JDBC

Java **D**ata**b**ase **C**onnectivity

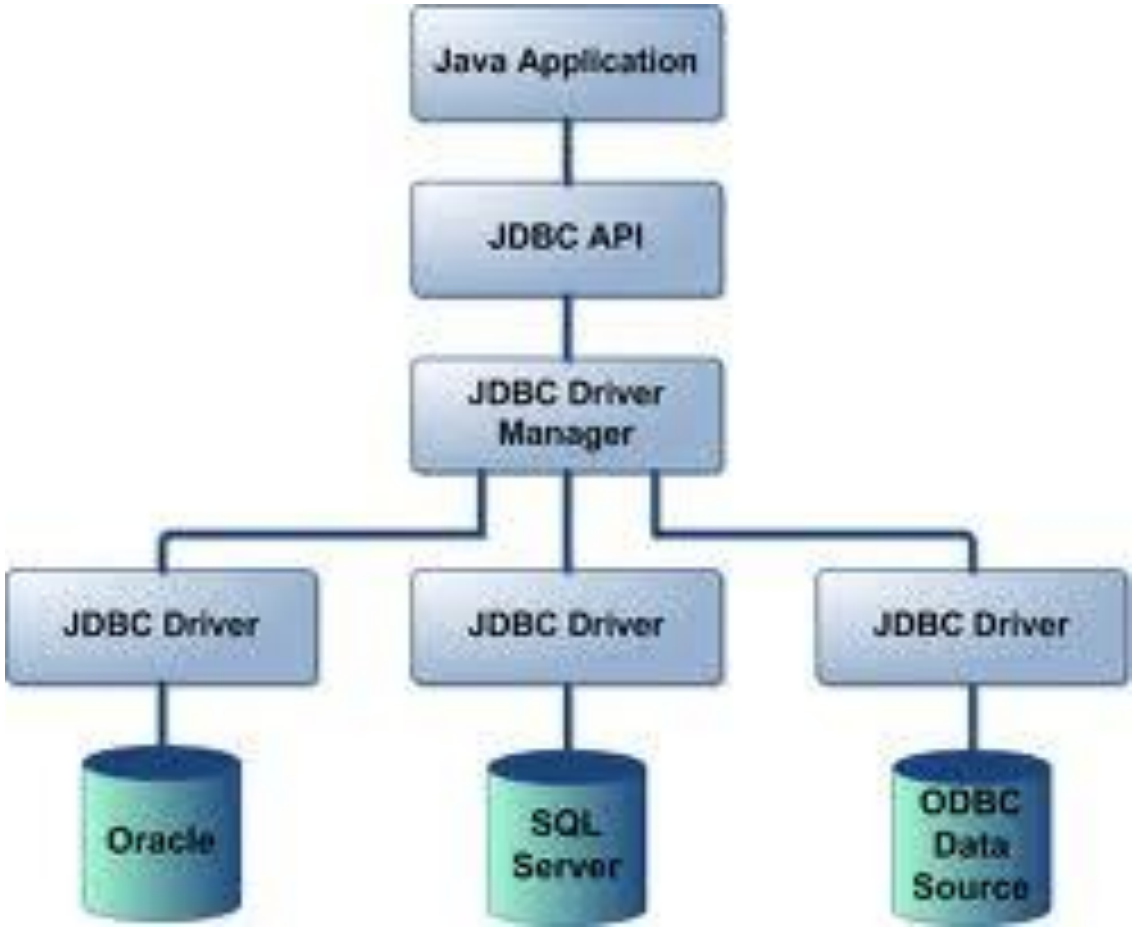
Contents

- JDBC Architecture
- Using JDBC
- Timeout
- ResultSet Object
- SQL Injection Attack

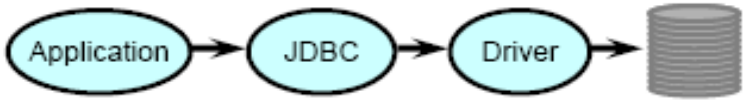
JDBC

- JDBC (Java Database Connectiveity) is an API (Application Programming Interface)
- JDBC is used for accessing databases from Java applications
- Information is transferred from relations to objects and vice-versa

