 ISSN: 2321-8169
223 – 232

Custom Annotation for Execution of Data Manipulation Commands in a Generic Manner – A Case Study

Dr. Poornima G. Naik
Professor
Department of Computer Studies
Chatrapati Shahu Institute of Business
Education and Research
Kolhapur, India
e-mail: pgnaik@siberindia.edu.in

Mr. Girish R. Naik
Associate Professor
Production Department
KIT's College of Engineering
Kolhapur, India
e-mail: girishnaik2025@gmail.com

Abstract: Java annotations are the tags employed for providing meta data for Java code. They can be attached to a class, method, or a field to provide some additional information to the compiler and JVM. This paper introduces the concept of Annotations in Java with an emphasis on various in-built annotations in Java and the annotations that are used by other annotations. The reader is introduced to J2EE standard annotations and those employed by Hibernate as a replacement for XML-based mapping document. Steps in designing and using custom annotations are highlighted. A custom annotation design is illustrated with the help of an example for execution of DML commands in a generic way in a database management system independent manner.

Keywords: *Data Manipulation Language, Eclipse, Hibernate, J2EE, JNDI, Java Virtual Machine, Extensible Markup Language*

*****

INTRODUCTION

Java Annotation is an extremely useful concept in Java for emitting metadata for Java Compiler, JVM, etc. Custom annotations enable code reusability. Most of the Java technology such as JNDI, Hibernate, Spring, etc. make abundant use of Java Annotations. Majority of the annotations do not directly affect the execution of code. Annotations have no direct effect on the operation of the code which they annotate. The purpose of the Java annotations is three fold. Annotations in Java are used for providing:

- Instructions to the Compiler - Annotations provide information to the compiler which the compiler uses to detect errors, suppress warnings, etc.
- Instructions during Build time - Software tools can process annotation for generating code.
- Instructions during Runtime - Annotations can be detected through reflection at runtime.

A. Built-in Annotations used in Java Code

Compiler Instructions

There are three in-built annotations which provide instructions to the Java compiler [1-3].

@Deprecated

This annotation is employed for designating the class method which should no longer be used in code and alternatives exist for it. @Deprecated annotation marks the method as deprecated so compiler displays a warning message. It is a warning for the user indicating that it may be removed in the future versions. So, it is safe not to use such methods. When a method is qualified with this annotation, the IDEs such as Eclipse visually signify it by striking the method as shown in Figure 1(a) and Figure 1(b).

```
package com.siber;

public class TestDeprecated {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        DeprecatedMethods dm=new DeprecatedMethods();
        dm.method1();
        dm.method2();
    }

}
```

```
package com.siber;

public class DeprecatedMethods {
@Deprecated
public void method1()
{
    System.out.println("Inside method1");
}

@Deprecated
public void method2()
{
    System.out.println("Inside method2");
}
}
```

Figure 1(a)-1(b). Visual Notification for Deprecated Methods in Eclipse

@Override

This annotation is used to qualify the method that overrides the superclass method. It assures that the subclass method overrides the corresponding superclass method. The compiler will flag an error is the prototype of the overridden method does not match its user class counterpart. This annotation comes in handy to detect the errors in case superclass method is accidentally deleted or changed. Without @Override annotation, the method will be treated as the new functionality added to the subclass. The usage of @Override annotation is depicted in Figure 2(a) and Figure 2(b).

