

Build COMPETENCY
across your TEAM



Microsoft Partner
Gold Cloud Platform
Silver Learning

Java Database Connectivity



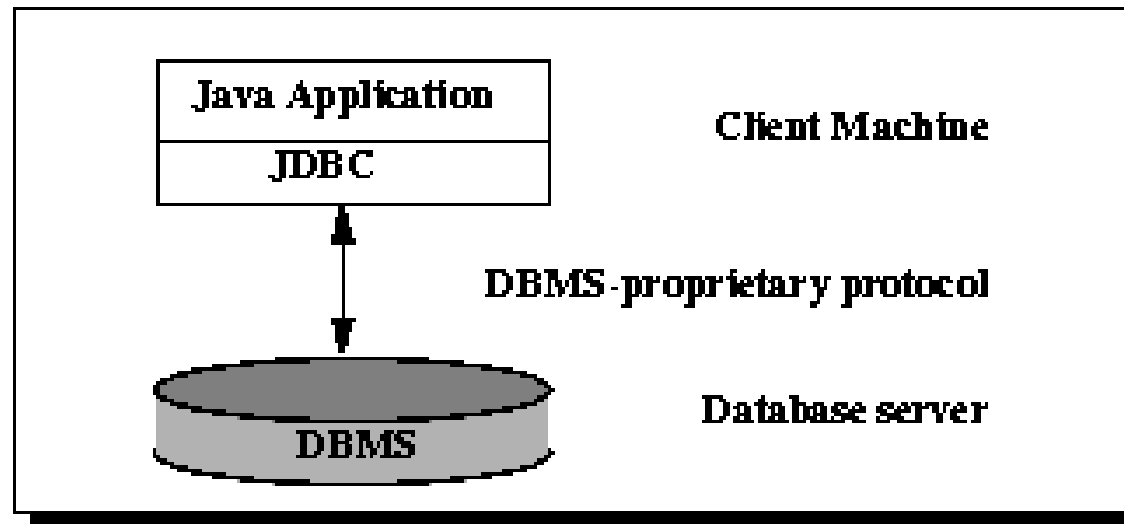
A cluster of approximately 15 light gray squares of various sizes, some overlapping, arranged in a loose, abstract pattern in the upper right quadrant of the slide.

JDBC APIs Architecture
Database Access Steps
Calling database procedures
Using Transaction
Connection Pooling
Working with Rowsets
DAO Design Pattern
Security



Two-tier Architecture

- The JDBC API supports both two-tier and three-tier processing models for database access.
- **Figure 1: Two-tier Architecture for Data Access.**

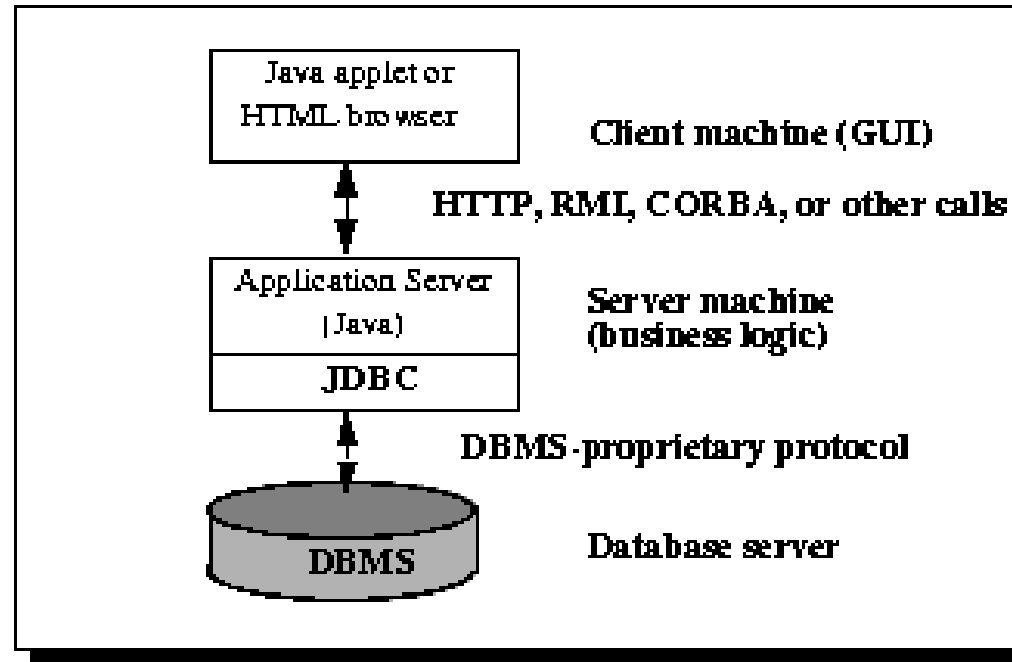


Two-tier Architecture

- In the two-tier model, a Java application talks directly to the data source.
- This requires a JDBC driver that can communicate with the particular data source being accessed.
- A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user.
- The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server.
- The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

Three-tier Architecture for Data Access.

- **Figure 2: Three-tier Architecture for Data Access.**



Three-tier Architecture for Data Access.

- In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source.
- The data source processes the commands and sends the results back to the middle tier, which then sends them to the user.
- MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data.
- Another advantage is that it simplifies the deployment of applications.
- Finally, in many cases, the three-tier architecture can provide performance advantages.

What's new in JDBC 4.0

JDBC 4.0 is new and advance specification of JDBC. It provides the following advance features

- Connection Management
- Auto loading of Driver Interface.
- Better exception handling
- Support for large object
- Annotation in SQL query.

