

# Persistencia de Datos

- 1 ¿qué es Persistencia?
- 2 ¿Dónde se ubica la capa de persistencia?
- 3 Tipos de persistencia con JAVA

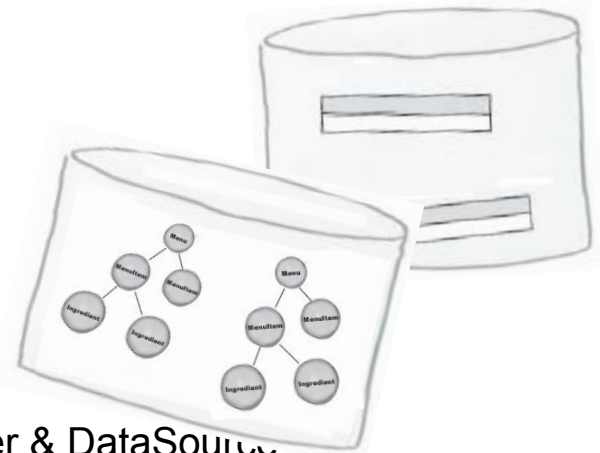
## (1) JDBC & SQL

Tipos de Drivers

La API JDBC

- Estableciendo una conexión: **DriverManager** & **DataSource**
- Sentencias SQL: objetos **Statement**, **PreparedStatement** y **CallableStatement**
- Soporte de Transacciones
- Manejo de Excepciones

## (2) Object Relational Mapping. JPA & Hibernate



# Persistencia

## Alternativas de la capa de persistencia

**Persistencia** es el almacenamiento de datos desde la memoria (donde trabaja un programa) a un repositorio permanente. En aplicaciones Orientadas a Objetos, la persistencia le permite a un objeto, “sobrevivir” a la aplicación que lo creó. El estado de los objetos puede almacenarse en disco, y un objeto con el mismo estado, puede ser re-creado en el futuro.

En los sistemas orientados a objetos, los objetos pueden hacerse persistentes de diferentes maneras. La elección del método de persistencia, es una parte importante del diseño de una aplicación. En Java, básicamente tenemos dos alternativas:

- **JDBC & SQL:** JDBC es una interface de programación, que permite independizar las aplicaciones del motor de base de datos usado. Incluye manejo de conexiones a base de datos, ejecución de sentencias SQL, *store procedures*, soporte de transacciones, etc. Son aconsejables cuando se tiene que reutilizar *store procedures*.
- **ORM (object/relational mapping):** es la persistencia automatizada y transparente de objetos pertenecientes a una aplicación java en tablas en una base de datos relacional, usando *metadata* que describen el mapeo entre los objetos y la base. ORM trabaja transformando datos desde una representación a otra. Se necesita usar motores de persistencia compatibles con JPA como Hibernate, Apache JPA, TopLink.

# Persistencia

## SQL & JDBC

El uso de SQL y JDBC para la persistencia de datos es laboriosa, se debe trabajar directamente con la API JDBC.

