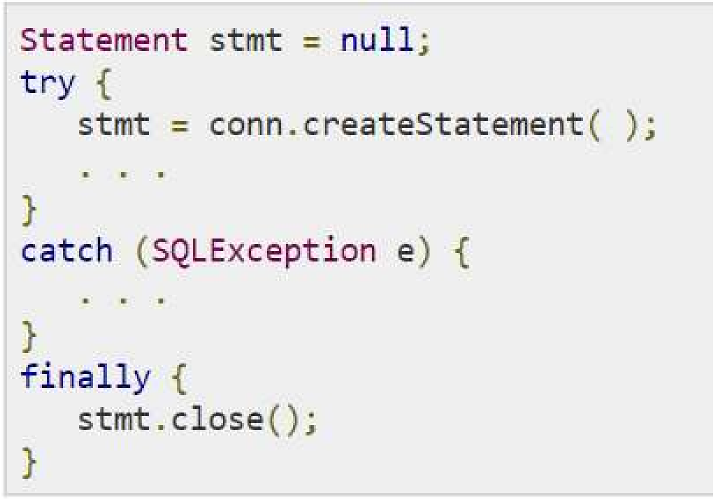
Information Systems – Design and Development

Statements, Query, Update and Transactions

Statements

- Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.
- **boolean execute (String SQL)**: Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate (String SQL)**: Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL)**: Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.
- Don't forget close()

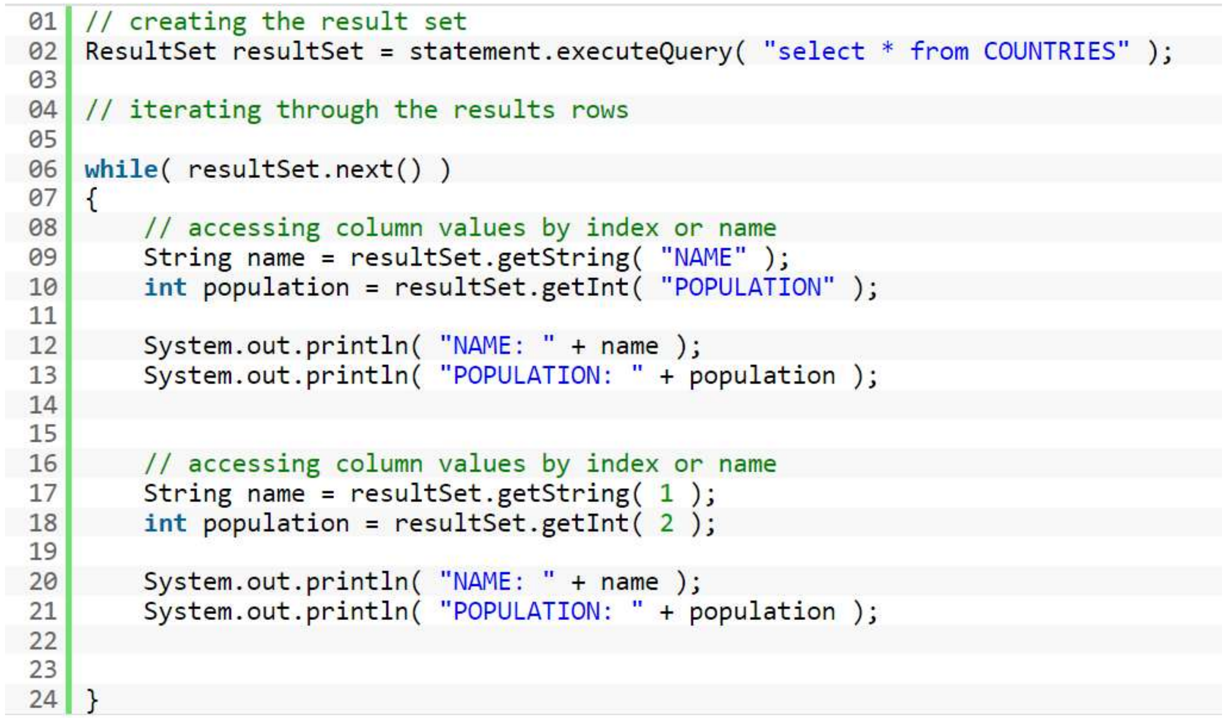


```
Statement stmt = null;
try {
   stmt = conn.createStatement( );
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   stmt.close();
}
```

https://www.tutorialspoint.com/jdbc

ResultSet

- The class java.sql.ResultSet represents a result set of database table. It is created, normally; by executing an SQL query
- It contains rows of data, where the data is stored. These data can be accessed by index (starting by 1) or by attribute name:

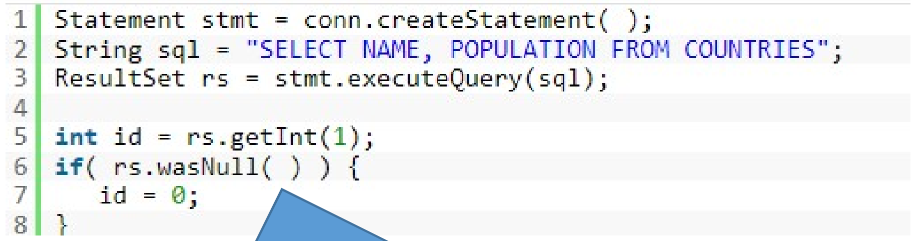


```
// creating the result set
ResultSet resultSet = statement.executeQuery( "select * from COUNTRIES" );

// iterating through the results rows

while( resultSet.next() )
{
    // accessing column values by index or name
    String name = resultSet.getString( "NAME" );
    int population = resultSet.getInt( "POPULATION" );

    System.out.println( "NAME: " + name );
    System.out.println( "POPULATION: " + population );


    // accessing column values by index or name
    String name = resultSet.getString( 1 );
    int population = resultSet.getInt( 2 );

    System.out.println( "NAME: " + name );
    System.out.println( "POPULATION: " + population );


}
```

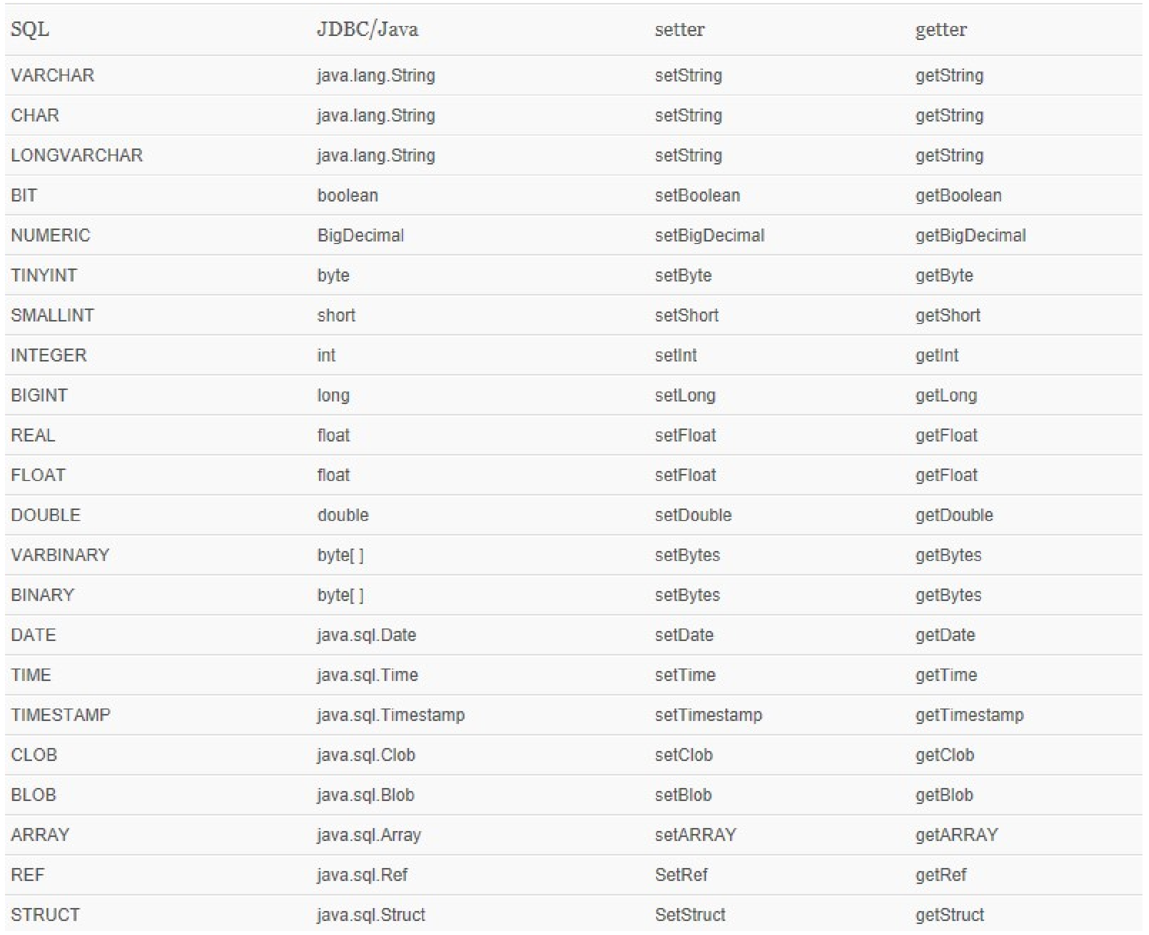
https://www.javacodegeeks.com/2015/02/jdbc-tutorial.html#resultsets

Data types

- JDBC converts the Java data types into proper JDBC types before using them in the database.