JDBC Architecture



JDBC Architecture



- Java code calls JDBC library
- JDBC loads a driver
- The driver talks to a particular DBMS
- An application can work with several DBMS by using corresponding drivers

“Movies” Relation

moviename	producer	releasedate
Movie1	Producer1	1.1.2012
Movie2	Producer2	1.1.2012
Movie3	Producer3	3.4.2012

7 Steps for Using JDBC

1. Load the driver
2. Define the connection URL
3. Establish the connection
4. Create a Statement object
5. Execute a query using the Statement
6. Process the result
7. Close the connection

1. Loading the Driver

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

- Class.forName loads the given class dynamically
- When the driver is loaded, it automatically
  - creates an instance of itself
  - registers this instance within DriverManager
- MySql JDBC driver can be downloaded from

  http://www.mysql.com/products/connector/
- We will be using:

  JDBC Driver for MySQL (Connector/J)

JDBCSample.java

```java
import java.sql.*;

public class JDBCSample {
    public static void main( String args[]) {
        String url = "jdbc:mysql://localhost:3306/";
        String dbName = "movies";
        String userName = "root";
        String password = "root";
        //  Change the connection string according to your db, ip, username and password

        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            // Load the Driver class.
            Class.forName("com.mysql.jdbc.Driver");
            // If you are using any other database then load the right driver here.

            //Create the connection using the static getConnection method
            con = DriverManager.getConnection (url+dbName, userName, password);

            //Create a Statement class to execute the SQL statement
            stmt = con.createStatement();

            //Execute the SQL statement and get the results in a Resultset
            rs = stmt.executeQuery("select moviename, releasedate from movies");

            // Iterate through the ResultSet, displaying two values
            // for each row using the getString method
            while (rs.next())
                System.out.println("Name= " + rs.getString("moviename") + " Date= " + rs.getString("releasedate"));
        }
        catch (SQLException e) {e.printStackTrace();}
        catch (Exception e) {e.printStackTrace();}
        finally { // Close the connection. Notice the order.
            if(rs!=null){
                try{rs.close();}   catch(Exception e){e.printStackTrace();} }
            if(stmt!=null){
                try{stmt.close();} catch(Exception e){e.printStackTrace();} }
            if(con!=null){
                try{con.close();}  catch(Exception e){e.printStackTrace();} }
        }
```

2. Define the connection URL

- Every database is identified by a URL
- The username, password and DB name are specified
- *localhost* – we assume the DB is located locally
- Remote connections are possible using the IP address of the DBMS server
- The default port for MySQL is 3306

JDBCSample.java

```java
import java.sql.*;

public class JDBCSample {
    public static void main( String args[]) {
        String url = "jdbc:mysql://localhost:3306/";
        String dbName = "movies";
        String userName = "root";
        String password = "root";
        //  Change the connection string according to your db, ip, username and password

        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            // Load the Driver class.
            Class.forName("com.mysql.jdbc.Driver");
            // If you are using any other database then load the right driver here.

            //Create the connection using the static getConnection method
            con = DriverManager.getConnection (url+dbName, userName, password);

            //Create a Statement class to execute the SQL statement
            stmt = con.createStatement();

            //Execute the SQL statement and get the results in a Resultset
            rs = stmt.executeQuery("select moviename, releasedate from movies");

            // Iterate through the ResultSet, displaying two values
            // for each row using the getString method
            while (rs.next())
                System.out.println("Name= " + rs.getString("moviename") + " Date= " + rs.getString("releasedate"));
        }
        catch (SQLException e) {e.printStackTrace();}
        catch (Exception e) {e.printStackTrace();}
        finally { // Close the connection. Notice the order.
            if(rs!=null){
                try{rs.close();}   catch(Exception e){e.printStackTrace();} }
            if(stmt!=null){
                try{stmt.close();} catch(Exception e){e.printStackTrace();} }
            if(con!=null){
                try{con.close();}  catch(Exception e){e.printStackTrace();} }
        }
```