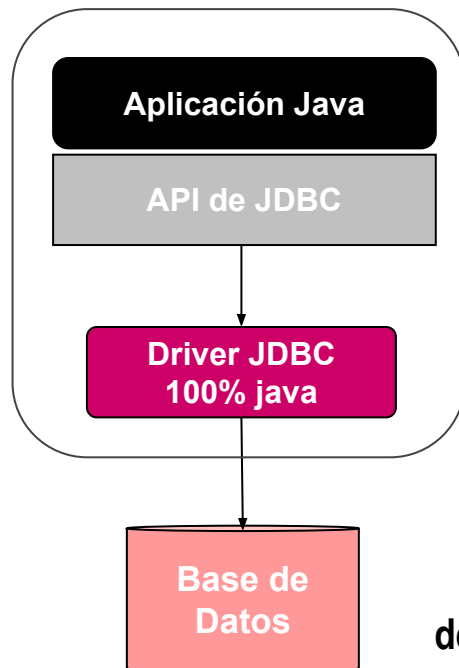
- La API JDBC provee un medio para acceder a una amplia variedad de fuentes de datos: DBMS, DBMS orientadas a objetos, planillas de cálculo, etc. El único requerimiento es que exista el driver JDBC apropiado.
- La API JDBC provee una interfaz de programación única, que independiza a las aplicaciones del motor de base de datos usado. Incluye **manejo de conexiones a base de datos, ejecución de sentencias SQL, store procedures, soporte de transacciones, etc.**
- JDBC define un conjunto de interfaces que un proveedor de base de datos implementa como una pieza de código llamada **driver**. Un **driver JDBC** traduce las invocaciones JDBC genéricas en invocaciones específicas de una Base de Datos.
- La API JDBC es una parte integral de la plataforma Java. La última especificación es la JDBC 4.3 que está disponible en la JSR 221 y es parte de Java SE 11.

Año	Versión	JSR (Java Specification Request)	JDK
1997	JDBC 1.0	--	JDK 1.1
1999	JDBC 2.0	--	JDK 1.2
2001	JDBC 3.0	JSR 54	JDK 1.4
2011	JDBC 4.3	JSR 221 - release 3	Java SE 11

# JDBC – Tipos de Drivers

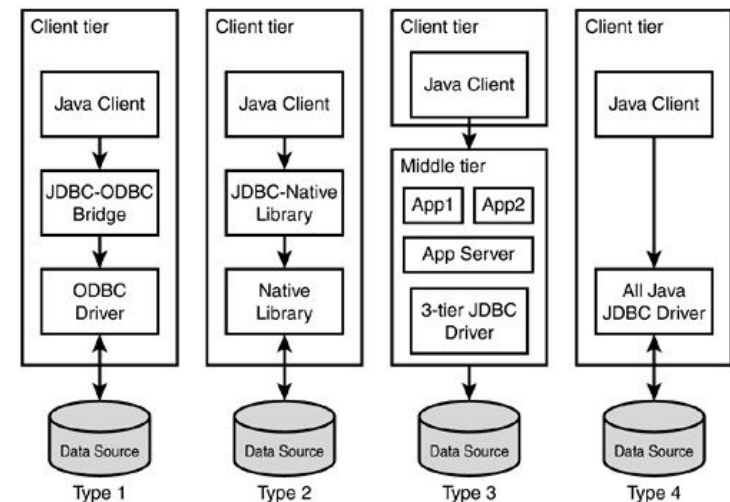
Existen diferentes tipos de drivers (tipo 1 al tipo 4), pero se recomienda utilizar el driver de tipo 4 porque:

- Es un driver **Java Puro** que habla directamente con la base de datos.
- No requiere de ninguna librería adicional ni de la instalación de un *middleware*, como en el caso de los otros tipos.
- La mayoría de los fabricantes de Base de Datos proveen drivers JDBC de tipo 4 para sus Bases de Datos.



Protocolo  
específico  
del proveedor

## Tipos de drive



# La API JDBC

Las clases e interfaces de la API JDBC están en los paquetes **java.sql** y **javax.sql**

En estos paquetes se encuentran definidos métodos que permiten: **conectarse a una BD, recuperar información acerca de la BD, realizar consultas SQL, ejecutar Stored Procedures y trabajar con los resultados.**