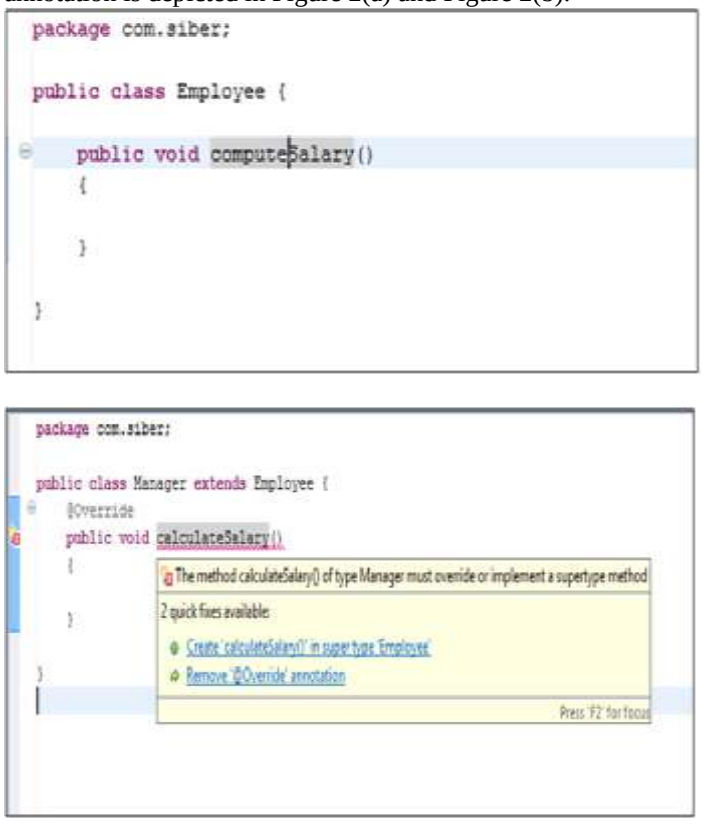


Figure 2(a)-2(b). Visual Notification for Overridden Methods in Eclipse

The calculateSalary() method of super class Employee which has been overridden in Manager class has later been changed to computeSalary() which is flagged as an error in subclass.

@SuppressWarnings

Java compiler generates the warning messages in cases where the method makes calls to deprecated methods or makes an insecure typecast or users non-generic collection, etc. In such cases, the compiler can be informed to suppress warning messages using @SuppressWarning annotation as shown in Figure 3.

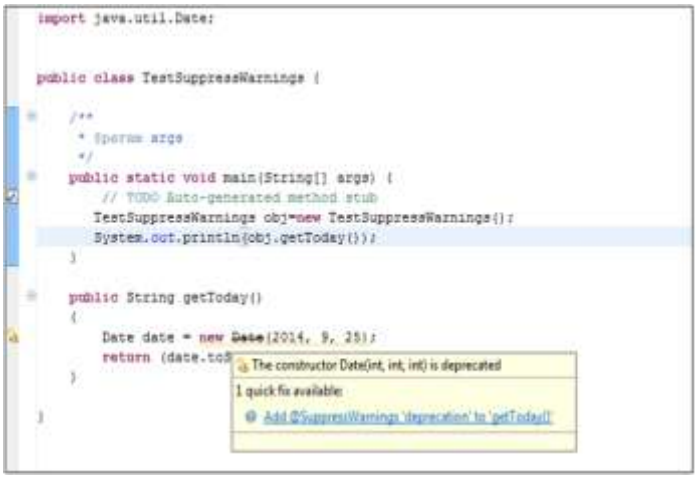


Figure 3. Suppressing Warning Messages using @SuppressWarning annotation

B. Built-in Annotations used by Other Annotations

@Retention Annotation

It is used to indicate the existence level of some other annotation. There are three levels of existence as shown below:

- RetentionPolicy.SOURCE - Exists in source code and is ignored by the compiler.
- RetentionPolicy.CLASS - Existence is identified by the compiler and is ignored by JVM at runtime.
- RetentionPolicy.RUNTIME - Existence is identified by both compiler and JVM.

@Target Annotation

This annotation is used to indicate the scope of the annotation and the entity with which it is associated. The different entities associated with annotation are enumerated below:

- ElementType.TYPE - attached on the declaration of Class, interface, enum, annotation
- ElementType.FIELD - attached on the declaration of field and the enum constant
- ElementType.METHOD - attached on the declaration of method
- ElementType.PARAMETER - attached on the declaration of parameter
- ElementType.CONSTRUCTOR - attached on the declaration of constructor
- ElementType.LOCAL_VARIABLE - attached on the local variable
- ElementType.ANNOTATION_TYPE - attached on the declaration of Annotation
- ElementType.PACKAGE - attached on the declaration of package

C. *J2EE Standard Annotations*

@Resource
J2EE enables resource injection through javax.annotation.Resource annotation which is used to declare a reference to a resource. @Resource can be attached to a class, a field, or a method. The container will inject the resource referred to by @Resource into the component either at runtime, if the annotation is attached to a class or when the component is initialized, if the annotation is attached to field/method.

@inject
CDI API supports @Inject annotation to inject a CDI bean to another bean. CDI provides numerous ways for injecting a bean to an application. A CDI bean can be injected using the Field Dependency Injection, the Constructor Dependency Injection, or the Dependency Injection through the setter method.

@Autowired
Spring provides annotation based auto-wiring by providing @Autowired annotation which can be used to auto wire spring bean on setter method, instance variable, and constructor. If you use @Autowired annotation, spring container auto-wires the bean by matching data type.

D. *Hibernate Annotations*

Hibernate supports a set of annotations which replace XML-based mapping document.

@Entity
Hibernate employs @Entity annotation to mark the class as an entity bean.

@Table
@Table annotation allows to specify the table that will be used to persist the entity in the database.

