



Application Program Development

Segment : JDBC

Presenter Name




Agenda for Week 8

- Lecture
 - Introduction to JDBC
 - CRUD operations with JDBC
 - JavaFx with JDBC
 - Lab
 - Part 1 : In-Lab - Design and create a GUI based Application
 - Part 2 : DIY - using the GUI designed in-lab to write events
- 



Outcomes

- Understanding of why Database Programming?
 - Understanding architecture of JDBC.
 - Understanding of JDBC Interface
 - Understanding the Technology of JDBC
 - Connectivity
 - Read operations
 - Write operations
- 

Why Java for Database Programming?

- First, Java is platform independent. You can develop platform-independent database applications using SQL and Java for any relational database systems.
- Second, the support for accessing database systems from Java is built into Java API, so you can create database applications using all Java code with a common interface.
- Third, Java is taught in almost every university either as the first programming language or as the second programming language.

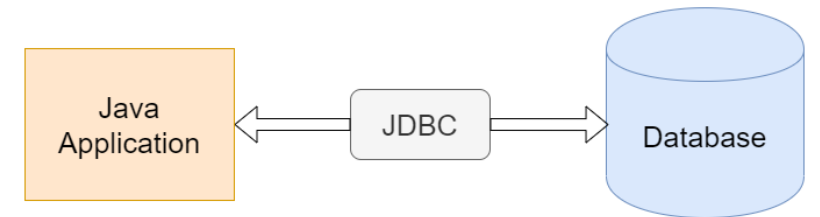
Database Applications Using Java

- GUI
- Client/Server
- Server-Side programming

What is JDBC?

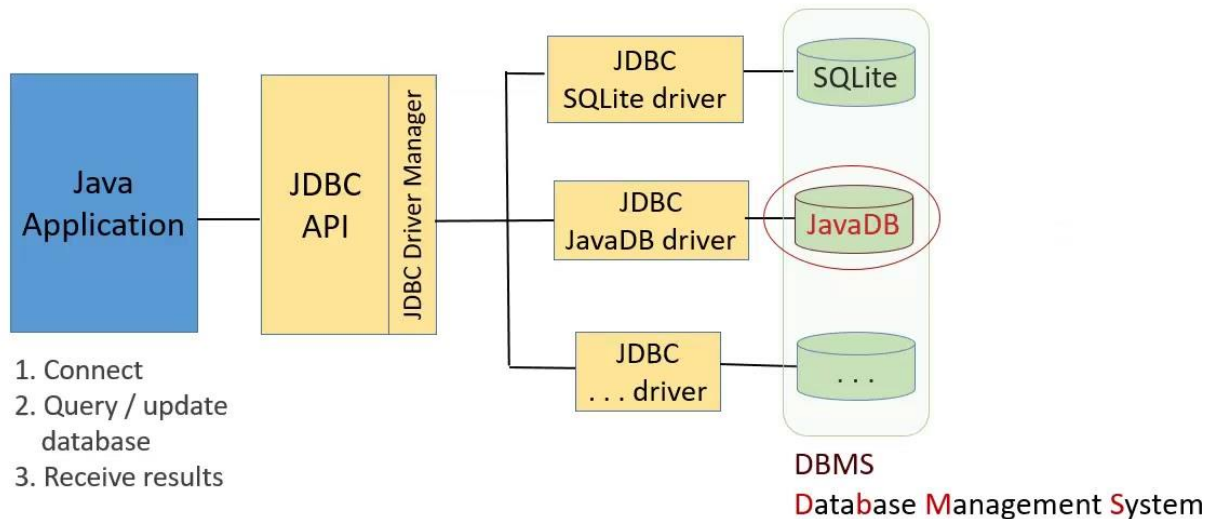
- Java Database Connectivity or JDBC API provides industry-standard and database-independent connectivity between the Java applications and relational database servers (relational databases, spreadsheets, and flat files).
- To keep it simple, JDBC allows a Java application to connect to a relational database. The major databases are supported such as Oracle, Microsoft SQL Server, DB2 and many others.

