Programació

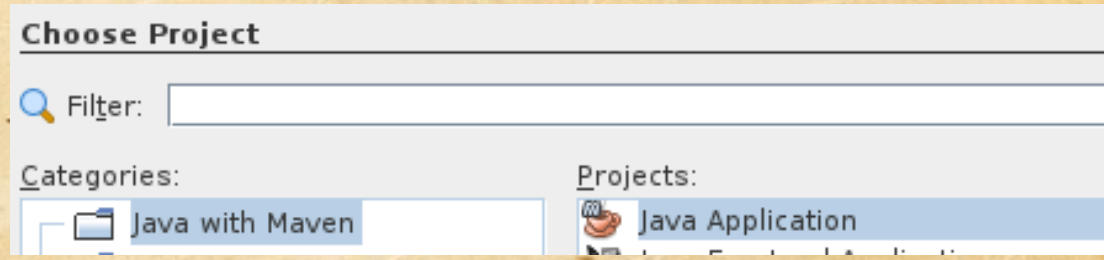
UT10.1. Connexió amb una base de dades relacional

JDBC

- És un controlador (driver) que ofereix una API per a l'execució d'operacions sobre bases de dades des del llenguatge de programació Java, independentment del sistema operatiu on s'execute.
- En el nostre cas connectarem amb una base de dades Oracle, que és amb la que heu treballant durant el curs al mòdul de Bases de Dades.
- Hi ha diverses versions, en funció de la versió de JDK.

Projecte NetBeans

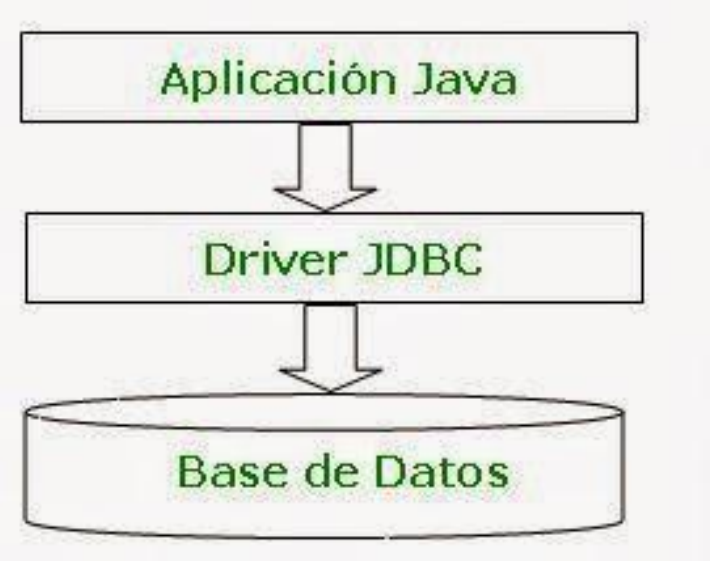
- Crearem un projecte de tipus Maven
- En "Project Files" obrirem el fitxer "pom.xml" i afegim abans de </project>:




```
<dependencies>
      <dependency>
          <groupId>com.oracle.database.jdbc</groupId>
          <artifactId>ojdbc11</artifactId>
          <version>23.4.0.24.05</version>
     </dependency>
</dependencies>
```

JDBC

- Esta API inclou moltes classes que permeten poder treballar amb una base de dades:



- Establir una connexió amb una base de dades.
- Recuperar i manipular la informació que conté la base de dades connectada.

Passos per a establir connexió amb una base de dades

- Una volta tenim disponible la lliberia ojdbc anem a vore quins serien els passos per estabilir la connexió amb la BD.
- La principal classe responsable d'això és la classe DriverManager, proporcionada per l'API del JDBC.

Pas 1: Registrar el driver JDBC

- El driver ha de ser adequat a funció de la base de dades que es va a gestionar junt amb la màquina virtual deJava (JVM) que es vaja a utilitzar.
- Registrar consisteix en carregar la classe que corresponga al driver a la JVM.
- Per això, es faran ús de metaclasses.

Variarà en funció del driver utilitzat

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver");
} catch (ClassNotFoundException ex{
   System.out.println("S'ha produït un error al no trobar eixe driver");
}
```

Pas 1: Registrar el driver JDBC

- **`Class.forName`** càrrega la classe indicada en memòria. Al fer això, tots els elements "static" d'eixa classe s'iniciaran. (es fa automàticament, no hem de programar res)
- En un d'estos iniciadors "static", es realitza la invocació al mètode **`registerDriver`** dela classe **`DriverManager`**

```
public class Driver extends NonRegisteringDriver implements java.sql.Driver {
    //
    // Register ourselves with the DriverManager
    //
    static {
        try {
            java.sql.DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            throw new RuntimeException("Can't register driver!");
        }
    }
```

Pas 1: Registrar undriver

- Per exemple, el driver **`sun.jdbc.odbc.JdbcOdbcDriver`** serviria per controlar una base de dades gestionada per **Access** o **`com.mysql.jdbc.Driver`**per connectar ambuna base de dades **MySQL.**
- La classe `DriverManager` manté un registre dels drivers que s'han carregat (en una mateixa aplicació es poden carregar diversos drivers)
- `Class.forName` pot llançar l'excepció `ClassNotFoundException`, degut a que no trobe eixe driver. Eixa invocació registra (càrrega) el driver en la aplicació invocant internament al mètode de `DriverManager` que calga.

Pas 2: Establir la connexió

- Una volta registrat el driver a el **`DriverManager`**, es podrà establir una connexió amb la BD.
- Per realitzar-la cal utilitzar el mètode **`getConnection`** que es tracta d'un mètode estàtic de la classe**`DriverManager`**
- És un mètode sobrecarregat que té **3 implementacions diferents**

```
public static Connection getConnection(String url,
                                       Properties info)
                                throws SQLException
```

```
public static Connection getConnection(String url,
                                       String user,
                                       String password)
                                throws SQLException
```

```
public static Connection getConnection(String url)
                                throws SQLException
```

Exemple de connexió amb una base de dades Oracle

