UT9

ACCESO A BASES DE DATOS RELACIONALES

1. INTRODUCCIÓN

- Java tiene una API que permite interactuar con fuentes de datos de manera que podemos:
  - Conectarnos a una base de datos (BD)
  - Enviar consultas de selección, inserción y actualización de la BD
  - Recuperar datos de una consulta y manejarlos

- El acceso a bases de datos desde Java se realiza mediante el estándar **JDBC**, que permite un acceso a las BD **independientemente del SGBD**.

1. INTRODUCCIÓN

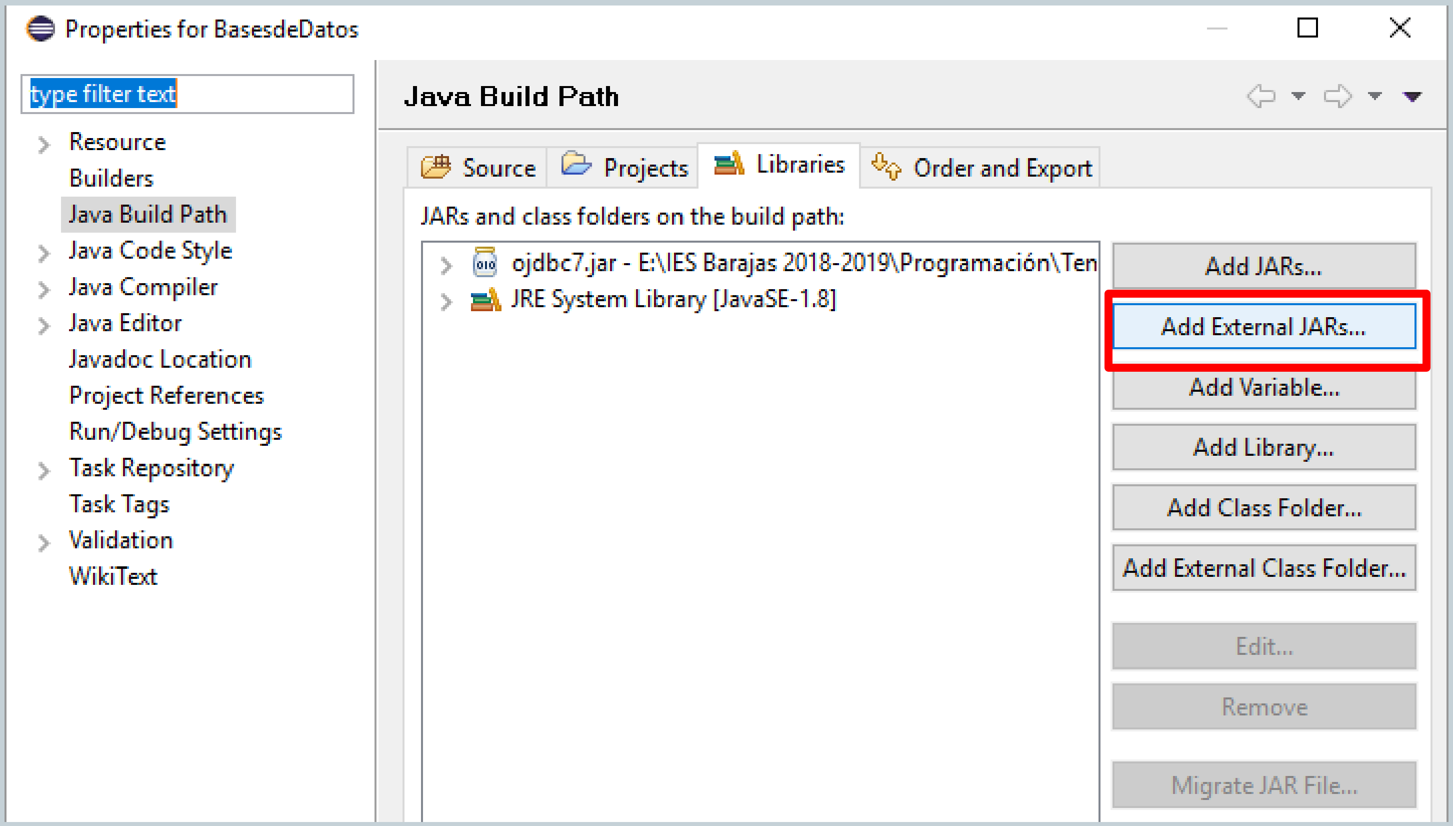
- ***Java Database Connectivity (JDBC)***, es una API que permite la ejecución de operaciones de BD desde el lenguaje de programación Java
  - Es independiente del sistema operativo
  - Es independiente del SGBD
  - Utiliza el dialecto SQL del modelo de base de datos que se utilice.
- Las aplicaciones escritas en Java no necesitan conocer las especificaciones de un SGBD en particular, basta con comprender el funcionamiento de JDBC.
- Cada SGBD que quiera utilizarse con JDBC debe contar con un adaptador o **controlador**.

2. Conexión desde JAVA

- Es necesario descargar el **driver JDBC específico** para conectar Java con el SGBD, *en nuestro caso Oracle*.
- Oracle provee de forma libre dicho driver: https://www.oracle.com/technetwork/database/features/jdbc/default-2280470.html
- El primer paso para conectarnos a una base de datos desde nuestro proyecto JAVA es incorporar el driver JDBC al classpath del mismo.
- El fichero jar es necesario para ejecutar la aplicación
  - Para utilizar nuestra app **el jar de JDBC debe estar disponible**

2. Conexión desde JAVA

- Botón derecho sobre el proyecto > Propiedades



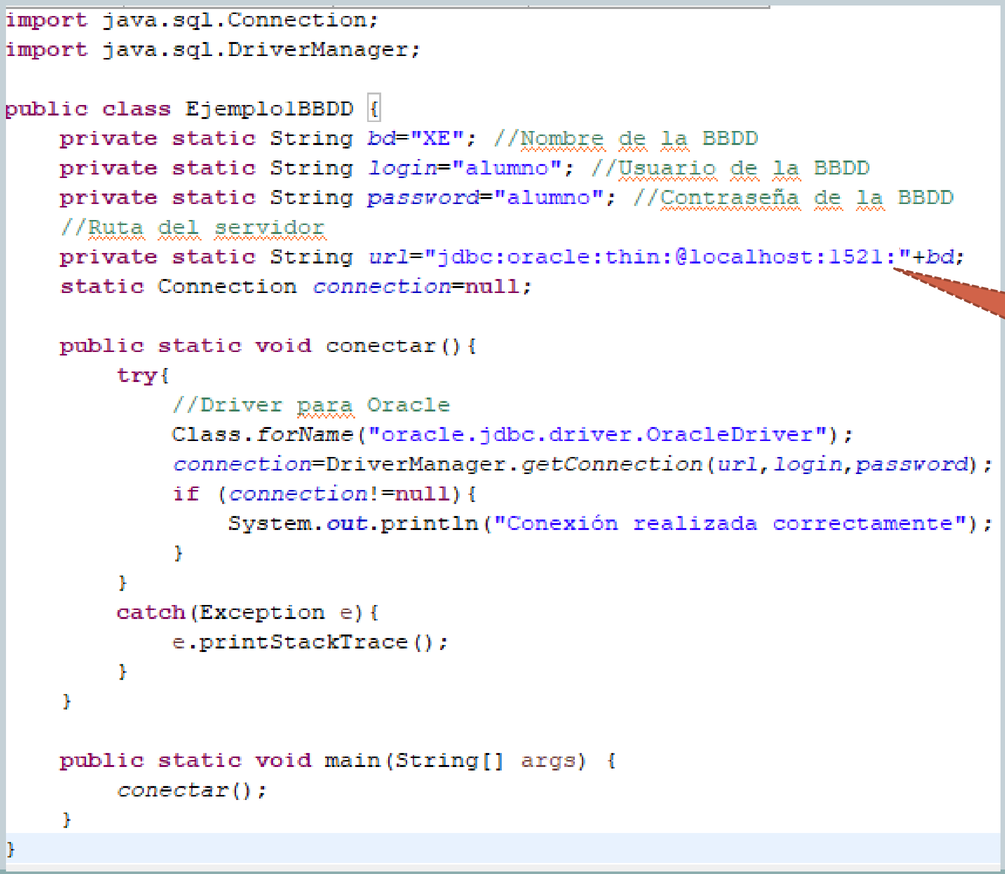
Properties for BasesdeDatos

JARs and class folders on the build path:

- ojdbc7.jar - E:\IES Barajas 2018-2019\Programación\Ten
- JRE System Library [JavaSE-1.8]

3. Establecimiento de la conexión

- Para establecer la conexión con una BBDD, se debe realizar a través de un objeto **java.sql.Connection:**