JDBC allows a Java application to connect to a relational database



The Architecture of JDBC

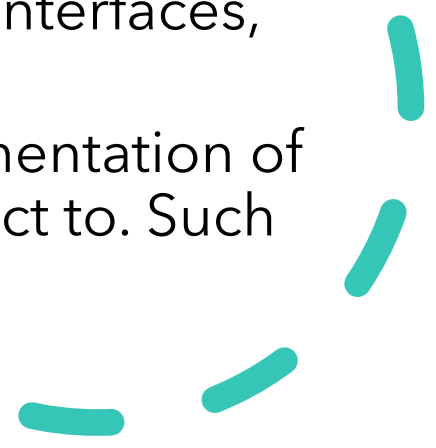
JDBC - Java Database Connectivity



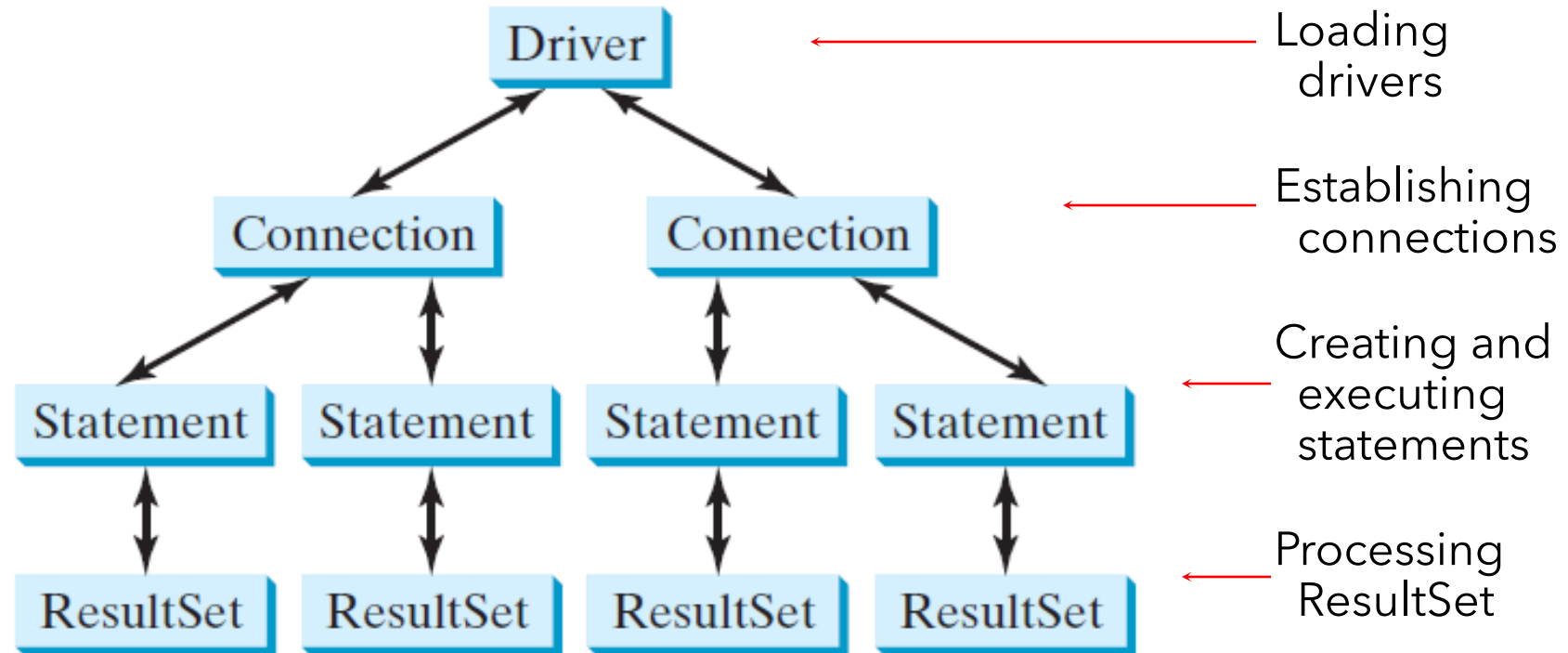
- JDBC helps you to write Java applications that manage these three programming activities:
 - Connect to a data source, like a database
 - Send queries and update statements to the database
 - Retrieve and process the results received from the database in answer to your query

