

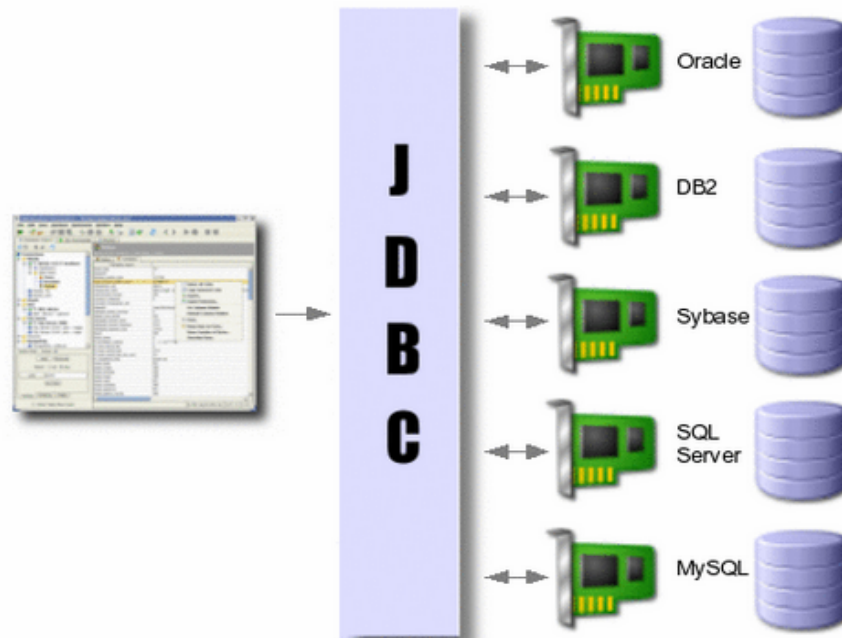
DATABASE PROGRAMMING WITH JDBC

- **Java JDBC Tutorial**
- **Working steps**
- **DriverManager class**
- **JDBC Statement**
- **JDBC ResultSet**
- **JDBC PreparedStatement (with Parameter)**
- **JDBC Callablestatement**
- **Transaction Management in JDBC**
- **Batch Processing in JDBC**

Section 1

JAVA JDBC TUTORIAL

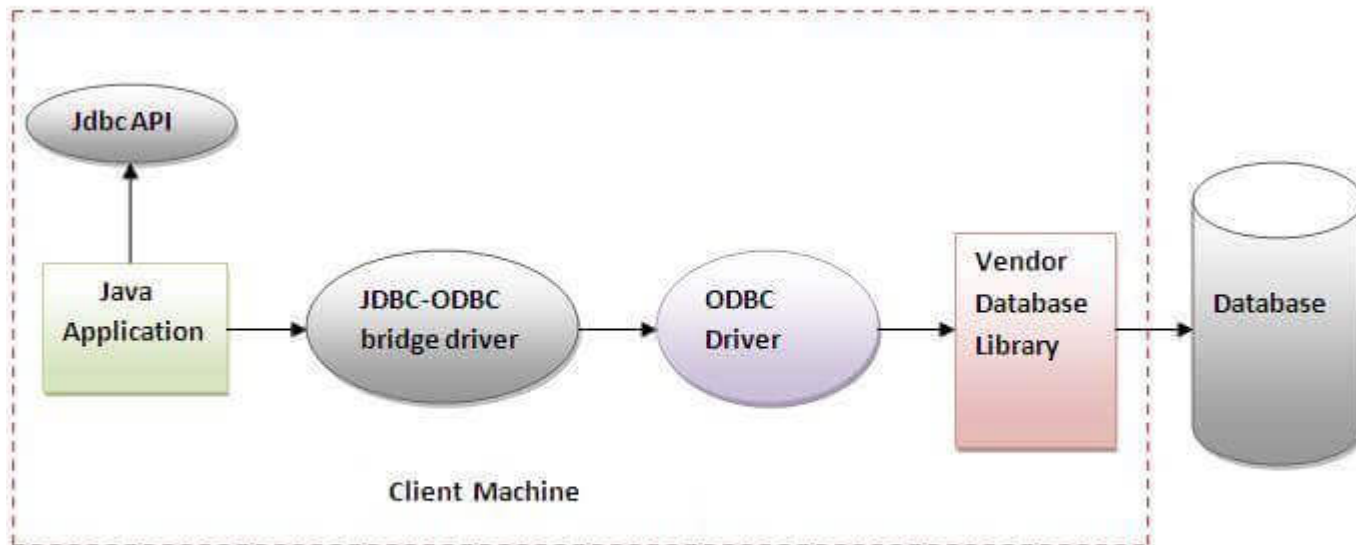
- ❖ JDBC (Java Database Connectivity) API allows Java programs to connect to databases
- ❖ Database access is the **same for all database vendors**
- ❖ The JVM uses a **JDBC driver** to translate generalized JDBC calls into vendor specific database calls.