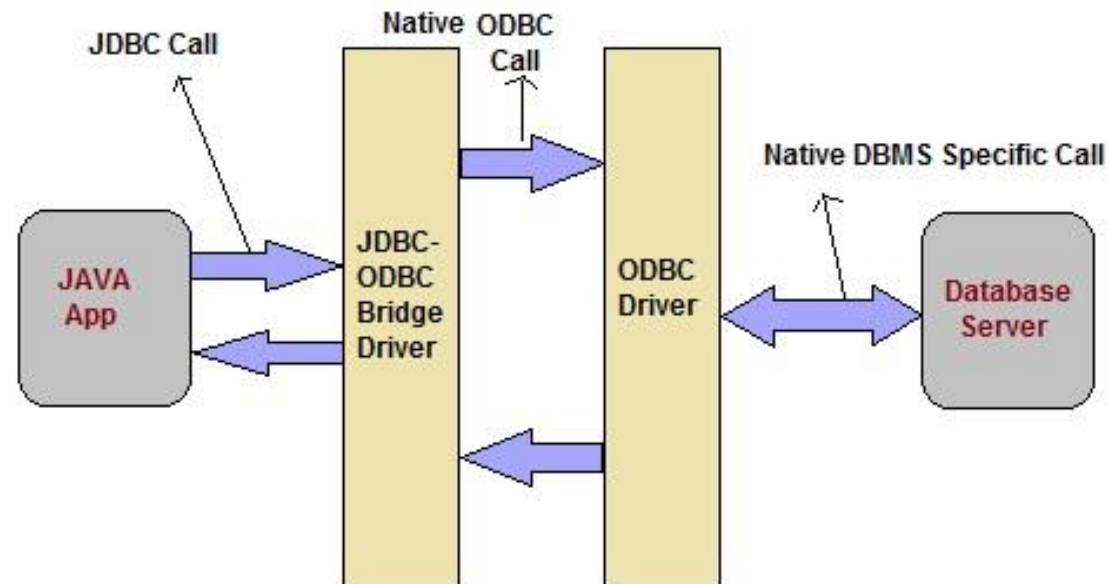
JDBC Driver

JDBC Driver is required to process SQL requests and generate result. The following are the different types of driver available in JDBC.

- **Type-1 Driver** or **JDBC-ODBC bridge**
- **Type-2 Driver** or **Native API Partly Java Driver**
- **Type-3 Driver** or **Network Protocol Driver**
- **Type-4 Driver** or **Thin Driver**
-

Type 1 Driver: JDBC-ODBC bridge

- **Type-1 Driver** act as a bridge between JDBC and other database connectivity mechanism(ODBC). This driver converts JDBC calls into ODBC calls and redirects the request to the ODBC driver.



Type 1 Driver: JDBC-ODBC bridge

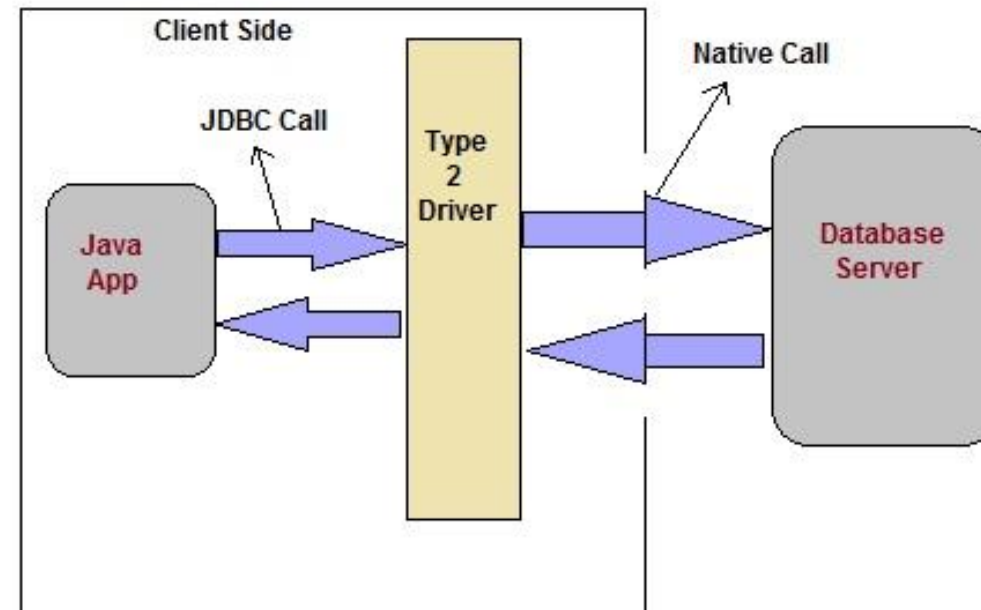
- **Advantage**
- Easy to use
- Allow easy connectivity to all database supported by the ODBC Driver.
- **Disadvantage**
- Slow execution time
- Dependent on ODBC Driver.
- Uses Java Native Interface(JNI) to make ODBC call.

Type2 Driver: Native API Driver

- This type of driver make use of Java Native Interface(JNI) call on database specific native client API. These native client API are usually written in C and C++.

Type 2 Driver: Native API Driver

- This type of driver make use of Java Native Interface(JNI) call on database specific native client API. These native client API are usually written in C and C++.



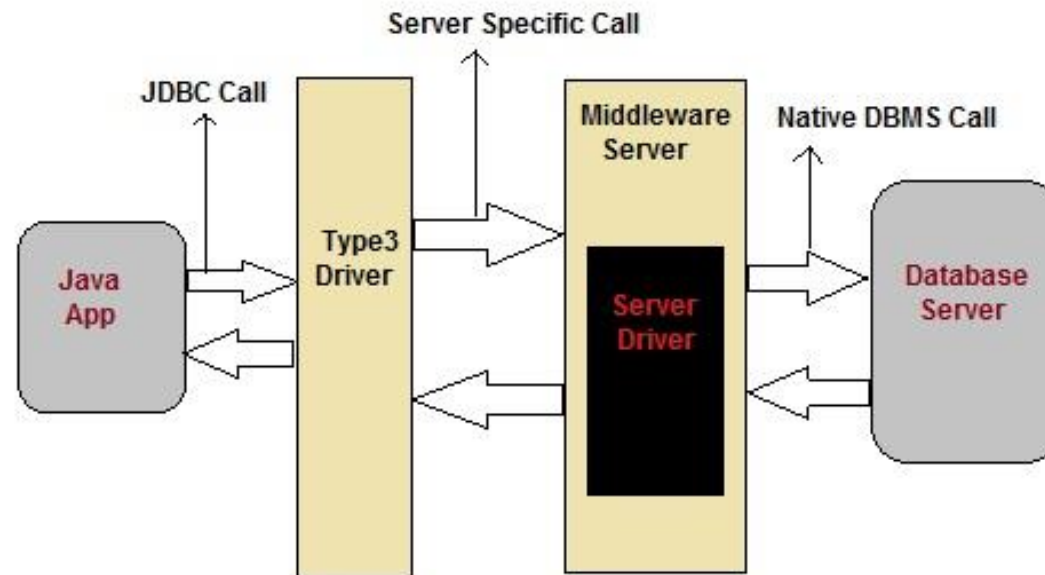
Type 2 Driver: Native API Driver

- **Advantage**
- faster as compared to **Type-1 Driver**
- Contains additional features.
- **Disadvantage**
- Requires native library
- Increased cost of Application



Type 3 Driver: Network Protocol Driver

- This driver translates the JDBC calls into a database server independent and Middleware server-specific calls. Middleware server further translates JDBC calls into database specific calls.