```java
public static void main(String[] args) {
    final String URL = "jdbc:oracle:thin:javauser/javauser@localhost:1521:ORCLCDB"; // Nom d'usuari i contrasenya: javauser
    final String sentenciaSQL = "SELECT * FROM categoria ORDER BY codigo";

    try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(URL);              // Obtenim la connexió amb "getConnection"
        Statement statement = conn.createStatement();                    // Statement ens permet executar sentències SQL
        ResultSet resultSet = statement.executeQuery(sentenciaSQL);      // executeQuery executa la sentència i retorna un "resultSet"
        while (resultSet.next()) {                                       // Mentre queden registres dins del "resultSet" els recorrem
            int codi = resultSet.getInt("codigo"); //                    // getInt obté un valor enter de la columna "codigo" del registre actual recorregut
            String nom = resultSet.getString("nombre");                  // getString obté un valor String de la columna "nombre" del registre actual recorregut
            System.out.println("Codi: " + codi + " - Nom: " + nom);
        }
        conn.close();                                                    // finalment hem de tancar la connexió
    } catch (ClassNotFoundException ex) {
        System.out.println("No s'ha trobat eixe driver");
    } catch (SQLException ex) {
        System.out.println("S'ha produït un error al executar la sentència SQL");
    }
}
```

```
--- exec-maven-plugin:3.0.0:exec (default-cli) @ ProjecteBD ---
Codi: 1 - Nom: Sobremesa
Codi: 2 - Nom: Portátiles
Codi: 3 - Nom: Móviles
Codi: 4 - Nom: Tecnología
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

https://www.codejava.net/java-se/jdbc/connect-to-oracle-database-via-jdbc

Pool de connexions

- Totes les aplicacions web actuals utilitzen, per connectar amb una base de dades un pool de connexions.
- Si fem l'establiment de connexió com hem vist fins ara, ens trobem el problema de que cada usuari que vulga consultar informació sobre la BD tindrà que fer una petició al servidor per a que este, **obriga una connexió** amb la base de dades i una vegada servida la informació la tanque.
- Obrir i tancar una connexió consumeix molts recursos al servidor.

Pool de connexions

- Si són diversos els usuaris que realitzen peticions d'obertura i tancament de connexió amb la base de dades, el servidor web al final caurà.

Pool de connexions

- Per solucionar esta caiguda del servidor, es crea el pool deconnexions.
- Este pool de connexions especifica **un número de connexions determinades** que estan sempre disponibles (és a dir, el servidor tindrà sempre obertes estes connexions per a poder ser usades per l'usuari que fa la petició)
- A mesura que els usuaris es connecten, li concedirem una d'aquestes connexions.
- Si un usuari ja no necessita més serveis, la connexió amb eixe usuari es tanca, però el servidor segueix amb esta connexió disponible.

Pool de connexions

- Si hi ha més usuaris que connexions disponible, deixa a l'usuari a l'espera de que hi haja una connexió lliure.
- Això provoca una optimització de l'ús de recursos, a més de no provocar una caiguda del sistema.
- **Pertant, el procediment vist anteriorment, s'ha de substituir per la creació d'un pool de connexions.**

Crear un pool de connexions en un projecte Netbeans

- Primer de tot hem d'afegir al nostre projecte una dependència més:

```
<dependency>
        <groupId>org.apache.tomcat</groupId>
        <artifactId>tomcat-jdbc</artifactId>
        <version>10.1.0-M12</version>
      </dependency>
```

```
<dependencies>
    <dependency>
        <groupId>com.oracle.database.jdbc</groupId>
        <artifactId>ojdbc8</artifactId>
        <version>21.5.0.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat</groupId>
        <artifactId>tomcat-jdbc</artifactId>
        <version>10.1.0-M12</version>
    </dependency>
</dependencies>
```

Crear un pool de connexions en un projecte Netbeans

- Ara farem la connexió de la següent manera:

```
public static void main(String[] args) {
    final String sentenciaSQL = "SELECT * FROM categoria ORDER BY codigo";
    final String URL = "jdbc:oracle:thin:@localhost:1521:ORCLCDB";          // Ara ja no indiquem usuari ni password a l'URL
    final DataSource datasource;
    final PoolProperties pool;

    try {
        pool = new PoolProperties();                                        // Creem un nou pool de connexions i definim algunes propietats del pool
        pool.setUrl(URL);
        pool.setDriverClassName("oracle.jdbc.driver.OracleDriver");
        pool.setUsername("javauser");
        pool.setPassword("javauser");
        pool.setMaxActive(15);
        pool.setMaxIdle(10);;
        pool.setMaxWait(5000);

        datasource = new DataSource();                                      // Creem un datasource amb les propietats del pool creat anteriorment
        datasource.setPoolProperties(pool);

        Connection conn = datasource.getConnection();                       // Obtenim la connexió amb "getConnection" del "datasource"
        Statement statement = conn.createStatement();                       // Statement ens permet executar sentències SQL
        ResultSet resultSet = statement.executeQuery(sentenciaSQL);         // executeQuery executa la sentència i retorna un "resultSet"
        while (resultSet.next()) {                                          // Mentre queden registres dins del "resultSet" els recorrem
            int codi = resultSet.getInt("codigo"); //                       // getInt obté un valor enter de la columna "codigo" del registre actual recorregut
            String nom = resultSet.getString("nombre");                     // getString obté un valor String de la columna "nombre" del registre actual recorregut
            System.out.println("Codi: " + codi + " - Nom: " + nom);
        }
        conn.close();                                                       // finalment hem de tancar la connexió

    } catch (SQLException ex) {
        System.out.println("S'ha produït un error al executar la sentència SQL");
    }
}
```

Crear un pool de connexions en un projecte Netbeans

- **maxActive:** Indica el número máximo de conexiones que pueden estar abiertas al mismo tiempo.
- **maxIdle:** El numero máximo de conexiones inactivas que permanecerán abiertas, si el numero de conexiones inactivas es muy bajo puede darse el caso de que las conexiones se cierren porque se llega al máximo de conexiones inactivas y se vuelvan a abrir inmediatamente reduciendo la eficiencia ya que se perdería la ventaja del uso de pool de conexiones en cuanto a que no hay que abrir una conexión cada vez que es necesario.
- **maxWait:** Es el tiempo máximo (en ms) que se esperará a que haya una conexión disponible (inactiva), si se supera este tiempo se lanza una excepción.
- **username:** Usuario de la base de datos.
- **password:** Password para el usuario introducido en username.
- **driverClassName:** Nombre del driver JDBC para conectar con la base de datos.
- **url:** URL de la base de datos a la que nos queremos conectar.

