- JDBC
- Annotation
- Reflection

DAO class

- In enterprise applications, there are multiple tables and frequent data transfer from database is needed.
- Instead of writing a JDBC code in multiple Java files of the application (as and when needed), it is good practice to keep all the JDBC code in a centralized place -- in a single application layer.
- DAO (Data Access Object) class is standard way to implement all CRUD operations specific to a table. It is advised to create different DAO for different table.
- DAO classes makes application more readable/maintainable.
- Example 1:

```
class StudentDao implements AutoClosable {
private Connection con;
public StudentDao() throws Exception {
  con = DriverManager.getConnection(DbUtil.DB_URL, DbUtil.DB_USER,
DbUtil.DB_PASSWORD);
}
public void close() {
  try{
    if(con != null)
    con.close();
  } catch(Exception ex) {
  }
}
public int update(Student s) throws Exception {
  int count = 0;
  String sql = "UPDATE students SET name=?, marks=? WHERE roll=?"
  try(PreparedStatement stmt = con.prepareStatement(sql)) {
  // optionally you may create PreparedStatement in constructor (as implemented)
  stmt.setString(1, s.getName());
  stmt.setDouble(2, s.getMarks());
  stmt.setInt(3, s.getRoll());
  count = stmt.executeUpdate();
}
return count;
}
}

// in main()
try(StudentDao dao = new StudentDao()) {
  System.out.print("Enter roll to be updated: ");
  int roll = sc.nextInt();
  System.out.print("Enter new name: ");
  String name = sc.next();
  System.out.print("Enter new marks: ");
  double marks = sc.next();
  Student s = new Student(roll, name, marks);
```

```
  int cnt = dao.update(s);
  System.out.println("Rows updated: " + cnt);
} // dao.close()
catch(Exception ex) {
  ex.printStackTrace();
}
```

- Example 2:

```
// POJO (Entity)
class Emp {
private int empno;
private String ename;
private Date hire;
// ...
}

class DbUtil {
public static final String DB_DRIVER = "com.mysql.cj.jdbc.Driver";
public static final String DB_URL = "jdbc:mysql://localhost:3306/test";
public static final String DB_USER = "root";
public static final String DB_PASSSWD = "root";
static {
  try {
    Class.forName(DB_DRIVER);
} catch (ClassNotFoundException e) {
    e.printStackTrace();
    System.exit(0);
}
}
public static Connection getConnection() throws Exception {
  return DriverManager.getConnection(DB_URL, DB_USER, DB_PASSSWD);
}
}

class EmpDao implements AutoClosable {
private Connection con;
public EmpDao() throws Exception {
  con = DbUtil.getConnection();
}
public void close() {
  try {
    if(con != null)
    con.close();
} catch(Exception ex) {
  ex.printStackTrace();
}
}

public int update(Emp e) throws Exception {
  String sql = "UPDATE emp SET ename=?, hire=? WHERE id=?";
  try(PreparedStatement stmt = con.prepareStatement(sql)) {
```

```
    stmt.setString(1, e.getEname());
    java.util.Date uDate = e.getHire();
    java.sql.Date sDate = new java.sql.Date(uDate.getTime());
    stmt.setDate(2, sDate);
    stmt.setInt(3, e.getEmpno());
    int cnt = stmt.executeUpdate();
    return cnt;
} // stmt.close();
}
// ...
}

// in main()
try(EmpDao dao = new EmpDao()) {
  Emp e = new Emp();
// input emp data from end user (Scanner)
/*
String dateStr = sc.next(); // dd-MM-yyyy
SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
java.util.Date uDate = sdf.parse(dateStr);
e.setHire(uDate);
*/
  int cnt = dao.update(e);
  System.out.println("Emps updated: " + cnt);
} // dao.close();
catch(Exception ex) {
ex.printStackTrace();
}
```

