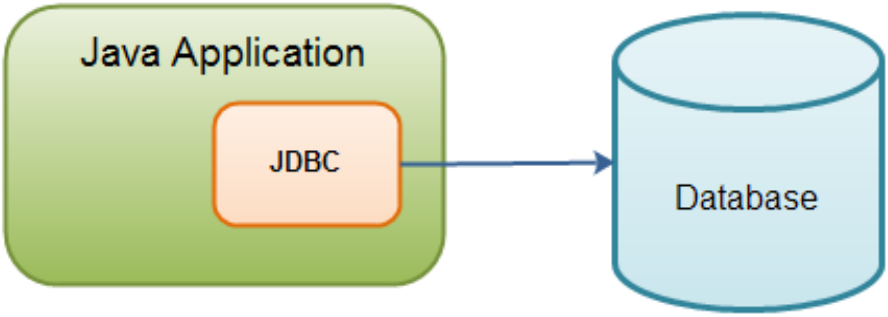
U.T. 7 - PROGRAMACIÓ AMB ACCÉS A BASES DE DADES RELACIONALS



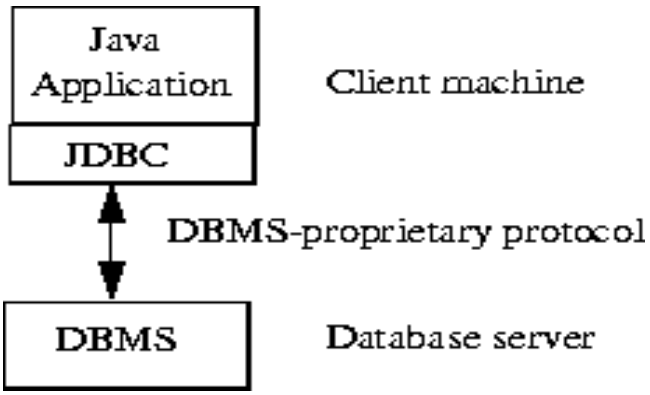
1. INTRODUCCIÓ A JDBC

JDBC (Java Data Base Connectivity) -> API per a:

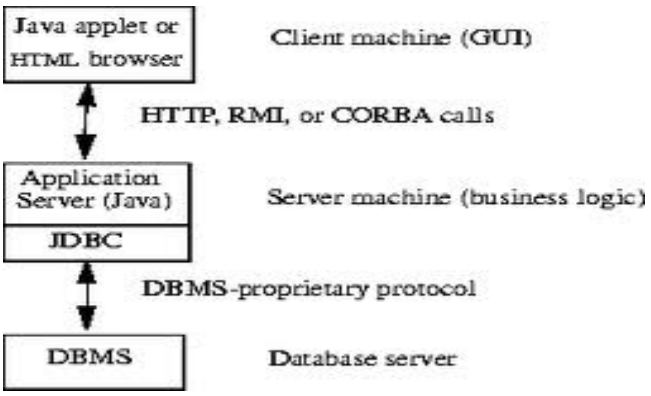
- establir una connexió amb una base de dades
- executar sentències SQL
- processar els resultats de l'execució de les sentències

2 models d'arquitectura JDBC:

- en 2 capes:



- en 3 capes: el client interactua des d'un navegador i l'aplicació es troba en un servidor intermedi, a l'altre extrem està el servidor de base de dades.



Primer pas: establir connexió -> es farà amb la classe **DriverManager**, veure-la en l'API i buscar el mètode *getConnection*..., ¿però per a què s'utilitza aquesta classe? La seva funció primordial és la de seleccionar el controlador adequat per connectar l'aplicació o applet amb una base de dades determinada. Quins són els principals drivers JDBC?

1. driver pont sobre ODBC (una altra API programada en C creada per Microsoft). No és la millor opció, especialment en entorns UNIX-Linux, per motius de rendiment i incompatibilitats entre els llenguatges Java i C, referents a l'ús dels punters. A més, no es poden utilitzar des de miniaplicacions.
2. drivers purs Java: tradueixen les crides JDBC al protocol usat pel SGBD. És el que utilitzarem, inclou el corresponent per a MySQL (o MariaDB que ve a ser el mateix).

JDBC inclou un conjunt de classes i interfícies, que podem agrupar atenent a les funcions que compleixen. Les principals poden ser les següents.

Per establir una connexió a un origen de dades:

- la ja esmentada DriverManager
- Driver
- Connection

Execució de sentències SQL:

- *Statement*
- *PreparedStatement*
- *CallableStatement*

Obtenció i modificació dels resultats d'una sentència SQL:

- *ResultSet*

Per al mapeig de dades SQL a tipus de dades del llenguatge Java:

- Array
- Blob
- Ref
- Struct
- Time ...

Mapeig de tipus SQL definits per l'usuari a classes Java:

- SQLData
- SQLInput
- SQLOutput

Oferir informació sobre la base de dades i sobre les columnes d'un objecte *ResultSet*:

- DatabaseMetaData
- ResultSetMetaData

JDBC ofereix simplement una especificació comuna per a l'accés a bases de dades; el fabricant d'un driver per a un SGBD concret haurà de seguir les recomanacions d'aquesta especificació si vol construir un driver per JDBC. Tot driver JDBC hauria d'implementar la interfície *Driver*.