## java.sql (Core API)



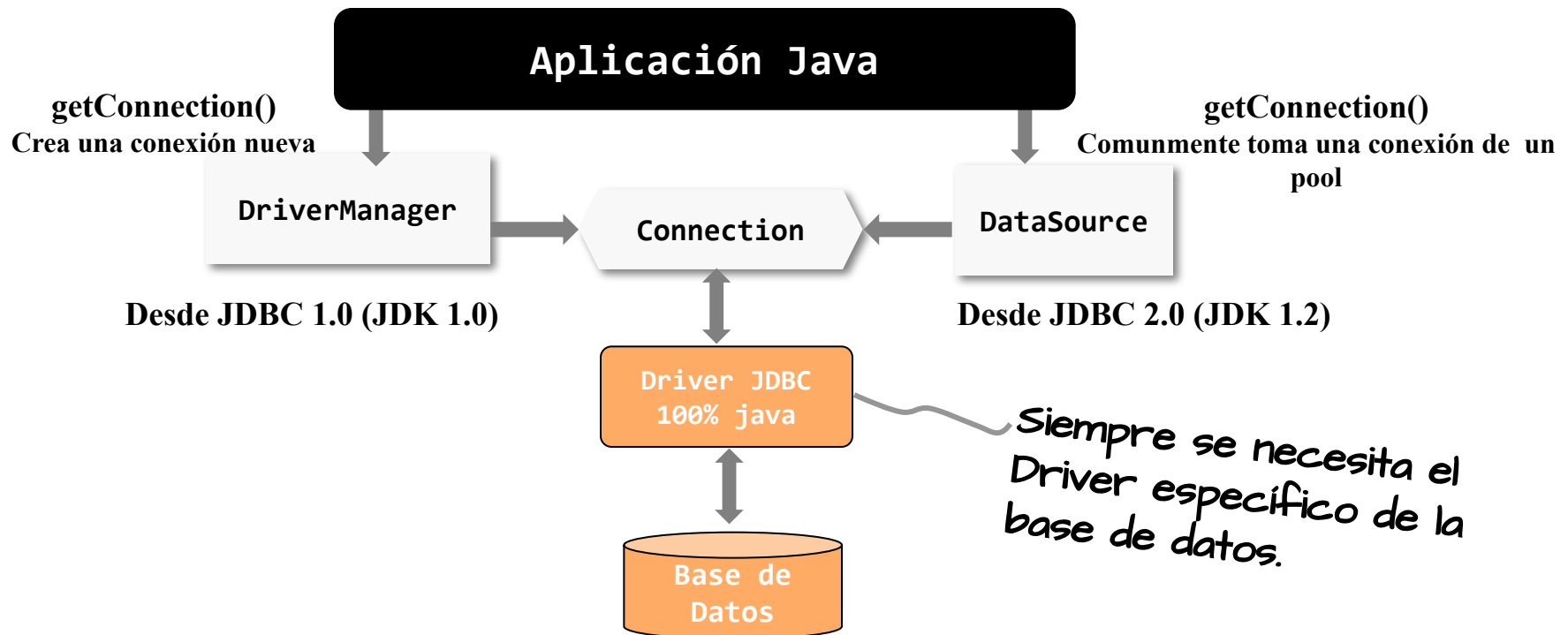
## javax.sql (Standard Extension)



# Persistencia

## SQL & JDBC

Las clases e interfaces de la API JDBC están en los paquetes `java.sql` y `javax.sql`. En estos paquetes se encuentran definidos métodos que permiten: conectarse a una base de datos, recuperar información relacionada con la base de datos, realizar consultas SQL, ejecutar Stored Procedures y trabajar con los resultados.



Una vez obtenido un objeto **Connection**, se pueden enviar comandos SQL desde la aplicación a la base de datos. Si la conexión no se puede establecer, se dispara una excepción SQL.

# Estableciendo una Conexión

## URL JDBC

Una base de datos en JDBC es identificada por una URL. La sintaxis recomendada para la URL de JDBC es la siguiente:

**jdbc:<subprotocolo>:<subnombre>**

**subprotocolo:** nombra a un mecanismo particular de conectividad a una base, que puede ser soportado por uno o más drivers.