3. Establish the connection

Connection con =
DriverManager.*getConnection(url+dbName, userName, password);*

- Given a URL, DriverManager looks for the driver that can talk to the corresponding database

JDBCSample.java

```java
import java.sql.*;

public class JDBCSample {
    public static void main( String args[]) {
        String url = "jdbc:mysql://localhost:3306/";
        String dbName = "movies";
        String userName = "root";
        String password = "root";
        //  Change the connection string according to your db, ip, username and password

        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            // Load the Driver class.
            Class.forName("com.mysql.jdbc.Driver");
            // If you are using any other database then load the right driver here.

            //Create the connection using the static getConnection method
            con = DriverManager.getConnection (url+dbName, userName, password);

            //Create a Statement class to execute the SQL statement
            stmt = con.createStatement();

            //Execute the SQL statement and get the results in a Resultset
            rs = stmt.executeQuery("select moviename, releasedate from movies");

            // Iterate through the ResultSet, displaying two values
            // for each row using the getString method
            while (rs.next())
                System.out.println("Name= " + rs.getString("moviename") + " Date= " + rs.getString("releasedate"));
        }
        catch (SQLException e) {e.printStackTrace();}
        catch (Exception e) {e.printStackTrace();}
        finally { // Close the connection. Notice the order.
            if(rs!=null){
                try{rs.close();}   catch(Exception e){e.printStackTrace();} }
            if(stmt!=null){
                try{stmt.close();} catch(Exception e){e.printStackTrace();} }
            if(con!=null){
                try{con.close();}  catch(Exception e){e.printStackTrace();} }
        }
```

4. Create a Statement object

- We use Statement objects in order to
  - Query the DB
  - Update the db(insert, update, create, drop, …)

- executeQuery returns a ResultSet object representing the query result (discussed later…)

JDBCSample.java

```java
import java.sql.*;

public class JDBCSample {
    public static void main( String args[]) {
        String url = "jdbc:mysql://localhost:3306/";
        String dbName = "movies";
        String userName = "root";
        String password = "root";
        //  Change the connection string according to your db, ip, username and password

        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            // Load the Driver class.
            Class.forName("com.mysql.jdbc.Driver");
            // If you are using any other database then load the right driver here.

            //Create the connection using the static getConnection method
            con = DriverManager.getConnection (url+dbName, userName, password);

            //Create a Statement class to execute the SQL statement
            stmt = con.createStatement();

            //Execute the SQL statement and get the results in a Resultset
            rs = stmt.executeQuery("select moviename, releasedate from movies");

            // Iterate through the ResultSet, displaying two values
            // for each row using the getString method
            while (rs.next())
                System.out.println("Name= " + rs.getString("moviename") + " Date= " + rs.getString("releasedate"));
        }
        catch (SQLException e) {e.printStackTrace();}
        catch (Exception e) {e.printStackTrace();}
        finally { // Close the connection. Notice the order.
            if(rs!=null){
                try{rs.close();}   catch(Exception e){e.printStackTrace();} }
            if(stmt!=null){
                try{stmt.close();} catch(Exception e){e.printStackTrace();} }
            if(con!=null){
                try{con.close();}  catch(Exception e){e.printStackTrace();} }
        }
```

5. Execute a query using the Statement

- executeQuery returns a ResultSet object representing the query result (discussed later…)

