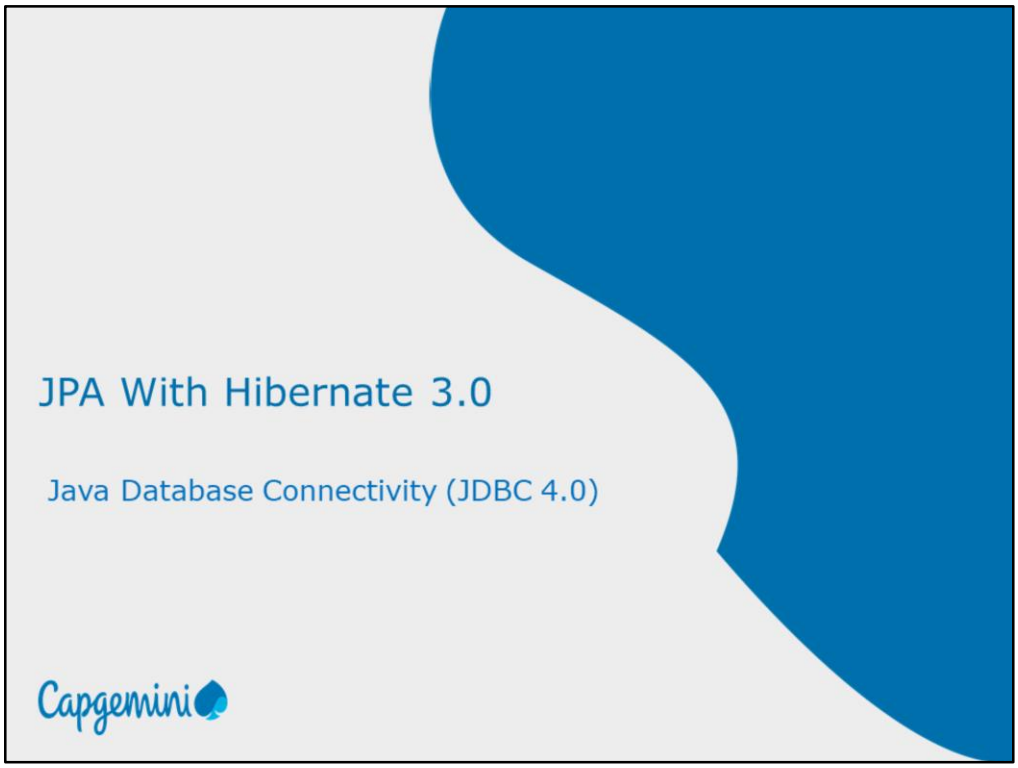

JPA With Hibernate 3.0
Java Database Connectivity (JDBC 4.0)
Capgemini

Lesson Objectives

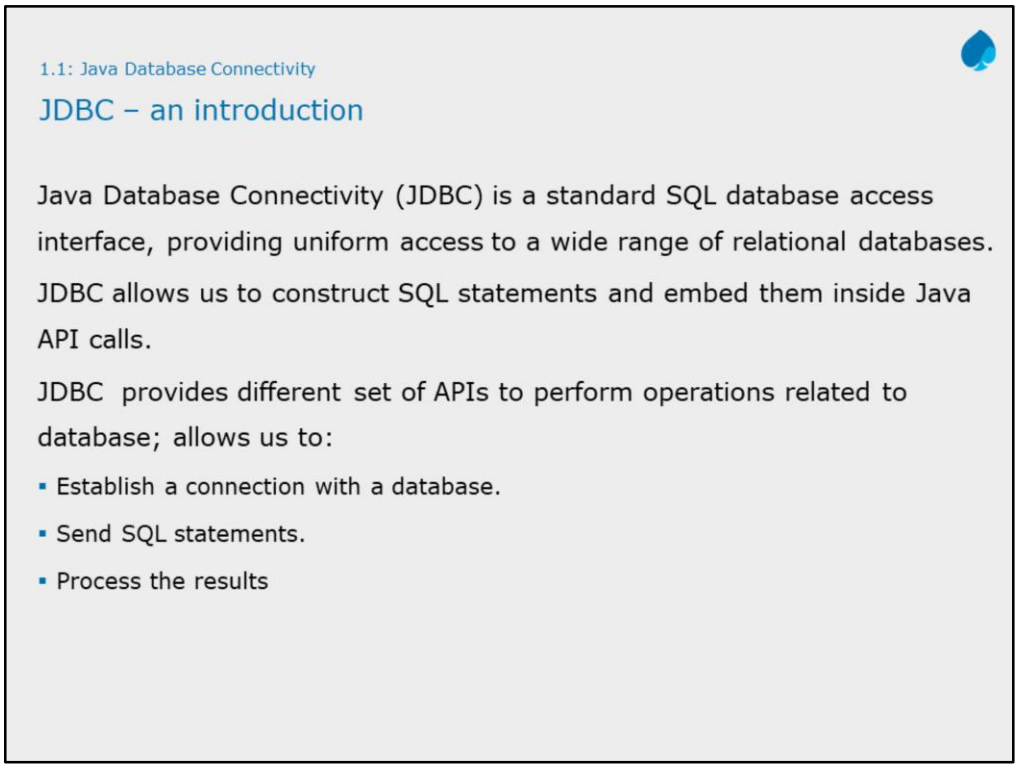
After completing this lesson, participants will be able to

- Understand concept database connectivity architecture
- Work with JDBC API 4.0
- Access database through Java programs
- Understand advance features of JDBC API

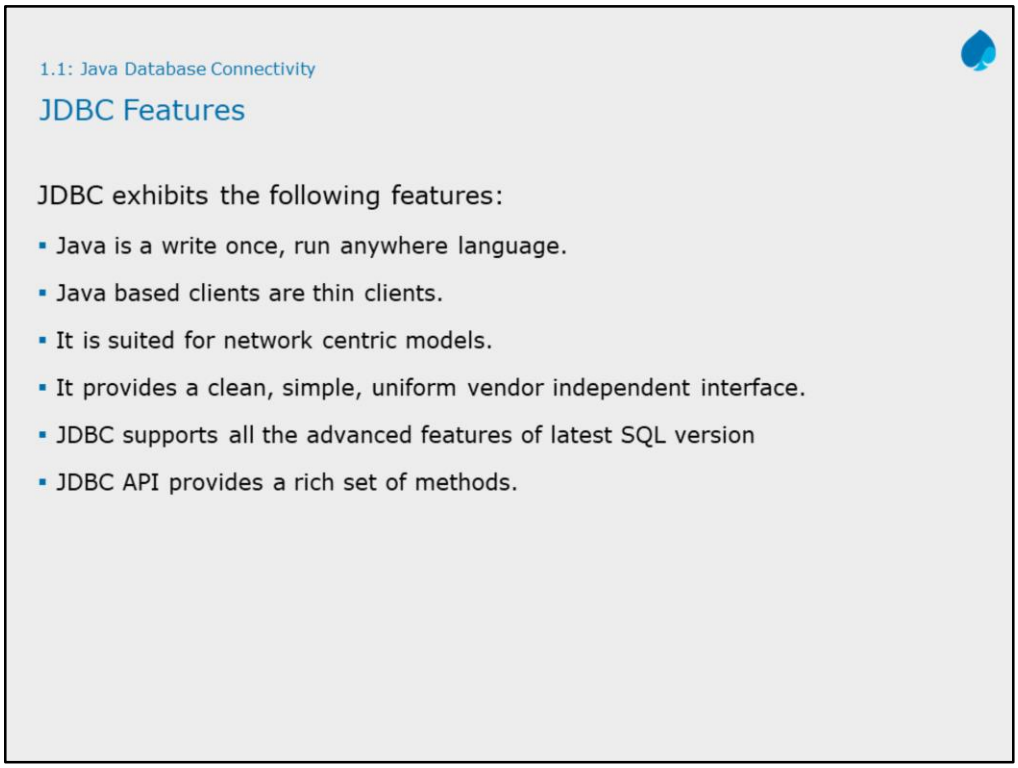
This lesson covers JDBC API, used to work with database.

Lesson outline:

1.1: Java Database Connectivity - Introduction
1.2: Database Connectivity Architecture
1.3: JDBC APIs
1.4: Database Access Steps
1.5: Calling database procedures/functions
1.6: Using Transaction
1.7: Best Practices



What is JDBC?
JDBC is used to allow Java applications to connect to the database and perform different data manipulation operations such as insertion, modification, deletion, and so on.



Why JDBC?
JDBC Features:

With JDBC technology, businesses are not locked in any proprietary architecture, and can continue to use their installed databases and access information easily – even if it is stored on different database management systems.

The combination of the Java API and the JDBC API makes application development easy and economical.

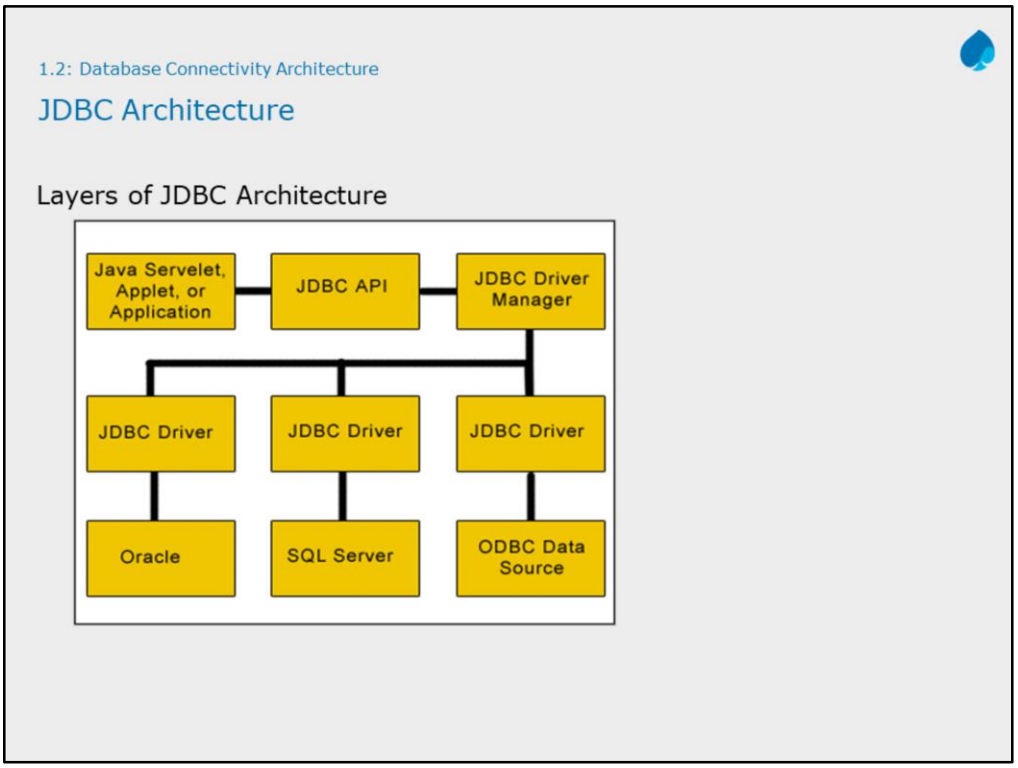
JDBC hides the complexity of many data access tasks, doing most of the “heavy lifting” for the programmer behind the scenes.