**subnombre:** dependen del subprotocolo, pero en general, responde a una de las siguientes sintaxis:

```
database
//host/database
//host:port/database
```

### Ejemplos:

"jdbc:odbc:empleadosDB"      El origen de datos ODBC es **empleadosDB**, debe haberse definido en el cliente

"jdbc:mysql://localhost:3306/cursoJ2EE"      Una URL para mysql con el driver **Connector/J**

"jdbc:db2://server:50000/EMPLE"      Permite conectarse a la base de IBM de nombre **EMPLE**

"jdbc:sqlite:test.db"      URL para un driver de SQLite

"jdbc:postgresql:stock"      Los valores por defecto para **postgreSQL** son: localhost y port=5432.

# Estableciendo una Conexión

## Driver Manger - Datasource

### La clase DriverManager

- Es una clase que fue introducida en el original JDBC 1.0. Cuando una aplicación intenta conectarse por primera vez a una fuente de datos, especificando la URL, **DriverManager** cargará automáticamente cualquier driver JDBC, encontrado en el CLASSPATH y a partir de ahí intenta establecer una conexión.

```
miConexion = DriverManager.getConnection("jdbc:odbc:empleadosDB", usr, contra);  
  
miConexion = DriverManager.getConnection("jdbc:sqlite:test.db");
```

- Es una clase que viene con la API, con lo cual, un proveedor no puede optimizarla. Mantiene internamente drivers JDBC y dada una URL JDBC retorna una conexión usando el driver apropiado.

### La interfaz DataSource

- Esta interfaz fue introducida en JDBC 2.0 y representa una fuente de datos particular.
- La interface **DataSource** es implementada por los proveedores de DB (Drivers). Permite elegir las mejores técnicas para lograr un acceso óptimo a la base de datos y definir qué atributos son necesarios para crear las conexiones. Es el mecanismo preferido para obtener una conexión porque permite que los detalles acerca de los datos subyacentes, se mantengan transparentes para la aplicación. Para crear una conexión usando DataSource no se necesita información sobre la base, el servidor, el usuario, la clave, etc., en el código java.



# Estableciendo una Conexión

## Estableciendo una conexión con DriverManager

La clase **DriverManager** dispone de 3 métodos de clase que permiten establecer una conexión con una fuente de datos.

- `getConnection(String url)`
- `getConnection(String url, String usr, String pwd)`
- `getConnection(String url, Properties info)`

```
. . .  
Connection con=null;  
try {  
    con=DriverManager.getConnection("jdbc:sqlite:test.db");  
    Statement st = con.createStatement();  
    . . .  
    st.close();  
    con.close();  
} catch (SQLException e) {  
    System.out.println("no se pudo conectar a la BD");  
}
```

# Estableciendo una Conexión

## La interface Connection

Las aplicaciones usan la interfaz Connection para especificar atributos de transacciones y para crear objetos Statement, PreparedStatement o CallableStatement. Estos objetos son usados para ejecutar sentencias SQL y recuperar resultados. Esta interface provee los siguientes métodos:

Statement **createStatement()** throws SQLException  
Statement **createStatement(int resultSetType, int resultSetConcurrency)**  
throws SQLException {}

(permite definir si es *READ\_ONLY*  
(por defecto) o *UPDATABLE*.

(permite definir si es *FORWARD* (por defecto) o en ambas direcciones)  
Por ejemplo: **ResultSet.TYPE\_FORWARD\_ONLY**)

Sentencia SQL a ser ejecutada

PreparedStatement **prepareStatement(String sql)** throws SQLException  
PreparedStatement **prepareStatement(String sql, int resultSetType,  
int resultSetConcurrency)** throws SQLException {}

Store Procedure a ejecutarse

CallableStatement **prepareCall(String sql)** throws SQLException  
CallableStatement **prepareCall(String sql, int resultSetType,  
int resultSetConcurrency)** throws SQLException {}

# Estableciendo una Conexión

## Objetos `java.sql.Statement`

Un objeto `Statement` se crea con el método `createStatement()` y puede ejecutarse con `executeUpdate()` o `executeQuery()`.