@Id and @GeneratedValue
These annotations to specify the primary key associated with entity bean.

@Column
The @Column annotation is used to specify the column to which a field or property will be mapped.

E. *Java Reflection to identify Annotation*

Java reflection can identify classes, methods, and fields annotated with annotations which are qualified with the built-in annotation.

@Retention(RetentionPolicy.RUNTIME).

F. *Inheriting Annotations*

By default, annotations are not inherited from super classes to sub classes. The @Inherited annotation marks the annotation to be inherited to subclasses.

G. *Creating Custom Annotation*

Declaring the Annotation
@interface keyword is used for declaring the annotation.

<address> @interface CustomAnnotation{ } </address>

Here, CustomAnnotation is the custom annotation name.
The elements in the annotation must conform to the following properties:

- There is no function body.
- There is no function parameter.
- Method should not have any throws clauses.
- Annotation can have single or multiple methods.
- The return type of an element can be:
- a primitive data type
- a String or
- an Enum
- An element can have default values.
- Annotation can be applied to:
- Type - class, enum or interface declaration
- Field
- Method
- Parameter
- Constructor
- Local Variable or
- Package
-

In the following section, the authors describe creating the custom annotation for execution of Data Manipulation Language command in a database independent manner.

LITERATURE REVIEW

The journey of software development process from few decades back to till date is not abrupt. The development process has witnessed dramatic changes over a period. Java technology started as a single large player in a software development solving platform dependency issues, security issues, eliminating lacunae present in the language, adding web flavour to software development through applets early in 1980s. Custom tags play an important role in web applications. JSP custom tags are written to extract data from database using drop down menu to generate options dynamically [4-5]. Each technology has its own merits and demerits. Different application frameworks have been proposed for scalability and code maintenance. Recently, the authors have provided aggregation of the benefits offered by the three most widely used technologies viz., struts, hibernate and spring in a single J2EE application thereby bringing in best of three worlds to a single application [6]. The application is boosted with powerful struts tag library, persistent layer provided by hibernate and dependency injection or Inversion of Control (IoC) user the same roof. A case study is presented to demonstrate the integration of three diverse technologies, struts, hibernate and spring. JBOSS application server is employed as a container for J2EE components.

Java provides support for general purpose annotations, also referred to as metadata, that allows developers to define their your own annotation types. Annotations have been one of the features introduced early in J2SE 5.0 released in 2004 [7]. Annotations are

a metadata feature that provides the ability to associate additional information to Java elements (like classes and methods). Annotations have no direct effect on the code they annotate, and they do not change the way in which the source code is compiled. Bergmayr et al. [8] have demonstrated the practical value of migration from Java Annotations to UML profiles by providing mapping between Java and UML to generate profiles from annotation-based libraries. Flanagan et. al. [9] introduced the Extended Static Checker for Java (ESC/Java), an experimental compile-time program checker that finds common programming errors by employing simple annotation language. Darwin [10, 11] has provided an elaborated description of AnnaBot, one of the first tools employed for verifying the correct usage of annotation-based metadata in the Java programming language. Rocha et al. [12] have carried out an empirical study of how annotations are used in Java.

In this digital era, the trend of software development is drastically moving from thick clients to thin clients. Responsive sites supported by bootstrap technology are gaining tremendous importance in industry. All these applications operate in multi tiers for achieving scalability and employ some sort of database management systems for persistent data storage. Database interface plays a key role in applications of all types and sizes. Keeping in view the importance of database technology in application development, in the current work the authors have proposed a case study for implementation of a custom Annotation for execution of data manipulation commands in a generic manner.

IMPLEMENTATION OF CUSTOM ANNOTATION

The application is implemented in Eclipse. Figure 4 depicts the application class hierarchy. The main components of the application are:

- Connect.java which declares annotation for storing connection information required towards connection to a backend database management system.
- DocumentClass.java which applies the annotations to different methods.
- AnnotaionProcessor class which uses Java reflection mechanism for retrieving the annotation, reading values and loading appropriate JDBC driver for performing various DML operations.
- run.bat batch file encapsulates Java commands for compiling different source files.
- process.bat batch file is used for setting up environment variable, ClassPath and executing the AnnotationProcessor.