- ❖ JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition).
- ❖ JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:
 - ✓ JDBC-ODBC Bridge Driver,
 - ✓ Native Driver,
 - ✓ Network Protocol Driver, and
 - ✓ Thin Driver
- ❖ A **JDBC driver** is a **set of Java classes** that implement the JDBC interfaces,
 - ✓ targeting a **specific database**.
- ❖ The JDBC interfaces comes with **standard Java**,
 - ✓ but the implementation of these interfaces is specific **to the database** you need to connect to. Such an implementation is called a JDBC driver.
- ❖ We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database.

JDBC-ODBC bridge driver

- ❖ The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

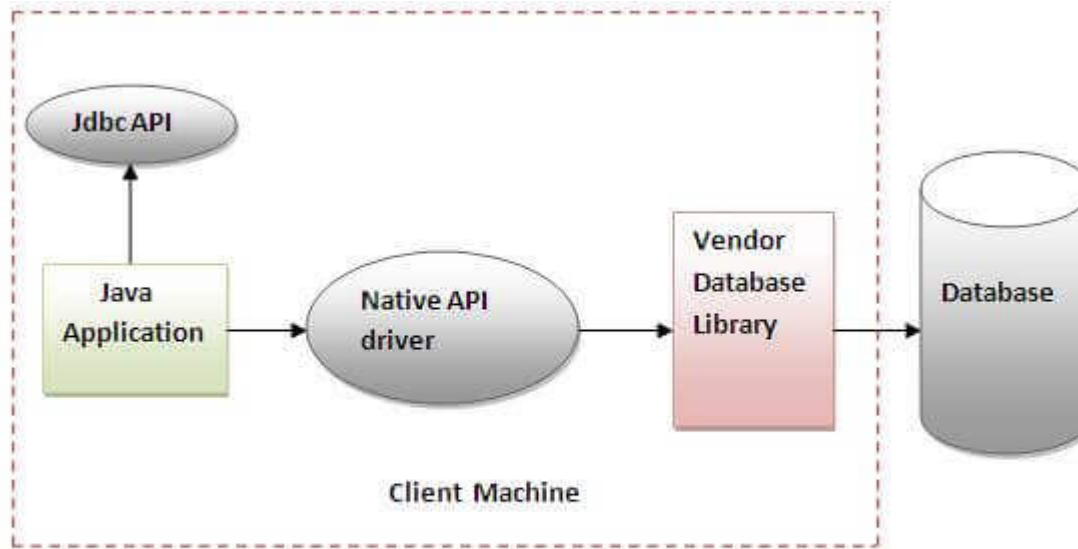


In Java 8, the JDBC-ODBC Bridge has been removed.

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

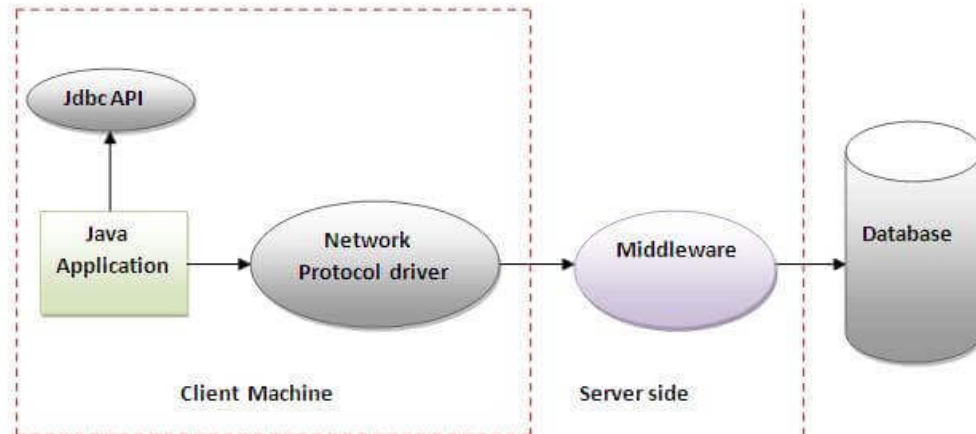
Native-API driver

- ❖ The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.



- ❖ **Advantage:**
 - ✓ performance upgraded than JDBC-ODBC bridge driver.
- ❖ **Disadvantage:**
 - ✓ The Native driver needs to be installed on the each client machine.
 - ✓ The Vendor client library needs to be installed on client machine.

- ❖ The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.