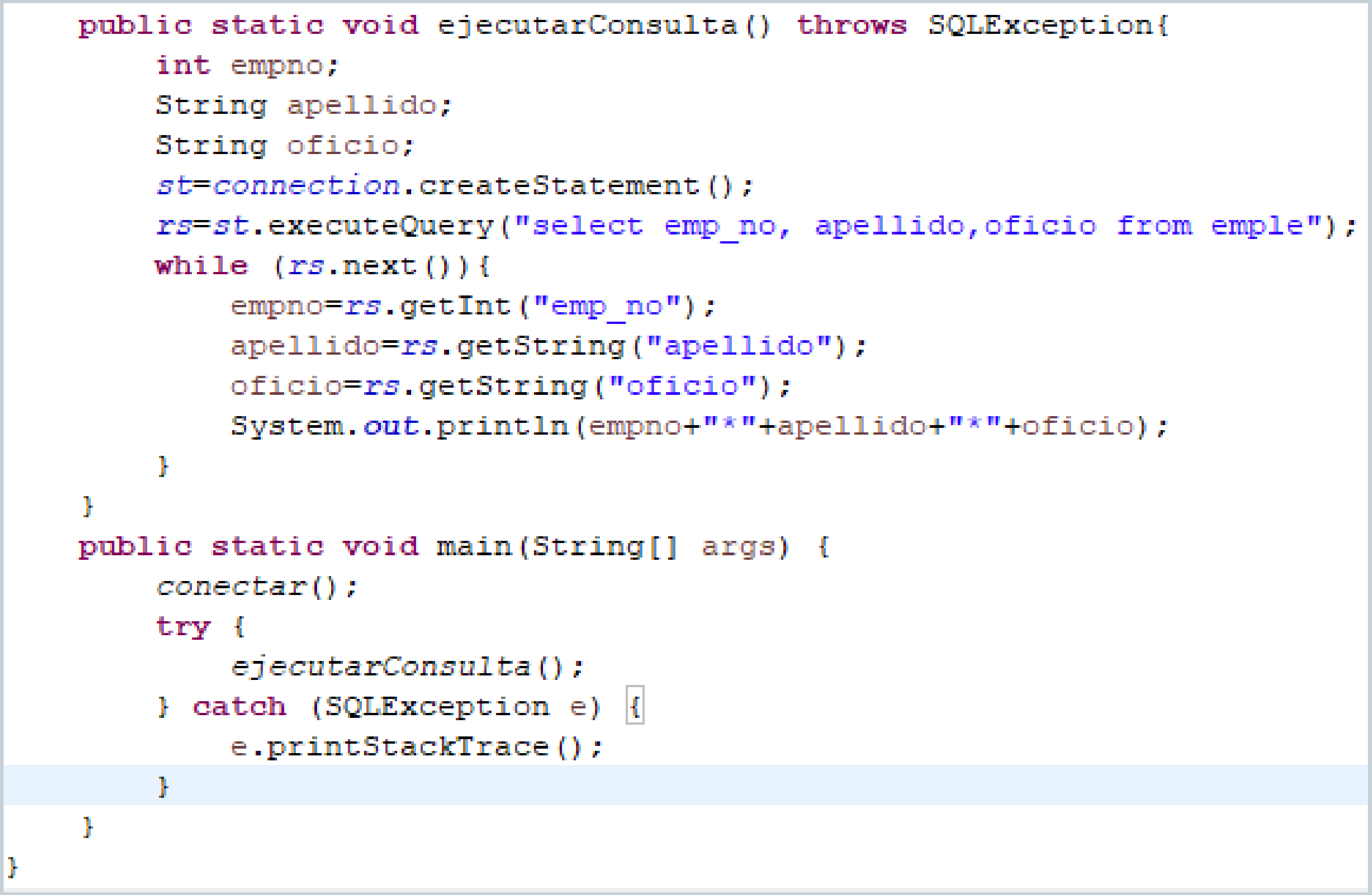
```
import java.sql.Connection;
import java.sql.DriverManager;

public class Ejemplo1BBDD {
    private static String bd="XE"; //Nombre de la BBDD
    private static String login="alumno"; //Usuario de la BBDD
    private static String password="alumno"; //Contraseña de la BBDD
    //Ruta del servidor
    private static String url="jdbc:oracle:thin:@localhost:1521:"+bd;
    static Connection connection=null;

    public static void conectar(){
        try{
            //Driver para Oracle
            Class.forName("oracle.jdbc.driver.OracleDriver");
            connection=DriverManager.getConnection(url,login,password);
            if (connection!=null){
                System.out.println("Conexión realizada correctamente");
            }
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        conectar();
    }
}
```

Puede ser / en lugar de :

4. Procesador de consultas

- Establecida la conexión, se pueden ejecutar consultas SQL a la base de datos conectada.
- Las consultas se realizan a través de un objeto **java.sql.Statement,** obtenido de un objeto **Connection**.
- El resultado de una consulta es un objeto **java.sql.ResultSet.**
- Un ResultSet contiene la tabla resultante de una consulta.

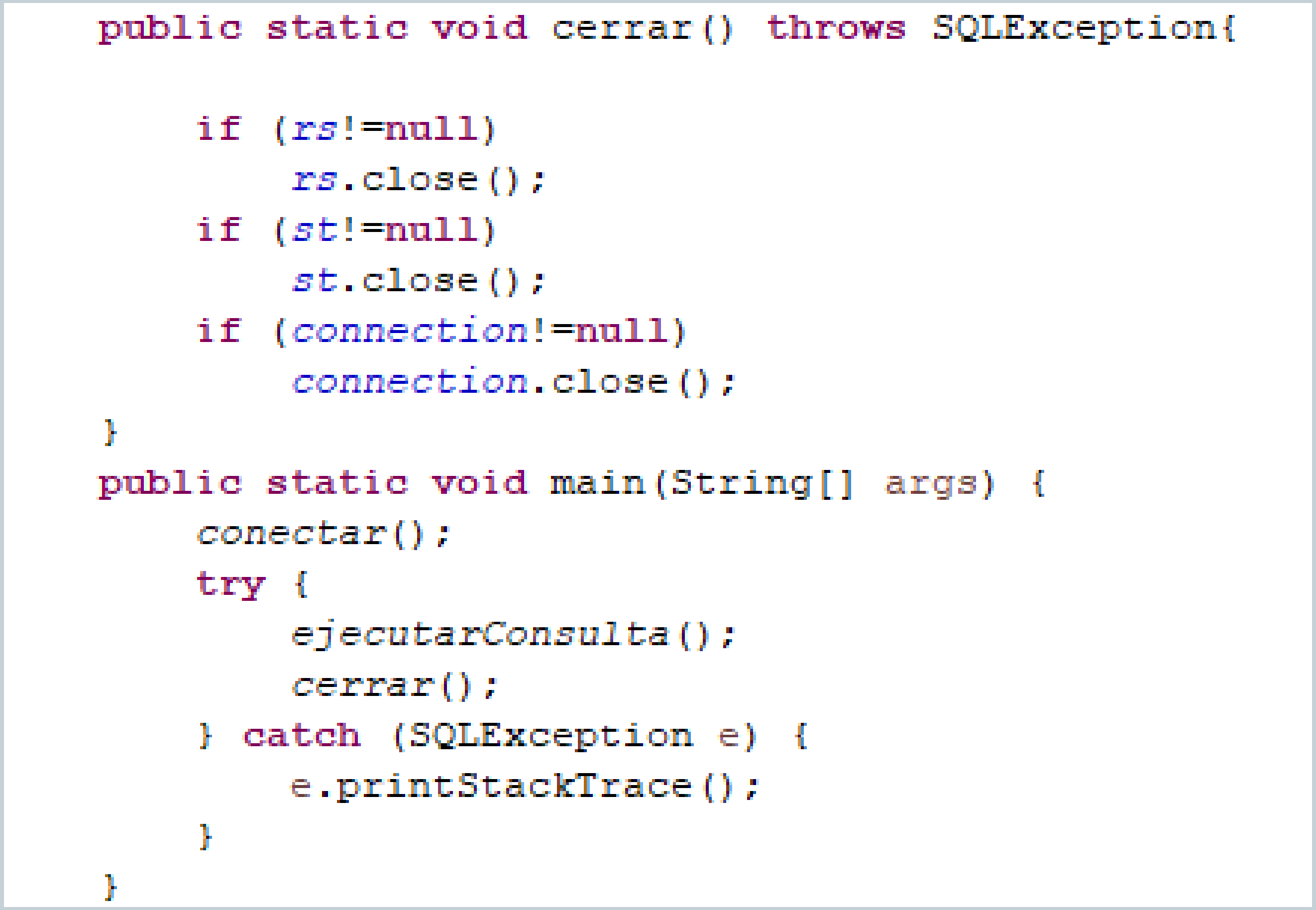
4. Procesador de consultas



```
    public static void ejecutarConsulta() throws SQLException{
        int empno;
        String apellido;
        String oficio;
        st=connection.createStatement();
        rs=st.executeQuery("select emp_no, apellido,oficio from emple");
        while (rs.next()){
            empno=rs.getInt("emp_no");
            apellido=rs.getString("apellido");
            oficio=rs.getString("oficio");
            System.out.println(empno+"*"+apellido+"*"+oficio);
        }
    }
    public static void main(String[] args) {
        conectar();
        try {
            ejecutarConsulta();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

4. Procesador de consultas

- Por último, debemos cerrar todo tras su uso:



```
public static void cerrar() throws SQLException{

    if (rs!=null)
        rs.close();
    if (st!=null)
        st.close();
    if (connection!=null)
        connection.close();
}
public static void main(String[] args) {
    conectar();
    try {
        ejecutarConsulta();
        cerrar();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

4.1 La clase STATEMENT

- El objeto **st** de la clase **Statement** lo hemos utilizado para enviar sentencias SQL a la base de datos.
- Existen tres tipos de objetos Statement:
  - **Statement:** envia órdenes SQL a la base de datos **sin parámetros**
  - **PreparedStatement:** hereda de Statement. Se utiliza para ejecutar comandos SQL con o sin parámetros de entrada ya **precompilados**.
  - **CallableStatement:** hereda de PreparedStatement. Se utiliza para llamar a procedimientos almacenados en la base de datos.