La classe DriverManager manté una llista de classes *Driver* que s'han registrat elles mateixes cridant al mètode *registerDriver()* de la classe *DriverManager*. Totes les classes *Driver* haurien d'estar escrites amb una secció estàtica que creés una instància de la classe i tot seguit es registrara amb el mètode estàtic *DriverManager.registerDriver()*; aquest procés s'executa automàticament quan es carrega la classe del driver.

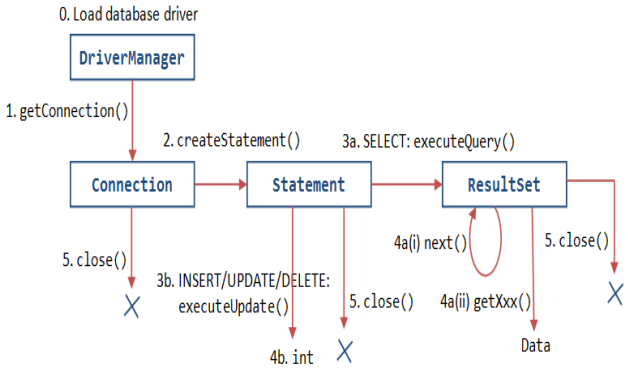
Per tant, gràcies al mecanisme anterior, un programador normalment no hauria de cridar directament al mètode *DriverManager.registerDriver()*, sinó que serà cridat automàticament pel driver quan és carregat. La manera aconsellada de carregar una classe *Driver*, i per tant registrar amb el *DriverManager*, és trucar al mètode estàtic *forName()* de la classe *Class*:

```
String driver = "com.mysql.jdbc.Driver";
Class.forName(driver).newInstance();
```

Però, amb les últimes versions dels drivers, inclús la càrrega del controlador es fa automàticament i no és necessari. Per tant, els nostres programes no requeriran les 2 línies anteriors.

Una vegada que les classes dels controladors s'han carregat i registrat, es pot establir la connexió amb la base de dades. Quan es realitza una petició de connexió amb una crida al mètode *DriverManager.getConnection()*, la classe *DriverManager* revisa cadascun dels drivers disponibles per a comprovar si pot establir la connexió.

Un objecte **Connection** representa una connexió amb una base de dades. Una sessió de connexió amb una base de dades inclou les sentències SQL que s'executaran i els resultats retornats.



La mateixa aplicació pot obrir més d'una connexió amb la mateixa base de dades, o amb diferents.

No obstant això, *Connection* no és una classe, sinó una interfície. ¿Com és que s'instància? La interfície és una plantilla o especificació implementada en la pràctica pel controlador concret; en el nostre cas, de MySQL. Per exemple:

```
String jdbcUrl = "jdbc:mysql://localhost:3306/empresa";
Connection connexio = DriverManager.getConnection(jdbcUrl,"root","");
```

Per a veure els primers exemples primer crearem una base de dades anomenada "empresa". Utilitzarem els scripts "script_mysql_empresa_estructura.sql" i "script_mysql_empresa_procedures.sql".

Des de /opt/lampp/bin (per a XAMPP):

```
mysql -u root empresa < script_mysql_empresa_procedures.sql
```

També es pot fet de manera més simple amb l'interfície gràfica de PhpMyAdmin. També si fem ús de *Docker*, creant un contenidor de *MariaDB* i altre del propi *PhpMyAdmin*. D'una formao una altra, comprova que s'ha creat tot correctament, taules, vistes i procediments.

2. LA INTERFÍCIE STATEMENT

En l'apartat anterior vèiem com establir una connexió amb una base de dades. Ara anem a fer un pas més en l'accés a bases de dades. Una vegada establerta la connexió, podrem enviar sentències SQL; per això comptem amb la interfície *Statement*.

Un objecte **Statement** és utilitzat per enviar sentències SQL a una base de dades. Hi han tres tipus d'objectes *Statement*. Aquests tipus són:

- *Statement*
- *PreparedStatement*
- *CallableStatement*.

PreparedStatement hereta de la interfície *Statement* i *CallableStatement* ho fa de *PreparedStatement*.

Un objecte *Statement* és utilitzat per a executar una sentència SQL simple sense paràmetres; un objecte *PreparedStatement* és utilitzat per executar sentències SQL precompilades amb o sense paràmetres d'entrada, se sol utilitzar també per executar sentències SQL d'ús freqüent; finalment, un objecte *CallableStatement* és utilitzat per executar una crida a un procediment emmagatzemat d'una base de dades, que pot tenir paràmetres d'entrada, d'eixida i entrada/eixida.

Un objecte *Statement* es crea amb el mètode *createStatement()* de la classe *Connection*, com es pot observar en el següent fragment de codi:

```
connexio = DriverManager.getConnection(jdbcUrl,"root","");
stmt = conexion.createStatement();
```

Observa com, de moment, el mètode *createStatement()* s'està cridant sense paràmetres, fet que suposarà algunes limitacions que veurem més endavant.

L'objecte *Statement* proporciona bàsicament 3 mètodes: *execute()*, *executeUpdate()* i *executeQuery()* , que actuen com a conductors d'informació amb la base de dades. La diferència entre cada un d'aquests mètodes la mostrem a continuació:

-*executeQuery()* : S'utilitza amb sentències SELECT (consultes o *querys*) i retorna un *ResultSet*.