JDBCSample.java

```java
import java.sql.*;

public class JDBCSample {
    public static void main( String args[]) {
        String url = "jdbc:mysql://localhost:3306/";
        String dbName = "movies";
        String userName = "root";
        String password = "root";
        //  Change the connection string according to your db, ip, username and password

        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            // Load the Driver class.
            Class.forName("com.mysql.jdbc.Driver");
            // If you are using any other database then load the right driver here.

            //Create the connection using the static getConnection method
            con = DriverManager.getConnection (url+dbName, userName, password);

            //Create a Statement class to execute the SQL statement
            stmt = con.createStatement();

            //Execute the SQL statement and get the results in a Resultset
            rs = stmt.executeQuery("select moviename, releasedate from movies");

            // Iterate through the ResultSet, displaying two values
            // for each row using the getString method
            while (rs.next())
                System.out.println("Name= " + rs.getString("moviename") + " Date= " + rs.getString("releasedate"));
        }
        catch (SQLException e) {e.printStackTrace();}
        catch (Exception e) {e.printStackTrace();}
        finally { // Close the connection. Notice the order.
            if(rs!=null){
                try{rs.close();}   catch(Exception e){e.printStackTrace();} }
            if(stmt!=null){
                try{stmt.close();} catch(Exception e){e.printStackTrace();} }
            if(con!=null){
                try{con.close();}  catch(Exception e){e.printStackTrace();} }
        }
```

6. Process the result

- We will discuss ResultSet in a while…

JDBCSample.java

```
import java.sql.*;

public class JDBCSample {
    public static void main( String args[]) {
        String url = "jdbc:mysql://localhost:3306/";
        String dbName = "movies";
        String userName = "root";
        String password = "root";
        //  Change the connection string according to your db, ip, username and password

        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            // Load the Driver class.
            Class.forName("com.mysql.jdbc.Driver");
            // If you are using any other database then load the right driver here.

            //Create the connection using the static getConnection method
            con = DriverManager.getConnection (url+dbName, userName, password);

            //Create a Statement class to execute the SQL statement
            stmt = con.createStatement();

            //Execute the SQL statement and get the results in a Resultset
            rs = stmt.executeQuery("select moviename, releasedate from movies");

            // Iterate through the ResultSet, displaying two values
            // for each row using the getString method
            while (rs.next())
                System.out.println("Name= " + rs.getString("moviename") + " Date= " + rs.getString("releasedate"));
        }
        catch (SQLException e) {e.printStackTrace();}
        catch (Exception e) {e.printStackTrace();}
        finally { // Close the connection. Notice the order.
            if(rs!=null){
                try{rs.close();}   catch(Exception e){e.printStackTrace();} }
            if(stmt!=null){
                try{stmt.close();} catch(Exception e){e.printStackTrace();} }
            if(con!=null){
                try{con.close();}  catch(Exception e){e.printStackTrace();} }
        }
```

7. Close the connection

- Close Connections, Statements, and Result Sets
  - *con.close();*
  - *stmt.close();*
  - *rs.close()*

JDBCSample.java

```java
import java.sql.*;

public class JDBCSample {
    public static void main( String args[]) {
        String url = "jdbc:mysql://localhost:3306/";
        String dbName = "movies";
        String userName = "root";
        String password = "root";
        //  Change the connection string according to your db, ip, username and password

        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            // Load the Driver class.
            Class.forName("com.mysql.jdbc.Driver");
            // If you are using any other database then load the right driver here.

            //Create the connection using the static getConnection method
            con = DriverManager.getConnection (url+dbName, userName, password);

            //Create a Statement class to execute the SQL statement
            stmt = con.createStatement();

            //Execute the SQL statement and get the results in a Resultset
            rs = stmt.executeQuery("select moviename, releasedate from movies");

            // Iterate through the ResultSet, displaying two values
            // for each row using the getString method
            while (rs.next())
                System.out.println("Name= " + rs.getString("moviename") + " Date= " + rs.getString("releasedate"));
        }
        catch (SQLException e) {e.printStackTrace();}
        catch (Exception e) {e.printStackTrace();}
        finally { // Close the connection. Notice the order.
            if(rs!=null){
                try{rs.close();}   catch(Exception e){e.printStackTrace();} }
            if(stmt!=null){
                try{stmt.close();} catch(Exception e){e.printStackTrace();} }
            if(con!=null){
                try{con.close();}  catch(Exception e){e.printStackTrace();} }
        }
```