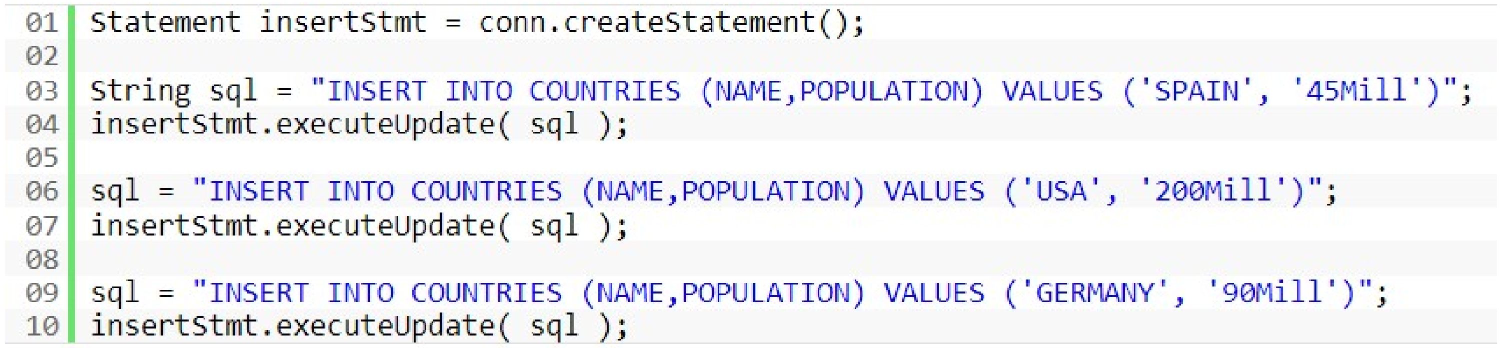
```java
Statement stmt = conn.createStatement( );
String sql = "SELECT NAME, POPULATION FROM COUNTRIES";
ResultSet rs = stmt.executeQuery(sql);

int id = rs.getInt(1);
if( rs.wasNull( ) ) {
   id = 0;
}
```



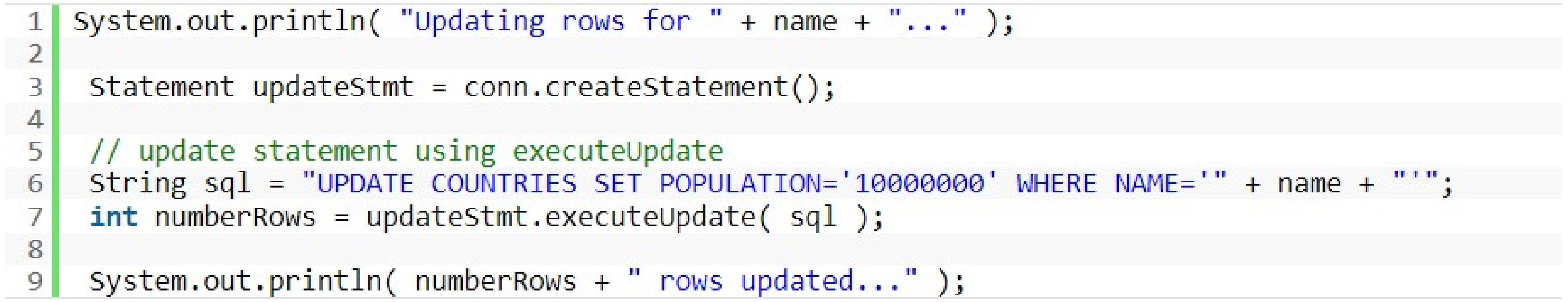
SQL	JDBC/Java	setter	getter
VARCHAR	java.lang.String	setString	getString
CHAR	java.lang.String	setString	getString
LONGVARCHAR	java.lang.String	setString	getString
BIT	boolean	setBoolean	getBoolean
NUMERIC	BigDecimal	setBigDecimal	getBigDecimal
TINYINT	byte	setByte	getByte
SMALLINT	short	setShort	getShort
INTEGER	int	setInt	getInt
BIGINT	long	setLong	getLong
REAL	float	setFloat	getFloat
FLOAT	float	setFloat	getFloat
DOUBLE	double	setDouble	getDouble
VARBINARY	byte[ ]	setBytes	getBytes
BINARY	byte[ ]	setBytes	getBytes
DATE	java.sql.Date	setDate	getDate
TIME	java.sql.Time	setTime	getTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	getTimestamp
CLOB	java.sql.Clob	setClob	getClob
BLOB	java.sql.Blob	setBlob	getBlob
ARRAY	java.sql.Array	setARRAY	getARRAY
REF	java.sql.Ref	SetRef	getRef
STRUCT	java.sql.Struct	SetStruct	getStruct