The JDBC API is simple to learn, easy to deploy, and inexpensive to maintain.

With the JDBC API, no configuration is required on the client side.

With a driver written in the Java programming language, all the information needed to make a connection is completely defined by the JDBC URL or by a DataSource object registered with a Java Naming and Directory Interface (JNDI) naming service.

Zero configuration for clients supports the network computing paradigm and centralizes software maintenance.



JDBC Architecture:

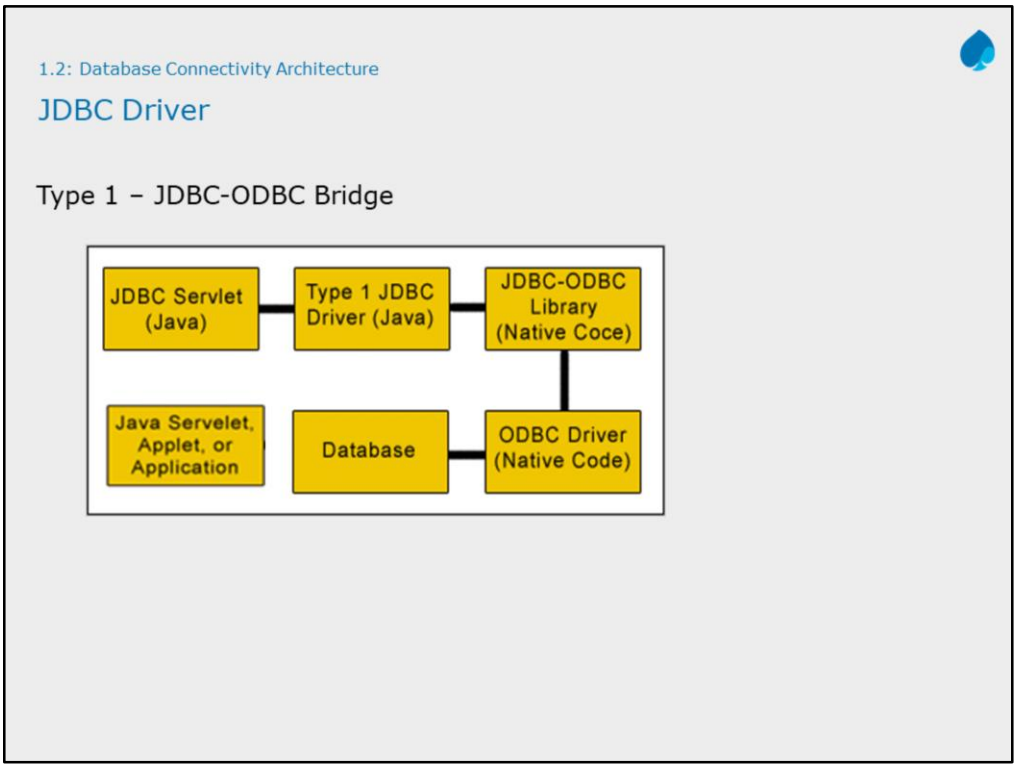
The JDBC Architecture can be classified as follows:

1. Type 1 – JDBC-ODBC Bridge
2. Type 2 – Java Native API
3. Type 3 – Java to Network Protocol
4. Type 4 – Java to Database Protocol

A JDBC driver translates standard JDBC calls into a network or database protocol or into a database library API call that facilitates communication with the database.

This translation layer provides JDBC applications with database independence.

If the back-end database changes, then only the JDBC driver needs to be replaced with few code modifications required. There are four distinct types of JDBC drivers.



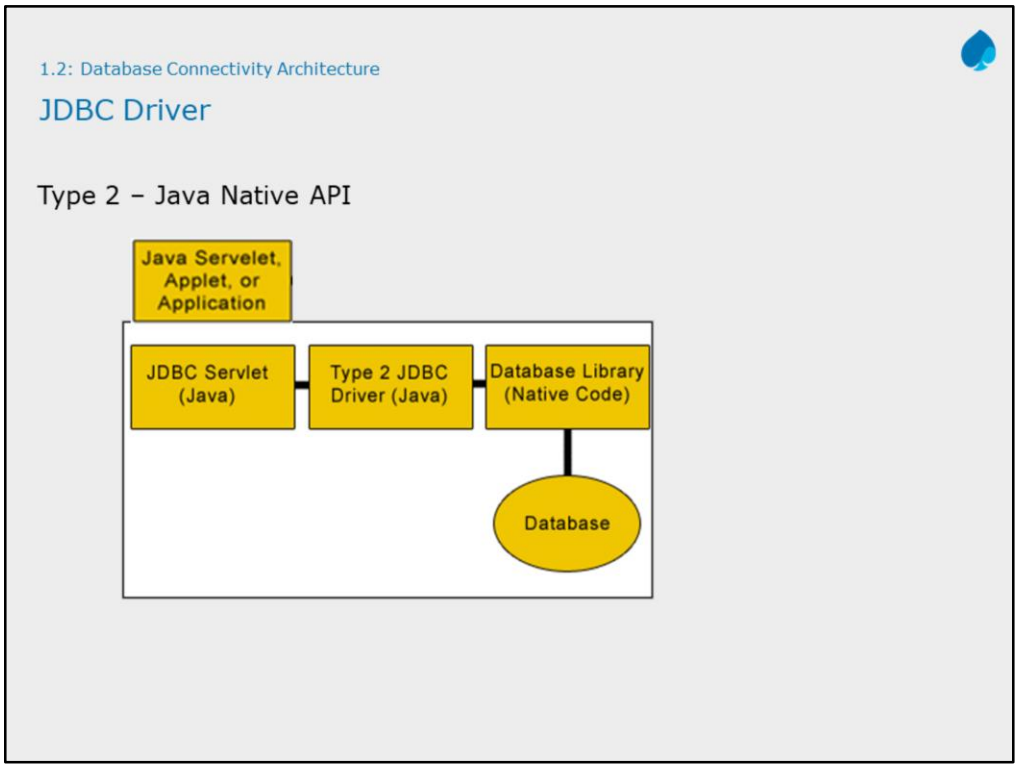
Type 1 JDBC-ODBC Bridge:

Type 1 drivers act as a "bridge" between JDBC and another database connectivity mechanism such as ODBC.

The JDBC- ODBC bridge provides JDBC access using most standard ODBC drivers.

This driver is included in the Java 2 SDK within the sun.jdbc.odbc package. In this driver the Java statements are converted to a JDBC statements.

JDBC statements call the ODBC by using the JDBC-ODBC Bridge. And finally the query is executed by the database. This driver has serious limitation for many applications.



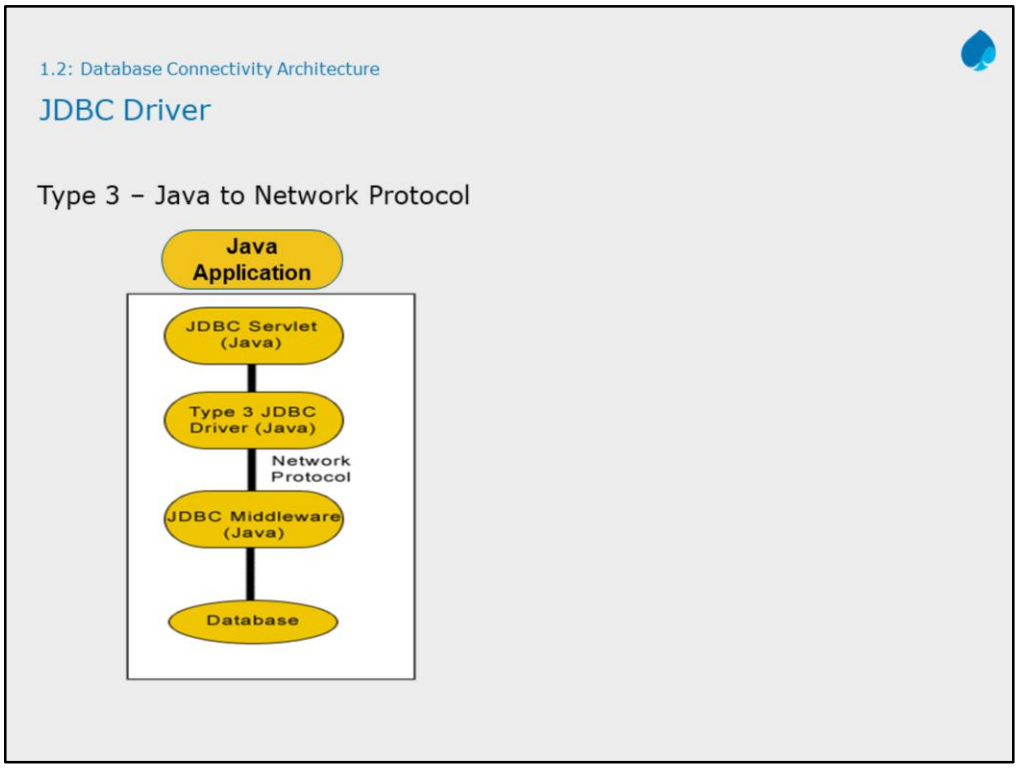
Type 2 Java to Native API:

Type 2 drivers use the Java Native Interface (JNI) to make calls to a local database library API.

This driver converts the JDBC calls into a database specific call for databases such as SQL, ORACLE, and so on. This driver communicates directly with the database server.

It requires some native code to connect to the database. Type 2 drivers are usually faster than Type 1 drivers.

Like Type 1 drivers, Type 2 drivers require native database client libraries to be installed and configured on the client machine.