<code>ResultSet executeQuery(String sql)</code>	Ejecuta la sentencia SQL que retorna un simple objeto <b>ResultSet</b> . Para sentencias <b>SELECT</b>
<code>int executeUpdate(String sql)</code>	Ejecuta la sentencia SQL que retorna un <b>int</b> que indica la cantidad de filas afectadas o 0 si se envía una sentencia DDL (Data Definition Language) que crean base de datos, tablas, etc. o actualizan base de datos.

Creación de un objeto `Statement`:

```
Statement sent = miConexion.createStatement();
```

Ejecución de sentencias SQL:

```
ResultSet resul=sent.executeQuery("select nombre,edad from empleado");  
int res=sent.executeUpdate("insert into empelado values('Juan', 56)");
```

# Consulta de datos a una Table

## `executeQuery` & `ResultSet`

El resultado de un `executeQuery()` para ejecutar un `select`, es un objeto `ResultSet`. Este objeto contiene un *cursor* que puede manipularse para hacer referencia a una fila particular del `ResultSet`. Inicialmente se ubica en la posición anterior a la primera fila. El método `next()` avanza una fila.

Métodos para recorrer el `ResultSet`:

```
boolean next() throws SQLException
boolean previous() throws SQLException
boolean first() throws SQLException
boolean last() throws SQLException
boolean absolute(int pos) throws SQLException
```

Devuelven **true** si el cursor está en una fila válida y **false** en caso contrario

Devuelve **true** si el cursor está en una fila válida y **false** si `pos` es `<1` o mayor que la cantidad de filas.

Recuperar y Actualizar campos del `ResultSet`:

Los campos de cada fila del `ResultSet` pueden obtenerse mediante su nombre o posición. El método a usar depende del tipo de dato almacenado:

```
String getString(int indiceColum) throws SQLException
String getString(String nombreCol) throws SQLException
int getInt(int indiceCol) throws SQLException
int getInt(String nombreCol) throws SQLException
void updateString(int indiceColum, String y) throws SQLException
```

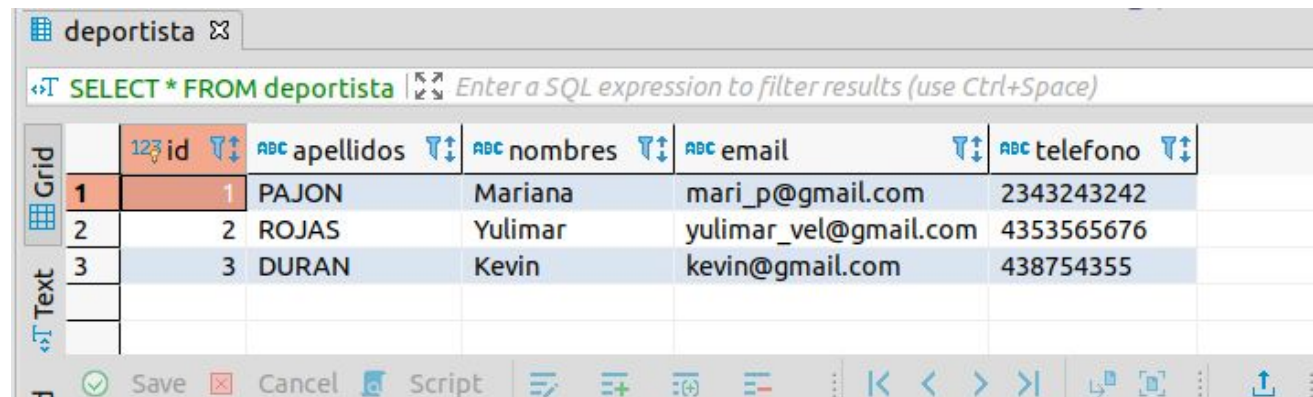
# Consulta de datos a una Table executeQuery & la sentencia SQL select

```
...
Connection con=null;
try {
    con=DriverManager.getConnection("jdbc:mysql://localhost:3306/tokio2021");
    Statement sent = con.createStatement();
    ResultSet resul = sent.executeQuery("SELECT * FROM deportista");
    // Si entra al while obtuvo al menos una fila
    while (resul.next()){
        System.out.println(resul.getString("apellidos")+ ", "+ resul.getString("nombres"));
    }
    sent.close();
    con.close();
} catch (SQLException e) {
    System.out.println(e.getMessage());
}
```