Pool de connexions

- Per obtenir una connexió del pool, el objecte `dataSource` disposa del mètode `getConnection()`

getConnection

```
Connection getConnection()
                    throws SQLException
```

Attempts to establish a connection with the data source that this `DataSource` object represents.

Returns:

a connection to the data source

Throws:

`SQLException` - if a database access error occurs

`SQLTimeoutException` - when the driver has determined that the timeout value specified by the setLoginTimeout method has been exceeded and has at least tried to cancel the current database connection attempt

Gestió de Connection

- Cada vegada que es concedeix una connexió al client corresponent, s'obtindrà un objecte de la classe `Connection`.
- Esta connexió, s'haurà tancar amb el client que la ha demanada (però el pool de connexions seguirà igual) una vegada que ja no la necessite.
- Per tancar una connexió, invocarem al mètode `close()` de l'objecte connection obtingut per evitar consumir recursos innecessaris.

```
Connection connection = this.dataSource.getConnection();
```

```
connection.close();
```

Gestió de Connection

- Des de la versió 7 de Java, disposem d'una estructura molt interessant nomenada `try-with-resources`.
- Consisteix en un bloc try que tindrà entre parentesi la creació d'un recurs (connexió a bd, a fitxers ...) i dins del cos (entre claus) utilitzarà este recurs. Al acabar el try es tanca automàticament el recurs. D'eixa manera, el programador no necessita invocar a `close()`

```
try (Connection conn = datasource.getConnection()) {
    // Dins del bloc "try" es té accés a la conexió "conn"
    // Es farà un close en acabar el bloc "try"
}
```

Pas 3.1: Recuperació de la informació

- Elmètode `createStatement` de `Connection` permet la connexió amb el driver que gestiona la BD. Crea un objecte de la classe `Statement` que tampoc es deu oblidar de tancar (millor és utilitzar també un bloc `try-with-resources`)

- Una volta connectats a la BD,gràcies a l'objecte **`Statement`** es pot accedir a la base de dades per a escriure en ella o recuperar informació

```
try (Connection conn = datasource.getConnection()) {
    try(Statement statement = conn.createStatement()){
        // Ací es tindrà accés tant a l'objecte "conn"
        // com a l'objecte "statement" que es tancaran
        // quan acabe el seu respectiu "try-with-resources"


    }
}
```

Pas 3.1: Recuperar informació

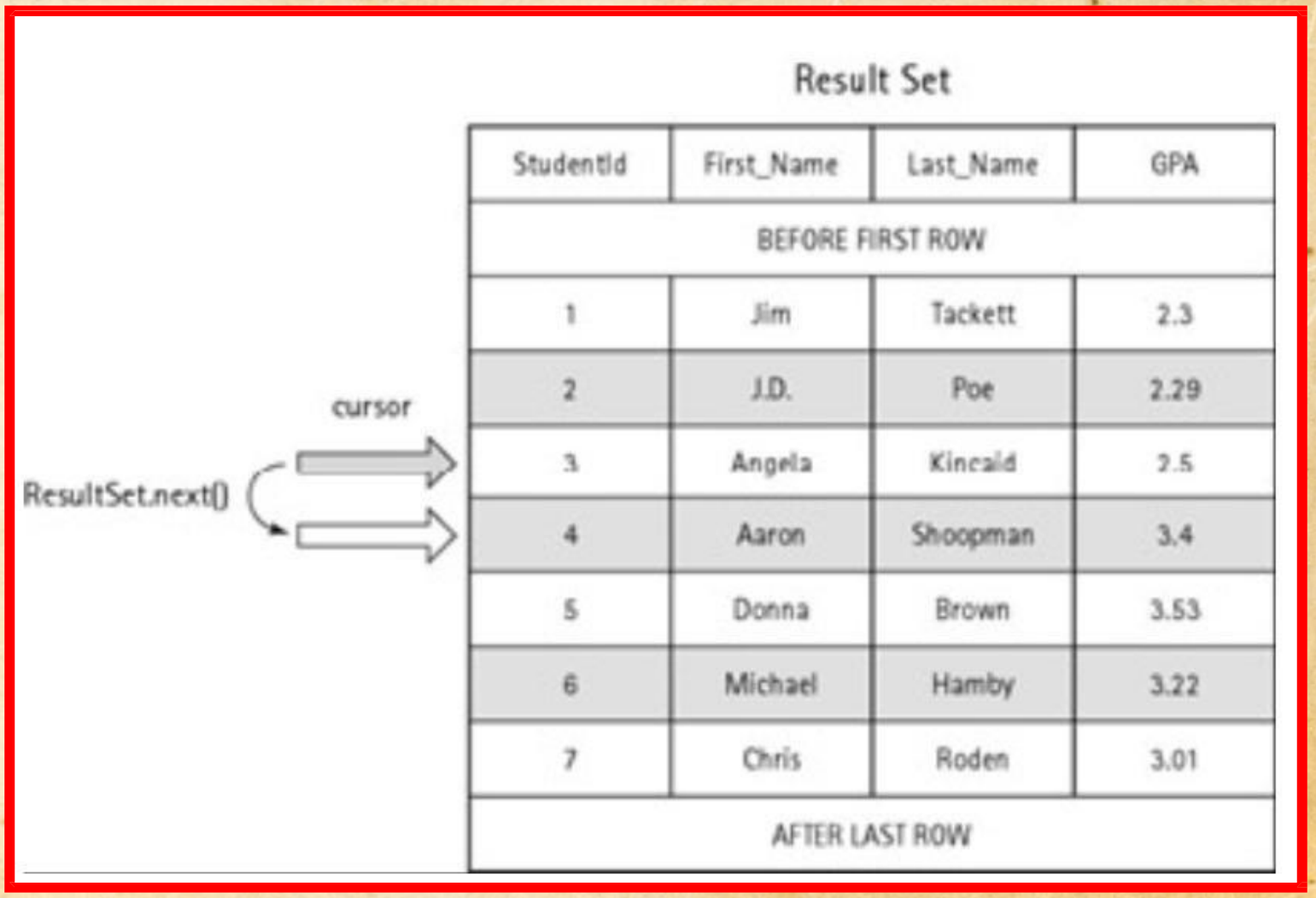
- Per a poder recuperar la informació que conté una BD, podem usar el métode **`executeQuery`** que conté la classe **`Statement`**. (revisa altres mètodes com **executeUpdate**)
- Este mètode retorna un objecte de la classe **`ResultSet`**.
- La informació que conté un objecte **`ResultSet`** està formada per una sèrie de files (registres) i utilitza el concepte de cursor per anar movent-se entre elles
- Tampoc hem d'oblidar tancar este descriptor. Farem servir de nou `try-with-resources`