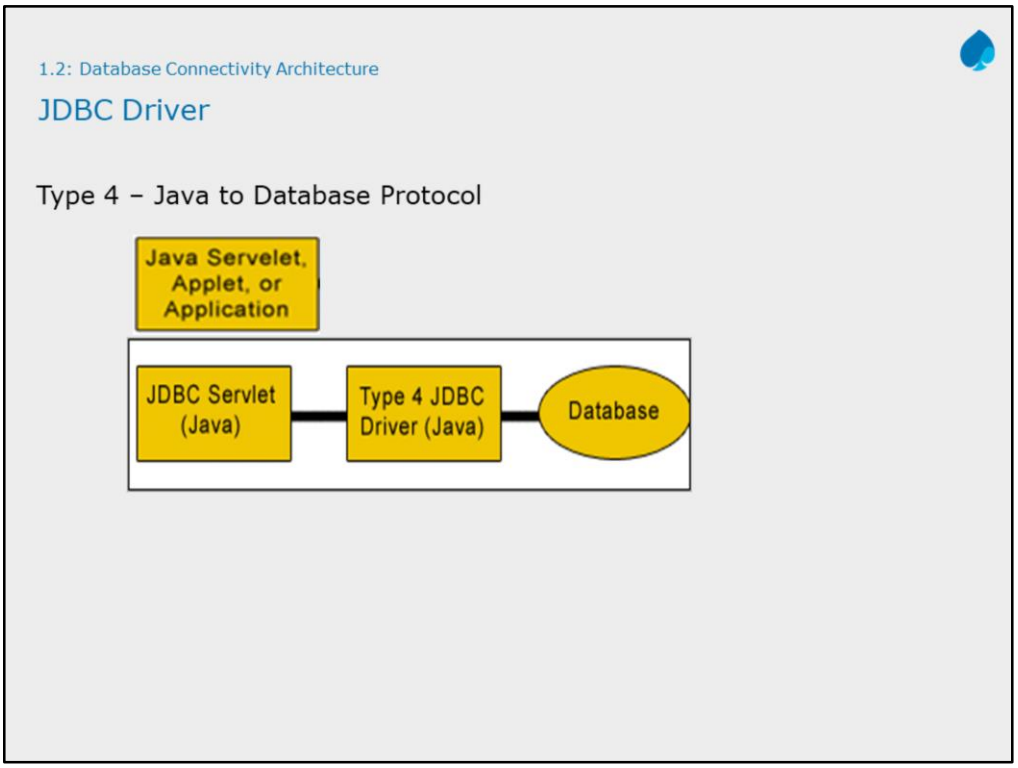
Type 3 Java to Network Protocol Or All- Java Driver:

Type 3 drivers are pure Java drivers that use a proprietary network protocol to communicate with JDBC middleware on the server.

The middleware then translates the network protocol to database-specific function calls.

Type 3 drivers are the most flexible JDBC solution because they do not require native database libraries on the client and can connect to many different databases on the back end.

Type 3 drivers can be deployed over the Internet without client installation.



Type 4 Java to Database Protocol:

Type 4 drivers are pure Java drivers that implement a proprietary database protocol to communicate directly with the database.

Like Type 3 drivers, they do not require native database libraries and can be deployed over the Internet without client installation.

One drawback to Type 4 drivers is that they are database specific. Unlike Type 3 drivers, if your back-end database changes, you may save to purchase and deploy a new Type 4 driver (some Type 4 drivers are available free of charge from the database manufacturer).

However, since Type 4 drivers communicate directly with the database engine rather than through middleware or a native library, they are usually the fastest JDBC drivers available.

This driver directly converts the Java statements to SQL statements.

JDBC APIs:

The JDBC API provides universal data access from the Java programming language.

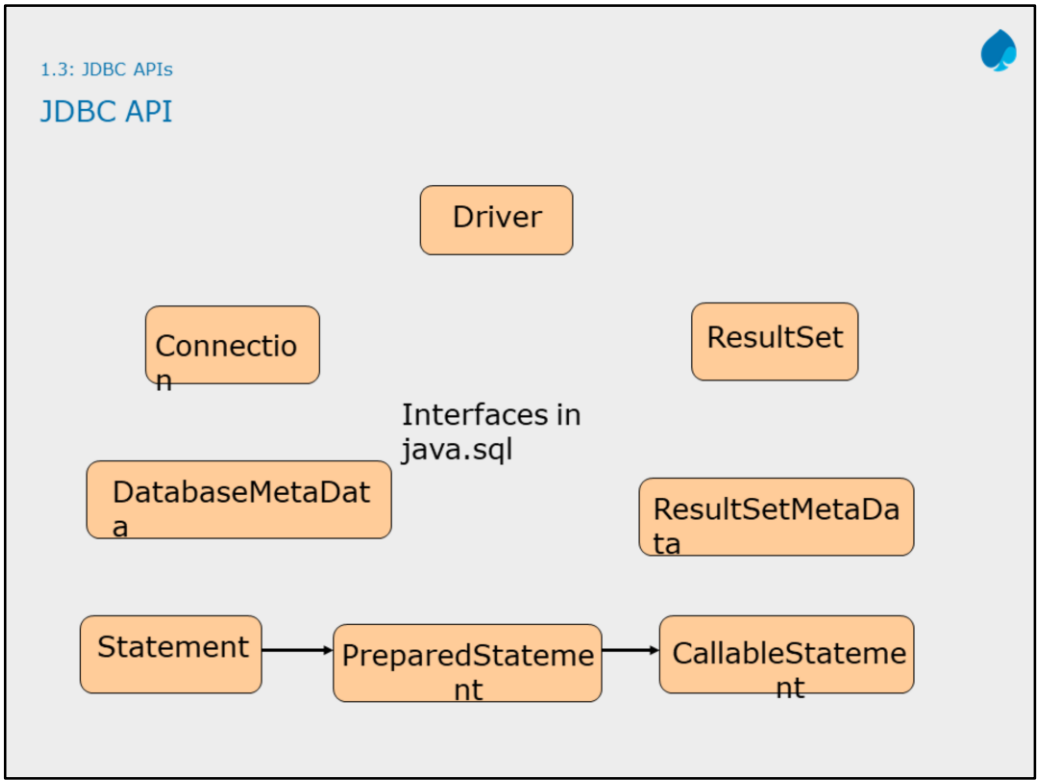
Using the JDBC 3.0 API, you can access virtually any data source, from relational databases to spreadsheets and flat files.

JDBC technology also provides a common base on which tools and alternate interfaces can be built.

The JDBC 3.0 API comprises of two packages:

1. java.sql package
2. javax.sql package

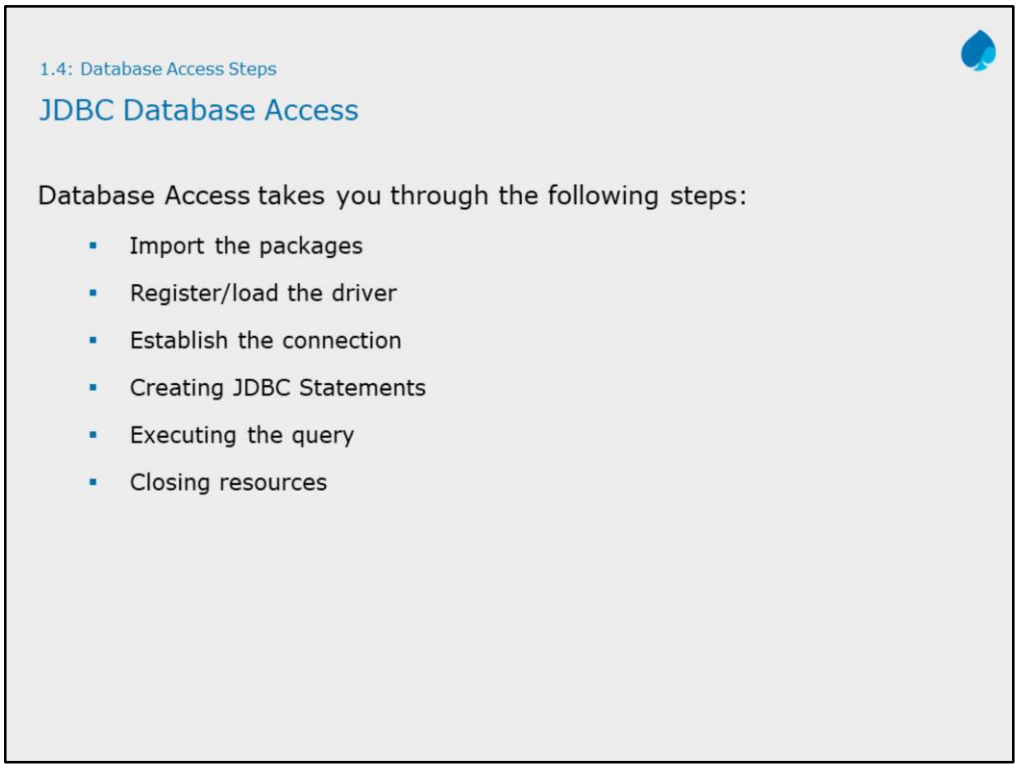
You automatically get both packages when you download the Java 2 Platform Standard Edition 5.0.