CRUD commands

- CRUD comes from Create, Read, Update and Delete



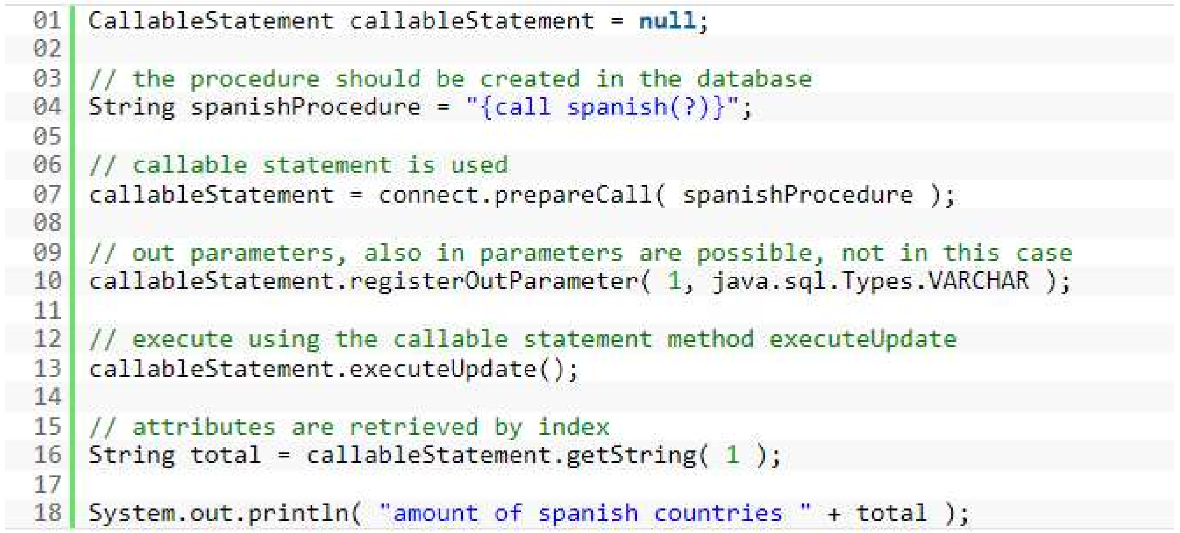
```java
Statement insertStmt = conn.createStatement();

String sql = "INSERT INTO COUNTRIES (NAME,POPULATION) VALUES ('SPAIN', '45Mill')";
insertStmt.executeUpdate( sql );

sql = "INSERT INTO COUNTRIES (NAME,POPULATION) VALUES ('USA', '200Mill')";
insertStmt.executeUpdate( sql );

sql = "INSERT INTO COUNTRIES (NAME,POPULATION) VALUES ('GERMANY', '90Mill')";
insertStmt.executeUpdate( sql );
```



```java
System.out.println( "Updating rows for " + name + "..." );

 Statement updateStmt = conn.createStatement();

 // update statement using executeUpdate
 String sql = "UPDATE COUNTRIES SET POPULATION='10000000' WHERE NAME='" + name + "'";
 int numberRows = updateStmt.executeUpdate( sql );

 System.out.println( numberRows + " rows updated..." );
```