Pas 3.1: Recuperar informació

```
try (Connection conn = datasource.getConnection()) {
    try(Statement statement = conn.createStatement()){
        try(ResultSet resultSet = statement.executeQuery(sentenciaSQL)){
            // Tindrem accés a conn, statement i resultSet
            // que es tancaran en acabar cada bloc
        }
    }
}
```

```
try (Connection conn = datasource.getConnection();
     Statement statement = conn.createStatement();
     ResultSet resultSet = statement.executeQuery(sentenciaSQL)) {
    //  Ací processem el ResultSet
} catch (SQLException e) {
    // Codi en cas d'excepció SQL
}
```

Exemple de ResultSet (cursor)



Mètodes de ResultSet

- **`boolean next()`** → Següent fila
- **`boolean first()`** → Primera fila
- **`void beforeFirst()`** → Abans de primera fila
- **`boolean previous()`** → Fila prèvia
- **`void afterLast()`** → Després d'última fila
- **`boolean absolut(int f)`** → Posiciona a f
- **`boolean relative(int f)`** → Desplaça f
- **`boolean last()`** → Última fila

Tots llancen una excepció SQLException si el cursor no apunta a cap fila

Mètodes de ResulSet per a recuperació d'informació de fila

- Es disposa de mètodes “get” amb la finalitat d'obtenir els valors dels diferents camps que formen la fila on està posicionat el cursor.
- Per accedir a eixes dades es pot utilitzar la seva posició (número de columna) o nom (nom de columna). Es recomana la segona opció (Oracle per exemple no te ordre de camps)
- Si s'accedeix per la seua posició hi ha que tindre en compte que els camps s'enumeren començant per la posició 1.

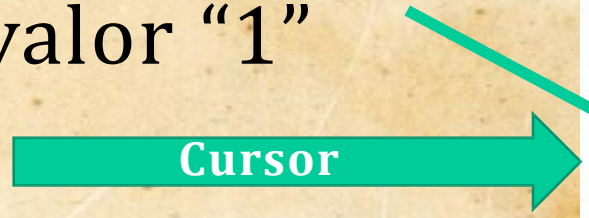
Mètodes de ResulSet per a recuperaciód'informació de fila

`resultset.getInt("StudentId")`

recuperarà el valor "1"

`resulset.getString(3)`

recuperarà el valor "Tackett"

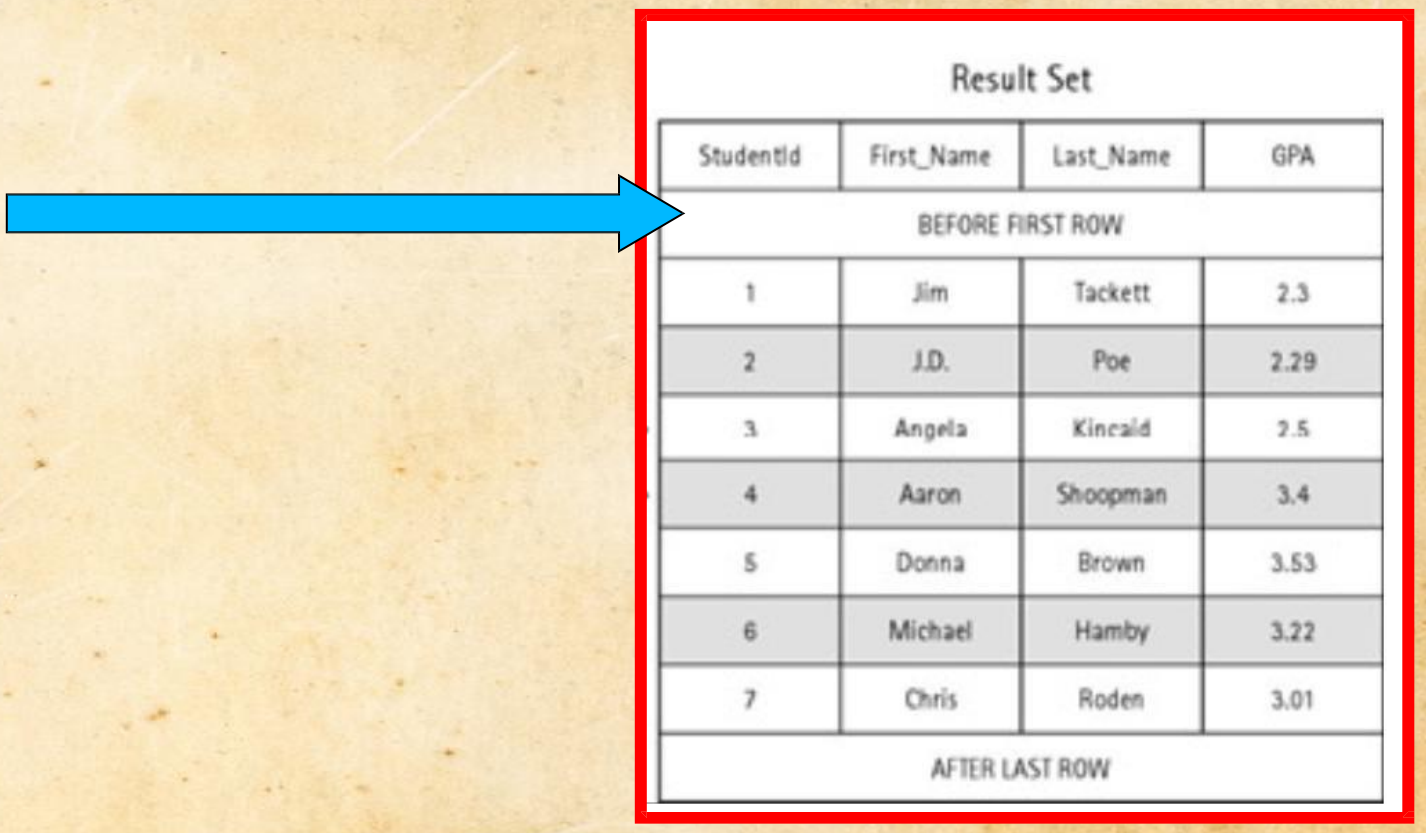


Result Set

StudentId	First_Name	Last_Name	GPA
BEFORE FIRST ROW			
1	Jim	Tackett	2.3
2	J.D.	Poe	2.29
3	Angela	Kincaid	2.5
4	Aaron	Shoopman	3.4
5	Donna	Brown	3.53
6	Michael	Hamby	3.22
7	Chris	Roden	3.01
AFTER LAST ROW			

Mètodes de ResulSet per a recuperació d'informació de fila

- Quan es fa una consulta, el cursor de l'objecte ResultSet **es col·loca ABANS de la primera fila**.