Java.sql package:

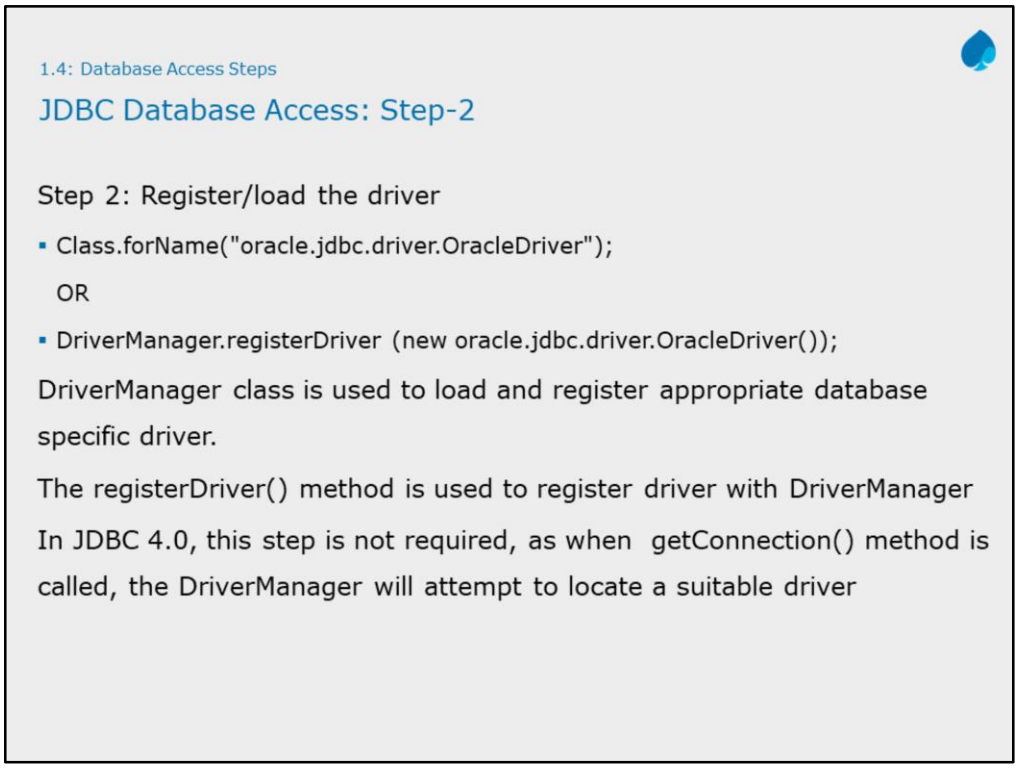
The **java.sql package** provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java programming language.
This API includes a framework whereby different drivers can be installed dynamically to access different data sources.
Although the JDBC API is mainly geared to passing SQL statements to a database, it provides for reading and writing data from any data source with a tabular format.
The **reader/writer** facility, available through the **javax.sql.RowSet** group of interfaces, can be customized to use and update data from a spread sheet, flat file, or any other tabular data source.

Please note: Here we are not discussing about javax.sql package.



13.4: Database Access Steps
- 13.4.1: Import the packages
- 13.4.2: Register the driver
- 13.4.3: Establishing the connection
- 13.4.4: Creating JDBC Statements
  - 13.4.4.1: Simple Statement
  - 13.4.4.2: Prepared Statement
  - 13.4.4.3: Callable Statement
- 13.4.5: Getting Data from a Table
- 13.4.6: Insert data into Table
- 13.4.7: Update table data

Import Packages:
The first step in accessing the data from database using JDBC APIs is importing the packages java.sql and javax.sql.
These packages provides the set of APIs which are used in accessing the database like Connection, Statement, and so on.



Register Driver:

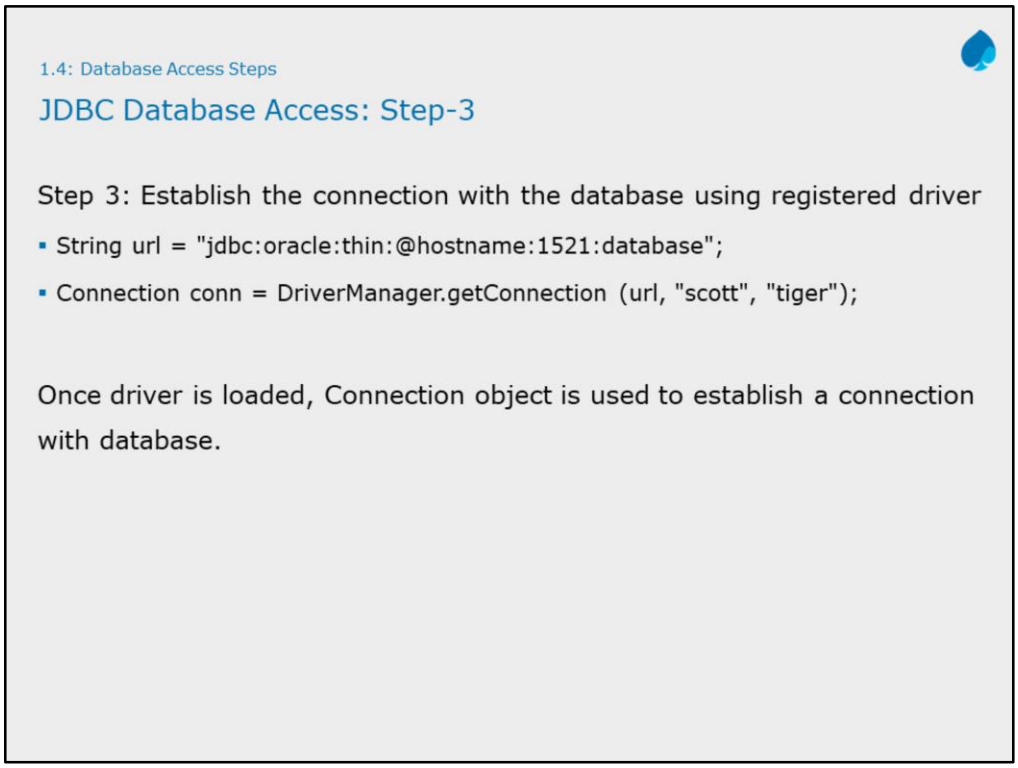
The second step is to register/load the driver for the respective database. There are two ways of loading the driver:

By using Class.forName() method

By calling DriverManager's registerDriver() method

Using Class.forName() you can load any class while DriverManger.registerDriver() is specific to JDBC driver class.

Note: In JDBC 4.0, we don't need the Class.forName() line. We can simply call getConnection() to get the database connection.

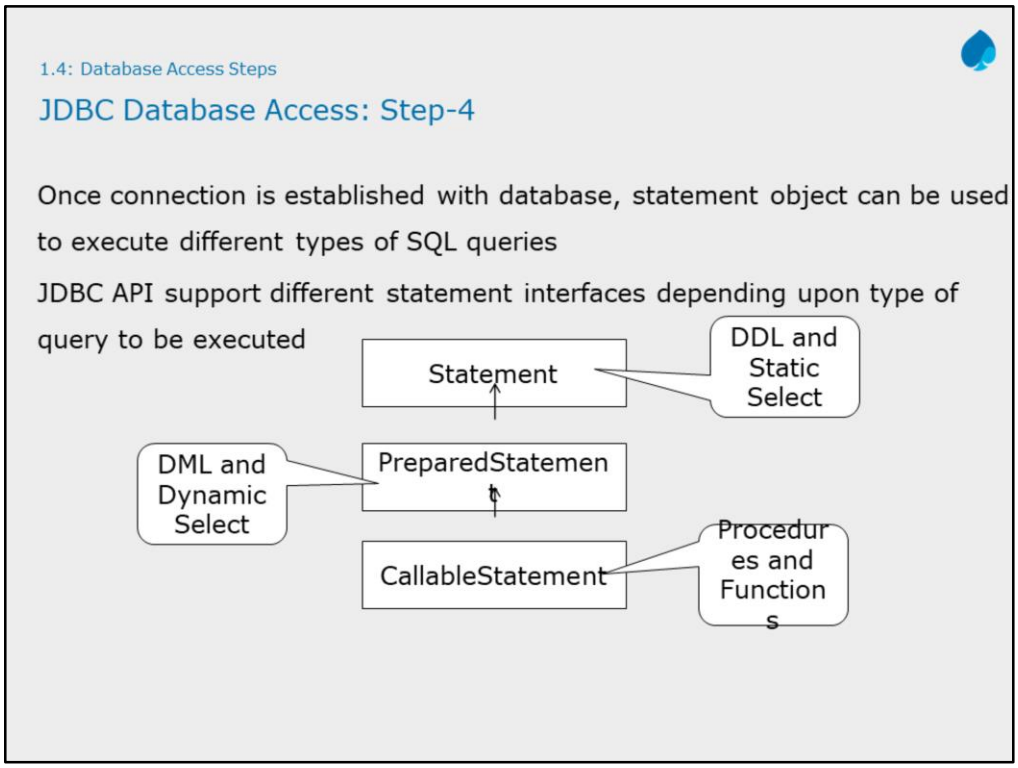


Establish the Connection:

The third step is to establish a connection with the database. There is a method DriverManager.getConnection() which returns Connection object.

This method take three arguments,

URL: This contains the information about host on which database server, database name, and the port used for communication.
Username: database userid
Password: database password

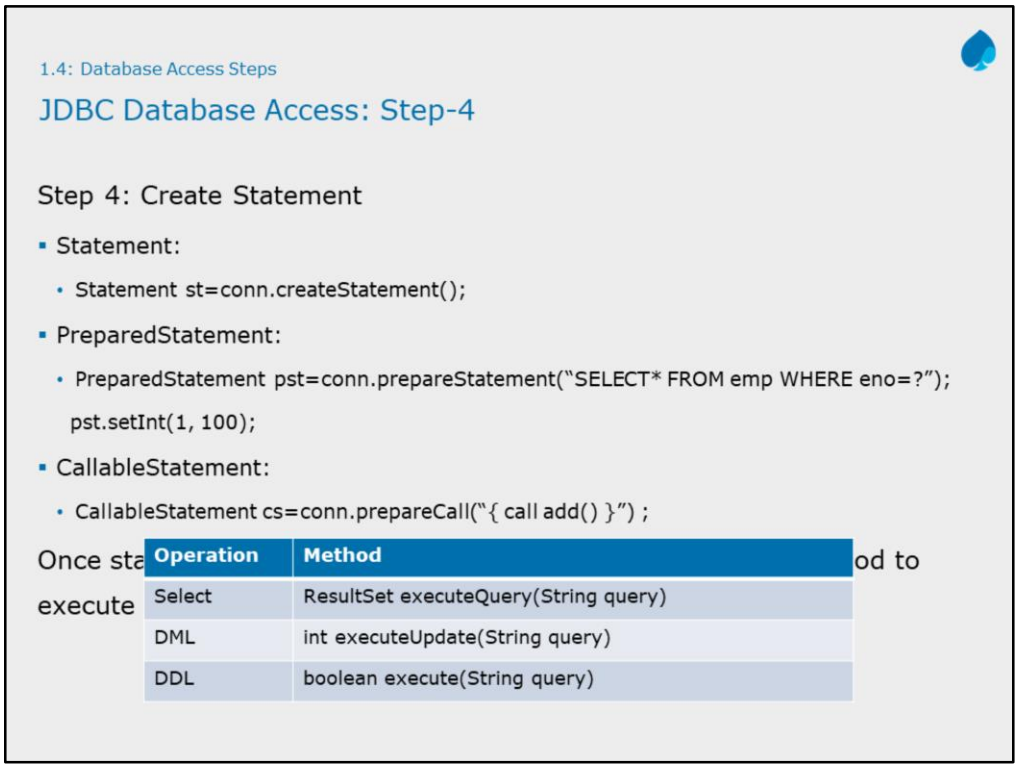


Using Statements:

java.sql.Statement interface in JDBC enables to send the SQL queries to database and retrieve data. Statement interface is suitable for executing DDL queries and Select queries which has no input.