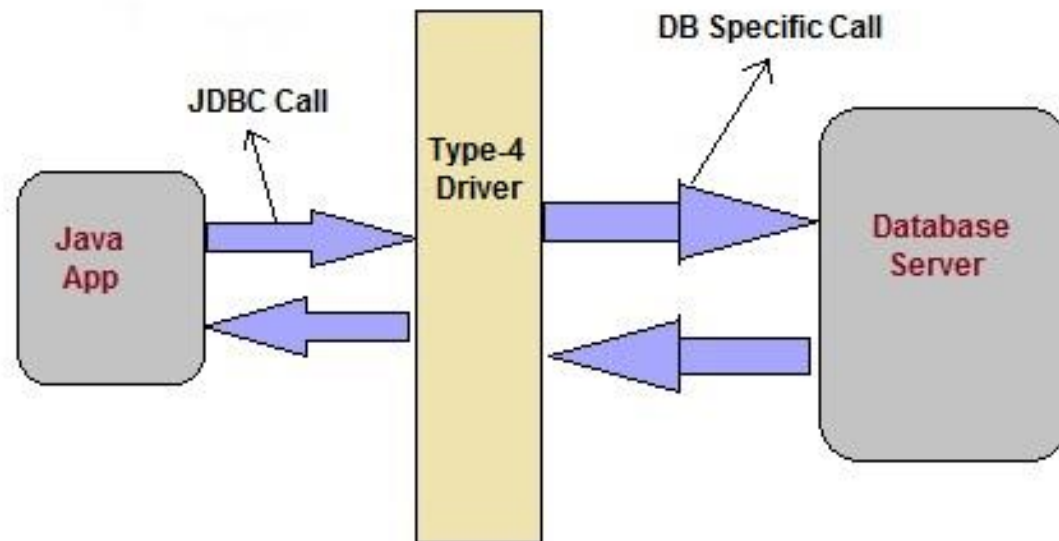


Type 3 Driver: Network Protocol Driver

- **Advantage**
- Does not require any native library to be installed.
- Database Independency.
- Provide facility to switch over from one database to another database.
- **Disadvantage**
- Slow due to increase number of network call.

Type 4 Driver: Thin Driver

- This is Driver called Pure Java Driver because. This driver interact directly with database. It does not require any native database library, that is why it is also known as Thin Driver.



Type 4 Driver: Thin Driver

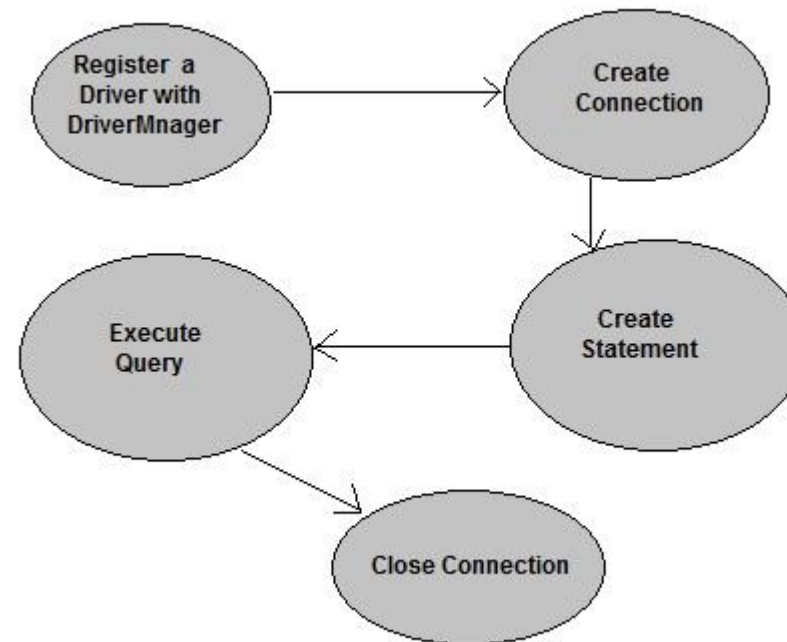
- **Advantage**
- Does not require any native library.
- Does not require any Middleware server.
- Better Performance than other driver.
- **Disadvantage**
- Slow due to increase number of network call.



Steps to connect a Java Application to Database

The following 5 steps are the basic steps involve in connecting a Java application with Database using JDBC.

- Register the Driver
- Create a Connection
- Create SQL Statement
- Execute SQL Statement
- Closing the connection



Register the Driver

- Class.forName() is used to load the driver class explicitly.
- Example to register with JDBC-ODBC Driver
- `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`

Create a Connection

- getConnection() method of DriverManager class is used to create a connection.
- Syntax
 - getConnection(String url)
 - getConnection(String url, String username, String password)
 - getConnection(String url, Properties info)
 - Example establish connection with Oracle Driver
- Connection con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","username","password");

Create SQL Statement

- createStatement() method is invoked on current Connection object to create a SQL Statement.
- Syntax
- public Statement createStatement() throws SQLException
- Example to create a SQL statement
- Statement s=con.createStatement();

Execute SQL Statement

- executeQuery() method of Statement interface is used to execute SQL statements.
- Syntax
- public ResultSet executeQuery(String query) throws SQLException
- Example to execute a SQL statement
- ```
ResultSet rs=s.executeQuery("select * from user");
```
- ```
while(rs.next())
```
- ```
{
```
- ```
System.out.println(rs.getString(1)+" "+rs.getString(2));
```
- ```
}
```

# Closing the connection

- After executing SQL statement you need to close the connection and release the session. The close() method of Connection interface is used to close the connection.
- Syntax
- public void close() throws SQLException
- Example of closing a connection
- `con.close();`

# Connecting to Oracle Database using Thin Driver

- Load Driver Class: The Driver Class for oracle database is `oracle.jdbc.driver.OracleDriver` and `Class.forName("oracle.jdbc.driver.OracleDriver")` method is used to load the driver class for Oracle database.
- **Create Connection:** For creating a connection you will need a Connection URL. The Connection URL for Oracle is

**jdbc:oracle:thin:@localhost:1521:XE**

↓   ↓   ↓   ↓   ↓   ↓

API   Database   Driver Type   server name on which oracle is running   Port Number   Oracle Service Name



You will also require **Username** and **Password** of your Oracle Database Server for creating connection.

- Loading jar file: To connect your java application with Oracle, you will also need to load ojdbc14.jar file. This file can be loaded into 2 ways.