JDBC API

- JDBC API consists of two packages
 - [java.sql](#)
 - We use java.sql package API for accessing and processing data stored in a data source (usually a relational database) using the Java programming language.
 - [javax.sql](#)
 - This is the JDBC driver(there are four different types of JDBC drivers) A JDBC driver is a set of Java classes that implement the JDBC interfaces, targeting a specific database.
 - The JDBC interfaces come 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*.



One Path of Core JDBC Interfaces



- **DriverManager** is used to load a *JDBC Driver*.
 - **Driver** is simply a java *library* containing classes that implement the JDBC API.
 - All JDBC drivers have to implement the same interface, it's not difficult to change the data source.
- **Connection** is being returned by the *DriverManager's* getConnection function.
- **Statement/ PreparedStatement**: In turn, a Connection is used to create a Statement, or to create and prepare a PreparedStatement or CallableStatement.
 - Statement and PreparedStatement objects are used to execute SQL statements.
 - CallableStatement objects are used to execute stored procedures.
 - A Connection can also be used to get a DatabaseMetaData object describing a database's functionality.

JDBC Driver Types


- A JDBC driver is a set of Java classes that implement the JDBC interfaces, targeting a specific database. The JDBC interfaces come with standard Java, but the implementation of these interfaces is specific to the database you need to connect to.
- There are 4 different types of JDBC drivers:
 1. Type 1: JDBC-ODBC bridge driver
 2. Type 2: Java + Native code driver
 3. Type 3: All Java + Middleware translation driver
 4. Type 4: All Java driver.



1. **Type 1:** JDBC-ODBC bridge driver:

- Drivers of this type are generally dependent on a native library, which limits their portability.
- The JDBC-ODBC Bridge should be considered a transitional solution. It is not supported by Oracle. Consider using this only if your DBMS does not offer a Java-only JDBC driver.

2. **Type 2:** Java + Native code driver:


- 
- This type of driver wraps a native API with Java classes. The Oracle Call Interface (OCI) driver is an example of a Type 2 driver. Because a Type 2 driver is implemented using local native code, it is expected to have better performance than a pure Java driver.



3. **Type 3:** All Java + Middleware translation driver

- This type of driver communicates using a network protocol to a middle-tier server. The middle tier in turn communicates to the database.
- Oracle does not provide a Type 3 driver. They do, however, have a program called Connection Manager that, when used in combination with Oracle's Type 4 driver, acts as a Type 3 driver in many respects.

4. **Type 4:** All Java driver.

- This type of driver, written entirely in Java, communicates directly with the database. No local native code is required.
 - Oracle's Thin driver is an example of a Type 4 driver.
- 

Installation of JDBC

- Installing a JDBC driver generally consists of copying the driver to your computer, then add the location of it to your classpath.
- In addition, many JDBC drivers other than Type 4 drivers require you to install a client-side API.
- No other special configuration is usually needed.

JDBC Technology

Four steps required to design apps with JDBC


- Connect to the database
- Create a statement
- Execute the query
- Look at the result set
- Close connection **// not needed if you are using try-with-resources**

Downloading the JDBC for SQLite

- You can [click here](#) to download the JDBC Driver for SQLite.
- You can also download and install SQLite browser for windows (GUI for SQLite)
- <http://sqlitebrowser.org/>



Creating Databases with JDBC in Java

- Create New project in Eclipse.
 - Add the sqlite jdbc jar file that we downloaded earlier to the library.
- 