Statement interface is further extended as PreparedStatement, which is recommended for DML and select queries that involves input parameters.

PreparedStatement is in turn extended as CallableStatement, which is suitable for calling database stored procedures and functions.



Create Statement:

The fourth step is to create a statement which indicates the details that you want to access from database.

There are three types of statements that you can create:

Statement:

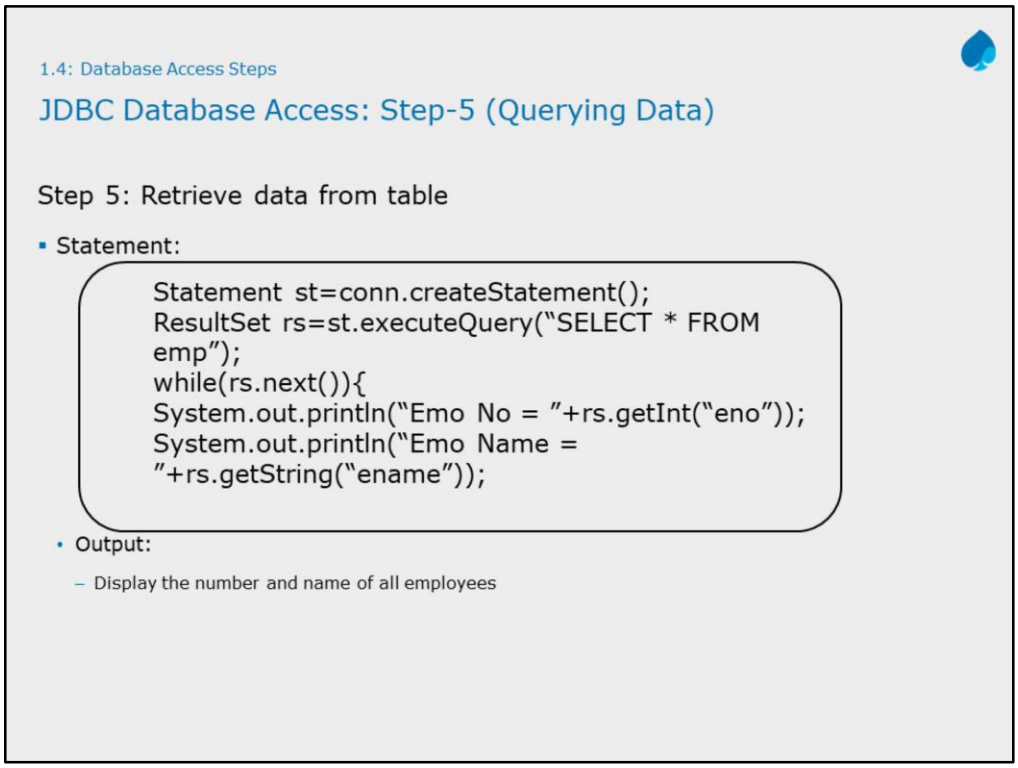
Statement object is created by calling createStatement() method on Connection object. The query is not associated with Statement object at the time of statement creation, rather it needs to be specified at the time of execution of the statement.

2. PreparedStatement:

PreparedStatement is used when you want to create parameterized query. Each parameter is represented by symbol "?" and set by setXXX() methods of PreparedStatement. The above slide shows a simple example to get more clear idea.

3. CallableStatement:

When you want to call any stored procedure from database from Java application, the CallableStatement will help you in doing that. The above example shows, how you can call add() procedure using CallableStatement.



Retrieve data from table:

Once your statement object is ready, you need to execute that statement to get the records from the database.

The executeQuery() method of the Statement returns the ResultSet object which represent the front end table of database records.

Further, you can traverse/iterate through ResultSet to retrieve the records one by one.

Use getXXX() methods to retrieve each data field value, for example rs.getInt("eno") will retrieve the employee number of the current record from rs.

Types of ResultSet:

Resultset contains results of the SQL query. There are three basic types of resultset.

Forward-only

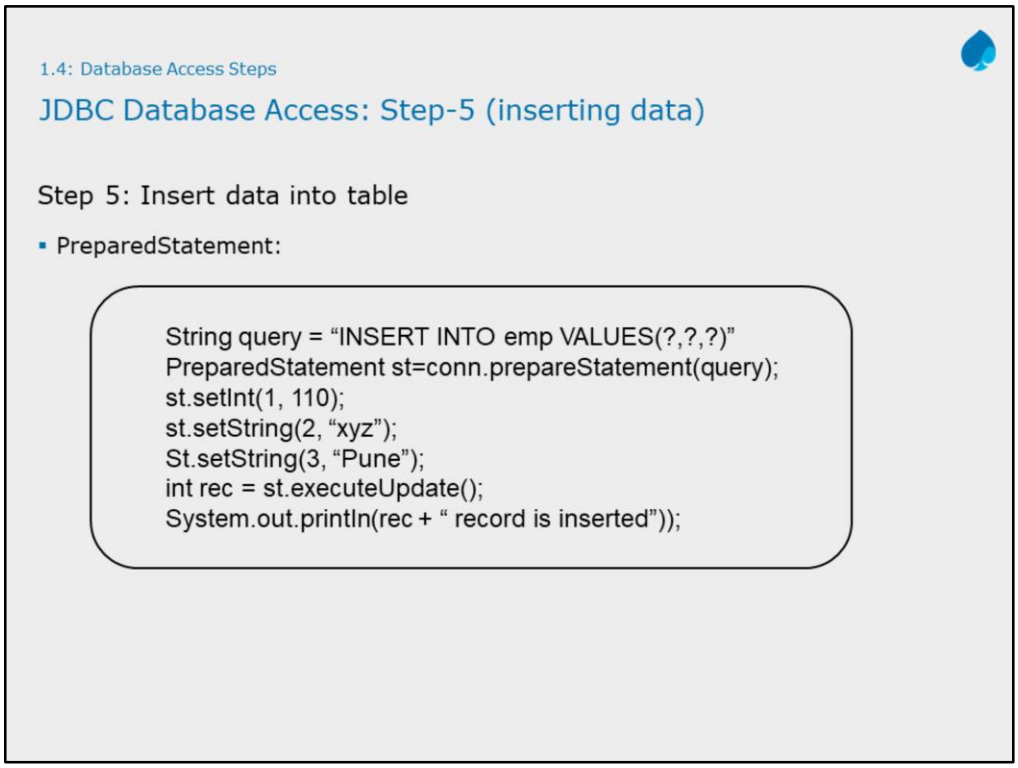
As the name suggests, this type can only move forward and are non-scrollable.

Scroll-insensitive

This type is scrollable which means the cursor can move in any direction. It is insensitive which means any change to the database will not show change in the resultset while it opens.

Scroll-sensitive

This type allows cursor to move in any direction and also propagates the changes done to the database.



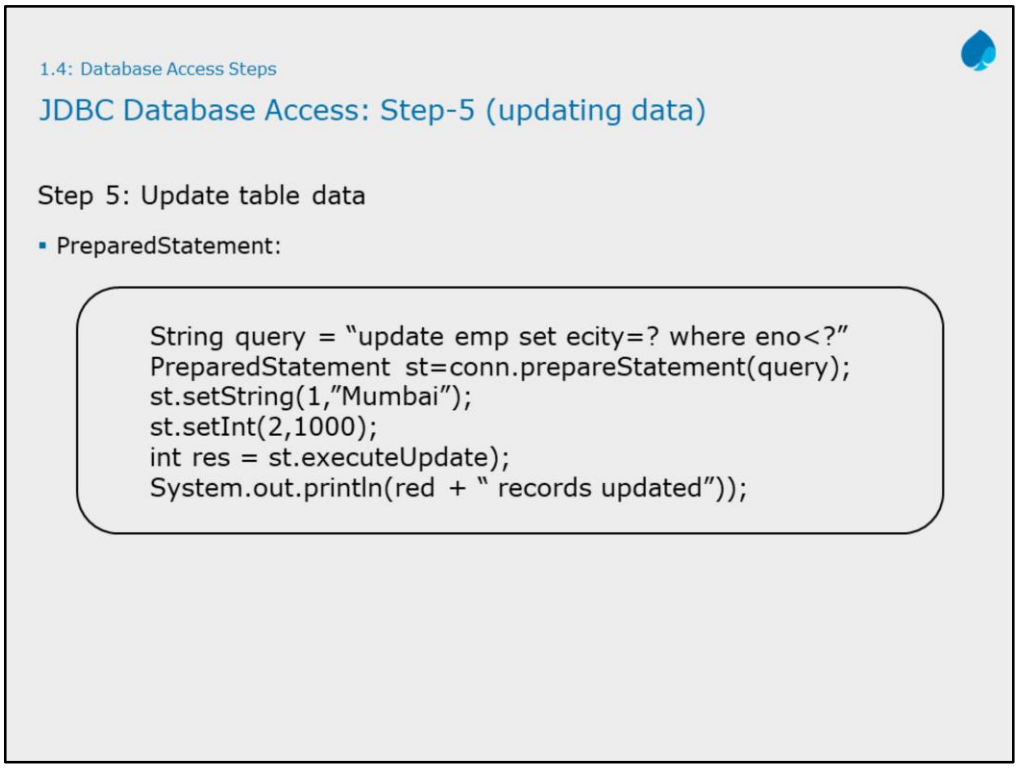
Insert data into table:

PreparedStatement is used to execute dynamic SQL queries with values being changed during runtime. These statements are pre-compiled and hence are faster as compared to Statement where every call need to be parsed and compiled before it is executed.

The "?" in query string are called as input parameters. These parameters indicates the value is not specified at compile time. A value for every "?" should be set before executing query. To set a value of parameter, setXXX() methods are used which accepts position of input parameter and value to replace before query execution.

The executeUpdate() method is used to insert and update records of the database.