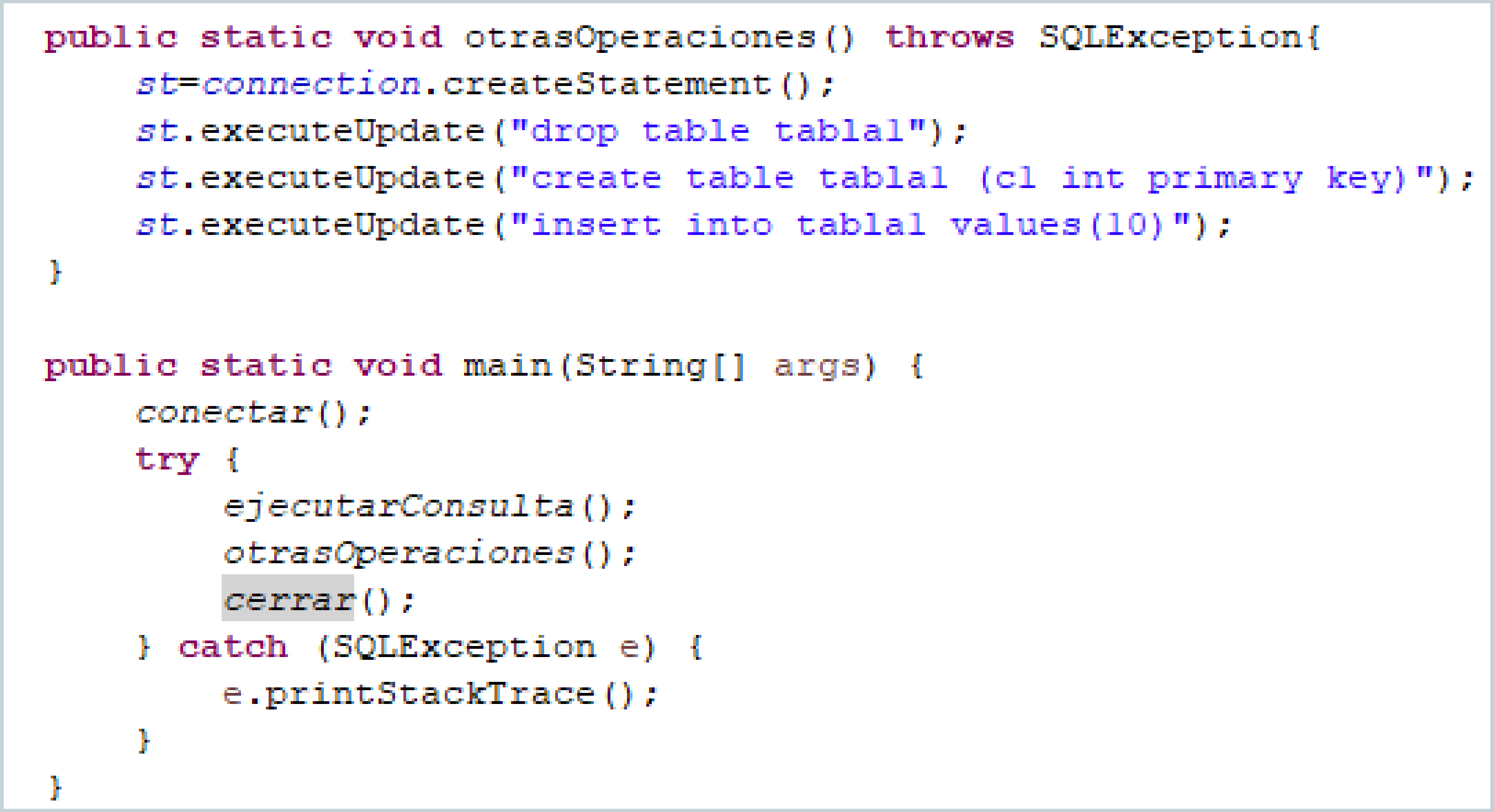
- Un objeto de la clase Statement se crea mediante el método de Connection **createStatement()**.

4.1 La clase STATEMENT

- Una vez realizada la conexión y creado el objeto **Statement** se pueden llamar a tres métodos diferentes para ejecutar sentencias SQL:
  - **executeQuery:** se utiliza para ejecutar sentencias SELECT y la llamada a este método devuelve un **ResultSet** que es un objeto que almacena los datos devueltos por la BD.
  - **executeUpdate:** se utiliza para ejecutar sentencias DDL (create table, drop table, etc.) y se puede utilizar para ejecutar sentencias INSERT, UPDATE y DELETE. Devuelve un entero que indica el número de filas afectadas por la sentencia (en sentencias DDL siempre es 0).
  - **execute:** utilizado en sentencias que devuelven más de un ResulSet. Se utiliza solamente en programación avanzada.

4.1 La clase STATEMENT

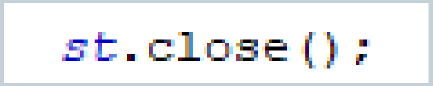
- Ejemplo:



```
public static void otrasOperaciones() throws SQLException{
    st=connection.createStatement();
    st.executeUpdate("drop table tabla1");
    st.executeUpdate("create table tabla1 (c1 int primary key)");
    st.executeUpdate("insert into tabla1 values(10)");
}

public static void main(String[] args) {
    conectar();
    try {
        ejecutarConsulta();
        otrasOperaciones();
        cerrar();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

4.1 La clase STATEMENT

- Como buena práctica se recomienda cerrar los objetos mediante este comando:



```
st.close();
```

- La llamada al método close() hace que se libere inmediatamente la basura y se eviten posibles problemas con la memoria.

- No obstante, los objetos Statement son cerrados automáticamente por el garbage collector de Java.

ACTIVIDAD

- Realiza los ejercicios del 1 al 4 de la Hoja de Ejercicios

5. Procesado de consultas preparadas

- Las sentencias preparadas de JDBC permiten consultas o actualizaciones más eficientes (precompiladas).
- Al compilar la sentencia SQL, se analiza cuál es la estrategia adecuada según las tablas, las columnas, los índices y las condiciones de búsqueda implicados.
- Esto consume tiempo de procesador, pero al realizar la compilación una sola vez, se logra mejorar el rendimiento en siguientes consultas iguales con valores diferentes.

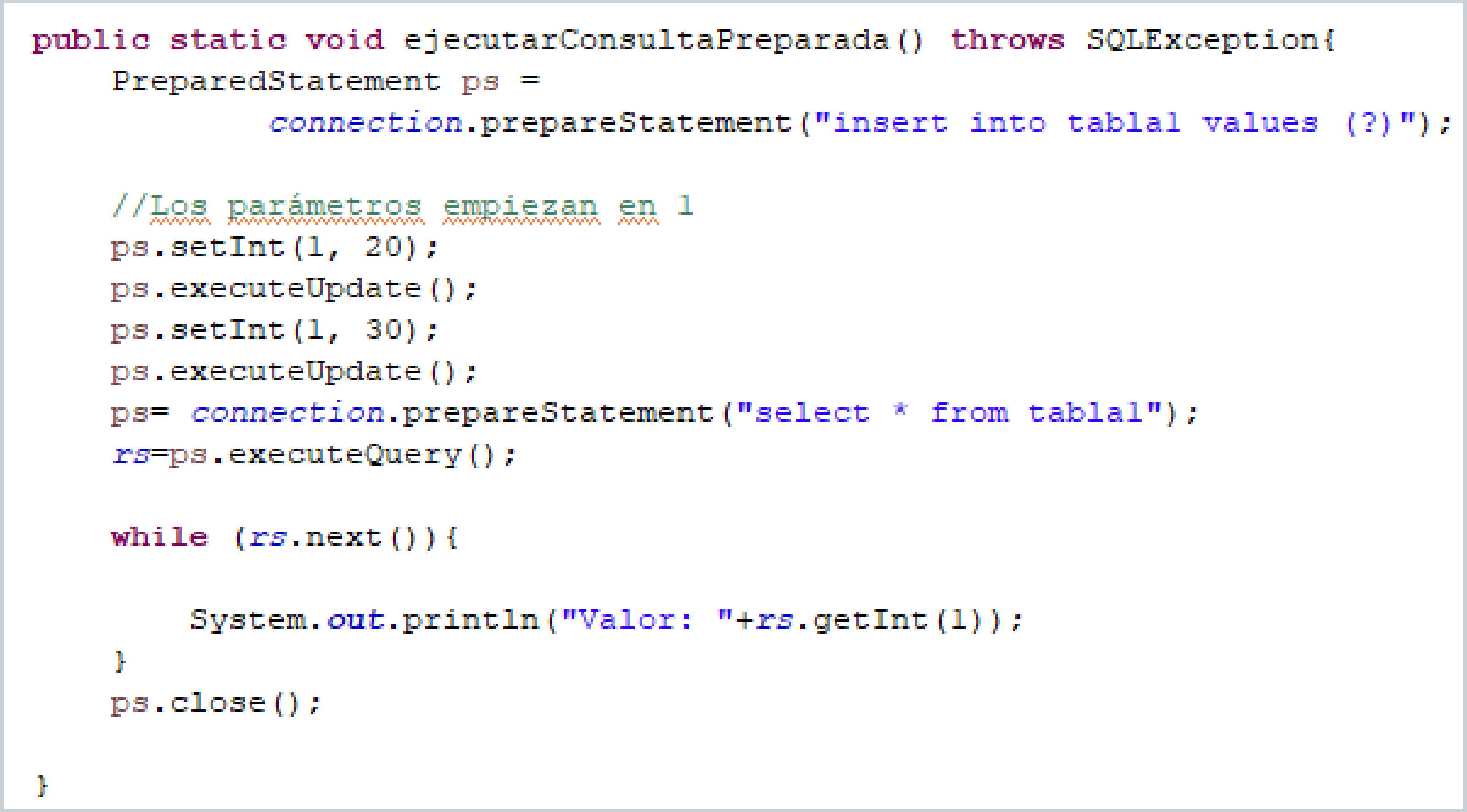
5. Procesado de consultas preparadas

- Otra ventaja de las sentencias preparadas es que permiten la **parametrización:**
  - La sentencia SQL se escribe una vez, indicando las posiciones de los datos que van a cambiar
  - Cada vez que se utilice, los argumentos necesarios serán sustituidos en los lugares correspondientes.
  - Los parámetros se especifican con el carácter '?'.

- El objeto **java.sql.PreparedStatement** (sentencia preparada) se obtiene a partir de una instancia de java.sql.Connection.

5. Procesado de consultas preparadas

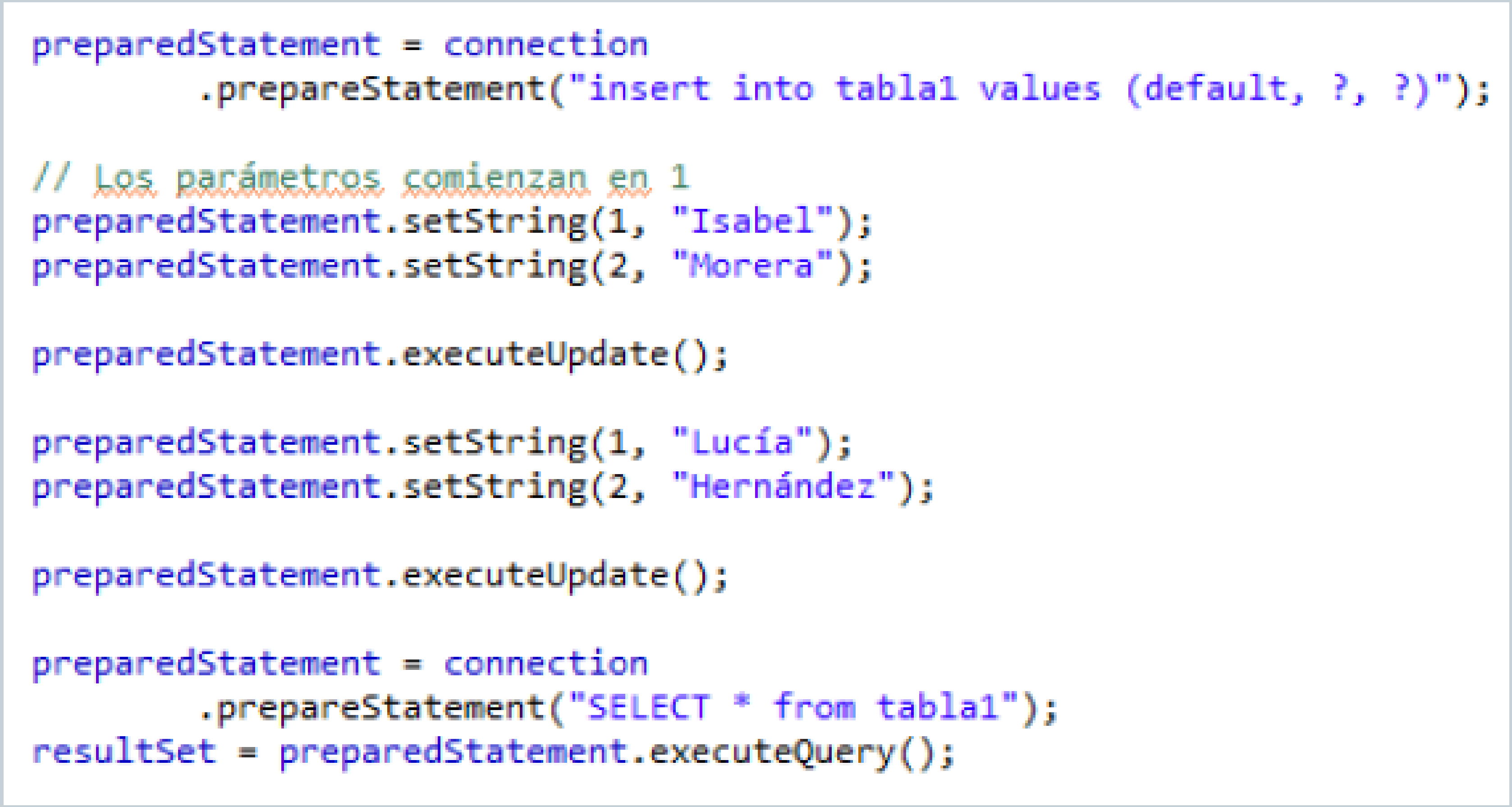
- Ejemplo:



```
public static void ejecutarConsultaPreparada() throws SQLException{
    PreparedStatement ps =
            connection.prepareStatement("insert into tabla1 values (?)");

    //Los parámetros empiezan en 1
    ps.setInt(1, 20);
    ps.executeUpdate();
    ps.setInt(1, 30);
    ps.executeUpdate();
    ps= connection.prepareStatement("select * from tabla1");
    rs=ps.executeQuery();

    while (rs.next()){

        System.out.println("Valor: "+rs.getInt(1));
    }
    ps.close();

}
```

5. Procesado de consultas preparadas

- Otro ejemplo:



```
preparedStatement = connection
        .prepareStatement("insert into tabla1 values (default, ?, ?)");

// Los parámetros comienzan en 1
preparedStatement.setString(1, "Isabel");
preparedStatement.setString(2, "Morera");

preparedStatement.executeUpdate();

preparedStatement.setString(1, "Lucía");
preparedStatement.setString(2, "Hernández");

preparedStatement.executeUpdate();

preparedStatement = connection
        .prepareStatement("SELECT * from tabla1");
resultSet = preparedStatement.executeQuery();
```

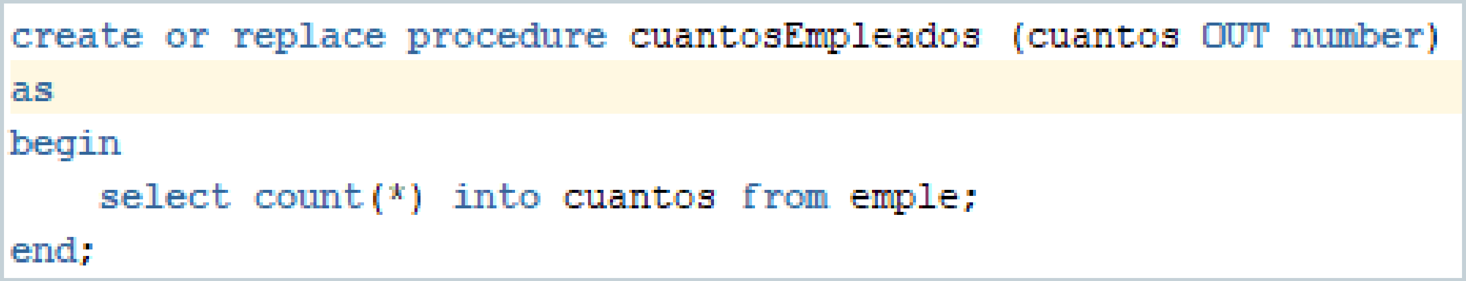
ACTIVIDAD

- Realiza los ejercicios del 5 al 8 de la Hoja de Ejercicios

6. Procesado de consultas ejecutables

- Una consulta ejecutable es aquella en la que se llama a un procedimiento almacenado (procedure) en la BD.

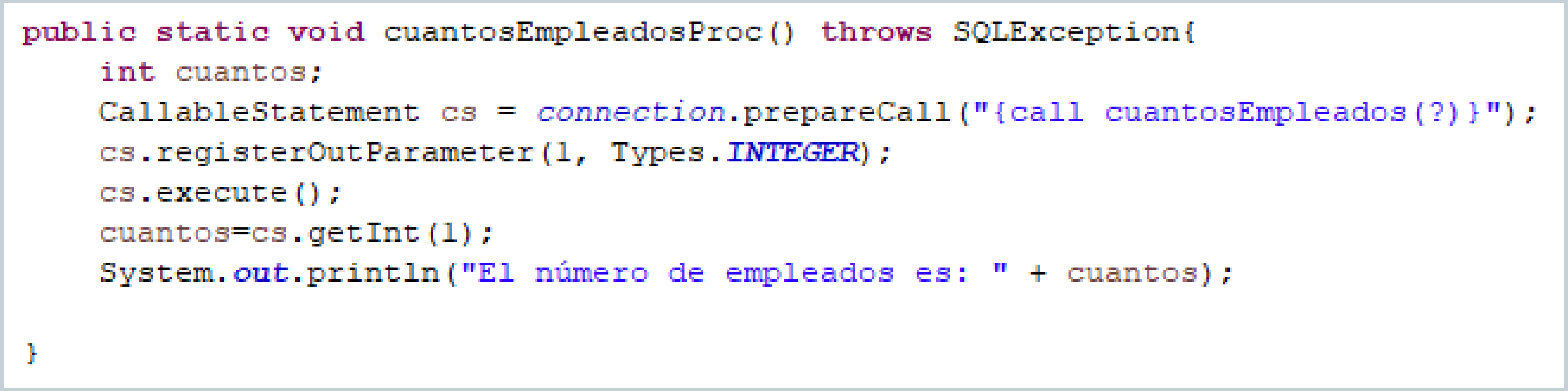
- Dado el siguiente procedimiento, creado previamente en la BD a la que se ha hecho conexión, con un parámetro de salida (OUT) que devuelve el número de filas de la tabla empleados:



```
create or replace procedure cuantosEmpleados (cuantos OUT number)
as
begin
    select count(*) into cuantos from emple;
end;
```

6. Procesado de consultas ejecutables

- El programa Java que ejecute dicho procedimiento deberá recoger el valor de salida:



```
public static void cuantosEmpleadosProc() throws SQLException{
    int cuantos;
    CallableStatement cs = connection.prepareCall("{call cuantosEmpleados(?)}");
    cs.registerOutParameter(1, Types.INTEGER);
    cs.execute();
    cuantos=cs.getInt(1);
    System.out.println("El número de empleados es: " + cuantos);

}
```

- Llamada a un procedimiento SIN parámetros:

  cs = connection.prepareCall("{call nomProc}");

6. Procesado de consultas ejecutables

- Llamada a un procedimiento con un parámetro OUT:

```
cs = connection.prepareCall("{call  nomProc(?)}");

//Hay que registrar el parámetro OUT con su tipo
//Se puede registrar o indicando el número de parámetro o mediante un nombre de parámetro
cs.registerOutParameter(1,Types.VARCHAR);  ó
cs.registerOutParameter("valor",Types.VARCHAR);

//Ejecutar el procedimiento  y recuperar el parámetro
cs.execute();
cs.getString(1); ó  cs.getString("valor");
```

6. Procesado de consultas ejecutables

- Llamada a un procedimiento con un parámetro IN:

```
cs = connection.prepareCall("{call  nomProc(?)}");

//Hay que actualizar el valor del parámetro IN
cs.setString(1,"HOLA");

 //Ejecutar el procedimiento
cs.execute();
```

6. Procesado de consultas ejecutables

- Llamada a un procedimiento con un parámetro IN/OUT:

```
cs = connection.prepareCall("{call  nomProc(?)}");

//Hay que registrar el parámetro IN/OUT con su tipo
cs.registerOutParameter(1,Types.VARCHAR);

//Hay que actualizar el valor del parámetro IN
cs.setString(1,"HOLA");

 //Ejecutar el procedimiento  y recuperar el parámetro
cs.execute();
cs.getString(1);
```

6. Procesado de consultas ejecutables

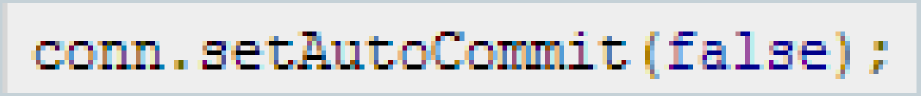
- Otro ejemplo: https://www.mkyong.com/jdbc/jdbc-callablestatement-stored-procedure-out-parameter-example/

7. Procesado de transacciones

- En ocasiones se necesita que las operaciones se ejecuten en bloque (todas o ninguna) para evitar estados inconsistentes de la BD.
- En los ejemplos anteriores los cambios se producen cuando la sentencia (borrar, actualizar, insertar) se ejecuta.
  - No hace falta ejecutar una orden para actualizar cambios en la BD.
- Esto es así porque está habilitado el modo **auto-commit** en la conexión con la BD.

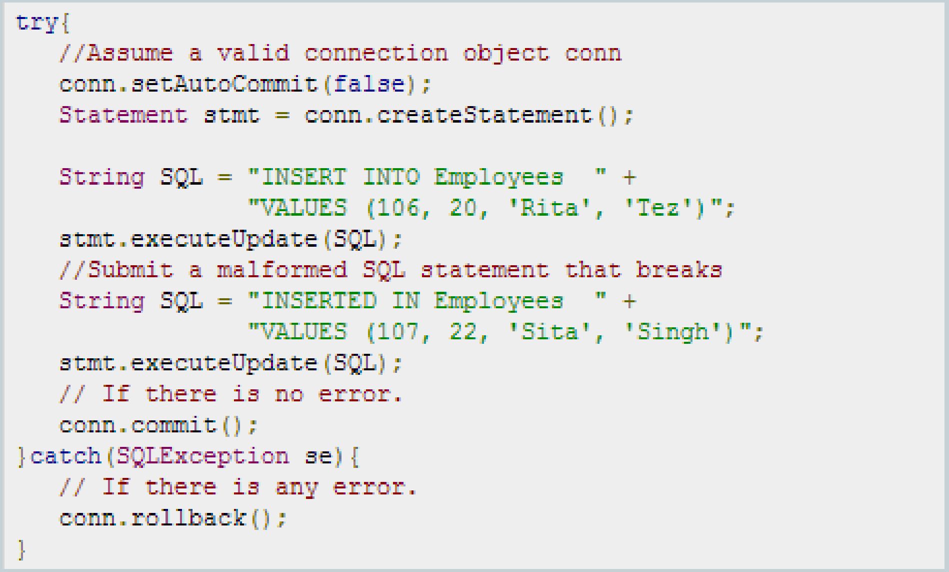
7. Procesado de transacciones

- Para deshabilitar el modo auto-commit en la base de datos hay que ejecutar la siguiente sentencia:



```
conn.setAutoCommit(false);
```

- Hecho esto, es posible ejecutar "virtualmente" una serie de consultas, y hacerlas efectivas al final con la llamada a **Connection.commit();**



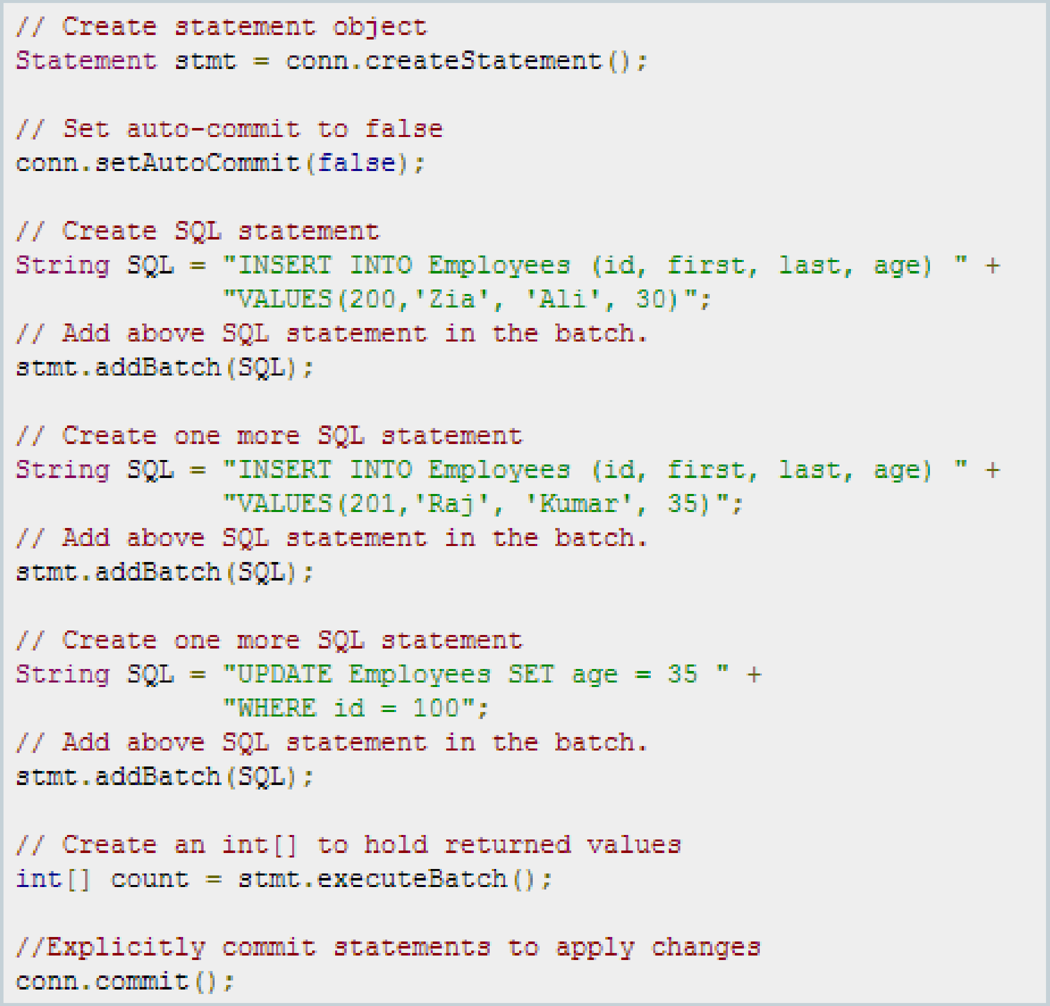
```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();

   String SQL = "INSERT INTO Employees  " +
                "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees  " +
                "VALUES (107, 22, 'Sita', 'Singh')";
   stmt.executeUpdate(SQL);
   // If there is no error.
   conn.commit();
}catch(SQLException se){
   // If there is any error.
   conn.rollback();
}
```

7. Procesado de transacciones

- Si ha ocurrido una excepción, se pueden deshacer los cambios con **Connection.rollback();**

- Es posible procesar consultas por lotes (batch).
  - Para ello, se añaden consultas secuencialmente a un mismo objeto Statement mediante el método **addBatch().**
  - Cuando todas las consultas deseadas se han añadido, se ejecutan con **commit().**

7. Procesado de transacciones



```
// Create statement object
Statement stmt = conn.createStatement();

// Set auto-commit to false
conn.setAutoCommit(false);

// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
             "VALUES(200,'Zia', 'Ali', 30)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
             "VALUES(201,'Raj', 'Kumar', 35)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "UPDATE Employees SET age = 35 " +
             "WHERE id = 100";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create an int[] to hold returned values
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```

Ventajas e Inconvenientes de Usar Conectores JDBC

2º DAM - Acceso a Datos

¿Qué es un Conector JDBC?

- Un conector JDBC (driver) es una librería que permite que Java se comunique con una base de datos.
- Actúa como intermediario entre la aplicación y el SGBD.
- Necesario para enviar consultas SQL desde Java.
- Cada SGBD posee su driver JDBC específico.

