reestablim l'autocommit
82              con.commit();
83              con.setAutoCommit(true);
84          } catch(Exception e){
85              System.out.println("Error: "+e.getMessage());
86              con.rollback();
87          };
88
89      }
90
91  }
```

3.2.5. Gestió d'errors

Com hem vist als exemples, les causes d'errors en d'SQL poden ser múltiples. Els errors d'SQL es troben definits en l'espeficicació estàndard, que descriu el valor de la variable anomenada SQLSTATE, que identifica l'estat d'uns sentència SQL després de la seua execució. Quan JDBC detecta que després d'una execució el valor d'SQLSTATE es correspon a un error, dispara una excepció del tipus SQLException, que contindrà un missatge que podem obtindre amb `getMessage()` i a més, el valor d'SQLSTATE, que podem recuperar amb `getSQLState()`.

Aquests codis SQLSTATE estan formats per cinc caràcters, els primers indiquen el tipus d'error i els

altres tres els concreten. Podem consultar aquests codis a la pròpia Wikipèdia: https://en.wikipedia.org/wiki/SQLSTATE

Com hem vist, amb la gestió d'excepcions de Java podem capturar excepcions de tipus `SQLException`, o derivades d'aquestes. Una vegada capturades, podme utilitzar el codi SQLSTATE per decidir què fer.

Per a més informació, podem consultar la documentació de l'OpenJDK sobre les excepcions en SQL.