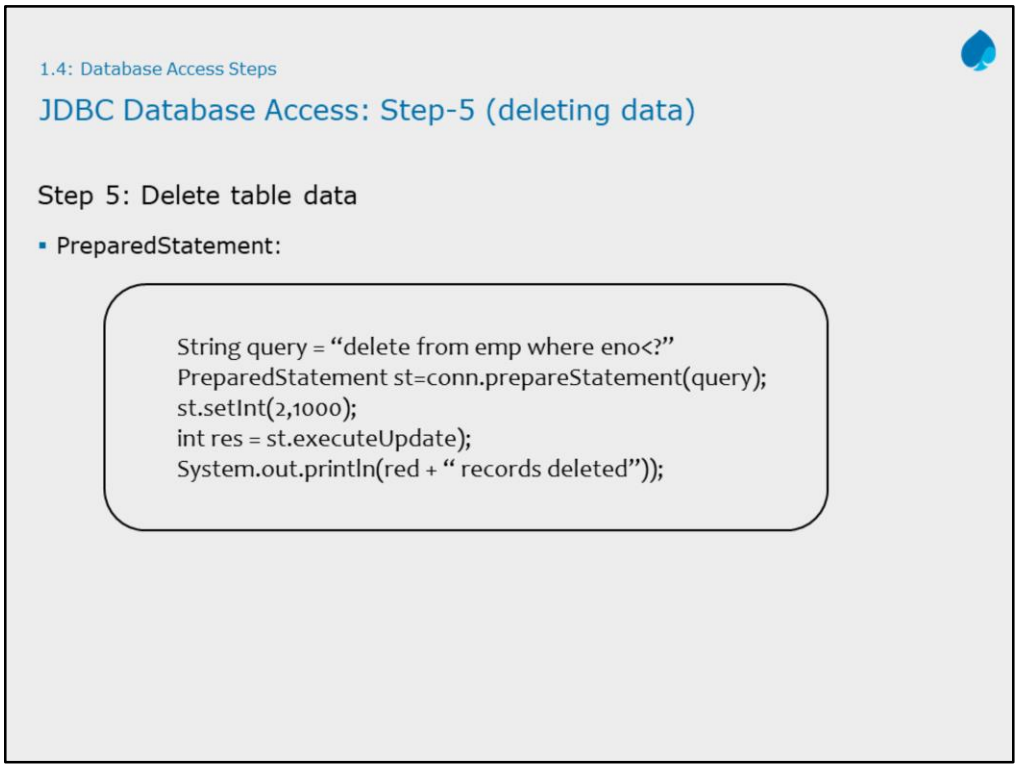
This method returns int type value indicating the number of records affected in the database.



Update Table Data:

The executeUpdate() method is used to update records from the database table.

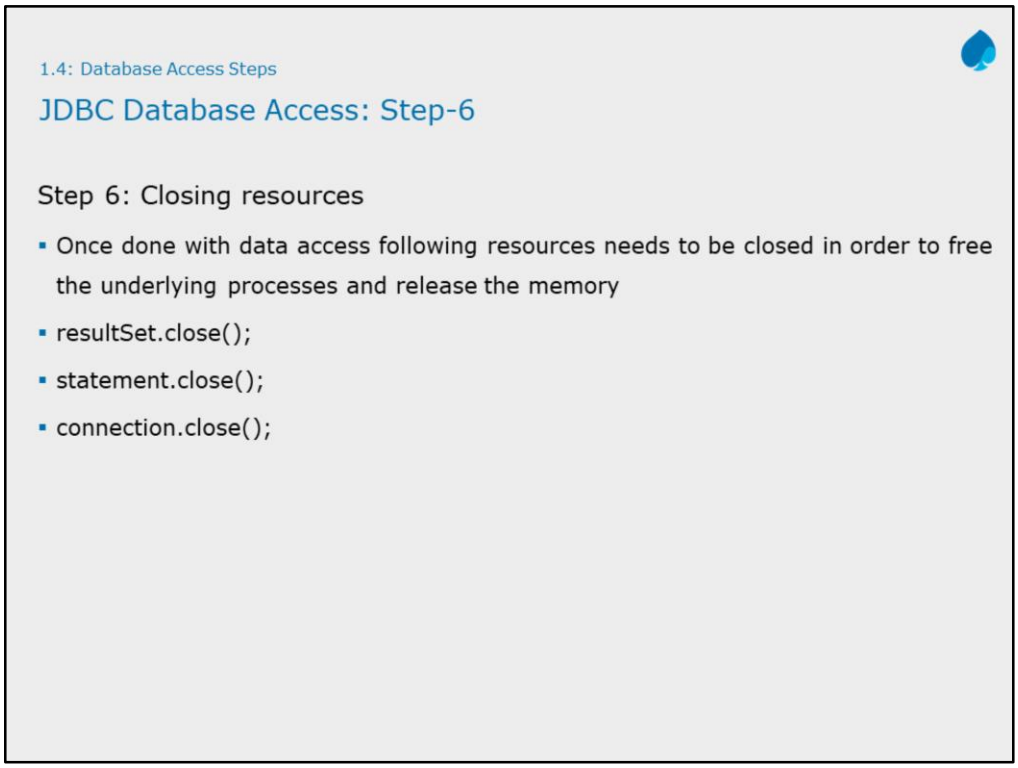
This method returns integer type value indicating the number of records affected in the database table.



Update Table Data:

The executeUpdate() method is also used to delete records from the database table.

This method returns integer type value indicating the number of records affected in the database table.



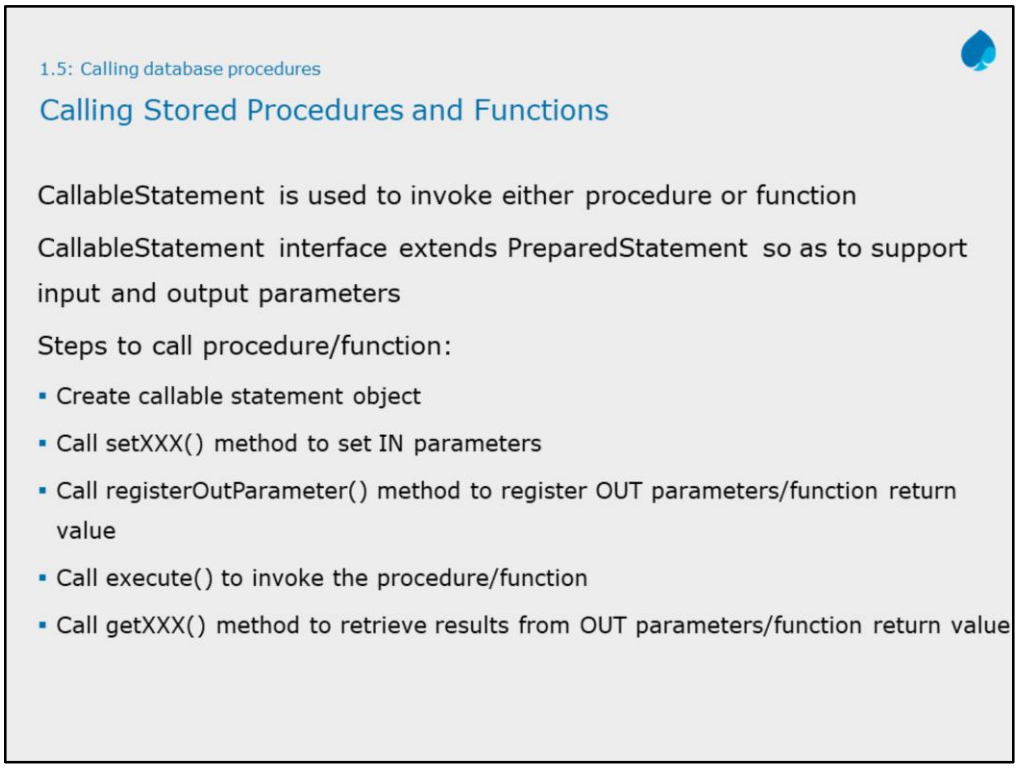
Closing resources:
The last step in database access is closing used resources in program. Closing statements and result sets is required in order to free the underlying processes and memory.
Connection object is also required to be closed. Failure in closing connection object results in database server running out of connections.
All of these interfaces provides close() method, which is used to close the respective resource.

1.4: Database Access Steps

Demo

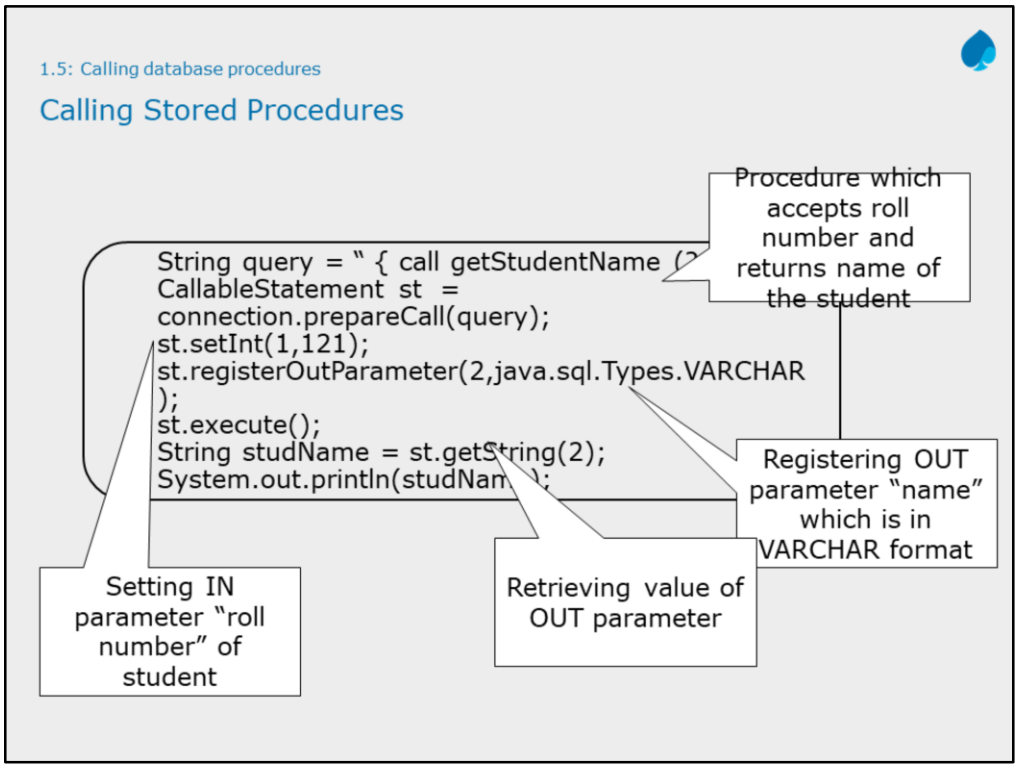
Execute the :