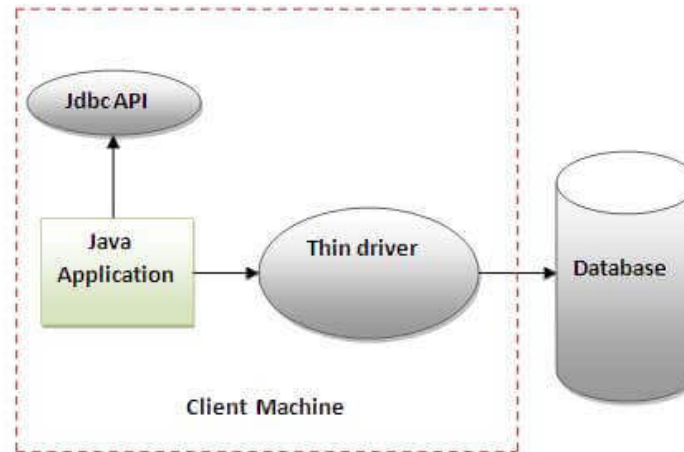
- ❖ **Advantage:**

- ✓ No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

- ❖ **Disadvantages:**

- ✓ Network support is required on client machine.
- ✓ Requires database-specific coding to be done in the middle tier.
- ✓ Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

- ❖ The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.



- ❖ **Advantage:**
 - ✓ Better performance than all other drivers.
 - ✓ No software is required at client side or server side.
- ❖ **Disadvantage:**
 - ✓ Drivers depend on the Database.

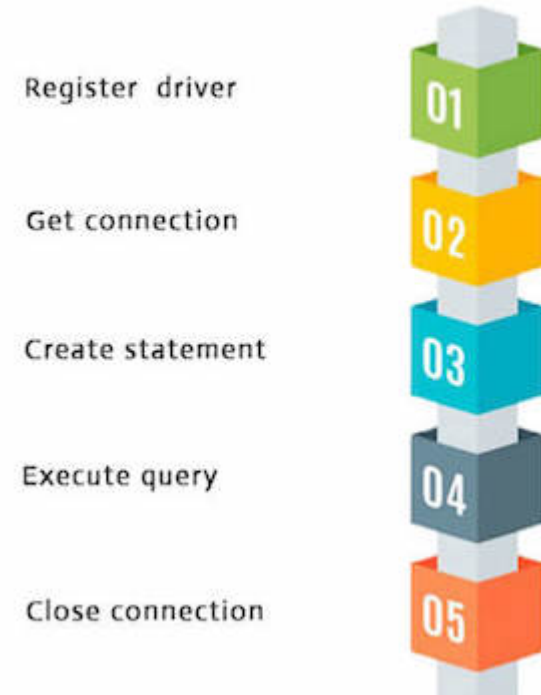
Section 2

WORKING STEPS

Working steps

1. Register the Driver class
2. Create connection
3. Create statement
4. Execute queries
5. Close connection

Java Database Connectivity



Register the driver class

- ❖ The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.
- ❖ Example to register the **OracleDriver** class:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- ❖ Example to register the **SQLServerDriver** class:

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

- ❖ Example to register the **MySqlServerDriver** class:

```
Class.forName("com.mysql.jdbc.Driver");
```

- ❖ **Note:** Since JDBC 4.0, explicitly registering the driver is optional. We just need to put vender's Jar in the classpath, and then JDBC driver manager can detect and load the driver automatically.

Create Connection

- ❖ The **getConnection()** method of DriverManager class is used to establish connection with the database.

- ❖ Syntax of getConnection() method:

- ✓ **public static** Connection getConnection(String url)**throws** SQLException
- ✓ **public static** Connection getConnection(String url,String name,String password)**throws** SQLException

- ❖ Example to establish connection with the **Oracle** database

```
Connection con=DriverManager.getConnection(
    "jdbc:oracle:thin:@localhost:1521:xe","system","password");
```

- ❖ Example to establish connection with the **My SQL Server** database

```
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/
ebookshop?
allowPublicKeyRetrieval=true&useSSL=false&serverTimezone=UTC",
    "myuser", "xxxx");
```

- ❖ Example to establish connection with the **MS SQL Server** database

```
String connectionUrl = "jdbc:sqlserver://localhost:1433;databaseName=Fsoft_Training";
Connection conn = DriverManager.getConnection(connectionUrl, "system","password");
```

Create Access Statement

- ❖ The **createStatement()** method of **Connection** interface is used to create statement. The object of statement is responsible to execute queries with the database.
 - ✓ Use for general-purpose **access to your database**.
 - ✓ Useful when you are using static **SQL statements** at runtime.
 - ✓ The **Statement** interface cannot accept parameters.

❖ Syntax:

```
Statement stmt = null;
try {
    stmt = conn.createStatement(); // or
    stmt = con.createStatement(ResultSetType,
                               ConcurencyType);
} catch (SQLException e) {
}
finally { stmt.close(); }
```

Execute the query

- ❖ The **executeQuery()** method of **Statement** interface is used to execute queries to the database. This method returns the object of **ResultSet** that can be used to get all the records of a table.
- ❖ Syntax of **executeQuery()** method:

✓ **public** **ResultSet** executeQuery(**String** sql)**throws** **SQLException**

- ❖ **Example:**

```
ResultSet rs = stmt.executeQuery("SELECT * FROM EMP");
```

```
while(rs.next()){  
    System.out.println(rs.getInt(1) + " " +  
rs.getString(2));  
}
```

Close the connection object

- ❖ By closing connection object statement and **ResultSet** will be closed automatically.
- ❖ The close() method of **Connection** interface is used to close the connection.
- ❖ Syntax of **close()** method:

```
public void close() throws SQLException
```

- ❖ **Example:**

```
con.close();
```

- ❖ **Note:** Since Java 7, JDBC has ability to use try-with-resources statement to automatically close resources of type Connection, ResultSet, and Statement.