Stored procedures – Callabled Statements

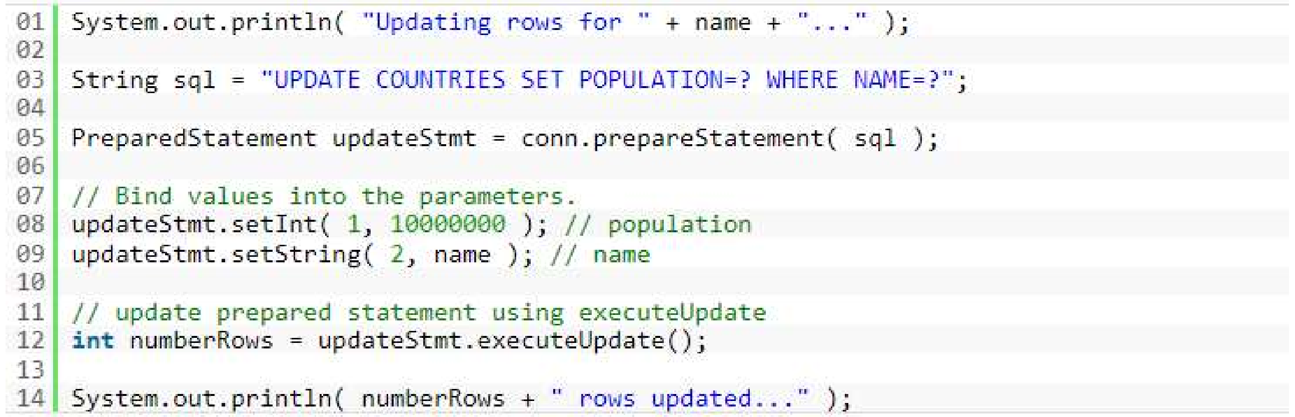
- Stored procedures are sets of SQL statements as part of a logical unit of executiion and performing a defined task. They are very useful while encapsulating a group of operations to be executed on a database.



```java
CallableStatement callableStatement = null;

// the procedure should be created in the database
String spanishProcedure = "{call spanish(?)}";

// callable statement is used
callableStatement = connect.prepareCall( spanishProcedure );

// out parameters, also in parameters are possible, not in this case
callableStatement.registerOutParameter( 1, java.sql.Types.VARCHAR );

// execute using the callable statement method executeUpdate
callableStatement.executeUpdate();

// attributes are retrieved by index
String total = callableStatement.getString( 1 );

System.out.println( "amount of spanish countries " + total );
```

Prepared Statements

- In case programmers need better efficiency when repeating SQL queries or parameterization they should use java.sql.PreparedStatement. This interface inherits the basic statement interface mentioned before and offers parameterization. Because of this functionalitiy, this interface is safer against SQL injection attacks.



```java
System.out.println( "Updating rows for " + name + "..." );

String sql = "UPDATE COUNTRIES SET POPULATION=? WHERE NAME=?";

PreparedStatement updateStmt = conn.prepareStatement( sql );

// Bind values into the parameters.
updateStmt.setInt( 1, 10000000 ); // population
updateStmt.setString( 2, name ); // name

// update prepared statement using executeUpdate
int numberRows = updateStmt.executeUpdate();

System.out.println( numberRows + " rows updated..." );
```

Transactions

- JDBC supports transactions and contains methods and functionalities to implement transaction based applications
  - java.sql.Connection.setAutoCommit(boolean): This method receives a Boolean as parameter, in case of true (which is the default behavior), all SQL statements will be persisted automatically in the database. In case of false, changes will not be persisted automatically, this will be done by using the method java.sql.Connection.commit().
  - java.sql.Connection.commit(). This method can be only used if the auto commit is set to false or disabled; that is, it only works on non automatic commit mode. When executing this method all changes since last commit / rollback will be persisted in the database.
  - java.sql.Connection.rollback(). This method can be used only when auto commit is disabled. It undoes or reverts all changes done in the current transaction.

```java
Class.forName( "com.mysql.jdbc.Driver" );
Connection connect = null;
try
{
    // connection to JDBC using mysql driver
    connect = DriverManager.getConnection( "jdbc:mysql://localhost/countries?"
                + "user=root&password=root" );
    connect.setAutoCommit( false );

    System.out.println( "Inserting row for Japan..." );
    String sql = "INSERT INTO COUNTRIES (NAME,POPULATION) VALUES ('JAPAN', '45000000')";

    PreparedStatement insertStmt = connect.prepareStatement( sql );

    // insert statement using executeUpdate
    insertStmt.executeUpdate( sql );
    connect.rollback();

    System.out.println( "Updating row for Japan..." );
    // update statement using executeUpdate -> will cause an error, update will not be
    // executed becaues the row does not exist
    sql = "UPDATE COUNTRIES SET POPULATION='1000000' WHERE NAME='JAPAN'";
    PreparedStatement updateStmt = connect.prepareStatement( sql );

    updateStmt.executeUpdate( sql );
    connect.commit();

}
catch( SQLException ex )
{
    ex.printStackTrace();
    //undoes all changes in current transaction
    connect.rollback();
}
finally
{
    connect.close();
}
```