- Select.java
- Insert.java
- Delete.java
- PreparedStat.java programs



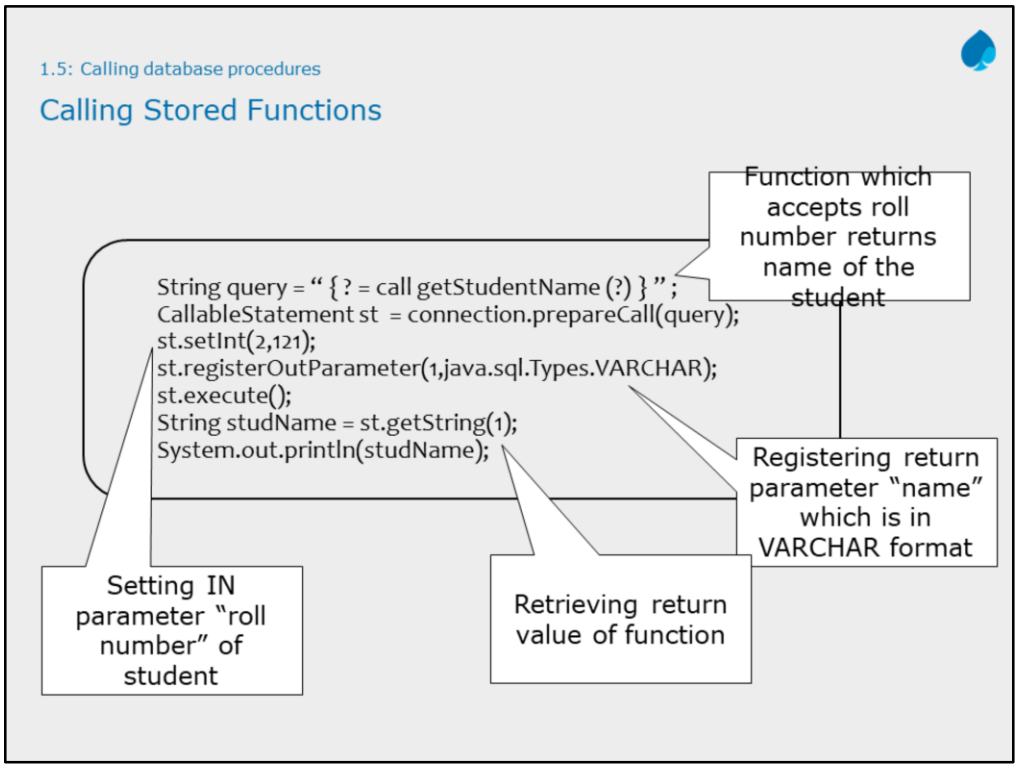
Callable Statements:

PreparedStatement is in turn extended as CallableStatement, which is suitable for calling database stored procedures and functions.



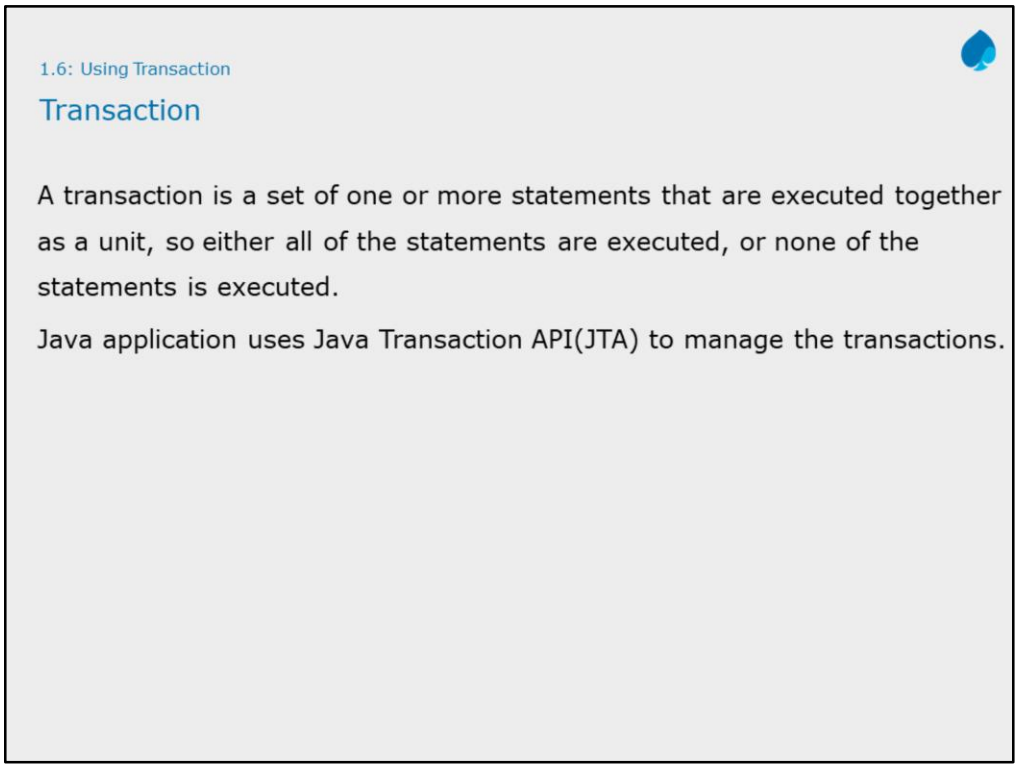
Callable Statements:

The above example shows how to use callable statement to invoke stored procedure.



Callable Statements:

The above example shows how to use callable statement to invoke stored function.

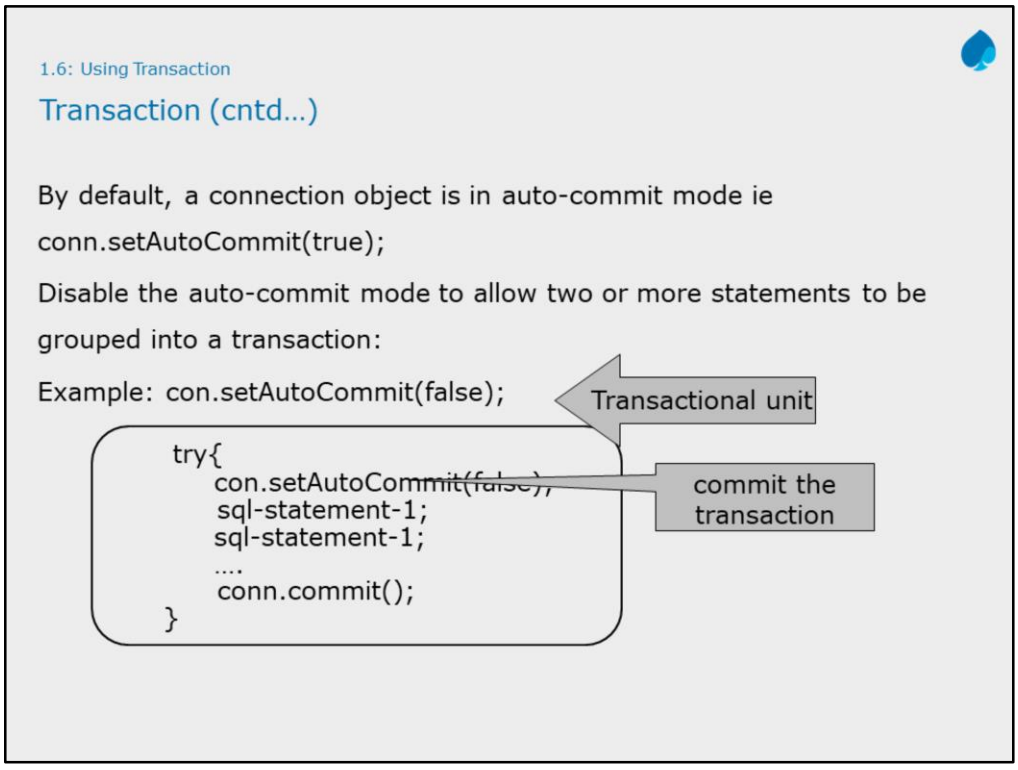


Using Transaction:

Transaction management in Java application is normally done using methods of the Connection interface.

Listed below are the Connection methods for transaction management:

- setAutoCommit()
- commit()
- rollback()
- setSavePoint()
- releaseSavePoint()



Disabling Auto-commit Mode:

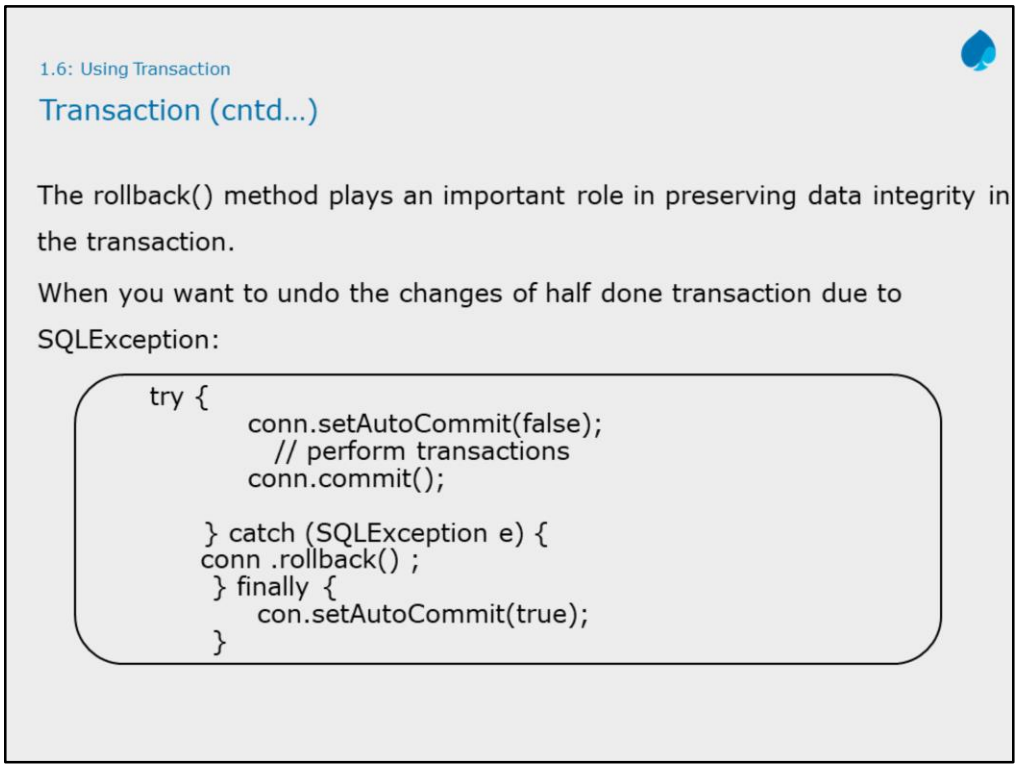
When Connection is created, by default, transaction mode in Java application is true.