Result Set

StudentId	First_Name	Last_Name	GPA
BEFORE FIRST ROW			
1	Jim	Tackett	2.3
2	J.D.	Poe	2.29
3	Angela	Kincaid	2.5
4	Aaron	Shoopman	3.4
5	Donna	Brown	3.53
6	Michael	Hamby	3.22
7	Chris	Roden	3.01
AFTER LAST ROW			

Exemple ús de ResultSet

```
try (Connection connection = this.dataSource.getConnection()) {
    try (Statement statement = connection.createStatement()) {
        try (ResultSet resultSet = statement.executeQuery(sentenciaSQL)) {

            while (resultSet.next()) {

                int codigo = resultSet.getInt("codigo");
                String nombre = resultSet.getString("nombre");
                int telefono = resultSet.getInt("telefono");
                TipoCliente tipo = TipoCliente.tipoSegunAbreviatura(resultSet.getString("tipo"));
                int numRevisiones = resultSet.getInt("numRevisiones");

                clientes.add(new Cliente(codigo, nombre, telefono, tipo, numRevisiones));
            }
        }
    }
}
```

Maneig de la informació amb ResultSet

- És important tenir en compte que el maneig de la informació que conté un objecte ResultSet serà diferent,depenent de la forma en que us connecteu amb el driver que gestiona la BD.
- Per això, hem d'utilitzar la implementació de `createStatement` que necesita dos paràmetres.

`createStatement(int resultSetType, int resultSetConcurrency)`

Creates a `Statement` object that will generate `ResultSet` objects with the given type and concurrency.

Maneig de lainformació amb ResultSet

- resultSetType indica el tipus de ResulSet que torna:
  - TYPE_FORWARD_ONLY: se crea un objecte ResultSet amb moviment únicament cap avant (forward-only). És el tipus de ResultSet per defecte.
  - TYPE_SCROLL_INSENSITIVE: es crea un objecte ResultSet que permet tot tipus de moviments. Però este tipus de ResultSet, mentre està obert, no serà conscient dels canvis que es realitzen sobre les dades que està mostrant i, per tant, no mostrarà estes modificacions. **(No ho anem a treballar en este curs)**
  - TYPE_SCROLL_SENSITIVE: igual que l'anterior permet tot tipus de moviments i a més, permet vore els canvis que es realitzen sobre les dades que conté. **(No ho anem a treballar en este curs)**

Maneig de lainformació amb ResultSet

- resultSetConcurrency indica si volem que es puguen canviar les dades que conté la BD a través de l'objecte ResultSet.
  - CONCUR_READ_ONLY: indica que el ResultSet és només delectura. És el valor per defecte.
  - CONCUR_UPDATABLE: permet realitzar modificacions sobre les dades que conté el ResultSet **(No ho anem a treballar en este curs)**

Pas 3.2: Inserir informació en la BD

- Per a poder emmagatzemar informació en la BD executem el mètode **`executeUpdate`** de la classe **`Statement`** proporcionant la sentència SQL d'**inserció, modificació o esborrat** que es vulga.
- El mètode retorna el recompte de files per a sentències de llenguatge de manipulació de dades SQL (DML) o 0 per a sentències SQL que no retornen res

```
String sentenciaSQL = "INSERT INTO " + NOMBRE_TABLA + " VALUES("
        + cliente.getCodigo() + ", '"
        + cliente.getNombre() + "', "
        + cliente.getTelefono() + ", '"
        + cliente.getTipo().getAbreviatura() + "', "
        + cliente.getNumRevisionesRealizadas() + ")";

try (Connection connection = BDUtil.getDataSource().getConnection()) {
    try (Statement stateMent = connection.createStatement()) {
        stateMent.executeUpdate(sentenciaSQL);
    }
}
```

SQL Injection

SQL Injection és una tècnica hacking molt coneguda que consisteix a inserir codi SQL en una consulta mitjançant l'entrada de dades. Per exemple amb este codi:

```
String nom;
System.out.println("Introdueix el nom de l'usuari a eliminar: ");
nom = new Scanner(System.in).nextLine();
String sql = "DELETE FROM usuaris WHERE nom = '" + nom + "'";
```

Si el nostre client té coneixements SQL i introdueix codi SQL com si fora el nom de l'usuari, este codi s'afegirà a la sentència, i pot eliminar per exemple tots els usuaris de la nostra base de dades:

```
run:
Introdueix el nom de l'usuari a eliminar:
xxx' OR '1' = '1
```



```
DELETE FROM usuaris
WHERE nom ='xxx' OR '1' = '1'
```

Sentències preparades o parametritzades

Com hem vist, per a evitar SQL Injection no és bona idea construir sentències com a concatenació de cadenes a partir de dades introduïdes per l'usuari.

En comptes d'això usarem sentències parametritzades, que són aquelles que inclouen uns marcadors o paràmetres que se substitueixen per valors. Este mecanisme permet adaptar i reutilitzar la mateixa consulta diverses voltes. En JDBC la interfície `PreparedStatement` representa una consulta parametritzada.

Per a reutilitzar la consulta és tan senzill com assignar nous valors als paràmetres.

IMPORTANT: Els paràmetres comencen des de l'1

Sentències preparades o parametritzades

```
String curs = "1DAW";
double notaDeTall = 6.43;

// Consulta amb paràmetres
String sql = "SELECT nom, mitjana FROM alumnes " +
             "WHERE curs = ? AND " +
             "mitjana > ?";

// Creem un objecte PreparedStatement:
PreparedStatement sentencia = con.preparedStatement(sql);
// Assignem els paràmetres
sentencia.setString(1, curs); // La primera "?" correspon a la dada "curs"
sentencia.setDouble(2, notaDeTall); // La segona "?" a la dada "notaDeTall"
ResultSet rs = sentencia.executeQuery(); // Executem la consulta
// ... Treballem amb les dades obtingudes ...
```

Sentències preparades o parametritzades

Per assignar paràmetres disposem dels mètodes:

- `void setString(int indexParametre, String valor)`
- `void setInt(int indexParametre, int valor)`
- `void setDouble(int indexParametre, double valor)`
- `void setBoolean(int indexParametre, boolean valor)`
- `void setDate(int indexParametre, Date valor)`

A voltes ens interessa guardar un valor nul en la base de dades:

- `void setNull(int indexParametre, int tipusSQL)`

tipusSQL és el tipus que correspon a la columna de la BD, i està definit com a enumerat en java.sql.Types amb els valors INTEGER, BOOLEAN, VARCHAR, DECIMAL...