- Example 3 (using the POJO and DBUtil same as Example 2)

```
class EmpDao implements AutoClosable {
private Connection con;
private PreparedStatement stmtFindById;
// ...
public EmpDao() throws Exception {
  con = DbUtil.getConnection();
  String sql = "SELECT * FROM emp WHERE empno=?";
  stmtFindById = con.prepareStatement(sql);
// ...
}
public void close() {
  try {
  // ...
  if(stmtFindById != null)
  stmtFindById.close();
  if(con != null)
  con.close();
} catch(Exception ex) {
  ex.printStackTrace();
}
}
```

```
public Emp findById(int empno) throws Exception {
  stmtFindById.setInt(1, empno);
  try(ResultSet rs = stmtFindById.executeQuery()) {
  if(rs.next()) {
  int empno = rs.getInt("empno");
  String ename = rs.getString("ename");
  java.sql.Date sDate = rs.getDate("hire");
  // ...
  java.util.Date uDate = new java.util.Date( sDate.getTime() );
  Emp e = new Emp(empno, ename, uDate);
  return e;
}
} // rs.close();
return null;
}
}
```
```java
// in main()
try(EmpDao dao = new EmpDao()) {
  System.out.print("Enter empno to find: ");
  id = sc.nextInt();
  e = dao.findById(id);
  System.out.println("Found: " + e);
  System.out.print("Enter empno to find: ");
  id = sc.nextInt();
  e = dao.findById(id);
  System.out.println("Found: " + e);
  System.out.print("Enter empno to find: ");
  id = sc.nextInt();
  e = dao.findById(id);
  System.out.println("Found: " + e);
}
catch(Exception ex) {
  ex.printStackTrace();
}
```

Call Stored Procedure using JDBC (without OUT parameters)

- Stored Procedure - Increment votes of candidate with given id

```
DELIMITER //
CREATE PROCEDURE sp_incrementvotes(IN p_id INT)
BEGIN
UPDATE candidates SET votes=votes+1 WHERE id=p_id;
END;
//
DELIMITER ;

CALL sp_incrementvotes(10);
```

- JDBC use CallableStatement interface to invoke the stored procedures.
- CallableStatement interface is extended from PreparedStatement interface.
- Steps to call Stored procedure are same as PreparedStatement.
    - Create connection.
    - Create CallableStatement using con.prepareCall("CALL ...").
    - Set IN parameters using stmt.setXYZ(...);
    - Execute the procedure using stmt.executeQuery() or stmt.executeUpdate().
    - Close statement & connection.
- To invoke stored procedure, in general stmt.execute() is called. This method returns true, if it is returning ResultSet (i.e.multi-row result). Otherwise it returns false, if it is returning update/affected rows count.

```
boolean isResultSet = stmt.execute();
if(isResultSet) {
ResultSet rs = stmt.getResultSet();
// process the ResultSet
}
else {
int count = stmt.getUpdateCount();
// process the count
}
```

Call Stored Procedure using JDBC (with OUT parameters)

- Stored Procedure - Get quote and author of given quote id -- using OUT parameters.

```
DELIMITER //
CREATE PROCEDURE sp_getpartyvotes(IN p_party CHAR(40), OUT p_votes INT)
BEGIN
SELECT SUM(votes) INTO p_votes FROM candidates WHERE party=p_party;
END;
//
DELIMITER ;

CALL sp_getpartyvotes('BJP', @votes);
SELECT @votes;
```

- Steps to call Stored procedure with out params.
    - Create connection.
    - Create CallableStatement using con.prepareCall("CALL ...").
    - Set IN parameters using stmt.setXYZ(...) and register out parameters using stmt.registerOutParam(...).
    - Execute the procedure using stmt.execute().
    - Get values of out params using stmt.getXYZ(paramNumber).
    - Close statement & connection.