Copy the jar file into C:\Program Files\Java\jre7\lib\ext folder.

or,

Set it into classpath. For more detail see how to set classpath

# Accessing record from Student table in Java application



```
import java.sql.*;

class Test
{
 public static void main(String []args)
 {
 try{
 //Loading driver
 Class.forName("oracle.jdbc.driver.OracleDriver");

 //creating connection
 Connection con = DriverManager.getConnection
 ("jdbc:oracle:thin:@localhost:1521:XE","username","password");

 Statement s=con.createStatement(); //creating statement

 ResultSet rs=s.executeQuery("select * from Student"); //executing statement

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

 con.close(); //closing connection
 }catch(Exception e){
 e.printStackTrace();
 }
 }
}
```

# Inserting record into a table using java application

```
import java.sql.*;

class Test
{
 public static void main(String []args)
 {
 try{
 //Loading driver...
 Class.forName("oracle.jdbc.driver.OracleDriver");

 //creating connection...
 Connection con = DriverManager.getConnection
 ("jdbc:oracle:thin:@localhost:1521:XE","username","password");

 PreparedStatement pst=con.prepareStatement("insert into Student values(?,?)");

 pst.setInt(1,104);
 pst.setString(2,"Alex");

 pst.executeUpdate();

 con.close(); //closing connection
 }catch(Exception e){
 e.printStackTrace();
 }
 }
}
```



# CallableStatement in Java

- CallableStatement in java is used to call stored procedure from java program. Stored Procedures are group of statements that we compile in the database for some task.
- Stored procedures are beneficial when we are dealing with multiple tables with complex scenario and rather than sending multiple queries to the database, we can send required data to the stored procedure and have the logic executed in the database server itself.

# CallableStatement in Java

- JDBC API provides support to execute Stored Procedures through CallableStatement interface.
- Stored Procedures requires to be written in the database specific syntax and for my tutorial, I will use Oracle database. We will look into standard features of CallableStatement with IN and OUT parameters.

## Step 1: Create table in database

CREATE TABLE EMPLOYEE

```
(
 "EMPID" NUMBER NOT NULL ENABLE,
 "NAME" VARCHAR2(10 BYTE) DEFAULT NULL,
 "ROLE" VARCHAR2(10 BYTE) DEFAULT NULL,
 "CITY" VARCHAR2(10 BYTE) DEFAULT NULL,
 "COUNTRY" VARCHAR2(10 BYTE) DEFAULT NULL,
 PRIMARY KEY ("EMPID")
);
```

## Step 2: Create Procedure

```
CREATE OR REPLACE PROCEDURE insertEmployee
(in_id IN EMPLOYEE.EMPID%TYPE,
 in_name IN EMPLOYEE.NAME%TYPE,
 in_role IN EMPLOYEE.ROLE%TYPE,
 in_city IN EMPLOYEE.CITY%TYPE,
 in_country IN EMPLOYEE.COUNTRY%TYPE,
 out_result OUT VARCHAR2)
AS
BEGIN
 INSERT INTO EMPLOYEE (EMPID, NAME, ROLE, CITY, COUNTRY)
 values (in_id,in_name,in_role,in_city,in_country);
 commit;

 out_result := 'TRUE';

EXCEPTION
 WHEN OTHERS THEN
 out_result := 'FALSE';
 ROLLBACK;
END;
```



## Step 3 : Write an Java Application


```
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Scanner;
```

```
public class JDBCStoredProcedureWrite {


 public static void main(String[] args) {
 Connection con = null;
 CallableStatement stmt = null;

 //Read User Inputs
 Scanner input = new Scanner(System.in);
 System.out.println("Enter Employee ID (int):");
```





```
int id = Integer.parseInt(input.nextLine());
System.out.println("Enter Employee Name:");
String name = input.nextLine();
System.out.println("Enter Employee Role:");
String role = input.nextLine();
System.out.println("Enter Employee City:");
String city = input.nextLine();
System.out.println("Enter Employee Country:");
String country = input.nextLine();
```



try{

```
con = DBConnection.getConnection();
stmt = con.prepareCall("{call insertEmployee(?,?,?,?,?,?)}");
stmt.setInt(1, id);
stmt.setString(2, name);
stmt.setString(3, role);
stmt.setString(4, city);
stmt.setString(5, country);
```

```
//register the OUT parameter before calling the stored procedure
stmt.registerOutParameter(6, java.sql.Types.VARCHAR);

stmt.executeUpdate();

//read the OUT parameter now
String result = stmt.getString(6);

System.out.println("Employee Record Save Success:."+result);
```

```
}catch(Exception e){
 e.printStackTrace();
}finally{
 try {
 stmt.close();
 con.close();
 input.close();
 } catch (SQLException e) {
 e.printStackTrace();
 }
}
}
}
```

# CallableStatement – Stored Procedure OUT Parameters



## Step 1 Create a procedure

```
create or replace
PROCEDURE getEmployee
(in_id IN EMPLOYEE.EMPID%TYPE,
 out_name OUT EMPLOYEE.NAME%TYPE,
 out_role OUT EMPLOYEE.ROLE%TYPE,
 out_city OUT EMPLOYEE.CITY%TYPE,
 out_country OUT EMPLOYEE.COUNTRY%TYPE
)
AS
BEGIN
 SELECT NAME, ROLE, CITY, COUNTRY
 INTO out_name, out_role, out_city, out_country
 FROM EMPLOYEE
 WHERE EMPID = in_id;

END;
```

## Step 2 : Java Application

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Scanner;
```

```
public class JDBCStoredProcedureRead {
 public static void main(String[] args) {
 Connection con = null;
 CallableStatement stmt = null;

 //Read User Inputs
 Scanner input = new Scanner(System.in);
 System.out.println("Enter Employee ID (int):");
 int id = Integer.parseInt(input.nextLine());
```

try{

```
con = DBConnection.getConnection();
stmt = con.prepareCall("{call getEmployee(?,?,?,?,?)}");
stmt.setInt(1, id);
```

procedure

```
//register the OUT parameter before calling the stored

stmt.registerOutParameter(2, java.sql.Types.VARCHAR);
stmt.registerOutParameter(3, java.sql.Types.VARCHAR);
stmt.registerOutParameter(4, java.sql.Types.VARCHAR);
stmt.registerOutParameter(5, java.sql.Types.VARCHAR);
```