H. Class Hierarchy

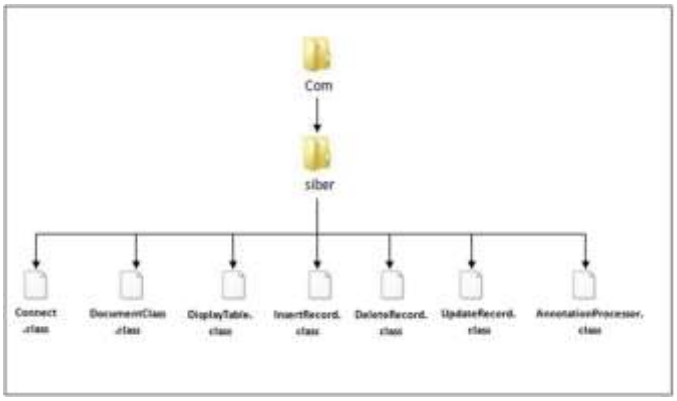


Figure 4. Class Hierarchy of Custom Annotation Implementation

I. Connect.java

```
// Declares annotation for storing the connection
information required for connecting to different
// back end database management systems

packagecom.siber;
importjava.lang.annotation.ElementType;
importjava.lang.annotation.Retention;
importjava.lang.annotation.RetentionPolicy;
importjava.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)

// Using for Class, interface, annotation, enum.
@Target(value = { ElementType.METHOD })
@interface Connect {

publicStringtablename() default"";
publicStringdsnname() default"";
publicString backend() default"access";
publicString username() default"";
publicString password() default"";
publicStringdatabasename() default"";
publicStringhoststring() default"";
}
```

J. DocumentClass.java

```
// This class applies custom annotations for performing
DML Commands

packagecom.siber;

packagecom.siber;
importjava.lang.annotation.ElementType;
importjava.lang.annotation.Retention;
importjava.lang.annotation.RetentionPolicy;
importjava.lang.annotation.Target;


classDocumentClass {
```

 ISSN: 2321-8169
223 – 232

```
//@Connect(backend="oracle", username="system",
//password="siber", hostname="orcl")
//@Connect(backend="mysql", databasename="test",
//username="root", password="mca")
//@Connect(backend="access",dsnname="college")
//@DisplayTable(tablename="student")
publicStringtableGenerator() {
return"Table Generator";
    }

//@Connect(backend="mysql", databasename="test",
//username="root", password="mca")
//@InsertRecord(tablename="student",
//columns={"Rollno","Name"},data={"4","x"})
publicStringinsertRow() {
return"Row Inserter";
    }

//@Connect(backend="mysql", databasename="test",
//username="root", password="mca")
//@DeleteRecord(tablename="student",
//columnname="Rollno", columnvalue="4")
publicStringdeleteRow() {
return"Row Deleter";
    }

//@Connect(backend="mysql", databasename="test",
//username="root", password="mca")
//@UpdateRecord(tablename="student",
columns={"Name"},data={"xyz"},
//columnname="Rollno", columnvalue="1")
publicStringupdateRow() {
return"Row Updater";
    }
}
```

K. AnnotationProcessor.java

```
// This class processes the annotations and executes the
requisite DML Commands on the
// specified table and back end database management system

importjava.lang.annotation.ElementType;
importjava.lang.annotation.Retention;
importjava.lang.annotation.RetentionPolicy;
importjava.lang.annotation.Target;
importjava.lang.reflect.Method;
importjava.sql.*;
import java.io.*;
importcom.siber.*;

publicclassAnnotationProcessor {

publicstaticvoid main(String[] args) {

StringBuildersb = newStringBuilder();
Stringtablename="";
Stringdsnname="";
String backend="";
Stringdatabasename="";
String username="";
String password="";
Stringhoststring="";
String[] columns;
String[] data;

Stringcolumnname="";
Stringcolumnvalue="";

      Connection con=null;
ResultSetrs=null;
ResultSetMetaDatarsmd=null;
      Statement st=null;
PreparedStatementps=null;

      Class<?>clazz = DocumentClass.class;
Method[] methods = clazz.getMethods();

for (Method method : methods) {
// Check if this method is annotated by Annotation Connect
or not.

if (method.isAnnotationPresent(Connect.class)) {
// Get the annotation
        Connect conn =
method.getAnnotation(Connect.class);
// Get the values of elements.
dsnname = conn.dsnname();
backend = conn.backend();
databasename = conn.databasename();
username = conn.username();
password = conn.password();
hoststring = conn.hoststring();
try
        {
if (backend.equals("access"))
          {
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
con=DriverManager.getConnection("jdbc:odbc:"+dsnname
);
          }
elseif(backend.equals("mysql"))
          {
Class.forName("com.mysql.jdbc.Driver");
con=DriverManager.getConnection

("jdbc:mysql://localhost:3306/"+databasename, username,
password);
          }
elseif(backend.equals("oracle"))
          {
Class.forName("oracle.jdbc.driver.OracleDriver");
con=DriverManager.getConnection

("jdbc:oracle:thin:@192.168.30.94:1521:"+hoststring,
username, password);
          }
System.out.println("Connected Successfully....");
        }
catch(Exception e)
        {
System.out.println("Connect:"+e);
        }
```