Section 3

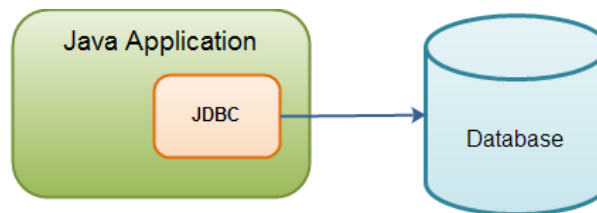
DRIVERMANAGER CLASS

DriverManager class

- ❖ The **DriverManager** class acts as an interface between **user** and **drivers**. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver.
- ❖ The **DriverManager** class maintains a list of Driver classes that have registered themselves by calling the method `DriverManager.registerDriver()`.
- ❖ **Methods:**

Method	Description
1) public static void registerDriver (Driver driver)	is used to register the given driver with DriverManager.
2) public static void deregisterDriver (Driver driver)	is used to deregister the given driver (drop the driver from the list) with DriverManager.
3) public static Connection getConnection (String url)	is used to establish the connection with the specified url.
4) public static Connection getConnection (String url, String userName, String password)	is used to establish the connection with the specified url, username and password.

- ❖ A **Connection** is the session between **Java application** and **database**.
- ❖ The Connection interface is a factory of **Statement**, **PreparedStatement** and **DatabaseMetaData**.
- ❖ *By default, connection commits the changes after executing queries.*
- ❖ **Methods:**
 - ✓ public Statement **createStatement()**: creates a statement object that can be used to execute SQL queries.
 - ✓ public Statement **createStatement**(int resultSetType, int resultSetConcurrency):
creates a Statement object that will generate ResultSet objects with the given type and concurrency.
 - ✓ public void **setAutoCommit**(boolean status): is used to set the commit status. By default it is **true**.



❖ Methods:

- ✓ public void **commit()**: saves the changes made since the previous commit/rollback permanent.
- ✓ public void **rollback()**: drops all changes made since the previous commit/rollback.
- ✓ public void **close()**: closes the connection and Releases a JDBC resources immediately
- ✓ public PreparedStatement **prepareStatement()**: creates a JDBC PreparedStatement object.
- ✓ public DatabaseMetaData **getMetaData()**: returns a JDBC DatabaseMetaData object which can be used to introspect the database the JDBC Connection is connected to

Section 4

JDBC STATEMENT

- ❖ The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

- ❖ **Create Statement:**

```
Statement statement = connection.createStatement();
```

```
Statement statement = connection.createStatement(  
    int resultSetType, int resultSetConcurrency);
```

```
Statement statement = connection.createStatement(  
    int resultSetType, int resultSetConcurrency,  
    int resultSetHoldability)
```

❖ Statement's **methods**:

- ✓ **boolean execute(String SQL)** : may be **any kind of SQL statement**. Returns a boolean value of true if a ResultSet object can be retrieved; false if the first result is an update count or there is no result.
- ✓ **int executeUpdate(String SQL)** : Returns the **numbers 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.
- ✓ **public int[] executeBatch()**: is used to execute batch of commands.

❖ Example 1: Execute a SELECT query via a Statement

```
// Create and execute an SQL statement that returns some
data.
String SQL1 = "SELECT TOP 10 * FROM Person";
Statement stmt=conn.createStatement();
//ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE
ResultSet rs = stmt.executeQuery(SQL);
```

❖ Example 2: Execute an INSERT via a Statement

```
// Create and execute an SQL statement that returns some
data.
String SQL2 = "INSERT INTO STOCK(STOCK_CODE, STOCK_NAME)
              VALUES('11', 'STOCK1')";
Statement stmt = conn.createStatement();
int no_of_row = stmt.executeUpdate(SQL);
```


❖ Retrieve data

```
// Iterate through the data in the result set and display it.
```

```
while (rs.next()) {  
    System.out.println(rs.getInt(1) + "\t" +  
  
    rs.getString(2)+"\t"+rs.getInt(3));  
}
```

❖ Close connection

```
statement.close();  
conn.close();
```

Statement Using Java Try With Resources

- ❖ In order to close a Statement correctly after use, you can open it inside a Java **Try With Resources** block.
- ❖ Here is an example of closing a Java JDBC Statement instance using the **try-with-resources** construct:

```
try (Statement statement =  
    connection.createStatement()) {  
    // use the statement in here.  
} catch (SQLException e) {  
    // TODO: handle exception  
}
```