```
stmt.execute();
```

```
//read the OUT parameter now
```

```
String name = stmt.getString(2);
```

```
String role = stmt.getString(3);
```

```
String city = stmt.getString(4);
```

```
String country = stmt.getString(5);
```

```
if(name !=null){
```

```
 System.out.println("Employee
```

```
Name="+name+",Role="+role+",City="+city+",Country="+country);
```



```
 }else{
 System.out.println("Employee Not Found with ID"+id);
 }
}catch(Exception e){
 e.printStackTrace();
}finally{
 try {
 stmt.close();
 con.close();
 input.close();
 } catch (SQLException e) {
 e.printStackTrace();
 }
}
}
}
```

# JDBC Transaction

- JDBC Transaction let you control how and when a transaction should commit into database.
- //transaction block start
- //SQL insert statement
- //SQL update statement
- //SQL delete statement
- //transaction block end

- In simple, JDBC transaction make sure SQL statements within a transaction block are all executed successful, if either one of the SQL statement within transaction block is failed, abort and rollback everything within the transaction block.

A cluster of overlapping squares in various shades of gray and blue, arranged in a non-uniform pattern in the upper right area of the slide.

To put this in a transaction, you can use

- `dbConnection.setAutoCommit(false);` to start a transaction block.
- `dbConnection.commit();` to end a transaction block.

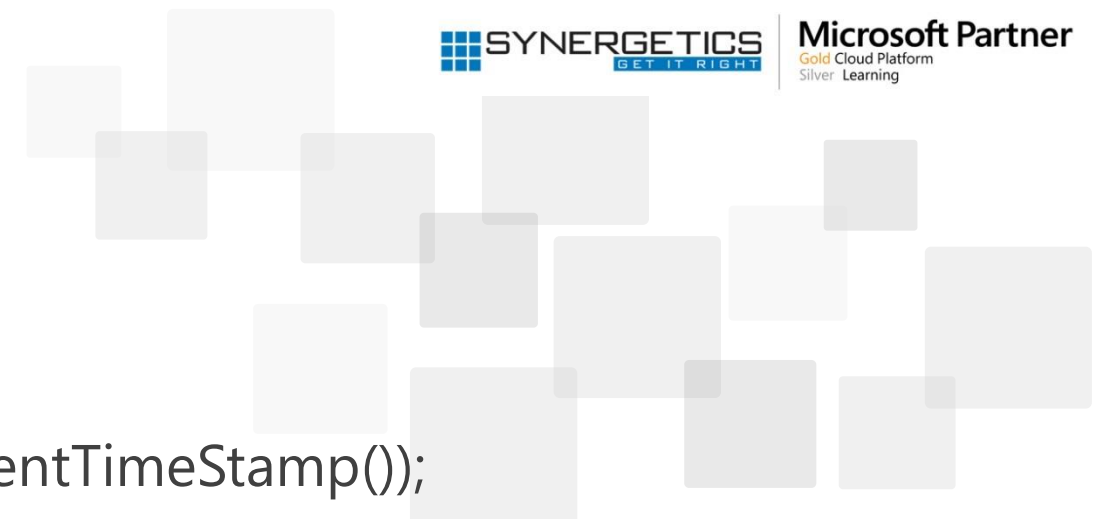


```
dbConnection.setAutoCommit(false); //transaction block start
```

```
String insertTableSQL = "INSERT INTO DBUSER"
 + "(USER_ID, USERNAME, CREATED_BY, CREATED_DATE) VALUES"
 + "(?,?,?,?)";
```

```
String updateTableSQL = "UPDATE DBUSER SET USERNAME =? "
 + "WHERE USER_ID = ?";
```


```
preparedStatementInsert = dbConnection.prepareStatement(insertTableSQL);
preparedStatementInsert.setInt(1, 999);
```



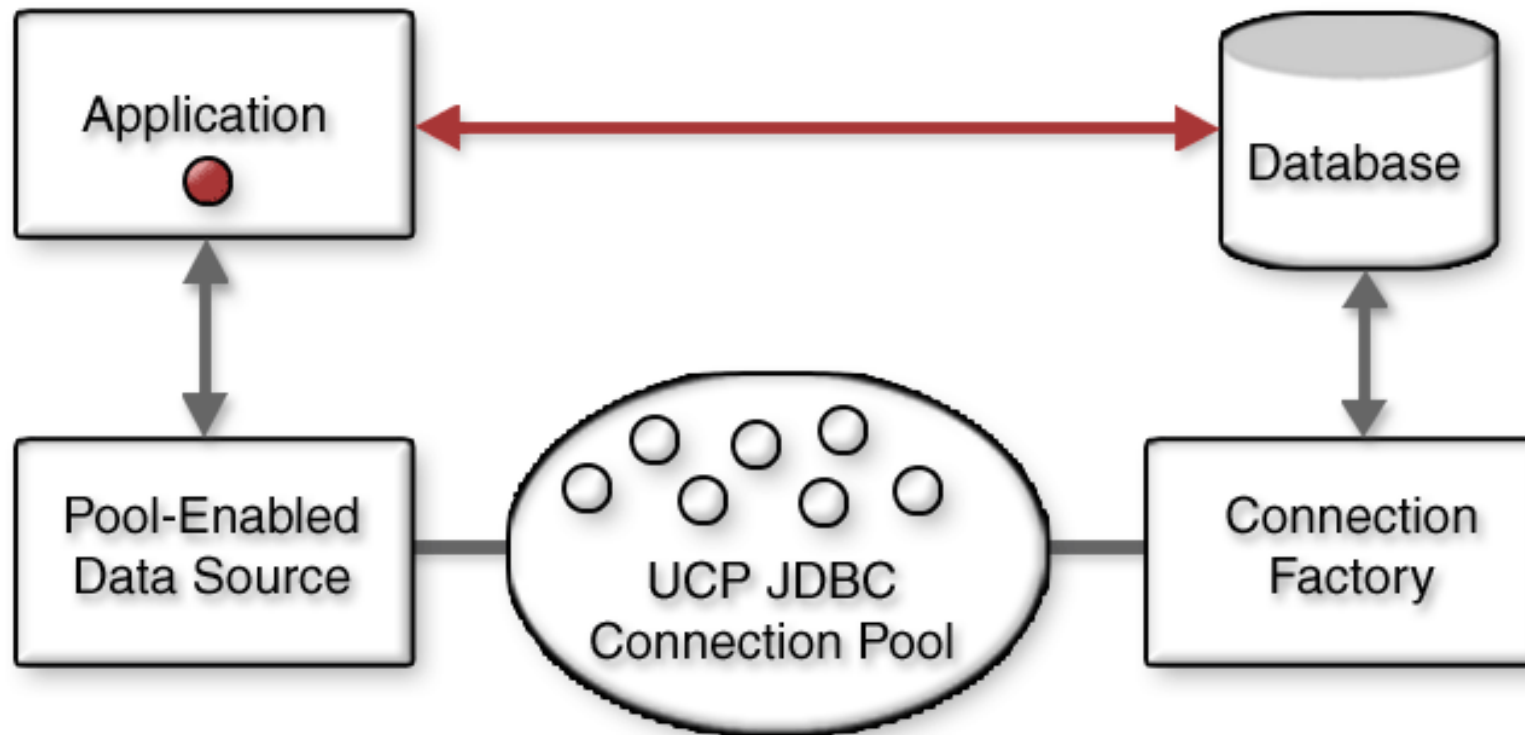
```
preparedStatementInsert.setString(2, "amit123");
preparedStatementInsert.setString(3, "system");
preparedStatementInsert.setTimestamp(4, getCurrentTimeStamp());
preparedStatementInsert.executeUpdate(); //data IS NOT commit yet
```