ISSN: 2321-8169
223 – 232

```
    } // End of Processing Connect Annotation


// Check if this method is annotated by Annotation
InsertRecord or not.
if (method.isAnnotationPresent(InsertRecord.class)) {
try
        {
InsertRecord ins =
method.getAnnotation(InsertRecord.class);
tablename = ins.tablename();
columns=ins.columns();
data=ins.data();
st=con.createStatement();
rs=st.executeQuery("SELECT * FROM " + tablename);
rsmd=rs.getMetaData();
String query="INSERT INTO " + tablename + " VALUES(";
for (int i=0;i<data.length-1;i++)
query+="?,";
query+="?)";
ps=con.prepareStatement(query);
for (int i=0;i<data.length;i++)
          {
if (rsmd.getColumnTypeName(i+1).equals("INTEGER"))
          {
ps.setInt(i+1,Integer.parseInt(data[i]));

          }
elseif(rsmd.getColumnTypeName(i+1).equals("VARCHAR"
) ||
rsmd.getColumnTypeName(i+1).equals("CHAR"))
          {
ps.setString(i+1,data[i]);
          }
           }
ps.executeUpdate();
System.out.println("Record Inserted...");
        }
catch(Exception e)
        {
System.out.println("Insert :"+e);
        }
      } // End of Processing InsertRecord Annotation

// Check if this method is annotated by Annotation
InsertRecord or not.
if (method.isAnnotationPresent(DeleteRecord.class)) {
try
      {
DeleteRecord del =
method.getAnnotation(DeleteRecord.class);
tablename = del.tablename();
columnname=del.columnname();
columnvalue=del.columnvalue();
st=con.createStatement();
rs=st.executeQuery("SELECT * FROM " + tablename);
rsmd=rs.getMetaData();
String query="DELETE FROM " + tablename + " WHERE "
+ columnname + " = ?";
ps=con.prepareStatement(query);
for (int i=0;i<rsmd.getColumnCount();i++)
          {
if(rsmd.getColumnName(i+1).equalsIgnoreCase(columnna
me))
          {
if
(rsmd.getColumnTypeName(i+1).equalsIgnoreCase("INTE
GER"))
          {
ps.setInt(1,Integer.parseInt(columnvalue));
          }
elseif(rsmd.getColumnTypeName(i+1).equals("VARCHA
R") ||
rsmd.getColumnTypeName(i+1).equals("CHAR"))
          {
ps.setString(1,columnvalue);
          }
           }
          }
ps.executeUpdate();
System.out.println("Record Deleted...");
          }
catch(Exception e)
          {
System.out.println("Delete :"+e);
          }
       } // End of Processing DeleteRecord Annotation

// Check if this method is annotated by Annotation
UpdateRecord or not.
if (method.isAnnotationPresent(UpdateRecord.class)) {
try
       {
UpdateRecordupdt =
method.getAnnotation(UpdateRecord.class);
tablename = updt.tablename();
columns=updt.columns();
data=updt.data();
columnname=updt.columnname();
columnvalue=updt.columnvalue();
st=con.createStatement();
rs=st.executeQuery("SELECT * FROM " + tablename);
rsmd=rs.getMetaData();
String query="UPDATE " + tablename + " SET ";
for (int i=0;i<data.length-1;i++)
query+=columns[i] + " = ?,";
query+=columns[data.length-1]+" = ? WHERE ";
query+=columnname;
          query+=" = ? ";
ps=con.prepareStatement(query);
for (int i=0;i<columns.length;i++)
          {
for(int j=1;j<=rsmd.getColumnCount();j++)
           {
if(rsmd.getColumnName(j).equalsIgnoreCase(columns[i]))
            {
if (rsmd.getColumnTypeName(j).equals("INTEGER"))
            {
ps.setInt(i+1,Integer.parseInt(data[i]));
          }
elseif(rsmd.getColumnTypeName(j).equals("VARCHAR")
||
```