Sentències preparades o parametritzades

A partir d'ara es recomana treballar amb sentències SQL "preparades"

```java
public int actualizar(Categoria c) throws SQLException {
    String sentenciaSQL = "UPDATE " + tabla + " SET nombre=? WHERE codigo=?";
    PreparedStatement prepared = getPrepared(sentenciaSQL);
    prepared.setString(1, c.getNombre());
    prepared.setInt(2, c.getCodigo());
    return prepared.executeUpdate();

}
```

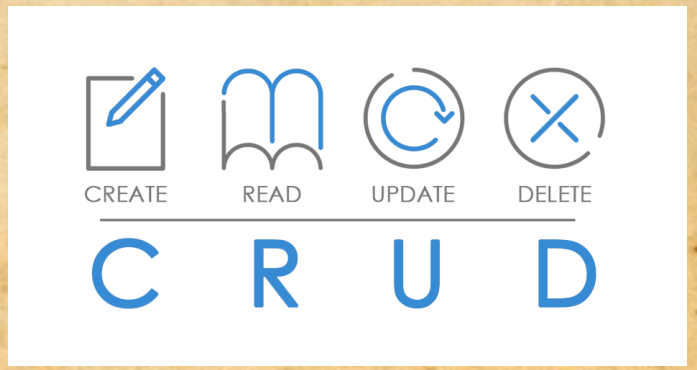
```java
public PreparedStatement getPrepared(String sql) throws SQLException {
    try {
        Class.forName("oracle.jdbc.OracleDriver");
        try (Connection conn = datasource.getConnection()) {
            return conn.prepareStatement(sql);
        }
    } catch (ClassNotFoundException ex) {
        Logger.getLogger(ConexionDAO.class.getName()).log(Level.SEVERE, null, ex);
    }
    return null;
}
```

Recorda que és important revisar l'API per a conèixer més:
https://docs.oracle.com/en/java/javase/21/docs/api/java.sql/java/sql/PreparedStatement.html

Operacions CRUD

CRUD són les sigles en anglés de "crear, llegir, actualitzar i esborrar", i s'usa per a referir-se a les operacions bàsiques que es realitzen en una base de dades.

Fins ara hem treballat amb dades aïllades com a simples variables, per a conéixer una mica millor la API de JDBC, però esta manera de treballar no és la més habitual. Ara manejarem classes i objectes, als quals incorporarem operacions CRUD que s'encarregaran de gestionar l'objecte en la base de dades.

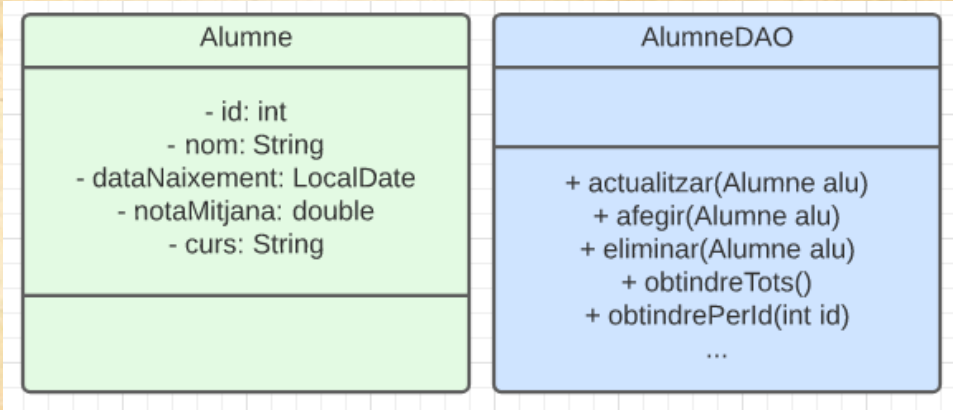


Objectes d'accés a dades

El fet d'haver d'afegir mètodes relacionats amb les operacions de la BD a una classe com pot ser Alumne és una distorsió de l'abstracció que es fa de la realitat (una operació CRUD no pertany al món d'un alumne).

Per això el que farem és treballar sobre dues classes. La primera d'elles coneguda com a DTO (Data Transfer Object) ens permet emmagatzemar la informació de les nostres entitats, i ens ajudara a transmetre tota eixa informació. La segona coneguda com a DAO (Data Access Object) proporcionarà els mètodes necessaris per a inserir, actualitzar, esborrar i consultar la informació de la BD.

Data Transfer Object



Data Access Object

Tecnologies actuals (extra)

Tecnología	Descripción
JDBC Moderno	Se sigue usando JDBC, pero ahora con mejoras del lenguaje como *try-with-resources* para el cierre seguro de recursos y el uso de conexiones pool para optimizar el rendimiento.
JPA y Frameworks ORM	La Java Persistence API (JPA) permite mapear entidades Java a tablas de bases de datos (ORM). Implementaciones como Hibernate o EclipseLink automatizan la generación de SQL y el manejo de transacciones, facilitando el desarrollo de aplicaciones complejas.
Spring Data y Spring JDBC Template	Spring Framework ofrece plantillas que abstraen la complejidad de JDBC, ofreciendo una manera más simple y configurable de ejecutar operaciones en bases de datos, ya sea a través de JDBC Template o integraciones con Spring Data JPA.
MyBatis	MyBatis equilibra el uso de SQL escrito de forma explícita y el mapeo automático entre los resultados de las consultas y objetos Java, brindando mayor control a los desarrolladores que requieren optimizar consultas específicas.
jOOQ y QueryDSL	Estas bibliotecas permiten construir consultas SQL de forma fluida y segura en tiempo de compilación, ofreciendo una sintaxis tipo DSL que facilita la creación de consultas complejas sin renunciar a la potencia del SQL tradicional.
JDBI	JDBI es una biblioteca que simplifica el acceso a bases de datos relacionales sobre JDBC, proporcionando una API más fluida y segura para mapear resultados a objetos Java, sin ocultar el SQL. Ideal para proyectos que buscan mejorar la legibilidad y reducir el código repetitivo.
R2DBC	R2DBC es una especificación para acceso reactiva y no bloqueante a bases de datos relacionales. Proporciona una API asíncrona que permite ejecutar operaciones de base de datos sin bloquear hilos, mejorando la escalabilidad y el rendimiento en aplicaciones reactivas.