ResultSet

- *ResultSet* objects provide **access** to the tables generated as results of executing Statement queries.
- Only one *ResultSet* per Statement can be open at a given time
- The table rows are retrieved in sequence:
  - A *ResultSet* maintains a cursor pointing to its current row.
  - *next()* moves the cursor to the next row

ResultSet Methods

- *boolean next()*
  - Activates the next row
  - First call to next() activates the first row
  - Returns false if there are no more rows
  - Not all of the next calls actually involve the DB
- *void close()*
  - Disposes of the ResultSet
  - Allows to re-use the Statement that created it
  - Automatically called by most Statement methods
- *Type getType(int columnIndex)*
  - Returns the given field as the given type
  - Indices start at 1 and not 0
- *Type getType(String columnName)*
  - Same, but uses name of field
- *int findColumn(String columnName)*
  - Looks up column index given column name

Timeout

- Use *setQueryTimeOut(int seconds)* of *Statement* to set a timeout for the driver to wait for a query to be completed.
- If the operation is not completed in the given time, an *SQLException* is thrown

Database Time

- Times in SQL are notoriously non-standard
- Java defines three classes to help
- java.sql.Date
  - *year, month, day*
- java.sql.Time
  - *hours, minutes, seconds*
- java.sql.Timestamp
  - *year, month, day, hours, minutes, seconds, nanoseconds*
  - *Usually use this one*

ResultSet

- ResultSet is a Java interface defined in the package *java.sql*
- It is a table of data representing the results of a query on a DB
- A ResultSet object maintains a cursor pointing to its current row of data.
- A default ResultSet object is not updatable and has a cursor that moves forward only.

ResultSet – next()

- Initially the cursor is positioned before the first row.
- The *next* method moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set.

```
while (rs.next())
        System.out.println(".....");
```

- When rs.next() is false the while loop terminates

Moving Between Rows in the ResultSet

- The current row in the result set can be identified using *getRow()*
- Move to the next row in the result set using *next()*

```
System.out.println("The Current Row is: "+rs.getRow());
rs.next();
System.out.println("The Current Row is: "+rs.getRow());
rs.next();
System.out.println("The Current Row is: "+rs.getRow());
rs.next();
```

Output:

The Current Row is: 0
The Current Row is: 1
The Current Row is: 2

ResultSet – Extracting Data From a Row

- **getString**(int columnIndex)
- **getString**(String columnName)
- **findColumn**(String columnName)

ResultSet

- The ResultSet interface provides *getter* methods (getString, getBoolean, getLong, etc) for retrieving column values from the current row.
- Values can be retrieved using either the index number of the column or the name of the column.
- Columns are numbered from 1.
- For the getter methods, a JDBC driver attempts to convert the underlying data to the Java type specified in the getter method and returns a suitable Java value

Mapping Java Types to SQL Types

SQL Type	Java Type
CHAR, VARCHAR, LONGVARCHAR	String
NUMERIC, DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT, DOUBLE	double
BINARY, VARBINARY, BYTEA	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

ResultSet – getString()

```
rs = stmt.executeQuery("select * from movies");

while(rs.next()){
    System.out.println("The Current Row is: "+rs.getRow());
    System.out.println("Value Stored in Column 1 is:"+rs.getString(1));
    System.out.println("Value Stored in Column 2 is:"+rs.getString(2));
    System.out.println("Value Stored in Column 3 is:"+rs.getString(3));
  }
```

OUTPUT

```
The Current Row is: 1
Value Stored in Column 1 is:Movie1
Value Stored in Column 2 is:Producer1
Value Stored in Column 3 is:1.1.2012
The Current Row is: 2
Value Stored in Column 1 is:Movie2
Value Stored in Column 2 is:Producer2
Value Stored in Column 3 is:1.1.2012
The Current Row is: 3
Value Stored in Column 1 is:Movie3
Value Stored in Column 2 is:Producer3
Value Stored in Column 3 is:3.4.2012
```

moviename	producer	releasedate
Movie1	Producer1	1.1.2012
Movie2	Producer2	1.1.2012
Movie3	Producer3	3.4.2012