- ❖ *Once the try block exits, the **Statement** will be closed automatically.*

Section 5

JDBC RESULTSET

- ❖ The Java JDBC **java.sql.ResultSet** interface represents the result of a database query.
- ❖ This ResultSet is then iterated to inspect the result.
- ❖ **A ResultSet Contains Records:**
 - ✓ A JDBC ResultSet contains records. Each records contains a set of columns. Each record contains the same amount of columns, although not all columns may have a value. A column can have a null value.
 - ✓ The following ResultSet has 3 different columns (Name, Age, Gender), and 3 records with different values for each column

Name	Age	Gender
John	27	Male
Jane	21	Female
Jeanie	31	Female

ResultSet example - records with columns

Creating a ResultSet

- ❖ You create a **ResultSet** by executing a **Statement** or **PreparedStatement**, like this:

```
Statement statement = connection.createStatement();
```

```
ResultSet result = statement.executeQuery("SELECT * FROM  
dbo.Course");
```

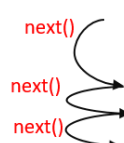
- ❖ Or like this:

```
String selectQuery = "SELECT * FROM dbo.Course";
```

```
PreparedStatement statement =  
connection.prepareStatement(selectQuery);
```

```
ResultSet result = statement.executeQuery();
```

- ❖ **ResultSet data:**



	1	2	3	4	5
	course_id	subject_id	course_code	title	number_of_credits
1	11111	CSCI	1301	Introduction to Java I	4
2	11112	CSCI	1302	Introduction to Java II	3
3	11113	CSCI	3720	Database Systems	3
4	11114	CSCI	4750	Rapid Java Application	3
5	11115	MATH	2750	Calculus I	5
6	11116	MATH	3750	Calculus II	5
7	11117	EDUC	1111	Reading	3
8	11118	ITEC	1344	Database Administration	3

```
while (result.next()) {  
    System.out.println(result.getString(1)  
        + "\t" + result.getString(2)  
        + "\t" + result.getString(3)  
        + "\t" + result.getString(4)  
        + "\t" + result.getInt(5));  
}
```

ResultSet Type, Concurrency

❖ When you create a **ResultSet** there are three attributes you can set:

✓ Type

✓ Concurrency

```
// for use with ResultSet only
// No "previous" method using, no update
Statement statement =
connection.createStatement(
    ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.CONCUR_READ_ONLY,
    ResultSet.HOLD_CURSORS_OVER_COMMIT);

// with "previous" method using, update
Statement statement =
connection.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
```

- ❖ **Type of ResultSet:** The possible Type are given below, If you do not specify any ResultSet type, you will automatically get one that is **TYPE_FORWARD_ONLY**.

Type	Description
ResultSet. TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet. TYPE_SCROLL_INSENSITIVE	The cursor can scroll forwards and backwards, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet. TYPE_SCROLL_SENSITIVE	The cursor can scroll forwards and backwards, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

- ❖ **Concurrency of ResultSet:** The possible RSConcurrency are given below, If you do not specify any Concurrency type, you will automatically get one that is **CONCUR_READ_ONLY**.

Concurrency	Description
ResultSet. CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet. CONCUR_UPDATABLE	Creates an updateable result set.

❖ ResultSet methods:

.N.	Methods & Description
1	public void beforeFirst() throws SQLException Moves the cursor to just before the first row
2	public void afterLast() throws SQLException Moves the cursor to just after the last row
3	public boolean first() throws SQLException Moves the cursor to the first row
4	public void last() throws SQLException Moves the cursor to the last row.
5	public boolean absolute(int row) throws SQLException Moves the cursor to the specified row
6	public boolean relative(int row) throws SQLException Moves the cursor the given number of rows forward or backwards from where it currently is pointing.

❖ ResultSet methods:

.N.	Methods & Description
7	public boolean previous() throws SQLException Moves the cursor to the previous row. This method returns false if the previous row is off the result set
8	public boolean next() throws SQLException Moves the cursor to the next row. This method returns false if there are no more rows in the result set
9	public int getRow() throws SQLException Returns the row number that the cursor is pointing to.
10	public void moveToInsertRow() throws SQLException Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered.
11	public void moveToCurrentRow() throws SQLException Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing

❖ Viewing a Result Set:

S. N.	Methods & Description
1	public int getInt(String columnName) throws SQLException Returns the int in the current row in the column named columnName
2	public int getInt(int columnIndex) throws SQLException Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.
3	public XXX getXXX(int columnIndex) throws SQLException

- ❖ The ResultSet interface contains a collection of **update methods** for **updating the data of a result set**.
- ❖ As with the get methods, there are two update methods for each data type:
 - ✓ One that takes in a column name.
 - ✓ One that takes in a column index.
- ❖ **For example:**

S.N	Methods & Description
1	public void updateString(int columnIndex, String s) throws SQLException Changes the String in the specified column to the value of s.
2	public void updateString(String columnName, String s) throws SQLException Similar to the previous method, except that the column is specified by its name instead of its index.