ISSN: 2321-8169
223 – 232

```
rsmd.getColumnTypeName(j).equals("CHAR"))
        {
ps.setString(i+1,data[i]);
        }
          }
        }
       }

for (int i=0;i<rsmd.getColumnCount();i++)
       {
if(rsmd.getColumnName(i+1).equalsIgnoreCase(columnnam
e))
        {
if
(rsmd.getColumnTypeName(i+1).equalsIgnoreCase("INTEG
ER"))
        {
ps.setInt(columns.length+1,Integer.parseInt(columnvalue));
       }
elseif(rsmd.getColumnTypeName(i+1).equals("VARCHAR"
) ||
rsmd.getColumnTypeName(i+1).equals("CHAR"))
       {
ps.setString(columns.length+1,columnvalue);
       }
         }
       }
ps.executeUpdate();
System.out.println("Record Updated...");
       }
catch(Exception e)
       {
System.out.println("Update :"+e);
       }
      } // End of Processing UpdateRecord Annotation

// Check if this method is annotated by Annotation
displayTable or not.
if (method.isAnnotationPresent(DisplayTable.class)) {
DisplayTabledisp =
method.getAnnotation(DisplayTable.class);
// Get the values of elements.
tablename = disp.tablename();
try
        {
st=con.createStatement();
rs=st.executeQuery("SELECT * FROM " + tablename);
rsmd=rs.getMetaData();
int count=rsmd.getColumnCount();
sb.append("<html>");
sb.append("\n");
sb.append("<body>");
sb.append("\n");
sb.append("<h1><font color=blue>Contents of "+
tablename+" Table</font></h1>");
sb.append("\n");
sb.append("<table border>");
sb.append("<trbgcolor=yellow><th>");
for (int i=1;i<=count;i++)
          {
sb.append(rsmd.getColumnName(i));
sb.append("<th>");
          }
sb.append("</tr>");
sb.append("\n");
while(rs.next())
          {
sb.append("<tr>");
for (int i=1;i<=count;i++)
            {
sb.append("<td>");
if (rsmd.getColumnTypeName(i).equals("INTEGER")||
rsmd.getColumnTypeName(i).equals("INT"))
            {
sb.append(rs.getInt(i));
       }
elseif(rsmd.getColumnTypeName(i).equals("VARCHAR")
||
rsmd.getColumnTypeName(i).equals("CHAR"))
       {
sb.append(rs.getString(i));
       }
          }
sb.append("</tr>");
sb.append("\n");
          }
sb.append("</tr>");
sb.append("\n");
sb.append("</table>");
sb.append("\n");
sb.append("</body>");
sb.append("\n");
sb.append("</html>");
writeToFile(tablename + ".html", sb);
          }
catch(Exception e)
          {
System.out.println(e);
          }
          } // End of Processing displayTable Annotation
         } // End of for loop
   }

// Write to Console (Or file)
privatestaticvoidwriteToFile(StringfileName,
StringBuildersb) {
try
      {
FileOutputStreamfos=new FileOutputStream(fileName);
fos.write(sb.toString().getBytes());
fos.close();
System.out.println("HTML Generated and Written to the
file "+fileName);
      }
catch(Exception e)
      {
System.out.println(e);
      }
   }
}
```

 ISSN: 2321-8169
223 – 232

L. Structure of Batch Files

run.bat

```
Hide   Copy Code
javac -d . Connect.java
javac -d . InsertRecord.java
javac -d . DeleteRecord.java
javac -d . UpdateRecord.java
javac -d . DisplayTable.java
javac -d . DocumentClass.java
javac -d . AnnotationProcessor.java
```

process.bat

```
set classpath=c:\mysql-connector-java-5.1.15-
bin.jar;ojdbc14.jar;.
javac -d . DocumentClass.java
javac -d . AnnotationProcessor.java
java com.siber.AnnotationProcessor
```

RESULTS AND ANALYSIS

The application architecture proposed in section III is implemented in Java. The DML operations are fired on student table with the structure depicted in Figure 5.

M. Execution of Application

The format of custom annotation for connecting to the MS-Access backend and displaying the contents of student table is shown below:

```
@Connect(backend="access",dsnname="college")
@DisplayTable(tablename="student")
```

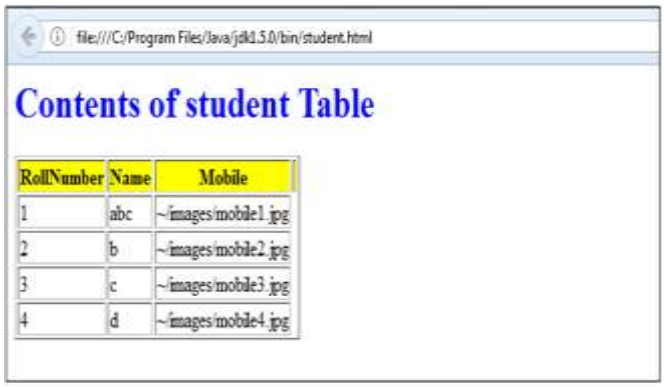


Figure 5. Content of MS-Access Student Table

The format of custom annotation for connecting to the MySQL backend and displaying the contents of student table is shown below:

```
@Connect(backend="mysql", databasename="test",
username="root", password="mca")

@DisplayTable(tablename="student")
```