Ventajas

- **Independencia del gestor de base de datos.** El programa se escribe usando la API estándar JDBC.Si cambiamos de gestor, solo cambia el driver, no el código (o muy poco).→ Facilita portabilidad.
- **API JDBC estandarizada y fácil de aplicar**. La forma de conectar, insertar, consultar, etc., es **la misma en cualquier base de datos**:
- **PreparedStatement mejora la seguridad y evita inyecciones SQL**. Los conectores permiten usar **sentencias preparadas**, que evitan inyecciones SQL.
- **Reutilización y precompilación de consultas (mejor rendimiento).** Las consultas preparadas se precompilan y se pueden reutilizar.→ Rápido cuando se repite la misma consulta muchas veces.
- **Drivers disponibles y con amplia documentación**. Todos los SGBD grandes ofrecen drivers oficiales o compatibles.→ Si hay problemas, es fácil encontrar ayuda.

Inconvenientes

- **Dependencia de versiones concretas del driver.** Si el driver es incompatible o está desactualizado: la aplicación puede dar errores, puede no funcionar con nuevas versiones del SGBD.
- **Requiere configuración inicial del proyecto** (classpath o gestionado por Maven).
- **Diferencias entre dialectos SQL según el SGBD.** Aunque JDBC es estándar, cada base de datos tiene diferencias en SQL: tipos de datos , funciones, sentencias avanzadas→ A veces hay que adaptar consultas cuando se cambia de SGBD.
- **Necesidad de gestionar manualmente el cierre de conexiones** correctamente.
- **No soluciona problemas derivados de un mal diseño de la BBDD.** Usar JDBC no evita: malas estructuras de tablas, redundancia de datos, falta de índices…→ El diseño de base de datos sigue siendo **responsabilidad del desarrollador**.

Comparativa Ventajas vs

Ventajas	Inconvenientes
Independencia del gestor	Dependencia del driver
Código unificado	Configuración inicial necesaria
Mayor seguridad	Diferencias SQL entre SGBD
Mayor rendimiento	Riesgo de no cerrar conexiones
Buena documentación	No soluciona mal diseño de BD

SQLite con java

1. Inclusión del driver JDBC en Eclipse

- Descargar el **driver JDBC para SQLite**: sqlite-jdbc-x.x.x.jar (disponible en Maven/Repositorios oficiales).
- En Eclipse:
    1. Botón derecho sobre el proyecto → **Properties**.
    2. Ir a **Java Build Path** → **Libraries**.
    3. Seleccionar **Add External JARs…**.
    4. Añadir el fichero sqlite-jdbc.jar.
- El jar debe estar disponible en el classpath para que la aplicación pueda ejecutarse.

2. Conexión y desconexión

El proceso sigue la lógica de JDBC, pero con la **URL de conexión** y el **nombre del driver** específicos de SQLite.

Conexión

Para SQLite, la base de datos es un **archivo**, por lo que la URL apunta a su ubicación.

- **Driver:** org.sqlite.JDBC
- **URL de Conexión:** jdbc:sqlite:/ruta/al/archivo/nombre_bd.db (La ruta puede variar según el sistema operativo y si es absoluta o relativa).
    - SQLite **no requiere usuario ni contraseña**.

Desconexión

- Es una **buena práctica** cerrar todos los recursos (ResultSet, Statement o PreparedStatement, y Connection) para liberar memoria.

3. Operaciones CRUD (Altas, Bajas, Modificaciones y Consultas)

- Se utilizan objetos **Statement** o **PreparedStatement** (recomendado para seguridad y eficiencia). Recordatorio:

Uso de Statement

Se usa el método **executeQuery()**, que devuelve un objeto **ResultSet**.

Uso de PreparedStatement

Se usa el método **executeUpdate()**, que es ideal para estas operaciones y devuelve el número de filas afectadas.

Small. Fast. Reliable.
Choose any three.

Home Menu About Documentation Download License Support Purchase

Search

Datatypes In SQLite

▸ Table Of Contents

1. Datatypes In SQLite

Most SQL database engines (every SQL database engine other than SQLite, as far as we know) uses static, rigid typing. With static typing, the datatype of a value is determined by its container - the particular column in which the value is stored.

SQLite uses a more general dynamic type system. In SQLite, the datatype of a value is associated with the value itself, not with its container. The dynamic type system of SQLite is backwards compatible with the more common static type systems of other database engines in the sense that SQL statements that work on statically typed databases work the same way in SQLite. However, the dynamic typing in SQLite allows it to do things which are not possible in traditional rigidly typed databases. Flexible typing is a feature of SQLite, not a bug.

Update: As of version 3.37.0 (2021-11-27), SQLite provides STRICT tables that do rigid type enforcement, for developers who prefer that kind of thing.

2. Storage Classes and Datatypes

Each value stored in an SQLite database (or manipulated by the database engine) has one of the following storage classes:

- **NULL**. The value is a NULL value.
- **INTEGER**. The value is a signed integer, stored in 0, 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
- **REAL**. The value is a floating point value, stored as an 8-byte IEEE floating point number.
- **TEXT**. The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
- **BLOB**. The value is a blob of data, stored exactly as it was input.

A storage class is more general than a datatype. The INTEGER storage class, for example, includes 7 different integer datatypes of different lengths. This makes a difference on disk. But as soon as INTEGER values are read off of disk and into memory for processing, they are converted to the most general datatype (8-byte signed integer). And so for the most part, "storage class" is indistinguishable from "datatype" and the two terms can be used interchangeably.

Any column in an SQLite version 3 database, except an INTEGER PRIMARY KEY column, may be used to store a value of any storage class.

All values in SQL statements, whether they are literals embedded in SQL statement text or parameters bound to precompiled SQL statements have an implicit storage class. Under circumstances described below, the database engine may convert values between numeric storage classes (INTEGER and REAL) and TEXT during query execution.

2.1. Boolean Datatype

SQLite does not have a separate Boolean storage class. Instead, Boolean values are stored as integers 0 (false) and 1 (true).

SQLite recognizes the keywords "TRUE" and "FALSE", as of version 3.23.0 (2018-04-02) but those keywords are really just alternative spellings for the integer literals 1 and 0 respectively.

2.2. Date and Time Datatype

SQLite does not have a storage class set aside for storing dates and/or times. Instead, the built-in Date And Time Functions of SQLite are capable of storing dates and times as TEXT, REAL, or INTEGER values:

- **TEXT** as ISO8601 strings ("YYYY-MM-DD HH:MM:SS.SSS").
- **REAL** as Julian day numbers, the number of days since noon in Greenwich on November 24, 4714 B.C. according to the proleptic Gregorian calendar.

- **INTEGER** as Unix Time, the number of seconds since 1970-01-01 00:00:00 UTC.

Applications can choose to store dates and times in any of these formats and freely convert between formats using the built-in date and time functions.

3. Type Affinity

SQL database engines that use rigid typing will usually try to automatically convert values to the appropriate datatype. Consider this:

```
CREATE TABLE t1(a INT, b VARCHAR(10));
INSERT INTO t1(a,b) VALUES('123',456);
```

Rigidly-typed database will convert the string '123' into an integer 123 and the integer 456 into a string '456' prior to doing the insert.

In order to maximize compatibility between SQLite and other database engines, and so that the example above will work on SQLite as it does on other SQL database engines, SQLite supports the concept of "type affinity" on columns. The type affinity of a column is the recommended type for data stored in that column. The important idea here is that the type is recommended, not required. Any column can still store any type of data. It is just that some columns, given the choice, will prefer to use one storage class over another. The preferred storage class for a column is called its "affinity".

Each column in an SQLite 3 database is assigned one of the following type affinities:

- TEXT
- NUMERIC
- INTEGER
- REAL
- BLOB

(Historical note: The "BLOB" type affinity used to be called "NONE". But that term was easy to confuse with "no affinity" and so it was renamed.)

A column with TEXT affinity stores all data using storage classes NULL, TEXT or BLOB. If numerical data is inserted into a column with TEXT affinity it is converted into text form before being stored.

A column with NUMERIC affinity may contain values using all five storage classes. When text data is inserted into a NUMERIC column, the storage class of the text is converted to INTEGER or REAL (in order of preference) if the text is a well-formed integer or real literal, respectively. If the TEXT value is a well-formed integer literal that is too large to fit in a 64-bit signed integer, it is converted to REAL. For conversions between TEXT and REAL storage classes, only the first 15 significant decimal digits of the number are preserved. If the TEXT value is not a well-formed integer or real literal, then the value is stored as TEXT. For the purposes of this paragraph, hexadecimal integer literals are not considered well-formed and are stored as TEXT. (This is done for historical compatibility with versions of SQLite prior to version 3.8.6 2014-08-15 where hexadecimal integer literals were first introduced into SQLite.) If a floating point value that can be represented exactly as an integer is inserted into a column with NUMERIC affinity, the value is converted into an integer. No attempt is made to convert NULL or BLOB values.

A string might look like a floating-point literal with a decimal point and/or exponent notation but as long as the value can be expressed as an integer, the NUMERIC affinity will convert it into an integer. Hence, the string '3.0e+5' is stored in a column with NUMERIC affinity as the integer 300000, not as the floating point value 300000.0.

A column that uses INTEGER affinity behaves the same as a column with NUMERIC affinity. The difference between INTEGER and NUMERIC affinity is only evident in a CAST expression: The expression "CAST(4.0 AS INT)" returns an integer 4, whereas "CAST(4.0 AS NUMERIC)" leaves the value as a floating-point 4.0.