Basic JDBC Connection

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class StatementExample {
    public static void main(String[] args) {
        try (Connection connection = DriverManager.getConnection("jdbc:sqlite:C:\\Users\\
            m_gur\\eclipse-workspace\\Winter2023\\APD545\\Practice\\JDBCConnectionExample
                \\src\\testDB.db")) {
            System.out.println(connection);
        } catch (SQLException e) {
            printSQLException(e);
        }
    }
    public static void printSQLException(SQLException ex) {
        for (Throwable e : ex) {
            if (e instanceof SQLException) {
                e.printStackTrace(System.err);
                System.err.println("SQLState: " + ((SQLException) e).getSQLState());
                System.err.println("Error Code: " + ((SQLException) e).getErrorCode());
                System.err.println("Message: " + e.getMessage());
                Throwable t = ex.getCause();
                while (t != null) {
                    System.out.println("Cause: " + t);
                    t = t.getCause();
                }
            }
        }
    }
}
```

Create Statement Object and execute

```
... ..  
import java.sql.Statement;  
  
Statement statement = connection.createStatement();  
statement.execute("Drop Table If Exists users");  
String query = "Create Table IF NOT EXISTS Users (id INTEGER NOT NULL  
PRIMARY KEY AUTOINCREMENT, username varchar(20) not null, email  
varchar(20) not null, country varchar(15), password varchar(20))";  
  
boolean success = statement.execute(query);  
if(success)  
    System.err.println("Table is not created as the resultSet is being  
                        returned");  
  
else  
    System.err.println("Table is created");  
  
... ..
```

Execute statement with Insert

```
. . .  
int value=0;  
  
query = "Insert into users (username, email, country, password) Values  
        ('mali','m@ali.com','Canada','1234')";  
  
value = statement.executeUpdate(query);  
  
if(value!=0)  
    System.err.println("1 row updated in the table");  
  
query = "Insert into users (username, email, country, password) Values  
        ('frank','frank@f.com','Canada','5678')";  
  
success = statement.execute(query);  
  
if(value!=0)  
    System.err.println("1 row updated in the table");
```

... ..

Execute statement with ResultSet

. . .

```
query = "Select * from users";
```

```
ResultSet rs = statement.executeQuery(query);
```

```
while(rs.next())
```

```
{
```

```
    System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3)+"  
    "+rs.getString(4)+" "+rs.getString(5));
```