Transaction Management

- RDBMS Transactions
- Transaction is set of DML operations to be executed as a single unit. Either all queries in tx should be successful or all should be discarded.
- The transactions must be atomic. They should never be partial.

```
CREATE TABLE accounts(id INT, type CHAR(30), balance DOUBLE);
INSERT INTO accounts VALUES (1, 'Saving', 30000.00);
INSERT INTO accounts VALUES (2, 'Saving', 2000.00);
INSERT INTO accounts VALUES (3, 'Saving', 10000.00);
SELECT * FROM accounts;
START TRANSACTION;
--SET @@autocommit=0;
UPDATE accounts SET balance=balance-3000 WHERE id=1;
UPDATE accounts SET balance=balance+3000 WHERE id=2;
SELECT * FROM accounts;
COMMIT;
-- OR
ROLLBACK;
```

- JDBC transactions (Logical code)

```
try(Connection con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD)) {
con.setAutoCommit(false); // start transaction

String sql = "UPDATE accounts SET balance=balance+? WHERE id=?";

try(PreparedStatement stmt = con.prepareStatement(sql)) {
  stmt.setDouble(1, -3000.0); // amount=3000.0
  stmt.setInt(2, 1); // accid = 1
  cnt1 = stmt.executeUpdate();
  stmt.setDouble(1, +3000.0); // amount=3000.0
  stmt.setInt(2, 2); // accid = 2
  cnt2 = stmt.executeUpdate();
  if(cnt1 == 0 || cnt2 == 0)
    throw new RuntimeException("Account Not Found");
}
con.commit(); // commit transaction
}
catch(Exception e) {
  e.printStackTrace();
  con.rollback(); // rollback transaction
}
```

Reflection

- It is a technique to read the metadata and work with that data.

- .class = Byte-code + Meta-data + Constant pool + ...
- When class is loaded into JVM all the metadata is stored in the object of java.lang.Class (heap area).
- This metadata includes class name, super class, super interfaces, fields (field name, field type, access modifier, flags), methods (method name, method return type, access modifier, flags, method arguments, ...), constructors (access modifier, flags, ctor arguments, ...), annotations (on class, fields, methods, ...).

Reflection applications

- Inspect the metadata (like javap)
- Build IDE/tools (Intellisense)
- Dynamically creating objects and invoking methods
- Access the private members of the class

Get the java.lang.Class object

- way 1: When you have class-name as a String (taken from user or in properties file)

```
Class<?> c = Class.forName(className);
```

- way 2: When the class is in project/classpath.

```
Class<?> c = ClassName.class;
```

- way 3: When you have object of the class.

```
Class<?> c = obj.getClass();
```

Access metadata in java.lang.Class

- Name of the class

```
String name = c.getName();
```

- Super class of the class

```
Class<?> supcls = c.getSuperclass();
```

- Super interfaces of the class

```
Class<?> supintf[] = c.getInterfaces();
```

- Fields of the class

```
Field[] fields = c.getFields(); // all fields accessible (of class & its
super class)
```

```
Field[] fields = c.getDeclaredFields(); // all fields in the class
```

- Methods of the class

```
Method[] methods = c.getMethods(); // all methods accessible (of class & its
super class)
```

```
Method[] methods = c.getDeclaredMethods(); // all methods in the class
```

- Constructors of the class

```
Constructor[] ctors = c.getConstructors(); // all ctors accessible (of class
& its super class)
```

```
Constructor[] ctors = c.getDeclaredConstructor(); // all ctors in the class
```

Annotations

- Added in Java 5.0.
- Annotation is a way to associate metadata with the class and/or its members.
- Annotation applications
  - Information to the compiler
  - Compile-time/Deploy-time processing
  - Runtime processing
- Annotation Types
  - Marker Annotation: Annotation is not having any attributes.
    - @Override, @Deprecated, @FunctionalInterface ...
  - Single value Annotation: Annotation is having single attribute -- usually it is "value".
    - @SuppressWarnings("deprecation"), ...

    - Multi value Annotation: Annotation is having multiple attribute
        - @RequestMapping(method = "GET", value = "/books"), ...

Pre-defined Annotations

- @Override
    - Ask compiler to check if corresponding method (with same signature) is present in super class.
    - If not present, raise compiler error.
- @FunctionalInterface
    - Ask compiler to check if interface contains single abstract method.
    - If zero or multiple abstract methods, raise compiler error.
- @Deprecated
    - Inform compiler to give a warning when the deprecated type/member is used.
- @SuppressWarnings
    - Inform compiler not to give certain warnings: e.g. deprecation, rawtypes, unchecked, serial, unused
    - @SuppressWarnings("deprecation")
    - @SuppressWarnings({"rawtypes", "unchecked"})
    - @SuppressWarnings("serial")
    - @SuppressWarnings("unused")

Meta-Annotations

- Annotations that apply to other annotations are called meta-annotations.
- Meta-annotation types defined in java.lang.annotation package.