```
preparedStatementUpdate = dbConnection.prepareStatement(updateTableSQL);
preparedStatementUpdate.setString(1, "A very very long string caused DATABASE ERROR");
preparedStatementUpdate.setInt(2, 999);
preparedStatementUpdate.executeUpdate(); //Error, rollback, including the first insert
statement.
```

```
dbConnection.commit(); //transaction block end
```



# Connection Pooling





- JDBC Connection pooling is similar to any other object pooling. Connection pooling is very useful for any application which uses database database as backend.
- Database connection is very expensive to create over network, for this we need to create a network connection, then initializing a database, creating a session, doing transaction and then after closing the connection, this take more time, therefore application becomes slower.
- The valuable database resources such as memory, cursors, locks , temporary tables all tends to increase on numbers of concurrent connections.

- In connection pooling, we create limited numbers of connection objects pools at a time, such as 10 connections, 50 connections, 100 connections etc.
- This depends upon the capacity of the database that how much connections it can handle at a time. If any request comes we allocate a connection object to it.
- When it completed their work then it releases the connection object and this object is added into the pool.

# MainClaz

```
import java.sql.*;

public class MainClaz {
 public static void main(String[] args) {
 try {
 Class.forName("com.mysql.jdbc.Driver");
 } catch (Exception E) {
 System.out.println("Unable to load a driver " + E.toString());
 }
 try {
 JDBCConnectionPooling connectionPooling = new
JDBCConnectionPooling(
 "jdbc:mysql://localhost:3306/student", "root", "root");
```

```
Connection[] connArr = new Connection[7];

 for (int i = 0; i < connArr.length; i++) {
 connArr[i] = connectionPooling.connectionCheck();
 System.out.println("Checking Connections..." + connArr[i]);
 System.out.println("Connections Available... "
 + connectionPooling.availableCount());
 }
} catch (SQLException sqle) {
 System.out.println(sqle.toString());
} catch (Exception e) {
 System.out.println(e.toString());
}

}
```

# JDBCConnectionPooling

```
import java.util.*;
import java.sql.*;
```

```
public class JDBCConnectionPooling implements Runnable {
 int initialConnections = 5;
 Vector connectionsAvailable = new Vector();
 Vector connectionsUsed = new Vector();

 String connectionUrl ;
 String userName;
 String userPassword ;
 public JDBCConnectionPooling(String url,String userName, String userPass) throws
 SQLException {
 try {
```

# JDBCConnectionPooling

```
 this.connectionUrl = url;
 this.userName = userName;
 this.userPassword = userPass;
 Class.forName("com.mysql.jdbc.Driver");
 for (int count = 0; count < initialConnections; count++) {
 connectionsAvailable.addElement(getConnection());
 }
 } catch (ClassNotFoundException e) {
 System.out.println(e.toString());
 }
}

private Connection getConnection() throws SQLException {
 return DriverManager.getConnection(connectionUrl, userName,
 userPassword);
}
```

# JDBCConnectionPooling

```
}
public synchronized Connection connectionCheck() throws SQLException {
 Connection newConnection = null;
 if (connectionsAvailable.size() == 0) {
 // creating a new Connection
 newConnection = getConnection();
 // adding Connection to used list
 connectionsUsed.addElement(newConnection);
 } else {
 newConnection = (Connection) connectionsAvailable.lastElement();

 connectionsAvailable.removeElement(newConnection);

 connectionsUsed.addElement(newConnection);
 }
 return newConnection;
}
```

# JDBCConnectionPooling

```
public int availableCount() {
 return connectionsAvailable.size();
}
public void run() {
 try {
 while (true) {
 synchronized (this) {
 while (connectionsAvailable.size() > initialConnections) {
 Connection connection = (Connection) connectionsAvailable
 .lastElement();
 connectionsAvailable.removeElement(connection);

 connection.close();
 }
 }
 }
 }
}
```



# JDBCConnectionPooling

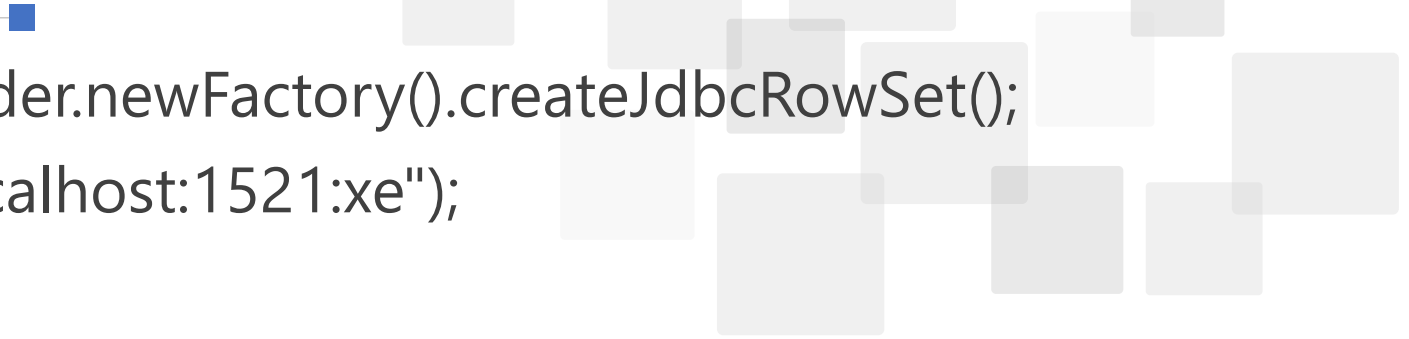
```
 } catch (SQLException sqle) {
 sqle.printStackTrace();
 } catch (Exception e) {
 e.printStackTrace();
 }
 }
}
```



- The instance of RowSet is the java bean component because it has properties and java bean notification mechanism. It is introduced since JDK 5.
- It is the wrapper of ResultSet. It holds tabular data like ResultSet but it is easy and flexible to use.


The implementation classes of RowSet interface are as follows:

- JdbcRowSet
- CachedRowSet
- WebRowSet
- JoinRowSet
- FilteredRowSet