A column with REAL affinity behaves like a column with NUMERIC affinity except that it forces integer values into floating point representation. (As an internal optimization, small floating point values with no fractional component and stored in columns with REAL affinity are written to disk as integers in order to take up less space and are automatically converted back into floating point as the value is read out. This optimization is completely invisible at the SQL level and can only be detected by examining the raw bits of the database file.)

A column with affinity BLOB does not prefer one storage class over another and no attempt is made to coerce data from one storage class into another.

3.1. Determination Of Column Affinity

For tables not declared as STRICT, the affinity of a column is determined by the declared type of the column, according to the following rules in the order shown:

1. If the declared type contains the string "INT" then it is assigned INTEGER affinity.
2. If the declared type of the column contains any of the strings "CHAR", "CLOB", or "TEXT" then that column has TEXT affinity. Notice that the type VARCHAR contains the string "CHAR" and is thus assigned TEXT affinity.
3. If the declared type for a column contains the string "BLOB" or if no type is specified then the column has affinity BLOB.
4. If the declared type for a column contains any of the strings "REAL", "FLOA", or "DOUB" then the column has REAL affinity.
5. Otherwise, the affinity is NUMERIC.

Note that the order of the rules for determining column affinity is important. A column whose declared type is "CHARINT" will match both rules 1 and 2 but the first rule takes precedence and so the column affinity will be INTEGER.

3.1.1. Affinity Name Examples

The following table shows how many common datatype names from more traditional SQL implementations are converted into affinities by the five rules of the previous section. This table shows only a small subset of the datatype names that SQLite will accept. Note that numeric arguments in parentheses that following the type name (ex: "VARCHAR(255)") are ignored by SQLite - SQLite does not impose any length restrictions (other than the large global SQLITE_MAX_LENGTH limit) on the length of strings, BLOBs or numeric values.

Example Typenames From The CREATE TABLE Statement or CAST Expression	Resulting Affinity	Rule Used To Determine Affinity
INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT UNSIGNED BIG INT INT2 INT8	INTEGER	1

CHARACTER(20)
VARCHAR(255)
VARYING CHARACTER(255)
NCHAR(55)
NATIVE CHARACTER(70)
NVARCHAR(100)
TEXT
CLOB
	TEXT	2

BLOB *no datatype specified*	BLOB	3
REAL DOUBLE DOUBLE PRECISION FLOAT	REAL	4
NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME	NUMERIC	5

Note that a declared type of "FLOATING POINT" would give INTEGER affinity, not REAL affinity, due to the "INT" at the end of "POINT". And the declared type of "STRING" has an affinity of NUMERIC, not TEXT.

3.2. Affinity Of Expressions

Every table column has a type affinity (one of BLOB, TEXT, INTEGER, REAL, or NUMERIC) but expressions do not necessarily have an affinity.

Expression affinity is determined by the following rules:

- The right-hand operand of an IN or NOT IN operator has no affinity if the operand is a list, or has the same affinity as the affinity of the result set expression if the operand is a SELECT.
- When an expression is a simple reference to a column of a real table (not a VIEW or

subquery) then the expression has the same affinity as the table column.

    - Parentheses around the column name are ignored. Hence if X and Y.Z are column names, then (X) and (Y.Z) are also considered column names and have the affinity of the corresponding columns.

    - Any operators applied to column names, including the no-op unary "+" operator, convert the column name into an expression which always has no affinity. Hence even if X and Y.Z are column names, the expressions +X and +Y.Z are not column names and have no affinity.

- An expression of the form "CAST(*expr* AS *type*)" has an affinity that is the same as a column with a declared type of "*type*".

- A COLLATE operator has the same affinity as its left-hand side operand.

- Otherwise, an expression has no affinity.

3.3. Column Affinity For Views And Subqueries

The "columns" of a VIEW or FROM-clause subquery are really the expressions in the result set of the SELECT statement that implements the VIEW or subquery. Thus, the affinity for columns of a VIEW or subquery are determined by the expression affinity rules above. Consider an example:

```
CREATE TABLE t1(a INT, b TEXT, c REAL);
CREATE VIEW v1(x,y,z) AS SELECT b, a+c, 42 FROM t1 WHERE b!=11;
```

The affinity of the v1.x column will be the same as the affinity of t1.b (TEXT), since v1.x maps directly into t1.b. But columns v1.y and v1.z both have no affinity, since those columns map into expression a+c and 42, and expressions always have no affinity.

3.3.1. Column Affinity For Compound Views

When the SELECT statement that implements a VIEW or FROM-clause subquery is a compound SELECT then the affinity of each column of the VIEW or subquery will be the affinity of the corresponding result column for one of the individual SELECT statements that make up the compound. However, it is indeterminate which of the SELECT statements will be used to determine affinity. Different constituent SELECT statements might be used to determine affinity at different times during query evaluation. The choice might vary across different versions of SQLite. The choice might change between one query and the next in the same version of SQLite. The choice might be different at different times within the same query. Hence, you can never be sure what affinity will be used for columns of a compound SELECT that have different affinities in the constituent subqueries.

Best practice is to avoid mixing affinities in a compound SELECT if you care about the datatype of the result. Mixing affinities in a compound SELECT can lead to surprising and unintuitive results. See, for example, forum post 02d7be94d7.

3.4. Column Affinity Behavior Example

The following SQL demonstrates how SQLite uses column affinity to do type conversions when values are inserted into a table.

```
CREATE TABLE t1(
    t  TEXT,     -- text affinity by rule 2
    nu NUMERIC,  -- numeric affinity by rule 5
    i  INTEGER,  -- integer affinity by rule 1
    r  REAL,     -- real affinity by rule 4
    no BLOB      -- no affinity by rule 3
);

-- Values stored as TEXT, INTEGER, INTEGER, REAL, TEXT.
INSERT INTO t1 VALUES('500.0', '500.0', '500.0', '500.0', '500.0');
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;
text|integer|integer|real|text

-- Values stored as TEXT, INTEGER, INTEGER, REAL, REAL.
DELETE FROM t1;
INSERT INTO t1 VALUES(500.0, 500.0, 500.0, 500.0, 500.0);
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;
text|integer|integer|real|real

-- Values stored as TEXT, INTEGER, INTEGER, REAL, INTEGER.
DELETE FROM t1;
INSERT INTO t1 VALUES(500, 500, 500, 500, 500);
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;
text|integer|integer|real|integer

-- BLOBs are always stored as BLOBs regardless of column affinity.
DELETE FROM t1;
INSERT INTO t1 VALUES(x'0500', x'0500', x'0500', x'0500', x'0500');
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;
blob|blob|blob|blob|blob

-- NULLs are also unaffected by affinity
DELETE FROM t1;
INSERT INTO t1 VALUES(NULL,NULL,NULL,NULL,NULL);
SELECT typeof(t), typeof(nu), typeof(i), typeof(r), typeof(no) FROM t1;
null|null|null|null|null
```

4. Comparison Expressions

SQLite version 3 has the usual set of SQL comparison operators including "=", "==", "<", "<=", ">", ">=", "!=", "", "IN", "NOT IN", "BETWEEN", "IS", and "IS NOT", .

4.1. Sort Order

The results of a comparison depend on the storage classes of the operands, according to the following rules:

- A value with storage class NULL is considered less than any other value (including another value with storage class NULL).
- An INTEGER or REAL value is less than any TEXT or BLOB value. When an INTEGER or REAL is compared to another INTEGER or REAL, a numerical comparison is performed.
- A TEXT value is less than a BLOB value. When two TEXT values are compared an appropriate collating sequence is used to determine the result.
- When two BLOB values are compared, the result is determined using memcmp().

4.2. Type Conversions Prior To Comparison

SQLite may attempt to convert values between the storage classes INTEGER, REAL, and/or TEXT before performing a comparison. Whether or not any conversions are attempted before the comparison takes place depends on the type affinity of the operands.

To "apply affinity" means to convert an operand to a particular storage class if and only if the conversion does not lose essential information. Numeric values can always be converted into TEXT. TEXT values can be converted into numeric values if the text content is a well-formed integer or real literal, but not a hexadecimal integer literal. BLOB values are converted into TEXT values by simply interpreting the binary BLOB content as a text string in the current database encoding.

Affinity is applied to operands of a comparison operator prior to the comparison according to the following rules in the order shown:

- If one operand has INTEGER, REAL or NUMERIC affinity and the other operand has TEXT or BLOB or no affinity then NUMERIC affinity is applied to the other operand.
- If one operand has TEXT affinity and the other has no affinity, then TEXT affinity is applied to the other operand.
- Otherwise, no affinity is applied and both operands are compared as is.

The expression "a BETWEEN b AND c" is treated as two separate binary comparisons "a >= b AND a <= c", even if that means different affinities are applied to 'a' in each of the comparisons. Datatype conversions in comparisons of the form "x IN (SELECT y ...)" are handled as if the comparison were really "x=y". The expression "a IN (x, y, z, ...)" is equivalent to "a = +x OR a = +y OR a = +z OR ...". In other words, the values to the right of the IN operator (the "x", "y", and "z" values in this example) are considered to have no affinity, even if they happen to be column values or CAST expressions.