Thumbnail showing the structure of the 'deportista' table with columns: id, apellidos, nombres, email, and telefono.

id	apellidos	nombres	email	telefono
----	-----------	---------	-------	----------

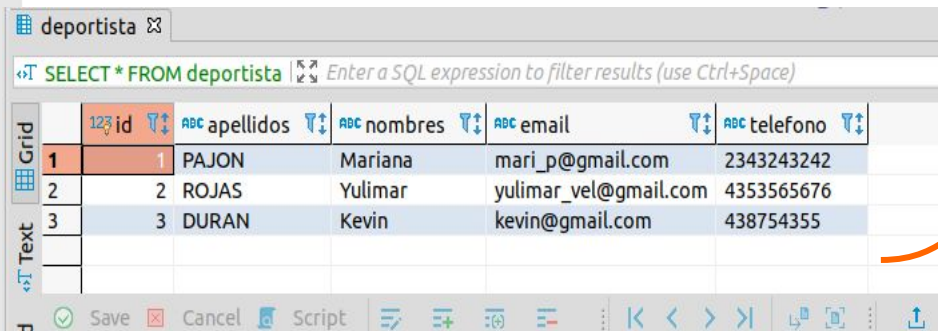


Screenshot of a database interface showing the 'deportista' table. The SQL query 'SELECT \* FROM deportista' is entered in the query bar. The table data is displayed below.

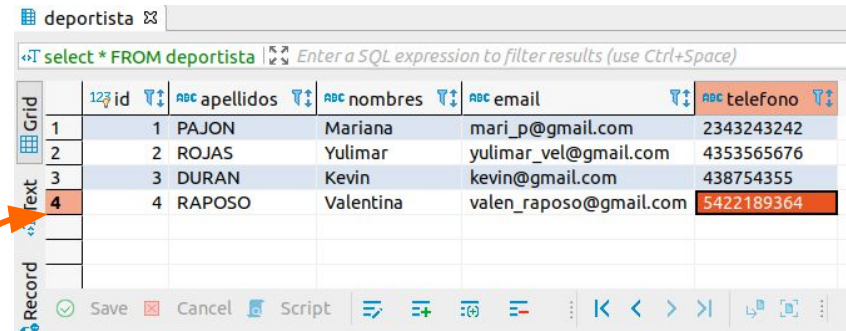
	id	apellidos	nombres	email	telefono
1	1	PAJON	Mariana	mari_p@gmail.com	2343243242
2	2	ROJAS	Yulimar	yulimar_vel@gmail.com	4353565676
3	3	DURAN	Kevin	kevin@gmail.com	438754355

# Consulta de datos a una Table executeUpdate & la sentencia SQL insert

```
...
Connection con=null;
try {
    con=DriverManager.getConnection("jdbc:mysql://localhost:3306/tokio2021");
    Statement st = con.createStatement();
    String query = "INSERT INTO deportista (apellidos, nombres, email, telefono)
        VALUES('RAPOSO','Valentina','valen_raposo@gmail.com', '5422189364')";
    res = sent.executeUpdate(query);
    st.close();
    con.close();
} catch (SQLException e) {
    System.out.println(e.getMessage());
}
```



	id	apellidos	nombres	email	telefono
1	1	PAJON	Mariana	mari_p@gmail.com	2343243242
2	2	ROJAS	Yulimar	yulimar_vel@gmail.com	4353565676
3	3	DURAN	Kevin	kevin@gmail.com	438754355



	id	apellidos	nombres	email	telefono
1	1	PAJON	Mariana	mari_p@gmail.com	2343243242
2	2	ROJAS	Yulimar	yulimar_vel@gmail.com	4353565676
3	3	DURAN	Kevin	kevin@gmail.com	438754355
4	4	RAPOSO	Valentina	valen_raposo@gmail.com	5422189364

