DATABASE TECHNOLOGY
WEEK 9 ASSIGNMENT

1.With suitable diagram explain in detail about steps involved in executing SQLJ Translator component.

1. The JVM invokes the SQLJ translator.

2. The translator parses the SQLJ and Java code in the .sqlj file, checking for proper SQLJ syntax
and looking for type mismatches between the declared SQL data types and corresponding Java host variables. Host variables are Java local variables that are used as input or output parameters in SQL operations.

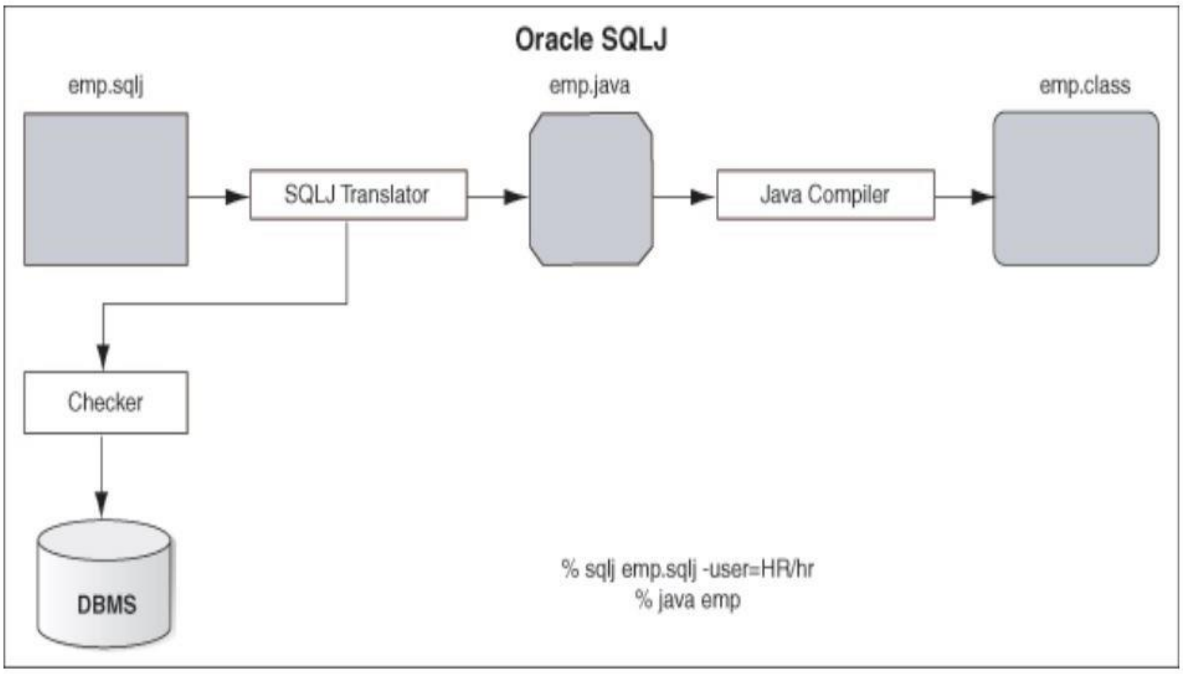
3. Depending on the SQLJ option settings, the translator invokes the online semantics-checker,
the offline parser, neither, or both. This is to verify syntax of embedded SQL and PL/SQL statements and to check the use of database elements in the code against an appropriate database schema, for online checking. Even when neither is specified, some basic level of checking is performed.

4. When online checking is specified, SQLJ will connect to a specified database schema to verify
that the database supports all the database tables, stored procedures, and SQL syntax that the application uses. It also verifies that the host variable types in the SQLJ application are compatible with data types of corresponding database columns.
For Oracle-specific SQLJ code generation (-codegen=oracle, which is default), SQL operations are converted directly into Oracle JDBC calls.

5. The JVM invokes the Java compiler, which is usually, but not necessarily, the standard javac provided with the Sun Microsystems JDK.

6. The compiler compiles the Java source file generated in Step 4 and produces Java .class files
as appropriate. This will include a .class file for each class that is defined, each of the SQLJ declarations.