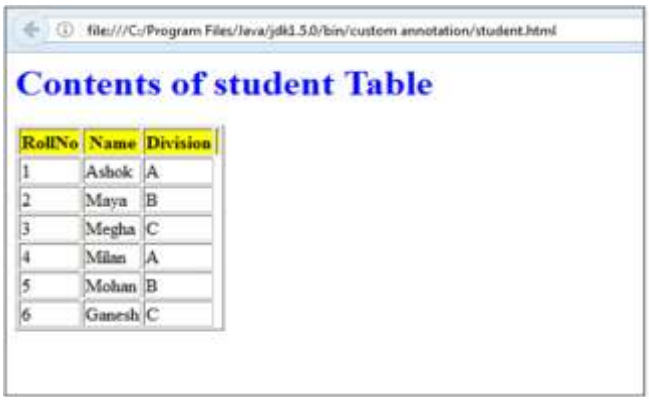


Figure 6. Content of MySQL Student Table

The format of custom annotation for connecting to the Oracle 10g backend database management system and displaying the contents of student table is shown below:

```
@Connect(backend="oracle", username="system",
password="siber", hostname="orcl")

@DisplayTable(tablename="student")
```

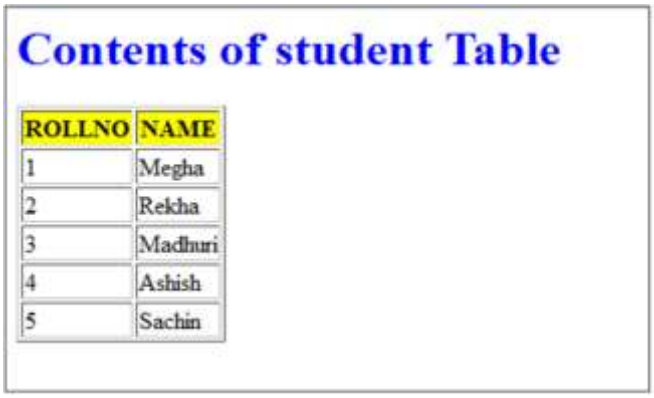


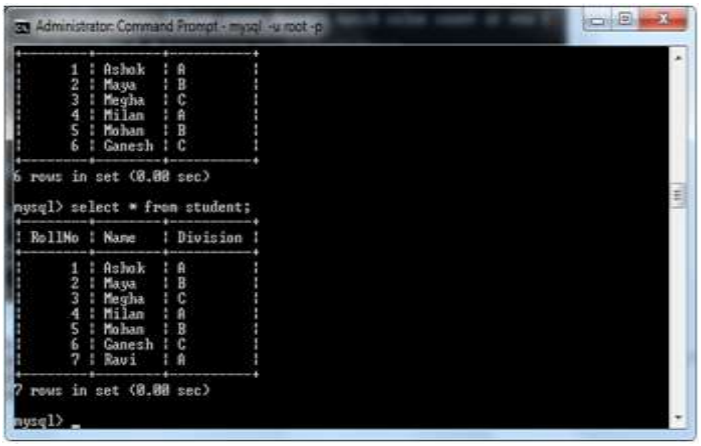
Figure 7. Content of Oracle 10g Student Table

N. Inserting a Record in a MySQL Student Table

The format of custom annotation for connecting to MySQL database and inserting a row in student is shown below:

```
@Connect(backend="mysql", databasename="test",
username="root", password="mca")

@InsertRecord(tablename="student",
columns={"Rollno","Name"},data={"4","x"})
```

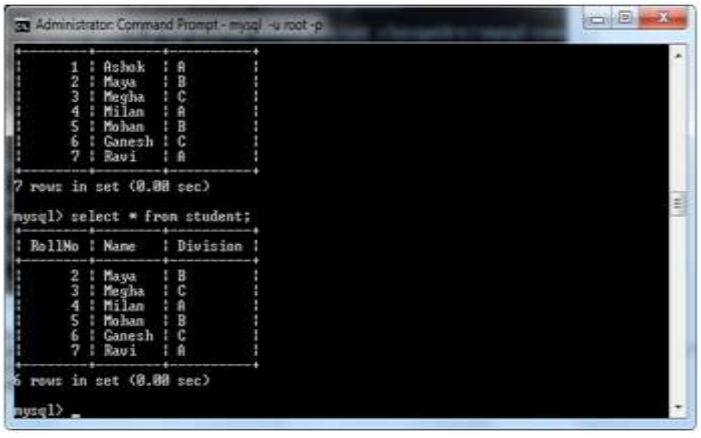


O. *Deleting a Record From the MySQL Student Table*

The format of custom annotation for connecting to MySQL database and deleting a row from a student is shown below:

```
@Connect(backend="mysql", databasename="test",
        username="root", password="mca")

@DeleteRecord(tablename="student",
        columnname="Rollno", columnvalue="4")
```

