ASSIGNMENT

SQL INJECTION:

SQL injection, also known as SQLI, is a common attack vector that uses malicious SQL code for back end database manipulation to access information that was not intended to be displayed. This information may include any number of items, including sensitive company data, user lists .

EXAMPLE:

```
SELECT ItemName, ItemDescription
  FROM Items
 WHERE ItemNumber = 999; DROP TABLE USERS
```

STATEMENT:

Statement is used for general-purpose access to the database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters**.**

boolean execute(String SQL) :

Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use the truly dynamic SQL.

int executeUpdate(String SQL):

Returns the number 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.

Example: 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.

EXAMPLE:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class Example {
static final String DB_URL =
"jdbc:mysql://localhost/NEWFOLDER";
static final String USER = "guest";
static final String PASS = "guest123";
static final String QUERY = "SELECT id,
first,  last,          age FROM Employees";
static final String UPDATE_QUERY =
"UPDATE  Employees set age=30 WHERE id=103";

public static void main(String[] args) {
try(Connection conn =
DriverManager.getConnection(DB_URL, USER, PASS);
Statement stmt = conn.createStatement();
) {

Boolean ret = stmt.execute(UPDATE_QUERY);
System.out.println("Return value is : "
+        ret.toString() );
```

```
                int rows =
stmt.executeUpdate(UPDATE_QUERY);
                System.out.println("Rows impacted : "
+   rows );
        ResultSet rs = stmt.executeQuery(QUERY);
  while (rs.next()) {
     System.out.print("ID: " + rs.getInt("id"));
                  System.out.print(", Age: " + r
s.getInt("age"));
                    System.out.print(", First: " +
      rs.getString("first"));
                  System.out.println(", Last: " +
   rs.getString("last"));
       }
       rs.close();
     } catch (SQLException e) {
       e.printStackTrace();
     }
   }
 }
```

PREPARED STATEMENT:

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

- Prepared statement gives you the flexibility of supplying arguments dynamically.

EXAMPLE:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
public class Example {
```

```
                static final String DB_URL =
                 "jdbc:mydb://localhost/NEWFOLDER";
                static final String USER = "guest";
                static final String PASS = "guest123";
               static final String QUERY = "SELECT id, first, last, age
                 FROM Employees";
                 static final String UPDATE_QUERY = "UPDATE
                   Employees set age=? WHERE id=?";
                  public static void main(String[] args) {
          try(Connection conn =
               DriverManager.getConnection(DB_URL, USER, PASS);
              PreparedStatement stmt =
                conn.prepareStatement(UPDATE_QUERY);
               )
                  {
               stmt.setInt(1, 35);
               stmt.setInt(2, 102);
              int rows = stmt.executeUpdate();
               System.out.println("Rows impacted : " + rows );
             ResultSet rs = stmt.executeQuery(QUERY);
       while (rs.next()) {
      System.out.print("ID: " + rs.getInt("id"));
         System.out.print(", Age: " + rs.getInt("age"));
         System.out.print(", First: " + rs.getString("first"));
         System.out.println(", Last: " + rs.getString("last"));
       }
       rs.close();
     } catch (SQLException e) {
       e.printStackTrace();
     }
   }
}
```

CALLABLE STATEMENT:

Same as a Connection object creates the Statement and Prepared Statement objects, it also creates the Callable Statement object, which would be used to execute a call to a database stored procedure..

EXAMPLE:

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class Example {
    static final String DB_URL =
"jdbc:mysql://localhost/NEWFOLDER";
static final String USER = "guest";
static final String PASS = "guest123";
static final String QUERY = "{call getEmpName (?, ?)}";
public static void main(String[] args) {
    try(Connection conn =
DriverManager.getConnection(DB_URL, USER, PASS);
CallableStatement stmt = conn.prepareCall(QUERY);
) {
stmt.setInt(1, 102);
stmt.registerOutParameter(2, java.sql.Types.VARCHAR);
System.out.println("Executing stored procedure..." );
stmt.execute();
String empName = stmt.getString(2);
System.out.println("Emp Name with ID: 102 is " +
empName);
} catch (SQLException e) {
e.printStackTrace();
}
}
}
```

INNER CLASS:

Inner class means one class which is a member of another class. There are basically four types of inner classes in java.

- Nested Inner class
- Method Local inner classes
- Anonymous inner classes
- Static nested classes

Nested Inner class:

Nested Inner class can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier.
Like class, interface can also be nested and can have access specifiers.

EXAMPLE:

```
class Example{
class Inner {
 public void show() {
 System.out.println("In a nested class method");
      }
   }
  }
class Main {
public static void main(String[] args) {
Example.Inner in = new Example().new Inner();
    in.show();
}
```

```
}
```

Method Local Inner class:

Inner class can be declared within a method of an outer class. Inner is an inner class in outerMethod().

EXAMPLE:

```
class Outer {
void outerMethod() {
System.out.println("inside outerMethod");
class Inner {
void innerMethod() {
System.out.println("inside innerMethod");
}
}
Inner y = new Inner();
y.innerMethod();
}
}
class MethodDemo {
public static void main(String[] args) {
Outer x = new Outer();
x.outerMethod();
}
}
```

Anonymous inner classes:

Anonymous inner classes are declared without any name at all.

EXAMPLE:

```
class Example {
```

```
static Hello h = new Hello() {
  public void show() {
        System.out.println("i am in
anonymous class");
  }
};

public static void main(String[] args) {
  h.show();
}
        }
interface Hello {
  void show();
  }
```

Static nested classes:

Static nested classes are not technically an inner class. They are like a static member of outer class.

EXAMPLE:

```
        class Outer {
  private static void outerMethod() {
System.out.println("inside outerMethod");
    }
  static class Inner {
 public static void main(String[] args) {
   System.out.println("inside inner class Method");
   outerMethod();
      }
    }
```

}

RELATIONSHIPS IN JAVA:

There are three types of relationships in java:

- ❖ Is-A relationship
- ❖ Has-A relationship
- ❖ Uses-A relationship

Is-A relationship:

an Is-A relationship depends on inheritance. Further inheritance is of two types, class inheritance and interface inheritance. It is used for code reusability in Java.

Example: apple is a fruit.

Has-A relationship:

a Has-A relationship is also known as composition. It is also used for code reusability in Java. In Java, a Has-A relationship simply means that an instance of one class has a reference to an instance of another class or an other instance of the same class.

Example: a dog has a tail.

Uses-A relationship:

When we create an object of a class inside a method of another class, this relationship is called dependence relationship in Java, or simply Uses-A relationship. In other words, when a method of a class uses an object of another class, it is called dependency in java.

Example: teacher uses a blackboard.