7.
Generated Java code is put into a .java output file containing the following:
• Any class definitions and Java code from the. sqlj source file
• Class definitions created as a result of the SQLJ iterator and connection context declarations
• Calls to Oracle JDBC drivers to implement the actions of the embedded SQL operations



A simple program in SQLJ is presented here, this program illustrates the essential steps that are needed to write an SQLJ program.

Step: 1 Import necessary classes. In addition to the JDBC classes, java.sql.*, every SQLJ program will need to include the SQLJ run-time classes sqlj.runtime.* and sqlj.runtime.ref.*. In addition, to establish the default connection to Oracle, the Oracle class from the oracle.sqlj.runtime.* package is required. So, a typical set of statements to import packages would be:

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
 import java.io.*;
 import oracle.sqlj.runtime.*;
```

Step 2. Register the JDBC driver, if needed. If a non-Oracle JDBC driver is being used, a call to the registerDriver method of the DriverManager class is necessary. For the purposes of this chapter, an Oracle JDBC driver is assumed. Therefore, this statement is not shown in any of the examples.

3. Connect to the database. Connecting to the Oracle database is done by first obtaining a DefaultContext object using the getConnection method (of the Oracle class1), whose specification is

```
public static DefaultContext getConnection (String URL, String user, String password, Boolean autoCommit)
            throws SQLException
```

URL is the database URL, and user and password are the Oracle user ID and password, respectively.

Setting autoCommit to true would create the connection in auto commit mode, and setting it to false would create a connection in which the transactions must be committed by the programmer.

A sample invocation is shown here:

```
DefaultContext cx1 = Oracle.getConnection("jdbc:oracle:oci8:@", "book","book",true);
```

The DefaultContext object so obtained is then used to set the static default context, as follows: DefaultContext.setDefaultContext(cx1); This DefaultContext object now provides the default connection to the database.

Step 4. Embed SQL statements in the Java program. Once the default connection has been established, SQL statements can be embedded within the Java program using the following syntax:

```
#sql {sql statement}
```

where #sql indicates to the SQLJ translator, called sqlj, that what follows is an SQL statement and is any valid SQL statement, which may include host variables and host expressions. Host variables are prefixed with a colon, much like in Pro*C/Pro*C++

To execute SQLJ program
Once you have created your SQLJ files, you are going to want to compile and execute them. The following syntax demonstrates how to do this:
> sqlj << filename.sqlj >>
Compile *.sqlj file with proper libraries in your CLASSPATH results in the *.java and *.class file generated.

Program :

```
import java.sql.*;
import oracle.jdbc.*;

public class SimpleDemoJDBC                                  // line 7
{

//TO DO: make a main that calls this

  public Address getEmployeeAddress(int empno, Connection conn)
    throws SQLException                                  // line 13
  {
    Address addr;
    PreparedStatement pstmt =                            // line 16
      conn.prepareStatement("SELECT office_addr FROM employees" +
      " WHERE empnumber = ?");
    pstmt.setInt(1, empno);
    OracleResultSet rs = (OracleResultSet)pstmt.executeQuery();
    rs.next();                                       // line 21
     //TO DO: what if false (result set contains no data)?
    addr = (Address)rs.getORAData(1, Address.getORADataFactory());
    //TO DO: what if additional rows?
    rs.close();                                      // line 25
    pstmt.close();
    return addr;                                      // line 27
  }
  public Address updateAddress(Address addr, Connection conn)
    throws SQLException                                  // line 30

  {
    OracleCallableStatement cstmt = (OracleCallableStatement)
      conn.prepareCall("{ ? = call UPDATE_ADDRESS(?) }");  //line 34
    cstmt.registerOutParameter(1, Address._SQL_TYPECODE, Address._SQL_NAME);
                                                // line 36
    if (addr == null) {
      cstmt.setNull(2, Address._SQL_TYPECODE, Address._SQL_NAME);
    } else {
      cstmt.setORAData(2, addr);
    }
    cstmt.executeUpdate();                    // line 43
    addr = (Address)cstmt.getORAData(1, Address.getORADataFactory());
    cstmt.close();                            // line 45
    return addr;
  }

}
```