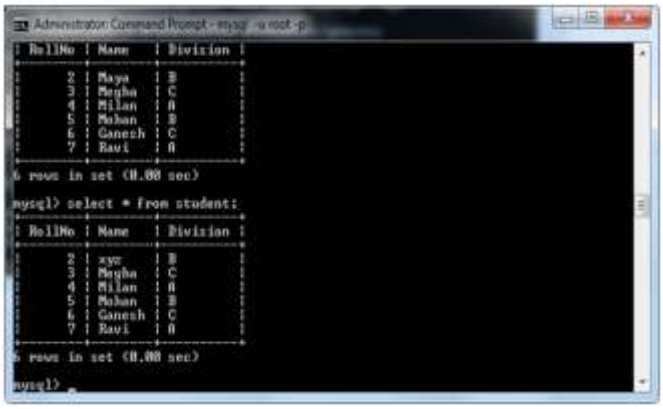


P. *Updating a Record in a MySQL Student Table*

The format of custom annotation for connecting to MySQL database and updating a row in a student is shown below:

```
@Connect(backend="mysql", databasename="test",
username="root", password="mca")

@UpdateRecord(tablename="student",
columns={"Name"},data={"xyz"},
      columnname="Rollno", columnvalue="1")
```



CONCLUSION AND SCOPE FOR FUTURE WORK

Java Annotation is an extremely useful concept in Java for emitting metadata for Java Compiler, JVM, etc. The current work introduces the concept of Annotations in Java with an emphasis on various in-built annotations in Java and the annotations that are used by other annotations. The reader is introduced to J2EE standard annotations and those employed by Hibernate as a replacement for XML-based mapping document. Steps in designing and using custom annotations are highlighted. A custom annotation design is illustrated with the help of an example for execution of DML commands in a generic way in a database management system independent manner.

REFERENCES

[1] https://docs.oracle.com/javase/tutorial/java/annotations/
[2] http://docs.oracle.com/javaee/6/tutorial/doc/bncjk.html
[3] .http://beginnersbook.com/2014/09/java-annotations/
[4] Dr. Poornima G. Naik, Mr Girish R. Naik, JSP Custom Tag for Displaying Master-Detail Relationship in a Hierarchical Grid Control – A Case Study, International Journal of Engineering Applied Sciences and Technology, 2016, Vol. 1, Issue 8, pp. 65-71
[5] Dr. Poornima G. Naik, Mr Girish R. Naik,JSP Custom Tag for Pagination, Sorting and Filtering – A Case Study, International Journal on Recent and Innovation Trends in Computing and Communication, Vol. 4, Issue 6, pp. 404-411
[6] Dr. Poornima G. Naik, Mr Girish R. Naik, Struts, Hibernate and Spring Integration – A Case Study, International Journal on Recent and Innovation Trends in Computing and Communication, Vol. 5 Issue 3, pp. 261 – 268
[7] Gosling, B. Joy, G. Steele, and G. Brach, Java Language Specification, 3rd Edition Addison-Wesley, 2005
[8] Bergmayr A., Grossniklaus M., Wimmer M., Kappel G. (2014) JUMP—From Java Annotations to UML Profiles. In: Dingel J., Schulte W., Ramos I., Abrahão

S., Insfran E. (eds) Model-Driven Engineering Languages and Systems. MODELS 2014. Lecture Notes in Computer Science, vol 8767. Springer, Cham

[9] Cormac Flanagan, K. Rustan, M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, Raymie Stata, Extended Static Checking for Java, Compaq Systems Research Center, 130 Lytton Ave. Palo Alto, CA 94301, USA

[10] Darwin, Annabot: a static verifier for Java annotation usage. In 2nd International Workshop on Defects in Large Software Systems, pp. 21–28, 2009.

[11] AnnaBot: A Static Verifier for Java Annotation Usage, Ian Darwin, Advances in Software Engineering, Vol., 2010, pp. 1-7, 2010.

[12] Henrique Rocha, Marco Tulio Valente, How Annotations are Used in Java : An Empirical Study, Department of Computer Science, Federal University of Minas Gerais (UFMG).