```
}
```

... ..

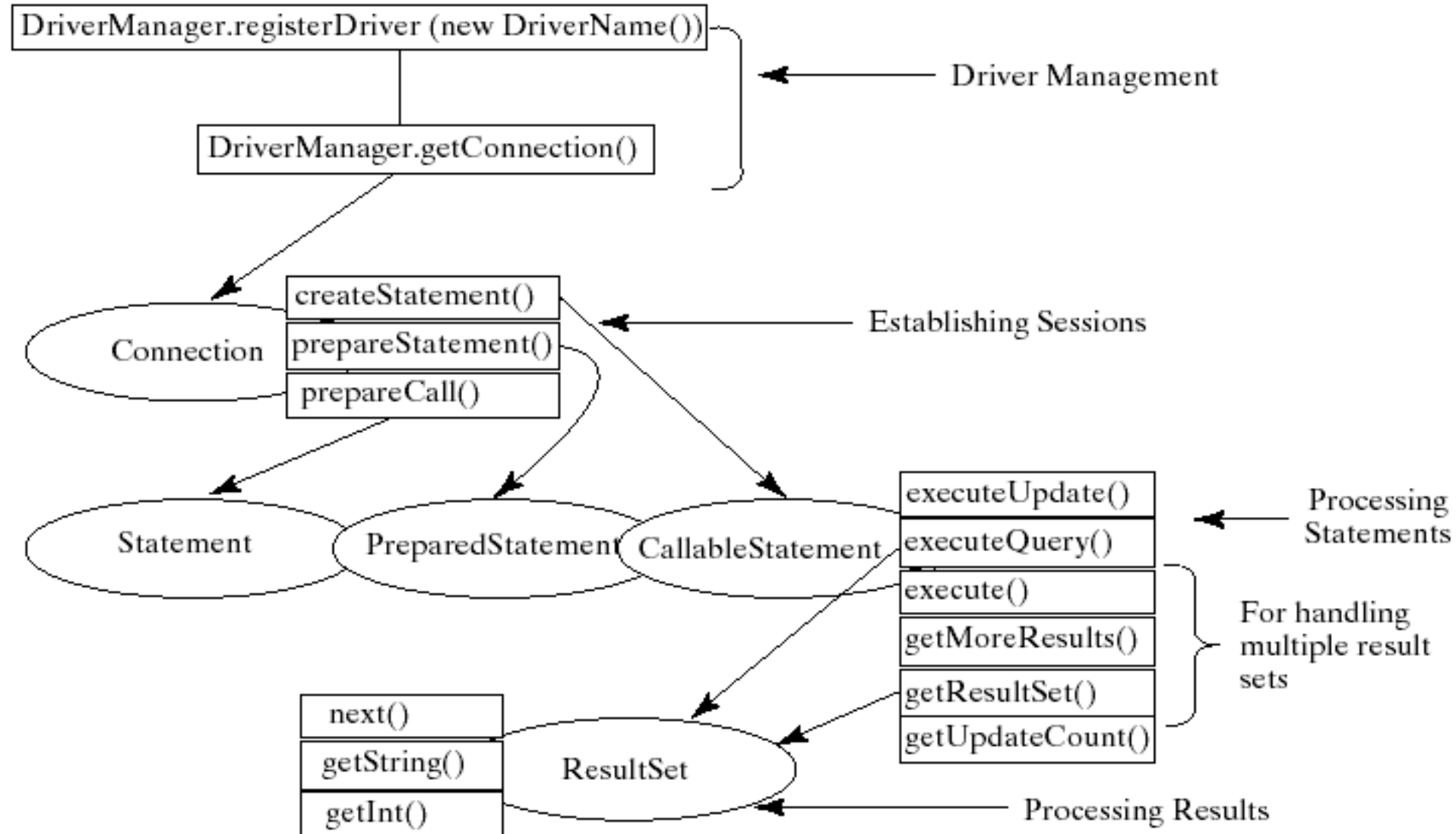
Prepared Statement

- Prepared statements can be used to insert, update, delete, or select data.
- However, prepared statements are precompiled statements that can be reused to execute identical SQL statements with different values more efficiently.
- They make only one trip to the database for metadata, whereas **statements** make a round trip with each execution.
- In addition, since bind variables are used, the database compiles and caches the prepared SQL statement and reuses it on subsequent executions to improve the database's performance.
- Prepared statements are also useful because some types of values, such as BLOBs, objects, collections, REFs, etc., are not representable as SQL text.
 - To support this added functionality, you use a question mark as a placeholder within the text of a SQL statement for values that you wish to specify when you execute that statement.
 - You can then replace that question mark with an appropriate value using one of the many available setXXX() accessor methods. setXXX() methods are available for setting every data type, just as getXXX() methods are available for getting the values for any data type from a result set.

Processing Statements

- Once a connection to a particular database is established, it can be used to
 - send SQL statements from your program to the database.
- JDBC provides the Statement,
 - PreparedStatement
 - CallableStatement interfaces
 - to facilitate sending statements to a database for execution and receiving execution results from the database more efficiently.

Processing Statements Diagram



The execute, executeQuery, and executeUpdate Methods

- The methods for executing SQL statements are
 - execute,
 - executeQuery, and
 - executeUpdate
- Each of which accepts a string containing a SQL statement as an argument.
- This string is passed to the database for execution.
- The execute method should be used if the execution produces
 - multiple result sets,
 - multiple update counts, or
 - a combination of result sets and update counts.

Benefits of using PreparedStatement

- PreparedStatement allows you to write a dynamic and parametric query.
 - By using PreparedStatement in Java you can write parameterized SQL queries and send different parameters by using the same SQL queries which is a lot better than creating different queries.
- PreparedStatement is faster than Statement in Java.
 - One of the major benefits of using PreparedStatement is better performance.
 - PreparedStatement gets pre-compiled
 - In database and their access plan is also cached in the database, which allows the database to execute parametric queries written using prepared statements much faster than normal queries because it has less work to do.
- PreparedStatement prevents SQL Injection attacks in Java
 - If you have been working in Java web applications you must be familiar with the infamous SQL Injection attacks, few years back Sony got a victim of SQL injection and compromised several Sony play station user data.
 - In an SQL Injection attack, malicious users pass SQL meta-data combined with input which allowed them to *execute SQL queries of their choice*, If not validated or prevented before sending a query to the database.
 - By using parametric queries and PreparedStatement you prevent many forms of SQL injection because all the parameters passed as part of the place-holder will be escaped automatically by JDBC Driver.
- At last PreparedStatement queries are more readable and secure than cluttered string concatenated queries.

Creating Table using PreparedStatement

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
public class PreparedStatementExample {
    public static void main(String[] args) {
        try (Connection connection = DriverManager.getConnection("jdbc:sqlite:C:\\Users\\m_gur\\eclipse-
                                                                    workspace\\Winter2023\\APD545\\Practice\\JDBCConnectionExample\\src\\testPreDB.db")) {

            System.out.println(connection);
            PreparedStatement ps = null;
            String query = "Create Table IF NOT EXISTS Books (id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT, bname varchar(20)"
                           + " not null, bcategory varchar(20) not null, bprice number(15), bisbn varchar(20))";

            ps = connection.prepareStatement(query);
            boolean success = ps.execute();
            if(success)
                System.err.println("Table is not created as the resultSet is being returned");
            else
                System.err.println("Table is created");
        } catch (SQLException e) {
            printSQLException(e);
        }
    }

    public static void printSQLException(SQLException ex) {
        for (Throwable e : ex) {
            if (e instanceof SQLException) {
                e.printStackTrace(System.err);
                System.err.println("SQLState: " + ((SQLException) e).getSQLState());
                System.err.println("Error Code: " + ((SQLException) e).getErrorCode());
                System.err.println("Message: " + e.getMessage());
                Throwable t = ex.getCause();
                while (t != null) {
                    System.out.println("Cause: " + t);
                    t = t.getCause();
                }
            }
        }
    }
}
```