-*executeUpdate()* : S'utilitza amb sentències INSERT, UPDATE i DELETE, o bé, amb sentències DDL SQL, com la mateixa creació d'una nova taula.

-*execute()* : Utilitzat per a qualsevol sentència DDL (de definició de dades com CREATE, per modificar o crear l'estructura de les taules), DML (de manipulació de dades com INSERT, UPDATE, DELETE o SELECT), o també comandaments específics de la base de dades.

Vegem un exemple de mètode *executeQuery()*; com hem vist anteriorment aquest mètode retorna un *ResultSet*. Una vegada s'ha realitzat la consulta els valors els emmagatzemarem en un *ResultSet* per a la seva posterior consulta:

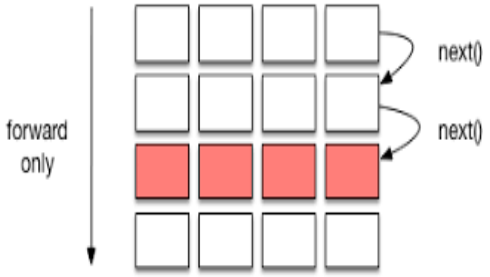
```
String jdbcUrl = "jdbc:mysql://localhost:3306/empresa";
Connection conn = DriverManager.getConnection(jdbcUrl, "root","");
/* Ús del mètode executeQuery
 Observa com createStatement NO té paràmetres: això limitarà el tipus de ResultSet */
Statement stmt = conn.createStatement();
```

```
String sql = "select * from articles";
ResultSet rs = stmt.executeQuery(sql);
```

EL mètode *createStatement()* de la interfície *Connection* es troba sobrecarregat. Si utilitzem la seva versió sense paràmetres, a l'hora d'executar sentències SQL sobre l'objecte *Statement* s'obtindrà el tipus d'objecte *ResultSet* per defecte, és a dir, s'obtindria un tipus de cursor de només lectura i amb moviment únicament cap endavant. Però l'altra versió del mètode *createStatement()* ofereix dos paràmetres que ens permeten definir el tipus de *ResultSet* que es retornarà com a resultat de l'execució de la consulta, com es mostra en el següent codi:

```
String jdbcUrl = "jdbc:mysql://localhost:3306/empresa";
Connection conn = DriverManager.getConnection(jdbcUrl,"root","");
// Ús de l'mètode executeQuery
stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                ResultSet.CONCUR_READ_ONLY);
String sql = "select * from articles";
rs = stmt.executeQuery(sql);
```

El primer dels paràmetres indica el tipus d'objecte *ResultSet* que es crearà, i el segon d'ells indica si el *ResultSet* és només de lectura o si permet modificacions; aquest paràmetre també es denomina tipus de concurrència. Si no indiquem cap paràmetre en el mètode *createStatement()* , es crearà un objecte *ResultSet* amb els valors per defecte.

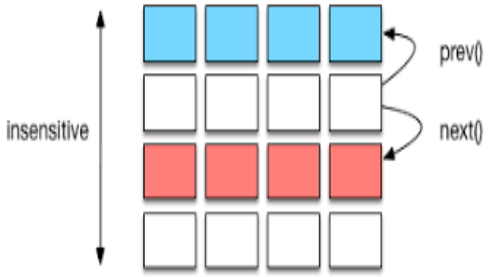


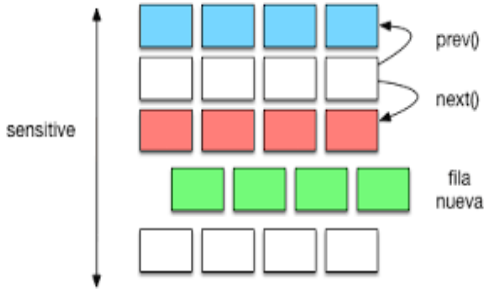
Els tipus de *ResultSet* diferents que es poden crear depenen de la valor del primer paràmetre. Aquests valors es corresponen amb constants definides en la interfície *ResultSet*. Aquestes constants es descriuen a continuació:

TYPE_FORWARD_ONLY: es crea un objecte *ResultSet* amb moviment únicament cap endavant (forward-only). És el tipus de *ResultSet* per defecte.

TYPE_SCROLL_INSENSITIVE: es crea un objecte *ResultSet* que permet tota mena de moviments. Però aquest 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 aquestes modificacions.

TYPE_SCROLL_SENSITIVE: igualment que l'anterior permet tot tipus de moviments, però permet veure els canvis que es realitzen sobre les dades que conté.





Els valors que pot tenir el segon paràmetre que defineix la creació d'un objecte *ResultSet*, són també constants definides en la interfície *ResultSet* i són els següents:

CONCUR_READ_ONLY: indica que el *ResultSet* és només de lectura. És el valor per defecte.

CONCUR_UPDATABLE: permet realitzar modificacions sobre les dades que conté el *ResultSet*.

3. TREBALLANT AMB ResultSets

En un objecte *ResultSet* es troben els resultats de l'execució d'una sentència SQL. Per tant, un objecte *ResultSet* conté les files que satisfan les condicions d'una sentència SQL, i ofereix l'accés a les dades a través d'una sèrie de mètodes *getXXX()* que permeten accedir a les columnes de la fila actual.