```
JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
rowSet.setUsername("system");
rowSet.setPassword("oracle");

rowSet.setCommand("select * from emp400");
rowSet.execute();
```



# Advantage of RowSet

- The advantages of using RowSet are given below:
- It is easy and flexible to use
- It is Scrollable and Updatable by default



# RowSetExample

```
public class RowSetExample {
 public static void main(String[] args) throws Exception {
 Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
//Creating and Executing RowSet
```

```
JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
rowSet.setUsername("system");
rowSet.setPassword("oracle");
```

```
rowSet.setCommand("select * from emp400");
rowSet.execute();
```

# RowSetExample

```
while (rowSet.next()) {
 // Generating cursor Moved event
 System.out.println("Id: " + rowSet.getString(1));
 System.out.println("Name: " + rowSet.getString(2));
 System.out.println("Salary: " + rowSet.getString(3));
}

}

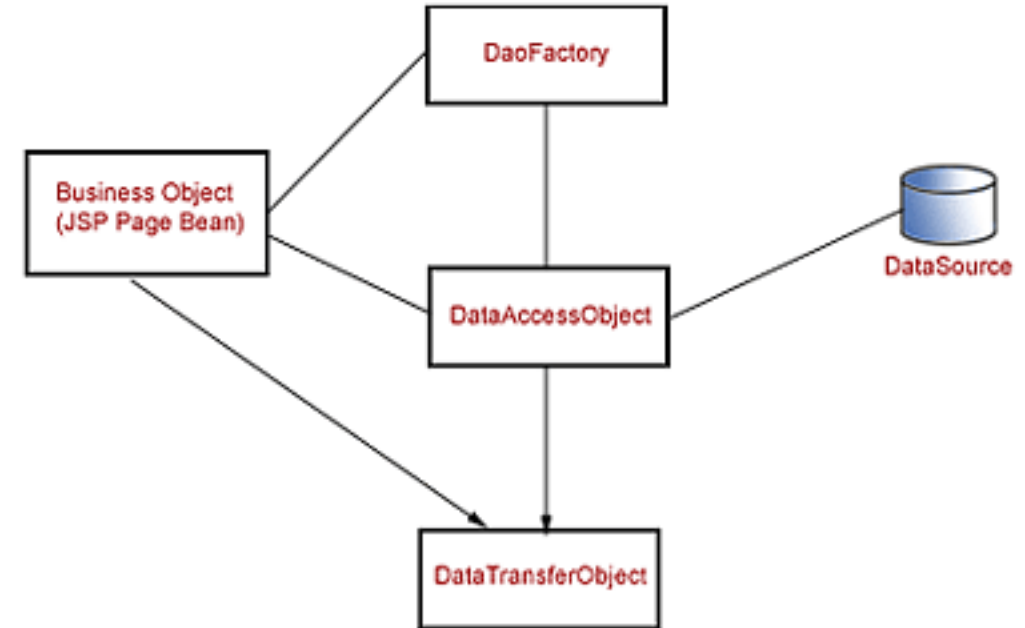
}
```

# DAO Design Pattern

DAO stands for Data Access Object. DAO Design Pattern is used to separate the data persistence logic in a separate layer.

This way, the service remains completely in dark about how the low-level operations to access the database is done.

This is known as the principle of Separation of Logic.





# DAO Design Pattern

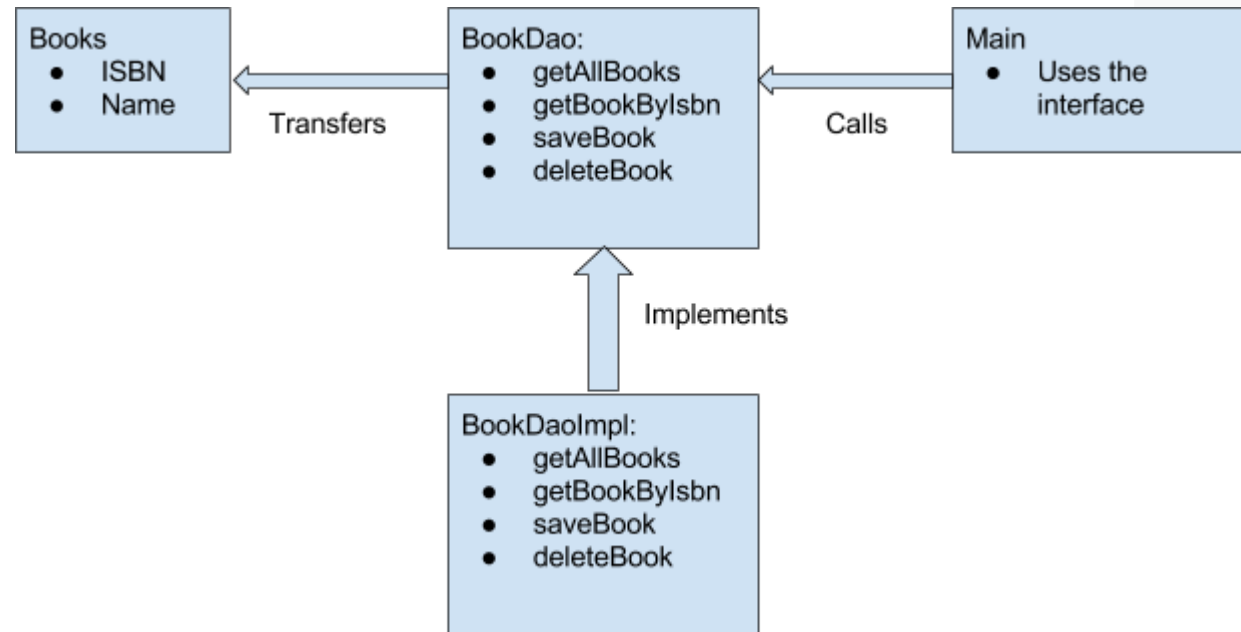
With DAO design pattern, we have following components on which our design depends:

- The model which is transferred from one layer to the other.
- The interfaces which provides a flexible design.
- The interface implementation which is a concrete implementation of the persistence logic.

# Implementing DAO pattern

With above mentioned components, let's try to implement the DAO pattern. We will use 3 components here:

- The Book model which is transferred from one layer to the other.
- The BookDao interface that provides a flexible design and API to implement.
- BookDaoImpl concrete class that is an implementation of the BookDao interface.