PreparedStatement

The PreparedStatement interface is designed to execute dynamic SQL statements and SQL-stored procedures with "IN" parameters. These SQL statements and stored procedures are precompiled for efficient use when repeatedly executed.

```
Statement pstmt = connection.prepareStatement ("insert into Student (firstName, mi, lastName) +  
values (?, ?, ?)");
```

Insert Data using PreparedStatement

```
... ..  
int value=0;  
query = "Insert into books (bname, bcategory, bprice, bisbn) Values (?, ?, ?, ?)";  
ps = connection.prepareStatement(query);  
ps.setString(1, "Harry Potter");  
ps.setString(2, "Fantasy");  
ps.setDouble(3, 65.3);  
ps.setString(4, "123456789");  
value = ps.executeUpdate();  
  
if(value!=0)  
System.err.println("1 row updated in the table");  
ps.setString(1, "Dog Man");  
ps.setString(2, "Fantasy");  
ps.setDouble(3, 9.3);  
ps.setString(4, "123456789");  
value = ps.executeUpdate();  
if(value!=0)  
System.err.println("1 row updated in the table");  
... ..
```

Select Data using PreparedStatement

... ..

```
query = "Select * from books where bname = ?";
ps = connection.prepareStatement(query);
ps.setString(1, "Harry Potter");
ResultSet rs = ps.executeQuery();
while(rs.next())
{
    System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3)+"
"+rs.getString(4)+" "+rs.getString(5));
}
```

... ..