El mètode *next()* de la interfície *ResultSet* és utilitzat per desplaçar-se a la següent fila del *ResultSet*, fent que la propera fila siga l'actual; a més d'aquest mètode de desplaçament bàsic, segons el tipus de *ResultSet* podrem realitzar desplaçaments lliures utilitzant mètodes com *last()*, *relative()* o *previous()* .

L'aspecte que sol tenir un *ResultSet* és una taula amb capçaleres de columnes i els valors corresponents retornats per una consulta. Un *ResultSet* manté un cursor que apunta a la fila actual de dades. El cursor es mou cap avall cada vegada que el mètode *next()* és llançat. Inicialment està posicionat abans de la primera fila; d'aquesta manera, la primera crida a *next()* situarà el cursor a la primera fila, passant a ser la fila actual.

Les fileres del *ResultSet* són retornades de dalt a baix segons es va desplaçant el cursor amb les successives crides al mètode *next()* . Un cursor és vàlid fins que l'objecte *ResultSet*, o el seu objecte pare *Statement*, siga tancat.

Obtenint dades del *ResultSet*: els mètodes getXXX() ofereixen els mitjans per recuperar els valors de les columnes (camps) de la fila (registre) actual del *ResultSet*. No cal que les columnes siguen recuperades utilitzant un ordre determinat, però, per a una major portabilitat entre diferents bases de dades, es recomana que els valors de les columnes es recuperen d'esquerra a dreta i només una vegada.

Per designar una columna podem utilitzar el seu nom o bé el seu número d'ordre. Per exemple, si la segona columna d'un objecte *rs* de la classe *ResultSet* es diu "titol" i emmagatzema dades de tipus String, es podrà recuperar el seu valor de les formes que mostra el següent fragment de codi:

```
// rs és un objecte de tipus ResultSet
String valor = rs.getString(2); // primera forma
String valor = rs.getString("titol");   // segona, completament equivalent
```

S'ha d'assenyalar que les columnes es numeren d'esquerra a dreta començant amb la columna 1, i que els noms de les columnes no són *case sensitive*, és a dir, no distinguixen entre majúscules i minúscules.

La informació referent a les columnes que conté el *ResultSet* es troba disponible cridant al mètode *getMetaData()* ; aquest mètode retornarà un objecte *ResultSetMetaData* que contindrà el nombre, tipus i propietats de les columnes del *ResultSet*.

Si coneixem el nom d'una columna, però no el seu índex, el mètode *findColumn()* pot ser utilitzat per obtenir el nombre de columna, passant-li com a argument un objecte String que siga el nom de la columna corresponent, retornant un enter que serà l'índex corresponent a la columna.

Tipus de dades i conversions: quan es llança un mètode *getXXX()* determinat sobre un objecte *ResultSet* per obtenir el valor d'un camp del registre actual, el controlador JDBC converteix la dada que es vol recuperar al tipus Java especificat. Per exemple, si utilitzem el mètode *getString()* i el tipus en la base de dades és VARCHAR, el controlador JDBC convertirà la dada VARCHAR a un objecte String de Java, per tant el valor de retorn de *getString()* serà un objecte de la classe String.

Aquesta conversió de tipus es pot realitzar gràcies a la classe *java.sql.Types*. En aquesta classe es defineixen el que es denominen tipus de dades JDBC, que es correspon amb els tipus de dades SQL estàndard. Això ens permet abstreure'ns de l'tipus SQL específic de la base de dades amb la qual estem treballant, ja que els tipus JDBC són tipus de dades SQL genèrics. Normalment aquests tipus genèrics ens serviran per a totes les nostres aplicacions JDBC.

La classe *Types* està definida com un conjunt de constants (final static); aquestes constants es fan servir per identificar els tipus SQL. Si el tipus de la dada SQL que tenim a la nostra base de dades és específic d'aquesta base de dades, i no es troba entre les constants definides per la classe *Types*, s'utilitzarà el tipus *Types.OTHER*.

Desplaçament en un ResultSet: El mètode *next()* retorna un valor booleà (tipus boolean de Java), true si el registre següent existeix i false si hem arribat a la fi de l'objecte *ResultSet*, és a dir, no hi ha més registres. Aquest era l'únic mètode que oferia la interfície *ResultSet* en la versió 1.0 de JDBC, però en JDBC 2.0, a més de tenir el mètode *next()* , disposem dels següents mètodes per al desplaçament i moviment dins d'un objecte *ResultSet*.

-*boolean absolute(int registre)*: desplaça el cursor el nombre de registres indicat. Si el valor és negatiu, es posiciona en el número de registre indicat però començant pel final. Aquest mètode retornarà false si ens hem desplaçat després de l'últim registre o abans del primer registre de l'objecte *ResultSet*. Per poder utilitzar aquest mètode, l'objecte *ResultSet* ha de ser de tipus TYPE_SCROLL_SENSITIVE o de tipus TYPE_SCROLL_INSENSITIVE. A un *ResultSet* que és de qualsevol d'aquests dos tipus es diu que és de tipus "Scrollable". Si a aquest mètode li passem un valor zero es llançarà una *SQLException*.

-*void afterLast()*: es desplaça a la fi de l'objecte *ResultSet*, després de l'últim registre. Si el *ResultSet* no posseeix registres, la crida no té cap efecte.