JPA (Java Persistence API)

És una especificació de Java per a la **persistència d'objectes en bases de dades relacionals**. Permet mapar classes Java a taules SQL i gestionar operacions com a **insercions, actualitzacions i consultes** de manera eficient. (Es veurà a la UT13)

- **Àmbits d'ús:**

- ✔ Aplicacions empresarials amb bases de dades SQL.
- ✔ Sistemes que necessiten un ORM per a simplificar l'accés a dades.
- ✔ Integració amb **Spring Boot, Jakarta EE i Hibernate**.

JPA (Java Persistence API)

```java
// 1. Definir una entitat JPA
@Entity
@Table(name = "usuaris")
public class Usuari {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nom;

    // Getters i setters
}
```

```java
// 2. Repositori JPA
@Repository
public interface UsuariRepository extends JpaRepository<Usuari, Long> {
    List<Usuari> findByNom(String nom);
}
```

```java
// 3. Ús del repositori en un servei
@Service
public class UsuariService {
    private final UsuariRepository usuariRepository;

    public UsuariService(UsuariRepository usuariRepository) {
        this.usuariRepository = usuariRepository;
    }

    public Usuari guardarUsuari(String nom) {
        return usuariRepository.save(new Usuari(nom));
    }

    public List<Usuari> trobarPerNom(String nom) {
        return usuariRepository.findByNom(nom);
    }
}
```

Spring Data JDBC

Spring Data JDBC és una solució intermèdia entre **JPA** i **JDBC pur**. Ofereix una manera simple i eficient de connectar aplicacions Java amb bases de dades **sense la complexitat d'un ORM**.

- **Àmbits d'ús:**

✔ Aplicacions amb **models de dades senzills** i sense necessitat de gestió d'estats complexos.
✔ Quan es vol una **alternativa lleugera a JPA** amb consultes SQL previsibles.
✔ Ideal per a **microserveis** o sistemes on el rendiment és crític.

Spring Data JDBC

A nivell de codi, **Spring Data JDBC i JPA** són molt semblants en l'ús bàsic, però la gran diferència està en el **model de persistència**.

🔹 Característica	🟢 Spring Data JDBC	🔴 JPA (Hibernate)
Gestió d'Entitats	Entitats són simples POJOs sense proxies ni gestió d'estats.	Hibernate gestiona l'estat i la persistència automàticament.
Relacions	No suporta relacions complexes (OneToMany, ManyToOne requereixen SQL manual).	Gestió avançada d'entitats relacionades amb @OneToMany, @ManyToOne, etc.
Rendiment	Lleuger, ràpid i previsible (consulta SQL clara).	Pot generar SQL complex i consumir més recursos.
Transaccions	No té un context de persistència, s'ha de fer manualment si cal.	Hibernate manté un **context d'entitat**, fent la gestió d'estats automàticament.

MyBatis

MyBatis és un framework de persistència per a Java que facilita la interacció amb bases de dades **usant SQL explícit**. A diferència de JPA, **no és un ORM**, sinó un mapper SQL que permet un control total sobre les consultes.

- **Àmbits d'ús:**

- ✔ Aplicacions que necessiten **control precís sobre les consultes SQL**.
- ✔ Quan es vol evitar la **sobrecàrrega d'un ORM** com Hibernate.
- ✔ Projectes amb **SQL complex** o procediments emmagatzemats.

MyBatis

```java
// 1. Definir una entitat
public class Usuari {
    private Long id;
    private String nom;

    public Usuari(Long id, String nom) {
        this.id = id;
        this.nom = nom;
    }
    // Getters i setters...
}
```

```java
// 2. Crear un mapper amb anotacions (també es pot fer amb XML)

@Mapper
public interface UsuariMapper {

    @Select("SELECT * FROM usuaris WHERE id = #{id}")
    Usuari trobarPerId(Long id);

    @Insert("INSERT INTO usuaris (nom) VALUES (#{nom})")
    @Options(useGeneratedKeys = true, keyProperty = "id")
    void guardarUsuari(Usuari usuari);

    @Select("SELECT * FROM usuaris WHERE nom = #{nom}")
    List<Usuari> trobarPerNom(String nom);
}
```

```java
// 3. Ús del mapper en un servei

@Service
public class UsuariService {
    private final UsuariMapper usuariMapper;

    public UsuariService(UsuariMapper usuariMapper) {
        this.usuariMapper = usuariMapper;
    }

    public Usuari guardarUsuari(String nom) {
        Usuari usuari = new Usuari(null, nom);
        usuariMapper.guardarUsuari(usuari);
        return usuari;
    }

    public List<Usuari> trobarPerNom(String nom) {
        return usuariMapper.trobarPerNom(nom);
    }
}
```

jOOQ (Java Object Oriented Querying)

És una llibreria que permet construir i executar **consultes SQL de manera typesafe** i programàtica en Java. No és un ORM, sinó una API que **genera codi Java a partir de l'esquema de la base de dades** oferint un DSL per a fer-ho fàcilment.