ResultSet Example

```
public List<Course> findCourseByName(String name) throws SQLException
{

    Connection connection = null;
    Statement statement = null;
    ResultSet result = null;

    List<Course> courses = new ArrayList<Course>();

    try {

        connection = DBUtils.getConnection();

        statement =
connection.createStatement(ResultSet.TYPE_FORWARD_ONLY,
        ResultSet.CONCUR_READ_ONLY);

        result = statement.executeQuery(
            "SELECT * FROM dbo.Course WHERE title LIKE '%" + name +
            "%'");

        Course course;
```

ResultSet Example

```
    while (result.next()) {
        course = new Course(result.getString(1),
result.getString(2),
        result.getString(3), result.getString(4),
result.getInt(5));

        courses.add(course);
    }
} finally {
    if (statement != null) {
        statement.close();
    }

    if (result != null) {
        result.close();
    }
}

return courses;
}
```

ResultSet Example

```
public class CourseTest {  
  
    public static void main(String[] args) {  
        CourseDao courseDao = new CourseDaoImpl();  
        String name = "Java";  
  
        try {  
            List<Course> courses =  
courseDao.findCourseByName(name);  
            courses.forEach(c -> System.out.println(c));  
  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
  
    }  
  
}
```

JDBC Update using ResultSet

```
ResultSet rs = statement.executeQuery(query);  
...  
// for update  
rs.updateBoolean(1, false); // change the first column  
rs.updateInt("Age", 25); // change the column named "Age"  
rs.updateRow();  
  
// to delete  
rs.deleteRow();
```


Section 6

JDBC PREPAREDSTATEMENT

(with Parameter)

PreparedStatement Interface

- ❖ The **PreparedStatement** interface extends the **Statement** interface which gives you **added functionality** with a couple of advantages over a generic Statement object.

```
public interface PreparedStatement extends Statement {  
  
}
```

- ❖ It is used to execute **parameterized query**.
- ❖ **Improves performance:** The performance of the application will be faster if you use PreparedStatement interface because **query is compiled only once**.

PreparedStatement Interface

- ❖ The `prepareStatement()` method of `Connection` interface is used to return the object of `PreparedStatement`.

- ❖ **Syntax:**

```
public PreparedStatement prepareStatement(String query)  
throws SQLException{}
```

- ❖ This statement gives you the flexibility of supplying **arguments dynamically**.

```
PreparedStatement pstmt = null;  
try {  
    String SQL = "Update Employees SET age = ? WHERE id =  
?";  
    pstmt = conn.prepareStatement(SQL);  
} catch (SQLException e) {  
    //TODO  
}  
finally {  
    //TODO  
}
```

Methods of PreparedStatement interface

Method	Description
<code>public void setInt(int paramIndex, int value)</code>	Sets the integer value to the given parameter index.
<code>public void setString(int paramIndex, String value)</code>	Sets the String value to the given parameter index.
<code>public void setFloat(int paramIndex, float value)</code>	Sets the float value to the given parameter index.
<code>public void setDouble(int paramIndex, double value)</code>	Sets the double value to the given parameter index.
<code>public int executeUpdate()</code>	Executes the query. It is used for create, drop, insert, update, delete etc.
<code>public ResultSet executeQuery()</code>	Executes the select query. It returns an instance of ResultSet.

❖ The **setXXX()** methods bind values to the parameters.

❖ **Examples:**

```
pstmt.setInt(1, 23);
```

```
pstmt.setString(2, "Roshan");
```

```
pstmt.setString(3, "CEO");
```

```
pstmt.executeUpdate();
```

PreparedStatement Example

```
public boolean save(Course course) throws SQLException {  
  
    PreparedStatement preparedStatement = null;  
  
    Connection connection = null;  
  
    int result;  
    try {  
        connection = DBUtils.getConnection();  
  
        String query = "INSERT INTO dbo.Course VALUES (?, ?, ?, ?, ?)";  
        preparedStatement = connection.prepareStatement(query);  
  
        preparedStatement.setString(1, course.getCourseId());  
        preparedStatement.setString(2, course.getSubjectId());  
        preparedStatement.setString(3, course.getCourseCode());  
        preparedStatement.setString(4, course.getCourseTitle());  
        preparedStatement.setInt(5, course.getNumOfCredits());  
  
        result = preparedStatement.executeUpdate();  
  
    } finally {  
        if (preparedStatement != null) {  
            preparedStatement.close();  
        }  
        if (connection != null) {  
            connection.close();  
        }  
    }  
  
    return (result > 0);  
}
```

PreparedStatement Example

```
public static void main(String[] args) {  
    CourseDao courseDao = new CourseDaoImpl();  
  
    Course course = new Course("11119", "ITC", "1205",  
        "Java SE Programming Language", 5);  
  
    try {  
        boolean resultSave = courseDao.save(course);  
  
        System.out.println(resultSave);  
  
    } catch (SQLException e1) {  
        e1.printStackTrace();  
    }  
}
```

Results:

true

Section 7

JDBC CALLABLESTATEMENT

❖ **CallableStatement** interface is used to call the **stored procedures and functions**.