ResultSet – getString() Using a Column Name

```
rs = stmt.executeQuery("select * from movies");

    // Iterate through the ResultSet, displaying two values
    // for each row using the getString method
    while(rs.next()){
      System.out.println("The Current Row is: "+rs.getRow());
      System.out.println("Value Stored in Column moviename is:"
                                                  +rs.getString("moviename"));
    }
```

Output:

```
The Current Row is: 1
Value Stored in Column moviename is:Movie1
The Current Row is: 2
Value Stored in Column moviename is:Movie2
The Current Row is: 3
Value Stored in Column moviename is:Movie3
```

ResultSet - findColumn

findColumn

public int **findColumn**(String columnName) throws SQLException

Maps the given ResultSet column name to its ResultSet column index.

Parameters:

columnName - the name of the column

Returns:

the column index of the given column name

ResultSet - findColumn

```
int columnIndex = rs.findColumn("releasedate");
System.out.println("The column Index For
                    releasedate is:"+columnIndex);
```

Output:

The column Index For releasedate is:3

Review - Data Manipulation Language (DML)

• Data manipulation language comprises the SQL data change statements, which modify stored data but not the schema or database objects.

1.SELECT ... FROM ... WHERE ...

2.INSERT INTO ... VALUES ...

3.UPDATE ... SET ... WHERE ...

4.DELETE FROM ... WHERE ...

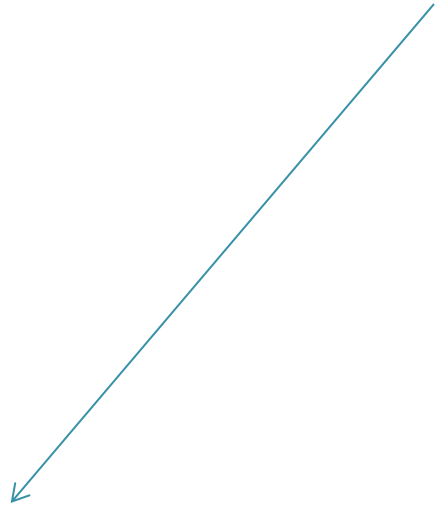
SQL DML - CRUD

- SQL DML commands are sometimes referred to as CRUD:
  - Create - INSERT - to store new data
  - Read - SELECT - to retrieve data
  - Update - UPDATE - to change or modify data.
  - Delete - DELETE - delete or remove data

- Another action sometimes considered is list e.g. “List all Customers”

CRUD – Create (Insert)

```
stmt.executeUpdate("INSERT INTO movies VALUES
                    ('Movie4','Producer4','5.6.2012')");
```



Output:

Movie1 Producer1 1.1.2012
Movie2 Producer2 1.1.2012
Movie3 Producer3 3.4.2012
Movie4 Producer4 5.6.2012

CRUD – Read (Select)

*rs = stmt.executeQuery("select * from movies");*

Output:
Movie1 Producer1 1.1.2012
Movie2 Producer2 1.1.2012
Movie3 Producer3 3.4.2012
Movie4 Producer4 5.6.2012

*select * from movies where producer = 'producer3';*

Output:
Movie3 Producer3 3.4.2012

CRUD – Update

```
stmt.executeUpdate("UPDATE movies SET releasedate
        = '6.6.2012' WHERE moviename = 'Movie4'");
```

Output:

Movie1 Producer1 1.1.2012
Movie2 Producer2 1.1.2012
Movie3 Producer3 3.4.2012
Movie4 Producer4 6.6.2012

CRUD – Delete

```
stmt.executeUpdate("DELETE FROM movies WHERE
                                    moviename='MOVIE1'");
```

Output:

Movie2 Producer2 1.1.2012
Movie3 Producer3 3.4.2012
Movie4 Producer4 6.6.2012

ResultSetMetaData

- ResultSetMetaData is an interface defined in java.sql
- It provides an object that can be used to get information about the types and properties of the columns in a ResultSet object.