-*void beforeFirst()*: mou el cursor a l'inici de l'objecte *ResultSet*, abans del primer registre. Només es pot utilitzar sobre objectes *ResultSet* de tipus *Scrollable*.

-*boolean first()*: desplaça el cursor a la primera entrada. Retorna *true* si el cursor s'ha desplaçat a un registre vàlid; per contra, ha de retornar false en un altre cas o bé si l'objecte *ResultSet* no conté registres. Com als mètodes anteriors, només es pot utilitzar en objectes *ResultSet* de tipus "Scrollable".

-*boolean last()*: desplaça el cursor a l'últim registre de l'objecte *ResultSet*. Retornarà *true* si el cursor es troba en un registre vàlid, i *false* en un altre cas o si l'objecte *ResultSet* no té registres.

-*void moveToCurrentRow()*: mou el cursor a la posició recordada, normalment el registre actual. Aquest mètode només té sentit quan estem situats dins el *ResultSet* en un registre que s'ha inserit, és a dir, amb objectes *ResultSet* que permeten la modificació, definits mitjançant la constant CONCUR_UPDATABLE.

-*boolean previous()*: desplaça el cursor a l'registre anterior. És el mètode contrari al mètode *next()*. Retornarà true si el cursor es troba en un registre o fila vàlids, i false en cas contrari. Només és vàlid aquest mètode amb objectes *ResultSet* de tipus *Scrollable*, en cas contrari llançarà una excepció SQLException.

-*boolean relative(int registres)*: mou el cursor un nombre relatiu de registres; aquest nombre pot ser positiu o negatiu. Si el nombre és negatiu el cursor es desplaçarà cap al principi de l'objecte *ResultSet* el nombre de registres indicats, i si és positiu es desplaçarà cap al final. Aquest mètode només es pot utilitzar si el *ResultSet* és de tipus *Scrollable*.

També hi han altres mètodes dins de la interfície *ResultSet* que estan relacionats amb el desplaçament:

- boolean *isAfterLast()*: indica si ens trobem després de l'últim registre de l'objecte *ResultSet*. Només es pot utilitzar en objectes *ResultSet* de tipus *Scrollable*.
- boolean *isBeforeFirst()*: indica si ens trobem abans del primer registre de l'objecte *ResultSet*. Només es pot utilitzar en objectes *ResultSet* de tipus *Scrollable*.
- boolean *isFirst()*: indica si el cursor es troba en el primer registre. Només es pot utilitzar en objectes *ResultSet* de tipus *Scrollable*.
- boolean *isLast()*: indica si ens trobem en l'últim registre del *ResultSet*. Només es pot utilitzar en objectes *ResultSet* de tipus *Scrollable*.
- int *getRow()*: retorna el número de registre actual. El primer registre serà el número 1, el segon el 2, etc. Retornarà zero si no hi ha registre actual.

Si volem recórrer un *ResultSet* complet cap endavant ho podrem fer a través d'un bucle en què es llança el mètode *next()* sobre l'objecte *ResultSet*. L'execució d'aquest bucle finalitzarà quan arribem al final del conjunt de registres.

4. USANT ResultSet ACTUALITZABLES

Modificant els ResultSet. Mètodes *updateXXX()*: per poder modificar les dades que conté un *ResultSet* hem de crear un *ResultSet* de tipus modificable; per a això hem d'utilitzar la constant *ResultSet.CONCUR_UPDATABLE* dins el mètode *createStatement()*.

Encara que un *ResultSet* que permet modificacions sol permetre diferents desplaçaments, és a dir, se sol utilitzar la constant *ResultSet.TYPE_SCROLL_INSENSITIVE* o *ResultSet.TYPE_SCROLL_SENSITIVE*, però no és del tot necessari ja que també pot ser del tipus només cap endavant (forward-only).

Per modificar els valors d'un registre existent s'utilitzen una sèrie de mètodes *updateXXX()* de la interfície *ResultSet*. Les XXX indiquen el tipus de la dada a l'igual que passa amb els mètodes *getXXX()* d'aquesta mateixa interfície. El procés per a realitzar la modificació d'una fila d'un *ResultSet* és el següent: ens situem sobre el registre que volem modificar i cridem als mètodes *updateXXX()* adequats, passant-li com a argument els nous valors. A continuació cridem al mètode *updateRow()* perquè els canvis tinguen efecte sobre la base de dades.

El mètode *updateXXX()* rep dos paràmetres, el camp o columna a modificar i el nou valor. La columna la podem indicar pel seu número d'ordre o bé pel seu nom, igual que en els mètodes *getXXX()* .