**Nota:** Las tablas pueden estar definidas con su clave primaria autoincremental o no

# La API JDBC

## Objetos java.sql.PreparedStatement

- Un objeto **PreparedStatement**, es un tipo de sentencia sql que se precompila, y puede ser utilizada repetidas veces sin recompilar -> mejora la performance.
- A diferencia de las sentencias tradicionales cuando se crean requieren de la sentencia SQL como argumento del constructor. Esta sentencia es enviada al motor de la base de datos para su compilación y cuando se ejecuta, no se recompila.
- Como la sentencia puede ejecutarse repetidas veces, puede especificarse que recibirá parámetros usando "?". Los métodos **set<datatype>()** permite setear los valores para la ejecución de la sentencia.

Creación de una sentencia preparada:

```
PreparedStatement p_sent=  
miConexion.prepareStatement("SELECT * FROM Empleados WHERE edad>? AND Sexo=?")
```

Parámetros de la sentencia SELECT




Configuración de los parámetros de la sentencia y ejecución:

```
p_sent.clearParameters();  
p_sent.setInt(1,55);  
p_sent.setChar(2,"F");
```

← Opcional, para limpiar cualquier parámetro seteado anteriormente

← Seta los parámetros de la sentencia pre-compilada



```
ResultSet resul = p_sent.executeQuery();
```

# La API JDBC

## Objetos `java.sql.CallableStatement`

- Un objeto `CallableStatement` provee una manera para llamar a stored procedures (SP) para cualquier DBMSs. Los SP son programas almacenados que ejecutan en el propio motor de la Base de Datos. Típicamente se escriben en el lenguaje propio de la base de datos, aunque es posible hacerlo en Java.
- Los Store Procedures se parametrizan a través de los métodos `set<datatype>()` de la misma manera que las sentencias preparadas.

### Creación y ejecución de un procedimiento almacenado (SP)

#### Sin parámetros:

```
CallableStatement miSP = miConexion.prepareCall("call SP_CONSULTA");  
ResultSet resul = miSP.executeQuery();
```

#### Con parámetros:

```
CallableStatement miSP = miConexion.prepareCall("call SP_CONSULTA[(?,?)]");  
miSP.setString(1, "Argentino");  
miSP.setFloat(2, "12,56f");  
ResultSet resul = miSP.executeQuery();
```



# La API JDBC

## Soporte de transacciones

El objeto de tipo `java.sql.Connection` soporta el manejo de transacciones SQL. Una transacción SQL es un conjunto de sentencias SQL que deben ser ejecutadas como una unidad atómica. Para que la transacción sea exitosa cada sentencia debe serlo.

Cuando se crea un objeto `Connection`, automáticamente es configurado para que la ejecución de cada sentencia actualice en la base de datos (*commit* implícito). En este caso no se pueda deshacer dicha acción (no soporta *rollback*).

Con el objeto `Connection` se puede controlar cuando las sentencias SQL son efectivizadas (*committed*). Los siguientes métodos son provistos:

- **`void setAutoCommit(boolean autoCommit)`**: configura si las sentencias SQL son automáticamente *committed* o no. Si es `true` cuando una sentencia se ejecuta se efectiviza en la DB. Si es `false`, las sentencias no son *committed* hasta que no se ejecute el método `commit()`.
- **`boolean getAutoCommit()`**: devuelve el modo de la conexión, `true` si efectiviza cada sentencia o `false` si requiere el `commit()` explícito para actualizar.
- **`void commit()`**: es usado para efectivizar un conjunto de sentencias SQL.
- **`void rollback()`**: este método deshace todas las sentencias SQL ejecutadas después del último `commit()`.