```
public class Books {

 private int isbn;
 private String bookName;

 public Books() {
 }

 public Books(int isbn, String bookName) {
 this.isbn = isbn;
 this.bookName = bookName;
 }

 // getter setter methods
}
```

# DAO Pattern Interface

```
import java.util.List;

public interface BookDao {

 List<Books> getAllBooks();
 Books getBookByIsbn(int isbn);
 void saveBook(Books book);
 void deleteBook(Books book);
}
```



# DAO Pattern Implementation

```
import java.util.ArrayList;
import java.util.List;

public class BookDaoImpl implements BookDao {

 //list is working as a database
 private List<Books> books;

 public BookDaoImpl() {
 books = new ArrayList<>();
 books.add(new Books(1, "Java"));
 books.add(new Books(2, "Python"));
 books.add(new Books(3, "Android"));
 }
}
```

```
@Override
public List<Books> getAllBooks() {
 return books;
}
```

```
@Override
public Books getBookByIsbn(int isbn) {
 return books.get(isbn);
}
```

```
@Override
public void saveBook(Books book) {
 books.add(book);
}
```

@Override

```
public void deleteBook(Books book) {
 books.remove(book);
}
}
```



# Using DAO Pattern

```
public class AccessBook {

 public static void main(String[] args) {

 BookDao bookDao = new BookDaoImpl();

 for (Books book : bookDao.getAllBooks()) {
 System.out.println("Book ISBN : " + book.getIsbn());
 }

 //update student
 Books book = bookDao.getAllBooks().get(1);
 book.setBookName("Algorithms");
 bookDao.saveBook(book);
 }
}
```



# Advantages of DAO pattern

- While changing a persistence mechanism, service layer doesn't even have to know where the data comes from. For example, if you're thinking of shifting from using MySQL to MongoDB, all changes are needed to be done in the DAO layer only.
- DAO pattern emphasis on the low coupling between different components of an application. So, the View layer have no dependency on DAO layer and only Service layer depends on it, even that with the interfaces and not from concrete implementation.
- As the persistence logic is completely separate, it is much easier to write Unit tests for individual components. For example, if you're using JUnit and Mockito for testing frameworks, it will be easy to mock the individual components of your application.
- As we work with interfaces in DAO pattern, it also emphasizes the style of "work with interfaces instead of implementation" which is an excellent OOPs style of programming.

# Injection Flaws

SQL Injection flaws in your applications. SQL Injection attacks are unfortunately very common, and this is due to two factors:

- the significant prevalence of SQL Injection vulnerabilities, and
  - the attractiveness of the target (i.e., the database typically contains all the interesting/critical data for your application).
- 
- It's somewhat very bad that there are so many successful SQL Injection attacks occurring, because it is EXTREMELY simple to avoid SQL Injection vulnerabilities in your code.

- SQL Injection flaws are introduced when software developers create dynamic database queries that include user supplied input.
- To avoid SQL injection flaws is simple. Developers need to either:
  - a) stop writing dynamic queries; and/or
  - b) prevent user supplied input which contains malicious SQL from affecting the logic of the executed query.

## Primary Defenses:

- **Option 1: Use of Prepared Statements (with Parameterized Queries)**
- **Option 2: Use of Stored Procedures**
- **Option 3: White List Input Validation**
- **Option 4: Escaping All User Supplied Input**

## Additional Defenses:

- **Also: Enforcing Least Privilege**
- **Also: Performing White List Input Validation as a Secondary Defense**

## Unsafe Example

- SQL injection flaws typically look like this:
- The following (Java) example is UNSAFE, and would allow an attacker to inject code into the query that would be executed by the database.
- The unvalidated "customerName" parameter that is simply appended to the query allows an attacker to inject any SQL code they want.
- Unfortunately, this method for accessing databases is all too common.



```
String query = "SELECT account_balance FROM user_data WHERE user_name = "
+ request.getParameter("customerName");
```

```
try {
```

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

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

```
}
```





## Defense Option 1: Prepared Statements (with Parameterized Queries)

- The use of prepared statements with variable binding (aka parameterized queries) is how all developers should first be taught how to write database queries.
- They are simple to write, and easier to understand than dynamic queries. Parameterized queries force the developer to first define all the SQL code, and then pass in each parameter to the query later.
- This coding style allows the database to distinguish between code and data, regardless of what user input is supplied.
- Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker.
- In the safe example below, if an attacker were to enter the userID of tom' or '1'='1, the parameterized query would not be vulnerable and would instead look for a username which literally matched the entire string tom' or '1'='1.

- Language specific recommendations:
- Java EE – use `PreparedStatement()` with bind variables
- .NET – use parameterized queries like `SqlCommand()` or `OleDbCommand()` with bind variables
- PHP – use PDO with strongly typed parameterized queries (using `bindParam()`)
- Hibernate - use `createQuery()` with bind variables (called named parameters in Hibernate)
- SQLite - use `sqlite3_prepare()` to create a statement object

The following code example uses a PreparedStatement, Java's implementation of a parameterized query, to execute the same database query.

```
String custname = request.getParameter("customerName"); // This should REALLY
be validated too
```

```
// perform input validation to detect attacks
```

```
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
```

```
PreparedStatement pstmt = connection.prepareStatement(query);
```

```
pstmt.setString(1, custname);
```

```
ResultSet results = pstmt.executeQuery();
```

## Defense Option 2: Stored Procedures

- Stored procedures are not always safe from SQL injection. However, certain standard stored procedure programming constructs have the same effect as the use of parameterized queries when implemented safely\* which is the norm for most stored procedure languages.
- They require the developer to just build SQL statements with parameters which are automatically parameterized unless the developer does something largely out of the norm.
- The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, and then called from the application.
- Both of these techniques have the same effectiveness in preventing SQL injection so your organization should choose which approach makes the most sense for you.

- \*Note: 'Implemented safely' means the stored procedure does not include any unsafe dynamic SQL generation.
- Developers do not usually generate dynamic SQL inside stored procedures.
- However, it can be done, but should be avoided. If it can't be avoided, the stored procedure must use input validation or proper escaping as described in this article to make sure that all user supplied input to the stored procedure can't be used to inject SQL code into the dynamically generated query.
- Auditors should always look for uses of `sp_execute`, `execute` or `exec` within SQL Server stored procedures.
- Similar audit guidelines are necessary for similar functions for other vendors.

## Safe Java Stored Procedure Example

- The following code example uses a CallableStatement, Java's implementation of the stored procedure interface, to execute the same database query. The "sp\_getAccountBalance" stored procedure would have to be predefined in the database and implement the same functionality as the query defined above.

```
String custname = request.getParameter("customerName"); // This should REALLY be
validated
try {
 CallableStatement cs = connection.prepareCall("{call sp_getAccountBalance(?)}");
 cs.setString(1, custname);
 ResultSet results = cs.executeQuery();
 // ... result set handling
} catch (SQLException se) {
 // ... logging and error handling
}
```

Q & A

Contact: [amitmahadik@synergetics-india.com](mailto:amitmahadik@synergetics-india.com)

Thank You