It means that as soon as the statement gets executed, it reflects the changes permanently in the database. This might violate the atomicity property of the transaction. Due to this reason before transaction start, you must set the auto-commit to false.

Once you disable auto-commit, it will not reflect the effect of queries permanently in the database until you call conn.commit() method explicitly.

In the code snippet above, auto-commit mode is disabled for the connection conn. This means that the enclosed sql statements are committed together when the commit method is called.

Whenever the commit method is called (either automatically when auto-commit mode is enabled or explicitly when it is disabled), all changes resulting from statements in the transaction are made permanent.



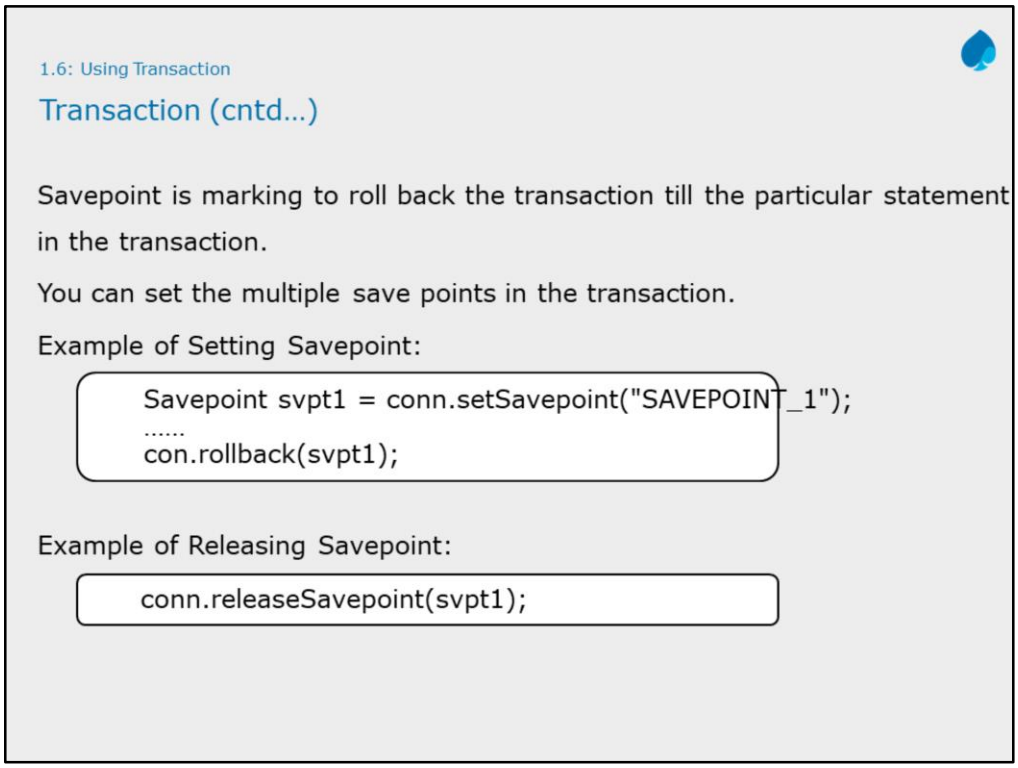
When do you call the Rollback Method:

Calling the rollback method aborts a transaction and returns any values that were modified to their previous values. If you are trying to execute one or more statements in a transaction and get an SQLException, you should call the rollback method to abort the transaction and start the transaction all over again. That is the only way to be sure of what has been committed and what has not been committed.

Catching an SQLException tells you that something is wrong. However, it does not tell you what was or was not committed. Since you cannot count on the fact that nothing was committed, calling the method rollback is the only way to be sure.

The example in the above slide, demonstrates a transaction and includes a catch block that invokes the rollback method. In this particular situation, it is not really necessary to call rollback and we do it mainly to illustrate how it is done.

However, if the application continued and used the results of the transaction, it would be necessary to include a call to rollback in the catch block in order to protect against using possibly incorrect data.



Setting and Releasing the Savepoint:

The JDBC 3.0 API adds the method Connection.setSavepoint, which sets a savepoint within the current transaction.

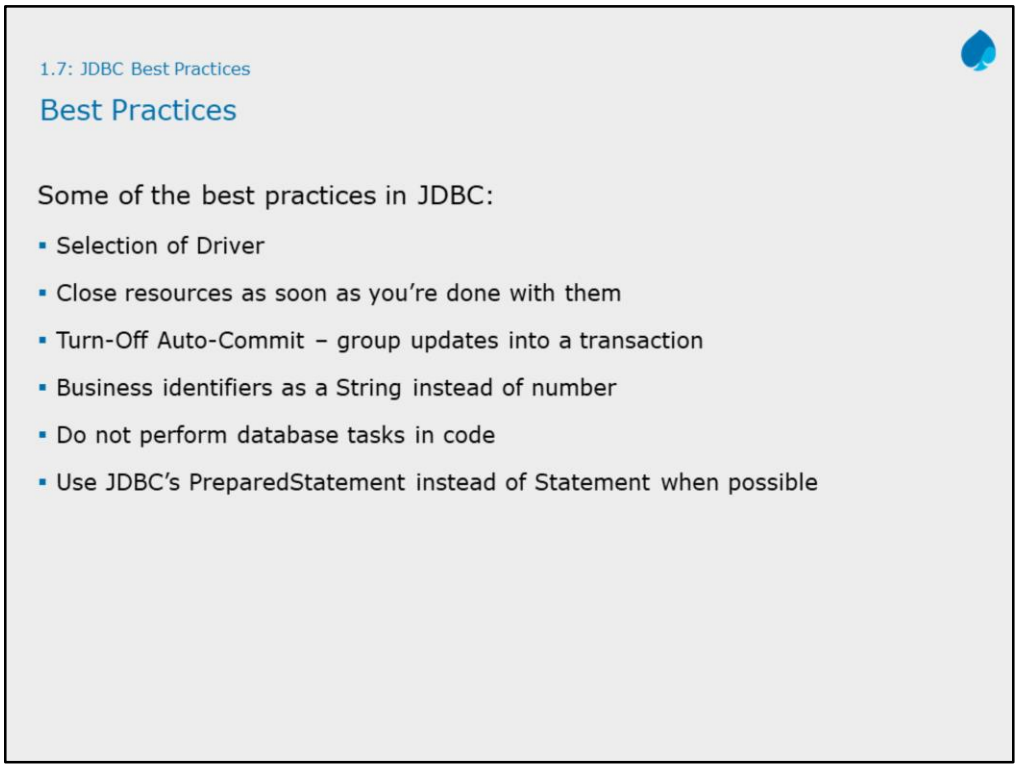
The Connection.rollback() method has been overloaded to take a savepoint argument.

The method Connection.releaseSavepoint() takes a Savepoint object as a parameter and removes it from the current transaction. Once a savepoint has been released, attempting to reference it in a rollback operation causes an SQLException to be thrown.

Any savepoints that have been created in a transaction are automatically released and become invalid when the transaction is committed, or when the entire transaction is rolled back. Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints that were created after the savepoint.

Demo : Transaction

Execute the Transaction.java program



JDBC Best Practices:

Following are some of the best practices used in JDBC:

Selection of Driver

Select a certified, high performance type 2 (Thin) JDBC driver and use the latest drivers release.

Close resources as soon as you are done with them (in finally)

For example: Statements, Connections, Resultsets, and so on.

Failure to do so will eventually cause the application to "hang", and fail to respond to user actions.

Turn-Off Auto-Commit

It is best practice to execute the group of the statement together which are the part of the same transaction. It helps to avoid the overhead of data inconsistency.

JDBC Best Practices:

4. Business identifiers as a String instead of number

Many problem domains use numbers as business identifiers. Credit card numbers, bank account numbers, and the like, are often simply that - numbers. *It should always be kept in mind, however, that such items are primarily identifiers, and their numeric character is usually completely secondary*. Their primary function is to identify items in the problem domain, not to represent quantities of any sort.

For example: It almost never makes sense to operate on numeric business identifiers as true numbers - adding two account numbers, or multiplying an account number by -1, are meaningless operations. That is, one can strongly argue that modeling an account number as a Integer is inappropriate, simply because it does not behave as an Integer.

Furthermore, new business rules can occasionally force numeric business identifiers to be abandoned in favor of alphanumeric ones. It seems prudent to treat the content of such a business identifier as a business rule, subject to change. As usual, a program should minimize the ripple effects of such changes.

In addition, Strings can contain leading zeros, while numeric fields will remove them.

5. Do not perform database tasks in code

Databases are a mature technology, and they should be used to do as much work as possible. Do not do the following in code, if it can be done in SQL instead:

ordering (ORDER BY)
filtering based on criteria (WHERE)
joining tables (WHERE, JOIN)
summarizing (GROUP BY, COUNT, AVG, STDDEV)

Any corresponding task implemented entirely in code would very likely:

be *much* less robust and efficient
take longer to implement
require more maintenance effort

6. Use JDBC's PreparedStatement instead of Statement when possible for the following reasons:

It is in general more secure. When a Statement is constructed dynamically from user input, it is vulnerable to SQL injection attacks. PreparedStatement is not vulnerable in this way.

There is usually no need to worry about escaping special characters if repeated compilation is avoided, its performance is usually better.

In general, it seems safest to use a Statement only when the SQL is of fixed, known form, with no parameters

1.8: Java Database Connectivity

Lab : JDBC

Lab 1: JDBC 4.0

Summary

In this lesson, you have learnt:

- Establishing the connection with database to perform database operations
- Different types of statement creation
- Different ways of executing statements
- Transaction management using JDBC APIs
- Use of connection pooling to increase the performance of application
- And best practices for JDBC applications

Review Question

Question 1 : Which Statement is used when you want to pass parameter to the query?

- **Option 1 :** Statement
- **Option 2 :** PreparedStatement
- **Option 3 :** CallableStatement

Question 2 : _____ method is best suited to execute DDL Queries.

Question 3 : By default, a connection object is in auto-commit mode

- True/False