El següent fragment de codi mostra com es pot modificar el camp adreça de l'últim registre d'un *ResultSet* que conté el resultat d'una SELECT sobre la taula de clients:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                      ResultSet.CONCUR_UPDATABLE);
// Executem la SELECT sobre la taula clients
String sql = "select * from clients";
rs = stmt.executeQuery(sql);
System.out.println( "Situem el cursor al final");
// Ens situem en l'últim registre del ResultSet i fem la modificació
rs.last();
```

```
rs.updateString("adreça","c/ Pepe Ciges, 3");
rs.updateRow();
```

Si ens desplacem dins del *ResultSet* abans de llançar el mètode *updateRow()*, es perdran les modificacions realitzades. I si volem cancel·lar les modificacions llançarem el mètode *cancelRowUpdates()* sobre l'objecte *ResultSet*, en lloc del mètode *updateRow()* .

Una vegada que hem invocat el mètode *updateRow()*, el mètode *cancelRowUpdates()* no tindrà cap efecte. El mètode *cancelRowUpdates()* cancel·la les modificacions de tots els camps d'un registre, és a dir, si hem modificat dos camps amb el mètode *updateXXX()* s'han de cancel·lar les dues modificacions.

Mètodes *insertRow()* i *deleteRow()*: a més de poder realitzar modificacions directament sobre les files d'un *ResultSet*, també podem afegir noves files (registres) i eliminar les existents. Aquests mètodes són: *moveToInsertRow()* i *deleteRow()*. Per a inserir una nova fila el procés és:

1. El primer pas per inserir un registre o fila en un *ResultSet* és moure el cursor (punter que indica el registre actual) del *ResultSet*, això s'aconsegueix mitjançant el mètode *moveToInsertRow()*.
2. El següent pas és donar un valor a cada un dels camps que formaran part del nou registre, per a això s'utilitzen els mètodes *updateXXX()* adequats.
3. Per finalitzar el procés es llança el mètode *insertRow()* , que crearà el nou registre tant en el *ResultSet* com a la taula de la base de dades corresponents. Fins que no es llança el mètode *insertRow()* , la fila no s'inclou dins el *ResultSet*; és una fila especial denominada "fila d'inserció" (insert row) i és similar a un buffer completament independent de l'objecte *ResultSet*.
4. Quan hem inserit el nostre nou registre en l'objecte *ResultSet*, podrem tornar a l'antiga posició en què ens trobàvem dins el *ResultSet*, abans d'haver llançat el mètode *moveToInsertRow()*, cridant al mètode *moveToCurrentRow()*. Aquest mètode només es pot utilitzar en combinació amb el mètode *moveToInsertRow()*.

El següent fragment de codi dóna d'alta un nou registre a la taula de clients:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                      ResultSet.CONCUR_UPDATABLE);
// Executem la SELECT sobre la taula clients
String sql = "select * from clients";
rs = stmt.executeQuery(sql);
// Creem un nou registre a la taula de clients
rs.moveToInsertRow();
rs.updateString(2, "Killy Lopez");
rs.updateString(3, "Wall Street 3674");
rs.insertRow();
```

Si no facilitem valors a tots els camps del nou registre amb els mètodes *updateXXX()*, aquest camp tindrà un valor NULL, i si la base de dades no admet nuls es produirà una excepció *SQLException*.

A més d'inserir files en el nostre objecte *ResultSet*, també podrem eliminar files o registres. El mètode que s'ha d'utilitzar per a aquesta tasca és el mètode *deleteRow()* . Per eliminar un registre no hem de fer res més que moure'ns a aquest registre, i llançar el mètode *deleteRow()* sobre l'objecte *ResultSet* corresponent.

El següent fragment de codi esborra l'últim registre de la taula de clients:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                      ResultSet.CONCUR_UPDATABLE);
// Executem la SELECT sobre la taula clients
String sql = "select * from clients";
rs = stmt.executeQuery(sql);
```

```
// Ens situem a la fi del ResultSet
rs.last();
rs.deleteRow();
```

5. TRANSACCIONS

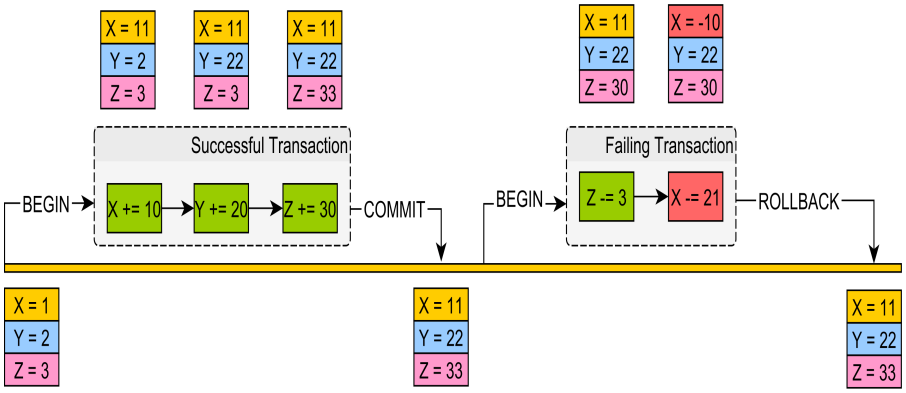
En alguns moments podem necessitar que diverses sentències SQL s'executen com un tot, i si falla alguna d'elles es desfacen els canvis realitzats per les sentències anteriors i es torne a la situació inicial. Així per exemple, al realitzar una transferència d'un compte d'un banc a un altre compte, les dues sentències encarregades de realitzar l'actualització en el saldo de cada compte s'han d'executar, però si una d'elles falla s'han de desfer els canvis realitzats. Aquestes dues sentències s'han d'executar en una mateixa transacció.

Les sentències que s'executen dins d'una mateixa transacció s'han d'executar TOTES amb èxit; si falla alguna es desfan els canvis per evitar qualsevol tipus d'inconsistència a la base de dades. Es pot dir que dins d'una transacció es dóna un "tot o res".

Quan establim una connexió a una base de dades amb JDBC, per defecte es crea en mode "autocommit" o "commit automàtic". Això vol dir que cada sentència SQL modifica la base de dades res més executar-se, és a dir, es tracta com una transacció que conté una única sentència.

Si volem agrupar diverses sentències SQL en una mateixa transacció utilitzarem el mètode *setAutoCommit()* de la interfície *Connection*, passant-li com a paràmetre el valor false, per indicar que cada sentència no s'execute automàticament.

Una vegada que s'ha deshabilitat la manera autocommit, les sentències que executem no modificaran la base de dades fins que no cridem al mètode *commit()* de la interfície Connection. Totes les sentències que s'han executat prèviament a la crida de *commit()* s'inclouran dins d'una mateixa transacció i es duran a terme sobre la base de dades com un tot.



En el cas d'una transferència entre dos comptes, i si suposem que el número del compte d'origen de la transferència és l'1 i el de destinació el 2, el següent codi efectuaria aquestes dues modificacions dins d'una transacció:

```
try {
        conn.setAutoCommit(false);
        int trans = 5000;
        int compteOrigen = 1;
        int compteDesti = 2;
        Statement stmt = conn.createStatement ();
        stmt.executeUpdate( "UPDATE Comptes SET quantitat = quantitat -"
             + trans +  "WHERE NumCompte =" + compteOrigen);
        stmt.executeUpdate( "UPDATE Comptes SET quantitat = quantitat +"
             + trans +  "WHERE NumCompte =" + compteDesti);
        conn.commit();
```

```
 }
catch (SQLException ex) {
        conn.rollback();
        System.err.println( "La transacció ha fallat.");
 }
 finally
 {
    conn.setAutoCommit(true);
 }
```

JDBC Batch Processing (**processament per lots**): La interfície *Statement* també suporta el processament de múltiples sentències en mode *batch*. D'aquesta manera minimitzem el nombre d'accesos a la base de dades, factor crític en algunes aplicacions.

Per exemple, suposem una aplicació que necessite realitzar moltes sentències de tipus INSERT i UPDATE. Si no utilitzem processament *batch* podem sobrecarregar en excés amb crides a la base de dades i disminuir el rendiment general de l'aplicació. Per això, si utilitzem el processament *batch* podem agrupar diverses sentències INSERT i UPDATE i llançar-les cap a la base de dades en una única crida. Per a fer això és necessari saber si la nostra base de dades suporta processament *batch*; podem consultar-ho amb la interfície *DatabaseMetaData* i trucar al mètode *supportBatchUpdates()*. Si torna true, vol dir que la nostra base de dades sí que el suporta.

```
Connection conn = DriverManager.getConnection(jdbcUrl, "root", "");
DatabaseMetaData dm = conn.getMetaData();
System.out.println("Suporta processament batch ->" + dm.supportsBatchUpdates());
```

El procediment per a utilitzar el processament per lots consisteix a cridar al mètode *addBatch()* que rep com a paràmetre la sentència SQL, i aquest procés el repetim per a cadascuna de les sentències SQL que vulguem afegir en el lot. Una vegada tinguem totes les sentències cridem al mètode *executeBatch()* de la interfície *Statement*. Prèviament hem d'haver desactivat l'*autocommit* amb el mètode *setAutoCommit()*, i finalment invocar el mètode *commit()* per realitzar els canvis.

En el següent exemple es creen 3 articles nous en mode batch sobre la base de dades empresa en MySQL:

```
Connection conn = DriverManager.getConnection(jdbcUrl,"root","");
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
// Afegim sentències SQL a manera Batch
String sql = "insert into articles values(8, 'HD 120G', 100.0, 'HD120', 2)";
stmt.addBatch(sql);
sql = "insert into articles values(9, 'HD 160G', 120.0, 'HD160', 2)";
stmt.addBatch(sql);
sql = "insert into articles values(10, 'HD 200G', 140.0, 'HD200', 2)";
stmt.addBatch(sql);
int result [] = stmt.executeBatch();
conn.commit();
conn.setAutoCommit(true);
```

Observa com el mètode *executeBatch()* retorna un array d'enters indicatiu de les files afectades per cada instrucció SQL.

Quan treballem amb JDBC podem trobar 4 tipus d'excepcions: *SQLException*, *SQLWarning*, *BatchUpdateException* i *DataTruncation*. La classe *SQLException* hereta d'*Exception*, i per tant té tota la funcionalitat per gestionar excepcions Java. Quasi tots els mètodes de l'API JDBC llancen una *SQLException*.

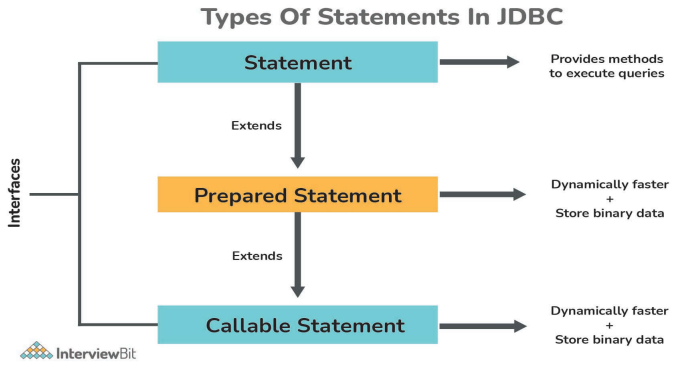
6. OBJECTES PreparedStatement

Aquesta interfície, al igual que la interfície *Statement*, ens permet executar sentències SQL sobre una connexió establerta amb una base de dades. Però, en aquest cas, anem a executar sentències SQL més especialitzades; aquestes sentències SQL es denominen **precompilades** i van a rebre paràmetres d'entrada.

La interfície *PreparedStatement* hereta de la interfície *Statement* i es diferencia d'ella en dues coses:

- les instàncies de *PreparedStatement* contenen sentències SQL que ja han estat compilades. Això és el que fa a una sentència "Prepared" (preparada).
- la sentència SQL que conté un objecte *PreparedStatement* pot contenir un, o més, paràmetres d'entrada. Un paràmetre d'entrada és aquell el valor del qual no s'especifica en la sentència; en la seva posició la sentència tindrà un signe d'interrogació (?) per cada paràmetre d'entrada. Abans d'executar la sentència s'ha d'especificar un valor per a cada un dels paràmetres a través dels mètodes *setXXX()* apropiats. Aquests mètodes *setXXX()* els afegeix la interfície *PreparedStatement*.

A causa de que les sentències dels objectes *PreparedStatement* estan precompilades, la seva execució serà més ràpida que la dels objectes *Statement*. Per tant, una sentència SQL que serà executada diverses vegades se sol crear com un objecte *PreparedStatement* per guanyar en eficiència. També s'utilitzarà aquest tipus de sentències per passar-li paràmetres d'entrada a les sentències SQL.



En heretar de la interfície *Statement*, la interfície *PreparedStatement* hereta totes les funcionalitats de *Statement*. A més, afegeix una sèrie de mètodes que permeten assignar un valor a cada un dels paràmetres d'entrada d'aquest tipus de sentències.

Els mètodes *execute()*, *executeQuery()* i *executeUpdate()* són sobrecarregats i en aquesta versió, és a dir, per als objectes *PreparedStatement* no prenen cap tipus d'arguments, d'aquesta manera, a aquests mètodes mai s'els haurà de passar per paràmetre l'objecte String que representava la sentència SQL a executar. En aquest cas, un objecte *PreparedStatement* ja és una sentència SQL per si mateixa, a diferència del que passava amb les sentències *Statement* que posseïen un significat només en el moment en què s'executaven.

Creant objectes *PreparedStatement*: per això s'ha de llançar el mètode *prepareStatement()* de la interfície *Connection* sobre l'objecte que representa la connexió establerta amb la base de dades.

En el següent fragment de codi es pot veure com es crearia un objecte *PreparedStatement* que representa una sentència SQL amb dos paràmetres d'entrada. L'objecte *pstmt* contindrà la sentència SQL indicada, la qual, ja ha estat enviada al gestor de la base de dades, i ja ha estat preparada per a la seva execució. En aquest cas s'ha creat una sentència SQL amb dos paràmetres d'entrada:

```
conn = DriverManager.getConnection(jdbcUrl,"root","");
pstmt = conn.prepareStatement("update facturas set serie = ? where id = ?");
```

Utilitzant els paràmetres d'entrada: abans de poder executar un objecte *PreparedStatement* s'ha d'assignar un valor per cadascun dels seus paràmetres. Això es realitza mitjançant la crida a un mètode

setXXX(), on XXX és el tipus apropiat per al paràmetre. Per exemple, si el paràmetre és de tipus long, el mètode a utilitzar serà *setLong()*.

El primer argument dels mètodes *setXXX()* és la posició ordinal del paràmetre, i el segon argument és el valor a assignar. Per exemple, en el següent fragment de codi crea un objecte *PreparedStatement* i acte seguit li assigna a el primer dels seus paràmetres un valor de tipus String i al segon un valor de tipus enter llarg:

```
String jdbcUrl = "jdbc:mysql://localhost:3306/empresa";
conn = DriverManager.getConnection(jdbcUrl,"root","");
pstmt = conn.prepareStatement("update facturas set serie=? where id=?");
pstmt.setString(1, "B");
pstmt.setLong(2, 1);
pstmt.executeUpdate();
```

Una vegada que s'han assignat uns valors als paràmetres d'entrada d'una sentència, l'objecte *PreparedStatement* es pot executar múltiples vegades, fins que siguin esborrats els paràmetres amb el mètode *clearParameters()*, encara que no cal trucar a aquest mètode quan es vulguin modificar els paràmetres d'entrada, sinó que al llançar els nous mètodes *setXXX()* els valors dels paràmetres seran reemplaçats. Per finalitzar, es crida al mètode *executeUpdate()* de l'objecte *PreparedStatement* i d'aquesta manera es veuran reflectits els canvis a la base de dades.

Aquesta documentació, creada per Ricardo Cantó Abad, es distribueix sota els termes de la llicència Creative Commons Reconeixement-NoComercial-CompartirIgual 3.0 No adaptada (CC BY-NC-SA 3.0) (http://creativecommons.org/licenses/by-nc-sa/3.0/es/deed.ca).