- ✓ We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.
- ✓ Example: you need the *get the age of the employee based on the date of birth*, you may create a function that receives date as the input and returns age of the employee as the output.

❖ **How to get the instance of CallableStatement:**

- ✓ The `prepareCall()` method of Connection interface returns the instance of CallableStatement.
- ✓ **Syntax:**

```
CallableStatement stmt=con.prepareCall("{call myprocedure(?,?) }");
```

CallableStatement Example

❖ For User Stored Procedure:

```
CREATE PROCEDURE usp_UpdateCourse(  
    @course_id VARCHAR(5),  
    @subject_id VARCHAR(4),  
    @course_code VARCHAR(10),  
    @title VARCHAR(50),  
    @number_of_credits INT,  
    @status INT OUTPUT  
)  
AS  
BEGIN  
    SET NOCOUNT ON;
```

```
UPDATE dbo.Course  
SET    subject_id = @subject_id,  
        title = @title,  
        number_of_credits = @number_of_credits  
WHERE course_code = @course_code;  
  
IF( @@ROWCOUNT > 0 )  
BEGIN  
    SET @status = 1;  
END  
ELSE  
BEGIN  
    SET @status = 0;  
END  
END
```

CallableStatement Example

```
public boolean update(Course course) throws SQLException {  
  
    CallableStatement callableStatement = null;  
  
    Connection connection = null;  
  
    int result;  
    try {  
        connection = DBUtils.getConnection();  
  
        callableStatement = connection  
            .prepareCall("{CALL usp_UpdateCourse(?,?,?,?,?,?)}");  
        callableStatement.setString(1, course.getCourseId());  
        callableStatement.setString(2, course.getSubjectId());  
        callableStatement.setString(3, course.getCourseCode());  
        callableStatement.setString(4, course.getCourseTitle());  
        callableStatement.setInt(5, course.getNumOfCredits());  
        callableStatement.registerOutParameter(6, Types.INTEGER);  
  
        callableStatement.execute();  
  
        result = callableStatement.getInt(6);  
    }  
}
```

CallableStatement Example

```
    } finally {  
        if (callableStatement != null) {  
            callableStatement.close();  
        }  
        if (connection != null) {  
            connection.close();  
        }  
    }  
  
    return (result > 0);  
}
```

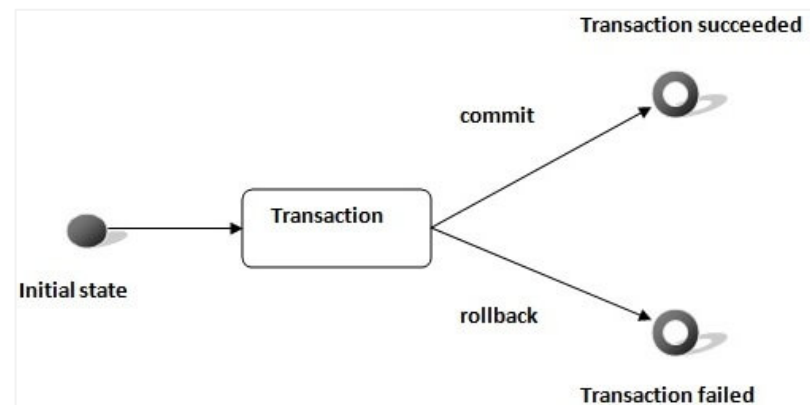
```
public class CourseTest {  
  
    public static void main(String[] args) {  
        CourseDao courseDao = new CourseDaoImpl();  
        Course course = new Course("11120", "ITCS", "1206", "Technologies for Java Web  
2", 5);  
        try {  
            boolean resultSave = courseDao.update(course);  
  
            System.out.println(resultSave);  
        } catch (SQLException e1) {  
            e1.printStackTrace();  
        }  
    }  
}
```

Section 8

TRANSACTION MANAGEMENT IN JDBC

Transaction

- ❖ Transaction represents a **single unit of work**.
- ❖ The **ACID** properties describes the transaction management well.
- ❖ ACID stands for Atomicity, Consistency, isolation and durability.
 - ✓ **Atomicity** means either all successful or none.
 - ✓ **Consistency** ensures bringing the database from one consistent state to another consistent state.
 - ✓ **Isolation** ensures that transaction is isolated from other transaction.
 - ✓ **Durability** means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.



- ❖ **Connection interface** provides methods to manage transaction.

Method	Description
void setAutoCommit (boolean status)	It is true by default means each transaction is committed by default.
void commit ()	Commits the transaction.
void rollback ()	Cancels the transaction.

- ❖ **Advantage of Transaction Management**

- ✓ **Fast performance:** It makes the performance fast because database is hit at the time of commit.