@Retention

- RetentionPolicy.SOURCE
    - Annotation is available only in source code and discarded by the compiler (like comments).
    - Not added into .class file.
    - Used to give information to the compiler.
    - e.g. @Override, ...
- RetentionPolicy.CLASS
    - Annotation is compiled and added into .class file.
    - Discared while class loading and not loaded into JVM memory.
    - Used for utilities that process .class files.
    - e.g. Obfuscation utilities can be informed not to change the name of certain class/member using @SerializedName, ...
- RetentionPolicy.RUNTIME
    - Annotation is compiled and added into .class file. Also loaded into JVM at runtime and available for reflective access.
    - Used by many Java frameworks.
    - e.g. @RequestMapping, @Id, @Table, @Controller, ...

@Target

- Where this annotation can be used.

- ANNOTATION_TYPE, CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE, TYPE_PARAMETER, TYPE_USE
- If annotation is used on the other places than mentioned in @Target, then compiler raise error.

@Documented

- This annotation should be documented by javadoc or similar utilities.

@Repeatable

- The annotation can be repeated multiple times on the same class/target.

@Inherited

- The annotation gets inherited to the sub-class and accessible using c.getAnnotation() method.

Custom Annotation

- Annotation to associate developer information with the class and its members.

```
@Inherited
@Retention(RetentionPolicy.RUNTIME) // the def attribute is considered as
"value" = @Retention(value = RetentionPolicy.RUNTIME )
@Taget({TYPE, CONSTRUCTOR, FIELD, METHOD}) // { } represents array
@interface Developer {
    String firstName();
    String lastName();
    String company() default "Sunbeam";
    String value() default "Software Engg";
}

@Repeatable
@Retention(RetentionPolicy.RUNTIME)
@Taget({TYPE})
@interface CodeType {
    String[] value();
}
```

```
//@Developer(firstName="Nilesh", lastName="Ghule", value="Technical
Director") // compiler error -- @Developer is not @Repeatable
@CodeType({"businessLogic", "algorithm"})
@Developer(firstName="Nilesh", lastName="Ghule", value="Technical Director")
class MyClass {
    // ...
    @Developer(firstName="Shubham", lastName="Borle", company="Sunbeam Karad
")
    private int myField;
    @Developer(firstName="Rahul", lastName="Sansuddi")
    public MyClass() {
```

```
    }
    @Developer(firstName="Shubham", lastName="Borle", company="Sunbeam Karad
")
    public void myMethod() {
        @Developer(firstName="James", lastName="Bond") // compiler error
        int localVar = 1;
    }
}
```

```
// @Developer is inherited
@CodeType("frontEnd")
@CodeType("businessLogic") // allowed because @CodeType is @Repeatable
class YourClass extends MyClass {
    // ...
}
```

Annotation processing (using Reflection)

```
Annotation[] anns = MyClass.class.getDeclaredAnnotations();
for (Annotation ann : anns) {
    System.out.println(ann.toString());
    if(ann instanceof Developer) {
        Developer devAnn = (Developer) ann;
        System.out.println(" - Name: " + devAnn.firstName() + " " + devAnn.
lastName());
        System.out.println(" - Company: " + devAnn.company());
        System.out.println(" - Role: " + devAnn.value());
    }
}
System.out.println();

Field field = MyClass.class.getDeclaredField("myField");
anns = field.getAnnotations() ;
for (Annotation ann : anns)
    System.out.println(ann.toString());
System.out.println();

//anns = YourClass.class.getDeclaredAnnotations();
anns = YourClass.class.getAnnotations();
for (Annotation ann : anns)
    System.out.println(ann.toString());
System.out.println();
```