# La API JDBC

## Soporte de transacciones

La Base de Datos es responsable de guardar las sentencias ejecutadas y es capaz de deshacer todas aquellas sentencias SQL ejecutadas después del último `commit()`, cuando encuentra el método `rollback()`.

```
try {  
    ...  
    miConexion.setAutoCommit(false);  
    Statement s = miConexion.createStatement();  
    int tran1 = s.executeUpdate("INSERT INTO TCuentas VALUES ('1001', -100)");  
    int tran2 = s.executeUpdate("INSERT INTO TCuentas VALUES ('1002', 100)");  
    if ((tran1>0) && (tran2>0)) {  
        miConexion.commit();  
    }  
    else  
        miConexion.rollback();  
} catch (SQLException e1) {  
    ...  
}
```

Permite tener el control de qué y cuándo confirmar operaciones sobre la conexión.

```
try {  
    java.sql.Statement statement = miConexion.createS  
    java.sql.ResultSet results = statement.executeQuer  
    miConexion.commit();  
    miConexion.close();  
} catch (Throwable t) {  
    miConexion.rollback();  
    miConexion.close();  
}
```

# La API JDBC

## Manejo de excepciones

Un objeto JDBC que encuentra un error serio (falla la conexión a la base, sentencias SQL mal formadas, falta de privilegios, etc.) dispara una excepción **SQLException**.

La clase **SQLException** extiende **java.lang.Exception** y define 3 métodos adicionales útiles:

- **getNextException()**: permite recorrer una cadena de errores sql.
- **getSQLState()**: devuelve un código de error SQL según ANSI-92.
- **getErrorCode()**: retorna un código de error específico del proveedor.

```
try {  
    // código de acceso a la base de datos  
} catch (SQLException e) {  
    while (e!=null){  
        System.out.println("SQL Exception:" + e.getMessage());  
        System.out.println("Error SQL ANSI-92:" + e.getSQLState());  
        System.out.println("Código de error del  
Proveedor:" + e.getErrorCode());  
        e = e.getNextException();  
    }  
}
```

Hasta las versiones JDBC 3.0, para todo tipo de excepción se disparaba SQLException.

# La API JDBC

## Soporte de transacciones

JDBC 4 ha mejorado el manejo de excepciones, incorporando:

- Clasificación de `SQLException`: **no transitorias**, representan fallas provocadas por alguna condición que debe ser resuelta antes de reintentar y **transitorias**, representan fallas que sin intervención podrían ser exitosas en otro intento.

```
SQLException
+---> SQLNonTransientException
|   +---> SQLDataException
|   +---> SQLFeatureNotSupportedException
|   +---> SQLIntegrityConstraintViolationException
|   +---> SQLInvalidAuthorizationException
|   +---> SQLNonTransientConnectionException
|   +---> SQLSyntaxErrorException
+---> SQLTransientException
|   +---> SQLTimeoutException
|   +---> SQLTransactionRollbackException
|   +---> SQLTransientConnectionException
```

```
try {
    s.execute("Hola Mundo!!! ");
} catch (SQLSyntaxErrorException ex) {
    System.out.println("Problema con la Sintaxis SQL");
}
s.execute("create table testTable(id int,name varchar(8))");
try {
    s.execute("insert into testTable values (1,'Juan Moreno')");
} catch (SQLDataException ex) {
    System.out.println("Problema permanente con los datos de entrada");
}
```

- La clase `SQLException` implementa la **interface** `Iterable`, por lo que soporta la estructura de control **for each** (disponible a partir del J2SE 5.0) para recorrer las excepciones.

```
try {
    // código de acceso a la base de datos
} catch (SQLException e) {
    for (Throwable t: e)
        System.out.println("Error SQL"+ t);
}
```

"para cada t de tipo  
Throwable en e"