JDBC With Parameter

```
// in string query using Statement
String query = "INSERT INTO Person " +
               "VALUES (" + <name> + ", "
               <age> + ... + ")";

// using PreparedStatement
String query = "INSERT INTO Person " +
               "VALUES (?, ?)"

PreparedStatement statement = connect.prepareStatement(query);
connect.setAutoCommit(false);
statement.setString(1, "Titi");
statement.setInt(2, 25);
statement.executeQuery();           // insert 1
statement.setString(1, "Tata");
statement.setInt(2, 28);
statement.executeQuery();           // Insert 2
connect.commit();
connect.setAutoCommit(true);
```


Transaction Example

```
public boolean saveAll(List<Course> courses) throws SQLException
{
    PreparedStatement preparedStatement = null;
    Connection connection = null;

    int result = 0;
    try {
        connection = DBUtils.getConnection();
        connection.setAutoCommit(false);

        String query = "INSERT INTO dbo.Course VALUES (?, ?, ?, ?, ?)";

        preparedStatement = connection.prepareStatement(query);

        for (Course course : courses) {
            preparedStatement.setString(1, course.getCourseId());
            preparedStatement.setString(2, course.getSubjectId());
            preparedStatement.setString(3, course.getCourseCode());
            preparedStatement.setString(4, course.getCourseTitle());
            preparedStatement.setInt(5, course.getNumOfCredits());

            result += preparedStatement.executeUpdate();
        }
    }
```

Transaction Example

```
        connection.commit();

    } finally {
        if (preparedStatement != null) {
            preparedStatement.close();
        }
        if (connection != null) {
            connection.close();
        }
    }

    return (result == courses.size());
}
```

Section 9

BATCH PROCESSING IN JDBC

- ❖ Instead of executing a **single query**, we can execute a **batch (group) of queries**.
- ❖ It makes the performance fast.
- ❖ The **java.sql.Statement** and **java.sql.PreparedStatement** interfaces provide methods for batch processing.
- ❖ **Advantage of Batch Processing:** Fast Performance.
- ❖ **Methods of Statement interface:**

Method	Description
void addBatch (String query)	It adds query into batch.
int[] executeBatch ()	It executes the batch of queries.

JDBC Batch with String Query

❖ Step 1:

```
connect.setAutoCommit(false);
```

❖ Step 2:

```
Statement statement = connect.createStatement();  
statement.addBatch(<Insert query>);  
statement.addBatch(<Insert query>);  
statement.addBatch(<Update query>);  
statement.addBatch(<Delete query>);
```

❖ Step 3:

```
int[] updateCounts = statement.executeBatch();  
connect.commit();  
statement.close();  
connect.setAutoCommit(true);
```

JDBC Batch with PreparedStatement

```
public boolean saveBatch(List<Course> courses) throws SQLException {  
  
    PreparedStatement preparedStatement = null;  
  
    Connection connection = null;  
  
    int result[];  
    try {  
        connection = DBUtils.getConnection();  
  
        String query = "INSERT INTO dbo.Course VALUES (?, ?, ?, ?, ?)";  
  
        preparedStatement = connection.prepareStatement(query);  
  
        for (Course course : courses) {  
            preparedStatement.setString(1, course.getCourseId());  
            preparedStatement.setString(2, course.getSubjectId());  
            preparedStatement.setString(3, course.getCourseCode());  
            preparedStatement.setString(4, course.getCourseTitle());  
            preparedStatement.setInt(5, course.getNumOfCredits());  
  
            preparedStatement.addBatch();  
        }  
    }  
}
```

JDBC Batch with PreparedStatement

```
        result = preparedStatement.executeBatch();

    } finally {
        if (preparedStatement != null) {
            preparedStatement.close();
        }
        if (connection != null) {
            connection.close();
        }
    }

    return (result.length == courses.size());
}
```

JDBC Batch with PreparedStatement

```
public class CourseTest {  
  
    public static void main(String[] args) {  
        CourseDao courseDao = new CourseDaoImpl();  
  
        Course course1 = new Course("11129", "ITCS", "1219", "Java Web 9",  
5);  
        Course course2 = new Course("11128", "ITCS", "1218", "Java Web 8",  
5);  
  
        List<Course> courses = new ArrayList<Course>();  
        courses.add(course1);  
        courses.add(course2);  
  
        courseDao.print();  
  
        try {  
            boolean resultSave = courseDao.saveBatch(courses);  
  
            System.out.println(resultSave);  
  
        } catch (SQLException e1) {  
            System.err.println(e1.getLocalizedMessage());  
        }  
    }  
}
```


- **Java JDBC Tutorial**
- **Working steps**
- **DriverManager class**
- **JDBC Statement**
- **JDBC ResultSet**
- **JDBC PreparedStatement (with Parameter)**
- **Jdbc Callablestatement**
- **Transaction Management in JDBC**
- **Batch Processing in JDBC**