4.3. Comparison Example

```
CREATE TABLE t1(
    a TEXT,      -- text affinity
    b NUMERIC,   -- numeric affinity
    c BLOB,      -- no affinity
    d            -- no affinity
);

-- Values will be stored as TEXT, INTEGER, TEXT, and INTEGER respectively
INSERT INTO t1 VALUES('500', '500', '500', 500);
SELECT typeof(a), typeof(b), typeof(c), typeof(d) FROM t1;
text|integer|text|integer

-- Because column "a" has text affinity, numeric values on the
-- right-hand side of the comparisons are converted to text before
-- the comparison occurs.
SELECT a < 40,   a < 60,   a < 600 FROM t1;
0|1|1

-- Text affinity is applied to the right-hand operands but since
-- they are already TEXT this is a no-op; no conversions occur.
SELECT a < '40', a < '60', a < '600' FROM t1;
0|1|1
```

```
-- Column "b" has numeric affinity and so numeric affinity is applied
-- to the operands on the right.  Since the operands are already numeric,
-- the application of affinity is a no-op; no conversions occur.  All
-- values are compared numerically.
SELECT b < 40,   b < 60,   b < 600 FROM t1;
0|0|1

-- Numeric affinity is applied to operands on the right, converting them
-- from text to integers.  Then a numeric comparison occurs.
SELECT b < '40', b < '60', b < '600' FROM t1;
0|0|1

-- No affinity conversions occur.  Right-hand side values all have
-- storage class INTEGER which are always less than the TEXT values
-- on the left.
SELECT c < 40,   c < 60,   c < 600 FROM t1;
0|0|0

-- No affinity conversions occur.  Values are compared as TEXT.
SELECT c < '40', c < '60', c < '600' FROM t1;
0|1|1

-- No affinity conversions occur.  Right-hand side values all have
-- storage class INTEGER which compare numerically with the INTEGER
-- values on the left.
SELECT d < 40,   d < 60,   d < 600 FROM t1;
0|0|1

-- No affinity conversions occur.  INTEGER values on the left are
-- always less than TEXT values on the right.
SELECT d < '40', d < '60', d < '600' FROM t1;
1|1|1
```

All of the results in the example are the same if the comparisons are commuted - if expressions of the form "a<40" are rewritten as "40>a".

5. Operators

Mathematical operators (+, -, *, /, %, <<, >>, &, and |) interpret both operands as if they were numbers. STRING or BLOB operands automatically convert into REAL or INTEGER values. If the STRING or BLOB looks like a real number (if it has a decimal point or an exponent) or if the value is outside the range that can be represented as a 64-bit signed integer, then it converts to REAL. Otherwise the operand converts to INTEGER. The implied type conversion of mathematical operands is slightly different from CAST to NUMERIC in that string and BLOB values that look like real numbers but have no fractional part are kept as REAL instead of being converted into INTEGER as they would be for CAST to NUMERIC. The conversion from STRING or BLOB into REAL or INTEGER is performed even if it is lossy and irreversible. Some mathematical operators (%, <<, >>, &, and |) expect INTEGER operands. For those operators, REAL operands are converted into INTEGER in the same way as a CAST to INTEGER. The <<, >>, &, and | operators always return an INTEGER (or NULL) result, but the % operator returns either INTEGER or REAL (or NULL) depending on the type of its operands. A NULL operand on a mathematical operator yields a NULL result. An operand on a mathematical operator that does not look in any way numeric and is not NULL is converted to 0 or 0.0. Division by zero gives a result of NULL.

6. Sorting, Grouping and Compound SELECTs

When query results are sorted by an ORDER BY clause, values with storage class NULL come first, followed by INTEGER and REAL values interspersed in numeric order, followed by TEXT values in collating sequence order, and finally BLOB values in memcmp() order. No storage class conversions occur before the sort.

When grouping values with the GROUP BY clause values with different storage classes are considered distinct, except for INTEGER and REAL values which are considered equal if they are numerically equal. No affinities are applied to any values as the result of a GROUP by clause.

The compound SELECT operators UNION, INTERSECT and EXCEPT perform implicit comparisons between values. No affinity is applied to comparison operands for the implicit comparisons associated with UNION, INTERSECT, or EXCEPT - the values are compared as is.

7. Collating Sequences

When SQLite compares two strings, it uses a collating sequence or collating function (two terms for the same thing) to determine which string is greater or if the two strings are equal. SQLite has three built-in collating functions: BINARY, NOCASE, and RTRIM.

- **BINARY** - Compares string data using memcmp(), regardless of text encoding.
- **NOCASE** - Similar to binary, except that it uses sqlite3_strnicmp() for the comparison. Hence the 26 upper case characters of ASCII are folded to their lower case equivalents before the comparison is performed. Note that only ASCII characters are case folded. SQLite does not attempt to do full UTF case folding due to the size of the tables required. Also note that any U+0000 characters in the string are considered string terminators for comparison purposes.
- **RTRIM** - The same as binary, except that trailing space characters are ignored.

An application can register additional collating functions using the sqlite3_create_collation() interface.

Collating functions only matter when comparing string values. Numeric values are always compared numerically, and BLOBs are always compared byte-by-byte using memcmp().

7.1. Assigning Collating Sequences from SQL

Every column of every table has an associated collating function. If no collating function is explicitly defined, then the collating function defaults to BINARY. The COLLATE clause of the column definition is used to define alternative collating functions for a column.

The rules for determining which collating function to use for a binary comparison operator (=, <, >, <=, >=, !=, IS, and IS NOT) are as follows:

1. If either operand has an explicit collating function assignment using the postfix COLLATE operator, then the explicit collating function is used for comparison, with precedence to the collating function of the left operand.

2. If either operand is a column, then the collating function of that column is used with precedence to the left operand. For the purposes of the previous sentence, a column name preceded by one or more unary "+" operators and/or CAST operators is still considered a column name.

3. Otherwise, the BINARY collating function is used for comparison.

An operand of a comparison is considered to have an explicit collating function assignment (rule 1 above) if any subexpression of the operand uses the postfix COLLATE operator. Thus, if a COLLATE operator is used anywhere in a comparison expression, the collating function defined by that operator is used for string comparison regardless of what table columns might be a part of that expression. If two or more COLLATE operator subexpressions appear anywhere in a comparison, the left most explicit collating function is used regardless of how deeply the COLLATE operators are nested in the expression and regardless of how the expression is parenthesized.

The expression "x BETWEEN y and z" is logically equivalent to two comparisons "x >= y AND x <= z" and works with respect to collating functions as if it were two separate comparisons. The expression "x IN (SELECT y ...)" is handled in the same way as the expression "x = y" for the purposes of determining the collating sequence. The collating sequence used for expressions of the form "x IN (y, z, ...)" is the collating sequence of x. If an explicit collating sequence is required on an IN operator it should be applied to the left operand, like this: "x COLLATE nocase IN (y,z, ...)".

Terms of the ORDER BY clause that is part of a SELECT statement may be assigned a collating sequence using the COLLATE operator, in which case the specified collating function is used for sorting. Otherwise, if the expression sorted by an ORDER BY clause is a column, then the collating sequence of the column is used to determine sort order. If the expression is not a column and has no COLLATE clause, then the BINARY collating sequence is used.

7.2. Collation Sequence Examples

The examples below identify the collating sequences that would be used to determine the results of text comparisons that may be performed by various SQL statements. Note that a text comparison may not be required, and no collating sequence used, in the case of numeric, blob or NULL values.

```
CREATE TABLE t1(
    x INTEGER PRIMARY KEY,
    a,                 /* collating sequence BINARY */
    b COLLATE BINARY,  /* collating sequence BINARY */
    c COLLATE RTRIM,   /* collating sequence RTRIM  */
    d COLLATE NOCASE   /* collating sequence NOCASE */
);
                   /* x   a     b     c       d */
INSERT INTO t1 VALUES(1,'abc','abc', 'abc  ','abc');
INSERT INTO t1 VALUES(2,'abc','abc', 'abc',  'ABC');
INSERT INTO t1 VALUES(3,'abc','abc', 'abc ', 'Abc');
INSERT INTO t1 VALUES(4,'abc','abc ','ABC',  'abc');

/* Text comparison a=b is performed using the BINARY collating sequence. */
SELECT x FROM t1 WHERE a = b ORDER BY x;
--result 1 2 3

/* Text comparison a=b is performed using the RTRIM collating sequence. */
SELECT x FROM t1 WHERE a = b COLLATE RTRIM ORDER BY x;
--result 1 2 3 4

/* Text comparison d=a is performed using the NOCASE collating sequence. */
SELECT x FROM t1 WHERE d = a ORDER BY x;
--result 1 2 3 4

/* Text comparison a=d is performed using the BINARY collating sequence. */
SELECT x FROM t1 WHERE a = d ORDER BY x;
--result 1 4
```

```
/* Text comparison 'abc'=c is performed using the RTRIM collating sequence. */
SELECT x FROM t1 WHERE 'abc' = c ORDER BY x;
--result 1 2 3

/* Text comparison c='abc' is performed using the RTRIM collating sequence. */
SELECT x FROM t1 WHERE c = 'abc' ORDER BY x;
--result 1 2 3

/* Grouping is performed using the NOCASE collating sequence (Values
** 'abc', 'ABC', and 'Abc' are placed in the same group). */
SELECT count(*) FROM t1 GROUP BY d ORDER BY 1;
--result 4

/* Grouping is performed using the BINARY collating sequence.  'abc' and
** 'ABC' and 'Abc' form different groups */
SELECT count(*) FROM t1 GROUP BY (d || '') ORDER BY 1;
--result 1 1 2

/* Sorting or column c is performed using the RTRIM collating sequence. */
SELECT x FROM t1 ORDER BY c, x;
--result 4 1 2 3

/* Sorting of (c||'') is performed using the BINARY collating sequence. */
SELECT x FROM t1 ORDER BY (c||''), x;
--result 4 2 3 1

/* Sorting of column c is performed using the NOCASE collating sequence. */
SELECT x FROM t1 ORDER BY c COLLATE NOCASE, x;
--result 2 4 3 1
```

This page last modified on 2025-05-31 13:08:22 UTC