- **Àmbits d'ús:**

- ✔ Quan es vol **escriure SQL directament** però amb seguretat de tipus.
- ✔ Aplicacions que necessiten **alta eficiència en consultes complexes**.
- ✔ Ideal per a bases de dades amb **consultes dinàmiques i optimitzades**.

jOOQ

```java
// 1. Configurar jOOQ i establir la connexió
DSLContext dsl = DSL.using(
    "jdbc:postgresql://localhost:5432/mi_db",
    "usuari",
    "contrasenya"
);
```

```java
// 2. Inserir un usuari
dsl.insertInto(USUARIS)
   .columns(USUARIS.NOM)
   .values("Joan")
   .execute();
```

```java
// 3. Consultar usuaris per nom
Result<Record> result = dsl.select()
    .from(USUARIS)
    .where(USUARIS.NOM.eq("Joan"))
    .fetch();

for (Record r : result) {
    Long id = r.get(USUARIS.ID);
    String nom = r.get(USUARIS.NOM);
    System.out.println("Usuari: " + id + " - " + nom);
}
```

```java
// 4. Actualitzar nom d'usuari per ID
dsl.update(USUARIS)
   .set(USUARIS.NOM, "Joan Actualitzat")
   .where(USUARIS.ID.eq(1L))
   .execute();
```

```java
// 5. Avançada: Usuaris amb comandes asociades
Result<Record> result = dsl.select(USUARIS.NOM, DSL.count(COMANDES.ID).as("total_comandes"))
    .from(USUARIS)
    .leftJoin(COMANDES)
    .on(USUARIS.ID.eq(COMANDES.USUARI_ID))
    .groupBy(USUARIS.NOM)
    .having(DSL.count(COMANDES.ID).greaterThan(2))
    .orderBy(DSL.count(COMANDES.ID).desc())
    .fetch();

for (Record r : result) {
    String nom = r.get(USUARIS.NOM);
    int totalComandes = r.get("total_comandes", int.class);
    System.out.println(nom + " té " + totalComandes + " comandes.");
}
```

JDBI (Java Database Interface)

JDBI és una llibreria que facilita la interacció amb bases de dades a **Java**. És una alternativa lleugera a **JDBC pur**, amb una API simple per a **executar consultes SQL** i mapar resultats a **objectes Java**.

- **Àmbits d'ús:**

✔ Aplicacions que volen evitar la **verbositat de JDBC** sense utilitzar un ORM.
✔ Ideal per a **consultes SQL complexes** amb un **mapeig directe de resultats** a objectes.
✔ Quan es vol **màxim control** sobre les consultes SQL sense la sobrecàrrega d'un ORM com JPA.

JDBI

```java
// 1. Definir una entitat
public class Usuari {
    private Long id;
    private String nom;

    public Usuari(Long id, String nom) {
        this.id = id;
        this.nom = nom;
    }
    // Getters i setters...
}
```

```java
// 2. Crear un DAO (Data Access Object)
public interface UsuariDAO {
    @SqlUpdate("INSERT INTO usuaris (nom) VALUES (:nom)")
    @GetGeneratedKeys
    Long guardarUsuari(@Bind("nom") String nom);

    @SqlQuery("SELECT * FROM usuaris WHERE id = :id")
    Usuari trobarPerId(@Bind("id") Long id);

    @SqlQuery("SELECT * FROM usuaris WHERE nom = :nom")
    List<Usuari> trobarPerNom(@Bind("nom") String nom);
}
```

```java
// 3. Configurar JDBI i executar consultes
public class UsuariService {
    private final Jdbi jdbi;

    public UsuariService() {
        this.jdbi = Jdbi.create("jdbc:postgresql://localhost:5432/mi_db", "usuari", "contrasenya");
    }

    public Long guardarUsuari(String nom) {
        return jdbi.withExtension(UsuariDAO.class, dao -> dao.guardarUsuari(nom));
    }

    public List<Usuari> trobarPerNom(String nom) {
        return jdbi.withExtension(UsuariDAO.class, dao -> dao.trobarPerNom(nom));
    }
}
```

```java
// 4. Exemple amb consulta avançada
public List<Usuari> consultarUsuarisAmbComandes() {
    String sql = "SELECT usuaris.id, usuaris.nom, COUNT(comandes.id) AS total_comandes "
               + "FROM usuaris "
               + "LEFT JOIN comandes ON usuaris.id = comandes.usuaris_id "
               + "GROUP BY usuaris.id "
               + "HAVING COUNT(comandes.id) > :minComandes";

    return jdbi.withHandle(handle ->
        handle.createQuery(sql)
              .bind("minComandes", 2)
              .mapToBean(Usuari.class)
              .list()
    );
}
```

R2DBC (Reactive Relational Database Connectivity)

R2DBC és una API reactiva per a la connexió amb bases de dades relacionals en Java, dissenyada per treballar amb un model **no bloquejant** i **basat en reactivitat** (Project Reactor). Ofereix un mètode més eficient per treballar amb bases de dades en entorns reactius. I és l'única opció realment asíncrona.

- **Àmbits d'ús:**

✔ Aplicacions **reactives** que necessiten treballar amb bases de dades relacionals.
✔ Quan es vol **manejament eficient de la concurrència** i **optimització en sistemes no bloquejants**.
✔ Ideal per a **microserveis** amb arquitectura reactiva.

R2DBC

```java
// 1. Configurar connexió amb R2DBC
OracleConnectionFactory connectionFactory = new OracleConnectionFactory(
    OracleConnectionConfiguration.builder()
        .host("localhost")
        .port(1521)
        .database("mi_db")
        .username("usuari")
        .password("contrasenya")
        .build()
);
```

```java
// 2. Executar una consulta simple
Mono<Connection> connectionMono = Mono.from(connectionFactory.create());

connectionMono.flatMapMany(conn ->
    Flux.from(conn.createStatement("SELECT * FROM usuaris").execute())
        .flatMap(result -> result.map((row, meta) -> row.get("nom", String.class)))
        .doFinally(signalType -> conn.close())
).subscribe(System.out::println);
```

```java
// 3. Inserir usuari en la BD
Mono<Void> insert = Mono.from(connectionFactory.create())
    .flatMap(conn ->
        Mono.from(conn.createStatement("INSERT INTO usuaris (nom) VALUES ('Joan')")
            .execute())
        .doFinally(signalType -> conn.close())
    );
insert.subscribe();
```

```java
// 4. Consulta avançada amb R2DBC: Usuaris amb comandes
Mono<Connection> connectionMono = Mono.from(connectionFactory.create());

connectionMono.flatMapMany(conn ->
    Flux.from(conn.createStatement(
            "SELECT usuaris.nom, COUNT(comandes.id) AS total_comandes " +
            "FROM usuaris " +
            "LEFT JOIN comandes ON usuaris.id = comandes.usuaris_id " +
            "GROUP BY usuaris.nom " +
            "HAVING COUNT(comandes.id) > 2"
        ).execute())
        .flatMap(result -> result.map((row, meta) ->
            row.get("nom", String.class) + " té "
            + row.get("total_comandes", Integer.class)
            + " comandes"))
        .doFinally(signalType -> conn.close())
).subscribe(System.out::println);
```