"List All" Data Using ResultSetMetaData

```
rs = stmt.executeQuery("select * from movies");
ResultSetMetaData aRSMetaData= rs.getMetaData();
int aColumnCount = aRSMetaData.getColumnCount();

    while(rs.next()){
       System.out.print("\n");
       for(int i=1;i<=aColumnCount;i++){
              System.out.print(" "+rs.getString(i));
       }
    }
```

Output:

Movie1 Producer1 1.1.2012
Movie2 Producer2 1.1.2012
Movie3 Producer3 3.4.2012

SQL Injection

- SQL injection is where malicious SQL statements are inserted into an entry field for execution.
- SQL injection must exploit a security vulnerability in an application's software, for example, when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements and unexpectedly executed.

SQL Injection example

- "SELECT * FROM userinfo WHERE id =" + a_variable + ";“
- It is clear from this statement that the author intended a_variable to be a number correlating to the "id" field.
- However, if it is in fact a string then the end-user may manipulate the statement as they choose, thereby bypassing the need for escape characters. For example, setting a_variable to 1;DROP TABLE users will drop (delete) the "users" table from the database, since the SQL becomes:
- SELECT * FROM userinfo WHERE id=1;DROP TABLE users;

Causes/types of SQL Injection attacks

1. Incorrectly filtered escape characters
2. Incorrect type handling
3. Blind SQL injection

Incorrectly filtered escape characters

```
statement = " SELECT * FROM users WHERE name = ' " + userName + " ';"
```

- if the "userName" variable is crafted in a specific way by a malicious user, the SQL statement may do more than the code author intended. For example, setting the "userName" variable as:

```
' OR '1'='1
```

- If this code were to be used in an authentication procedure then this example could be used to force the selection of every data field (*) from all users rather than from one specific user name as the coder intended, because the evaluation of '1'='1' is always true

Incorrect type handling

- This could take place when a numeric field is to be used in a SQL statement, but the programmer makes no checks to validate that the user supplied input is numeric. For example:

```
statement := " SELECT * FROM userinfo WHERE id =" + a_variable + ";"
```

```
1;DROP TABLE users
```

- This will drop (delete) the "users" table from the database

Blind SQL Injection

- Blind SQL Injection is used when a web application is vulnerable to an SQL injection but the results of the injection are not visible to the attacker. The page with the vulnerability may not be one that displays data but will display differently depending on the results of a logical statement injected into the legitimate SQL statement called for that page. This type of attack has traditionally been considered time-intensive because a new statement needed to be crafted for each bit recovered, and depending on its structure, the attack may consist of many unsuccessful requests

Blind SQL Injection e.g. TimeBased

- This type of blind SQL injection relies on the database pausing for a specified amount of time, then returning the results, indicating successful SQL query executing. Using this method, an attacker enumerates each letter of the desired piece of data using the following logic:

- If the first letter of the first database's name is an 'A', wait for 10 seconds.

- If the first letter of the first database's name is an 'B', wait for 10 seconds. etc.

Second order SQL injection

- Second order SQL injection occurs when submitted values contain malicious commands that are stored rather than executed immediately.
- In some cases, the application may correctly encode an SQL statement and store it as valid SQL. Then, another part of that application without controls to protect against SQL injection might execute that stored SQL statement.
- This attack requires more knowledge of how submitted values are later used.

Protecting against an SQL Injection Attack

1. Use Object Relational Mapping Libraries

e.g. JPA (Java Persistence API)

2. Database Permissions

Limiting the permissions on the database login used by the web application to only what is needed may help reduce the effectiveness of any SQL injection attacks. E.g. a database logon could be restricted from selecting some of the system tables

3. Pattern check

Integer, float or boolean, string parameters can be checked if their value is valid representation for the given type. Strings that must follow some strict pattern (date,, alphanumeric only, etc.) can be checked if they match this pattern.